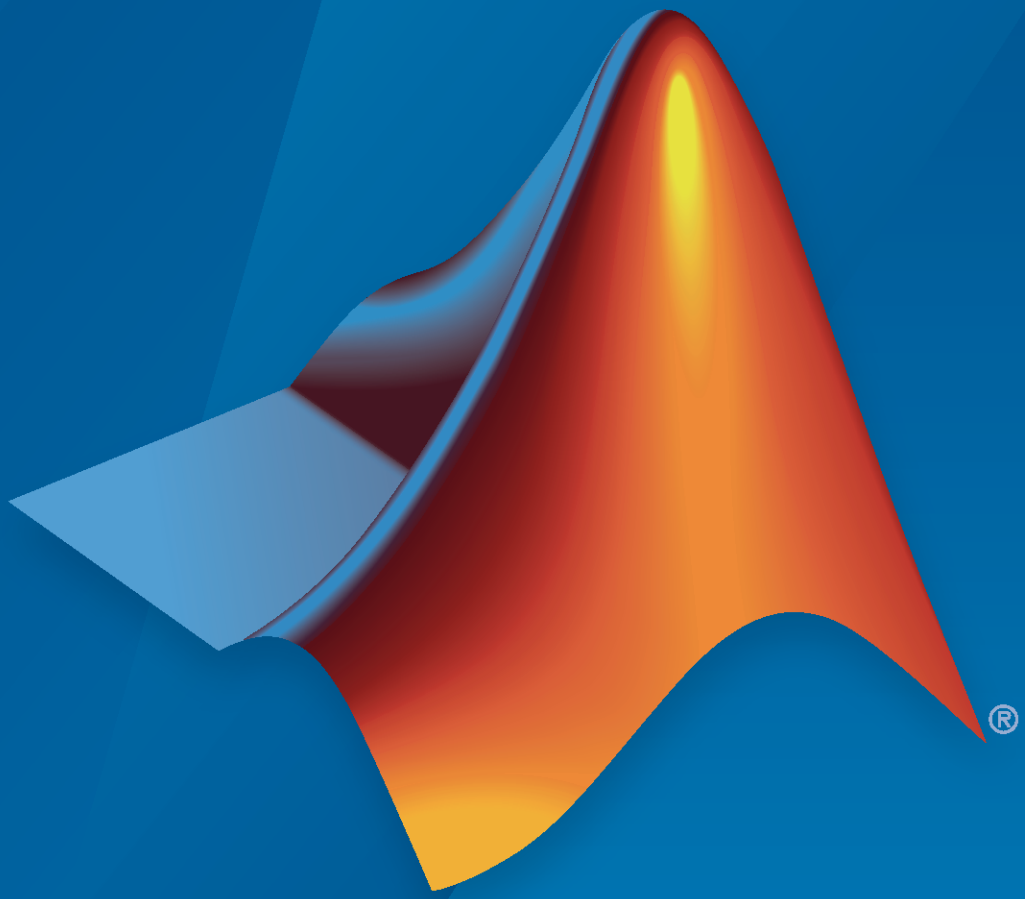
Aerospace Toolbox

User's Guide



MATLAB®

R2023a

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

© COPYRIGHT 2006–2023 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

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)
March 2020	Online only	Revised for Version 3.3 (Release 2020a)
September 2020	Online only	Revised for Version 3.4 (Release 2020b)
March 2021	Online only	Revised for Version 4.0 (Release 2021a)
September 2021	Online only	Revised for Version 4.1 (Release 2021b)
March 2022	Online only	Revised for Version 4.2 (Release 2022a)
September 2022	Online only	Revised for Version 4.3 (Release 2022b)
March 2023	Online only	Revised for Version 4.4 (Release 2023a)

Contents

1 Getting Started

Aerospace Toolbox Product Description 1-2

2 Using Aerospace Toolbox

Fundamental Coordinate System Concepts 2-2
- Definitions 2-2
- Approximations 2-2
- Passive Transformations 2-3
- Motion with Respect to Other Planets 2-3

Coordinate Systems for Modeling 2-4
- Body Coordinates 2-4
- Wind Coordinates 2-5

Coordinate Systems for Navigation 2-6
- Geocentric and Geodetic Latitudes 2-6
- NED Coordinates 2-6
- ECI Coordinates 2-7
- ECEF Coordinates 2-8

Coordinate Systems for Display 2-9

Aerospace Units 2-10

Digital DATCOM Data 2-11
- Digital DATCOM Data Overview 2-11
- USAF Digital DATCOM File 2-11
- Data from DATCOM Files 2-11
- Imported DATCOM Data 2-12
- Missing DATCOM Data 2-13
- Aerodynamic Coefficients 2-15

Visualization Tools 2-18
- Axes Transformation Functions 2-18
- Flight Simulation Interface Functions 2-18
- Flight Instrument Functions 2-19
- MATLAB Graphics-Based Animation 2-20

Aerospace Toolbox Animation Objects 2-22

Aero.Animation Objects **2-23**

Simulated and Actual Flight Data Using Aero.Animation Objects **2-24**
Creating and Configuring an Animation Object **2-24**
Loading Recorded Data for Flight Trajectories **2-24**
Displaying Body Geometries in a Figure Window **2-25**
Recording Animation Files **2-25**
Playing Back Flight Trajectories Using the Animation Object **2-25**
Viewing Recorded Animation Files **2-26**
Manipulating the Camera **2-26**
Moving and Repositioning Bodies **2-27**
Creating a Transparency in the First Body **2-28**
Changing the Color of the Second Body **2-29**
Turning Off the Landing Gear of the Second Body **2-30**

Aero.VirtualRealityAnimation Objects **2-31**

Visualize Aircraft Takeoff via Virtual Reality Animation Object **2-32**

Aero.FlightGearAnimation Objects **2-42**
About the FlightGear Interface **2-42**
Configuring Your Computer for FlightGear **2-42**
Install and Start FlightGear **2-43**
Installing Additional FlightGear Scenery **2-44**
Flight Simulator Interface Example **2-44**

Flight Trajectory Data **2-47**
Loading Recorded Flight Trajectory Data **2-47**
Creating a Time Series Object from Trajectory Data **2-47**
Creating a FlightGearAnimation Object **2-47**
Modifying the FlightGearAnimation Object Properties **2-48**
Generating the Run Script **2-48**
Starting the FlightGear Flight Simulator **2-49**
Playing Back the Flight Trajectory **2-49**

Create and Configure Flight Instrument Component and an Animation Object **2-51**
Load and Visualize Data **2-51**
Create Flight Instrument Components **2-52**

Flight Instrument Components in App Designer **2-54**
Start App Designer and Create a New App **2-54**
Drag Aerospace Components into the App **2-54**
Add Code to Load and Visualize Data for the App **2-55**
Add Code to Trigger a Display of the Animation Object **2-56**
Add Code to Close the Animation Window with UIfigure Window **2-57**
Save and Run the App **2-58**

Work with Fixed-Wing Aircraft Using Functions **2-60**
Suggested Workflow **2-60**
Static Stability Analysis **2-61**
Linear Analysis **2-61**

Analyze Fixed-Wing Aircraft with Objects **2-62**
Suggested Workflow **2-62**

Static Stability Analysis 2-63
Linear Analysis 2-63
Examples 2-63

Satellite Scenario Key Concepts 2-65
Coordinate Systems 2-65
Orbital Elements 2-70
Two Line Element (TLE) Files 2-72

Satellite Scenario Overview 2-74

Model, Visualize, and Analyze Satellite Scenario 2-77

Flight Control Analysis Tools 2-80
Plot Short-Period Undamped Natural Frequency Results 2-80

3 Add-On for Ephemeris and Geoid Data Support

Add Ephemeris and Geoid Data for Aerospace Products 3-2

4 Functions

5 Aerospace Toolbox Examples

Import from USAF Digital DATCOM Files 5-2

Create a Flight Animation from Trajectory Data 5-17

Estimate G Forces for Flight Data 5-21

Calculating Best Glide Quantities 5-26

Overlaying Simulated and Actual Flight Data 5-31

Comparing Zonal Harmonic Gravity Model to Other Gravity Models . . . 5-39

Calculating Compressor Power Required in a Supersonic Wind Tunnel 5-46

Analyzing Flow with Friction Through an Insulated Constant Area Duct 5-52

Determine Heat Transfer and Mass Flow Rate in a Ramjet Combustion Chamber **5-57**

Solving for the Exit Flow of a Supersonic Nozzle **5-63**

Visualize World Magnetic Model Contours for 2020 Epoch **5-73**

Visualizing Geoid Height for Earth Geopotential Model 1996 . . . . . **5-82**

Marine Navigation Using Planetary Ephemerides **5-88**

Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation **5-96**

Display Flight Trajectory Data Using Flight Instruments and Flight Animation **5-99**

Aerospace Flight Instruments in App Designer **5-104**

Analyze State-Space Model for Linear Control and Static Stability Analysis **5-106**

Customize Fixed-Wing Aircraft with Additional Aircraft States **5-113**

Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft **5-121**

Modeling Satellite Constellations Using Ephemeris Data **5-130**

Satellite Constellation Access to Ground Station **5-140**

Comparison of Orbit Propagators **5-151**

Detect and Track LEO Satellite Constellation with Ground Radars . . . **5-159**

Get Started with Fixed-Wing Aircraft **5-170**

Customize Fixed-Wing Aircraft with the Object Interface **5-186**

Multi-Hop Path Selection Through Large Satellite Constellation **5-196**

Modeling Custom Satellite Attitude and Gimbal Steering **5-204**

Aircraft Design Optimization with the Fixed-Wing Object **5-215**

Compute Indicated Airspeed for Pitot-Static Airspeed Indicator **5-221**

Differences Between Corrected Airspeeds **5-224**

Analyze Access Between a Satellite Constellation and an Aircraft **5-228**

Satellite Conjunction Finder **5-237**

AC3D Files and Thumbnails

A

AC3D Files and Thumbnails Overview **A-2**

Getting Started

Aerospace Toolbox Product Description

Analyze and visualize aerospace vehicle motion using reference standards and models

Aerospace Toolbox provides standards-based tools and functions for analyzing the motion, mission, and environment of aerospace vehicles. It includes aerospace math operations, coordinate system and spatial transformations, and validated environment models for interpreting flight data. The toolbox also includes 2D and 3D visualization tools and standard cockpit instruments for observing vehicle motion.

For flight vehicles, you can import Data Compendium (Datcom) files directly into MATLAB® to represent vehicle aerodynamics. The aerodynamics can be combined with reference parameters to define your aircraft configuration and dynamics for control design and flying qualities analysis.

Aerospace Toolbox lets you design and analyze scenarios consisting of satellites and ground stations. You can propagate satellite trajectories from orbital elements or two-line element sets, load in satellite and constellation ephemerides, perform mission analysis tasks such as line-of-sight access, and visualize the scenario as a ground track or globe.

Using Aerospace Toolbox

- “Fundamental Coordinate System Concepts” on page 2-2
- “Coordinate Systems for Modeling” on page 2-4
- “Coordinate Systems for Navigation” on page 2-6
- “Coordinate Systems for Display” on page 2-9
- “Aerospace Units” on page 2-10
- “Digital DATCOM Data” on page 2-11
- “Visualization Tools” on page 2-18
- “Aerospace Toolbox Animation Objects” on page 2-22
- “Aero.Animation Objects” on page 2-23
- “Simulated and Actual Flight Data Using Aero.Animation Objects” on page 2-24
- “Aero.VirtualRealityAnimation Objects” on page 2-31
- “Visualize Aircraft Takeoff via Virtual Reality Animation Object” on page 2-32
- “Aero.FlightGearAnimation Objects” on page 2-42
- “Flight Trajectory Data” on page 2-47
- “Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
- “Flight Instrument Components in App Designer” on page 2-54
- “Work with Fixed-Wing Aircraft Using Functions” on page 2-60
- “Analyze Fixed-Wing Aircraft with Objects” on page 2-62
- “Satellite Scenario Key Concepts” on page 2-65
- “Satellite Scenario Overview” on page 2-74
- “Model, Visualize, and Analyze Satellite Scenario” on page 2-77
- “Flight Control Analysis Tools” on page 2-80

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. For more information on geographic information, see mat.

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 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.

Passive Transformations

All quaternions in Aerospace Toolbox are passive transformations. In a passive transformation, the vector is unchanged and the coordinate system in which it is defined is rotated. For more information on transformations, see Active and passive transformations.

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

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

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

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

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

See Also

Related Examples

- “Coordinate Systems for Modeling” on page 2-4
- “Coordinate Systems for Navigation” on page 2-6
- “Coordinate Systems for Display” on page 2-9

External Websites

- Office of Geomatics

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.

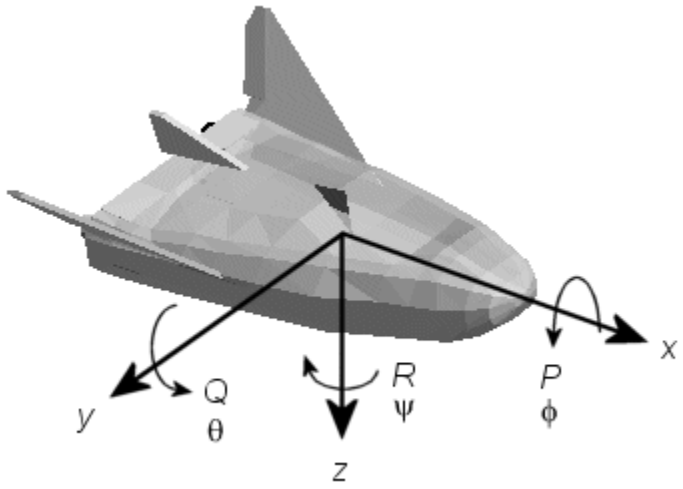
Translational Degrees of Freedom

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.

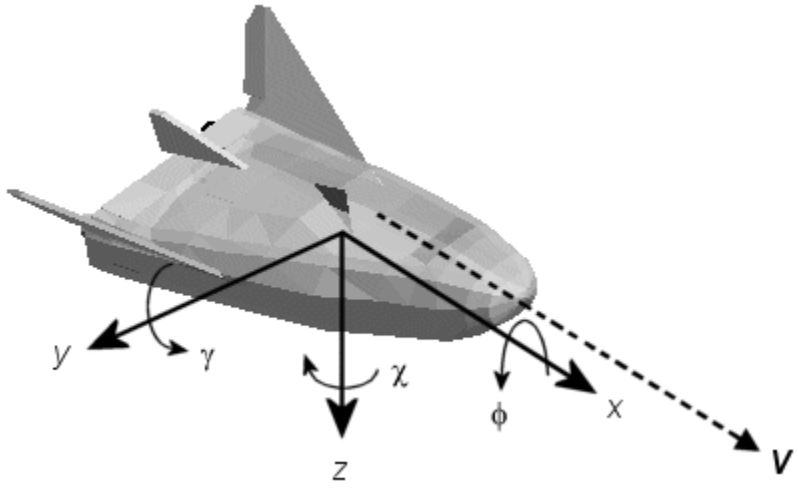
Translational Degrees of Freedom

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.

See Also

Related Examples

- “Fundamental Coordinate System Concepts” on page 2-2
- “Coordinate Systems for Navigation” on page 2-6
- “Coordinate Systems for Display” on page 2-9

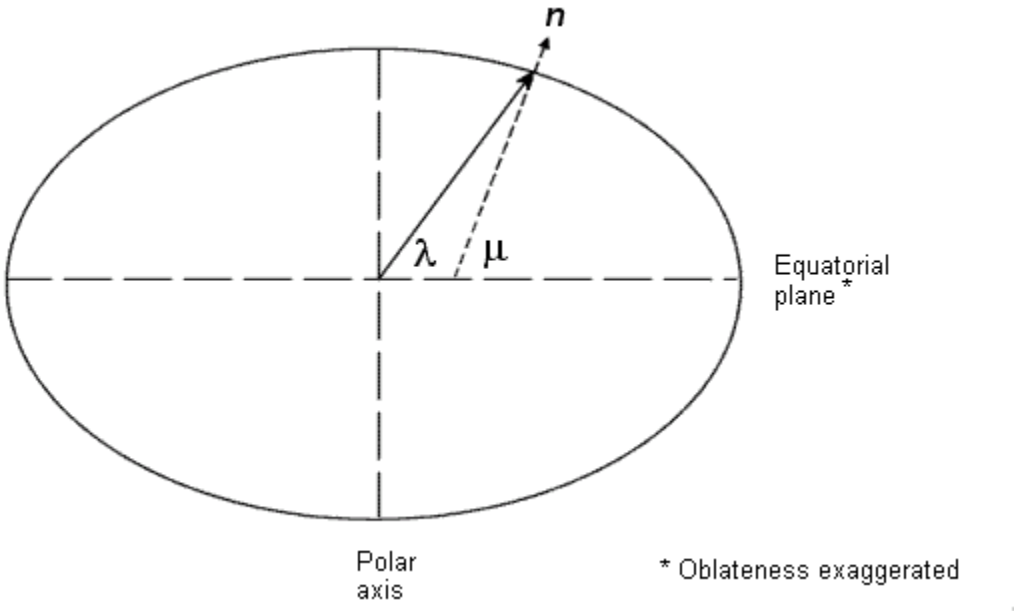
Coordinate Systems for Navigation

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.

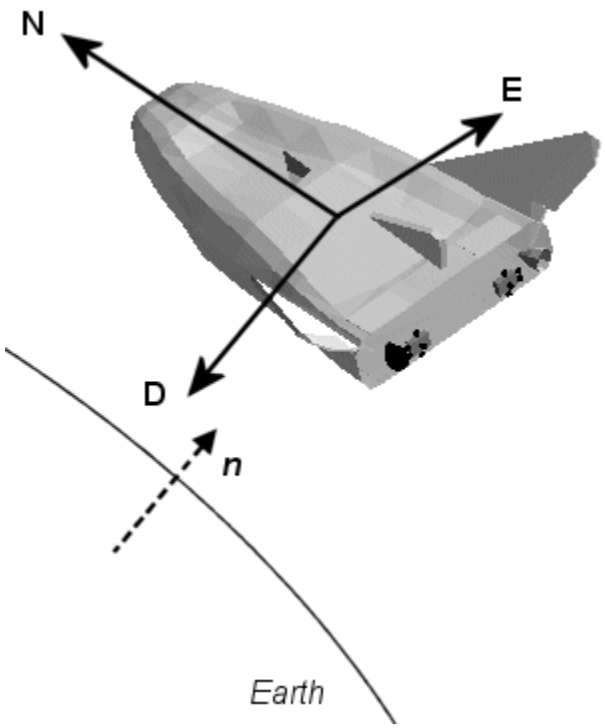


NED Coordinates

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*.

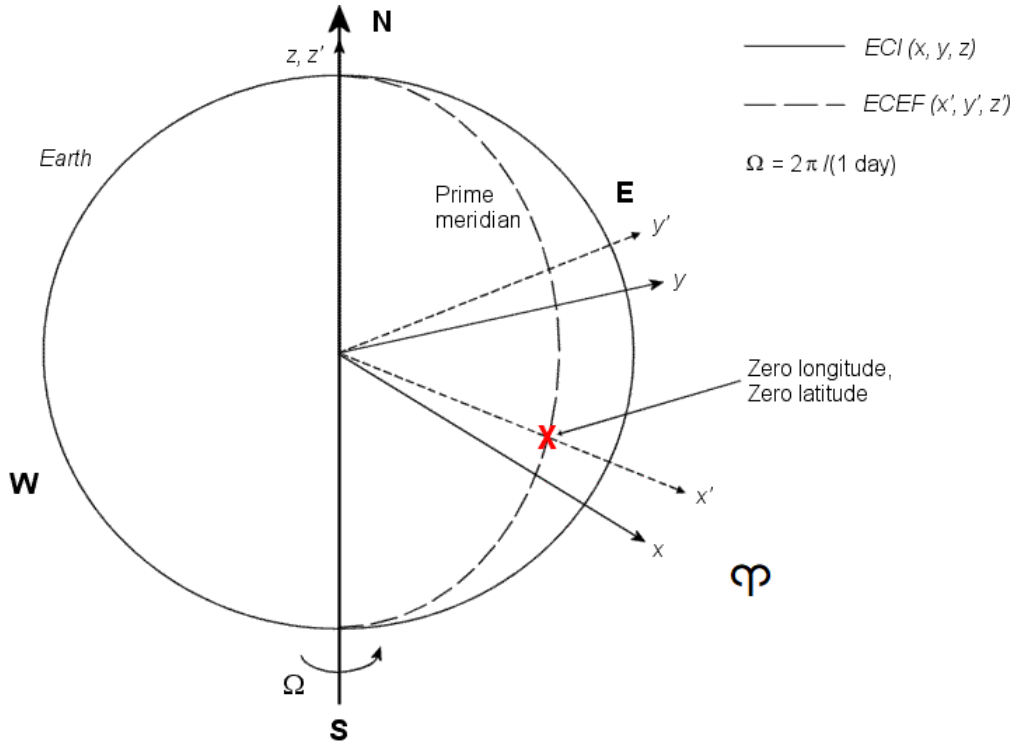
  Flying at a constant altitude means flying at a constant *z* above the Earth's surface.



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.

See Also

Related Examples

- “Fundamental Coordinate System Concepts” on page 2-2
- “Coordinate Systems for Modeling” on page 2-4
- “Coordinate Systems for Display” on page 2-9

Coordinate Systems for Display

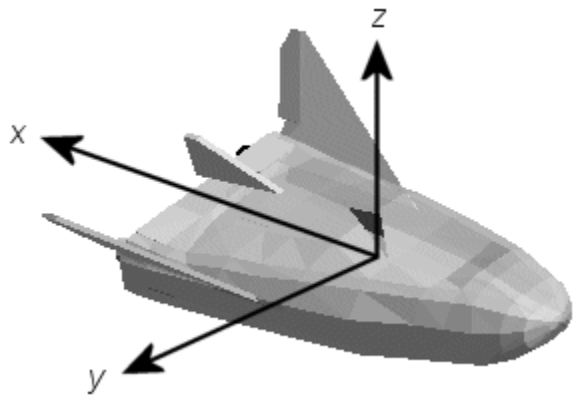
The Aerospace Toolbox software lets you use FlightGear coordinates for rendering motion.

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

- “Flight Simulator Interface Example” on page 2-44 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.



See Also

Related Examples

- “Fundamental Coordinate System Concepts” on page 2-2
- “Coordinate Systems for Modeling” on page 2-4
- “Coordinate Systems for Navigation” on page 2-6

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)
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” on page 2-11
“USAF Digital DATCOM File” on page 2-11
“Data from DATCOM Files” on page 2-11
“Imported DATCOM Data” on page 2-12
“Missing DATCOM Data” on page 2-13
“Aerodynamic Coefficients” on page 2-15

Digital DATCOM Data Overview

To import United States Air Force (USAF) Digital DATCOM files into the MATLAB environment, use the `datcomimport` function. For more information, see the `datcomimport` function reference page. This topic explains how to import data from a USAF Digital DATCOM file using the “Import from USAF Digital DATCOM Files” on page 5-2 example.

USAF Digital DATCOM File

Aerospace Toolbox provides `astdatcom.in`, 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. It calculates static and dynamic derivatives.

```
 $FLTCON NMACH=2.0,MACH(1)=0.1,0.2$
 $FLTCON NALT=2.0,ALT(1)=5000.0,8000.0$
 $FLTCON NALPHA=5.,ALSCHD(1)=-2.0,0.0,2.0,
  ALSCHD(4)=4.0,8.0,LOOP=2.0$
 $OPTINS SREF=225.8,CBARR=5.75,BLREF=41.15$
 $SYNTHS XCG=7.08,ZCG=0.0,XW=6.1,ZW=-1.4,ALIW=1.1,XH=20.2,
    ZH=0.4,ALIH=0.0,XV=21.3,ZV=0.0,VERTUP=.TRUE.$
 $BODY NX=10.0,
    X(1)=-4.9,0.0,3.0,6.1,9.1,13.3,20.2,23.5,25.9,
    R(1)=0.0,1.0,1.75,2.6,2.6,2.6,2.0,1.0,0.0$
 $WGPLNF CHRDTP=4.0,SSPNE=18.7,SSPN=20.6,CHRDR=7.2,SAVSI=0.0,CHSTAT=0.25,
    TWISTA=-1.1,SSPNDD=0.0,DHDADI=3.0,DHDADO=3.0,TYPE=1.0$
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
```

To view 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, `type astdatcom.out` in the MATLAB Command Window.

Data from DATCOM Files

To import Digital DATCOM data into the MATLAB environment, use the `datcomimport` function.

```
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, the function sets missing data points to 99999. It sets data points to NaN when no DATCOM methods exist or when 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 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.nmach
        for h = 1:data.nalt
            data.(aerotab{k})(:,m,h) = data.(aerotab{k})(1,m,h);
        end
    end
end
```

The updated imported data values are:

```
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
```

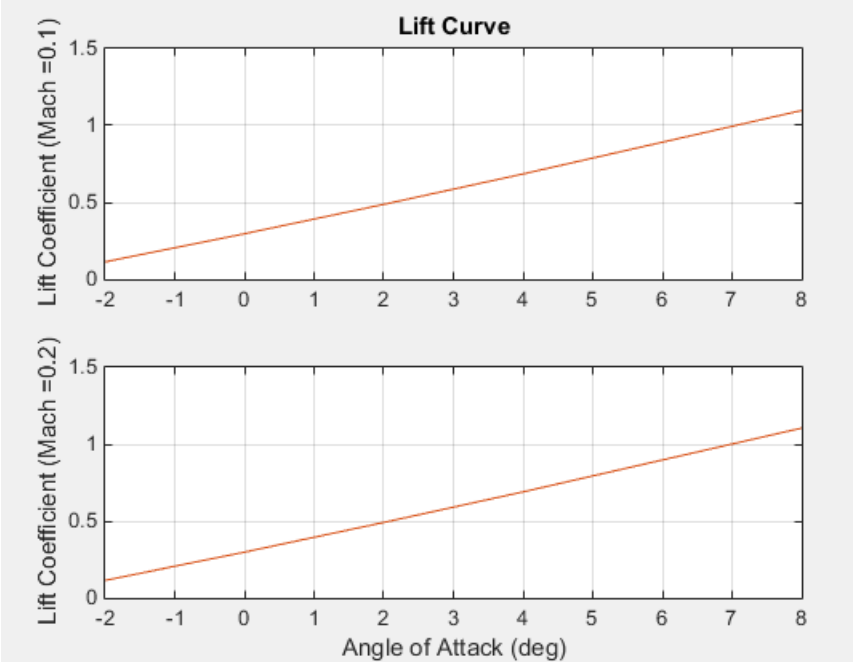
```
    0.0974    0.0984
    0.0974    0.0984


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:

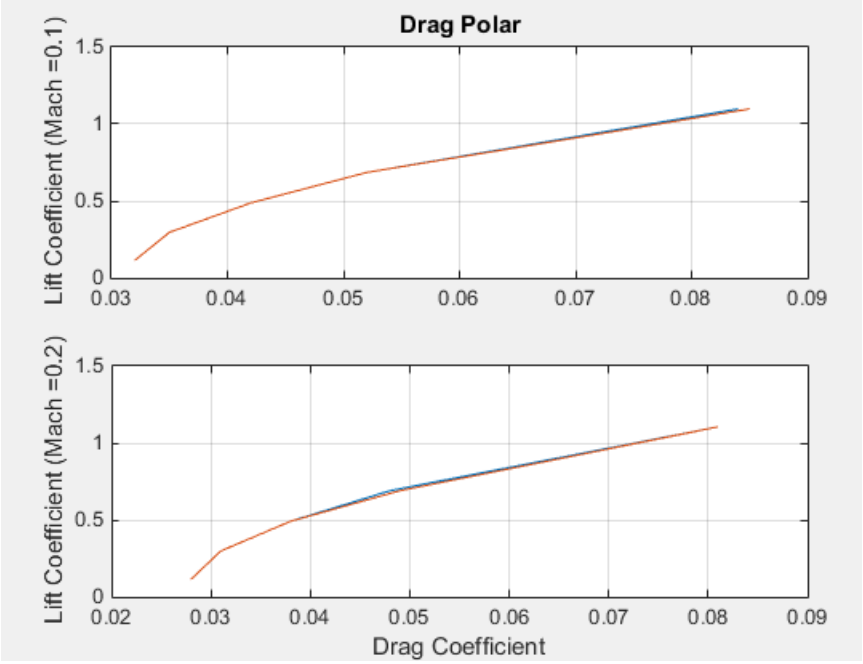
Plotting Lift Curve Moments

```
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)')
```



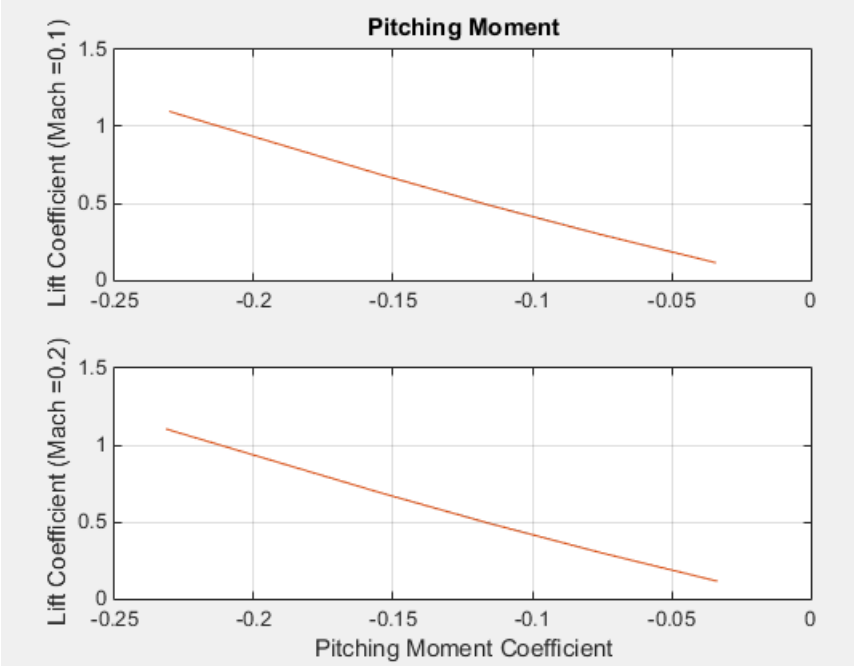
Plotting Drag Polar Moments

```
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')
```



See Also

`datcomimport`

Related Examples

- “Import from USAF Digital DATCOM Files” on page 5-2

Visualization Tools

You can visualize stages of your Aerospace Toolbox application using flight simulator, flight instrument, and MATLAB graphics-based functions and classes. You can also transform coordinate axes to different types.

Axes Transformation Functions

Use the axes transform functions to convert your data to the desired coordinate system. Axes transformation functions support these transformation:

- Rotation angles and quaternions, direction cosine matrices, or Euler-Rodriguez vectors
- Direction cosine matrices and quaternions, geodetic latitude and longitude, stability frames, Euler-Rodriguez vectors, attack and sideslip angles, Earth-centered inertial (ECI) to Earth-centered Earth-fixed (ECEF)
- Earth-centered inertial (ECI) coordinates and azimuth or geodetic latitude, longitude, altitude (LLA) coordinates
- Geocentric latitude and geodetic latitude
- Flat Earth positions from geodetic latitude, longitude, and altitude

Flight Simulation Interface Functions

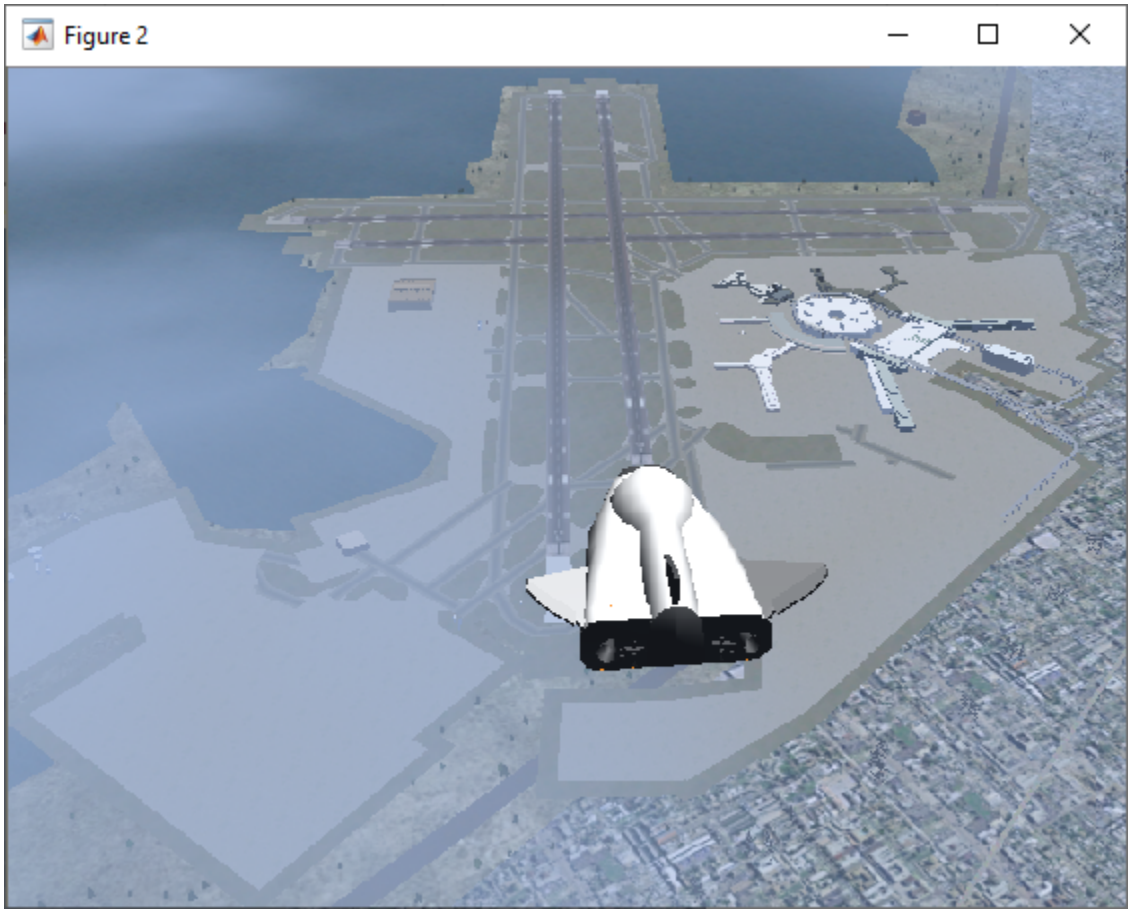
Use flight simulator interface objects to visualize flight paths using FlightGear and input FlightGear data to models.

To use FlightGear to visualize flight paths, first install FlightGear software. For more information, see `Aero.FlightGearAnimation`.

Use `Aero.FlightGearAnimation` to create a FlightGear animation object. Use the object methods to work with the FlightGear object, such as:

- Clearing and deleting timer for animation of FlightGear flight simulator.
- Generating run scripts for FlightGear flight simulator
- Animating FlightGear flight simulator using given position/angle time series.
- Updating position data to FlightGear animation object.

Here is an example of a flight trajectory from “Create a Flight Animation from Trajectory Data” on page 5-17.



Flight Instrument Functions

Use functions representing standard cockpit instruments to display flight status information from the application. Each function creates a cockpit instrument gauge that works with figures created with `uifigure`. After creating a flight instrument panel to contain these gauges, you can load them into MATLAB App Designer. To control the appearance and behavior of these cockpit instrument components, use the associated properties.

- `uiaeroairspeed` — Create airspeed indicator component.
- `uiaeroaltimeter` — Create altimeter component.
- `uiaeroclimb` — Create climb rate indicator component.
- `uiaeroegt` — Create exhaust gas temperature (EGT) indicator component.
- `uiaeroheading` — Create heading indicator component.
- `uiaerohorizon` — Create artificial horizon component.
- `uiaerorpm` — Create revolutions per minute (RPM) indicator component.
- `uiaeroturn` — Create turn coordinator component.

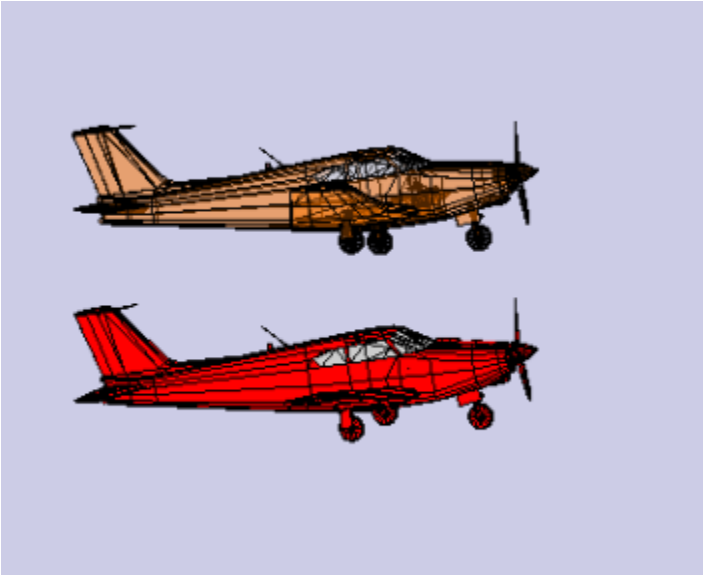
Here is an example of a flight instruments pane from “Aerospace Flight Instruments in App Designer” on page 5-104.

MATLAB Graphics-Based Animation

Use MATLAB graphics-based animation objects to visualize flight paths using MATLAB graphics software. These animation objects use the MATLAB `timeseries` object to visualize flight data.

- `Aero.Animation` — Visualize aerospace animation.
- `Aero.Body` — Create body object for use with animation object.
- `Aero.Camera` — Construct camera object for use with animation object.
- `Aero.Geometry` — Construct 3-D geometry for use with animation object.

Here is an example of a flight trajectory playback image from “Overlaying Simulated and Actual Flight Data” on page 5-31.



See Also

Related Examples

- “Visualize Trajectory and Attitude”

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 2-23.
- `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 2-31.
- `Aero.FlightGearAnimation` — Visualize flight data with the FlightGear simulator. For more information, see “Aero.FlightGearAnimation Objects” on page 2-42.

Aero.Animation Objects

The toolbox interface to animation objects uses the Handle Graphics capability. The “Overlaying Simulated and Actual Flight Data” on page 5-31 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.

See Also

`Aero.Animation` | `Aero.Body` | `Aero.Camera` | `Aero.Geometry`

Related Examples

- “Overlaying Simulated and Actual Flight Data” on page 5-31

Simulated and Actual Flight Data Using Aero.Animation Objects

Creating and Configuring an Animation Object

This topic presents the “Overlaying Simulated and Actual Flight Data” on page 5-31 example.

These functions create an animation object and configure 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.

- `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 the folder for the example.

```
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 2-25.

```
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.

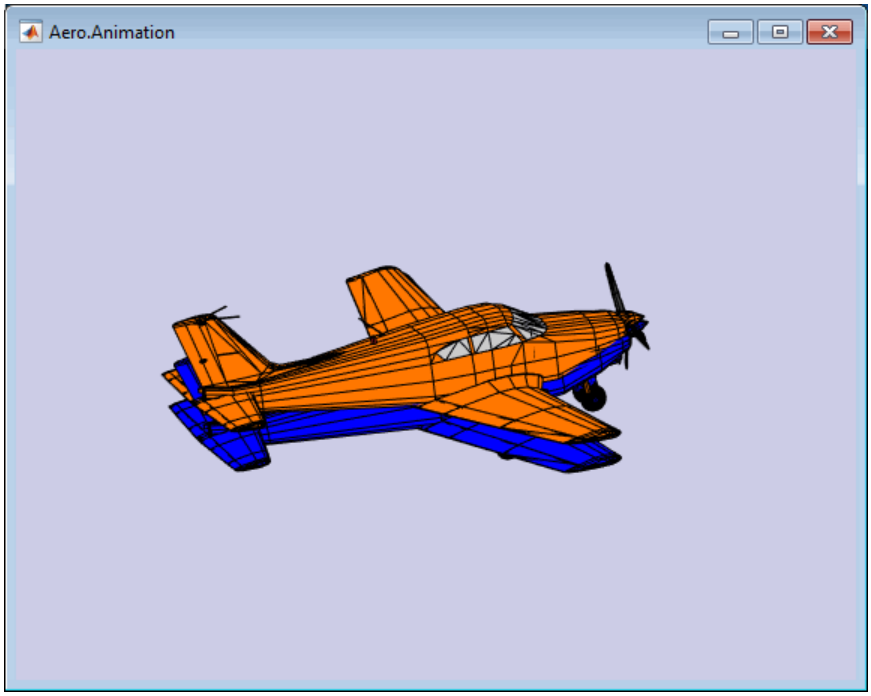
Note When choosing the video compression type, keep in mind that you will need the corresponding viewer software. For example, if you create an AVI format, you need a viewer such as Windows Media® Player to view the file.

After you play the animation as described in “Playing Back Flight Trajectories Using the Animation Object” on page 2-25, `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 2-26 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 2-25.

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 2-25, 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 folder for the example.

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);
   ```

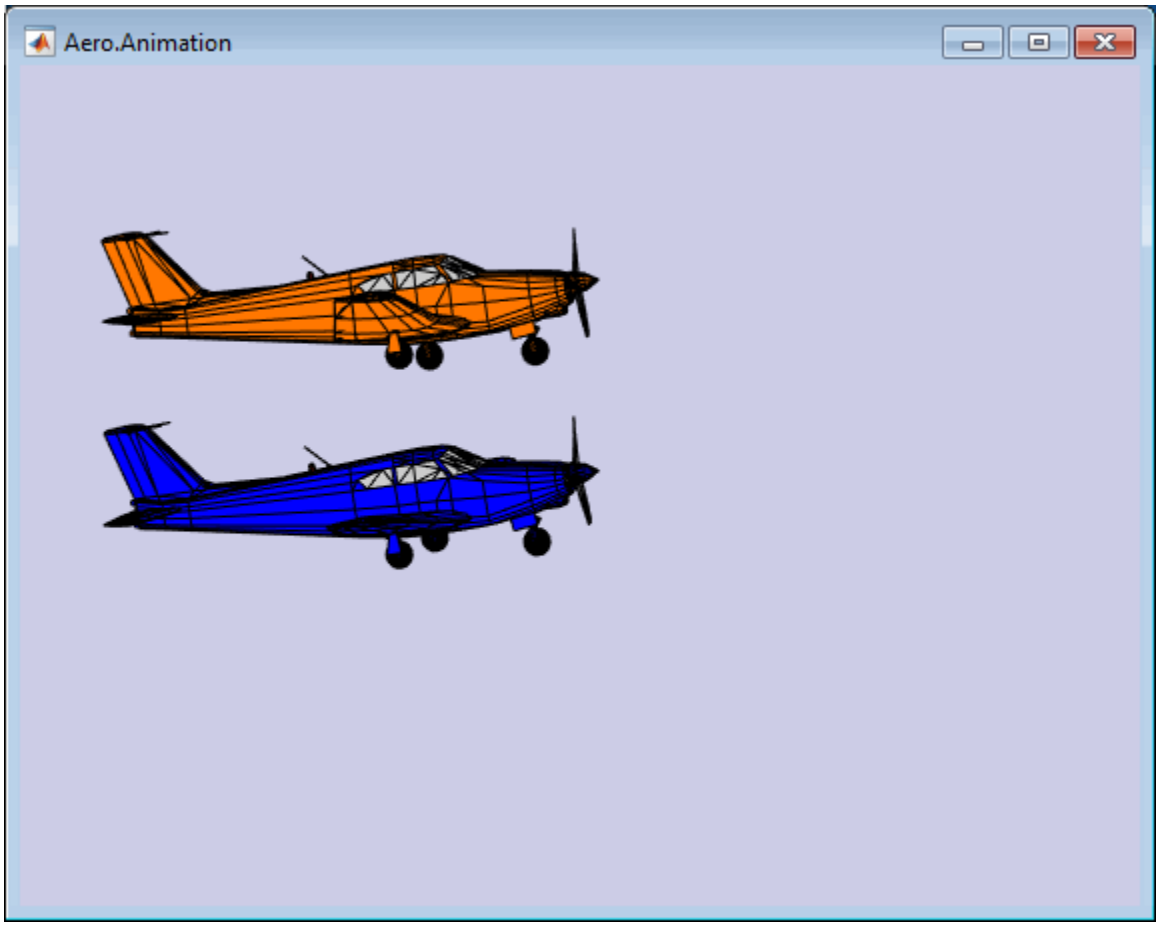
4. Reposition the bodies by first getting the current body position, then separating the bodies.

   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;
   ```

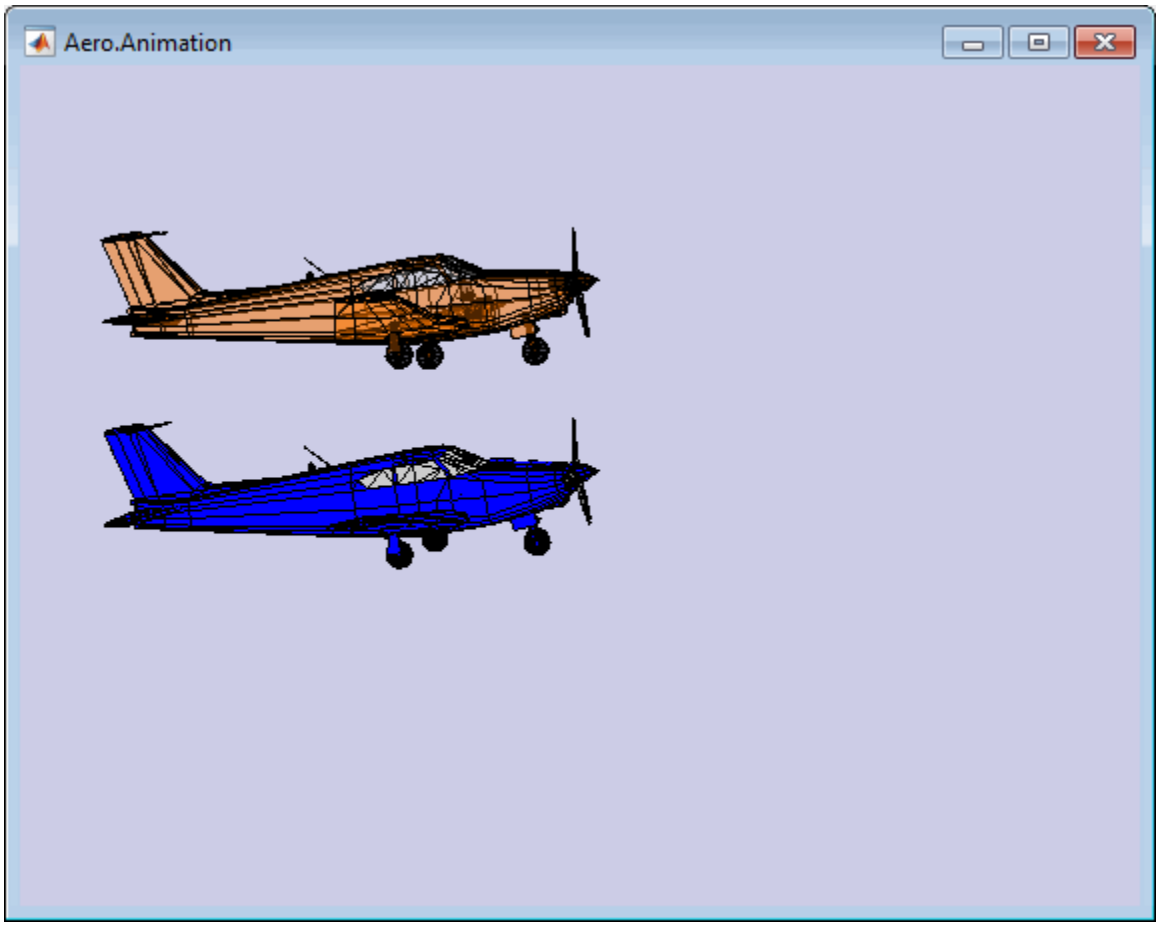
2. Set the face and edge alpha values that you want for the transparency.

   ```
   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
```

See Also

`Aero.Animation` | `Aero.Body` | `Aero.Camera` | `Aero.Geometry`

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.

See Also

`Aero.VirtualRealityAnimation` | `Aero.Node` | `Aero.Viewpoint`

Related Examples

- “Visualize Aircraft Takeoff via Virtual Reality Animation Object” on page 2-32

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;
```

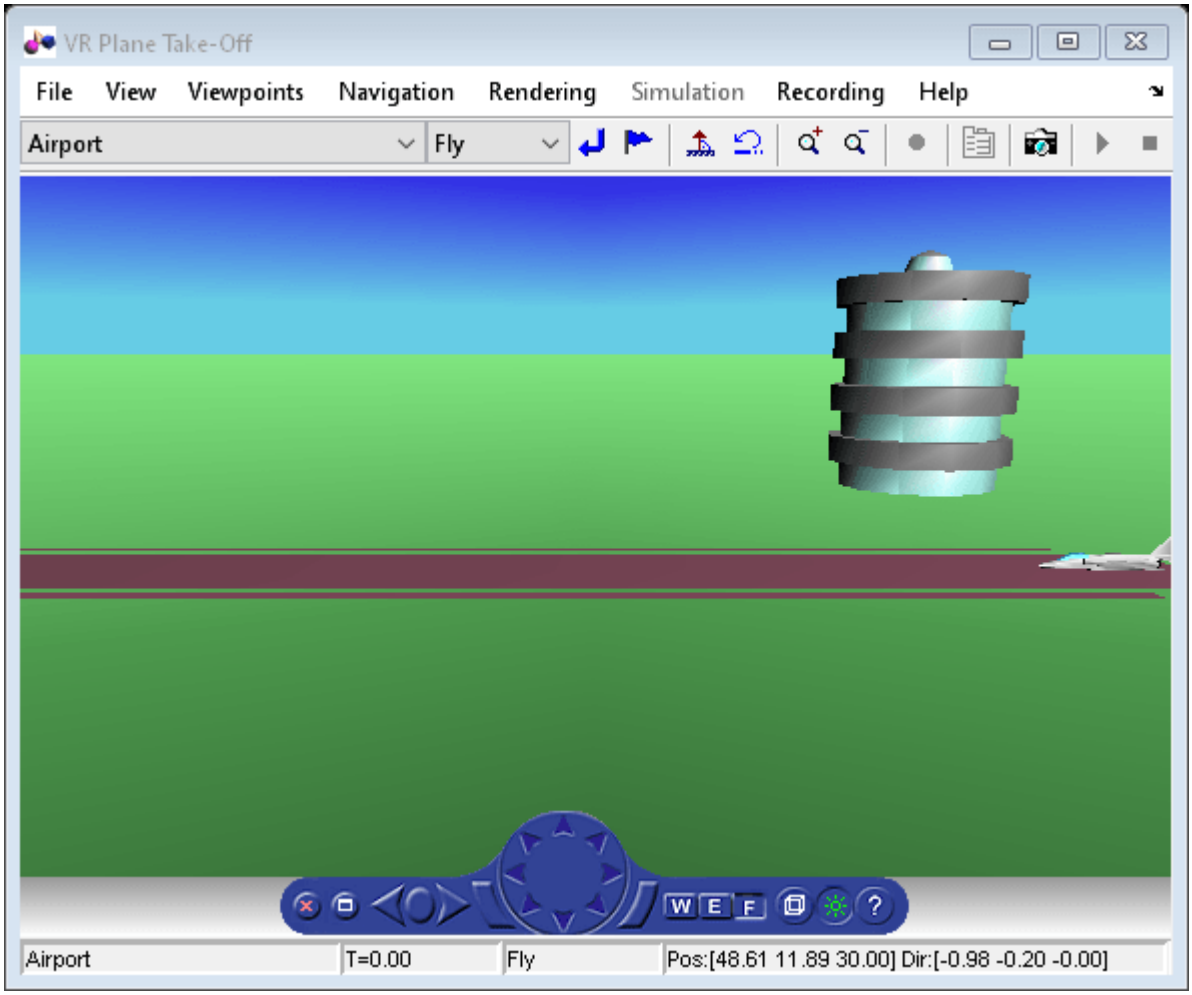
This code sets the .wrl file to be used in the virtual reality animation.

```
h.VRWorldFilename = "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();
```



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].

You can see the custom transformation function in the file 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;
```

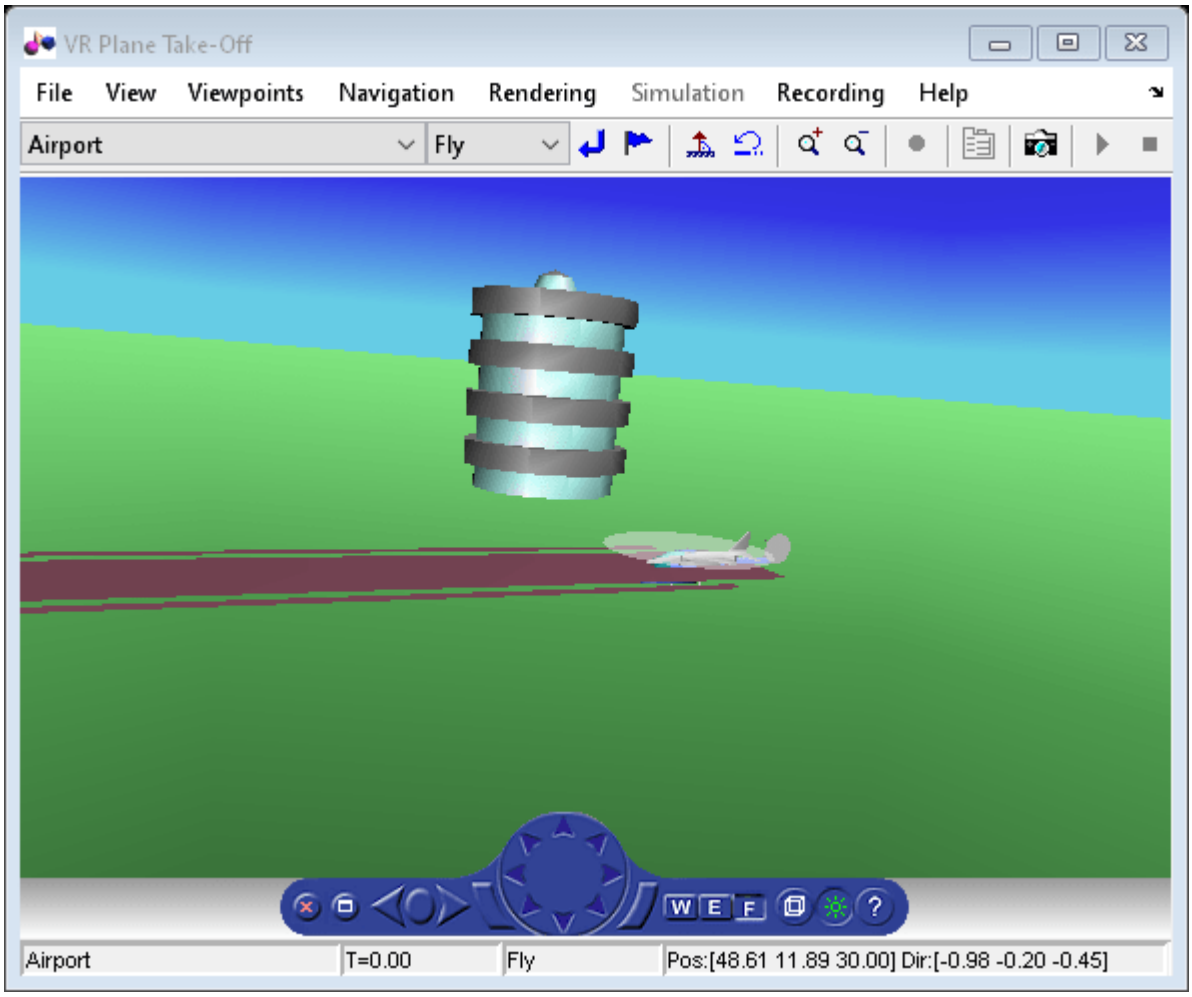
```
Node Information
1    Camera1
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
```

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

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

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 = 0;
h.addNode('Lynx',fullfile(pwd,"chaseHelicopter.wrl"));
```



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.Nodes{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

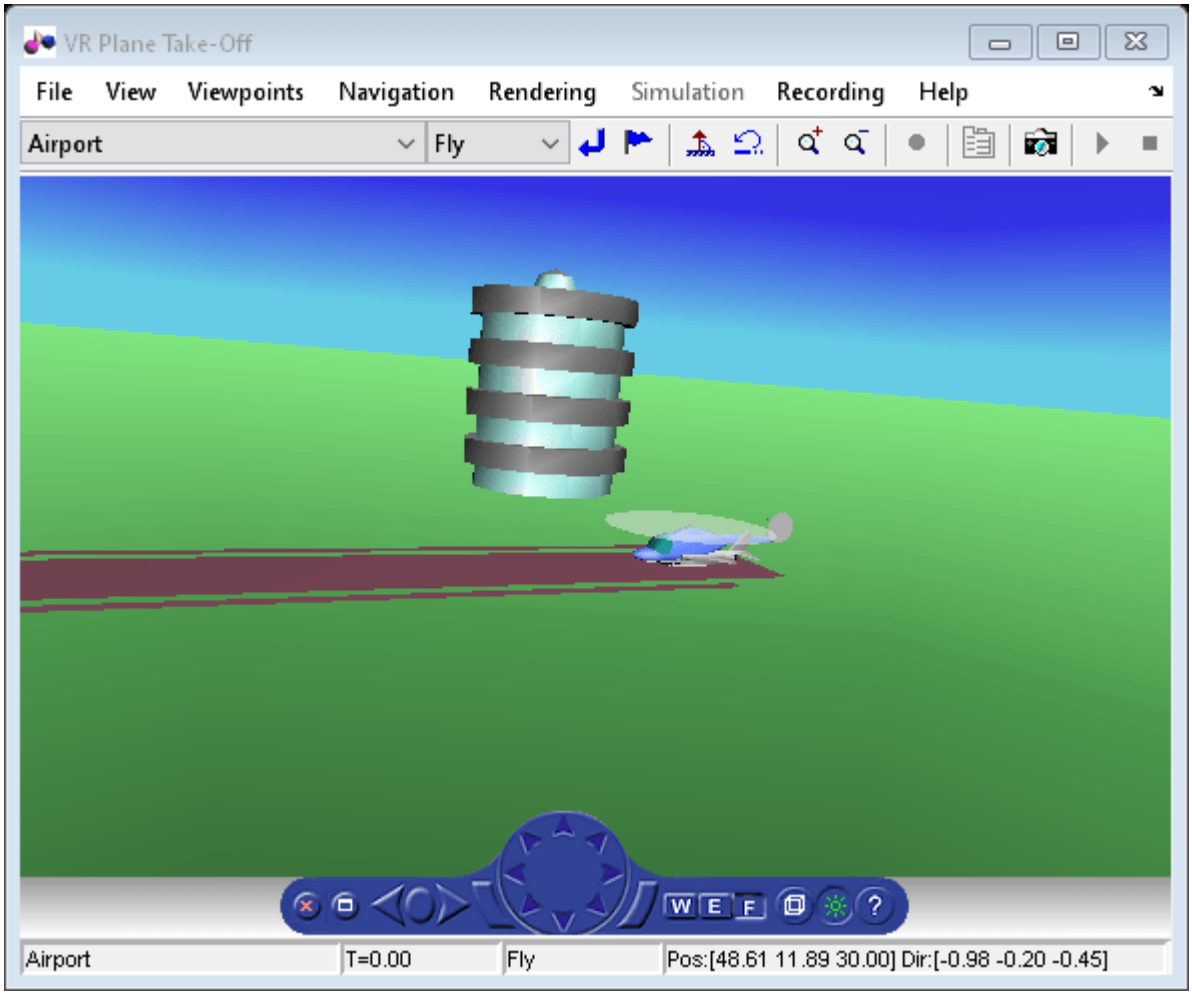
function (`vranimCustomTransform`) is used for this node as the preceding node. The previous call to `nodeInfo` returned the node index (2).

```
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');
```



Play Animation

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

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

Wait

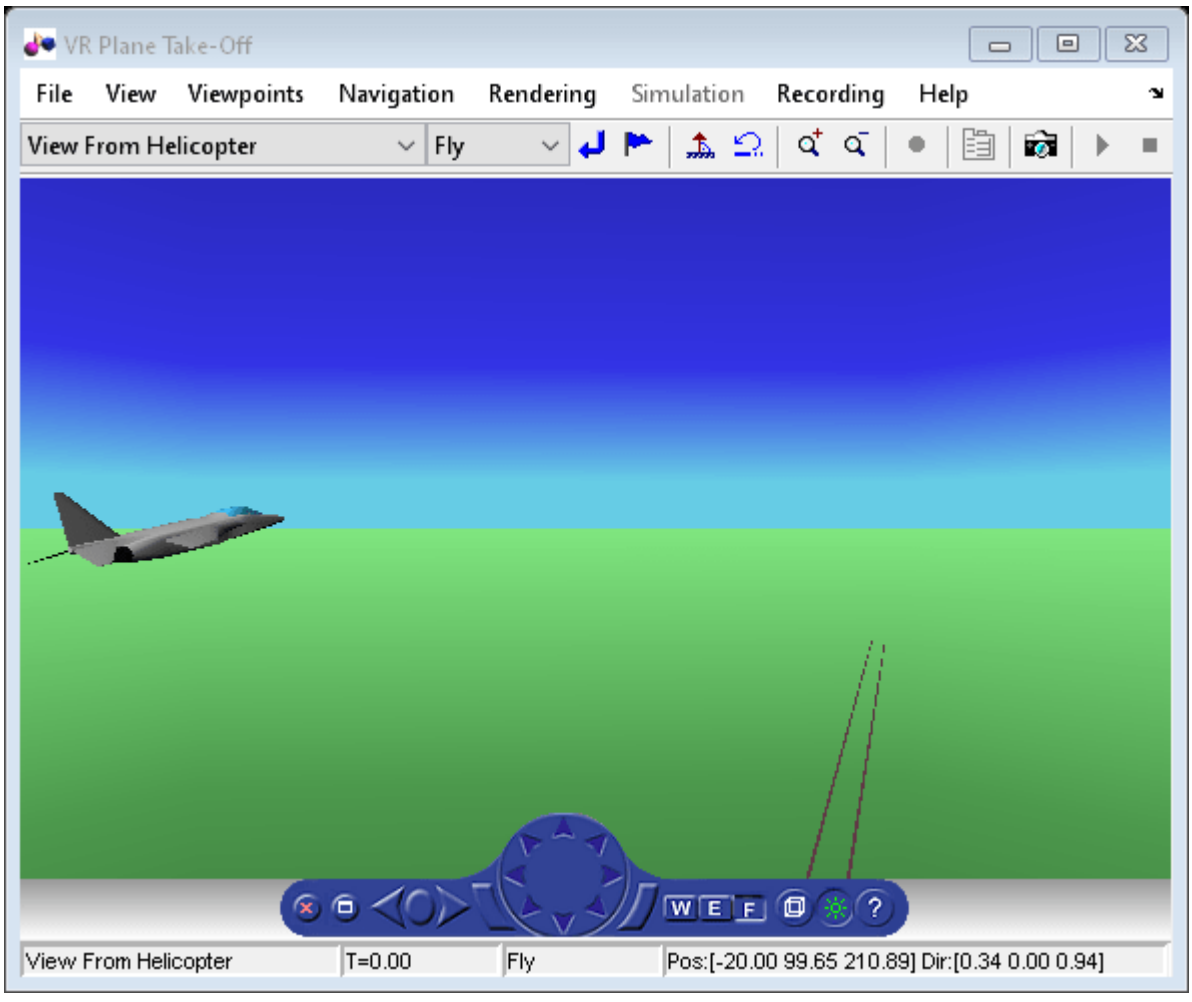
Wait for the animation to stop playing before the modifying the object.

```
h.wait();
```

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.

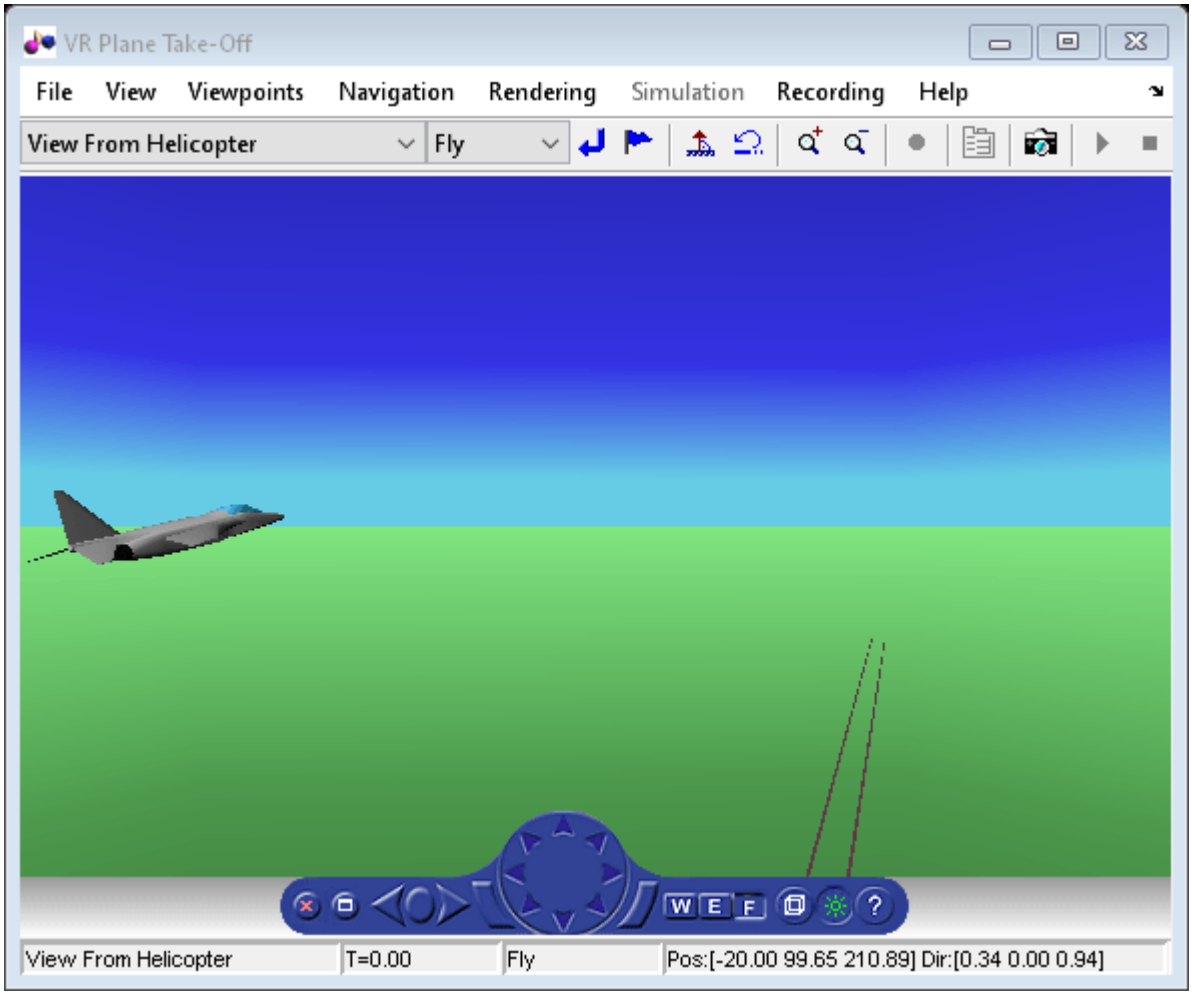
```
[~, idxChaseView] = find(strcmp('chaseView',h.nodeInfo));
h.Nodes{idxChaseView}.VRNode.orientation = [0 1 0 convang(200,'deg','rad')];
set(h.VRFigure,'Viewpoint','View From Helicopter');
```



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');
```



The scene from the helicopter viewpoint

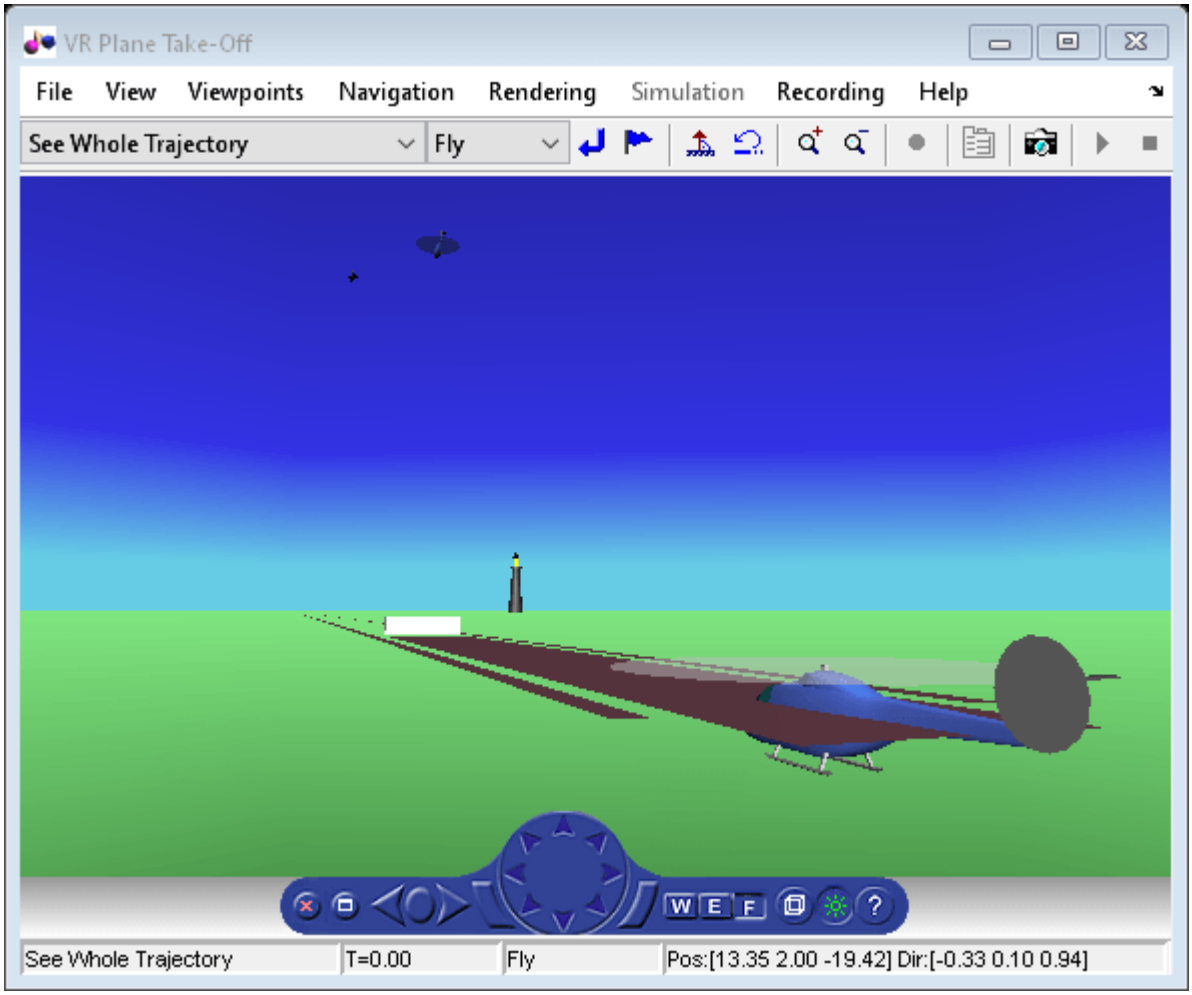
This code plays the animation.

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

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',"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
```

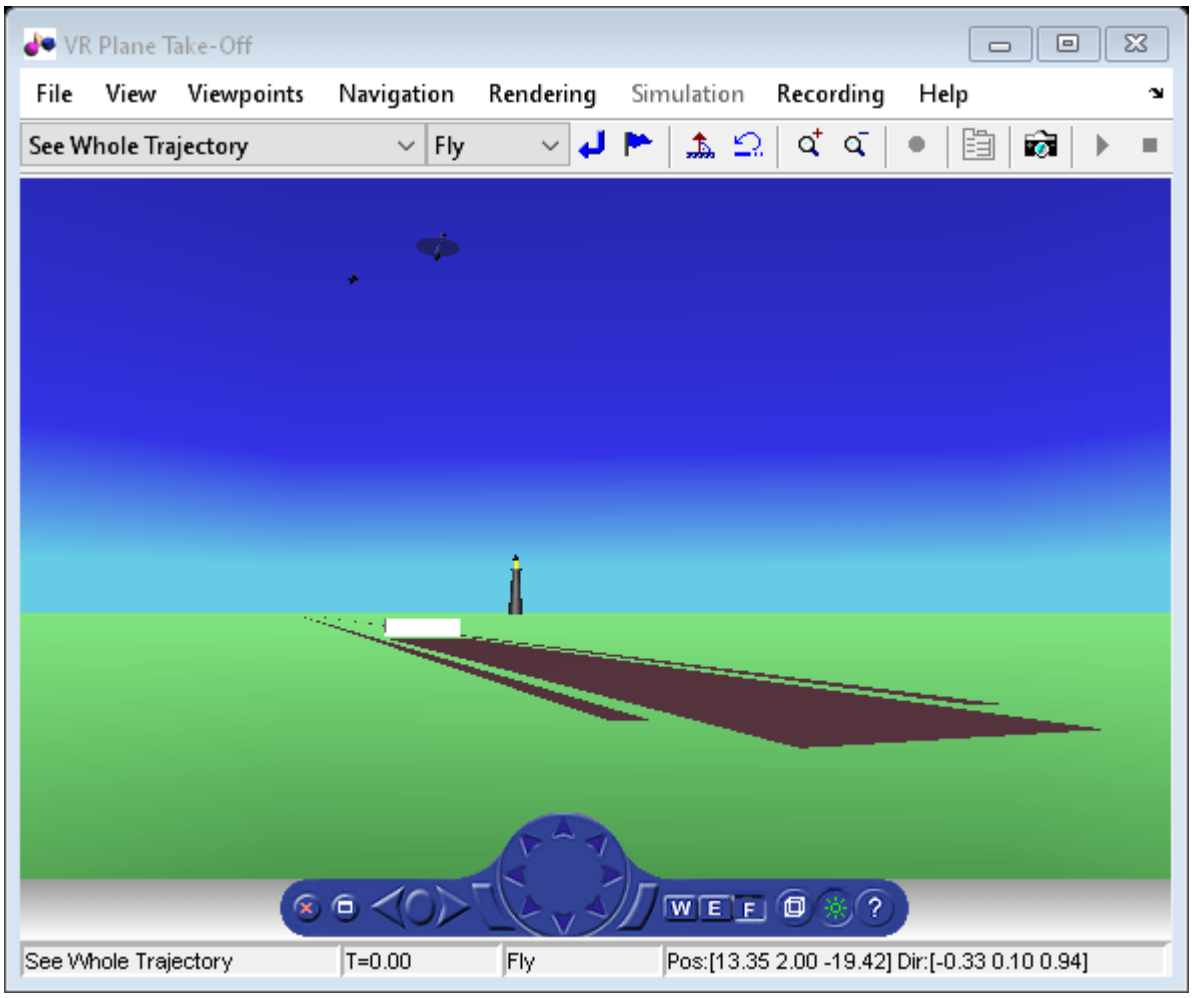
Adjust newly added helicopter to sit above runway.

```
[~, idxLynx1] = find(strcmp('Lynx1',h.nodeInfo));
h.Nodes{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');
```



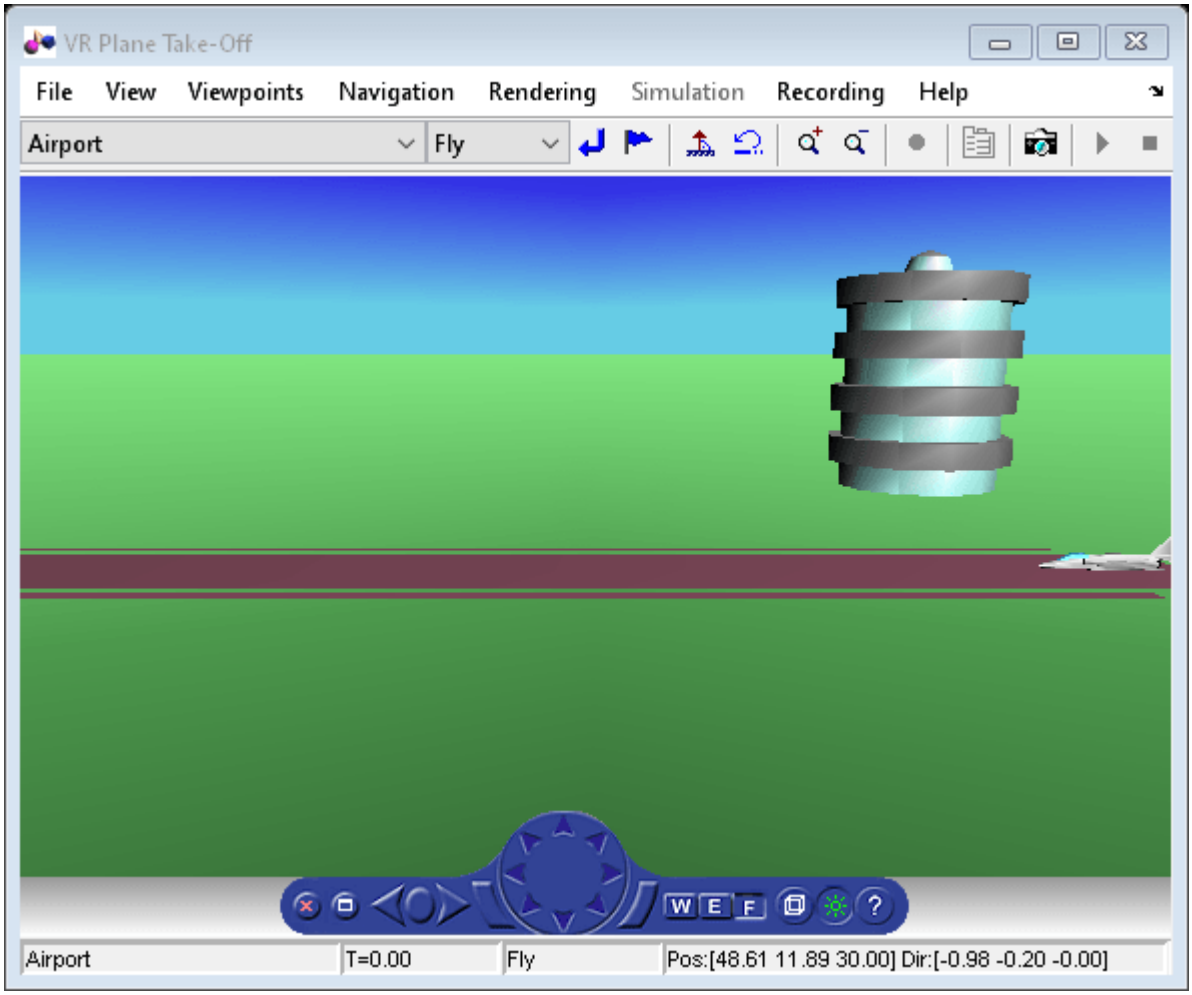

```
h.nodeInfo
```

```
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
```

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();
```



Close and Delete World

To close and `delete`

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

See Also

`Aero.Node` | `Aero.Viewpoint` | `Aero.VirtualRealityAnimation`

Related Examples

- “Aero.VirtualRealityAnimation Objects” on page 2-31

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. For an example of the Aerospace Toolbox interface to the FlightGear flight simulator, see “Create a Flight Animation from Trajectory Data” on page 5-17.

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 2-44.

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/contact_us.html.
- 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 FlightGear versions starting from v2.6.

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 `https://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 2-42.
- 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
		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 2-42.

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 2-43.

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. 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\animation` folder to the `FlightGear\data\Aircraft` folder. This folder contains the preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file `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/animation` folder to the $*`FlightGearBaseDirectory`*`/data/Aircraft/` folder. This folder contains the preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file `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. 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/animation` 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 `HL20.xml` defines the geometry.

See Also

`Aero.FlightGearAnimation`

Related Examples

- “Create a Flight Animation from Trajectory Data” on page 5-17
- “Flight Trajectory Data” on page 2-47

Flight Trajectory Data

Loading Recorded Flight Trajectory Data

The flight trajectory data for this example is stored in a comma separated value formatted file. To read this data, use `readmatrix`.

```
tdata = readmatrix('asthl20log.csv');
```

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\FlightGear<your_FlightGear_version>';
   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: '2018.1'
            OutputFileName: 'runfg.bat'
  FlightGearBaseDirectory: 'C:\Program Files\FlightGear<your_FlightGear_version>'
        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 2-48).

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 2-48).

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 2-49).

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 2-49). If you cannot view scenes, see “Installing Additional FlightGear Scenery” on page 2-44.

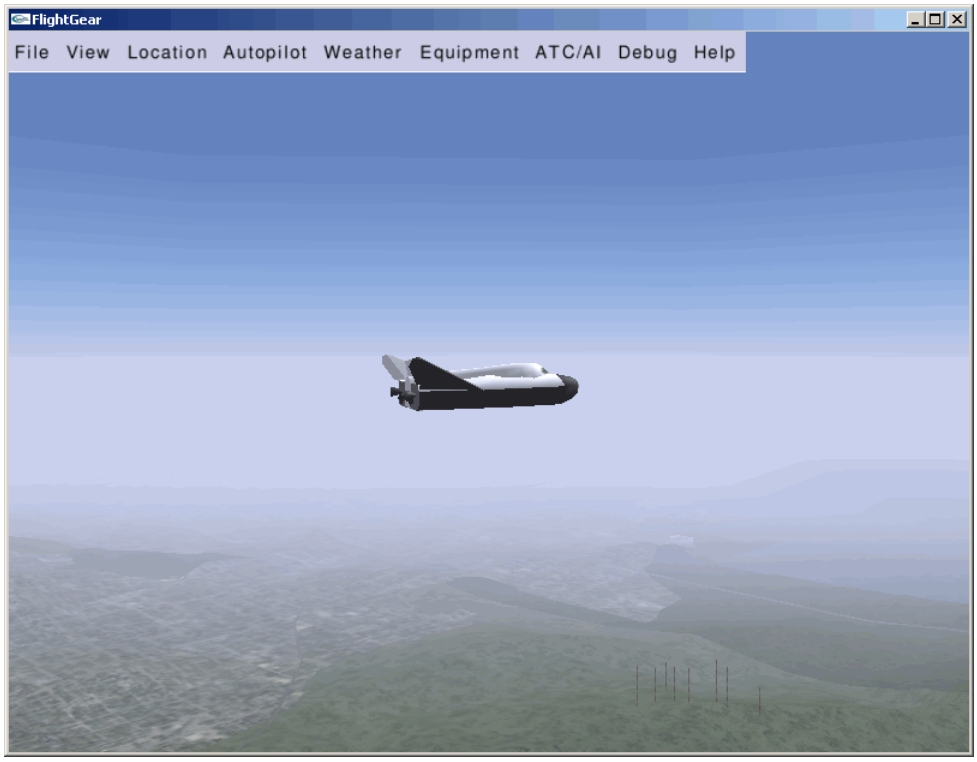
Tip If FlightGear uses more computer resources than you want, you can change its scheduling priority to a lesser one. For example, see commands like Windows `start` and Linux `nice` or their equivalents.

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.



See Also

`Aero.FlightGearAnimation`

Related Examples

- “Aero.FlightGearAnimation Objects” on page 2-42

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. It relies on the “Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99 example.

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 2-52

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 2-51.

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 `flightInstrumentsAnimationCallback` helper function, which is available only if you click **Open Live Script** for the `Display Flight Trajectory Data Using Flight Instruments and Flight Animation` example.

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

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” on page 5-99.

See Also

Functions

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

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 5-99
- “Flight Instrument Components in App Designer” on page 2-54

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 “Develop Apps Using App Designer”. 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:

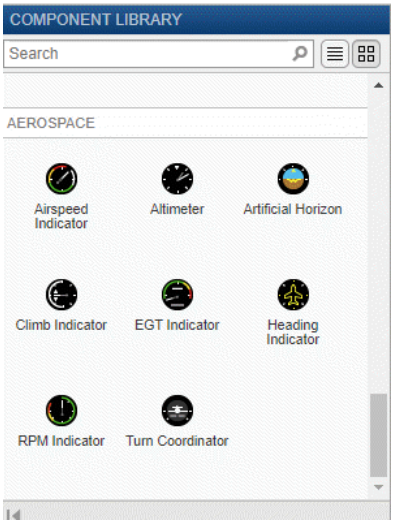
   ```
   appdesigner
   ```

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 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 theta psi]
   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','deg');
       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);
```

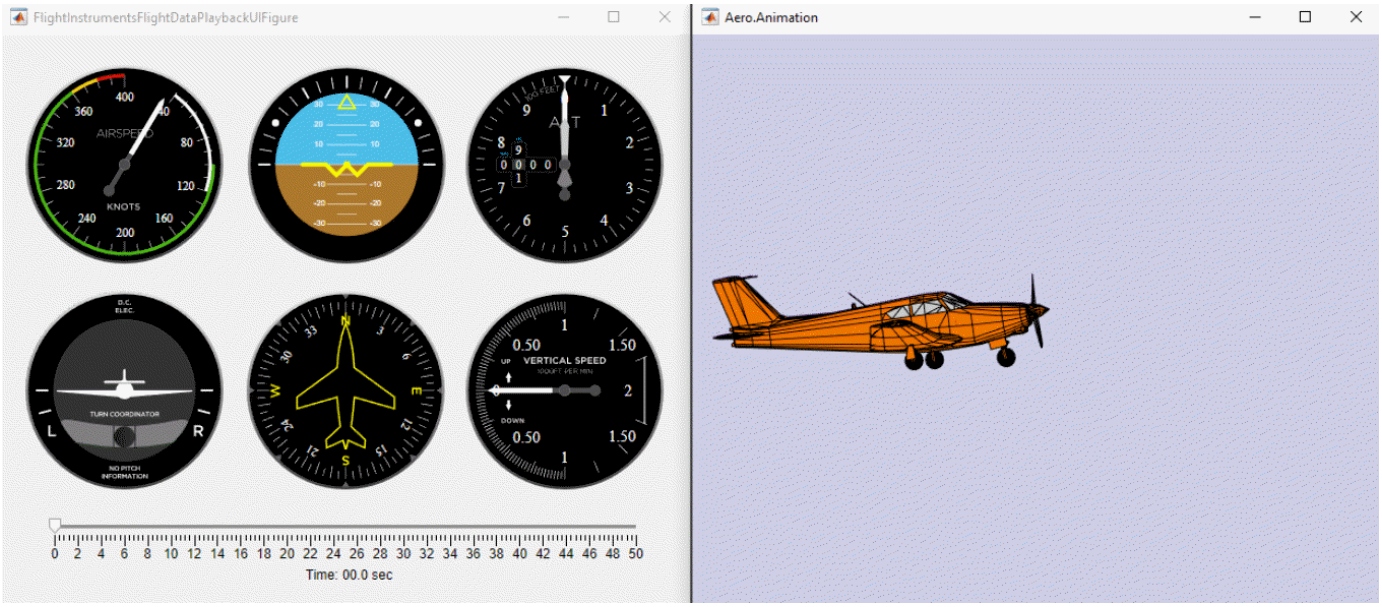
Save and Run the App

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 5-104.

See Also

Functions

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

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 2-51
- “Aerospace Flight Instruments in App Designer” on page 5-104
- “Flight Instruments”
- “Develop Apps Using App Designer”

Work with Fixed-Wing Aircraft Using Functions

To easily create fixed-wing aircraft in the Aerospace Toolbox, use the `fixedWingAircraft` function and its supporting functions. Use these functions to:

- Define aircraft dynamics.
- Define aircraft dynamics from DATCOM files.
- Perform static stability analyses using object methods.
- Generate state-space representation with linearization methods.

Suggested Workflow

Consider this workflow when designing and building your fixed-wing aircraft objects with these functions.

Objective	Use
Define a fixed-wing aircraft.	`fixedWingAircraft` — The aircraft object holds the main definition of fixed-wing aircraft. The aircraft has a main set of body coefficients, which you can manipulate with the `fixedWingCoefficient` function.
Define the condition (state) of a fixed-wing aircraft at an instance in time.	`fixedWingState` — Use this function when: • Your calculations require a specific aircraft state, such as those for forces and moments. • Gathering specific points of data from `Simulink.LookupTable` objects. This action requires a Simulink license.
Define data for any and all coefficients that describe the behavior of the aircraft.	`fixedWingCoefficient` — Numeric coefficient objects hold the data for all coefficients that describe the behavior of the aircraft.
Define an aerodynamic surface on a fixed-wing aircraft.	`fixedWingSurface` — Control surfaces hold the definitions of the aircraft aerodynamic surfaces.
Define a thrust vector on a fixed-wing aircraft.	`fixedWingThrust` — Thrust vector objects hold the definitions of the aircraft thrust.
Define the fixed-wing aircraft state environment.	`aircraftEnvironment` — Aircraft environment objects hold the fixed-wing aircraft state environment such as air temperature, pressure, density, and gravity.
Define the properties for the fixed-wing aircraft.	`aircraftProperties` — Aircraft property objects define common properties to maintain and define aircraft. Use this object throughout the fixed-wing aircraft design process.

After creating these fixed-wing aircraft components, use object methods to work with them. For example, use object methods to perform static stability and linear analysis.

Static Stability Analysis

To perform static stability analysis of your fixed-wing aircraft, use associated object methods:

1. Create a criteria table against which to perform static stability analysis.

   To create a criteria table, use the `Aero.FixedWing.criteriaTable` method. This method creates a 6-by-*N* table, where *N* is the number of criteria variables.
2. To evaluate the changes in forces and moments after a perturbation as either greater than, equal to, or less than 0 using the matching entry in the criteria table, use the `staticStability` method. The method uses this evaluation process.

   - If the evaluation of a criteria is met, the aircraft is statically stable at that condition.
   - If the evaluation of a criteria is not met, the aircraft is statically unstable at that condition.
   - If the result of the perturbation is `0`, the aircraft is statically neutral at that condition.

   Use this method only in the preliminary design phase. The `staticStability` method does not perform a requirements-based analysis.

For more information on object methods, see “Analyze Fixed-Wing Aircraft with Objects” on page 2-62.

For an example of static stability analysis, see “Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121 .

Linear Analysis

To perform the linear analysis of the fixed-wing object at a given fixed-wing state, use the `linearize` method. This method linearizes a fixed-wing aircraft around an initial state and creates a state-space model for the linear analysis. To perform linear analysis:

1. Calculate the static stability of the fixed-wing aircraft using the `staticStability` method.
2. Linearize the fixed-wing aircraft using the `linearize` method.

For an example of fixed-wing aircraft linear analysis, see “Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106.

Linear analysis requires the Control System Toolbox™ license.

See Also

`fixedWingAircraft` | `fixedWingState` | `fixedWingCoefficient` | `fixedWingSurface` | `fixedWingThrust` | `aircraftEnvironment` | `aircraftProperties`

Related Examples

- “Get Started with Fixed-Wing Aircraft” on page 5-170

Analyze Fixed-Wing Aircraft with Objects

To analyze fixed-wing aircraft in Aerospace Toolbox, use the `Aero.FixedWing` class and its supporting classes. These classes enable you to:

- Define aircraft dynamics
- Define aircraft dynamics from DATCOM files
- Perform static stability analyses
- Generate state-space representation with linearization methods

Suggested Workflow

As a guideline, consider this workflow when designing and building your fixed-wing aircraft with these classes:

To	Use
Define a fixed-wing aircraft.	`Aero.FixedWing` — `Aero.FixedWing` objects hold the main definition of fixed-wing aircraft. The object has a main set of body coefficients, which you can manipulate with the `Aero.FixedWing.Coefficient` object.
Define the condition (state) of a fixed-wing aircraft at an instance in time.	`Aero.FixedWing.State` — Use these objects when: • Your calculations require a specific aircraft state, such as those for forces and moments. • Gathering specific points of data from `Simulink.LookupTable` objects (requires a Simulink license).
To define data for any and all coefficients that describe the behavior of the aircraft.	`Aero.FixedWing.Coefficient` — `Aero.FixedWing.Coefficient` objects hold the data for all Coefficients that describe the behavior of the aircraft.
Define an aerodynamic surface on a fixed-wing aircraft.	`Aero.FixedWing.Surface` — `Aero.Aircraft.Surface` objects hold the definitions of the aircraft aerodynamic surfaces.
Define a thrust vector on a fixed-wing aircraft.	`Aero.FixedWing.Thrust` — `Aero.Aircraft.Thrust` Objects hold the definitions of the aircraft thrust.
Define the fixed-wing aircraft state environment.	`Aero.Aircraft.Environment` — `Aero.Aircraft.Environment` objects hold the fixed-wing aircraft state environment such as air temperature, pressure, density, gravity, and so forth.

To	Use
Define the properties for the fixed-wing aircraft.	`Aero.Aircraft.Properties` — `Aero.Aircraft.Properties` objects define common properties to maintain and define aircraft. Use this object throughout the fixed-wing aircraft design process.
To define the control states of a fixed-wing state.	`Aero.Aircraft.ControlState` — `Aero.Aircraft.ControlState` holds the definitions of the aircraft control surface deflection angles.

Static Stability Analysis

To perform static stability analysis of your fixed-wing aircraft:

1. Create a criteria table against which to perform static stability analysis.

   To create a criteria table, use the `Aero.FixedWing.criteriaTable` method. This method creates a 6-by-*N* table, where *N* is the number of criteria variables.
2. To evaluate the changes in forces and moments after a perturbation as either greater than, equal to, or less than 0 using the matching entry in the criteria table, use `staticStability` method. The method uses this evaluation process:

   - If the evaluation of a criteria is met, the aircraft is statically stable at that condition.
   - If the evaluation of a criteria is not met, the aircraft is statically unstable at that condition.
   - If the result of the perturbation is `0`, the aircraft is statically neutral at that condition.

   Use this method only in the preliminary design phase. The `staticStability` method does not perform a requirements-based analysis.

For an example of static stability analysis, see “Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121 .

Linear Analysis

To perform the linear analysis of the fixed-wing object at a given fixed-wing state, use the `linearize` method. This method linearizes a fixed-wing aircraft around an initial state and creates a state-space model for the linear analysis. To perform linear analysis:

1. Calculate the static stability of the fixed-wing aircraft using the `staticStability` method.
2. Linearize the fixed-wing aircraft using the `linearize` method.

For an example of fixed-wing aircraft linear analysis, see “Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106.

Linear analysis requires the Control System Toolbox license.

Examples

Aerospace Toolbox provides these examples to help you work with fixed-wing aircraft using the fixed-wing classes.

Action	Example
Create and analyze a fixed-wing aircraft in MATLAB using Cessna C182 geometry and coefficient data.	“Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121
Convert a fixed-wing aircraft to a linear time invariant (LTI) state-space model for linear analysis.	“Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106
Construct and define a custom state for a fixed-wing aircraft.	“Customize Fixed-Wing Aircraft with Additional Aircraft States” on page 5-113

See Also

`Aero.Aircraft.ControlState` | `Aero.Aircraft.Environment` | `Aero.Aircraft.Properties` | `Aero.FixedWing` | `Aero.FixedWing.Coefficient` | `Aero.FixedWing.State` | `Aero.FixedWing.Surface` | `Aero.FixedWing.Thrust`

Related Examples

- “Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121
- “Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106
- “Customize Fixed-Wing Aircraft with Additional Aircraft States” on page 5-113

Satellite Scenario Key Concepts

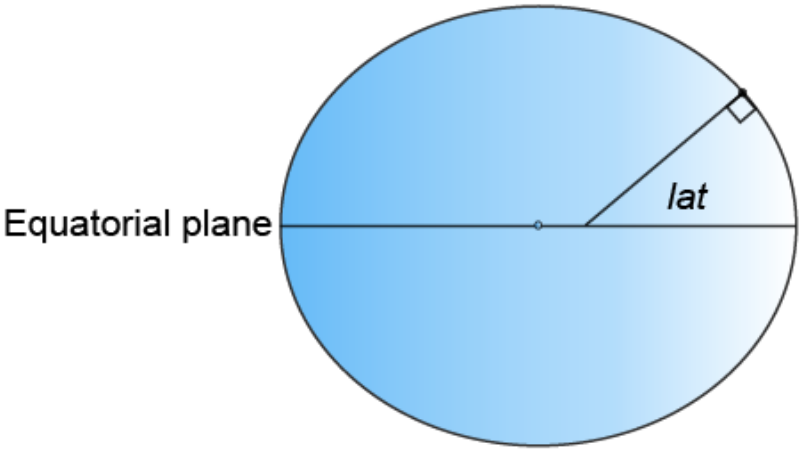
Aerospace Toolbox provides the ability to model and visualize satellites in orbit, compute access with ground stations, visualize and analyze communication links using the `satelliteScenario` object. This topic provides an overview of the technical terms frequently encountered in scenario visualization.

Coordinate Systems

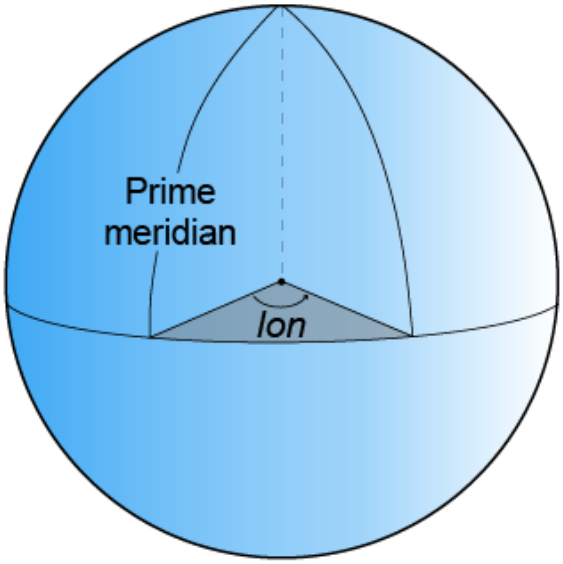
Geodetic Coordinates

A geodetic system uses the coordinates (*lat*,*lon*,*h*) to represent position relative to a reference ellipsoid. All geodetic coordinates in satellite scenario use the WGS84 ellipsoid as the reference ellipsoid. The coordinate origin of WGS 84 is meant to be located at the Earth's center of mass.

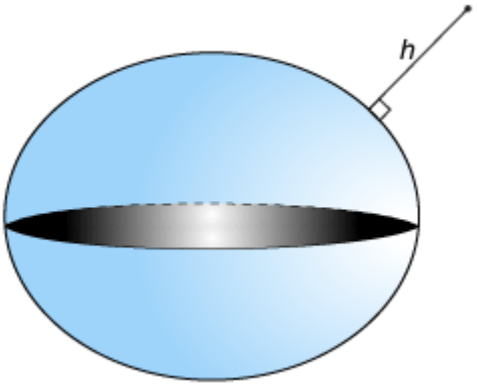
- *lat*, the latitude, originates at the equator. More specifically, the latitude of a point is the angle a normal to the ellipsoid at that point makes with the equatorial plane, which contains the center and equator of the ellipsoid. An angle of latitude is within the range [–90°, 90°]. Positive latitudes correspond to north and negative latitudes correspond to south.



- *lon*, the longitude, originates at the prime meridian. More specifically, the longitude of a point is the angle that a plane containing the ellipsoid center and the meridian containing that point makes with the plane containing the ellipsoid center and prime meridian. Positive longitudes are measured in a counterclockwise direction from a vantage point above the North Pole. Typically, longitude is within the range [–180°, 180°] or [0°, 360°].



- h, the ellipsoidal height, is measured along a normal of the reference spheroid.

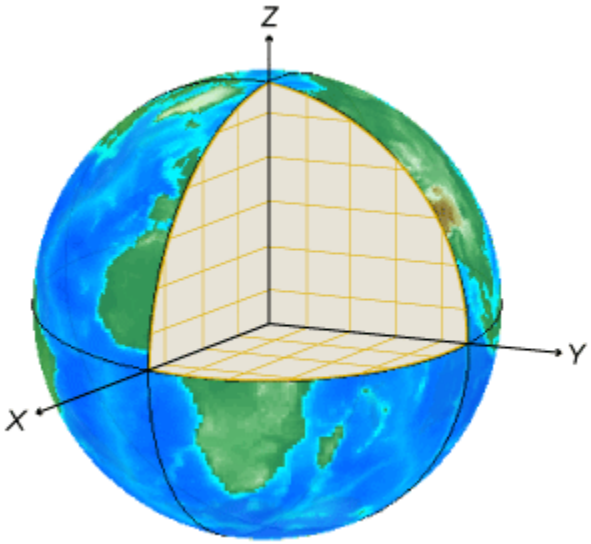


Earth-Centered Earth-Fixed Coordinates

An Earth-centered Earth-fixed (ECEF) system uses the Cartesian coordinates (X,Y,Z) to represent position relative to the center of the reference ellipsoid. The distance between the center of the ellipsoid and the center of the Earth depends on the reference ellipsoid.

- The positive X-axis intersects the surface of the ellipsoid at 0° latitude and 0° longitude, where the equator meets the prime meridian.
- The positive Y-axis intersects the surface of the ellipsoid at 0° latitude and 90° longitude.
- The positive Z-axis intersects the surface of the ellipsoid at 90° latitude and 0° longitude, the North Pole.

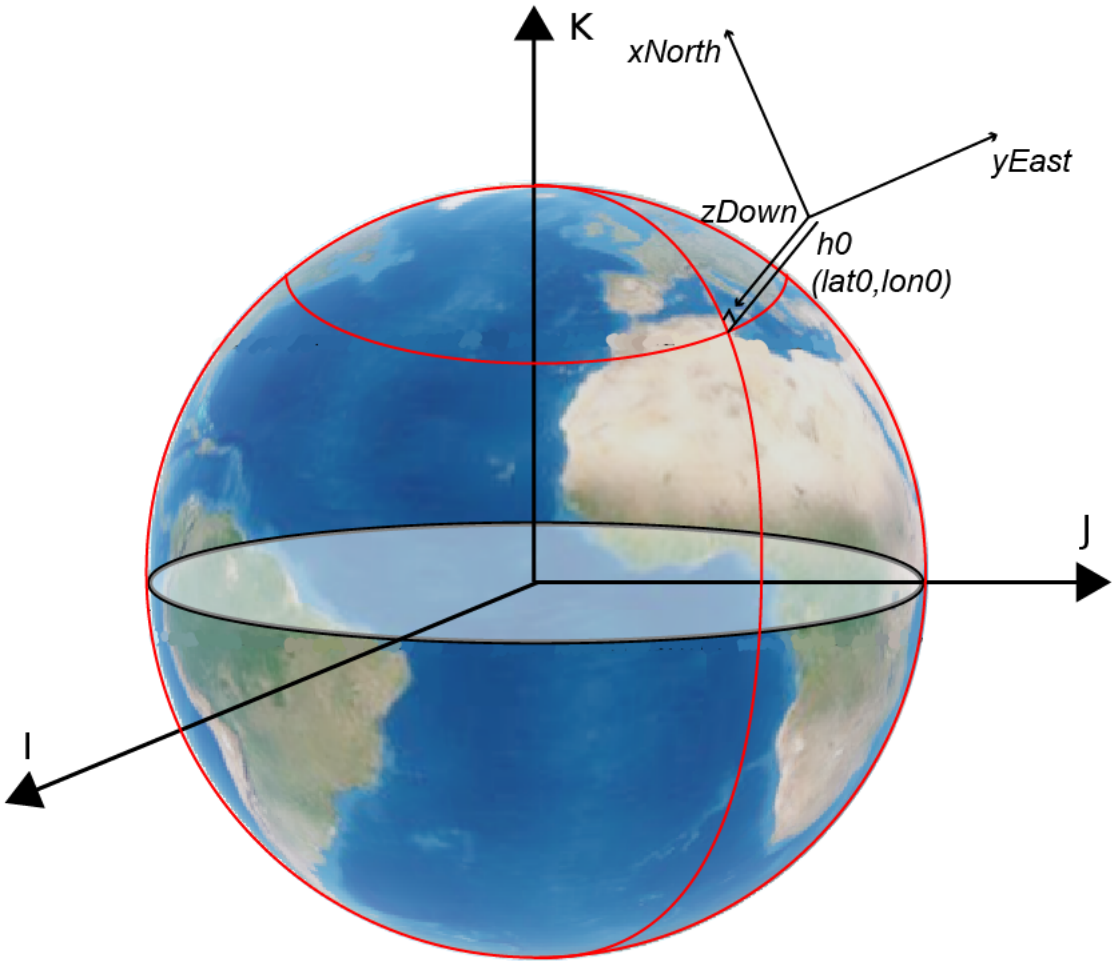
1



Frame of Reference and North East Down (NED) Frame

To describe a point in space, you need a frame of reference that does not rotate with respect to the stars. The Geocentric Celestial Reference Frame (GCRF), with the origin at the Earth's center and orthogonal vectors **I**, **J**, and **K**, is used as frame of reference while adding `satellite` objects to `satelliteScenario`. The fundamental plane is the **I**, **J** plane, which is closely aligned with the equator with a small offset, and **K** aligns closely with the north pole. You can describe the location of a satellite using a position vector and a velocity vector in the geocentric-equatorial coordinate system.

1 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks®.



When referring to a satellites position, velocity, acceleration, orientation and angular velocity, the coordinate system in which they are expressed should always be mentioned. Global systems such as GCRF and geodetic systems describe the position of an object using a triplet of coordinates. Local systems such as NED, and AER systems require two triplets of coordinates: one triplet describes the location of the origin, and the other triplet describes the location of the object with respect to the origin.

A north-east-down (NED) system uses the Cartesian coordinates (*xNorth*,*yEast*,*zDown*) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (*lat0*,*lon0*,*h0*). Typically, the local origin of an NED system is above the surface of the Earth.

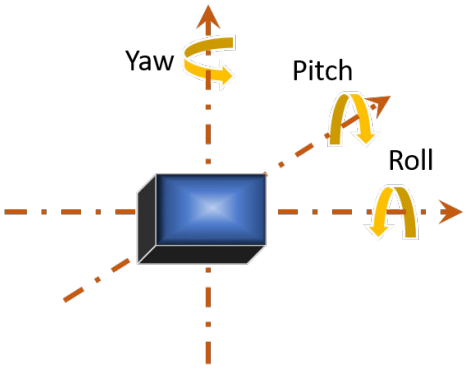
- The positive *xNorth*-axis points north along the meridian of longitude containing *lon0*.
- The positive *yEast*-axis points east along the parallel of latitude containing *lat0*.
- The positive *zDown*-axis points downward along the ellipsoid normal.

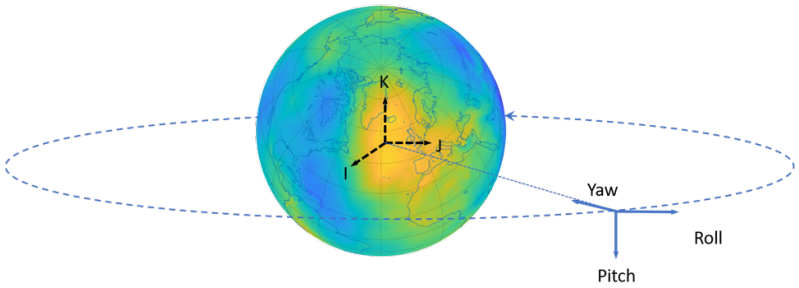
An NED coordinate system is commonly used to specify location relative to a moving satellite. Note that the coordinates are not fixed to the frame of the satellite.

Roll, Pitch, and Yaw

Three lines run through a satellite and intersect at right angles at the satellite's center of mass. These axes move with the satellite and rotate relative to the Earth along with the craft.

- Rotation around the front-to-back axis is called roll.
- Rotation around the side-to-side axis is called pitch.
- Rotation around the vertical axis is called yaw.



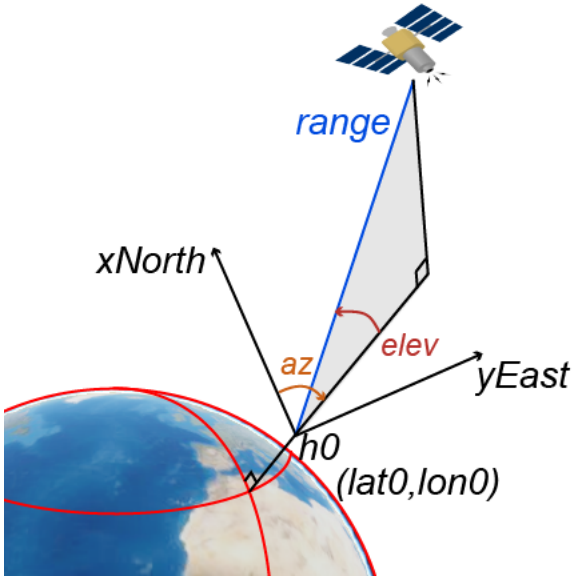


The yaw, pitch, and roll angles of satellites follow an ISO convention. These angles have positive clockwise directions when looking in the positive direction of the axes. Unless otherwise specified, by default Aerospace Toolbox uses yaw-pitch-roll rotation order for these angles.

Azimuth-Elevation-Range Coordinates

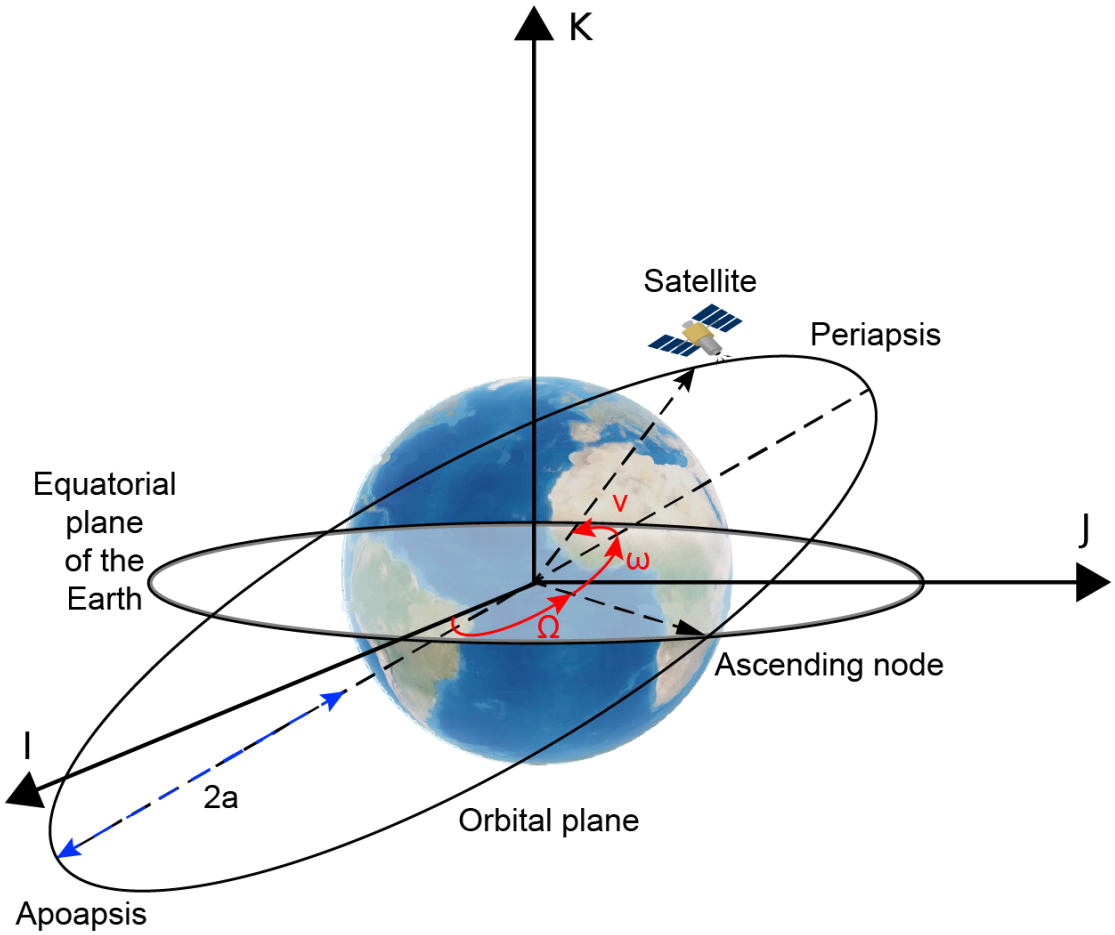
An azimuth-elevation-range (AER) system uses the spherical coordinates (*az*,*elev*,*range*) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (*lat0*,*lon0*,*h0*). Azimuth, elevation, and slant range depend on a local Cartesian system, for example, an NED system.

- *az*, the azimuth, is the clockwise angle in the (*xNorth*)(*yEast*)-plane from the positive *xNorth*-axis to the projection of the object into the plane.
- *elev*, the elevation, is the angle from the (*xNorth*)(*yEast*) plane to the object.
- *range*, the slant range, is the Euclidean distance between the object and the local origin.



Orbital Elements

Orbital elements are parameters required to uniquely identify a specific orbit. It takes at least six parameters to uniquely define an orbit and a satellite's position within the orbit. Three of the parameters describe what the orbital plane looks like and the position of the satellite in the ellipse, and the other three parameters describe how that plane is oriented in the celestial inertial reference frame and where the satellite is in that plane. These six parameters are called the Keplerian elements or orbital elements.



In this diagram, the orbital plane (yellow) intersects a reference plane (gray). For Earth-orbiting satellites, the reference plane is usually the I-J plane of the Geocentric Celestial Reference Frame (GCRF).

Two elements define the shape and size of the ellipse:

- **Eccentricity (e)** — Shape of the ellipse, describing how elongated it is compared to a circle.
- **Semimajor axis (a)** — Sum of the periapsis and apoapsis distances divided by two. Periapsis is the point at which an orbiting object is closest to the center of mass of the body it is orbiting. Apoapsis is the point at which an orbiting object is farthest away from the center of mass of the body it is orbiting. For classic two-body orbits, the semimajor axis is the distance between the centers of the bodies.

The next two elements define the orientation of the orbital plane in which the ellipse is embedded:

- **Inclination (i)** — Vertical tilt of the ellipse with respect to the reference plane, measured at the ascending node (where the orbit passes upward through the reference plane, the green angle *i* in the diagram). Tilt angle is measured perpendicular to line of intersection between orbital plane and reference plane. Any three points on an ellipse will define the ellipse orbital plane.

Starting with an equatorial orbit, the orbital plane can be tilted up. The angle tilted up from the equator is referred to as the inclination angle, *i*. Since the center of the earth must always be in the orbital plane, the point in the orbit where the satellite passes the equator on its way up is referred to as the ascending node, and the point where the satellite passes the equator on the way down is the descending node. Drawing a line through these two points on the equator is what defines the line of nodes.

- **Right ascension of ascending node (Ω)** — Horizontal orientation of the ascending node of the ellipse (where the orbit passes upward through the reference plane) with respect to the reference frame's I axis.

  The rotation of the right ascension of the ascending node (RAAN) can be any number between 0 and 360°.

The remaining two elements are as follows:

- **Argument of periapsis (ω)** — Orientation of the ellipse in the orbital plane, as an angle measured from the ascending node to the periapsis.
- **True Anomaly (v)** — Position of the orbiting body along the ellipse at a specific time. The satellites position on the path is measured counter-clockwise from periapsis and is called the true anomaly, ν.

Two Line Element (TLE) Files

Aerospace Toolbox accepts Two Line Element (TLE) files as inputs to `satellite`. To download TLE files, visit the Space track website.

A two-line element set is a data format encoding a list of orbital elements of an Earth orbiting object for a given point in time, the *epoch*. Orbital elements parameters can be encoded as text in a number of formats. The most common of them is the NASA/NORAD "two-line elements" format. As commonly used today, each satellite gets three lines – one line containing the satellite's name, followed by the standard two lines of elements.

Data for each satellite consists of three lines.

```
Satellite 1
1 25544U 98067A   04236.56031392  .00020137  00000-0  16538-3 0  9993
2 25544  51.6335 344.7760 0007976 126.2523 325.9359 15.70406856328906
```

- Line 1 is a eleven-character satellite name.
- Line 2 and 3 are the standard Two-Line element set format identical to that used by NORAD and NASA.

Column	Description	Example
1	Line Number	`1`
3 — 7	Satellite Number	`25544`
8	Elset Classification	`U`
10 — 17	International Designator	`98067A`
19 — 32	Element Set Epoch (UTC)	`04236.56031392`
34 — 43	First derivative of the Mean Motion with respect to time	`.00020137`

Column	Description	Example
45 — 52	Second derivative of the Mean Motion with respect to Time (decimal point assumed)	`00000-0`
54 — 61	BSTAR Drag Term.	`16538-3`
63	Element set type	`0`
65 — 68	Element number	`999`
69	Check Sum (Modulo 10)	`3`

Column	Description	Examples
1	Line Number of Element Data	`2`
3 — 7	Satellite Number	`25544`
9 — 16	Inclination [Degrees]	`51.6335`
18 — 25	Right Ascension of the Ascending Node [Degrees]	`344.7760`
27 — 33	Eccentricity (Leading decimal point assumed)	`0007976`
35 — 42	Argument of Perigee [Degrees]	`126.2523`
44 — 51	Mean Anomaly [Degrees]	`325.9359`
53 — 63	Mean Motion [Revs per day]	`15.70406856`
64 — 68	Revolution number at epoch [Revs]	`32890`
69	Check Sum (Modulo 10)	`6`

Depending on the application and object orbit, the data derived from TLEs older than 30 days can become unreliable. Orbital positions can be calculated from TLEs through the SGP4 and SDP4 algorithms.

References

[1] “HSF - Orbital Elements.” Accessed November 30, 2020. https://spaceflight.nasa.gov/realdata/elements/graphs.html.

[2] “CelesTrak: ‘FAQs: Two-Line Element Set Format,” March 26, 2016. https://web.archive.org/web/20160326061740/http://celestrak.com/columns/v04n03/.

See Also

Objects
`satellite` | `satelliteScenario` | `groundStation` | `access` | `satelliteScenarioViewer`

Functions
`show` | `play`

More About

- “Satellite Scenario Overview” on page 2-74
- “Model, Visualize, and Analyze Satellite Scenario” on page 2-77

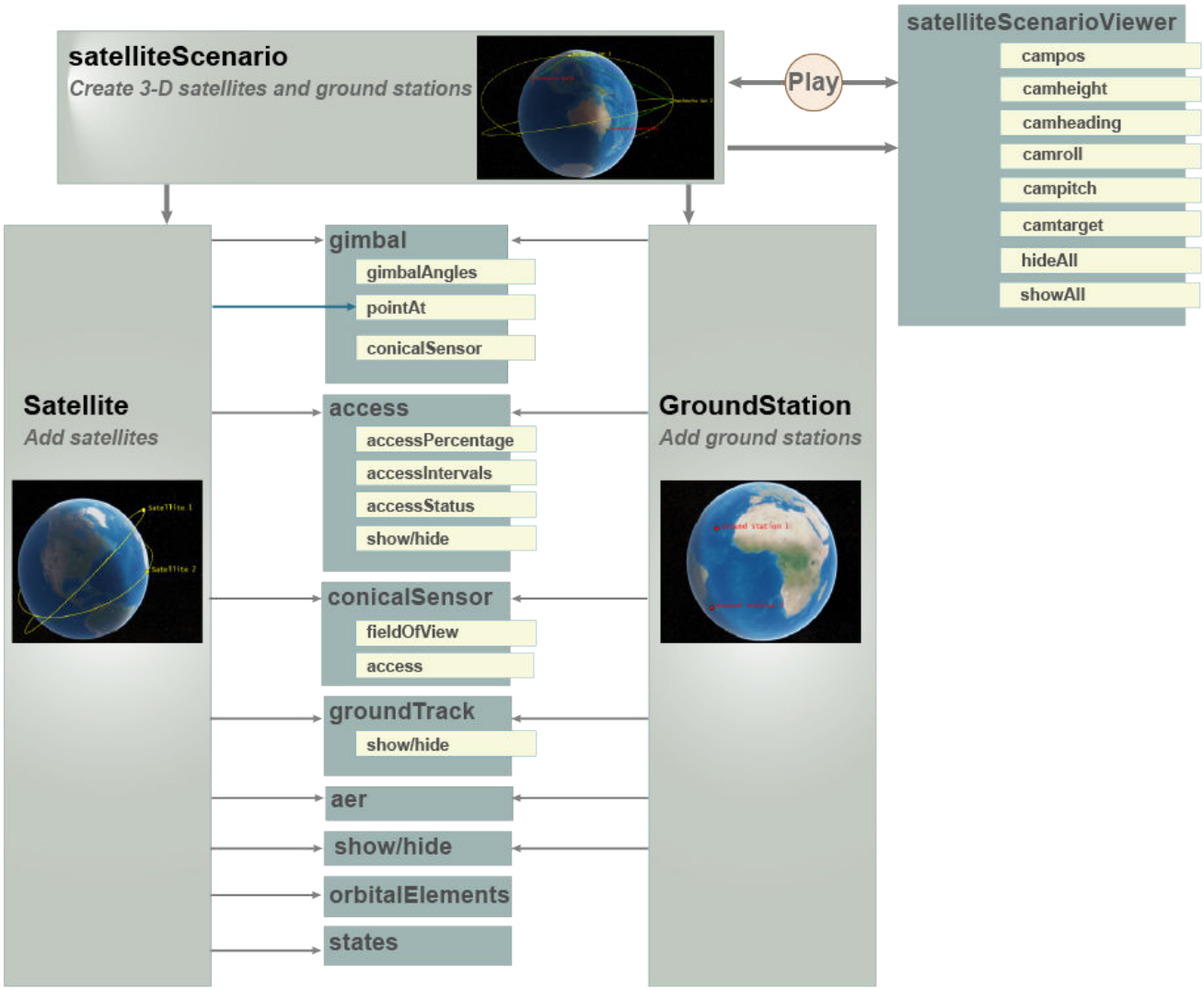
Satellite Scenario Overview

You can build a complete satellite scenario simulation using functions and objects. You can extend satellite scenarios for detailed communication simulations using Aerospace Toolbox. The workflow for satellite scenario simulation consists of four main components. These components are

- `satelliteScenario` represents a 3-D arena consisting of satellites, ground stations, and the interactions between them. Use this object to model satellite constellations, model ground station networks, perform access analyses between the satellites and ground stations, and visualize the results.
- `satellite` adds satellites to the scenario using two line element (TLE) files or orbital elements. For more details on orbital elements and TLE files, see “Two Line Element (TLE) Files” on page 2-72.
- `groundStation` adds ground stations to the scenario using default parameters or the specified latitude and longitude.
- `satelliteScenarioViewer` creates a 3D viewer for the scenario.
- `play` simulates the satellite scenario and plays the results in the visualization window specified by `satelliteScenarioViewer`.

Many of the methods included in the flowchart are created by multiple objects.

- `access` is created by `satellite`, `groundStation`, and `conicalSensor`.
- `show` and `hide` are created by `satellite`, `groundStation`, `groundTrack` and `access`.
- `conicalSensor` is created by `satellite`, `groundStation` and `gimbal`.



See Also

Objects
`satelliteScenario` | `satellite` | `access` | `groundStation` | `satelliteScenarioViewer` | `conicalSensor`

Functions
`show` | `play` | `hide`

More About

- “Satellite Scenario Key Concepts” on page 2-65

- “Model, Visualize, and Analyze Satellite Scenario” on page 2-77

Model, Visualize, and Analyze Satellite Scenario

This example shows how to model satellites in orbit, analyze access between the satellites and the ground stations, and visualize the fields of view and ground tracks of the satellites.

Create Satellite Scenario

Create a satellite scenario with the start time of 02-June-2020 8:23:00 AM UTC and a stop time of five hours later. Set the simulation sample time to `60` seconds.

```
startTime = datetime(2020,6,02,8,23,0);
stopTime = startTime + hours(5);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add Satellites to Scenario

Add satellites to the scenario from the `threeSatelliteConstellation` TLE file.

```
sat = satellite(sc,"threeSatelliteConstellation.tle");
```

Show the satellites in orbit and plot their ground tracks over 20 minutes.

```
show(sat)
groundTrack(sat,"LeadTime",1200);
```

Return Orbital Elements and Position of Satellites

Display the orbital elements of each satellite in the scenario.

```
ele1 = orbitalElements(sat(1))
```

```
ele1 = struct with fields:
                      MeanMotion: 9.1649e-04
                    Eccentricity: 1.0000e-03
                     Inclination: 55
    RightAscensionOfAscendingNode: 175.0000
              ArgumentOfPeriapsis: 100
                      MeanAnomaly: 174.9900
                           Period: 6.8557e+03
                            Epoch: 02-Jun-2020 18:43:16
                            BStar: 1.0000e-04
```

```
ele2 = orbitalElements(sat(2))
```

```
ele2 = struct with fields:
                      MeanMotion: 8.5025e-04
                    Eccentricity: 1.0000e-03
                     Inclination: 55
    RightAscensionOfAscendingNode: 350.0000
              ArgumentOfPeriapsis: 90
                      MeanAnomaly: 310.0877
                           Period: 7.3898e+03
                            Epoch: 02-Jun-2020 18:33:26
                            BStar: 1.0000e-04
```

```
ele3 = orbitalElements(sat(3))

ele3 = struct with fields:
                       MeanMotion: 8.6605e-04
                     Eccentricity: 1.0000e-03
                      Inclination: 55
    RightAscensionOfAscendingNode: 270
              ArgumentOfPeriapsis: 95
                      MeanAnomaly: 119.9007
                           Period: 7.2550e+03
                            Epoch: 02-Jun-2020 18:37:40
                            BStar: 1.0000e-04
```

Return the latitude, longitude, and altitude of the first satellite at time 02-June-2020 12:30:00 PM UTC.

```
time = datetime(2020,6,02,12,30,0);
pos = states(sat(1),time,"CoordinateFrame","geographic")

pos = 3×1
10^6 ×

    0.0000
   -0.0001
    1.4212
```

Add Ground Stations

Specify the latitudes and longitudes of the Madrid and Canberra Deep Space Communications Complexes as ground stations of interest.

```
name = ["Madrid Deep Space Communications Complex", ...
      "Canberra Deep Space Communications Complex"];
lat = [40.43139, -35.40139];
lon = [-4.24806, 148.98167];
gs = groundStation(sc,"Name",name,"Latitude",lat, ...
    "Longitude", lon);
```

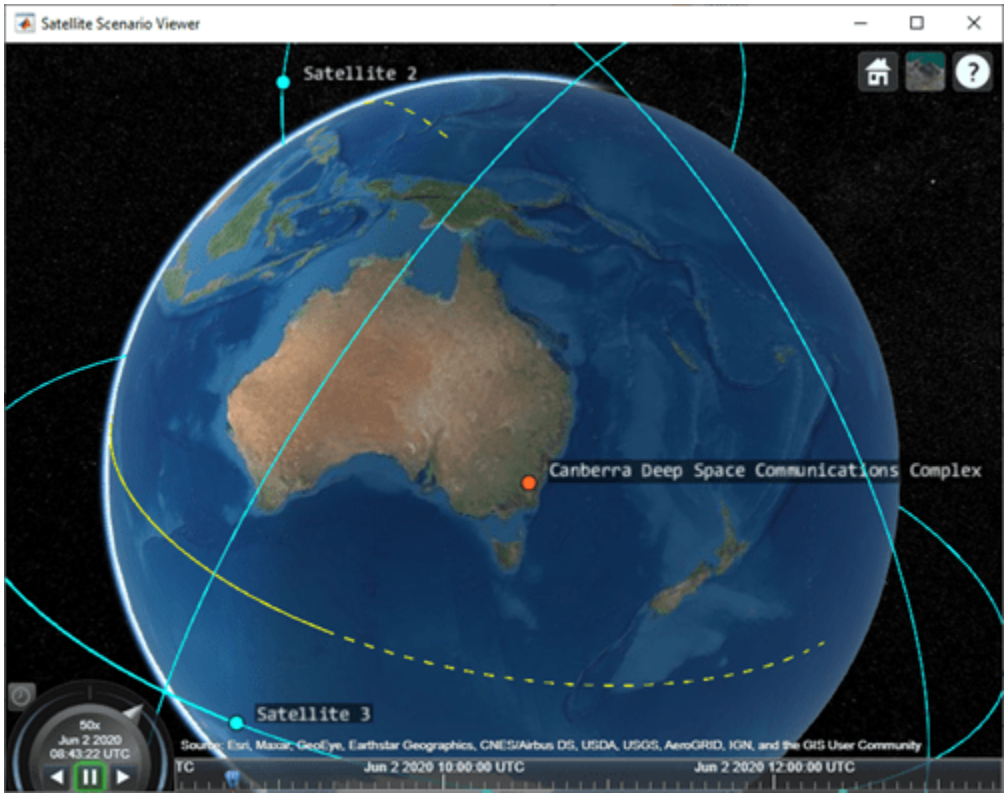
Return Azimuth Angle, Elevation Angle, and Range at Specified Time

Return the azimuth angle, elevation angle and range of the first satellite with respect to the Madrid Deep Space Communications Complex at time 02-June-2020 12:30:00 PM UTC.

```
time = datetime(2020,6,02,12,30,0);
[az,elev,r] = aer(gs(1),sat(1),time)

az = 264.2457

elev = -34.0669

r = 9.3088e+06
```

Play the satellite scenario with the satellites and ground stations.

```
play(sc)
```



See Also

Objects
`satellite` | `satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `play`

Related Examples

- “Satellite Scenario Key Concepts” on page 2-65
- “Satellite Scenario Overview” on page 2-74

Flight Control Analysis Tools

To help you visualize handling and flight control results, Aerospace Toolbox provides functions for:

- Short-period undamped natural frequency responses — `shortPeriodCategoryAPlot`, `shortPeriodCategoryBPlot`, and `shortPeriodCategoryCPlot`
- Boundary lines — `boundaryline`
- Altitude contours — `altitudeEnvelopeContour`

For an example of how to plot the results of the 3DOF airframe in the Sky Hogg model from the Aerospace Blockset, see “Plot Short-Period Undamped Natural Frequency Results” on page 2-80. This example describes how to use the Aerospace Blockset `asbFlightControlAnalysis` function to guide you through computing longitudinal and lateral-directional flying qualities. From the results, you can extract the short-period undamped natural frequency response for plotting using the `shortPeriodCategoryAPlot` function.

Plot Short-Period Undamped Natural Frequency Results

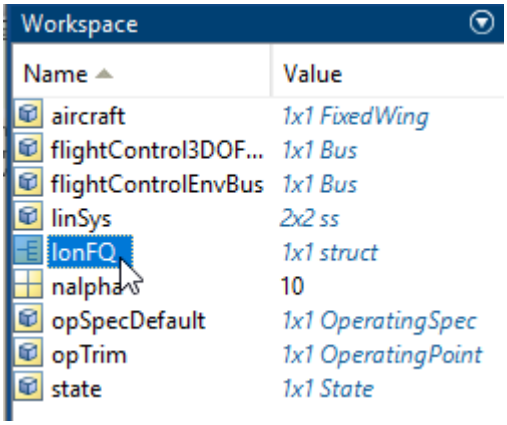
Aerospace Blockset flight analysis tools generate many variables that you can explore. For example, the analysis of a 3DOF or 6DOF airframe generates the short-period undamped natural frequency response ω_{nSP}. This example describes how to compute lateral-directional handling qualities and plot the category A flight phase for the short-period undamped natural frequency response ω_{nSP} using one of the `shortperiod` functions.

Note This topic requires an Aerospace Blockset license.

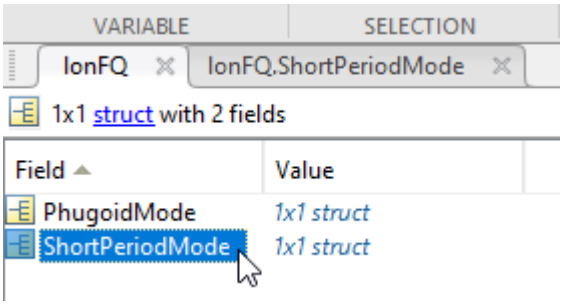
1. Start the flight control analysis template for the 3DOF configuration.

   ```
   asbFlightControlAnalysis('3DOF')
   ```

   The `3DOF Sky Hogg Longitudinal Flying Quality Analysis` project starts in the Simulink Editor.
2. To compute longitudinal and lateral-directional flying qualities, in the **Analysis Workflow** section, click through the guided workflow, click **OK** when prompted.
3. After computing longitudinal and lateral-directional flying qualities, find and double-click the `lonFQ` structure in your workspace.



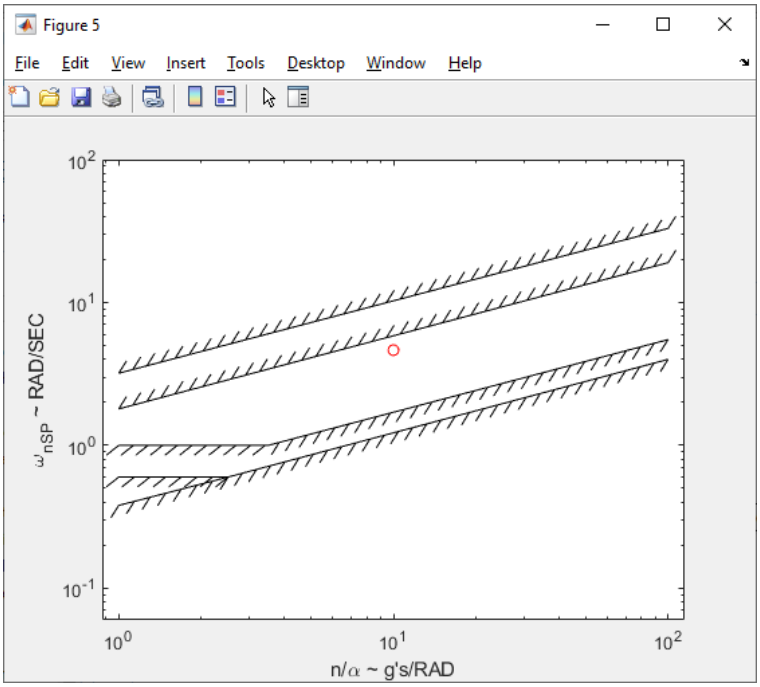
In the variables viewer, double-click the `ShortPeriodMode` variable.



4. Check that the `wn` variable exists. The `wn` variable is the short-period undamped natural frequency response you want to plot.
5. Plot the short-period undamped natural frequency response. In the MATLAB Command Window, use the `shortPeriodCategoryAPlot` function. For example, for a load factor per angle of attack of `10`, enter this command.

```
shortPeriodCategoryAPlot(10, lonFQ.ShortPeriodMode.wn, 'ro')
```

A figure window with the plotted short-period undamped natural frequency response displays.



6. To evaluate if the results are within your tolerance limits, check that the red dot is within your limits.

See Also

`altitudeEnvelopeContour` | `boundaryline` | `shortPeriodCategoryAPlot` | `shortPeriodCategoryBPlot` | `shortPeriodCategoryCPlot`

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.

See Also

`aeroDataPackage`

Functions

access

Package: `matlabshared.satellitescenario`

Add access analysis objects to satellite scenario

Syntax

```
access(asset1,asset2,...)
ac = access( ___ ,'Viewer',Viewer)
ac = access( ___ )
```

Description

`access(asset1,asset2,...)` adds `Access` analysis objects defined by nodes `asset1`, `asset2`, and so on.

`ac = access( ___ ,'Viewer',Viewer)` sets the viewer in addition to any input argument combination from previous syntaxes. For example, `'Viewer',v1` picks the viewer `v1`.

`ac = access( ___ )` returns added access analysis objects in the row vector `ac`.

Examples

Add Ground Stations to Scenario and Visualize Access Intervals

Create a satellite scenario and add ground stations from latitudes and longitudes.

```
startTime = datetime(2020,5,1,11,36,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
lat = 10;
lon = -30;
gs = groundStation(sc,lat,lon);
```

Add satellites using Keplerian elements.

```
semiMajorAxis = 10000000;
eccentricity = 0;
inclination = 10;
rightAscensionOfAscendingNode = 0;
argumentOfPeriapsis = 0;
trueAnomaly = 0;
sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
        rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly);
```

Add access analysis to the scenario and obtain the table of intervals of access between the satellite and the ground station.

```
ac = access(sat,gs);
intvls = accessIntervals(ac)
```

```
intvls=8×8 table
       Source              Target          IntervalNumber        StartTime                EndTi
    _____________    ___________________    ______________    ____________________    ____________

    "Satellite 2"    "Ground station 1"          1           01-May-2020 11:36:00    01-May-2020
    "Satellite 2"    "Ground station 1"          2           01-May-2020 14:20:00    01-May-2020
    "Satellite 2"    "Ground station 1"          3           01-May-2020 17:27:00    01-May-2020
    "Satellite 2"    "Ground station 1"          4           01-May-2020 20:34:00    01-May-2020
    "Satellite 2"    "Ground station 1"          5           01-May-2020 23:41:00    02-May-2020
    "Satellite 2"    "Ground station 1"          6           02-May-2020 02:50:00    02-May-2020
    "Satellite 2"    "Ground station 1"          7           02-May-2020 05:59:00    02-May-2020
    "Satellite 2"    "Ground station 1"          8           02-May-2020 09:06:00    02-May-2020
```

Play the scenario to visualize the ground stations.

```
play(sc)
```

Input Arguments

`asset1,asset2,...` — Satellite, ground station, or conical sensor
scalar | vector

`Satellite`, `GroundStation`, or `ConicalSensors` object, specified as a scalar or vector. These objects must belong to the same `satelliteScenario` object. The function adds the access analysis object to the `Accesses` property of the corresponding asset in `asset1`.

- If the asset in a given node is a scalar, every analysis object uses the same asset for that node position.
- If the asset in a given node is a vector, its length must equal the number of access analysis objects. Each access analysis object uses the corresponding element of the asset vector for that node location.

`Viewer` — Satellite scenario viewer
vector of `satelliteScenarioViewer` objects (default) | scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Satellite scenario viewer, specified as a scalar, vector, or array of `satelliteScenarioViewer` objects. If the `AutoSimulate` property of the scenario is `false`, adding a satellite to the scenario disables any previously available timeline and playback widgets.

Output Arguments

`ac` — Access analysis
scalar | vector

Access analysis between input objects, returned as either a scalar or vector. For more information about access analysis, see the `Access` object.

Data Types: `matlabshared.satellitescenario.Access`

Note When the `AutoSimulate` property is set to `false`, `SimulationStatus` must be `NotStarted` to call `access` function. Otherwise, use the `restart` function to reset the `SimulationStatus` to `NotStarted`. Note that `restart` also erases the simulation data.

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `play` | `hide` | `groundStation` | `conicalSensor`

Topics
“Satellite Scenario Key Concepts” on page 2-65

Access

Access analysis object belonging to scenario

Description

The Access object defines an access analysis object belonging to a `Satellite`, `GroundStation`, or `ConicalSensor` object. Use this object to determine if the line-of-sight (LOS) between two objects exists.

- If one of the objects belongs to `GroundStation`, the MinElevationAngle property of the `GroundStation` object limits the LOS visibility.
- If one of the objects belongs to `ConicalSensor`, the MaxViewAngle property of the `ConicalSensor` object limits the LOS visibility.

Creation

You can create an `Access` object using the `access` object function of `GroundStation` or `Satellite`.

Properties

`Sequence` — IDs of satellites, ground stations, or conical sensors

vector of positive numbers

IDs of the satellites, ground stations, and conical sensors defining access analysis, specified as a vector of positive numbers.

`LineWidth` — Visual width of access analysis object

`1` (default) | scalar

Visual width of access analysis object in pixels, specified as a scalar in the range (0, 10].

The line width cannot be thinner than the width of a pixel. If you set the line width to a value that is less than the width of a pixel on your system, the line displays as one pixel wide.

`LineColor` — Color of analysis line

`[0.5 0 1]` (default) | RGB triplet | hexadecimal color code | color name | short name

Color of access analysis line, specified as an RGB triplet, hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case

sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	
`"none"`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

Object Functions

show	Show object in satellite scenario viewer
accessStatus	Status of access between first and last node defining access analysis
accessIntervals	Intervals during which access status is true
accessPercentage	Percentage of time when access exists between first and last node in access analysis
hide	Hide satellite scenario entity from viewer

Examples

Add Ground Stations to Scenario and Visualize Access Intervals

Create a satellite scenario and add ground stations from latitudes and longitudes.

```
startTime = datetime(2020,5,1,11,36,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
lat = 10;
lon = -30;
gs = groundStation(sc,lat,lon);
```

Add satellites using Keplerian elements.

```
semiMajorAxis = 10000000;
eccentricity = 0;
inclination = 10;
rightAscensionOfAscendingNode = 0;
argumentOfPeriapsis = 0;
trueAnomaly = 0;
sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
        rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly);
```

Add access analysis to the scenario and obtain the table of intervals of access between the satellite and the ground station.

```
ac = access(sat,gs);
intvls = accessIntervals(ac)
```

```
intvls=8×8 table
       Source             Target          IntervalNumber         StartTime              EndTi
    ______________    ___________________    ______________    ____________________    _____

    "Satellite 2"    "Ground station 1"           1           01-May-2020 11:36:00    01-May-2020
    "Satellite 2"    "Ground station 1"           2           01-May-2020 14:20:00    01-May-2020
    "Satellite 2"    "Ground station 1"           3           01-May-2020 17:27:00    01-May-2020
    "Satellite 2"    "Ground station 1"           4           01-May-2020 20:34:00    01-May-2020
    "Satellite 2"    "Ground station 1"           5           01-May-2020 23:41:00    02-May-2020
    "Satellite 2"    "Ground station 1"           6           02-May-2020 02:50:00    02-May-2020
    "Satellite 2"    "Ground station 1"           7           02-May-2020 05:59:00    02-May-2020
    "Satellite 2"    "Ground station 1"           8           02-May-2020 09:06:00    02-May-2020
```

Play the scenario to visualize the ground stations.

```
play(sc)
```

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `groundStation` | `conicalSensor` | `satellite`

Topics

“Satellite Scenario Key Concepts” on page 2-65

accessIntervals

Package: `matlabshared.satellitescenario`

Intervals during which access status is true

Syntax

```
acinterval = accessIntervals(ac)
```

Description

`acinterval = accessIntervals(ac)` returns a table of intervals during which the access status corresponding to each `access` object in the input vector, `ac`, is `true`.

Examples

Add Ground Stations to Scenario and Visualize Access Intervals

Create a satellite scenario and add ground stations from latitudes and longitudes.

```
startTime = datetime(2020,5,1,11,36,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
lat = 10;
lon = -30;
gs = groundStation(sc,lat,lon);
```

Add satellites using Keplerian elements.

```
semiMajorAxis = 10000000;
eccentricity = 0;
inclination = 10;
rightAscensionOfAscendingNode = 0;
argumentOfPeriapsis = 0;
trueAnomaly = 0;
sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
        rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly);
```

Add access analysis to the scenario and obtain the table of intervals of access between the satellite and the ground station.

```
ac = access(sat,gs);
intvls = accessIntervals(ac)
```

```
intvls=8×8 table
       Source              Target          IntervalNumber         StartTime                EndTi
    _____________    ___________________    ______________    ____________________    ___________

    "Satellite 2"    "Ground station 1"          1           01-May-2020 11:36:00    01-May-2020
    "Satellite 2"    "Ground station 1"          2           01-May-2020 14:20:00    01-May-2020
```

```
    "Satellite 2"    "Ground station 1"          3          01-May-2020 17:27:00    01-May-2020
    "Satellite 2"    "Ground station 1"          4          01-May-2020 20:34:00    01-May-2020
    "Satellite 2"    "Ground station 1"          5          01-May-2020 23:41:00    02-May-2020
    "Satellite 2"    "Ground station 1"          6          02-May-2020 02:50:00    02-May-2020
    "Satellite 2"    "Ground station 1"          7          02-May-2020 05:59:00    02-May-2020
    "Satellite 2"    "Ground station 1"          8          02-May-2020 09:06:00    02-May-2020
```

Play the scenario to visualize the ground stations.

```
play(sc)
```

Input Arguments

ac — Access analysis

row vector of `Access` objects

Access analysis, specified as a row vector of `Access` objects.

Outputs Arguments

acinterval — Intervals during which access is true

table

Intervals during which access is `true`, returned as a table.

Each row of the table denotes a specific interval. The columns of the table are named `Source`, `Target`, `IntervalNumber`, `StartTime`, `EndTime`, `Duration` (in seconds), `StartOrbit`, and

`EndOrbit`. `Source` and `Target` are the names of the first and last node, respectively, which define the access analysis.

- If `Source` is a satellite or an object that is directly or indirectly attached to a satellite, then `StartOrbit` and `EndOrbit` correspond to the satellite associated with `Source`.
- If `Target` is a satellite or an object that is directly or indirectly attached to a satellite, then `StartOrbit` and `EndOrbit` correspond to the satellite associated with `Target`. Otherwise, `StartOrbit` and `EndOrbit` are `NaN` because they are associated with ground stations.

Note When the `AutoSimulate` property of satellite scenario is `true`, the access intervals between `StartTime` and `StopTime` are returned. When the property is `false`, the access intervals between `StartTime` and `SimulationTime` are returned.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `groundStation` | `accessPercentage` | `accessStatus` | `access`

Topics

“Satellite Scenario Key Concepts” on page 2-65

accessPercentage

Package: `matlabshared.satellitescenario`

Percentage of time when access exists between first and last node in access analysis

Syntax

```
acpercent = accessPercentage(ac)
```

Description

`acpercent = accessPercentage(ac)` returns the percentage of time from start time to stop time of the satellite scenario when access exists between the first and last node of each `access` object in the input vector, `ac`.

Examples

Calculate Access Percentage Between Ground Station and Satellites

Create a satellite scenario object.

```
startTime = datetime(2020,5,1,11,36,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add a ground station to the scenario.

```
gs = groundStation(sc);
```

Add satellites to the scenario.

```
semiMajorAxis = [10000000 10000000];                % meters
eccentricity = [0 0];
inclination = [0 30];                               % degrees
rightAscensionOfAscendingNode = [0 0];              % degrees
argumentOfPeriapsis = [0 0];                        % degrees
trueAnomaly = [0 10];                               % degrees
sat = satellite(sc,semiMajorAxis,eccentricity, ...
      inclination,rightAscensionOfAscendingNode, ...
      argumentOfPeriapsis,trueAnomaly);
```

Add access analysis between the ground station and each satellite.

```
access(gs,sat(1));
access(gs,sat(2));
```

Obtain the access percentage between the ground station and each satellite.

```
ac = gs.Accesses;
acPercent = accessPercentage(ac)
```

```
acPercent = 2×1

   15.0000
   14.9306
```

Input Arguments

ac — Access analysis

row vector of `Access` objects

Access analysis, specified as a row vector of `Access` objects.

Outputs Arguments

acpercent — Access percentage

row vector of nonnegative numbers

Access percentage, returned as a row vector of nonnegative numbers.

Note When the `AutoSimulate` property of satellite scenario is `true`, the access percentage corresponds to the duration between `StartTime` and `StopTime`. When the property is `false`, the access percentage corresponds to the duration between `StartTime` and `SimulationTime`.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `groundStation` | `satellite` | `access` | `accessIntervals` | `accessStatus`

Topics

“Satellite Scenario Key Concepts” on page 2-65

accessStatus

Package: `matlabshared.satellitescenario`

Status of access between first and last node defining access analysis

Syntax

```
s = accessStatus(ac)
s = accessStatus(ac,timeIn)
[s,timeOut] = accessStatus( ___ )
```

Description

`s = accessStatus(ac)` returns a matrix `s` of the access status history between the first and last node corresponding to each `access` object in the input vector `ac`.

`s = accessStatus(ac,timeIn)` returns the status of each access analysis object at the specified datetime in `timeIn`. Each element of `s` corresponds to an access object in `ac`.

`[s,timeOut] = accessStatus( ___ )` returns the status of each access analysis object and the corresponding datetime in Coordinated Universal Time (UTC).

Examples

Obtain Access Status Between Satellite and Ground Station

Create a satellite scenario object.

```
startTime = datetime(2021,4,30);                        % 30 April 2021, 12:00 AM UTC
stopTime = datetime(2021,5,1);                          % 1 May 2021, 12:00 AM UTC
sampleTime = 60;                                        % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add a satellite to the scenario.

```
semiMajorAxis = 10000000;                               % meters
eccentricity = 0;
inclination = 10;                                       % degrees
rightAscensionOfAscendingNode = 0;                      % degrees
argumentOfPeriapsis = 0;                                % degrees
trueAnomaly = 0;                                        % degrees
sat = satellite(sc,semiMajorAxis,eccentricity, ...
    inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly);
```

Add a ground station to the scenario.

```
gs = groundStation(sc);
```

Add access analysis between the satellite and the ground station.

```
ac = access(sat,gs);
```

Obtain the access status at 30 April 2021, 5:34 PM UTC.

```
time = datetime(2021,4,30,17,34,0);
s = accessStatus(ac,time)
```

```
s = logical
   0
```

Input Arguments

`ac` — Access analysis

row vector of `Access` objects

Access analysis, specified as a row vector of `Access` objects.

`timeIn` — Time at which output is calculated

`datetime` scalar

Time at which the output is calculated, specified as a `datetime` scalar. If no time zone is specified in `timeIn`, the time zone is assumed to be UTC.

Outputs Arguments

`s` — Access analysis status

column vector | matrix

Access analysis status, returned as a column vector or a matrix. If `timeIn` is specified, `s` is a column vector. Otherwise, the output is a matrix. The rows of the matrix correspond to the access object in `ac`, and the columns correspond to the time sample. The status at a given instant is `1` (`true`) if access exists between each pair of adjacent nodes defined by `Sequence`. For example, in a given pair, defined by node 1 and node 2, node 1 has access to node 2 and node 2 has access to node 1:

- If a node is a satellite, then the satellite has access to the adjacent node when both nodes are in line of sight of each other.
- If a node is a ground station, then the ground station has access to the adjacent node when the elevation angle of the node with respect to the ground station is greater than or equal to the `MinElevationAngle` property of `GroundStation`.
- If a node is a conical sensor, then the conical sensor has access to the adjacent node when the latter is in the field of view of the former. If the conical sensor is attached to a ground station directly or via a gimbal, then the elevation angle of the adjacent node with respect to the ground station must be greater than or equal to the `MinElevationAngle` property of `GroundStation`.

`timeOut` — Time samples of output access status

scalar | vector

Time samples of the output access status, returned as a scalar or vector. If the time history of the access status is returned, `timeOut` is a row vector.

Note When the `AutoSimulate` property of the satellite scenario is `true`, the access status history from `StartTime` to `StopTime` is returned. When the property is `false`, the access status history from `StartTime` to `SimulationTime` is returned.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `groundStation` | `conicalSensor`

Topics

“Satellite Scenario Key Concepts” on page 2-65

aer

Package: `matlabshared.satellitescenario`

Calculate azimuth angle, elevation angle, and range of another satellite or ground station in NED frame

Syntax

```
az = aer(asset,target)
[az,el] = aer(asset,target)
[az,el,range] = aer(asset,target)
[az,el,range,timeOut] = aer(asset,target)
[___] = aer(asset,target,timeIn)
[___] = aer(___,coordinateFrame='ned')
```

Description

`az = aer(asset,target)` returns a 2-D array of the history of azimuth angles `az`, between `asset` and `target` belonging to a given `satelliteScenario` object.

`[az,el] = aer(asset,target)` returns the history of elevation angles, `el`, between satellite or ground station `asset` and another satellite or ground station `target`.

`[az,el,range] = aer(asset,target)` returns row vectors of the history of the `range` of `Satellite` or `GroundStation` in `target` with respect to those in `asset`.

`[az,el,range,timeOut] = aer(asset,target)` returns the corresponding time in `timeOut`.

`[___] = aer(asset,target,timeIn)` returns the outputs at the specified datetime `timeIn`. `az`, `el`, and `range` are structured the same way as described in syntaxes with an exception that the size of the second dimension is fixed at 1, representing the values at the specified time `timeIn`.

`[___] = aer(___,coordinateFrame='ned')` returns the `az`, `el`, `range`, and `timeOut` based on the specified output arguments and the coordinate frame defined by the name-value argument.

Note By default, the `aer` function uses the north-east-down (NED) system to perform all the calculations.

Examples

Determine AER of Ground Station

Create a satellite scenario object.

```
startTime = datetime(2021,4,25);                    % April 25, 2021, 12:00 AM UTC
stopTime = datetime(2021,4,26);                     % April 26, 2021, 12:00 AM UTC
sampleTime = 60;                                    % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add a satellite to the scenario.

```
tleFile = "eccentricOrbitSatellite.tle";
sat = satellite(sc,tleFile);
```

Add a ground station to the scenario using default properties.

```
gs = groundStation(sc);
```

Determine the azimuth angle, elevation angle, and range of the ground station with respect to the satellite at April 25, 2021, 1:26 AM UTC.

```
time = datetime(2021,4,25,1,26,0);
[azimuth,elevation,range] = aer(sat,gs,time)
```

```
azimuth = 15.2962

elevation = -70.3858

range = 1.3442e+07
```

Input Arguments

`asset` — First scenario component

scalar | vector

First scenario component, specified as a `Satellite`, `GroundStation`, `ConicalSensor`, or a `Gimbal` object.

`target` — Second scenario component

scalar | vector

Second scenario component, specified as a `Satellite`, `GroundStation`, `ConicalSensor`, or a `Gimbal` object.

`timeIn` — Time at which output is calculated

datetime

Time at which output is calculated, specified as a datetime. If no time zone is specified in `timeIn`, the time zone is assumed to be UTC.

`coordinateFrame` — Coordinate frame

`'ned'` (default) | `'body'`

Coordinate frame, specified as either `'ned'` or `'body'`.

- When `coordinateFrame` is `'ned'` — The azimuth angle is defined in the North-East-Down (NED) frame of (and centered at) `asset` such that 0 degrees is North, 90 degrees is East, 180 degrees is South, and 270 degrees is West. The elevation angle is defined in the NED frame of (and centered at) `asset` such that 0 degrees implies `target` is on the North East (NE) plane, 90 degrees implies `target` is directly above `asset`, and -90 degrees implies `target` is directly below `asset`.
- When `coordinateFrame` is `'body'` — The azimuth angle is the angle between the projection of the relative position vector of `target` on the *x*-*y* plane of the body frame of `asset`, and the *x*-axis of `asset`. The angle is positive for positive (clockwise) rotation about the *z*-axis of `asset`. The elevation angle is the angle between the relative position vector of `target` on the *x*-*y* plane of the

body frame of `asset`. The angle is positive when the *z* component of the relative position of `target` defined in the body frame of `asset` is negative.

Output Arguments

`az` — Azimuth angles

vector | 2-D array | scalar

Azimuth angles of the target in the local azimuth, elevation, and range (AER) system in degrees, returned as a vector, 2-D array, or scalar in the range [0,360). Azimuths are measured clockwise from North. If the `timeIn` argument is not specified, the vector elements correspond to the time samples specified by the `SampleTime` property from the satellite scenario `StartTime` to `StopTime`.

- If both `asset` and `target` are scalars, `az` is a row vector where each element represents the azimuth angle of `target` with respect to `asset` in the NED frame of `asset` at a specified time sample.
- If `asset` is a scalar and `target` is a vector, `az` is a 2-D array, where each row represents the azimuth angle of each element in `target` with respect to `asset` in the NED frame of `asset` and the columns represent the time samples.
- If `asset` is a vector and `target` is a scalar, `az` is a 2-D array, where each row represents the azimuth angle of `target` with respect to each element in `asset` in the NED frame of the element in `asset` and the columns represent the time samples.
- If both `asset` and `target` are vectors, the length of `asset` must equal the length of `target`. The `az` is a 2-D array, where each row index corresponds to the index in `asset` and `target`, and represents the azimuth angle of the element at the index in `target` with respect to the element at the index in `asset` in the NED frame of that element in `asset`. The columns represent the time samples.

If the `timeIn` argument is not specified and when the `AutoSimulate` property of the satellite scenario is `true`, aer function returns the `az` history from `StartTime` to `StopTime`. Otherwise, it returns the `az` history from `StartTime` to `SimulationTime`.

`el` — Elevation angles

vector | 2-D array | scalar

Elevation angles of target in the local AER system in degrees, returned as a vector, 2-D array, or scalar in the range [-90 90]. Elevations are measured with respect to a plane that is perpendicular to the normal of the surface of the earth. If `asset` is on the surface of the Earth, then the plane is tangential to the Earth. If the `timeIn` argument is not specified, the vector elements correspond to the time samples specified by the `SampleTime` property from the satellite scenario `StartTime` to `StopTime`.

If the `timeIn` argument is not specified and when the `AutoSimulate` property of the satellite scenario is `true`, aer function returns the `el` history from `StartTime` to `StopTime`. Otherwise, it returns the `el` history from `StartTime` to `SimulationTime`.

`range` — Distances from local origin

vector | 2-D array | scalar

Distances from the local origin in meters, returned as a vector, 2-D array, or a scalar. The `range` array is structured the same way as the `az` and `el`, described in the above syntaxes.

If the `timeIn` argument is not specified and when the `AutoSimulate` property of the satellite scenario is `true`, aer function returns the `range` history from `StartTime` to `StopTime`. Otherwise, it returns the `range` history from `StartTime` to `SimulationTime`.

`timeOut` — Time samples between start and stop time of scenario
row vector | scalar

Time samples corresponding to `az`, `el`, and `range` in UTC, returned as a row vector, or a scalar.

If the `timeIn` argument is not specified and when the `AutoSimulate` property of the satellite scenario is `true`, aer function returns the time sample history from `StartTime` to `StopTime`. Otherwise, it returns the time sample history from `StartTime` to `SimulationTime`.

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `play` | `access` | `groundStation` | `conicalSensor` | `hide`

Topics
“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

addBody

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

Add body to animation object

Syntax

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

Description

`idx = addBody(h,b)` adds a loaded body, `b`, to the animation object, `h`, generates patches for the body, and returns the index of the body in `idx`.

Input Arguments

`h` — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

`b` — Loaded animation body

scalar

Loaded animation body, specified as a scalar.

Output Arguments

`idx` — Body index

scalar

Body index, returned as a scalar.

Examples

Add Second Body

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

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

```
idx2 = 2
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

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.

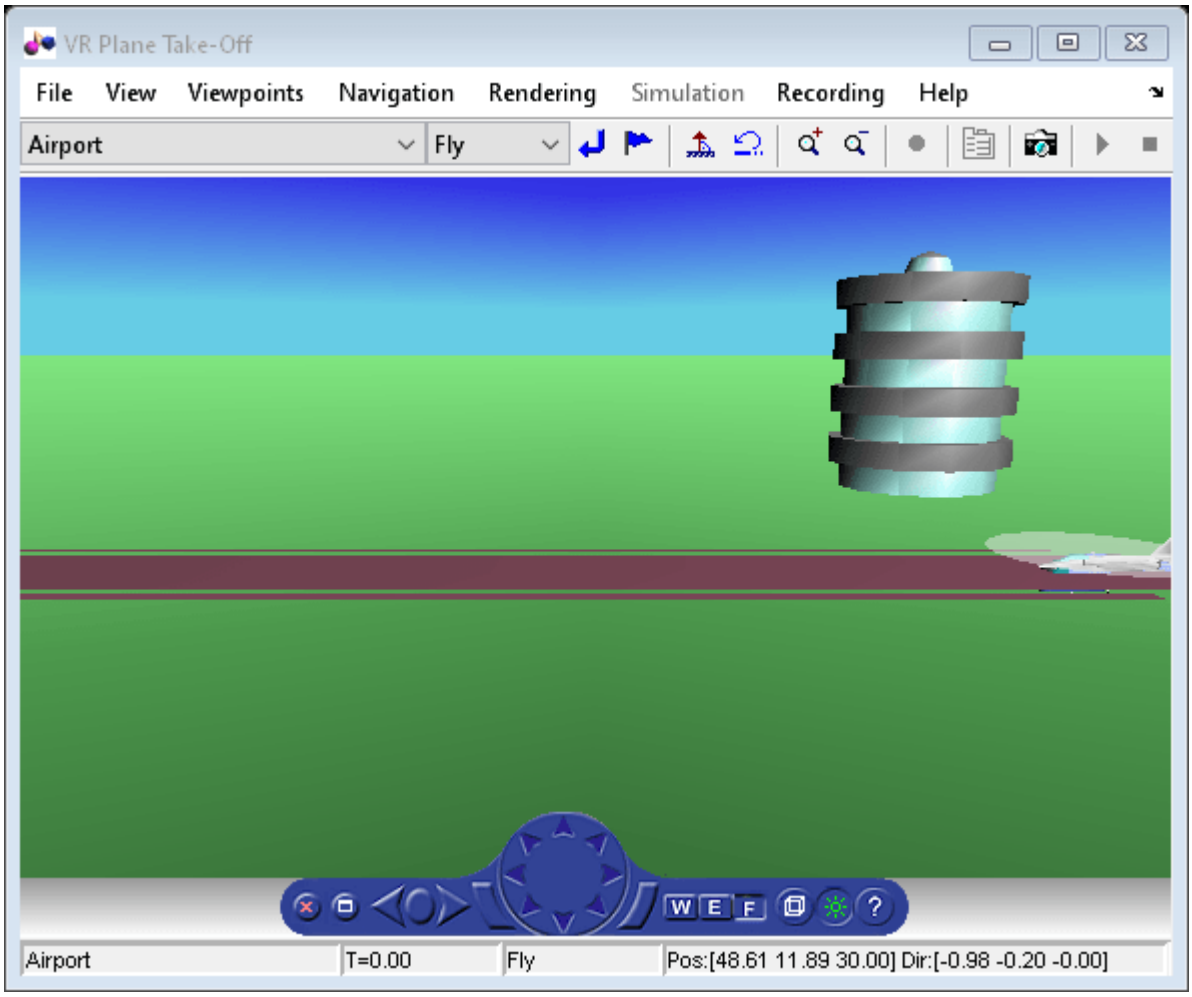
When you use the `addNode` method to add a node, all the objects in the `.wrl` file will be added to the virtual reality animation object under one node. If you want to add separate nodes for the objects in the `.wrl` file, place each node in a separate `.wrl` file.

Examples

Add Node to World

This example shows how to add a node to the world defined in `chaseHelicopter.wrl`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename ='asttkoff.wrl';
h.initialize();
h.addNode('Lynx','chaseHelicopter.wrl');
```



Version History

Introduced in R2007b

See Also

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

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

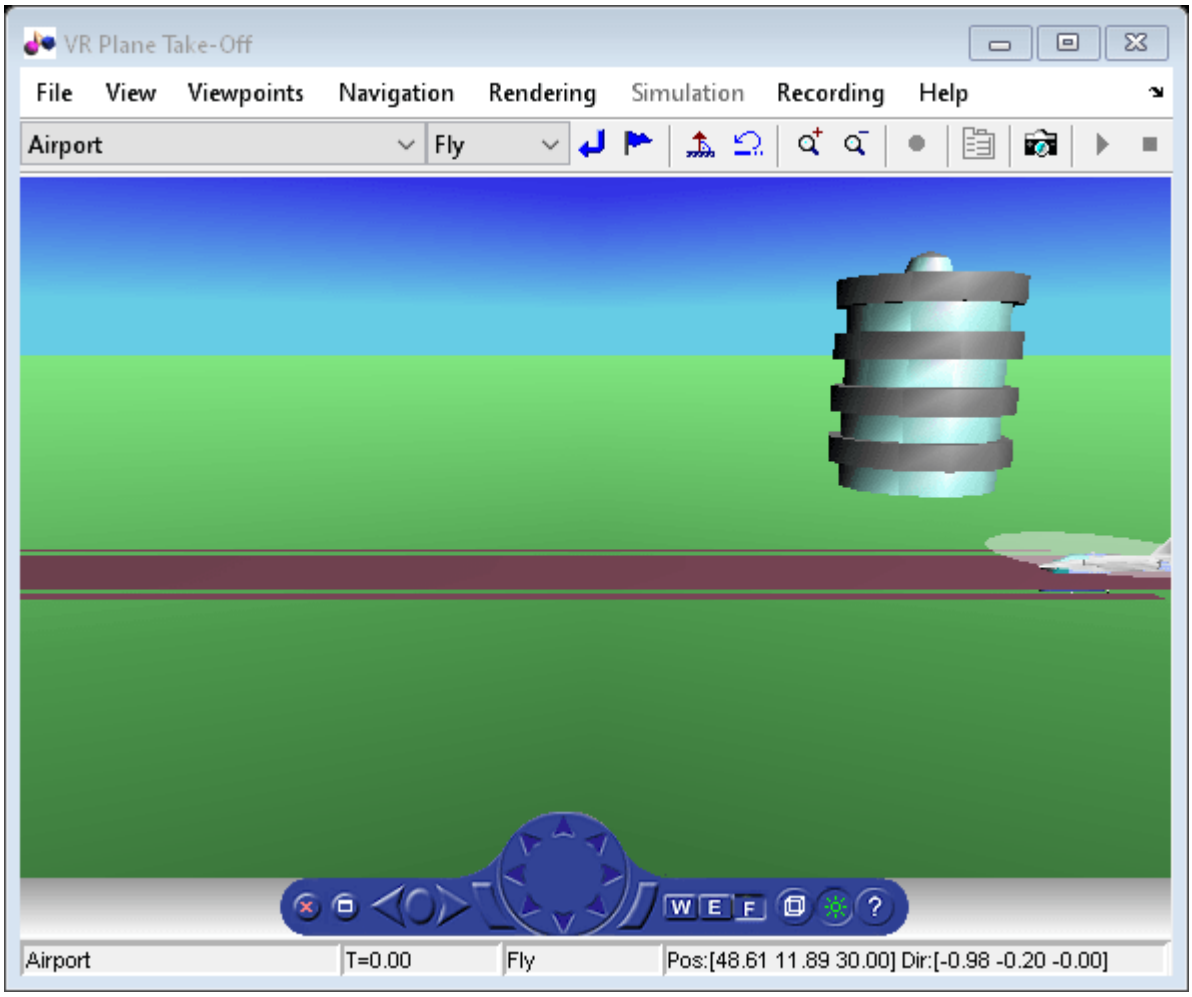
`addRoute(h, nodeOut, eventOut, nodeIn, eventIn)` and `h.addRoute(nodeOut, eventOut, nodeIn, eventIn)` add a VRML ROUTE statement to the virtual reality animation, where `nodeOut` is the node from which information is routed, `eventOut` is the event (property), `nodeIn` is the node to which information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

Examples

Add ROUTE Command to Connect Plane

This example shows how to add a `ROUTE` command to connect the `Plane` position to the `Camera1` node.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
h.addNode('Lynx','chaseHelicopter.wrl');
h.addRoute('Plane','translation','Camera1','translation');
```



Version History

Introduced in R2007b

See Also

`Aero.VirtualRealityAnimation` | `addViewpoint`

advance

Move simulation forward by one sample time

Syntax

```
isrunning = advance(sc)
```

Description

`isrunning = advance(sc)` moves the simulation forward by the amount of time specified by the `SampleTime` property of the scenario `sc`.

Examples

Manual Simulation of Satellite Scenario

Create a satellite scenario object and set the `AutoSimulate` property to `false` to enable manual simulation of the satellite scenario.

```
startTime = datetime(2022,1,12);
stopTime = startTime + days(0.5);
sampleTime = 60;                                % Seconds
sc = satelliteScenario('AutoSimulate',false);
```

Add a GPS satellite constellation to the scenario.

```
sat = satellite(sc,"gpsAlmanac.txt");
```

Simulate the scenario using the `advance` function.

```
while advance(sc)
end
```

Obtain the satellite position histories.

```
p = states(sat);
```

`AutoSimulate` is `false`, so restart the scenario before adding a ground station.

```
restart(sc);
```

Add a ground station to the scenario.

```
gs = groundStation(sc);
```

Add access analysis between each satellite and ground station.

```
ac = access(sat,gs);
```

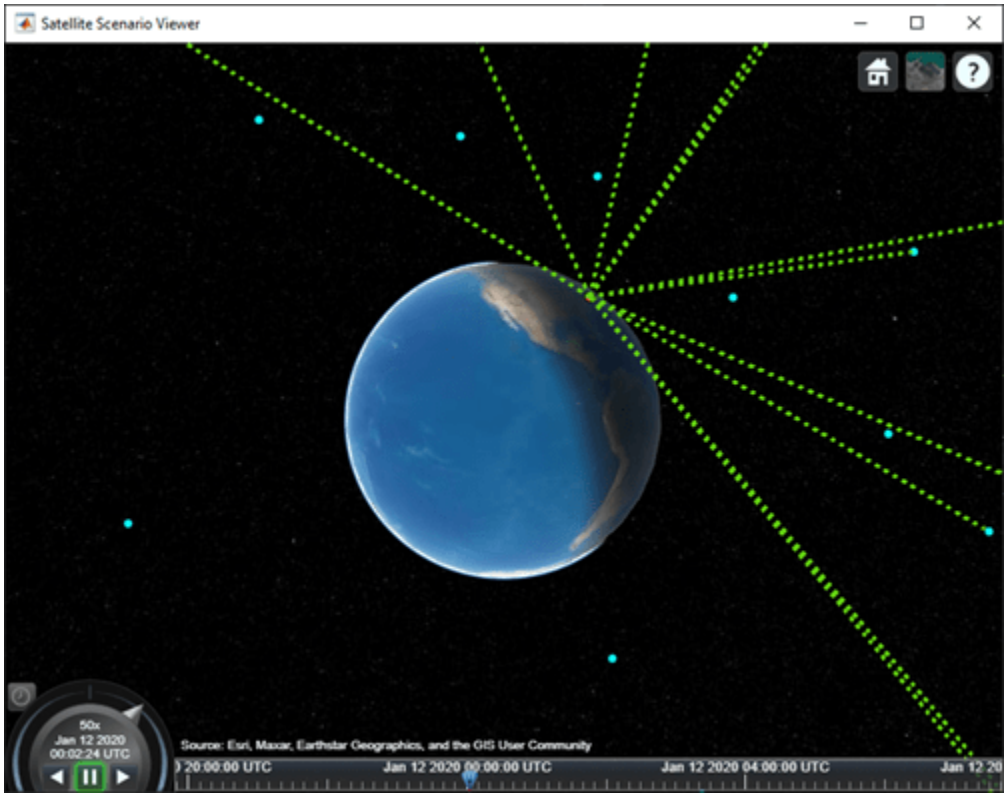
Simulate the scenario and determine the access intervals.

```
while advance(sc)
end
intvls1 = accessIntervals(ac)
```

```
intvls1=35×8 table
     Source           Target           IntervalNumber         StartTime              EndTime
    ________    ____________________    ______________    ____________________    ____________

    "PRN:1"     "Ground station 32"           1          11-Jan-2020 23:20:25    12-Jan-2020 05:1
    "PRN:2"     "Ground station 32"           1          12-Jan-2020 04:03:16    12-Jan-2020 07:4
    "PRN:3"     "Ground station 32"           1          11-Jan-2020 19:50:06    11-Jan-2020 21:3
    "PRN:3"     "Ground station 32"           2          12-Jan-2020 01:52:43    12-Jan-2020 06:4
    "PRN:4"     "Ground station 32"           1          11-Jan-2020 19:50:06    12-Jan-2020 00:1
    "PRN:4"     "Ground station 32"           2          12-Jan-2020 04:54:02    12-Jan-2020 07:4
    "PRN:5"     "Ground station 32"           1          12-Jan-2020 05:52:03    12-Jan-2020 07:4
    "PRN:6"     "Ground station 32"           1          12-Jan-2020 02:43:29    12-Jan-2020 07:4
    "PRN:7"     "Ground station 32"           1          11-Jan-2020 21:09:52    12-Jan-2020 03:2
    "PRN:8"     "Ground station 32"           1          11-Jan-2020 20:33:36    12-Jan-2020 03:1
    "PRN:9"     "Ground station 32"           1          11-Jan-2020 19:50:06    12-Jan-2020 00:4
    "PRN:9"     "Ground station 32"           2          12-Jan-2020 05:08:32    12-Jan-2020 07:4
    "PRN:10"    "Ground station 32"           1          12-Jan-2020 00:32:56    12-Jan-2020 01:5
    "PRN:11"    "Ground station 32"           1          11-Jan-2020 22:15:09    12-Jan-2020 04:3
    "PRN:12"    "Ground station 32"           1          12-Jan-2020 04:32:16    12-Jan-2020 07:4
    "PRN:13"    "Ground station 32"           1          12-Jan-2020 00:03:56    12-Jan-2020 02:5
      ⋮
```

Visualize the simulation results.

```
v = satelliteScenarioViewer(sc,'ShowDetails',false);
play(sc);
```



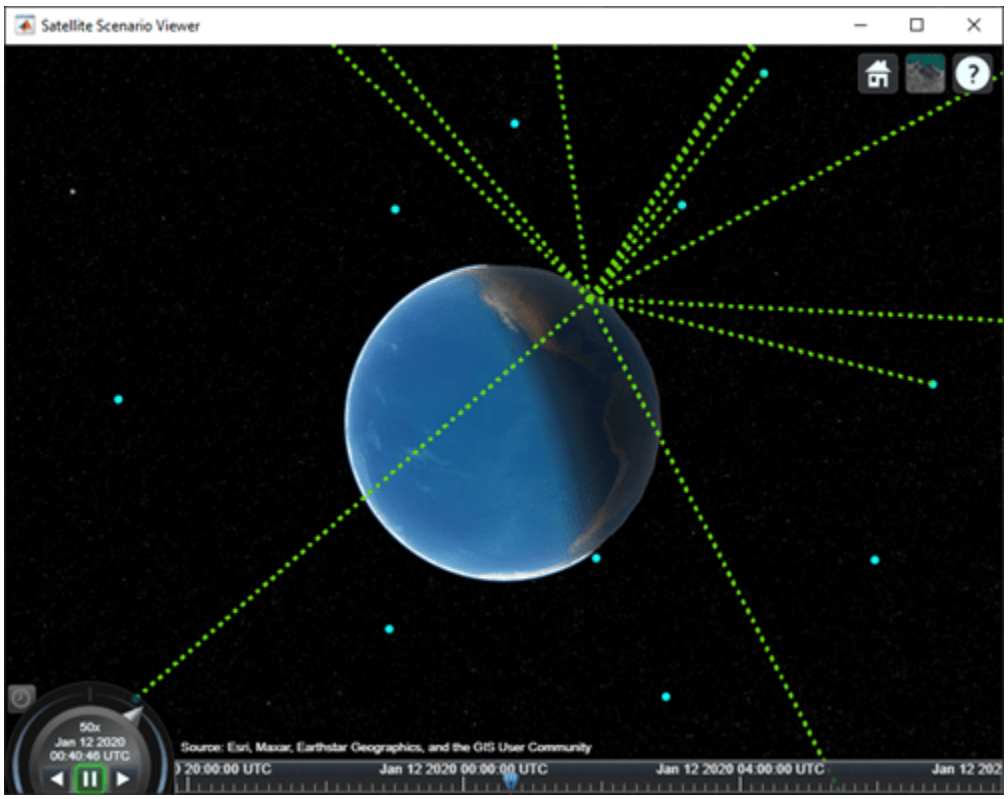
Verify that the access intervals are the same when you set the `AutoSimulate` property to `true`.

```
sc.AutoSimulate = true;
intvls2 = accessIntervals(ac)
```

```
intvls2=35×8 table
     Source          Target           IntervalNumber        StartTime              EndTime
    ________    ____________________    ______________    ____________________    ____________

    "PRN:1"     "Ground station 32"          1           11-Jan-2020 23:20:25    12-Jan-2020 05:
    "PRN:2"     "Ground station 32"          1           12-Jan-2020 04:03:16    12-Jan-2020 07:
    "PRN:3"     "Ground station 32"          1           11-Jan-2020 19:50:06    11-Jan-2020 21:
    "PRN:3"     "Ground station 32"          2           12-Jan-2020 01:52:43    12-Jan-2020 06:
    "PRN:4"     "Ground station 32"          1           11-Jan-2020 19:50:06    12-Jan-2020 00:
    "PRN:4"     "Ground station 32"          2           12-Jan-2020 04:54:02    12-Jan-2020 07:
    "PRN:5"     "Ground station 32"          1           12-Jan-2020 05:52:03    12-Jan-2020 07:
    "PRN:6"     "Ground station 32"          1           12-Jan-2020 02:43:29    12-Jan-2020 07:
    "PRN:7"     "Ground station 32"          1           11-Jan-2020 21:09:52    12-Jan-2020 03:
    "PRN:8"     "Ground station 32"          1           11-Jan-2020 20:33:36    12-Jan-2020 03:
    "PRN:9"     "Ground station 32"          1           11-Jan-2020 19:50:06    12-Jan-2020 00:
    "PRN:9"     "Ground station 32"          2           12-Jan-2020 05:08:32    12-Jan-2020 07:
    "PRN:10"    "Ground station 32"          1           12-Jan-2020 00:32:56    12-Jan-2020 01:
    "PRN:11"    "Ground station 32"          1           11-Jan-2020 22:15:09    12-Jan-2020 04:
    "PRN:12"    "Ground station 32"          1           12-Jan-2020 04:32:16    12-Jan-2020 07:
    "PRN:13"    "Ground station 32"          1           12-Jan-2020 00:03:56    12-Jan-2020 02:
      ⋮
```

Visualize the scenario.

```
play(sc);
```



Input Arguments

`sc` — Satellite scenario

`satelliteScenario` object

Satellite scenario, specified as a `satelliteScenario` object. The argument applies only if the `AutoSimulate` property of the `sc` object is `false`.

Output Arguments

`isrunning` — Running status of satellite scenario simulation

`true` or 1 | `false` or 0

Running status of the satellite scenario simulation, returned as a logical 1 (`true`) or 0 (`false`). The `isrunning` value is `true` until the scenario reaches the specified `StopTime` value.

Version History

Introduced in R2022a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`play` | `satellite` | `groundStation` | `restart`

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).

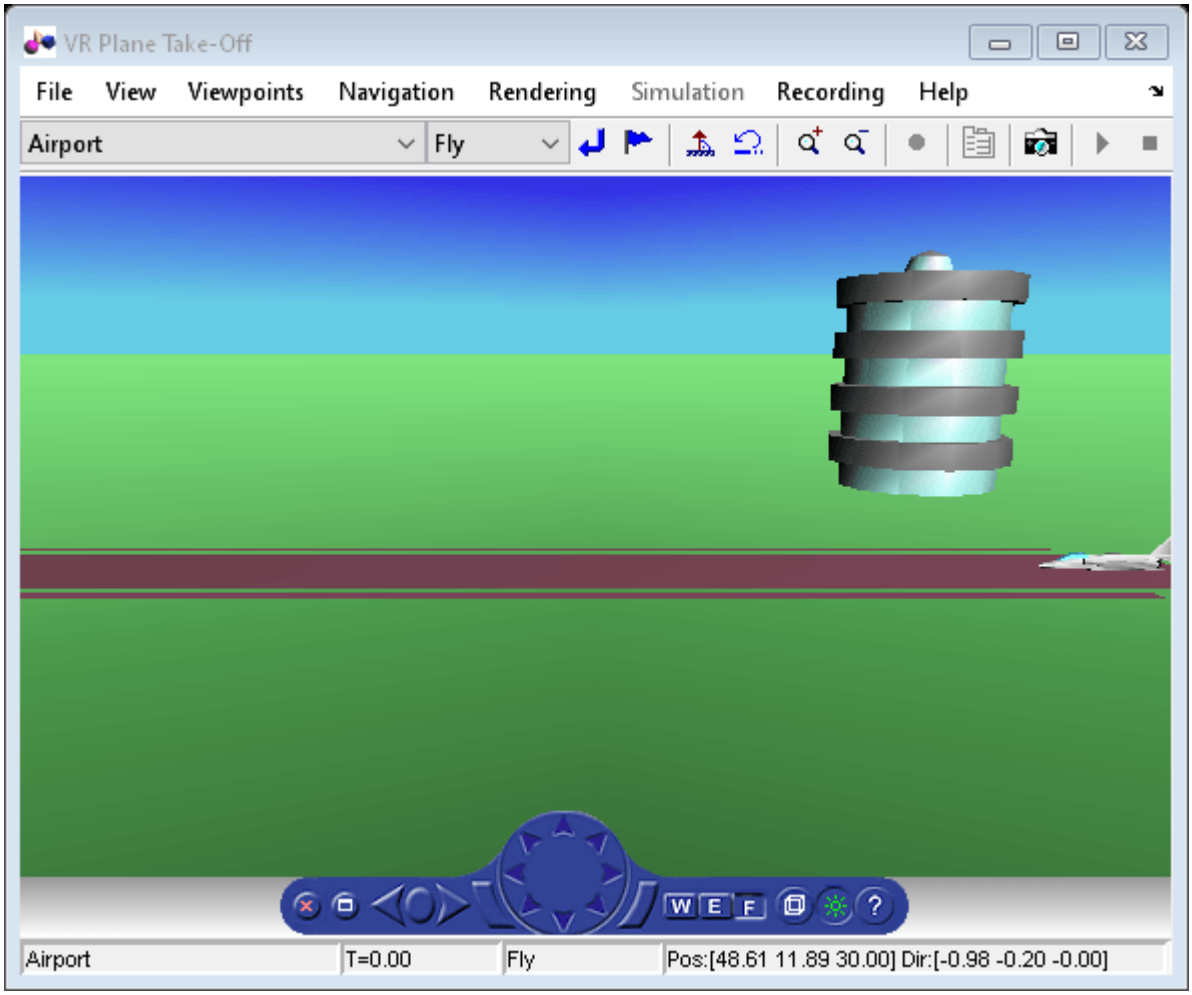
Note If you call `addViewpoint` with only the description argument, you must set the position and orientation of the viewpoint with the Simulink 3D Animation `vrnode/setfield` function. This requires you to use VRML coordinates.

Examples

Add a viewpoint

This example shows how to add a viewpoint named `chaseView`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
h.addViewpoint(h.Nodes{2}.VRNode,'children','chaseView','View From Helicopter');
```



Version History

Introduced in R2007b

See Also

`addRoute` | `removeViewpoint`

aeroReadSpaceWeatherData

Output MAT-file of space weather data

Syntax

```
matfile = aeroReadSpaceWeatherData(spaceWeatherFile)
matfile = aeroReadSpaceWeatherData(spaceWeatherFile,Name=Value)
[matfile,startdate] = aeroReadSpaceWeatherData( ___ )
[matfile,startdate,enddate] = aeroReadSpaceWeatherData( ___ )
```

Description

`matfile = aeroReadSpaceWeatherData(spaceWeatherFile)` reads space weather data from the space weather data file, `spaceWeatherFile`, downloaded from CelesTrak® and stores the data in a MAT-file, `matfile`. The function saves the MAT-file to the current working directory, and returns the filename along with its full path in `matfile`.

The `fluxSolarAndGeomagnetic` function extracts the solar flux and geomagnetic data from the MAT-file.

`matfile = aeroReadSpaceWeatherData(spaceWeatherFile,Name=Value)` creates `matfile` based on additional parameters specified by one or more name-value arguments.

`[matfile,startdate] = aeroReadSpaceWeatherData( ___ )` returns the earliest date `startdate` that corresponds to the data that is saved in `matfile`. Specify input argument combinations from the previous syntaxes.

`[matfile,startdate,enddate] = aeroReadSpaceWeatherData( ___ )` returns the earliest and latest dates, `startdate` and `enddate`, the correspond to the data that is saved in `matfile`.

Input Arguments

`spaceWeatherFile` — Space weather data file

filename | character vector | string

Space weather data file, a comma-separated values (CSV) file containing consolidated space weather data from `https://celestrak.org/spacedata/`, specified as a character vector or string. The extension `.csv` must be appended to the file name. The filename must be on the MATLAB path or you must specify the path with the filename.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Format='numeric'` specifies that the space weather data is stored as one-dimensional cell arrays (`F107_DATA_TYPE`) and numeric arrays (all other data in `matfile`) .

`StartDate` — Earliest date to read data
earliest date in `spaceWeatherFile` (default) | `datetime` object | scalar

Earliest date to read data from `spaceWeatherFile`, specified as a `datetime` object or scalar. The default value is the earliest date in `spaceWeatherFile`. `StartDate` cannot occur after `EndDate`.

`EndDate` — Latest date to read data
latest date in `spaceWeatherFile` (default) | `datetime` object | scalar

Latest date to read data from `spaceWeatherFile`, specified as a `datetime` object or scalar. The default value is the latest date in `spaceWeatherFile`. `EndDate` cannot occur before `EndDate`.

`FolderName` — Folder in which to save `matfile`
current folder with write permissions (default) | character vector | string scalar

Folder in which to save `matfile`, specified as a character vector or string scalar. The default folder is the current folder with write permissions. Create this folder with write permission first before specifying `FolderName`.

`Format` — Format in which space weather data is stored in MAT-file
`'numeric'` (default) | `'timetable'`

Format in which space weather data is stored in `matfile`, specified as one of these values:

- `'numeric'` — Data is stored as one-dimensional cell arrays (`F107_DATA_TYPE`) and numeric arrays (all other data in `matfile`). These variables are stored in `matfile`.

Space Weather Data File Variables	Description
YEAR	ISO 8601 year.
MONTH	ISO 8601 month.
DAY	ISO 8601 day.
BRSN	Bartels Solar Rotation Number. A sequence of 27-day intervals counted continuously from Feb 8, 1832.
ND	Number of day within the Bartels 27-day cycle (01-27).
KP1	Planetary 3-hour range index (Kp) for 0000-0300 coordinated universal time.
KP2	Planetary 3-hour range index (Kp) for 0300-0600 coordinated universal time.
KP3	Planetary 3-hour range index (Kp) for 0600-0900 coordinated universal time.
KP4	Planetary 3-hour range index (Kp) for 0900-1200 coordinated universal time.
KP5	Planetary 3-hour range index (Kp) for 1200-1500 coordinated universal time.

Space Weather Data File Variables	Description
KP6	Planetary 3-hour range index (Kp) for 1500-1800 coordinated universal time.
KP7	Planetary 3-hour range index (Kp) for 1800-2100 coordinated universal time.
KP8	Planetary 3-hour range index (Kp) for 2100-0000 coordinated universal time.
KP_SUM	Sum of the 8 planetary 3-hour range indices (Kp) for the day.
AP1	Planetary equivalent amplitude (Ap) for 0000-0300 coordinated universal time.
AP2	Planetary equivalent amplitude (Ap) for 0300-0600 coordinated universal time.
AP3	Planetary 3-hour range index (Kp) for 0600-0900 coordinated universal time.
AP4	Planetary equivalent amplitude (Ap) for 0900-1200 coordinated universal time.
AP5	Planetary equivalent amplitude (Ap) for 1200-1500 coordinated universal time.
AP6	Planetary equivalent amplitude (Ap) for 1500-1800 coordinated universal time.
AP7	Planetary equivalent amplitude (Ap) for 1800-2100 coordinated universal time.
AP8	Planetary equivalent amplitude (Ap) for 2100-0000 coordinated universal time.
AP_AVG	Arithmetic average of the 8 planetary equivalent amplitude indices (Ap) for the day.
CP	Cp or planetary daily character figure. A qualitative estimate of overall level of magnetic activity for the day determined from the sum of the 8 Ap indices. Cp ranges, in steps of one tenth, from 0 (quiet) to 2.5 (highly disturbed).
C9	A conversion of the 0 to 2.5 range of the Cp index to one digit between 0 and 9.
ISN	International sunspot number. Records contain the Zurich number through December 31, 1980, and the international Brussels number thereafter.

Space Weather Data File Variables	Description
F107_OBS	Observed 10.7-cm solar radio flux (F10.7). Measured in units of 10-22 W/m^2/Hz. at: • Ottawa, Canada at 1700 coordinated universal time daily from February 14, 1947, to May 31, 1991. • Penticton, British Columbia, Canada at 2000 coordinated universal time daily from June 1, 1991.
F107_ADJ	10.7-cm solar radio flux (F10.7) (adjusted to 1 AU).
F107_DATA_TYPE	Flux qualifier: • OBS — Observed flux measurement. • INT — CelesTrak linear interpolation of missing data. • PRD — 45-Day predicted flux. • PRM — Monthly predicted flux.
F107_OBS_CENTER81	Centered 81-day arithmetic average of F10.7 (observed).
F107_OBS_LAST81	Last 81-day arithmetic average of F10.7 (observed).
F107_ADJ_CENTER81	Centered 81-day arithmetic average of F10.7 (adjusted to 1 AU).
F107_ADJ_LAST81	Last 81-day arithmetic average of F10.7 (adjusted to 1 AU).

- `'timetable'` — Data is stored in a `timetable`.

  `matfile` contains a `timetable` variable named `'fluxData'`. The columns of the `timetable` have the same names as the `'numeric'` variables with the exception that `YEAR`, `MONTH`, and `DAY` are combined into a single `DATE` column that contains `datetime` objects.

Output Arguments

`matfile` — Full path to MAT-file containing space weather data

character array

Full path to new MAT-file containing space weather data, specified as a character array. The MAT-file name has the format `aerospaceweatherdata`*`YYYYMMDD`*`.mat`, where:

- *YYYY* —Year when the CelesTrak file was read.
- *MM* — Month when the CelesTrak file was read.
- *DD* — Calendar day when the CelesTrak file was read.

`startdate` — Earliest date

earliest date in `spaceWeatherFile` | 3-element vector formatted as `[year,dayOfYear,UTSeconds]`

Earliest date corresponding to the data saved in `matfile`, specified as a 3-element vector formatted as `[year,dayOfYear,UTSeconds]`. The default value is the earliest date in `spaceWeatherFile`.

`enddate` — Latest date
latest date in `spaceWeatherFile` | 3-element vector formatted as `[year,dayOfYear,UTSeconds]`

Latest date corresponding to the data saved in `matfile`, specified as a 3-element vector formatted as `[year,dayOfYear,UTSeconds]`. The default value is the latest date in `spaceWeatherFile`.

Version History

Introduced in R2023a

See Also

`fluxSolarAndGeomagnetic` | `atmosnrlmsise00`

Aero.Aircraft.ControlState class

Package: `Aero`

Define control states of fixed-wing state

Description

An object of the `Aero.Aircraft.ControlState` class defines and manages the control states of fixed-wing states.

Note This class supports fixed-wing objects. Do not directly use this class. To set up the command state vectors on a fixed-wing object, see the `setupControlStates` method.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`Position` — Current control state value

scalar numeric

Current control state value, specified as a scalar numeric.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `double`

`MaximumValue` — Maximum value of control surface

infinity (default) | scalar numeric

Maximum value of control surface, specified as a scalar numeric.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `double`

`MinimumValue` — Minimum value of control surface

negative infinity (default) | scalar numeric

Minimum value of control surface, specified as a scalar numeric.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

DependsOn — Control states

`["", ""]` (default) | two-element vector

Control states upon which the control state depends, specified as a two-element vector.

For asymmetrical control surfaces, the two asymmetrical control states are settable, but the resulting effective control state is not.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `char` | `string`

Settable — Current control state value

`'on'` | `'off'`

Current control state value, specified as `'on'` or `'off'`. Specify `'on'` to make the control state settable. Otherwise, set to `'off'`.

Tip For asymmetrical control surfaces, the two asymmetrical control states are settable, but the resulting effective control state is not.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

Properties — Aero.Aircraft.Properties object

scalar

`Aero.Aircraft.Properties` object, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create and Use Fixed-Wing Object

Create and set up dynamic behavior and the current state for the fixed-wing object aircraft.

Create a fixed-wing object.

```
aircraft = Aero.FixedWing()
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

To define the aircraft dynamic behavior, set a coefficient for it.

```
aircraft = setCoefficient(aircraft, "CD", "Zero", 0.27)
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

Define the aircraft's current state.

```
state = Aero.FixedWing.State("Mass", 500)
```

```
state =

  State with properties:

                 Alpha: 0
                  Beta: 0
              AlphaDot: 0
               BetaDot: 0
                  Mass: 500
               Inertia: [3×3 table]
        CenterOfGravity: [0 0 0]
       CenterOfPressure: [0 0 0]
            AltitudeMSL: 0
```

```
            GroundHeight: 0
                      XN: 0
                      XE: 0
                      XD: 0
                       U: 50
                       V: 0
                       W: 0
                     Phi: 0
                   Theta: 0
                     Psi: 0
                       P: 0
                       Q: 0
                       R: 0
                  Weight: 4905
             AltitudeAGL: 0
                Airspeed: 50
             GroundSpeed: 50
              MachNumber: 0.1469
            BodyVelocity: [50 0 0]
          GroundVelocity: [50 0 0]
                      Ur: 50
                      Vr: 0
                      Wr: 0
       FlightPathAngle: 0
           CourseAngle: 0
  InertialToBodyMatrix: [3×3 double]
  BodyToInertialMatrix: [3×3 double]
      BodyToWindMatrix: [3×3 double]
      WindToBodyMatrix: [3×3 double]
       DynamicPressure: 1.5312e+03
           Environment: [1×1 Aero.Aircraft.Environment]
            UnitSystem: "Metric"
           AngleSystem: "Radians"
     TemperatureSystem: "Kelvin"
         ControlStates: [1×0 Aero.Aircraft.ControlState]
      OutOfRangeAction: "Limit"
      DiagnosticAction: "Warning"
            Properties: [1×1 Aero.Aircraft.Properties]
```

Calculate the forces and moments on the aircraft.

```
[F, M] = forcesAndMoments(aircraft, state)
```

```
F =

           0
           0
        4905


M =

     0
```

```
     0
     0
```

Limitations

You cannot subclass `Aero.Aircraft.ControlState`.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Aero.FixedWing.State` | `setupControlStates`

Topics

“Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121
“Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106

Aero.Aircraft.Environment class

Package: `Aero`

Properties defining and managing aircraft environment

Description

An object of the `Aero.Aircraft.Environment` class defines and manages aircraft environments.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`aeroAircraftEnvironment = Aero.Aircraft.Environment` creates a single `Aero.Aircraft.Environment` object with default property values.

`aeroAircraftEnvironment = Aero.Aircraft.Environment(N)` creates an *N*-by-*N* matrix of `Aero.Aircraft.Environment` objects with default property values.

`aeroAircraftEnvironment = Aero.Aircraft.Environment(M,N,P,...)` or `Aero.Aircraft.Environment([M N P ...])` creates an *M*-by-*N*-by-*P*-by-... array of `Aero.Aircraft.Environment` objects with default property values.

`aeroAircraftEnvironment = Aero.Aircraft.Environment(size(A))` creates an `Aero.Aircraft.Environment` object that is the same size as `A` and all `Aero.Aircraft.Environment` objects.

`aeroAircraftEnvironment = Aero.Aircraft.Environment(__,property,propertyValue)` creates an array of `Aero.Aircraft.Environment` objects with *`property`*, *`propertyValue`* pairs applied to each of the `Aero.Aircraft.Environment` array objects. For a list of properties, see “Properties” on page 4-45.

Input Arguments

`N` — Number of aircraft objects

scalar

Number of aircraft objects, specified as a scalar.

`M` — Number of aircraft objects

scalar

Number of aircraft objects, specified as a scalar.

`P` — Number of aircraft objects
scalar

Number of aircraft objects, specified as a scalar.

`A` — Size of aircraft object
scalar

Size of aircraft object, specified as a scalar.

Properties

`WindVelocity` — Wind velocity in NED coordinates
`[0, 0, 0]` (default) | three-element vector

Wind velocity in NED coordinates, specified as a three-element vector in these units:

Unit	Unit System
Meters per second (m/s)	`'Metric'`
Feet per second (ft/s)	`'English (kts)'`
Knots (kts)	`'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`Density` — Density of air
`1.225` (default) | scalar numeric

Density of air, specified as a scalar, in these units:

Unit	Unit System
Kilograms per meter^3 (kg/m^3)	`'Metric'`
Slugs per foot^3 (slug/ft^3)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`Temperature` — Static air temperature
`288.15` (default) | scalar numeric

Static air temperature, specified as a scalar numeric in units specified by the temperature system.

Attributes:

GetAccess	public
SetAccess	public

Data Types: double

Pressure — Static air pressure

101325 (default) | scalar numeric

Static air pressure, specified as a scalar numeric in these units:

Unit	Unit System
Pascals (Pa)	'Metric'
Pounds per feet^2 (psf)	'English (kts)' and 'English (ft/s)'

Attributes:

GetAccess	public
SetAccess	public

Data Types: double

SpeedOfSound — Speed of sound

340.2941 (default) | scalar numeric

Speed of sound, specified as a scalar numeric in these units:

Unit	Unit System
Meters per second (m/s)	'Metric'
Feet per second (ft/s)	'English (kts)'
Knots (kts)	'English (ft/s)'

Attributes:

GetAccess	public
SetAccess	public

Data Types: double

Gravity — Acceleration due to gravity

9.81 (default) | scalar numeric

Acceleration due to gravity, specified as a scalar numeric in these units:

Unit	Unit System
Meters per second^2 (m/s^2)	'Metric'
Feet per second^s (ft/s^2)	'English (kts)' and 'English (ft/s)'

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

`Properties` — `Aero.Aircraft.Properties` object
scalar

`Aero.Aircraft.Properties` object, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create `Aero.Aircraft.Environment` Object

Create an `Aero.Aircraft.Environment` object.

Create an `Aero.Aircraft.Environment` object.

```
env = Aero.Aircraft.Environment('Gravity', 32.2)
```

```
env =

  Environment with properties:

    WindVelocity: [0 0 0]
         Density: 1.2250
     Temperature: 288.1500
        Pressure: 101352
    SpeedOfSound: 340.2941
         Gravity: 32.2000
      Properties: [1×1 Aero.Aircraft.Properties]
```

Limitations

You cannot subclass `Aero.Aircraft.Environment`.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `setupControlStates`

Topics

“Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121

“Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106

Aero.Aircraft.Properties class

Package: `Aero`

Properties defining and managing aircraft

Description

An object of the `Aero.Aircraft.Properetries` class defines and manages aircraft components. Use this object to model and analyze an aircraft. The object contains the static data for the aircraft, such as reference values, coefficients, and deflection angles.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`aeroAircraft = Aero.Aircraft.Properties` creates a single `Aero.Aircraft.Properties` object with default property values.

`aeroAircraft = Aero.Aircraft.Properties(N)` creates an *N*-by-*N* matrix of `Aero.Aircraft.Properties` objects with default property values.

`aeroAircraft = Aero.Aircraft.Properties(M,N,P,...)` or `Aero.Aircraft.Properties([M N P ...])` creates an *M*-by-*N*-by-*P*-by-... array of `Aero.Aircraft.Properties` objects with default property values.

`aeroAircraft = Aero.Aircraft.Properties(size(A))` creates an `Aero.Aircraft.Properties` object that is the same size as `A` and all `Aero.Aircraft.Properties` objects.

`aeroAircraft = Aero.Aircraft.Properties(__,property,propertyValue)` creates an array of `Aero.Aircraft.Properties` objects with *`property`*, *`propertyValue`* pairs applied to each of the `Aero.Aircraft.Properties` array objects. For a list of properties, see “Properties” on page 4-50.

Input Arguments

`N` — Number of aircraft objects

scalar

Number of aircraft objects, specified as a scalar.

`M` — Number of aircraft objects

scalar

Number of aircraft objects, specified as a scalar.

P — Number of aircraft objects
scalar

Number of aircraft objects, specified as a scalar.

A — Size of aircraft object
scalar

Size of aircraft object, specified as a scalar.

Properties

Name — Object name
scalar

Object name, specified as a scalar string or character vector. The `update` method of an `Aero.FixedWing.*` object uses this name to update the `Simulink.LookupTable.StructTypeInfo.Name` property.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `char` | `string`

Description — Object description
string array

Object description, specified as a scalar string or character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `char` | `string`

Type — Object type
scalar string

Object type, specified as a scalar string or character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `char` | `string`

Version — Object version
scalar string

Object version, specified as a scalar string or character vector.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `char` | `string`

Examples

Create `Aero.Aircraft.Properties` Object

Create an `Aero.Aircraft.Properties` object and set object name to `MyAircraft`.

Create a fixed-wing object.

```
props = Aero.Aircraft.Properties('Name','MyAircraft')
```

```
props =

  Properties with properties:

          Name: "MyAircraft"
    Description: ""
          Type: ""
       Version: ""
```

Limitations

- This class requires a Simulink license.
- You cannot subclass `Aero.Aircraft.Properties`.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing`

Topics

“Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121

Aero.Animation class

Package: `Aero`

Visualize aerospace animation

Description

`Aero.Animation` objects visualize flight data without any other tool or toolbox. You only need an Aerospace Toolbox license.

The `Aero.Animation` class is a `handle` class.

Creation

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."

Properties

`Bodies` — `Aero.Body` objects
0 x 0 cell array (default) | cell array

`Aero.Body` objects specified in a cell array.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Camera` — Camera contained by animation object
`[ ]` (default) | scalar `Aero.Camera`

Camera contained by animation object, specified as an `Aero.Camera` handle object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Figure` — Figure object
`[ ]` (default) | scalar `Figure` object

`Figure` object, specified as a MATLAB array.

Attributes:

GetAccess	public
SetAccess	public

FigureCustomizationFcn — Figure customization function

`[ ]` (default) | Function handle

Figure customization function, specified as a function handle. The function handle operates on the `Figure` handle stored by the `Figure` property.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

FramesPerSecond — Animation rate

`12` (default) | scalar

Animation rate, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

Name — Animation object name

`''` (default) | character vector | string scalar

Animation object name, specified as a character vector or string scalar.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `char` | `string`

TCurrent — Current time

`0` (default) | scalar

Current time, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

TFinal — End time

NaN (default) | scalar

End time, specified as a scalar.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`TStart` — Start time

NaN (default) | scalar

Start time, specified as a scalar.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`TimeScaling` — Scaling time

1 (default) | scalar

Scaling time, specified as a scalar.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`VideoCompression` — Video recording compression file type

`'Archival'` (default) | `'Motion JPEG AVI'` | `'Motion JPEG 2000'` | `'MPEG-4'` | `'Uncompressed AVI'`

Video recording compression file type, specified as one of these values. For more information on video compression, see `VideoWriter`.

Value	Description
`'Archival'`	Motion JPEG 2000 file with lossless compression.
`'Motion JPEG AVI'`	Compressed AVI file using Motion JPEG codec.
`'Motion JPEG 2000'`	Compressed Motion JPEG 2000 file.
`'MPEG-4'`	Compressed MPEG-4 file with H.264 encoding (Windows 7 systems only).
`'Uncompressed AVI'`	Uncompressed AVI file with RGB24 video.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`VideoFileName` — Video recording file name

`'temp'` (default) | character vector | string scalar

Video recording file name, specified as a character vector or a string.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `char` | `string`

`VideoQuality` — Video recording quality

`75` (default) | scalar value between `0` and `100`

Video recording quality, specified as a scalar value between `0` and `100`. For more information on video quality, see the `Quality` property in `VideoWriter`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`VideoRecord` — Option to enable video recording

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

Option to enable video recording, specified as one of these values.

Value	Action
`'on'`	Enable video recording.
`'off'`	Disable video recording.
`'scheduled'`	Schedule video recording. Use this setting with the “VideoTStart” on page 4-0 and “VideoTFinal” on page 4-0 properties.

If you are capturing frames of a plot that take a long time to generate or are repeatedly capturing frames in a loop, check your settings to make sure that your computer screen saver does not activate and that your monitor does not turn off for the duration of the capture. If either of these actions occurs, one or more of the captured frames might contain graphics from your screen saver or be blank.

Note In situations where MATLAB software is running on a virtual desktop that is not currently visible on your monitor, it might capture a region on your monitor that corresponds to the position occupied by the figure or axes on the hidden desktop. Check that the window to be captured is on the currently active desktop.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`VideoTFinal` — Video recording stop time for scheduled recording

NaN, which uses the value of `TFinal` (default) | scalar value between `TStart` and `TFinal`

Video recording stop time for scheduled recording, specified as a scalar value between `TStart` and `TFinal`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Dependencies

Use when `VideoRecord` is set to `'scheduled'`. Use `VideoTStart` to set the start time of the recording.

Data Types: `double`

`VideoTStart` — Video recording start time for scheduled recording

NaN, which uses the value of `TStart` (default) | scalar value between `TStart` and `TFinal`

Video recording start time for scheduled recording, specified as a scalar value between `TStart` and `TFinal`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Dependencies

Use this property when `VideoRecord` is set to `'scheduled'`. Use `VideoTFinal` to set the stop time of the recording.

Data Types: `double`

Methods

Public Methods

addBody	Add body to animation object
createBody	Create body and its associated patches in animation
delete	Destroy animation object
hide	Hide animation figure
initIfNeeded	Initialize animation graphics if needed
initialize	Create animation object figure and axes and build patches for bodies
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
wait	Wait until animation stops playing

Examples

Create `Aero.Animation` Object

```
h=Aero.Animation
```

```
h =
  Animation with properties:

                     Name: ''
                   Figure: []
    FigureCustomizationFcn: []
                   Bodies: {}
                   Camera: []
                 TCurrent: 0
              VideoRecord: 'off'
            VideoFileName: 'temp'
          VideoCompression: 'Motion JPEG AVI'
              VideoQuality: 75
               VideoTStart: NaN
               VideoTFinal: NaN
               TimeScaling: 1
            FramesPerSecond: 12
                    TStart: NaN
                    TFinal: NaN
```

Record Animation Object Simulation

Simulate and record flight data. Use the data to create an animation object.

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

Control the frame display rate.

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

Set the time-scaling property `TimeScaling` on the animation object to specify the data per second.

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

The combination of the `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 = createBody(h,'pa24-250_orange.ac','Ac3d');
```

Load simulated flight trajectory data, `simdata`.

```
load simdata;
```

Set the time series data for the body.

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

Create a figure object for the animation object.

```
show(h);
```

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]
                 TCurrent: 0
              VideoRecord: 'on'
            VideoFileName: 'temp'
         VideoCompression: 'Motion JPEG AVI'
             VideoQuality: 50
              VideoTStart: NaN
              VideoTFinal: NaN
              TimeScaling: 5
           FramesPerSecond: 10
                   TStart: NaN
                   TFinal: NaN
```

```
h.VideoFilename = 'astMotion_JPEG';
```

Play the animation.

```
play(h);
```

Verify that a file named `astMotion_JPEG.avi` was created in the current folder.

Wait

Wait for the animation to stop playing before the modifying the object.

```
wait(h);
```

Disable recording to preserve the file.

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

Record Animation for Four Seconds

Simulate flight data for four seconds.

Use the data to 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 the `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 = createBody(h,'pa24-250_orange.ac','Ac3d');
```

Load simulated flight trajectory data (`simdata`).

```
load simdata;
```

Set the time series data for the body.

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

Create a figure object for the animation object.

```
show(h);
```

Set up recording properties.

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

Play the animation from `TStart` to `TFinal`.

```
h.TSTart = 1;
h.TFinal = 5;
play(h);
```

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.

Wait

Wait for the animation to stop playing before the modifying the object.

```
wait(h);
```

Disable recording to preserve the file.

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

Version History

Introduced in R2007a

See Also

`Aero.Body` | `Aero.Camera` | `Aero.Geometry` | `Aero.FlightGearAnimation` | `Aero.VirtualRealityAnimation`

Topics

“Aerospace Toolbox Animation Objects” on page 2-22
“Aero.Animation Objects” on page 2-23
“Simulated and Actual Flight Data Using Aero.Animation Objects” on page 2-24
“Overlaying Simulated and Actual Flight Data” on page 5-31

Aero.Body class

Package: `Aero`

Create body object for use with animation object

Description

Use the `Aero.Body` class to create a body object for use with an `Aero.Animation` object. 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. This step requires an axes from a figure.
5. Set the `timeseries` 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 `timeseries` data. It expects the rotation order *z-y-x* (psi, theta, phi).

Convert `timeseries` data from other coordinate systems by registering a different `CoordTransformFcn` function.

The `Aero.Body` class is a `handle` class.

Creation

`h = Aero.Body` constructs a body for an animation object. The animation object is returned in `h`.

Properties

Name — Name of body

`''` (default) | character vector | string scalar

Name of body, specified as a character vector or string scalar.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `string` | `char`

CoordTransformFcn — Function that controls coordinate transformation

`@nullCoordTransform` (default) | function handle | character vector | string scalar

Function that controls coordinate transformation, specified as a function handle, character vector, or string scalar.

Attributes:

GetAccess	public
SetAccess	public

Position — Position of body

`[0 0 0]` (default) | MATLAB array

Position of body, specified as a MATLAB array.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

Rotation — Rotation of body

`[0 0 0]` (default) | MATLAB array

Rotation of body, specified as a MATLAB array.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

Geometry — Geometry of body

`[1×1 Aero.Geometry]` (default) | handle

Geometry of body, specified as a handle.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

PatchGenerationFcn — Patch generation function

`@generatePatchesFromFvcData` (default) | function handle | character vector | string scalar

Patch generation function, specified as a function handle, character vector, or string scalar.

Attributes:

GetAccess	public
SetAccess	public

PatchHandles — Patch handles

`[]` (default) | MATLAB array

Patch handles, specified as a MATLAB array.

Attributes:

GetAccess	public
SetAccess	public

Data Types: double

ViewingTransform — Viewing transform

[] (default) | scalar

Viewing transform, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

TimeSeriesSource — Time series data source

timeseries object | timetable object | StructureWithTime | double precision array

Time series source, specified as a time series source.

The time series data property stored in and interpreted according to the 'TimeSeriesSourceType' property.

Attributes:

GetAccess	public
SetAccess	public

TimeSeriesSourceType — Type of timeseries source

'Array6DoF' (default) | 'Timeseries' | 'Timetable' | 'StructureWithTime' | 'Array3DoF' | 'Custom'

Type of timeseries data stored in 'TimeSeriesSource' property, specified as one of these values:

timeseries Type	Description
'Array6DoF'	A double-precision array in n rows and seven columns for 6-DoF data: time x y z phi theta psi. If a double-precision array of eight or more columns is in 'TimeSeriesSource', the first seven columns are used as 6-DoF data.
'Timeseries'	MATLAB timeseries data with six values per time: x y z phi theta psi The values are resampled.

timeseries Type	Description
'Timetable'	MATLAB timetable 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 versus time using interp1.
'Array3DoF'	A double-precision array in n rows and four columns for 3-DoF data: time x z theta. If a double-precision array of five or more columns is in 'TimeSeriesSource', the first four columns are used as 3-DoF data.
'Custom'	Position and angle data are retrieved from 'TimeSeriesSource' by the currently registered 'TimeseriesReadFcn'.

Attributes:

GetAccess	public
SetAccess	public

Data Types: string | char

TimeSeriesReadFcn — timeseries read function

@interp6DoFArrayWithTime (default) | function handle | character vector | string

timeseries read function, specified as a function handle, character vector, or string.

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

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 animation body position and orientation
update	Update body position and orientation as function of time

Examples

Create Aero.Body Object

```
h = Aero.Body
```

```
h =
  Body with properties:

                    Name: ''
       CoordTransformFcn: @nullCoordTransform
                Position: [0 0 0]
                Rotation: [0 0 0]
                Geometry: [1x1 Aero.Geometry]
     PatchGenerationFcn: @generatePatchesFromFvcData
           PatchHandles: []
         ViewingTransform: []
         TimeSeriesSource: []
    TimeSeriesSourceType: 'Array6DoF'
       TimeSeriesReadFcn: @interp6DoFArrayWithTime
```

Version History

Introduced in R2007a

See Also

Aero.Animation | Aero.Geometry

Topics

“Aerospace Toolbox Animation Objects” on page 2-22
“Aero.Animation Objects” on page 2-23
“Simulated and Actual Flight Data Using Aero.Animation Objects” on page 2-24

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 5-31
- “Camera Graphics Terminology”
- “Low-Level Camera Properties”

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]`

Property	Description	Values
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 {[]}

Version History

Introduced in R2007a

See Also

Aero.Geometry

aeroDataPackage

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

Syntax

```
aeroDataPackage
```

Description

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

Examples

Start Add-On Explorer

Start the Add-On Explorer to add data.

```
aeroDataPackage
```

Limitations

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

Version History

Introduced in R2014a

See Also

Topics

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

Aero.FixedWing.Coefficient class

Package: `Aero`

Create `Aero.FixedWing` aircraft coefficient set

Description

`Aero.FixedWing.Coefficient` creates an `Aero.FixedWing` coefficient set that describes the behavior and body of an aircraft.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`fixedWingCoefficient = Aero.FixedWing.Coefficient` creates a single `Aero.FixedWing.Coefficient` object with default property values.

`fixedWingCoefficient = Aero.FixedWing.Coefficient(N)` creates an *N*-by-*N* matrix of `Aero.FixedWing.Coefficient` objects with default property values.

`fixedWingCoefficient = Aero.FixedWing.Coefficient(M,N,P,...)` or `Aero.FixedWing.Coefficient([M N P ...])` creates an *M*-by-*N*-by-*P*-by-... array of `Aero.FixedWing.Coefficient` objects with default property values.

`fixedWingCoefficient = Aero.FixedWing.Coefficient(size(A))` creates an `Aero.FixedWing.Coefficient` object that is the same size as `A` and all `Aero.FixedWing.Coefficient` objects.

`fixedWingCoefficient = Aero.FixedWing.Coefficient(__,property,propertyValue)` creates an array of `Aero.FixedWing.Coefficient` objects with *`property`*, *`propertyValue`* pairs applied to each of the `Aero.FixedWing.Coefficient` array objects. For a list of properties, see “Properties” on page 4-71.

Input Arguments

`N` — Number of fixed-wing coefficient objects

scalar

Number of fixed-wing coefficient objects, specified as a scalar.

`M` — Number of fixed-wing coefficient objects

scalar

Number of fixed-wing coefficient objects, specified as a scalar.

P — Number of fixed-wing coefficient objects
scalar

Number of fixed-wing coefficient objects, specified as a scalar.

A — Size of fixed-wing coefficient object
scalar

Size of fixed-wing coefficient object, specified as a scalar.

Properties

Public Properties

Table — Coefficient values
6-by-*N* table

Coefficient values, specified in a 6-by-*N* table. Each row in the table must be a member of and in the same order as the "StateOutput" on page 4-0 property.

Setting the `Table` property also sets the contents of the `Values` property and `StateVariables` to the `Table` property variables. To have a `Simulink.LookupTable` object and a constant value in the same column, use the `setCoefficient` or set the desired content of the `Values` property. Setting the `Table` property does not set the `ReferenceFrame`.

Note Tables must have a single data type per column. If there are both constant values and `Simulink.LookupTable` objects in a given column, the `Table` property automatically converts the constants to `Simulink.LookupTable` objects.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `double`

Values — Coefficient values
6-by-*N* cell array

Coefficient values, specified as a 6-by-*N* cell array. Each entry in the cell array must be a single coefficient value corresponding to the `StateOutput` (row) and `StateVariable` (column) properties. Each coefficient value must be a scalar numeric value or a `Simulink.LookupTable` object. If a value is a `Simulink.LookupTable` object, the `FieldName` of each breakpoint must be a valid property of the `Aero.FixedWing.State` object.

Unlike the `Table` property, `Values` do need to be a single data type per column.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `double`

`StateVariables` — State variable names

`'Zero'` (default) | 1-by-*N* vector

State variable names, specified as a 1-by-*N* vector of strings. Each entry in this property corresponds to a column in the `Values` property. Each entry in `StateVariables` must be a valid property in the `Aero.FixedWing.State` object. Adding a state variable adds a column of zeros to the end of the `Values` cell array.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `char` | `string`

`ReferenceFrame` — Reference frame for coefficients

`Wind` (default) | `Body` | `Stability`

Reference frame for coefficients, specified as `Wind`, `Body`, or `Stability` with these outputs:

Reference Frame	Coefficient Output
Wind	Forces: • drag (CD) • Y (CY) • lift (CL)
	Moments: • L (Cl) • M (Cm) • N (Cn)
Body	Forces: • X (CX) • Y (CY) • Z (CZ)
	Moments: • L (Cl) • M (Cm) • N (Cn)
Stability	Forces: • drag (CD) • Y (CY) • lift (Cn)

Reference Frame	Coefficient Output
	Moments: • L (Cl) • M (Cm) • N (Cn)

Example of `Wind` table:

Coefficient	State
`CD`	*state*
`CY`	*state*
`CL`	*state*
`Cl`	*state*
`Cm`	*state*
`Cn`	*state*

Example of `Body` table:

Coefficient	State
`CX`	*state*
`CY`	*state*
`CZ`	*state*
`Cl`	*state*
`Cm`	*state*
`Cn`	*state*

Example of `Stability` table:

Coefficient	State
`CD`	*state*
`CY`	*state*
`CL`	*state*
`Cl`	*state*
`Cm`	*state*
`Cn`	*state*

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `char` | `string`

`MultiplyStateVariables` — Option to multiply coefficients by state variables
`on` (default) | `off`

Option to multiply coefficients by state variables when calculating forces and moments. To multiply coefficients by state variables, set this property to `'on'`. Otherwise, set this property to `'off'`.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `char` | `string`

`NonDimensional` — Option to specify coefficients are nondimensional

`on` (default) | `off`

Option to specify that nondimensional coefficients. To specify nondimensional coefficients, set this property to `'on'`. Otherwise, set this property to `'off'`.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `char` | `string`

`Properties` — `Aero.Aircraft.Properties` object

scalar

`Aero.Aircraft.Properties` object, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

Protected Properties

`StateOutput` — Current state output

6-by-1 vector

Current state output, returned as one of these 6-by-1 vectors:

Wind	Body	Stability
CD	CX	CD
CY	CY	CY
CL	CZ	CL
Cl	Cl	Cl
Cm	Cm	Cm
Cn	Cn	Cn

This property depends on `ReferenceFrame`.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: `char` | `string`

Methods

Public Methods

getCoefficient	Get coefficient values from fixed-wing coefficient object
setCoefficient	Set coefficient values for fixed-wing coefficient object
update	Update Aero.FixedWing.Coefficient object

Examples

Create and Use Fixed-Wing Object

Create and set up dynamic behavior and the current state for the fixed-wing object aircraft.

Create a fixed-wing object.

```
aircraft = Aero.FixedWing()
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

To define the aircraft dynamic behavior, set a coefficient for it.

```
aircraft = setCoefficient(aircraft, "CD", "Zero", 0.27)
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

Define the aircraft's current state.

```
state = Aero.FixedWing.State("Mass", 500)
```

```
state =

  State with properties:

                  Alpha: 0
                   Beta: 0
               AlphaDot: 0
                BetaDot: 0
                   Mass: 500
                Inertia: [3×3 table]
         CenterOfGravity: [0 0 0]
       CenterOfPressure: [0 0 0]
             AltitudeMSL: 0
            GroundHeight: 0
                      XN: 0
                      XE: 0
                      XD: 0
                       U: 50
                       V: 0
                       W: 0
                     Phi: 0
                   Theta: 0
                     Psi: 0
                       P: 0
                       Q: 0
                       R: 0
                  Weight: 4905
              AltitudeAGL: 0
                 Airspeed: 50
              GroundSpeed: 50
               MachNumber: 0.1469
             BodyVelocity: [50 0 0]
           GroundVelocity: [50 0 0]
                      Ur: 50
                      Vr: 0
                      Wr: 0
          FlightPathAngle: 0
              CourseAngle: 0
    InertialToBodyMatrix: [3×3 double]
    BodyToInertialMatrix: [3×3 double]
        BodyToWindMatrix: [3×3 double]
        WindToBodyMatrix: [3×3 double]
         DynamicPressure: 1.5312e+03
             Environment: [1×1 Aero.Aircraft.Environment]
           ControlStates: [1×0 Aero.Aircraft.ControlState]
        OutOfRangeAction: "Limit"
        DiagnosticAction: "Warning"
              Properties: [1×1 Aero.Aircraft.Properties]
              UnitSystem: "Metric"
             AngleSystem: "Radians"
       TemperatureSystem: "Kelvin"
```

Calculate the forces and moments on the aircraft.

```
[F, M] = forcesAndMoments(aircraft, state)
```

```
F =

           0
           0
        4905


M =

     0
     0
     0
```

Limitations

- This class requires a Simulink license if the coefficient table contains `Simulink.LookupTable` objects.
- You cannot subclass `Aero.FixedWing.Coefficient`.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Aero.FixedWing.Surface` | `Aero.FixedWing.Thrust` | `Simulink.LookupTable` | `setCoefficient`

Aero.FixedWing.State class

Package: `Aero`

Define condition of `Aero.FixedWing` aircraft at time instant

Description

Use the `Aero.FixedWing.State` class to define the condition of an `Aero.FixedWing` aircraft at a time instant. The `Aero.FixedWing.State` object contains the information about the current state of an aircraft at a single instance in time. A subclass can inherit the `Aero.FixedWing.State`.

- To get dependent properties defined by subclass, use the `getState` method.
- To set dependent properties, use the `setState` method.
- To use custom state properties within the `Aero.FixedWing` object methods, create a subclass.

Class Attributes

`Sealed`	`false`

For information on class attributes, see “Class Attributes”.

Creation

Description

`Aero.FixedWing.State` creates a single `Aero.FixedWing.State` object with default property values..

`Aero.FixedWing.State(N)` creates an *N*-by-*N* matrix of `Aero.FixedWing.State`.

`Aero.FixedWing.State(M,N,P,...)` or `Aero.FixedWing.State([M N P ...])` creates an *M*-by-*N*-by-*P*-by-... array of `Aero.FixedWing.State`.

`Aero.FixedWing.State(size(A))` creates an `Aero.FixedWing.State` object that is the same size as `A` and all `Aero.FixedWing.State` objects.

`Aero.FixedWing.State(__,property,propertyValue)` creates an array of `Aero.FixedWing.State` objects with *`property`*, *`propertyValue`* pairs applied to each of the `Aero.FixedWing` array objects. For a list of properties, see “Properties” on page 4-79.

Input Arguments

`N` — Number of fixed-wing objects

scalar

Number of fixed-wing objects, specified as a scalar.

`M` — Number of fixed-wing objects

scalar

Number of fixed-wing objects, specified as a scalar.

`P` — Number of fixed-wing objects
scalar

Number of fixed-wing objects, specified as a scalar.

`A` — Size of fixed-wing object
scalar

Size of fixed-wing object, specified as a scalar.

Properties

Public Properties

`UnitSystem` — Unit system
`'Metric'` (default) | `'English (kts)'` | `'English (ft/s)'` | scalar string | character vector

Unit system, specified as a scalar string or character vector.

Attributes:

```
GetAccess                                       public
SetAccess                                       public
```

Data Types: `string` | `char`

`AngleSystem` — Angle system
`'Radians'` (default) | `'Degrees'`

Angle system, specified as `'Radians'` or `'Degrees'`.

Attributes:

```
GetAccess                                       public
SetAccess                                       public
```

Data Types: `string` | `char`

`TemperatureSystem` — Temperature system
`'Kelvin'` (default) | `'Celsius'` | `'Rankine'` | `'Fahrenheit'`

Temperature system, specified as `'Kelvin'`, `'Celsius'`, `'Rankine'`, or `'Fahrenheit'`.

Attributes:

```
GetAccess                                       public
SetAccess                                       public
```

Data Types: `string` | `char`

`Mass` — Fixed-wing aircraft mass
`0` (default) | scalar numeric

Fixed-wing aircraft mass, specified as a scalar numeric, in the units:

Unit	Unit System
newtons (N)	`'Metric'`
slugs (slug)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`Inertia` — Inertial matrix of aircraft

`3-by-3` table of numeric values (default) | scalar numeric

Inertial matrix of aircraft, specified as a 3-by-3 table of numeric values specifying the body in this matrix form:

	X	Y	Z
X	Ixx	Ixy	Ixz
Y	Iyx	Iyy	Iyz
Z	Izx	Izy	Izz

The matrix has these units:

Unit	Unit System
kilogram meters squared (kg m^2)	`'Metric'`
slug feet squared (slug ft^2)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`CenterOfGravity` — Location of center of gravity

[0, 0, 0] (default) | three-element vector

Location of center of gravity on the fixed-wing aircraft in the body frame, specified as a three-element vector in these units:

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`CenterOfPressure` — Location of center of pressure

[0, 0, 0] (default) | three-element vector

Location of center of pressure on the fixed-wing aircraft in the body frame, specified as a three-element vector, in these units:

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                       public
SetAccess                       public
```

Data Types: `string` | `char`

`AltitudeMSL` — Altitude above sea level

0 (default) | scalar numeric

Altitude above sea level, specified as a scalar numeric, in these units:

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                       public
SetAccess                       public
```

Data Types: `string` | `char`

`GroundHeight` — Ground height above sea level

0 (default) | scalar numeric

Ground height above sea level, specified as a scalar numeric in these units:

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                       public
SetAccess                       public
```

Data Types: `string` | `char`

`XN` — North position of fixed-wing aircraft

0 (default) | scalar numeric

North position of fixed-wing aircraft, specified as a scalar numeric in these units:

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

XE — East position of fixed-wing aircraft

0 (default) | scalar numeric

East position of fixed-wing aircraft, specified as a scalar numeric in these units:

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

U — Forward component of ground velocity

50 (default) | scalar numeric

Forward component of ground velocity, specified as a scalar numeric in these units:

Unit	Unit System
Meters per second (m/s)	`'Metric'`
Feet per second (ft/s)	`'English (kts)'`
Knots (kts)	`'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

V — Side component of ground velocity

0 (default) | scalar numeric

Side component of ground velocity, specified as a scalar numeric in these units:

Unit	Unit System
Meters per second (m/s)	`'Metric'`

Unit	Unit System
Feet per second (ft/s)	`'English (kts)'`
Knots (kts)	`'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`W` — Downward component of ground velocity

0 (default) | scalar numeric

Downward component of ground velocity, specified as a scalar numeric in these units:

Unit	Unit System
Meters per second (m/s)	`'Metric'`
Feet per second (ft/s)	`'English (kts)'`
Knots (kts)	`'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`Phi` — Euler roll angle

0 (default) | scalar numeric

Euler roll angle, specified as a scalar numeric in units of radians or degrees depending on the `AngleSystem` property.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`Theta` — Euler pitch angle

0 (default) | scalar numeric

Euler pitch angle, specified as a scalar numeric in units of radians or degrees depending on the `AngleSystem` property.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`Psi` — Euler yaw angle

0 (default) | scalar numeric

Euler yaw angle, specified as a scalar numeric in units of radians or degrees depending on the `AngleSystem` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `string` | `char`

`P` — Body roll rate

0 (default) | scalar numeric

Body roll rate, specified as a scalar numeric in units of radians per second or degrees per second depending on the `AngleSystem` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `string` | `char`

`Q` — Body pitch rate

0 (default) | scalar numeric

Body pitch rate, specified as a scalar numeric in units of radians per second or degrees per second depending on the `AngleSystem` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `string` | `char`

`R` — Body yaw rate

0 (default) | scalar numeric

Body yaw rate, specified as a scalar numeric in units of radians per second or degrees per second depending on the `AngleSystem` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `string` | `char`

`AlphaDot` — Angle of attack rate on fixed-wing aircraft

0 (default) | scalar numeric

Angle of attack rate on fixed-wing aircraft, specified as a scalar numeric in units of radians per second or degrees per second depending on the `AngleSystem` property.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `string` | `char`

`BetaDot` — Angle of sideslip rate on fixed-wing aircraft

0 (default) | scalar numeric

Angle of sideslip rate on the fixed-wing aircraft, specified as a scalar numeric in units of radians per second or degrees per second depending on the `AngleSystem` property.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `string` | `char`

`ControlStates` — Current control state values

vector

Current control state values, specified as a vector.

- To set up control states, use `setupControlStates`.
- To set the control state positions, use `setState`.
- To get the control state positions, use `getState`.

You cannot set effective control variables created with asymmetric control surfaces.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `string` | `char`

`Environment` — Definition of current environment

scalar

Definition of current environment, contained in an `Aero.Aircraft.Environment` object, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `string` | `char`

Protected Properties

`Weight` — Fixed-wing aircraft weight

scalar numeric

Fixed-wing aircraft weight, specified as a scalar numeric, in these units:

Unit	Unit System
newtons (N)	`'Metric'`
pound-force (lbf)	`'English (kts)'` and `'English (ft/s)'`

`Weight` depends on the values of the `Mass` and `Gravity` properties of the `Aero.Aircraft.Environment` object, with the equation

`Weight` = *`Mass`* * *`Environment.Gravity`*.

Attributes:

```
GetAccess                   Restricts access
SetAccess                   protected
```

Data Types: `double`

`AltitudeAGL` — Altitude above ground level
scalar numeric

Altitude above ground level, specified as a scalar numeric value in these units:

Unit	Unit System
meters (m)	`'Metric'`
feet (ft)	`'English (kts)'` and `'English (ft/s)'`

`AltitudeAGL` depends on the values of the `AltitudeMSL` and `GroundHeight` public properties, with the equation:

`AltitudeAGL` = *`AltitudeMSL`* - *`GroundHeight`*.

Attributes:

```
GetAccess                   Restricts access
SetAccess                   protected
```

Data Types: `double`

`XD` — Down position of fixed-wing aircraft
0 (default) | scalar numeric

Down position of fixed-wing aircraft, specified as a scalar numeric in these units:

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

`XD` depends on the value of the `AltitudeMSL` public property, with the equation

`XD` = `-1` * *`AltitudeMSL`*.

Attributes:

```
GetAccess                   Restricts access
SetAccess                   protected
```

Data Types: `double`

`Airspeed` — Current airspeed of fixed-wing aircraft

0 (default) | scalar numeric

Current airspeed of fixed-wing aircraft, specified as a scalar numeric in these units:

Unit	Unit System
Meters/sec (m/s)	`'Metric'`
Feet/sec (ft/s)	`'English (ft/s)'`
knots (kts)	`'English (kts)'`

`Airspeed` depends on the values of the `Ur`, `Vr`, and `Wr` public properties, with the equation

$Airspeed = \text{sqrt}(Ur^2 + Vr^2 + Wr^2)$.

Attributes:

```
GetAccess                   Restricts access
SetAccess                   protected
```

Data Types: `double`

`GroundSpeed` — Current ground speed of fixed-wing aircraft

three-element vector

Current ground speed of fixed-wing aircraft, specified as a three-element vector in these units:

Unit	Unit System
Meters/sec (m/s)	`'Metric'`
Feet/sec (ft/s)	`'English (ft/s)'`
knots (kts)	`'English (kts)'`

`Groundspeed` depends on the values of the `U`, `V`, and `R` public properties, with the equation

$Groundspeed = [U,\ V, W]$.

Attributes:

```
GetAccess                   Restricts access
SetAccess                   protected
```

Data Types: `double`

`MachNumber` — Mach number

numeric scalar

Mach number of fixed-wing aircraft, specified as a numeric scalar.

`MachNumber` depends on the values of the `AirSpeed` and `SpeedOfSound` public properties, with the equation

`MachNumber` = *`AirSpeed`*/*`Environment.SpeedOfSound`*.

Attributes:

`GetAccess`	`Restricts access`
`SetAccess`	`protected`

Data Types: `double`

`BodyVelocity` — Body velocity of fixed-wing aircraft
three-element vector

Body velocity of fixed-wing aircraft, specified as a three-element vector.

`BodyVelocity` depends on the values of the `GroundSpeed`, `Phi`, `Theta`, and `Psi` public properties, with the equation

`BodyVelocity` = *`GroundVelocity`* - *`InertialToBodyMatrix`* * *`Environment`*.*`WindVelocity`*.

Attributes:

`GetAccess`	`Restricts access`
`SetAccess`	`protected`

Data Types: `double`

`GroundVelocity` — Ground velocity of fixed-wing aircraft
three-element vector

Ground velocity of the fixed-wing aircraft, specified as a three-element vector, defined with the equation

`GroundVelocity` = [*U*, *V*, *W*].

Attributes:

`GetAccess`	`Restricts access`
`SetAccess`	`protected`

Data Types: `double`

`Ur` — *X* component of body velocity
scalar numeric

X component of body velocity, specified as scalar numeric. *`Ur`* depends on *`BodyVelocity`*.

Attributes:

`GetAccess`	`Restricts access`
`SetAccess`	`protected`

Data Types: `double`

`Vr` — *Y* component of body velocity
scalar numeric

Y component of body velocity, specified as scalar numeric. *Ur* depends on *BodyVelocity*.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

Wr — *Z* component of body velocity

scalar numeric

Z component of body velocity, specified as scalar numeric. *Ur* depends on *BodyVelocity*.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

FlightPathAngle — Flight path angle

scalar numeric

Flight path angle, specified as a scalar numeric in units of radians or degrees depending on the AngleSystem property. FlightPathAngle is defined with the equation:

FlightPathAngle = atan2(*W*,*U*).

.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

CourseAngle — Course angle

scalar numeric

Course angle, specified as a scalar numeric in units of radians or degrees depending on the AngleSystem property. CourseAngle depends on *V* and *U* with the equation

CourseAngle = atan2(*V*,*U*).

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

Alpha — Angle of attack

scalar numeric

Angle of attack, specified as a scalar numeric in units of radians or degrees depending on the AngleSystem property. Alpha depends on *Wr* and *Ur* with the equation:

Alpha = atan2(*Wr*,*Ur*).

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

Beta — Angle of side slip

scalar numeric

Angle of side slip, specified as a scalar numeric in units of radians or degrees depends on the AngleSystem property. Beta depends on *Vr* and Airspeed with the equation:

Beta = asin(*Vr*/*Airspeed*).

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

InertialToBodyMatrix — Inertial to body axes transformation matrix

3-by-3 matrix

Inertial to body axes transformation matrix, specified as a 3-by-3 matrix to convert stability axes to body axes. This property depends on the Phi, Theta, and Psi properties.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

BodyToInertialMatrix — Body axes to stability axes transformation matrix

3-by-3 matrix

Body axes to stability axes transformation matrix, specified as a 3-by-3 matrix to convert stability axes to body axes. This property depends on the Phi, Theta, and Psi properties.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

BodyToWindMatrix — Body to wind axes transformation matrix

3-by-3 matrix

Body to wind axes transformation matrix, specified as a 3-by-3 matrix to convert body axes to wind axes. This property depends on the Alpha and Beta properties.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: `double`

`WindToBodyMatrix` — Wind to body axes transformation matrix

3-by-3 matrix

Wind to body axes transformation matrix, specified as a 3-by-3 matrix to convert wind axes to the body axes. This property depends on the `Alpha` and `Beta` properties.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: `double`

`BodyToStabilityMatrix` — Body axes to stability axes transformation matrix

3-by-3 matrix

Body axes to stability axes transformation matrix, specified as a 3-by-3 matrix. For a definition of `BodyToStabilityMatrix`, see “Algorithms” on page 4-94.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: `string` | `char`

`StabilityToBodyMatrix` — Stability axes to body matrix axes transformation matrix

3-by-3 matrix

Stability axes to body matrix axes transformation matrix, specified as a 3-by-3 matrix. For a definition of `StabilityToBodyMatrix`, see “Algorithms” on page 4-94.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: `string` | `char`

`DynamicPressure` — Dynamic pressure at current state

scalar numeric

Dynamic pressure at current state, specified as a scalar numeric in these units:

Unit	Unit System
Pascals (Pa)	`'Metric'`
pounds per foot squared (lbf/ft^2)	`'English (ft/s)'` and `'English (kts)'`

This property is defined with the equation

DynamicPressure = 0.5 * *Environment.Density* * *Airspeed*^{2}.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: double

Methods

Public Methods

getState	Get state value
setState	Set state value to Aero.FixedWing.State object
setupControlStates	Set up control states for Aero.FixedWing.State object

Examples

Create and Use Fixed-Wing Object

Create and set up dynamic behavior and the current state for the fixed-wing object aircraft.

Create a fixed-wing object.

```
aircraft = Aero.FixedWing()
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

To define the aircraft dynamic behavior, set a coefficient for it.

```
aircraft = setCoefficient(aircraft, "CD", "Zero", 0.27)
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
```

```
     DegreesOfFreedom: "6DOF"
             Surfaces: [1×0 Aero.FixedWing.Surface]
              Thrusts: [1×0 Aero.FixedWing.Thrust]
          AspectRatio: NaN
           UnitSystem: "Metric"
          AngleSystem: "Radians"
    TemperatureSystem: "Kelvin"
           Properties: [1×1 Aero.Aircraft.Properties]
```

Define the aircraft's current state.

```
state = Aero.FixedWing.State("Mass", 500)
```

```
state =

  State with properties:

                     Alpha: 0
                      Beta: 0
                  AlphaDot: 0
                   BetaDot: 0
                      Mass: 500
                   Inertia: [3×3 table]
           CenterOfGravity: [0 0 0]
          CenterOfPressure: [0 0 0]
               AltitudeMSL: 0
              GroundHeight: 0
                        XN: 0
                        XE: 0
                        XD: 0
                         U: 50
                         V: 0
                         W: 0
                       Phi: 0
                     Theta: 0
                       Psi: 0
                         P: 0
                         Q: 0
                         R: 0
                    Weight: 4905
                AltitudeAGL: 0
                   Airspeed: 50
                GroundSpeed: 50
                 MachNumber: 0.1469
               BodyVelocity: [50 0 0]
             GroundVelocity: [50 0 0]
                         Ur: 50
                         Vr: 0
                         Wr: 0
           FlightPathAngle: 0
               CourseAngle: 0
    InertialToBodyMatrix: [3×3 double]
    BodyToInertialMatrix: [3×3 double]
        BodyToWindMatrix: [3×3 double]
        WindToBodyMatrix: [3×3 double]
         DynamicPressure: 1.5312e+03
             Environment: [1×1 Aero.Aircraft.Environment]
              UnitSystem: "Metric"
```

```
           AngleSystem: "Radians"
       TemperatureSystem: "Kelvin"
           ControlStates: [1×0 Aero.Aircraft.ControlState]
        OutOfRangeAction: "Limit"
        DiagnosticAction: "Warning"
              Properties: [1×1 Aero.Aircraft.Properties]
```

Calculate the forces and moments on the aircraft.

```
[F, M] = forcesAndMoments(aircraft, state)
```

```
F =

           0
           0
        4905


M =

     0
     0
     0
```

Algorithms

The `BodyToStabilityMatrix` transformation is defined by this matrix:

```
BodyToStabilityMatrix =
[cos(Alpha),  0, sin(Alpha)]
[     0,      1,      0    ]
[-sin(Alpha), 0, cos(Alpha)]
```

The `StabilityToBodyMatrix` transformation is the transpose of `BodyToStabilityMatrix` transformation:

```
StabilityToBodyMatrix = BodyToStabilityMatrix'
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `getState` | `setState` | `setupControlStates`

Aero.FixedWing class

Package: `Aero`

Define fixed-wing aircraft

Description

An object of the `Aero.FixedWing` class defines a fixed-wing aircraft. Use this object to model and analyze a fixed-wing aircraft. It contains the static data for the aircraft, such as reference values, coefficients, and deflection angles.

To perform static analysis of fixed-wing aircraft, use this object in conjunction with the `Aero.FixedWing.State` object. The `Aero.FixedWing.State` object contains the aircraft information at a particular aircraft state.

For more information on fixed-wing aircraft definitions, see “More About” on page 4-102.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`fixedWing = Aero.FixedWing` creates a single `Aero.FixedWing` object with default property values.

`fixedWing = Aero.FixedWing(N)` creates an *N*-by-*N* matrix of `Aero.FixedWing` objects with default property values.

`fixedWing = Aero.FixedWing(M,N,P,...)` or `Aero.FixedWing([M N P ...])` create an *M*-by-*N*-by-*P*-by-... array of `Aero.FixedWing` objects with default property values.

`fixedWing = Aero.FixedWing(size(A))` creates an `Aero.FixedWing` object that is the same size as `A` and all `Aero.FixedWing` objects.

`fixedWing = Aero.FixedWing(__,property,propertyValue)` creates an array of `Aero.FixedWing` objects with *`property`*, *`propertyValue`* pairs applied to each of the `Aero.FixedWing` array objects. For a list of properties, see “Properties” on page 4-96.

Input Arguments

`N` — Number of fixed-wing objects

scalar

Number of fixed-wing objects, specified as a scalar.

`M` — Number of fixed-wing objects
scalar

Number of fixed-wing objects, specified as a scalar.

`P` — Number of fixed-wing objects
scalar

Number of fixed-wing objects, specified as a scalar.

`A` — Size of fixed-wing object
scalar

Size of fixed-wing object, specified as a scalar.

Properties

Public Properties

`UnitSystem` — Unit system
`'Metric'` (default) | `'English (kts)'` | `'English (ft/s)'` | scalar string | character vector

Unit system, specified as a scalar string or character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`AngleSystem` — Angle system
`'Radians'` (default) | `'Degrees'`

Angle system, specified as `'Radians'` or `'Degrees'`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`TemperatureSystem` — Temperature system
`'Kelvin'` (default) | `'Celsius'` | `'Rankine'` | `'Fahrenheit'`

Temperature system, specified as `'Kelvin'`, `'Celsius'`, `'Rankine'`, or `'Fahrenheit'`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string` | `char`

`ReferenceArea` — Reference area
0 (default) | scalar numeric

Reference area, specified as a scalar numeric, commonly denoted as `'S'`, in units of:

Units	UnitSystem
meters squared (m^2)	`'Metric'`
feet squared (ft^2)	`'English (kts)'` or `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`ReferenceSpan` — Reference span

0 (default) | scalar numeric

Reference span, specified as a scalar numeric, commonly denoted as `'b'`, in units of:

Units	UnitSystem
meters squared (m)	`'Metric'`
feet squared (ft)	`'English (kts)'` or `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`ReferenceLength` — Reference length

0 (default) | scalar numeric

Reference length, specified as a scalar numeric, commonly denoted as `'c'`, in units of:

Units	UnitSystem
meters squared (m)	`'Metric'`
feet squared (ft)	`'English (kts)'` or `'English (ft/s)'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `double`

`Coefficients` — `Aero.FixedWing.Coefficients` class instance

scalar

`Aero.FixedWing.Coefficients` class instance, specified as a scalar that contains the coefficients defining the fixed-wing aircraft. This object ignores this property if no value is set.

Attributes:

GetAccess	public
SetAccess	public

Data Types: double

DegreesOfFreedom — Degrees of freedom

'6DOF' (default) | '3DOF' | 'PM4' | 'PM6'

Degrees of freedom, specified as a string or character vector.

Degrees of Freedom	Description
'6DOF'	Six degrees of freedom. Describes translational and rotational movement in 3-D space.
'3DOF'	Three degrees of freedom. Describes translational and rotational movement in 2-D space.
'PM4'	Fourth order point-mass. Describes translational movement in 2-D space.
'PM6'	Sixth order point-mass. Describes translational movement in 3-D space.

Attributes:

GetAccess	public
SetAccess	public

Data Types: string | char

Surfaces — Aero.FixedWing.Surface definitions

vector

Aero.FixedWing.Surface definitions, specified as a vector that contains the definitions of the surfaces on the fixed-wing aircraft. The object ignores this property if no value is set.

Attributes:

GetAccess	public
SetAccess	public

Data Types: double

Thrusts — Aero.FixedWing.Thrust definitions

vector

Aero.FixedWing.Thrust definitions, specified as a vector that contains the definitions of the thrust on the fixed-wing aircraft. The object ignores this property if no value is set.

Attributes:

GetAccess	public
SetAccess	public

Data Types:

Protected Properties

`AspectRatio` — Aspect ratio

scalar numeric

Aspect ratio, specified as a scalar numeric, commonly denoted as `'AR'`. This value depends on the values of `ReferencedArea` and `ReferenceSpan`, with this equation:

$$AspectRatio = ReferenceSpan^2/ReferencedArea$$

The object ignores this property if no value is set.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: `double`

Methods

Public Methods

criteriaTable	Construct criteria table for fixed-wing static stability analysis
datcomToFixedWing	Construct fixed-wing aircraft from Digital DATCOM structure
forcesAndMoments	Calculate forces and moments of fixed-wing aircraft
getCoefficient	Get coefficient value for Aero.FixedWing object
getControlStates	Get control states for Aero.FixedWing object
linearize	Return linear state-space model
nonlinearDynamics	Calculate dynamics of fixed-wing aircraft
setCoefficient	Set coefficient value for Aero.FixedWing object
staticStability	Calculate static stability of fixed-wing aircraft
update	Update Aero.FixedWing object

Examples

Create and Use Fixed-Wing Object

Create and set up dynamic behavior and the current state for the fixed-wing object aircraft.

Create a fixed-wing object.

```
aircraft = Aero.FixedWing()
```

```
aircraft =

  FixedWing with properties:
```

```
        ReferenceArea: 0
        ReferenceSpan: 0
      ReferenceLength: 0
         Coefficients: [1×1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1×0 Aero.FixedWing.Surface]
              Thrusts: [1×0 Aero.FixedWing.Thrust]
          AspectRatio: NaN
           UnitSystem: "Metric"
          AngleSystem: "Radians"
    TemperatureSystem: "Kelvin"
           Properties: [1×1 Aero.Aircraft.Properties]
```

To define the aircraft dynamic behavior, set a coefficient for it.

```
aircraft = setCoefficient(aircraft, "CD", "Zero", 0.27)
```

```
aircraft =

  FixedWing with properties:

        ReferenceArea: 0
        ReferenceSpan: 0
      ReferenceLength: 0
         Coefficients: [1×1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1×0 Aero.FixedWing.Surface]
              Thrusts: [1×0 Aero.FixedWing.Thrust]
          AspectRatio: NaN
           UnitSystem: "Metric"
          AngleSystem: "Radians"
    TemperatureSystem: "Kelvin"
           Properties: [1×1 Aero.Aircraft.Properties]
```

Define the aircraft's current state.

```
state = Aero.FixedWing.State("Mass", 500)
```

```
state =

  State with properties:

                  Alpha: 0
                   Beta: 0
                AlphaDot: 0
                 BetaDot: 0
                    Mass: 500
                 Inertia: [3×3 table]
          CenterOfGravity: [0 0 0]
       CenterOfPressure: [0 0 0]
           AltitudeMSL: 0
          GroundHeight: 0
                    XN: 0
                    XE: 0
                    XD: 0
                     U: 50
                     V: 0
                     W: 0
                   Phi: 0
```

```
                    Theta: 0
                      Psi: 0
                        P: 0
                        Q: 0
                        R: 0
                   Weight: 4905
               AltitudeAGL: 0
                  Airspeed: 50
               GroundSpeed: 50
                MachNumber: 0.1469
              BodyVelocity: [50 0 0]
            GroundVelocity: [50 0 0]
                        Ur: 50
                        Vr: 0
                        Wr: 0
          FlightPathAngle: 0
              CourseAngle: 0
    InertialToBodyMatrix: [3×3 double]
    BodyToInertialMatrix: [3×3 double]
        BodyToWindMatrix: [3×3 double]
        WindToBodyMatrix: [3×3 double]
         DynamicPressure: 1.5312e+03
             Environment: [1×1 Aero.Aircraft.Environment]
              UnitSystem: "Metric"
             AngleSystem: "Radians"
       TemperatureSystem: "Kelvin"
           ControlStates: [1×0 Aero.Aircraft.ControlState]
        OutOfRangeAction: "Limit"
        DiagnosticAction: "Warning"
              Properties: [1×1 Aero.Aircraft.Properties]
```

Calculate the forces and moments on the aircraft.

```
[F, M] = forcesAndMoments(aircraft, state)
```

```
F =

           0
           0
        4905


M =

     0
     0
     0
```

Limitations

You cannot subclass `Aero.FixedWing`.

More About

Fixed-Wing Definitions

The `FixedWing` object holds the main definition of a fixed-wing aircraft. The object has a main set of body coefficients defined by these tables:

Body Coefficients in Wind Frame

Body Coefficients	Wind Frame								
	Zero	U	Alpha	AlphaRate	Theta	Beta	BetaRate	Phi	Psi
CD									
CY									
CL									
CI									
Cm									
Cn									

Body Coefficients in Body Frame

Body Coefficients	Wind Frame								
	Zero	U	Alpha	AlphaRate	Theta	Beta	BetaRate	Phi	Psi
CX									
CY									
CZ									
CI									
Cm									
Cn									

Version History

Introduced in R2021a

See Also

`Aero.FixedWing.Coefficient` | `Aero.FixedWing.Surface` | `Aero.FixedWing.Thrust` | `Aero.FixedWing.State` | `getCoefficient` | `setCoefficient` | `Simulink.LookupTable`

Topics

“Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121
“Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106

“Customize Fixed-Wing Aircraft with Additional Aircraft States” on page 5-113

Aero.FixedWing.Surface class

Package: `Aero`

Define aerodynamic and control surfaces on `Aero.FixedWing` aircraft

Description

`Aero.FixedWing.Coefficient` defines the dynamic and control surfaces on an `Aero.FixedWing` aircraft.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`fixedWingSurface = Aero.FixedWing.Surface` creates a single `Aero.FixedWing.Surface` object with default property values.

`fixedWingSurface = Aero.FixedWing.Surface(N)` creates an *N*-by-*N* matrix of `Aero.FixedWing.Surface` objects with default property values.

`fixedWingSurface = Aero.FixedWing.Surface(M,N,P,...)` or `Aero.FixedWing.Surface([M N P ...])` creates an *M*-by-*N*-by-*P*-by-... array of `Aero.FixedWing.Surface` objects with default property values.

`fixedWingSurface = Aero.FixedWing.Surface(size(A))` creates an `Aero.FixedWing.Surface` object of the same size as `A` and all `Aero.FixedWing.Surface` objects.

`fixedWingSurface = Aero.FixedWing.Surface(__,property,propertyValue)` creates an array of `Aero.FixedWing.Surface` objects with *`property`*, *`propertyValue`* pairs applied to each of the `Aero.FixedWing.Surface` array objects. For a list of properties, see “Properties” on page 4-104.

Properties

Public Properties

`Surfaces` — `Aero.FixedWing.Surface` objects
vector

`Aero.FixedWing.Surface` objects providing nested control surfaces, specified as a vector.

Attributes:

GetAccess	public
SetAccess	public

`Coefficients` — `Aero.FixedWing.Coefficients` objects

scalar

`Aero.FixedWing.Coefficients` objects that define the control surface, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

`MaximumValue` — Maximum value of control surfaces

infinity (default) | scalar numeric

Maximum value of control surfaces, specified as a scalar numeric.

Dependencies

If `Symmetry` is set to `Asymmetric`, then this value applies to both control variables.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

`MinimumValue` — Minimum value of control surface

negative infinity (default) | scalar numeric

Minimum value of control surface, specified as a scalar numeric.

Dependencies

If `Symmetry` is set to `Asymmetric`, then this value applies to both control variables.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

`Controllable` — Controllable control surface

`off` (default) | `on`

Controllable control surface specified as `on` or `off`. To control the control surface, set this property to `on`. Otherwise, set this property to `off`.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

Symmetry — Symmetry of control surface
Symmetric (default) | Asymmetric

Symmetry of the control surface, specified as Symmetric or Asymmetric.

The Asymmetric option creates two control variables, denoted by the name on the properties and appended by _1 and _2. These control variables can be independently controlled but also produce an effective control variable specified by the name on the properties. You cannot set this effective control variable. This equation defines the control variable:

name = (*name*_1-*name*_2)/2.

You cannot set this effective control variable.

Attributes:

GetAccess	public
SetAccess	public

Data Types: char | string

Properties — Aero.Aircraft.Properties object
scalar

Aero.Aircraft.Properties object, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

Data Types: double

Protected Properties

ControlVariables — Control variable names
vector

Control variable names, specified as a vector. This property depends on Properties.Name, Controllable, and Symmetry.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: char | string

Methods

Public Methods

getCoefficient	Get coefficient for fixed-wing surface object
getControlStates	Get control states for Aero.FixedWing.Surface object
setCoefficient	Set coefficient values for Aero.FixedWing.Surface object
update	Update Aero.FixedWing.Surface object

Examples

Create and Use Fixed-Wing Object

Create and set up dynamic behavior and the current state for the fixed-wing object aircraft.

Create a fixed-wing object.

```
aircraft = Aero.FixedWing()
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

To define the aircraft dynamic behavior, set a coefficient for it.

```
aircraft = setCoefficient(aircraft, "CD", "Zero", 0.27)
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

Define the aircraft's current state.

```
state = Aero.FixedWing.State("Mass", 500)
```

```
state =

  State with properties:

                  Alpha: 0
```

```
                  Beta: 0
              AlphaDot: 0
               BetaDot: 0
                  Mass: 500
               Inertia: [3×3 table]
       CenterOfGravity: [0 0 0]
      CenterOfPressure: [0 0 0]
           AltitudeMSL: 0
          GroundHeight: 0
                    XN: 0
                    XE: 0
                    XD: 0
                     U: 50
                     V: 0
                     W: 0
                   Phi: 0
                 Theta: 0
                   Psi: 0
                     P: 0
                     Q: 0
                     R: 0
                Weight: 4905
            AltitudeAGL: 0
               Airspeed: 50
            GroundSpeed: 50
             MachNumber: 0.1469
           BodyVelocity: [50 0 0]
         GroundVelocity: [50 0 0]
                    Ur: 50
                    Vr: 0
                    Wr: 0
        FlightPathAngle: 0
            CourseAngle: 0
  InertialToBodyMatrix: [3×3 double]
  BodyToInertialMatrix: [3×3 double]
      BodyToWindMatrix: [3×3 double]
      WindToBodyMatrix: [3×3 double]
       DynamicPressure: 1.5312e+03
           Environment: [1×1 Aero.Aircraft.Environment]
            UnitSystem: "Metric"
           AngleSystem: "Radians"
     TemperatureSystem: "Kelvin"
        ControlStates: [1×0 Aero.Aircraft.ControlState]
      OutOfRangeAction: "Limit"
      DiagnosticAction: "Warning"
            Properties: [1×1 Aero.Aircraft.Properties]
```

Calculate the forces and moments on the aircraft.

```
[F, M] = forcesAndMoments(aircraft, state)
```

```
F =

          0
          0
       4905
```

```
M =

     0
     0
     0
```

Limitations

You cannot subclass `Aero.FixedWing.Surface`.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Aero.FixedWing.Coefficient` | `Aero.FixedWing.Thrust`

Aero.FixedWing.Thrust class

Package: `Aero`

Define thrust vector on fixed-wing aircraft

Description

`Aero.FixedWing.Thrust` creates an `Aero.FixedWing` thrust vector that describes the thrust of an aircraft.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`fixedWingThrust = Aero.FixedWing.Thrust` creates a single `Aero.FixedWing.Thrust` object with default property values.

`fixedWingThrust = Aero.FixedWing.Thrust(N)` creates an *N*-by-*N* matrix of `Aero.FixedWing.Thrust` objects with default property values.

`fixedWingThrust = Aero.FixedWing.Thrust(M,N,P,...)` or `Aero.FixedWing.Thrust([M N P ...])` creates an *M*-by-*N*-by-*P*-by-... array of `Aero.FixedWing.Thrust` objects with default property values.

`fixedWingThrust = Aero.FixedWing.Thrust(size(A))` creates an `Aero.FixedWing.Thrust` object that is the same size as `A` and all `Aero.FixedWing.Thrust` objects.

`fixedWing.Thrust = Aero.FixedWing.Thrust(__,property,propertyValue)` creates an array of `Aero.FixedWing.Thrust` objects with *`property`*, *`propertyValue`* pairs applied to each of the `Aero.FixedWing.Thrust` array objects. For a list of properties, see “Properties” on page 4-111.

Input Arguments

`N` — Number of fixed-wing thrust objects

scalar

Number of fixed-wing thrust objects, specified as a scalar.

`M` — Number of fixed-wing thrust objects

scalar

Number of fixed-wing thrust objects, specified as a scalar.

`P` — Number of fixed-wing thrust objects
scalar

Number of fixed-wing thrust objects, specified as a scalar.

`A` — Size of fixed-wing thrust object
scalar

Size of fixed-wing thrust object, specified as a scalar.

Properties

Public Properties

`Coefficients` — `Aero.FixedWing.Coefficients` object
scalar

`Aero.FixedWing.Coefficients` object, specified as a scalar, that defines the thrust vector.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`MaximumValue` — Maximum thrust value
`1` (default) | scalar numeric

Maximum thrust value, specified as a scalar numeric.

Dependencies

If `Symmetry` is set to `Asymmetric`, then this value applies to both control variables.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `double`

`MinimumValue` — Minimum thrust value
`0` (default) | scalar numeric

Minimum thrust value, specified as a scalar numeric.

Dependencies

If `Symmetry` is set to `Asymmetric`, then this value applies to both control variables.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `double`

`Controllable` — Controllable thrust value
`on` (default) | `off`

Controllable thrust value, specified as `on` or `off`. To control the thrust value, set this property to `on`. Otherwise, set this property to `off`.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

`Symmetry` — Symmetry of thrust control
`Symmetric` (default) | `Asymmetric`

Symmetry of the thrust control, specified as `Symmetric` or `Asymmetric`.

The `Asymmetric` option creates two control variables, denoted by the name on the properties and appended by `_1` and `_2`. These control variables can be independently controlled, but also produce an effective control variable specified by the name on the properties. You cannot set this effective control variable. This equation defines the control variable:

name = (*name*_1-*name*_2)/2.

You cannot set this effective control variable.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `char` | `string`

`Properties` — `Aero.Aircraft.Properties` object
scalar

`Aero.Aircraft.Properties` object, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `double`

Protected Properties

`ControlVariables` — Control variable names
vector

Control variable names, specified as a vector. This property depends on `Properties.Name`, `Controllable`, and `Symmetry`.

Attributes:

GetAccess	Restricts access
SetAccess	protected

Data Types: `char` | `string`

Methods

Public Methods

getCoefficient	Get coefficient for fixed-wing thrust object
getControlStates	Get control states for Aero.FixedWing.Thrust object
setCoefficient	Set coefficient values for Aero.FixedWing.Thrust object
update	Update Aero.FixedWing.Thrust object

Examples

Create and Use Fixed-Wing Object

Create and set up dynamic behavior and the current state for the fixed-wing object aircraft.

Create a fixed-wing object:

```
aircraft = Aero.FixedWing()
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

To define the aircraft dynamic behavior, set a coefficient for it:

```
aircraft = setCoefficient(aircraft, "CD", "Zero", 0.27)
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 0
       ReferenceSpan: 0
     ReferenceLength: 0
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: NaN
          UnitSystem: "Metric"
         AngleSystem: "Radians"
   TemperatureSystem: "Kelvin"
          Properties: [1×1 Aero.Aircraft.Properties]
```

Define aircraft current state:

```
state = Aero.FixedWing.State("Mass", 500)
```

```
state =

  State with properties:

                  Alpha: 0
                   Beta: 0
               AlphaDot: 0
                BetaDot: 0
                   Mass: 500
                Inertia: [3×3 table]
         CenterOfGravity: [0 0 0]
        CenterOfPressure: [0 0 0]
             AltitudeMSL: 0
            GroundHeight: 0
                      XN: 0
                      XE: 0
                      XD: 0
                       U: 50
                       V: 0
                       W: 0
                     Phi: 0
                   Theta: 0
                     Psi: 0
                       P: 0
                       Q: 0
                       R: 0
                  Weight: 4905
              AltitudeAGL: 0
                 Airspeed: 50
              GroundSpeed: 50
               MachNumber: 0.1469
             BodyVelocity: [50 0 0]
           GroundVelocity: [50 0 0]
                      Ur: 50
                      Vr: 0
                      Wr: 0
          FlightPathAngle: 0
              CourseAngle: 0
    InertialToBodyMatrix: [3×3 double]
    BodyToInertialMatrix: [3×3 double]
        BodyToWindMatrix: [3×3 double]
        WindToBodyMatrix: [3×3 double]
         DynamicPressure: 1.5312e+03
             Environment: [1×1 Aero.Aircraft.Environment]
              UnitSystem: "Metric"
             AngleSystem: "Radians"
       TemperatureSystem: "Kelvin"
           ControlStates: [1×0 Aero.Aircraft.ControlState]
        OutOfRangeAction: "Limit"
        DiagnosticAction: "Warning"
              Properties: [1×1 Aero.Aircraft.Properties]
```

Calculate the forces and moments on the aircraft:

```
[F, M] = forcesAndMoments(aircraft, state)
```

```
F =

           0
           0
        4905


M =

     0
     0
     0
```

Limitations

You cannot subclass `Aero.FixedWing.Thrust`.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Aero.FixedWing.Coefficient`

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 5-17

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.
`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.
`wait`	Wait until animation is done playing

Property Summary

Properties	Description
`TimeSeriesSource`	Specify variable that contains the time series data.

Properties	Description
`TimeSeriesSourceType`	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.
`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'`.
`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.
`TStart`	Specify start time as a double.
`TFinal`	Specify end time as a double.

Properties	Description
`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 `timeseries` data with six values per time: `lat lon alt phi theta psi` The values are resampled.
`'Timetable'`	MATLAB `timetable` 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
```

Version History

Introduced in R2007a

See Also

`fganimation` | `generaterunscript` | `play`

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 web site 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 web site `https://maia.usno.navy.mil/ser7/finals2000A.data`.

```
aeroReadIERSData(pwd)
```

Create File from Specified Web Site

Create the Earth orientation file for the current day, in the current folder, using data from the alternate web site `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 web site or Earth orientation data file

`https://maia.usno.navy.mil/ser7/finals2000A.data` (default) | web site address | file name

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

Note If you receive an error message while accessing the default site, use one of these alternate sites:

- `https://datacenter.iers.org/data/latestVersion/10_FINALS.DATA_IAU2000_V2013_0110.txt`
- `ftp://ftp.iers.org/products/eop/rapid/standard/finals2000A.data`

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/finals2000A.data`:

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

Column	Description
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*
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

Version History

Introduced in R2017b

See Also

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

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	
`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.

Version History

Introduced in R2007a

See Also

`read`

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* coordinate axes. The order of application of the rotation is *z*-*y*-*x* (yaw, pitch, and roll (*Y*-*P*-*R*)). 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
`findstartstoptimes`	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

Property	Description	Values
`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
`TimeseriesSourceType`	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 `timeseries` data with six values per time: `lat lon alt phi theta psi` The values are resampled.
`'Timetable'`	MATLAB `timetable` 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'`.

Version History

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

Version History

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
`VirtualRealityAnimation`	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.
`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.
`wait`	Wait until animation is done playing

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. Time source must be a `timeseries` or `timetable` object.	double
`TFinal`	Specify end time, in seconds. Time source must be a `timeseries` or `timetable` object.	double
`TCurrent`	Specify current time, in seconds.	double
`FramesPerSecond`	Specify rate, in frames per second.	double

Property	Description	Values
`ShowSaveWarning`	Specify save warning display setting.	double • `0` — No warning is displayed. • Non-zero — Warning is displayed.
`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
`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'`

Property	Description	Values
`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

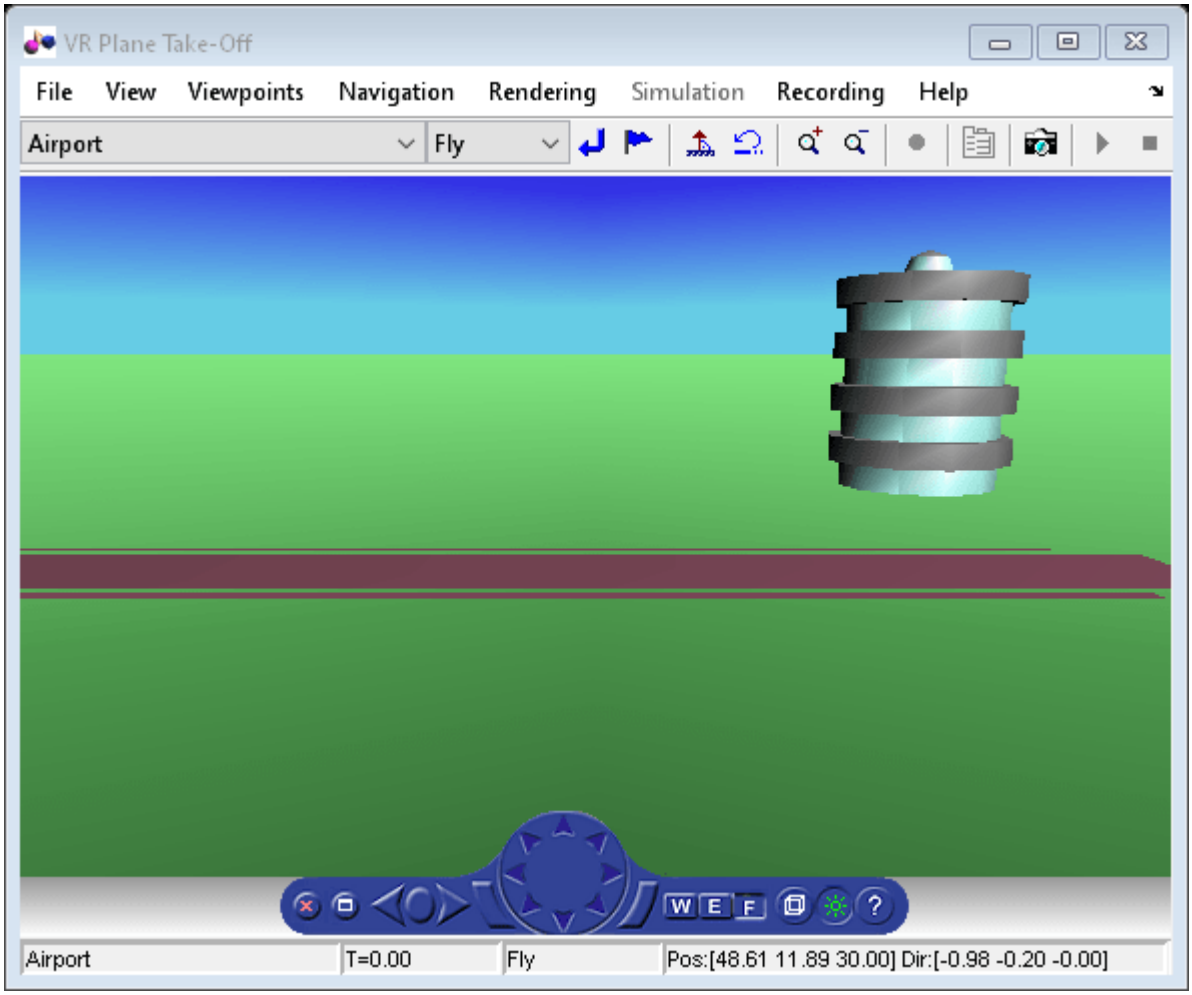
Record Virtual Reality Animation Object Simulation

This example shows how to

- Record 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 = '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();

% Wait for the animation to stop running before modifying the object.

h.wait();

% 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 Animation for Four Seconds

Simulate flight data for four seconds.

Use the data to 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 the `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 = createBody(h,'pa24-250_orange.ac','Ac3d');
```

Load simulated flight trajectory data (`simdata`).

```
load simdata;
```

Set the time series data for the body.

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

Create a figure object for the animation object.

```
show(h);
```

Set up recording properties.

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

Play the animation from `TStart` to `TFinal`.

```
h.TSTart = 1;
h.TFinal = 5;
play(h);
```

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.

Wait

Wait for the animation to stop playing before the modifying the object.

```
wait(h);
```

Disable recording to preserve the file.

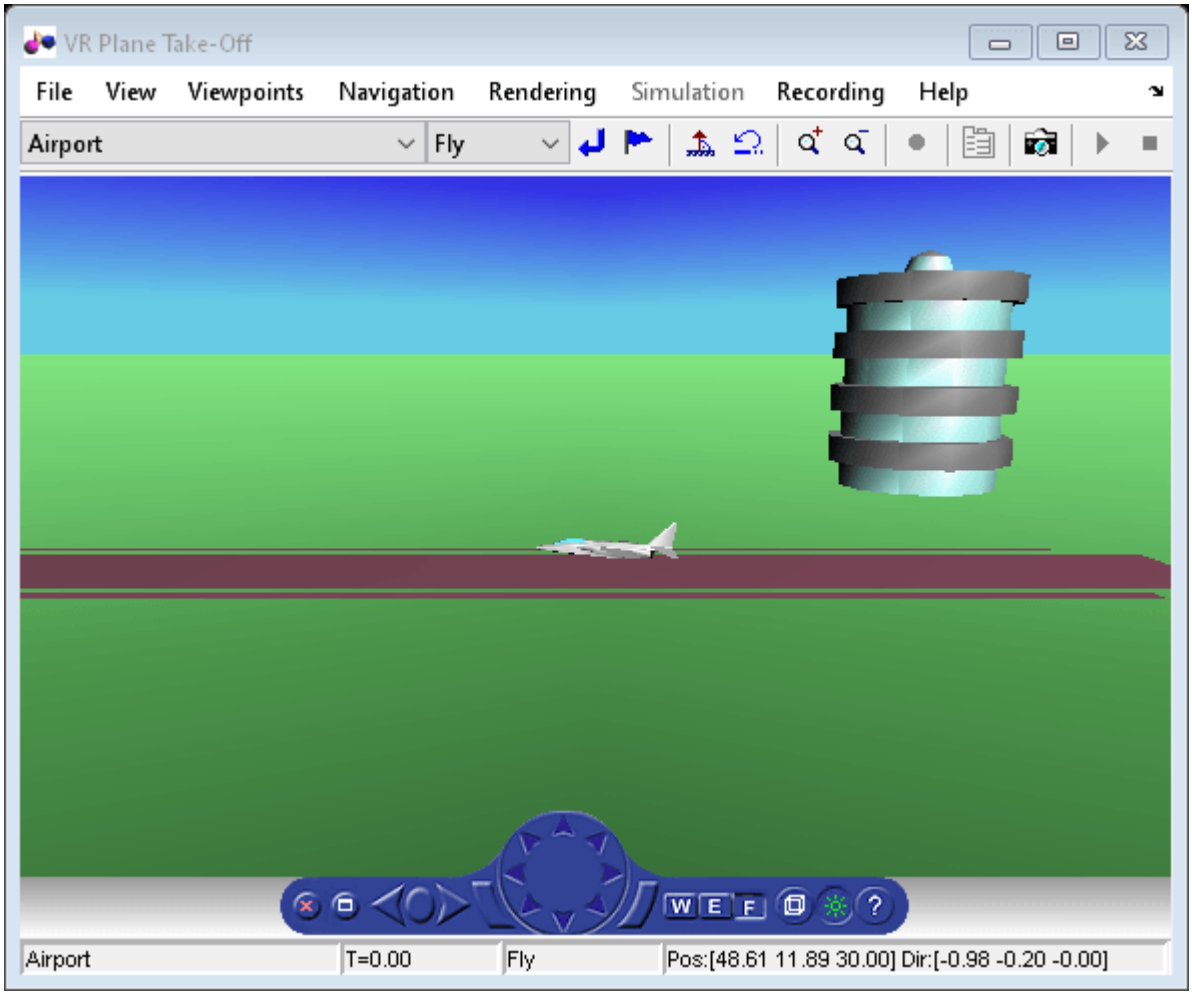
```
h.VideoRecord = 'off';
```

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 = '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();

% Wait for the animation to stop running before modifying the object.

h.wait();

% 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';
```



Version History

Introduced in R2007b

See Also

`Aero.Node` | `Aero.Viewpoint`

Topics

“Visualize Aircraft Takeoff via Virtual Reality Animation Object” on page 2-32
“Aero.VirtualRealityAnimation Objects” on page 2-31

aircraftEnvironment

Create aircraft environment

Syntax

```
environment = aircraftEnvironment(atmosphere,height)

environment = aircraftEnvironment(aircraft,atmosphere,height)

environment = aircraftEnvironment( ___ ,Name=Value)
```

Description

`environment = aircraftEnvironment(atmosphere,height)` creates an aircraft environment object, `environment`, specified by the atmospheric model `atmosphere` that is above sea level, `height`, for a default aircraft.

`environment = aircraftEnvironment(aircraft,atmosphere,height)` creates an aircraft environment object, `environment`, specified by the atmospheric model, `atmosphere`, that is above sea level, `height` for a specified aircraft, `aircraft`. The function uses the aircraft unit system.

`environment = aircraftEnvironment( ___ ,Name=Value)` creates an aircraft environment, `environment`. Specify one or more `Name=Value` arguments after any of the input argument combinations in the previous syntaxes.

Examples

Create Aircraft Environment Object Using Fixed-Wing Aircraft

Create an aircraft environment object using a fixed-wing aircraft and English units at 20,000 feet.

```
aircraft = fixedWingAircraft("myplane","UnitSystem","English (ft/s)");
environment = aircraftEnvironment(aircraft,"ISA",20000)
```

```
environment =

  Environment with properties:

    WindVelocity: [0 0 0]
         Density: 0.0013
     Temperature: 248.5260
        Pressure: 972.4941
    SpeedOfSound: 1.0369e+03
         Gravity: 32.1850
      Properties: [1×1 Aero.Aircraft.Properties]
```

Create Aircraft Environment Object with ISA Model

Create an aircraft environment object using the ISA model.

```
environment = aircraftEnvironment("ISA",1000)

environment =

  Environment with properties:

    WindVelocity: [0 0 0]
         Density: 1.1116
     Temperature: 281.6500
        Pressure: 8.9875e+04
    SpeedOfSound: 336.4341
         Gravity: 9.8100
      Properties: [1×1 Aero.Aircraft.Properties]
```

Create Aircraft Environment Object with COESA Model

Create an aircraft environment object using the COESA model at 0, 100, and 1,000 meters.

```
environment = aircraftEnvironment("COESA",[0,100,1000])

environment =

  1×3 Environment array with properties:

    WindVelocity
    Density
    Temperature
    Pressure
    SpeedOfSound
    Gravity
    Properties
```

Create Aircraft Environment Object with ISA Model Using English Units

Create an aircraft environment object using English units at 500 feet.

```
environment = aircraftEnvironment("ISA",500,"UnitSystem","English (ft/s)")

environment =

  Environment with properties:

    WindVelocity: [0 0 0]
         Density: 0.0023
     Temperature: 287.1594
        Pressure: 2.0783e+03
    SpeedOfSound: 1.1145e+03
         Gravity: 32.1850
      Properties: [1×1 Aero.Aircraft.Properties]
```

Input Arguments

`aircraft` — Fixed-wing aircraft

scalar (default)

Fixed-wing aircraft, specified as a scalar.

Data Types: `string`

`atmosphere` — Atmospheric model
`"ISA"` | `"COESA"`

Atmospheric model to calculate the aircraft environment, specified as `"ISA"` or `"COESA"`.

`height` — Height above sea level
numeric matrix

Height above sea level, specified as a numeric matrix.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"UnitSystem","English (ft/s)"`

`UnitSystem` — Unit system
`'Metric'` (default) | `"English (kts)"`

Unit system, specified as a string based on these units:

Unit	Unit System
Meters per second (m/s)	`"Metric"`
Feet per second (ft/s)	`"English (kts)"`
Knots (kts)	`"English (ft/s)"`

Example: `"AngleSystem","English (kts)"`

`AngleSystem` — Angle system
`"Radians"` (default) | `"Degrees"`

Angle system, specified as `"Radians"` or `"Degrees"`.

Example: `"AngleSystem","Degrees"`

`TemperatureSystem` — Temperature system
`"Kelvin"` (default) | `"Celsius"` | `"Rankine"` | `"Fahrenheit"`

Temperature system, specified as `"Kelvin"`, `"Celsius"`, `"Rankine"`, or `"Fahrenheit"`.

Example: `"TemperatureSystem","Rankine"`

Output Arguments

`environment` — `Aero.Aircraft.Environment` objects
matrix

`Aero.Aircraft.Environment` objects, returned as a matrix of the same size as `height`.

Version History

Introduced in R2021b

See Also

`aircraftProperties` | `fixedWingAircraft` | `fixedWingCoefficient` | `fixedWingState` | `fixedWingSurface` | `fixedWingThrust` | `atmoscira` | `atmoscoesa` | `atmosisa`

aircraftProperties

Create properties to define and manage aircraft

Syntax

```
properties = aircraftProperties( )
properties = aircraftProperties(name)
properties = aircraftProperties(name,description)
properties = aircraftProperties(name,description,type)
prooperties = aircraftProperties(name,description,type,version)
```

Description

`properties = aircraftProperties( )` returns an `Aero.Aircraft.Properties` object with default values for all properties.

`properties = aircraftProperties(name)` returns an `Aero.Aircraft.Properties` object with the specified name `name`.

`properties = aircraftProperties(name,description)` returns an `Aero.Aircraft.Properties` object with the specified description `description`.

`properties = aircraftProperties(name,description,type)` returns an `Aero.Aircraft.Properties` object with the specified type `type`.

`prooperties = aircraftProperties(name,description,type,version)` returns an `Aero.Aircraft.Properties` object with the specified version `version`.

Examples

Create Aircraft Properties Object

Create an aircraft properties object.

```
props = aircraftProperties()
```

```
props =

  Properties with properties:

          Name: ""
   Description: ""
          Type: ""
       Version: ""
```

Create Aircraft Properties Object with Name

Create an aircraft properties object and specify the name.

```
props = aircraftProperties("MyPlane")

props =

  Properties with properties:

          Name: "MyPlane"
    Description: ""
          Type: ""
       Version: ""
```

Create Aircraft Properties Object with Name, Description, Type, and Version

Create an aircraft properties object and specify the name, description, type, and version.

```
props = aircraftProperties("MyPlane","This is a plane","plane","1.0")

props =

  Properties with properties:

          Name: "MyPlane"
    Description: "This is a plane"
          Type: "plane"
       Version: "1.0"
```

Input Arguments

`name` — Object name

" " (default) | scalar string | character vector

Object name, specified as a scalar string or character vector.

Data Types: `char` | `string`

`description` — Object description

" " (default) | scalar string | character vector

Object description, specified as a scalar string or character vector.

Data Types: `char` | `string`

`type` — Object type

" " (default) | scalar string | character vector

Object type, specified as a scalar string or character vector.

Data Types: `char` | `string`

`version` — Object version

" " (default) | scalar string | character vector

Object version, specified as a scalar string or character vector.

Data Types: `char` | `string`

Output Arguments

properties — Aero.Aircraft.Properties object
Aero.Aircraft.Properties object

Aero.Aircraft.Properties object, returned as an Aero.Aircraft.Properties object.

Version History

Introduced in R2021b

See Also

aircraftEnvironment | fixedWingAircraft | fixedWingCoefficient | fixedWingState | fixedWingSurface | fixedWingThrust

airspeed

Airspeed from velocity

Syntax

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

Description

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

Examples

Determine Airspeed for One Velocity Array

Determine the airspeed for one velocity vector.

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

```
as = 91.4538
```

Determine Airspeed for Multiple Velocity Arrays

Determine the airspeed for multiple velocity vectors.

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

```
as = 2×1

   54.1849
    5.4083
```

Input Arguments

`velocities` — Cartesian velocity vectors

m-by-3 array | vector

Cartesian velocity vectors, specified as an *m*-by-3 array.

Data Types: `double`

Output Arguments

`airspeed` — Airspeed

scalar | vector

Airspeed, returned as a scalar or array of *m* airspeeds.

Version History

Introduced in R2006b

See Also

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

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) | 1-by-n string array | 1-by-n cell array | n-by-3 array of RGB triplets | hexadecimal color code | ...

Scale colors, specified as one of the following arrays:

- A 1-by-n string array of color options, such as `["blue" "green" "red"]`.
- An n-by-3 array of RGB triplets, such as `[0 0 1;1 1 0]`.
- A 1-by-n cell array containing RGB triplets, hexadecimal color codes, or named color options. For example, `{'#EDB120','#7E2F8E','#77AC30'}`.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- 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]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	"#A2142F"	

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” on page 5-99).

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) | on/off logical value

Visibility of the airspeed indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`. 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.

`ContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object created using the `uicontextmenu` function. Use this property to display a context menu when you right-click on a component.

`Enable` — Operational state of airspeed indicator

`'on'` (default) | on/off logical value

Operational state of airspeed indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- 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
`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 = uigridlayout([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 “Callbacks in App Designer”.

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 “Callbacks in App Designer”.

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) | on/off logical value

Callback interruption, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

This 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.

MATLAB determines callback interruption behavior whenever it executes a command that processes the callback queue. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines if the interruption occurs:

- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the object that owns the interrupting callback determines if the interrupting callback is discarded or added to the callback queue.

- If the value of `Interruptible` is `'on'`, then the interruption occurs. The next time MATLAB processes the callback queue, it stops the execution of the running callback and executes the interrupting callback. After the interrupting callback completes, MATLAB then resumes executing the running callback.

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.
- If the interrupting callback is owned by a `Timer` object, then the callback executes according to schedule regardless of the `Interruptible` property value.

Note 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.

The `BusyAction` property determines callback queuing behavior only when both of these conditions are met:

- The running callback contains a command that processes the callback queue, such as `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause`.
- The value of the `Interruptible` property of the object that owns the running callback is `'off'`.

Under these conditions, the `BusyAction` property of the object that owns the interrupting callback determines how MATLAB handles the interrupting callback. 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

on/off logical value

This property is read-only.

Deletion status, returned as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

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

`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”.

Version History

Introduced in R2018b

See Also

`uiaeroairspeed`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

alphabeta

Compute 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, `velocities`, and the body.

Examples

Determine Incidence and Sideslip Angles for Velocity for One Array

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 Incidence and Sideslip Angles for Velocity for Two Arrays

This example shows how to determine the incidence and sideslip angles for velocity for two arrays.

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

```
alpha = 2×1

    0.1194
    0.3805


beta = 2×1

    0.3780
    0.0926
```

Input Arguments

`velocities` — Velocity vector

m-by-3 array

Velocity vector in body axes, specified as an *m*-by-3 array.

Example: `[84.3905 33.7562 10.1269]`

Data Types: `double`

Output Arguments

`incidence` — Incidence angle

scalar

Incidence angle, returned as a scalar, in radians.

`sideslip` — Sideslip angle

scalar

Sideslip angle, returned as a scalar, in radians.

Version History

Introduced in R2006b

See Also

`airspeed` | `dcmbody2stability` | `machnumber`

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) | on/off logical value

Visibility of the altimeter, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`. 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.

`ContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object created using the `uicontextmenu` function. Use this property to display a context menu when you right-click on a component.

`Enable` — Operational state of altimeter

`'on'` (default) | on/off logical value

Operational state of altimeter, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- 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

Element	Description
`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 “Callbacks in App Designer”.

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 “Callbacks in App Designer”.

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) | on/off logical value

Callback interruption, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value

of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

This 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.

MATLAB determines callback interruption behavior whenever it executes a command that processes the callback queue. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines if the interruption occurs:

- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the object that owns the interrupting callback determines if the interrupting callback is discarded or added to the callback queue.
- If the value of `Interruptible` is `'on'`, then the interruption occurs. The next time MATLAB processes the callback queue, it stops the execution of the running callback and executes the interrupting callback. After the interrupting callback completes, MATLAB then resumes executing the running callback.

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.
- If the interrupting callback is owned by a `Timer` object, then the callback executes according to schedule regardless of the `Interruptible` property value.

Note 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.

The `BusyAction` property determines callback queuing behavior only when both of these conditions are met:

- The running callback contains a command that processes the callback queue, such as `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause`.
- The value of the `Interruptible` property of the object that owns the running callback is `'off'`.

Under these conditions, the `BusyAction` property of the object that owns the interrupting callback determines how MATLAB handles the interrupting callback. 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

on/off logical value

This property is read-only.

Deletion status, returned as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

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”.

Version History

Introduced in R2018b

See Also

`uiaeroaltimeter`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

altitudeEnvelopeContour

Draw altitude envelope contour plot

Syntax

```
altitudeEnvelopeContour(loadfactor)
altitudeEnvelopeContour(airspeed,altitude,loadfactor)

altitudeEnvelopeContour( ___ ,levels)
altitudeEnvelopeContour( ___ ,LineSpec)
altitudeEnvelopeContour( ___ ,Name,Value)
altitudeEnvelopeContour(ax, ___ )

[c,h,bline] = altitudeEnvelopeContour( ___ )
```

Description

Draw Contour Plots with Load Factors and Meshes

`altitudeEnvelopeContour(loadfactor)` draws a contour plot of the `loadfactor` matrix in the *x*-*y* plane. This function is based on the MATLAB `contour` function. The *x*-coordinates of the vertices correspond to the column indices of `loadfactor` and the *y*-coordinates correspond to the row indices of `loadfactor`. The contour automatically chooses the contour levels.

`altitudeEnvelopeContour(airspeed,altitude,loadfactor)` draws a contour plot of the `loadfactor` matrix using vertices from the mesh that `airspeed` and `altitude` define.

Draw Contour Plots with Customizations

`altitudeEnvelopeContour( ___ ,levels)` plots an altitude envelope contour specified by the desired levels `levels`.

`altitudeEnvelopeContour( ___ ,LineSpec)` plots an altitude envelope contour specified by the desired line spec `LineSpec`.

`altitudeEnvelopeContour( ___ ,Name,Value)` plots an altitude envelope contour specified by one or more `Name,Value` arguments.

`altitudeEnvelopeContour(ax, ___ )` draws an altitude contour plot onto the axes *ax*.

Return Matrix, Contour Object, and Boundary Line Object

`[c,h,bline] = altitudeEnvelopeContour( ___ )` returns contour matrix `c`, a contour object `h`, and a vector of `boundaryline` objects `b`. To label the plot, use the `c` and `bs` arguments as inputs to the `clabel` function when using the `LabelSpacing` property.

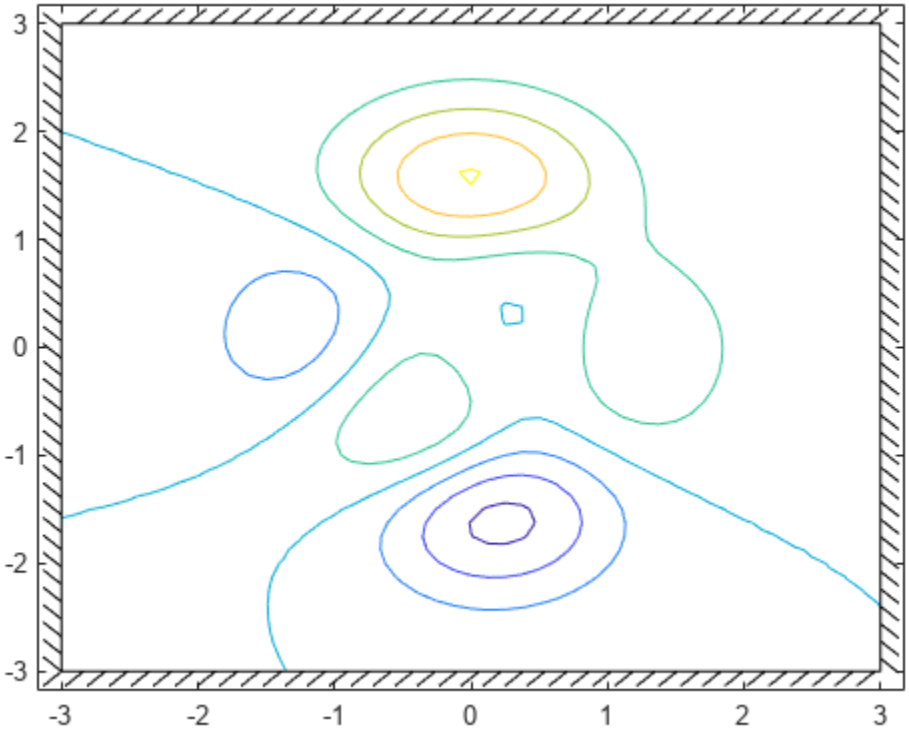
Tip For more information on the contour matrix, see the Contour Properties property for contour objects.

Examples

Plot Altitude Envelope Contour

Plot an altitude envelope contour with default levels and boundaries.

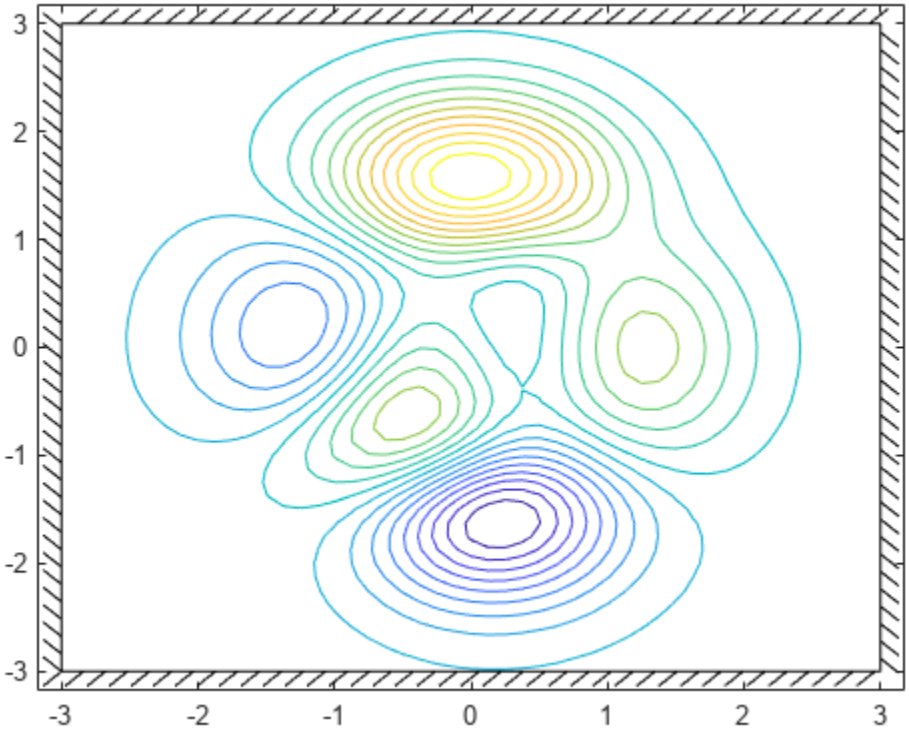
```
[speed,alt,loadfactor] = peaks();
altitudeEnvelopeContour(speed,alt,loadfactor)
```



Plot Altitude Envelope Contour with Levels

Plot an altitude envelope contour with 20 levels and default boundaries.

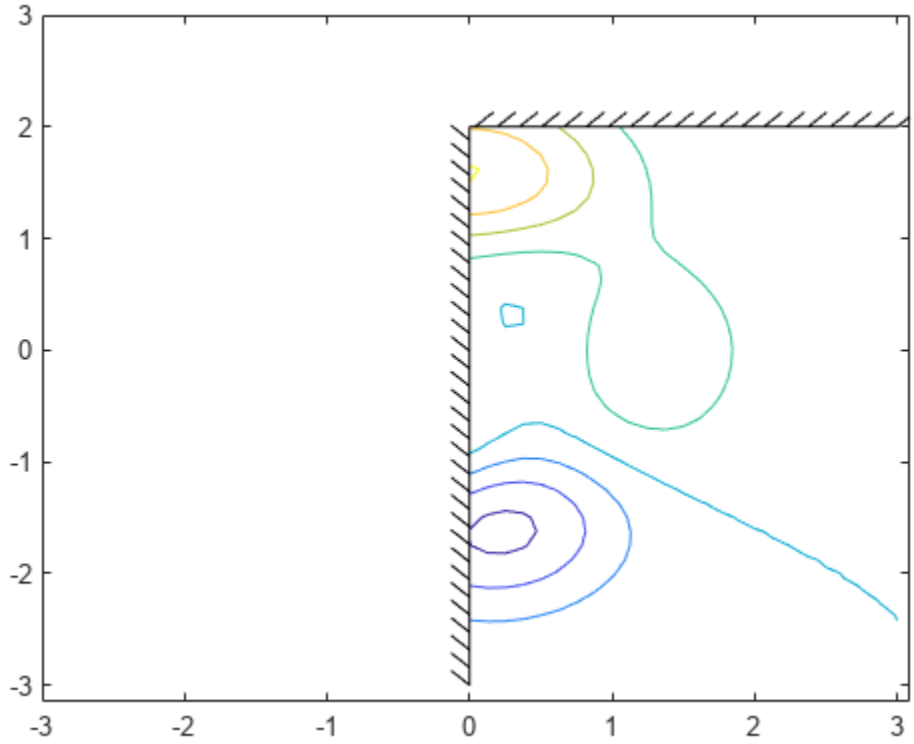
```
[speed,alt,loadfactor] = peaks();
altitudeEnvelopeContour(speed,alt,loadfactor,20)
```



Plot Altitude Envelope Contour with Constant Boundary Lines

Plot an altitude envelope contour with constant boundary lines, and turn off boundary line intersection clipping.

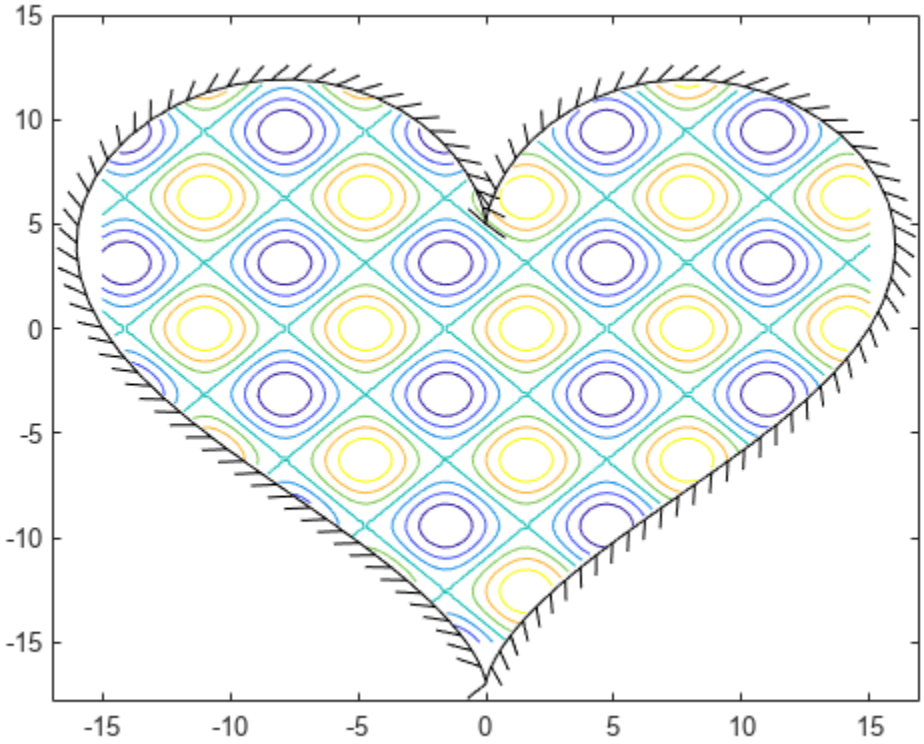
```
[speed,alt,loadfactor] = peaks();
altitudeEnvelopeContour(speed,alt,loadfactor,...
"MinimumSpeed",0,"MaximumAltitude",2,"ResolveBoundary","off")
```



Plot Altitude Envelope Contour with Custom Boundary Data

Plot an altitude envelope contour with custom boundary data. Return line objects and boundary line objects in *c*, *h*, and *b*.

```
x = linspace(-15,15);
y = linspace(-15,15);
[X,Y] = meshgrid(x,y);
Z = sin(X)+cos(Y);
t = linspace(2*pi,0);
boundaryX = 16*sin(t).^3;
boundaryY = 13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t);
[c,h,b]=altitudeEnvelopeContour(X,Y,Z,...
"BoundaryXData",boundaryX,"BoundaryYData",boundaryY);
```



Input Arguments

`loadfactor` — Load factor for each airspeed and altitude

numeric matrix

"Load Factor" on page 4-176 for each airspeed and altitude, specified as a numeric matrix typically in g's.

Data Types: `double`

`airspeed` — Aircraft airspeed

column indices of `loadfactor` (default) | numeric vector | numeric matrix

Aircraft airspeed for each corresponding index in `loadfactor`, specified as a numeric vector or matrix.

Data Types: `double`

`altitude` — Aircraft altitude

column indices of `loadfactor` (default) | numeric vector | numeric matrix

Aircraft altitude for each corresponding index in `loadfactor`, specified as a numeric vector or matrix.

Data Types: `double`

`levels` — Levels for which to develop contour

automatically chosen by `contour` function (default) | scalar | vector

Levels for which to develop contour lines, specified as a scalar or vector.

- If `levels` is a scalar, `levels` specifies the number of contour lines, and the contour levels are chosen automatically by `contour`.
- If `levels` is a vector, `levels` specifies the number and levels of contour lines to plot.

Tip To draw the contours at one height (*k*), specify levels as a two-element row vector `[k k]`.

Example: `[2 3]`

Data Types: `double`

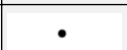
`LineSpec` — Line style, marker, and color

string scalar | character vector

Line style, marker, and color, specified as a string scalar or character vector containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `"--or"` is a red dashed line with circle markers.

Line Style	Description	Resulting Line
`"-"`	Solid line	
`"--"`	Dashed line	
`":"`	Dotted line	
`"-."`	Dash-dotted line	

Marker	Description	Resulting Marker
`"o"`	Circle	
`"+"`	Plus sign	
`"*"`	Asterisk	
`"."`	Point	
`"x"`	Cross	
`"_"`	Horizontal line	
`"|"`	Vertical line	

Marker	Description	Resulting Marker
`"square"`	Square	
`"diamond"`	Diamond	
`"^"`	Upward-pointing triangle	
`"v"`	Downward-pointing triangle	
`">"`	Right-pointing triangle	
`"<"`	Left-pointing triangle	
`"pentagram"`	Pentagram	
`"hexagram"`	Hexagram	

Color Name	Short Name	RGB Triplet	Appearance
`"red"`	`"r"`	`[1 0 0]`	
`"green"`	`"g"`	`[0 1 0]`	
`"blue"`	`"b"`	`[0 0 1]`	
`"cyan"`	`"c"`	`[0 1 1]`	
`"magenta"`	`"m"`	`[1 0 1]`	
`"yellow"`	`"y"`	`[1 1 0]`	
`"black"`	`"k"`	`[0 0 0]`	
`"white"`	`"w"`	`[1 1 1]`	

ax — Valid axes

scalar handle

Valid axes, specified as a scalar handle. By default, this function plots to the current axes, obtainable with the `gca` function.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Note The properties listed here are only a subset. For a full list, see Contour Properties.

Example: `"MinimumSpeed",0`

`MinimumAltitude` — Minimum altitude boundary
no minimum altitude boundary (default) | numeric scalar | *n*-by-2 matrix

Minimum altitude boundary, specified as a numeric scalar or *n*-by-2 matrix.

- If `MinimumAltitude` is a scalar, `MinimumAltitude` specifies a horizontal line that intersects with the *X* limits of the axes or with the intersection of `MinimumSpeed` and `MaximumSpeed`.
- If `MinimumAltitude` is an *n*-by-2 matrix, each row is an `airspeed` and `altitude` point of the boundary.

The function marks `loadfactor` values below `MinimumAltitude` as NaN.

Data Types: `double`

`MaximumAltitude` — Maximum altitude boundary
no maximum altitude boundary (default) | numeric scalar | *n*-by-2 matrix

Maximum altitude boundary, specified as a numeric scalar or *n*-by-2 matrix.

- If `MaximumAltitude` is a scalar, `MaximumAltitude` specifies a horizontal line that intersects with the *X* limits of the axes or with the intersection of `MinimumSpeed` and `MaximumSpeed`.
- If `MaximumAltitude` is an *n*-by-2 matrix, each row is an `airspeed` and `altitude` point of the boundary.

The function marks `loadfactor` values above `MaximumAltitude` as NaN.

Data Types: `double`

`MinimumSpeed` — Minimum speed boundary
no minimum speed boundary (default) | numeric scalar | *n*-by-2 matrix

Minimum speed boundary, specified as a numeric scalar or *n*-by-2 matrix.

- If `MinimumSpeed` is a scalar, `MinimumSpeed` specifies a vertical line that intersects with the *Y* limits of the axes or with the intersection of `MinimumAltitude` and `MaximumAltitude`.
- If `MinimumSpeed` is an *n*-by-2 matrix, each row is an `airspeed` and `altitude` point of the boundary.

The function marks `loadfactor` values behind `MinimumSpeed` as NaN.

Data Types: `double`

`MaximumSpeed` — Maximum speed boundary
no maximum speed boundary (default) | numeric scalar | *n*-by-2 matrix

Maximum speed boundary, specified as a numeric scalar or *n*-by-2 matrix.

- If `MaximumSpeed` is a scalar, `MaximumSpeed` specifies a vertical line that intersects with the *Y* limits of the axes or with the intersection of `MinimumAltitude` and `MaximumAltitude`.
- If `MaximumSpeed` is an *n*-by-2 matrix, each row is an `airspeed` and `altitude` point of the boundary.

The function marks as NaN `loadfactor` values in front of `MaximumSpeed`.

Data Types: `double`

`BoundaryXData` — Boundary line *X* data
no boundary line *X* data (default) | numeric vector

Boundary line *X* data, specified as a numeric vector.

Data Types: `double`

`BoundaryYData` — Boundary line *Y* data
no boundary line *Y* data (default) | numeric vector

Boundary line *Y* data, specified as a numeric vector.

Data Types: `double`

`ClipContour` — Contour data display
`'on'` (default) | `'off'`

Contour data display, specified as:

- `'on'` — Remove the contour data outside the specified boundary lines.
- `'off'` — Show all contour data and specified boundary lines.

Data Types: `double` | `logical` | `string`

`ResolveBoundary` — Boundary line intersection and enclosure
`'on'` (default) | `'off'`

Boundary line intersection and enclosure, specified as:

- `'on'` — Resolve boundary line segments to form a closed boundary around the line intersection points. Boundary data points outside the resolved boundary are removed. This method produces a well-formed boundary, but does not allow infinite limits or unconnected boundary lines.
- `'off'` — Do not resolve boundary line segments. This method leaves the boundary line data unmodified to allow infinite limits and unconnected boundary lines.

  Tip This method might produce a malformed boundary that affects the `ClipContour` behavior.

Data Types: `double` | `logical` | `string`

Output Arguments

`c` — Contour
numeric matrix

Contour, returned as a numeric matrix.

`h` — Contour graphics object
scalar

Contour graphics object, returned as a scalar.

`bline` — One or more boundary line objects
scalar | vector

One or more boundary line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line. For a list of properties, see Line Properties.

More About

Load Factor

Typically calculated as *lift*/*weight* where:

- *lift* — Lift of the aircraft.
- *weight* — Weight of the aircraft.

Version History

Introduced in R2021b

See Also

`boundaryline` | `clabel` | `contour` | `contourc` | `shortPeriodCategoryAPlot` | `shortPeriodCategoryBPlot` | `shortPeriodCategoryCPlot` | Contour Properties

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 `dcm` given three sets of rotation angles, `rotationAng1`, `rotationAng2`, and `rotationAng3`, specifying yaw, pitch, and roll. The rotation used in this function is a passive transformation between two coordinate systems.

`dcm = angle2dcm( ___ ,rotationSequence)` calculates the direction cosine matrix given the rotation sequence, `rotationSequence`.

Examples

Calculate Direction Cosine Matrix from Angles

Calculate the direction cosine matrix from three 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
```

Calculate Direction Cosine Matrix from Rotation Angle and Sequence

Calculate the direction cosine matrix from rotation angles and a 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 radians.

Data Types: `double` | `single`

`rotationAng2` — Second rotation angles

m-by-1 array

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

Data Types: `double` | `single`

`rotationAng3` — Third rotation angles

m-by-1 array

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

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

Version History

Introduced in R2006b

See Also

angle2quat | dcm2angle | dcmbody2stability | dcm2quat | quat2dcm | quat2angle

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. Aerospace Toolbox uses quaternions that are defined using the scalar-first convention. The rotation used in this function is a passive transformation between two coordinate systems.

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

Examples

Determine Quaternion from Rotation Angles

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 Quaternion from Rotation Angles and Sequence

Determine the quaternion from rotation angles using the YXZ 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
```

Input Arguments

`rotationAng1` — First rotation angles

m-by-1 array

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

Data Types: `double`

`rotationAng2` — Second rotation angles

m-by-1 array

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

Data Types: `double`

`rotationAng3` — Third rotation angles

m-by-1 array

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

Data Types: `double`

`rotationSequence` — Rotation sequence

`'ZYX'` (default) | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XZY'` | `'XYX'` | `'XZX'`

Rotation sequence, specified as:

- `'ZYX'`
- `'ZYZ'`
- `'ZXY'`
- `'ZXZ'`
- `'YXZ'`
- `'YXY'`
- `'YZX'`
- `'YZY'`
- `'XYZ'`
- `'XZY'`
- `'XYX'`
- `'XZX'`

where `rotationAng1` is *z*-axis rotation, `rotationAng2` is *y*-axis rotation, and `rotationAng3` is *x*-axis rotation.

Data Types: `char` | `string`

Output Arguments

`quaternion` — Converted quaternion
m-by-4 matrix

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

Version History

Introduced in R2007b

See Also

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

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`. The rotation used in this function is a passive transformation between two coordinate systems.

`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 $\vec{b}$ 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.

Version History

Introduced in R2017a

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` | `rod2quat` | `rod2angle` | `rod2dcm`

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) | on/off logical value

Visibility of the artificial horizon, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`. 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.

`ContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object created using the `uicontextmenu` function. Use this property to display a context menu when you right-click on a component.

`Enable` — Operational state of artificial horizon

`'on'` (default) | on/off logical value

Operational state of artificial horizon, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- 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 = uigridlayout([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 “Callbacks in App Designer”.

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 “Callbacks in App Designer”.

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.

The `BusyAction` property determines callback queuing behavior only when both of these conditions are met:

- The running callback contains a command that processes the callback queue, such as `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause`.
- The value of the `Interruptible` property of the object that owns the running callback is `'off'`.

Under these conditions, the `BusyAction` property of the object that owns the interrupting callback determines how MATLAB handles the interrupting callback. 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
on/off logical value

This property is read-only.

Deletion status, returned as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

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) | on/off logical value

Callback interruption, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

This 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.

MATLAB determines callback interruption behavior whenever it executes a command that processes the callback queue. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines if the interruption occurs:

- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the object that owns the interrupting callback determines if the interrupting callback is discarded or added to the callback queue.
- If the value of `Interruptible` is `'on'`, then the interruption occurs. The next time MATLAB processes the callback queue, it stops the execution of the running callback and executes the interrupting callback. After the interrupting callback completes, MATLAB then resumes executing the running callback.

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.
- If the interrupting callback is owned by a `Timer` object, then the callback executes according to schedule regardless of the `Interruptible` property value.

Note 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”.

Version History

Introduced in R2018b

See Also

`uiaerohorizon`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

atmoscoesa

Use 1976 COESA model

Syntax

```
[T,a,P,rho] = atmoscoesa(height)
[T,a,P,rho] = atmoscoesa(height,action)
```

Description

`[T,a,P,rho] = atmoscoesa(height)` implements the mathematical representation of the 1976 Committee on Extension to the Standard Atmosphere (COESA) United States standard lower atmospheric values. These values are absolute temperature, pressure, density, and speed of sound for the input geopotential altitude, `height`.

Below the geopotential altitude of 0 m (0 feet) and above the geopotential altitude of 84,852 m (approximately 278,386 feet), the `atmoscoesa` function extrapolates values.

`[T,a,P,rho] = atmoscoesa(height,action)` specifies the `action` for out-of-range input.

Examples

Calculate COESA Model at 1,000 Meters with Warnings Enabled

Calculate the COESA model at 1,000 meters with warnings for out-of-range inputs.

```
[T,a,P,rho] = atmoscoesa(1000)
```

```
T = 281.6500
```

```
a = 336.4341
```

```
P = 8.9875e+04
```

```
rho = 1.1116
```

Calculate COESA Model at 1,000, 11,000, and 20,000 Meters with Errors Enabled

Calculate the COESA model at 1,000, 11,000, and 20,000 meters with errors for out-of-range inputs.

```
[T,a,P,rho] = atmoscoesa([1000 11000 20000], 'Error')
```

```
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

scalar | vector | matrix

Geopotential heights, specified as a scalar, vector, or matrix in meters.

Data Types: `double`

`action` — Action

`'Warning'` (default) | `'Error'` | `'None'`

Action for out-of-range input, specified as:

- `'Error'` — Displays warning and indicates that the input is out of range.
- `'Warning'` — Displays error and indicates that the input is out of range.
- `'None'` — Does not display warning or error.

Data Types: `char` | `string`

Output Arguments

`T` — Temperatures

scalar | vector | matrix

Temperatures, returned as a scalar, vector, or matrix in the same size as the `height` argument, in kelvin. This function interpolates temperature values linearly.

`a` — Speeds of sound

scalar | vector | matrix

Speeds of sound, returned as a scalar, vector, or matrix in the same size as the `height` argument, in meters per second. This function calculates speed of sound using a perfect gas relationship.

`P` — Pressures

scalar | vector | matrix

Pressures, returned as a scalar, vector, or matrix in the same size as the `height` argument, in pascal. This function logarithmically calculates pressure.

`rho` — Densities
scalar | vector | matrix

Densities, returned as a scalar, vector, or matrix in the same size as the `height` argument, in kilograms per meter cubed. This function interpolates density values using a perfect gas relationship.

Version History

Introduced in R2006b

R2021b: `atmoscoesa` function changed input and returned value formats
Behavior changed in R2021b

The `atmoscoesa` function now:

- Accepts scalar, vector, or matrix values.
- Outputs scalar, vector, or matrix values.

As a result, the output values from this function might change from previous releases.

References

[1] *U.S. Standard Atmosphere*, 1976, U.S. Government Printing Office, Washington, D.C.

See Also

`atmoscira` | `atmosisa` | `atmoslapse` | `atmosnonstd` | `atmospalt`

atmoscira

Use COSPAR International Reference Atmosphere 1986 model

Syntax

```
[T,pressureOrAltitude,zonalWind] = atmoscira(latitude,ctype,coord)
[T,pressureOrAltitude,zonalWind] = atmoscira(latitude,ctype,coord,mtype)
[T,pressureOrAltitude,zonalWind] = atmoscira(latitude,ctype,coord,mtype,
month)
[T,pressureOrAltitude,zonalWind] = atmoscira(latitude,ctype,coord,month)
[T,pressureOrAltitude,zonalWind] = atmoscira( ___ ,action)
```

Description

`[T,pressureOrAltitude,zonalWind] = atmoscira(latitude,ctype,coord)` implements the mathematical representation of the Committee on Space Research (COSPAR) International Reference Atmosphere (CIRA) 1986 model. The CIRA 1986 model provides a mean climatology using a latitude, `latitude`, and a representation of coordinate type, `ctype`. The mean climatology consists of temperature `T`, zonal wind `zonalWind`, and pressure or geopotential height `pressureOrAltitude`. The provision 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.

`[T,pressureOrAltitude,zonalWind] = atmoscira(latitude,ctype,coord,mtype)` uses a mean value type, `mtype`, to implement these values.

`[T,pressureOrAltitude,zonalWind] = atmoscira(latitude,ctype,coord,mtype, month)` uses a monthly mean value type, `month`, to implement these values.

`[T,pressureOrAltitude,zonalWind] = atmoscira(latitude,ctype,coord,month)` uses a mean value type, `mtype`, and a monthly mean value type, `month`, to implement these values.

`[T,pressureOrAltitude,zonalWind] = atmoscira( ___ ,action)` uses the argument `action` to determine action reporting.

Examples

Calculate Temperature, Geopotential Height, and Zonal Wind Using CIRA 1986 Model

Using the CIRA 1986 model at 45 degrees latitude and 101,300 pascals 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
```

Calculate Temperature, Pressure, and Zonal Wind

Using the CIRA 1986 model at 45 degrees latitude and 20,000 m for October, calculate the mean monthly values. Calculate values for temperature (`T`), pressure (`pres`), and zonal wind (`zwind`).

c

```
[T,pres,zwind] = atmoscira(45,'GPHeight',20000,'Monthly',10)
```

```
T =
  215.8500
pres =
   5.5227e+03
zwind =
    9.5000
```

Calculate Temperature, Geopotential Height, and Zonal Wind Using CIRA 1986 Model at 45 and -30 Degrees

For September, use the CIRA 1986 model at 45 degrees latitude and –30 degrees latitude. Also use the model at 2000 pascals and 101,300 pascals. 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+04 *
    2.6692    0.0058
zwind =
    0.6300   -1.1000
```

Calculate Temperature, Geopotential Height, and Zonal Wind Using CIRA 1986 Model at 45 and 2000 Pascals

Using the CIRA 1986 model at 45 degrees latitude and 2000 pascals, 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    5.0098    6.5300    1.9499    1.3000    1.0499    1.3000
alt =
   1.0e+04 *
```

```
    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
```

Calculate Temperature, Pressure, and Zonal Wind Using CIRA 1986 Model at 45 and -30 Degrees and 20000 Meters

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+03 *
    5.5227    5.6550
zwind =
    9.5000    4.3000
```

Input Arguments

`latitude` — Geodetic latitudes

array

Geodetic latitudes, specified as an array, in degrees, where north latitude is positive and south latitude is negative.

Data Types: `double`

`ctype` — Representation of coordinate type

`'Pressure'` | `'GPHeight'`

Representation of coordinate type, specified as of these values.

Coordinate Type	Description
`'Pressure'`	Pressure in pascals
`'GPHeight'`	Geopotential height in meters

Dependencies

- When `ctype` is set to `'Pressure'`, `pressureOrAltitude` returns the altitude.
- When `ctype` is set to `'GPHeight'`, `pressureOrAltitude` returns the geopotential height.

Data Types: `char` | `string`

`coord` — Pressures or geopotential heights

array

Pressures or geopotential heights, specified as an array depending on the value of `ctype`:

Coordinate Type	Description
`'Pressure'`	Pressure in pascals
`'GPHeight'`	Geopotential height in meters

Dependencies

- When `ctype` is set to `'Pressure'`, the function interprets `coord` as an array of pressures.
- When `ctype` is set to `'GPHeight'`, the function interprets `coord` as an array of geopotential heights.

Data Types: `double`

`mtype` — Mean value type

`'Monthly'` (default) | `'Annual'`

Mean value type of data type string, specified as one of these values.

Mean Value Type	Description
`'Monthly'` (default)	Monthly values.
`'Annual'`	Annual values. Valid when `ctype` has a value of `'Pressure'`.

Dependencies

`'Annual'` is available only when `ctype` is set to `'Pressure'`.

Data Types: `char` | `string`

`month` — Month

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12

Month in which model takes mean values, specified as one of these values.

Value	Month
1	January
2	February
3	March
4	April
5	May
6	June
7	July
8	August
9	September
10	October
11	November
12	December

Data Types: `double`

`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 and model simulation continues.
`'Error'`	MATLAB returns an exception and model simulation stops.

Data Types: `char` | `string`

Output Arguments

`T` — Temperatures

array

Temperatures, returned as an array depending on the value of `mtype`.

mtype Value	Description
`'Monthly'`	Array of *m* temperatures, in kelvin
`'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

Dependencies

`'Annual'` is available only when `ctype` is set to `'Pressure'`.

`pressureOrAltitude` — Geopotential heights or pressures

array

Geopotential heights or pressures, returned as an array, depending on the value of `ctype`.

If `mtype` is `'Annual'`, `pressureOrAltitude` is 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

Dependencies

- When `ctype` is set to `'Pressure'`, `pressureOrAltitude` returns the altitude.
- When `ctype` is set to `'GPHeight'`, `pressureOrAltitude` returns the geopotential height.

`zonalWind` — Zonal winds
array

Zonal winds, returned as an array depending on the value of `mtype`:

mtype Value	Description
`'Monthly'`	Array in meters per second.
`'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

Limitations

- This function has the limitations of the CIRA 1986 model and limits the values for the CIRA 1986 model.
- The CIRA 1986 model that `atmoscira` implements limits values to latitudes of 80 degrees S to 80 degrees N on Earth and geopotential heights of 0 to 120 kilometers. `atmoscira` clips output beyond these ranges.

  The CIRA 1986 model data also has missing data points.

  - In each monthly mean data set, the model omits values at 80 degrees S for 101,300 pascals or 0 meters because these levels are within the Antarctic land mass.
  - In the annual mean data set for latitudes 70, 75, and 80 degrees S, the model also omits these values:

    - 101,300 and 78,895 pascals
    - 0 and 2,500 meters

As a result of the missing data points, input for pressures above 61,441 pascals or geopotential heights below 5000 meters at these latitudes generate `NaN` output.

For zonal mean pressure in constant altitude coordinates, pressure data is not available below 20 km. This value is the bottom level of the CIRA climatology.

When `atmoscira` detects an out-of-range input, it uses the `action` argument to determine the function behavior.

Version History

Introduced in R2007b

R2023a: `atmoscira` Function Update

The `atmoscira` function has been updated to take action when inputs to the function are outside the CIRA 1986 model limitations (see “Limitations” on page 4-202).

References

[1] Fleming, E. L., S. Chandra, M. R. Shoeberl, and J. J. Barnett. *Monthly Mean Global Climatology of Temperature, Wind, Geopotential Height and Pressure for 0-120 km*. NASA TM100697, February 1988.

See Also

External Websites
https://ccmc.gsfc.nasa.gov/

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 geographic coordinates: `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

Calculate 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.

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.

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. Values are held outside the range 0 to 500 km.

Example: 25000

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 Coordinated Universal 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 the number of requested geographic coordinates. 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. Values are wrapped to within 1 to 366 days.

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.

Specify the seconds as a value from 0 through 86,400. Values are wrapped to within 0 to 86400 seconds.

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.
- `'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: `char` | `string`

`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.
`'Warning'`	Warning in the MATLAB Command Window, model simulation continues.
`'Error'`	MATLAB returns an exception, model simulation stops.

Data Types: `char` | `string`

`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: `char` | `string`

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.

Version History

Introduced in R2016b

R2021b: `atmoshwm` Function Possible Changed Returned Values
Behavior changed in R2021b

The `atmoshwm` function now:

- Accepts `day` decimal input values.
- Limits `altitude` input values to `500` km.

As a result, the output values from this function might change from previous releases.

See Also

`atmoscoesa` | `atmosnrlmsise00` | `atmoscira`

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

atmosisa

Use International Standard Atmosphere model

Syntax

```
[T,a,P,rho,nu] = atmosisa(height)
[T,a,P,rho,nu] = atmosisa(height,Name=Value)
```

Description

`[T,a,P,rho,nu] = atmosisa(height)` implements the mathematical representation of the International Standard Atmosphere values for ambient temperature, `T`, pressure, `P`, speed of sound, `a`, ,density, `rho`, and kinematic viscosity, `nu`, for the input geopotential altitude, `height`, between the sea level and the tropopause.

`[T,a,P,rho,nu] = atmosisa(height,Name=Value)` specifies options using one or more name-value arguments. For example, to input geopotential altitude between -5000 meters and the mesopause, set `extended` to `true`.

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 with `extended=true` to implement the mathematical representation of the International Standard Atmosphere values to between -5000 meters and the mesopause..

```
[T, a, P, rho] = atmosisa([1000 11000 20000],extended=true)
```

```
T = 1×3

  281.6500  216.6500  216.6500
```

```
a = 1×3

  336.4340  295.0695  295.0695


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-*n* array

Geopotential heights, specified as an *m*-by-*n* array.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `extended,false` implements the mathematical representation of the International Standard Atmosphere values for ambient temperature, pressure, density, speed of sound, and kinematic viscosity for the input geopotential altitude between the sea level and the tropopause.

`extended` — Range of implementation

`false` or `'off'` (default) | `true` or `'on'`

Mathematical representation between sea level and tropopause, specified as false or true. Set `extended` to `false` or to `'off'` implement the mathematical representation of the International Standard Atmosphere values for ambient temperature, pressure, density, speed of sound, and kinematic viscosity for the input geopotential altitude between the sea level and the tropopause.

Set `extended` to `true` or `'on'` to implement the mathematical representation of the International Standard Atmosphere values for ambient temperature, pressure, density, speed of sound, and kinematic viscosity for the input geopotential altitude between -5000 meters and the mesopause.

`action` — Function behavior

`'Warning'` (default) | `'None'` | `'Error'`

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 and model simulation continues.
`'Error'`	MATLAB returns an exception and model simulation stops.

Dependencies

The `atmosisa` function takes the action into account when `'extended'` is set to `'true'`.

Output Arguments

`T` — Temperatures

m-by-*n* array

Temperatures, returned as an *m*-by-*n* array, in kelvin.

`a` — Speeds of sound

m-by-*n* array

Speeds of sound, returned as an *m*-by-*n* array, in meters per second. The function calculates speed of sound using a perfect gas relationship.

`P` — Pressures

m-by-*n* array

Pressures, returned as an *m*-by-*n* array, in pascal.

`rho` — Densities

m-by-*n* array

Densities, returned as an *m*-by-*n* array, in kilograms per meter cubed. The function calculates density using a perfect gas relationship.

`nu` — Kinematic viscosity

m-by-*n* array

Kinematic viscosity, returned as an *m*-by-*n* array, in m^2/s.

Limitations

Set upper and lower limits for geopotential altitudes using the `'extended'` name-value argument.

- If the `'extended'` flag is set to `'false'`, the function clips the output values below the geopotential altitude of 0 meters and above the geopotential altitude of the tropopause at 20,000 meters.
- If the `'extended'` flag is set to `'true'`, the function limits the implementation to the geopotential altitude of -5000 meters (minimum) and the mesopause at 84,852 meters (maximum).

If the action input is `'error'`, an input geopotential height outside these limits results in error. If the action input is `'warning'` or `'none'`, the outputs are held at the limits.

Version History

Introduced in R2006b

R2023a: `atmosisa` Function Changes

The `atmosisa` function now supports the ISA atmosphere model input geopotential altitude between -5000 meters and the mesopause. In previous releases, `atmosisa` supported input geopotential altitudes between sea level and the tropopause.

To support the new capability, the function has these changes:

- `extended` name-value argument, which lets you specify:
  - `false` or `'off'` — `atmosisa` supports input geopotential altitudes between the sea level and the tropopause.
  - `true` or `'on'` — `atmosisa` supports input geopotential altitudes between -5000 meters and the mesopause.
- `nu` output, which returns a numeric array of *M*-by-*N* values of kinematic viscosities in m^2/s.
- `'action'` name-value argument, which lets you specify the action for out-of-range input when `'extended'` is set to `'true'`.

References

[1] *U.S. Standard Atmosphere, 1976*. U.S. Government Printing Office, Washington, D.C.

See Also

`atmoscira` | `atmoscoesa` | `atmoslapse` | `atmosnonstd` | `atmospalt`

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.

`[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.

Examples

Calculate Lapse Rate Atmosphere Using International Standard Atmosphere

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+04
```

```
rho =
    1.1116
```

Input Arguments

`height` — Geopotential heights

m-by-*n* array

Geopotential heights, specified as an *m*-by-*n* array in meters.

Data Types: `double`

`g` — Acceleration

scalar

Acceleration due to gravity, specified as a scalar in meters per second squared.

Data Types: `double`

`heatRatio` — Heat ratio

scalar

Heat ratio, specified as a scalar.

Data Types: `double`

`characteristicGasConstant` — Characteristic gas constant

scalar

Characteristic gas constant, specified as a scalar in joules per kilogram-kelvin.

Data Types: `double`

`lapseRate` — Lapse rate

scalar

Lapse rate, specified as a scalar in kelvin per meter.

Data Types: `double`

`heightTroposphere` — Height of troposphere

scalar

Height of troposphere, specified as a scalar in meters.

Data Types: `double`

`heightTropopause` — Height of tropopause

scalar

Height of tropopause, specified as a scalar in meters.

Data Types: `double`

`density0` — Density at mean sea level

scalar

Density at mean sea level (MSL), specified as a scalar in kilograms per meter cubed.

Data Types: `double`

`pressure0` — Static pressure
scalar

Static pressure at MSL, specified as a scalar in pascal.

Data Types: `double`

`temperature0` — Absolute temperature
scalar

Absolute temperature at MSL, specified as a scalar in kelvin.

Data Types: `double`

`height0` — Minimum sea level altitude
0 (default) | scalar

Minimum sea level altitude, specified as a scalar in meters.

Data Types: `double`

Output Arguments

`T` — Temperatures
m-by-1 array

Temperatures, returned as an *m*-by-1 array in kelvin.

`a` — Speeds of sound
m-by-1 array

Speeds of sound, returned as an *m*-by-1 array in meters per second squared. The function calculates speed of sound using a perfect gas relationship.

`P` — Pressures
m-by-1 array

Pressures, returned as an *m*-by-1 array in pascal.

`rho` — densities
m-by-1 array

Densities, specified as an *m*-by-1 array kilograms per meter cubed. The function calculates density using a perfect gas relationship.

Version History

Introduced in R2006b

References

[1] *U.S. Standard Atmosphere*. Washington, DC: US Government Printing Office, 1976.

See Also

`atmoscira` | `atmoscoesa` | `atmosisa` | `atmosnonstd` | `atmospalt`

atmosnonstd

Use climatic data from MIL-STD-210 or MIL-HDBK-310

Syntax

```
[T,a,P,rho] = atmosnonstd(height,atmosphericType,extremeParameter,frequency,
extremeAltitude)
[T,a,P,rho] = atmosnonstd( ___ ,action)
[T,a,P,rho] = atmosnonstd( ___ ,specification)
```

Description

`[T,a,P,rho] = atmosnonstd(height,atmosphericType,extremeParameter,frequency, extremeAltitude)` 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). For exceptions on the envelope atmosphere model, see `atmosphericType`.

`[T,a,P,rho] = atmosnonstd( ___ ,action)` specifies the action for out-of-range input. Specify `action` after all other input arguments.

`[T,a,P,rho] = atmosnonstd( ___ ,specification)` specifies the MIL-STD-210C or MIL-STD-310 climatic data. Specify `specification` after all other input arguments.

Examples

Calculate Nonstandard Atmosphere Profile

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+04

rho =
    1.2620
```

Calculate Nonstandard Atmosphere Profile Specifying Atmosphere Model

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+04 *

    9.1598    2.5309    0.6129

rho =
     0     0     0
```

Input Arguments

`height` — Geopotential heights

m-by-1 array

Geopotential heights, specified as an *m*-by-1 array.

Data Types: `double`

`atmosphericType` — Atmospheric data type

`'Profile'` | `'Envelope'`

Atmospheric data, specified as one of these values.

Atmospheric Data Type	Description
`'Profile'`	Use for simulation of vehicles vertically traversing the atmosphere, or when you need the total influence of the atmosphere. Use this type for realistic atmospheric profiles associated with extremes at specified altitudes.

Atmospheric Data Type	Description
`'Envelope'`	Use for vehicles traversing the atmosphere horizontally, without much change in altitude. Use this type for extreme atmospheric values at each altitude. Due to lack of data in MIL-STD-210 or MIL-HDBK-310 for these conditions, this atmospheric model has these exceptions: • When extreme value is the only value provided as an output, the function interpolates pressure logarithmically. These 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. • When values are held below the geometric altitude of 1 km (approximately 3281 feet). • When values are above the geometric altitude of 30,000 m (approximately 98,425 feet).

Data Types: `string`

`extremeParameter` — Atmospheric parameter for extreme value
`'High temperature'` | `'Low temperature'` | `'High density'` | `'Low density'` | `'High pressure'` | `'Low pressure'`

Atmospheric parameter for extreme value, specified as:

- `'High temperature'`
- `'Low temperature'`
- `'High density'`
- `'Low density'`
- `'High pressure'`
- `'Low pressure'`

Dependencies

`'High pressure'` and `'Low pressure'` are available only if `atmosphericType` is set to `'Envelope'`.

Data Types: `double`

`frequency` — Percent of time extreme values can occur
`'Extreme values'` | `'1%'` | `'5%'` | `'10%` | `'20%'`

Percent of time that extreme values can occur, specified as:

- `'Extreme values'`
- `'1%'`
- `'5%'`
- `'10%`
- `'20%'`

Dependencies

- `'Extreme values'`, `'5%'`, and `'20%'` are available only if `atmosphericType` is set to `'Envelope'`.
- When `atmosphericType` is set to `'Envelope'` and `frequency` is set to `'5%'`, `'10%'`, or `'20%'`, `atmosnonstd` outputs valid output only for temperature `T`, density `rho`, and pressure `P`. All other parameter outputs are zero.

Data Types: `double`

`extremeAltitude` — Geometric altitude

`'5'` | `'10'` | `'20'` | `'30'` | `'40'`

Geometric altitude in kilometers, specified as one of these values.

Altitude in Kilometers	Altitude in Feet
`'5'`	16404 ft
`'10'`	32808 ft
`'20'`	65617 ft
`'30'`	98425 ft
`'40'`	131234 ft

Data Types: `double`

`action` — Action

`'Warning'` (default) | `'Error'` | `'None'`

Action for out-of-range input, specified as:

- `'Error'` — Displays warning and indicates that the input is out of range.
- `'Warning'` — Displays error and indicates that the input is out of range.
- `'None'` — Does not display warning or error.

Data Types: `char` | `string`

`specification` — Atmosphere model

`'310'` (default) | `'210c'`

Atmosphere model, specified as one of these values.

Specification	Description
`'310'`	MIL-HDBK-310
`'210c'`	MIL-STD-210C

Data Types: `double`

Output Arguments

`T` — Temperatures

m-by-1 array

Temperatures, returned as an *m*-by-1 array in kelvin. This function interpolates temperature values linearly.

a — Speeds of sound
m-by-1 array

Speeds of sound, returned as an *m*-by-1 array in meters per second. This function calculates speed of sound using a perfect gas relationship.

P — Pressures
m-by-1 array

Pressures, returned as an *m*-by-1 array in pascal. This function calculates pressure using a perfect gas relationship.

`rho` — Densities
m-by-1 array

Densities, returned as an *m*-by-1 array in kilograms per meter cubed. This function interpolates density values logarithmically.

Limitations

- 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 and MIL-HDBK-310 specifications, resulting in some inconsistency 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 these 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.

Version History

Introduced in R2006b

References

[1] *Global Climatic Data for Developing Military Products (MIL-STD-210C)*. Washington, DC: Department of Defense January 9, 1987.

[2] *Global Climatic Data for Developing Military Products (MIL-HDBK-310)*. Washington, DC: Department of Defense, June 23, 1997.

See Also

`atmoscira` | `atmoscoesa` | `atmosisa` | `atmoslapse` | `atmospalt`

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( ___ ,localApparentSolarTime)
[T rho] = atmosnrlmsise00( ___ ,f107Average,f107Daily,magneticIndex)
[T rho] = atmosnrlmsise00( ___ ,flags)
[T rho] = atmosnrlmsise00( ___ ,otype)
[T rho] = atmosnrlmsise00( ___ ,action)
```

Description

Syntax Using Default Arguments

`[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.

Syntaxes Using Specified Arguments

`[T rho] = atmosnrlmsise00( ___ ,localApparentSolarTime)` specifies an array of *m* local apparent solar time (hours). Specify `localApparentSolarTime` after all other input arguments in previous syntaxes.

`[T rho] = atmosnrlmsise00( ___ ,f107Average,f107Daily,magneticIndex)` specifies arrays of *m* 81-day average of F10.7 flux (centered on `dayOfYear`), *m*-by-1 daily F10.7 flux for previous day, and *m*-by-7 array of magnetic index information. Specify `f107Average`, `f107Daily`, and `magneticIndex` after all other input arguments in previous syntaxes. As a convenience, consider using the `fluxSolarAndGeomagnetic` function to specify the `f107Average`, `f107Daily`, and `magneticIndex` variables.

`[T rho] = atmosnrlmsise00( ___ ,flags)` specifies an array of 23 flags to enable or disable particular variations for the outputs. Specify `flags` after all other input arguments in previous syntaxes.

`[T rho] = atmosnrlmsise00( ___ ,otype)` specifies a character vector or string for total mass density output. Specify `otype` after all other input arguments in previous syntaxes.

`[T rho] = atmosnrlmsise00( ___ ,action)` specifies an out-of-range input action. Specify `action` after all other input arguments in previous syntaxes.

Examples

Calculate Temperatures and Densities Using the NRLMSISE-00 Model, Scalars, and Default Flux, Magnetic Index Data, and Local Solar Time

Calculate the temperatures and densities, not including anomalous oxygen, using the NRLMSISE-00 model at 10,000 m, 45 degrees latitude, and -50 degrees longitude. This calculation uses the date January 4, 2007 at 0 UT. The example 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 Temperatures and Densities Using the NRLMSISE-00 Model, Arrays, and Default Flux, Magnetic Index Data, and Local Solar Time

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. The example 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+03 *

    1.0273    0.2212
    1.0273    0.2116

rho =
   1.0e+24 *

    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 Temperatures and Densities Using the NRLMSISE-00 Model and Default Flux, Magnetic Index Data, and Local Solar Time

Calculate the temperatures, densities including anomalous oxygen using the NRLMSISE-00 model at 10,000 m, 45 degrees latitude, and –50 degrees longitude. This calculation uses the date January 4, 2007 at 0 UT seconds. The example 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+03 *

    1.0273    0.2212

rho =
```

```
   1.0e+24 *

    0.0000         0    6.6824    1.7927    0.0799    0.0000         0         0         0
```

Calculate Temperatures and Densities Using the NRLMSISE-00 Model and Specified Flux, Magnetic Index Data, and Default Local Solar Time

Calculate the temperatures, densities including anomalous oxygen using the NRLMSISE-00 model at 100,000 m, 45 degrees latitude, and –50 degrees longitude. This calculation uses the date January 4, 2007 at 0 UT seconds. It uses defined values for flux, and magnetic index data, and default local solar time. The example 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')
```

```
T =
   1.0e+03 *

    1.0273    0.1917

rho =
   1.0e+18 *

    0.0001    0.4241    7.8432    1.9721    0.0808    0.0000    0.0000    0.0000    0.0000
```

Input Arguments

`altitude` — Altitudes

m-by-1 array

Altitudes, specified as an *m*-by-1 array in meters.

Data Types: `double`

`latitude` — Geodetic latitudes

m-by-1 array

Geodetic latitudes, specified as an *m*-by-1 array in degrees.

Data Types: `double`

`longitude` — Geodetic longitudes

m-by-1 array

Geodetic longitudes, specified as an *m*-by-1 array in degrees.

Tip 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 variations from this equation for `localApparentSolarTime`, but they are of minor importance.

Data Types: `double`

`year` — Year
m-by-1 array

Year, specified as an *m*-by-1 array. This function ignores the value of `year`.

Data Types: `double`

`dayOfYear` — Day or year
m-by-1 array

Day or year, specified as an *m*-by-1 array.

Data Types: `double`

`UTseconds` — Universal time
m-by-1 array

Universal time (UT), specified as an *m*-by-1 array in seconds.

Tip 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 variations from this equation for `localApparentSolarTime`, but they are of minor importance.

Data Types: `double`

`localApparentSolarTime` — Local apparent solar times
m-by-1 array

Local apparent solar times, specified as an *m*-by-1 array in hours.

Tip 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 variations from this equation for `localApparentSolarTime`, but they are of minor importance.

Data Types: `double`

`f107Average` — 81 day average of F10.7 flux
m-by-1 array

81 day average of F10.7 flux, centered on day of year (`dayOfYear`), specified as an *m*-by-1 array. 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: `https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/`

For limitations, see “Limitations” on page 4-228.

Dependencies

If you specify `f107Average`, you must also specify `f107Daily` and `magneticIndex`.

Data Types: `string`

`f107Daily` — Daily F10.7 flux for previous day

m-by-1 array

Daily F10.7 flux for previous day, specified as an *m*-by-1 array. 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: `https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/`

For limitations, see “Limitations” on page 4-228.

Dependencies

If you specify `f107Daily`, you must also specify `f107Average` and `magneticIndex`.

Data Types: `string`

`magneticIndex` — Magnetic index information

m-by-7 array

Magnetic index information, specified as an *m*-by-7 array. 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. For limitations, see “Limitations” on page 4-228.

Dependencies

If you specify `magneticIndex`, you must also specify `f107Average` and `f107Daily`.

Data Types: `double`

`flags` — Output variations

m-by-1 array

Output variations, specified as a *m*-by-1 array. If the `flags` array length, *m*, is 23 and you have not specified all available inputs, this function assumes that `flags` is set.

The flags 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 seconds, longitudinal effects
`Flags(11)`	Longitudinal
`Flags(12)`	UT seconds and mixed UT seconds, longitudinal
`Flags(13)`	Mixed AP, UT seconds, 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

Data Types: `string`

`otype` — Total mass density output

`'Oxygen'` | `'NoOxygen'`

Total mass density output, specified as one of these values.

`'Oxygen'`	Total mass density outputs include anomalous oxygen number density.
`'NoOxygen'`	Total mass density outputs do not include anomalous oxygen number density.

Data Types: `string`

`action` — Action
`'Warning'` (default) | `'Error'` | `'None'`

Action for out-of-range input, specified as:

- `'Error'` — Displays warning and indicates that the input is out of range.
- `'Warning'` — Displays error and indicates that the input is out of range.
- `'None'` — Does not display warning or error.

Data Types: `char` | `string`

Output Arguments

`T` — Temperatures
N-by-2 array

Temperatures, returned as an *N*-by-2 array in kelvin. The first column of the array is exospheric temperatures. The second column of the array is temperatures at altitude.

`rho` — Densities
N-by-9 array

Densities (kg/m^3 or 1/m^3) in selected density units, returned as an *N*-by-9 array 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.

Limitations

- This function has the limitations of the NRLMSISE-00 model. For more information, see the NRLMSISE-00 model documentation.
- If array length, *m*, is `23` and all available inputs are not specified, the function assumes that `flags` is set.
- `f107Average` and `f107Daily` values that generate the model correspond to the 10.7 cm radio flux at the actual distance of the Earth from the Sun rather than the radio flux at 1 AU. This site

provides both classes of values: `https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/`.

Version History

Introduced in R2007b

See Also

`atmoscira` | `aeroReadSpaceWeatherData` | `fluxSolarAndGeomagnetic`

External Websites

https://ccmc.gsfc.nasa.gov/
ftp://ftp.ngdc.noaa.gov/STP/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/

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.

Examples

Calculate Pressure Altitude at Static Pressure

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 Pressure Altitude at Array of Static Pressures

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+04 *

         0    1.0000
```

Input Arguments

`pressure` — Ambient pressures

m-by-1 array

Ambient pressures, specified as an *m*-by-1 array in pascals.

Data Types: `double`

`action` — Action
`'Warning'` (default) | `'Error'` | `'None'`

Action for out-of-range input, specified as:

- `'Error'` — Displays warning and indicates that the input is out of range.
- `'Warning'` — Displays error and indicates that the input is out of range.
- `'None'` — Does not display warning or error.

Data Types: `char` | `string`

Output Arguments

`pressureAltitude` — Pressure altitudes or MSL altitudes
m-by-1 array

Pressure altitudes or MSL altitudes, returned as an *m*-by-1 array in meters.

Version History

Introduced in R2006b

References

[1] *U.S. Standard Atmosphere*. Washington, DC: U.S. Government Printing Office, 1976.

See Also

`atmoscira` | `atmoscoesa` | `atmosisa` | `atmoslapse`

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.

Version History

Introduced in R2007a

See Also

`Aero.Body`

boundaryline

Draw boundary line plot

Syntax

```
boundaryline(x,y)
boundaryline(x,y,LineSpec)

boundaryline(___,Name,Value)
boundaryline(ax, ___)

bline = boundaryline( ___ )
```

Description

Use Default Boundary and Line Specification

`boundaryline(x,y)` plots a boundary line specified by the *x* data `x` and the *y* data *y*. The boundary line contains hatch marks that extend from a fixed spacing and length along the boundary line.

`boundaryline(x,y,LineSpec)` plots a boundary line specified by the line specification `linespec`.

Specify Name,Value Arguments and Axis

`boundaryline(___,Name,Value)` plots a boundary line specified by one or more `Name,Value` pairs. Adjust the look of the boundary line with the `'Hatches'`, `'HatchLength'`, `'HatchTangency'`, `'HatchAngle'`, `'HatchSpacing'`, and `'FlipBoundary'` properties. Specify name-value pair arguments after all other input arguments.

`boundaryline(ax, ___)` plots a boundary line on the specified axes `ax` instead of the current axes, such as that from the `gca` function.

Return Boundary Line Object

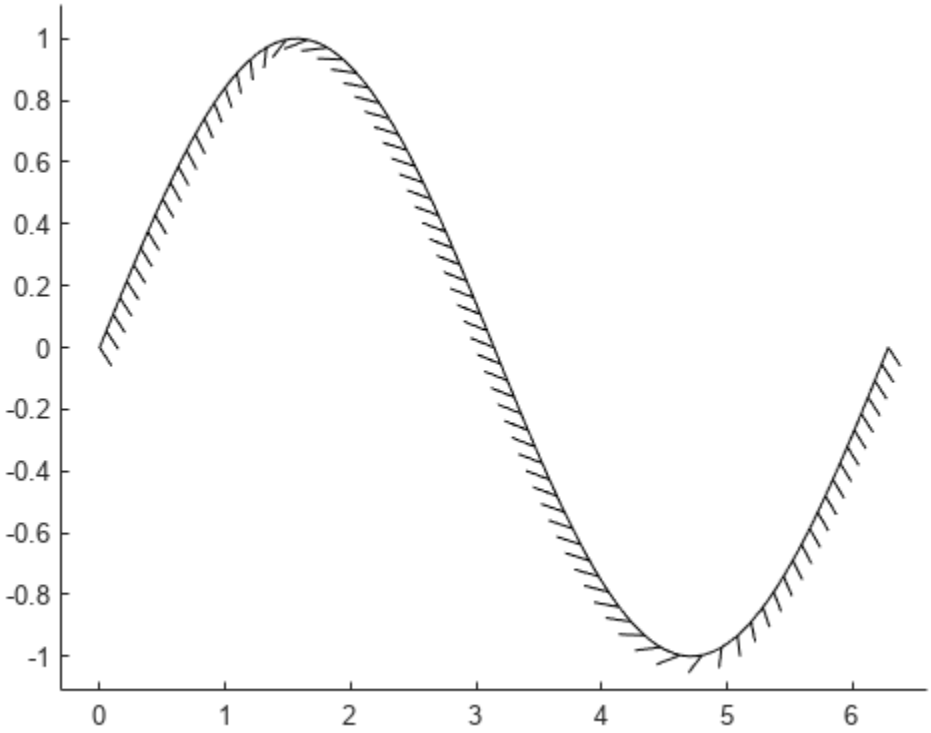
`bline = boundaryline( ___ )` returns a boundary line object using any of the input argument combinations in the previous syntaxes. Specify arguments as previously listed.

Examples

Plot Boundary Line Sine Wave

Plot the boundary line of a sine wave.

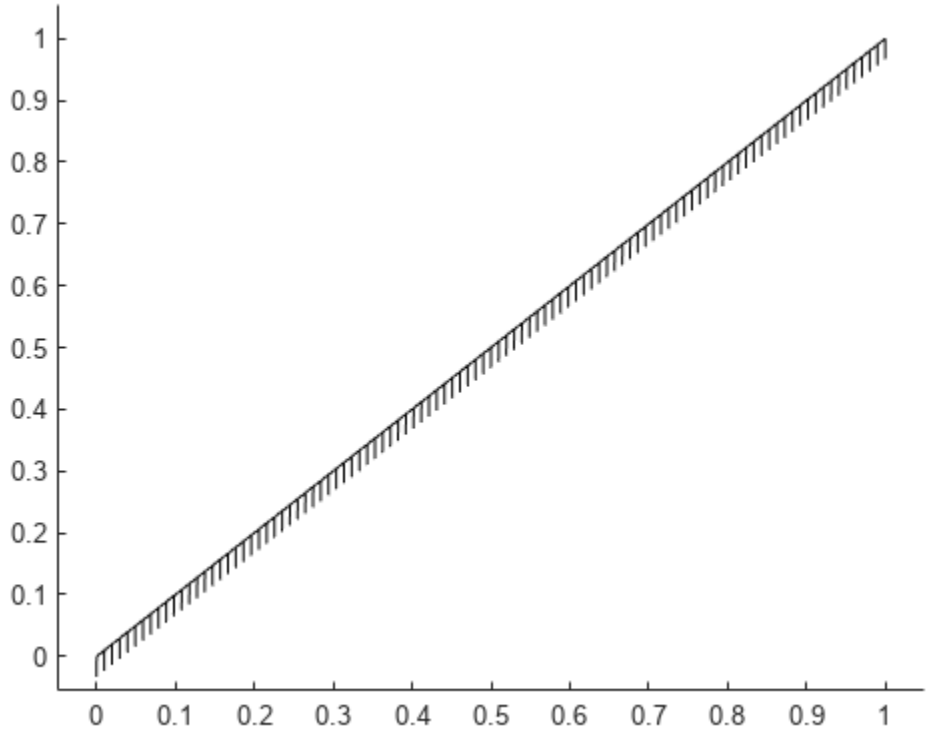
```
x = linspace(0,2*pi);
y = sin(x);
boundaryline(x,y)
```



Plot Boundary Line with Third-Spaced Hatches

Plot a boundary line with third-spaced hatches. Return the boundary line object in `b`.

```
b = boundaryline([0,1],[0,1],'Hatches','//')
```

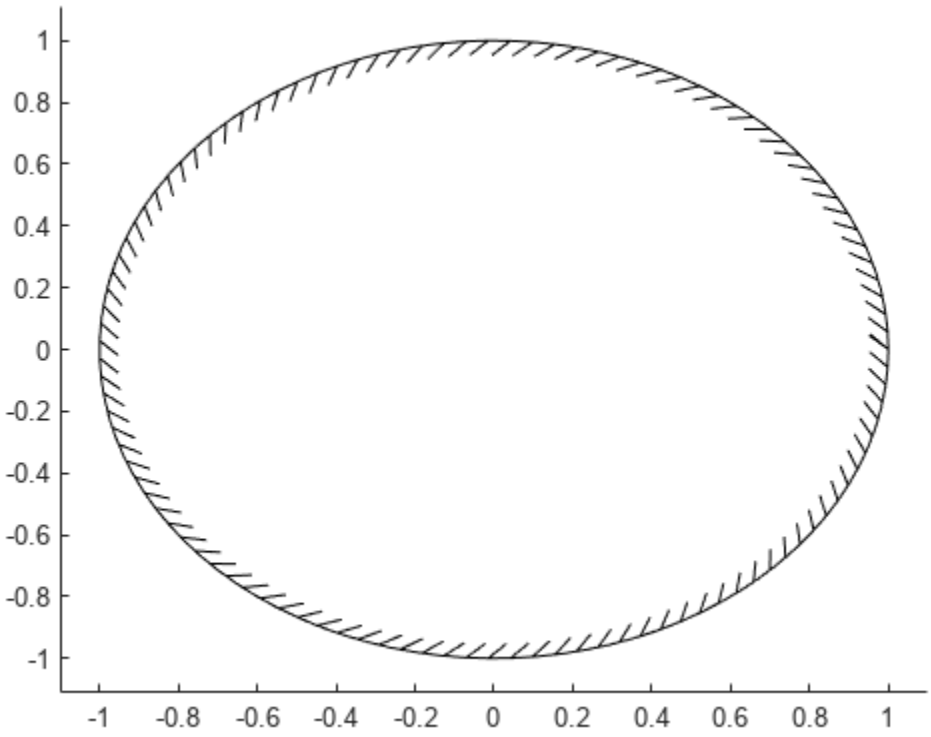



```
b =
  BoundaryLine with properties:

              Color: [0 0 0]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
    MarkerFaceColor: 'none'
         MarkerSize: 6
      HatchSpacing: 0.0125
       HatchLength: 0.0300
        HatchAngle: 225
           Hatches: '//'
     HatchTangency: on
      FlipBoundary: off
             XData: [0 1]
             YData: [0 1]

  Show all properties
```

Plot Circle Boundary Line with Flipped Boundary

Plot a circle boundary line and flip the boundary after creation. Return the boundary line object in `b`.

```
t = linspace(0, 2*pi);
      x = cos(t);
      y = sin(t);
      b = boundaryline(x,y)

b =
  BoundaryLine with properties:

              Color: [0 0 0]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
    MarkerFaceColor: 'none'
         MarkerSize: 6
       HatchSpacing: 0.0255
        HatchLength: 0.0300
         HatchAngle: 225
            Hatches: '/'
      HatchTangency: on
       FlipBoundary: off
              XData: [1 0.9980 0.9920 0.9819 0.9679 0.9501 0.9284 0.9029 0.8738 0.8413 0.8053 0.7
              YData: [0 0.0634 0.1266 0.1893 0.2511 0.3120 0.3717 0.4298 0.4862 0.5406 0.5929 0.6

  Show all properties


      b.FlipBoundary = true
```

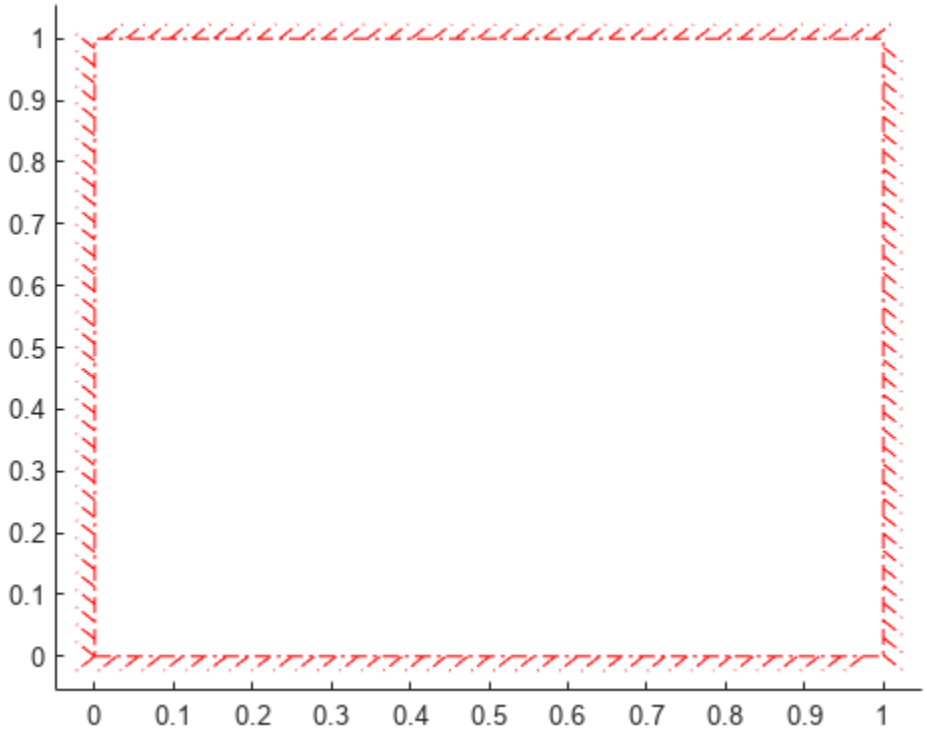


```
b =
  BoundaryLine with properties:

             Color: [0 0 0]
         LineStyle: '-'
         LineWidth: 0.5000
            Marker: 'none'
    MarkerFaceColor: 'none'
         MarkerSize: 6
       HatchSpacing: 0.0255
        HatchLength: 0.0300
         HatchAngle: 225
            Hatches: '/'
      HatchTangency: on
       FlipBoundary: on
             XData: [1 0.9980 0.9920 0.9819 0.9679 0.9501 0.9284 0.9029 0.8738 0.8413 0.8053 0.7
             YData: [0 0.0634 0.1266 0.1893 0.2511 0.3120 0.3717 0.4298 0.4862 0.5406 0.5929 0.6

  Show all properties
```

Plot Dashed and Dotted Square Boundary Line in Red

Plot a red, dotted and dashed, square boundary line on a specified axis `a` is the current axis.

```
a = gca;
boundaryline(a,[0,1,1,0,0],[0,0,1,1,0],'r-.')
```



Input Arguments

x — *x* coordinate data

numeric vector

x coordinate data, specified as a numeric vector. The function uses this data to plot the *x* coordinates of the boundary line.

Data Types: `double`

y — *y* coordinate data

numeric vector

y coordinate data, specified as a numeric vector. The function uses this data to plot the *y* coordinates of the boundary line.

Data Types: `double`

ax — Valid axes

scalar handle

Valid axes, specified as a scalar handle. By default, this function plots to the current axes, obtainable with the `gca` function.

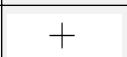
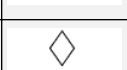
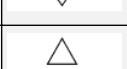
Data Types: `double`

`LineSpec` — Line style, marker, and color
string scalar | character vector

Line style, marker, and color, specified as a string scalar or character vector containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `"--or"` is a red dashed line with circle markers.

Line Style	Description	Resulting Line
`"-"`	Solid line	
`"--"`	Dashed line	
`":"`	Dotted line	
`"-."`	Dash-dotted line	

Marker	Description	Resulting Marker
`"o"`	Circle	
`"+"`	Plus sign	
`"*"`	Asterisk	
`"."`	Point	
`"x"`	Cross	
`"_"`	Horizontal line	
`"|"`	Vertical line	
`"square"`	Square	
`"diamond"`	Diamond	
`"^"`	Upward-pointing triangle	
`"v"`	Downward-pointing triangle	
`">"`	Right-pointing triangle	
`"<"`	Left-pointing triangle	
`"pentagram"`	Pentagram	
`"hexagram"`	Hexagram	

Color Name	Short Name	RGB Triplet	Appearance
"red"	"r"	[1 0 0]	
"green"	"g"	[0 1 0]	
"blue"	"b"	[0 0 1]	
"cyan"	"c"	[0 1 1]	
"magenta"	"m"	[1 0 1]	
"yellow"	"y"	[1 1 0]	
"black"	"k"	[0 0 0]	
"white"	"w"	[1 1 1]	

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Note These properties are only a subset. For a full list, see Line Properties.

Example: `'Hatches','//'`

`Hatches` — Hatch style

`'/'` (default) | `'\'` | `'|'`

Hatch style, specified as `'/'`, `'\'`, or `'|'`. The length of the string determines the hatch spacing. The more hatches specified, the closer the spacing. For example:

- For half-spaced forward slants, use `'Hatches','//'`.
- For a single-spaced perpendicular slant, use `'Hatches','|'`.
- For third-spaced backward slants, use `'Hatches','\\\'`.

Data Types: `char` | `string`

`FlipBoundary` — Flip boundary hatch angle

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

Flip boundary hatch angle by 180 degrees, specified as `'off'` or `'on'`.

- `'off'` — Do not flip the hatch angle.
- `'on'` — Flip the hatch angle by 180 degrees.

Data Types: `char` | `string`

`HatchTangency` — Hatch angle tangency

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

Hatch angle tangency, specified as `'on'` or `'off'`.

- `'on'` — Hatch angle is relative to the tangent of the line segment. The function determines the tangency by evaluating the line integral traversing from the start to the end of the *x* and *y* data.
- `'off'` — Hatch angle is relative to 0.

Data Types: `char` | `string`

`HatchLength` — Length of hatch segments
numeric scalar

Length of hatch segments, specified as a numeric scalar.

Data Types: `double`

`HatchAngle` — Angle of hatch segments
numeric scalar

Angle of hatch segments, specified as a numeric scalar. The function automatically calculates the hatch angle if you specify a style for `'Hatches'`.

Data Types: `char` | `string`

`HatchSpacing` — Spacing between hatch segments
numeric scalar

Spacing between hatch segments, specified as a numeric scalar.

Data Types: `char` | `string`

`LineStyle` — Line style
`"-"` (default) | `"--"` | `":"` | `"-."` | `"none"`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`"-"`	Solid line	————
`"--"`	Dashed line	- — — — —
`":"`	Dotted line	.....
`"-."`	Dash-dotted line	—·—·—·—·-
`"none"`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

The line width cannot be thinner than the width of a pixel. If you set the line width to a value that is less than the width of a pixel on your system, the line displays as one pixel wide.

Output Arguments

`bline` — boundary line object
`Aero.graphics.primitive.BoundaryLine` object

Boundary line object, returned as an `Aero.graphics.primitive.BoundaryLine` object.

Version History

Introduced in R2021b

See Also

`altitudeEnvelopeContour` | `line` | `shortPeriodCategoryAPlot` | `shortPeriodCategoryBPlot` | `shortPeriodCategoryCPlot` | `plot` | `contour` | `gca`

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.

Version History

Introduced in R2007a

See Also

`Aero.Camera`

camheading

Package: `matlabshared.satellitescenario`

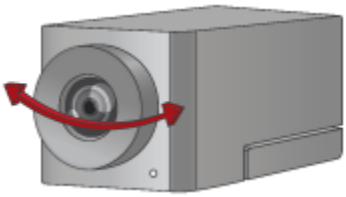
Set or get heading angle of camera for Satellite Scenario Viewer

Syntax

```
camheading(viewer,heading)
outHeading = camheading(viewer,___)
```

Description

`camheading(viewer,heading)` sets the heading angle of the camera for the specified Satellite Scenario Viewer. Setting the heading angle shifts the camera left or right about its *z*-axis.



`outHeading = camheading(viewer,___)` returns the heading angle of the camera. If the second input is `heading`, then the function sets the output equal to the input heading.

Examples

Set Camera Heading Angle of Satellite Scenario Viewer

Create a satellite scenario object.

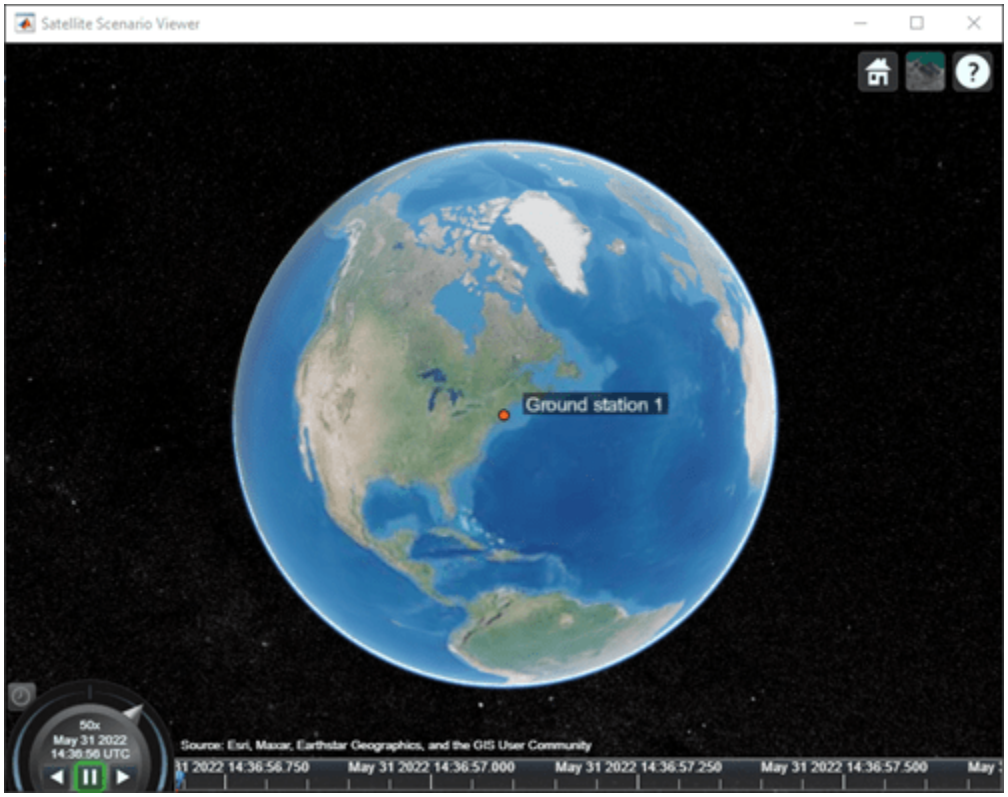
```
sc = satelliteScenario;
```

Add a ground station to the scenario.

```
latitude = 42.3001;                          % degrees
longitude = -71.3504;                        % degrees
groundStation(sc,latitude+0.05,longitude);
```

Launch the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```

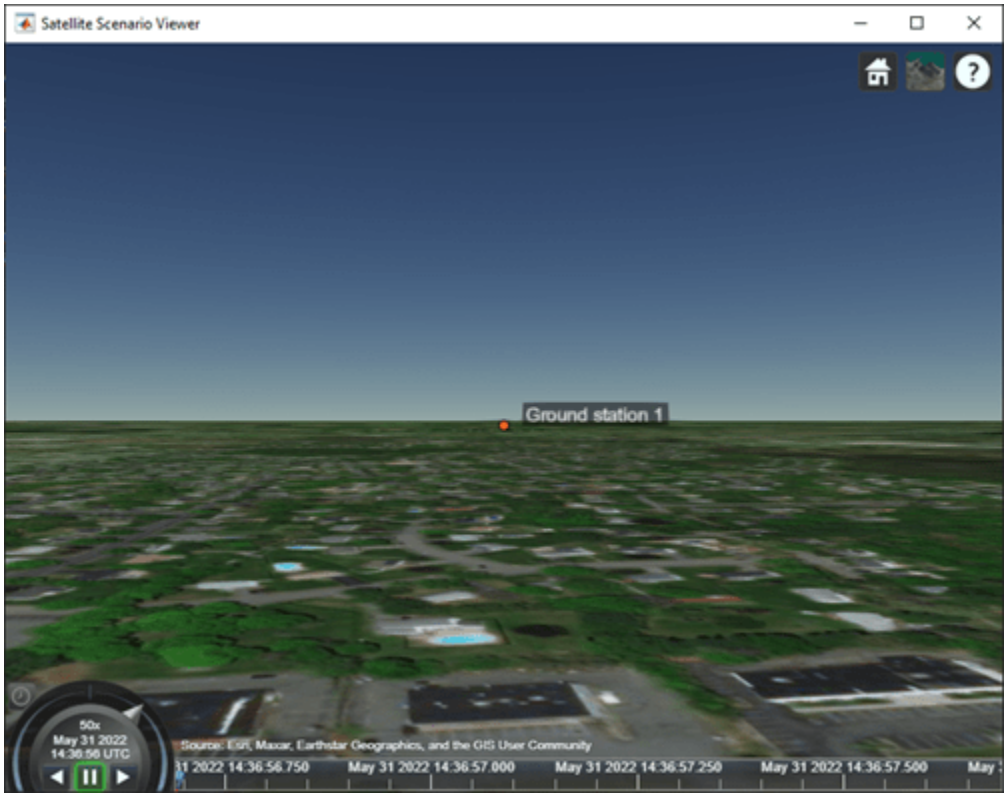


In the Satellite Scenario Viewer, set the height of the camera to 50 meters.

```
height = 50;                          % meters
campos(v,latitude,longitude,height);
pause(2);
```

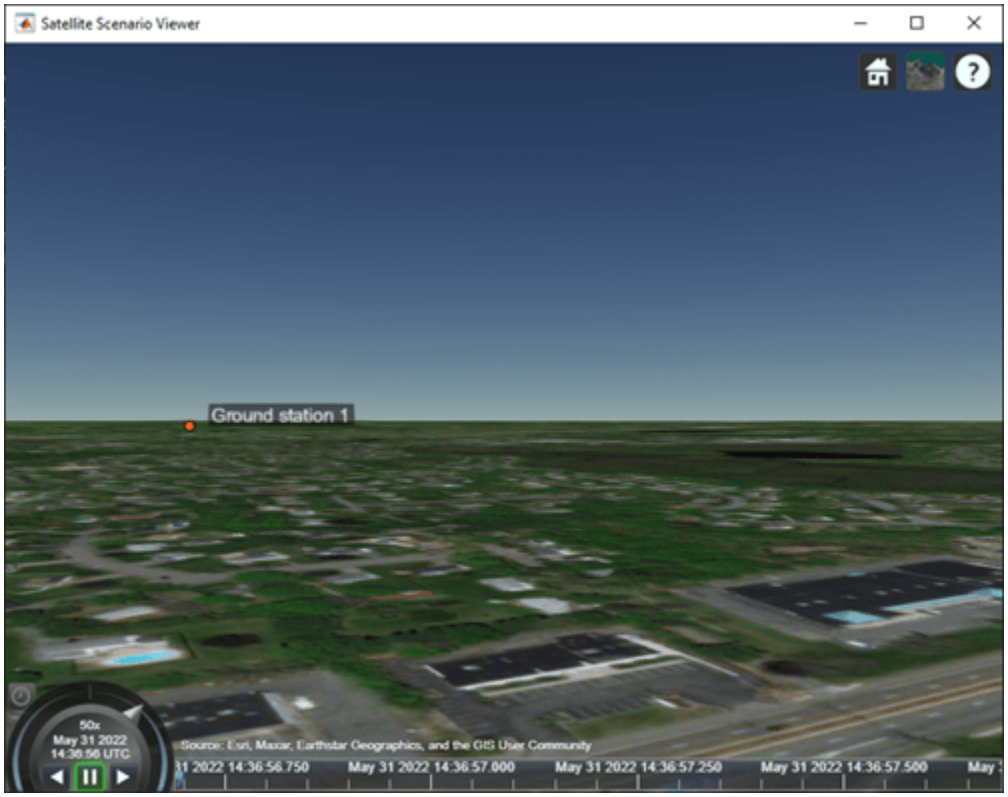
Set the pitch angle of the camera to 0 degrees.

```
pitch = 0;
campitch(v,pitch);
pause(2);
```



Set the heading angle of the camera to 20 degrees.

```
heading = 20;          % degrees
camheading(v,heading);
```



Input Arguments

`viewer` — Satellite Scenario Viewer

`satelliteScenarioViewer` object

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object.[2]

`heading` — Heading angle of camera

`360` (default) | numeric scalar in the range [–360, 360]

Heading angle of the camera in degrees, specified as a scalar value in the range [–360, 360].

Output Arguments

`outHeading` — Camera heading angle

numeric scalar in the range [-360, 360]

Camera heading angle in degrees, returned as a numeric scalar.

Tips

- When the pitch angle is near –90 (the default value) or 90 degrees, the camera loses one rotational degree of freedom. As a result, when you change the roll angle, the heading angle might

2 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

change instead. This phenomenon is called gimbal lock. To avoid the effects of gimbal lock, call the `camheading` function instead of the `camroll` function.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `camroll` | `campitch` | `campos` | `hideAll` | `camtarget` | `camheight`

Topics

“Satellite Scenario Key Concepts” on page 2-65

camheight

Package: `matlabshared.satellitescenario`

Set or get height of camera for Satellite Scenario Viewer

Syntax

```
camheight(viewer,height)
heightOut = camheight(viewer,___)
```

Description

`camheight(viewer,height)` sets the ellipsoidal height of the camera for the specified Satellite Scenario Viewer.

`heightOut = camheight(viewer,___)` returns the ellipsoidal height of the camera. If the second input is `height`, then the function sets the output equal to the input height.

Examples

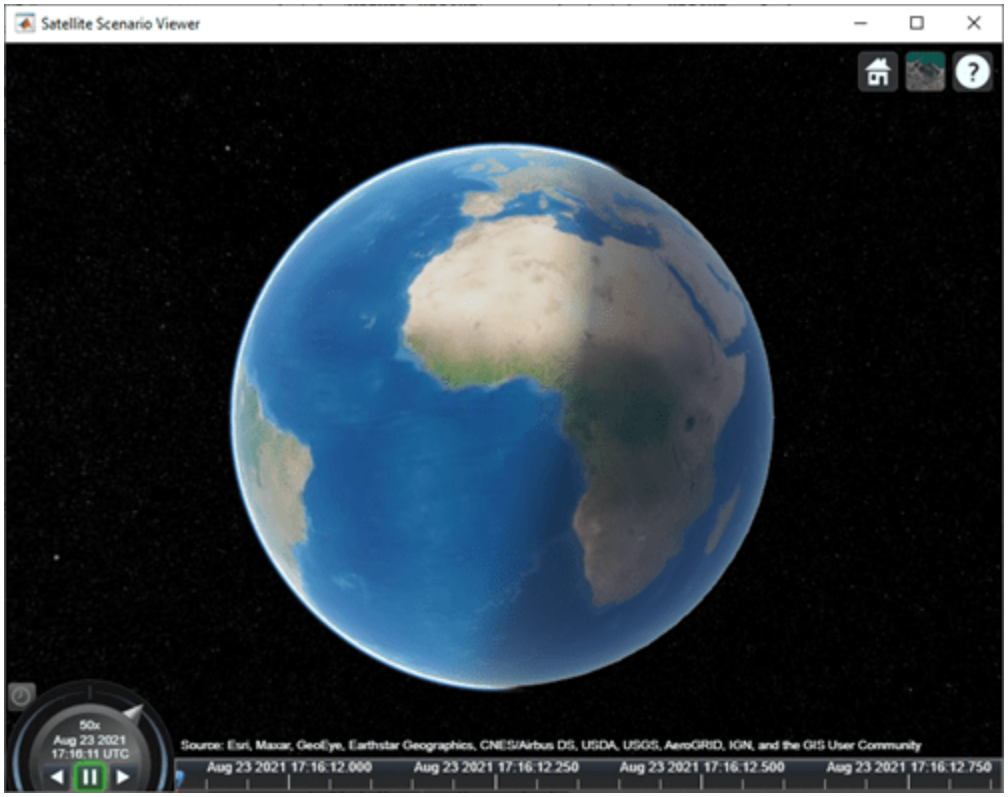
Retrieve Camera Height of Satellite Scenario Viewer

Create a satellite scenario object.

```
sc = satelliteScenario;
```

Launch the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```



Retrieve the height of the camera in the Satellite Scenario Viewer.

```
height = camheight(v)
```

```
height = 15000000
```

Input Arguments

`viewer` — Satellite Scenario Viewer

`satelliteScenarioViewer` object

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object.[3]

`height` — Ellipsoidal height of camera

`15000000` (default) | numeric scalar

Ellipsoidal height of the camera in meters, specified as a numeric scalar. `satelliteScenarioViewer` objects use the WGS84 reference ellipsoid. For more information about ellipsoidal height, see “Geodetic Coordinates” on page 2-65.

If you specify the height so that the camera is level with or below the surface of the Earth, then the `camheight` function sets the height to a value one meter above the surface.

3 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `camroll` | `campitch` | `campos` | `hideAll` | `camtarget` | `camheading`

Topics

“Satellite Scenario Key Concepts” on page 2-65

campitch

Package: `matlabshared.satellitescenario`

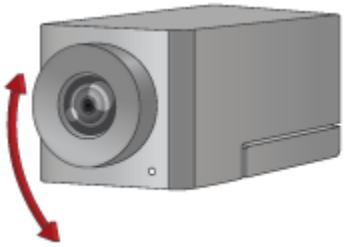
Set or get pitch angle of camera for Satellite Scenario Viewer

Syntax

```
campitch(viewer,pitch)
outPitch = campitch(viewer,___)
```

Description

`campitch(viewer,pitch)` sets the pitch angle of the camera for the specified Satellite Scenario Viewer. Setting the pitch angle tilts the camera up or down about its *y*-axis.



`outPitch = campitch(viewer,___)` returns the pitch angle of the camera. If the second input is `pitch`, then the function sets the output equal to the input pitch.

Examples

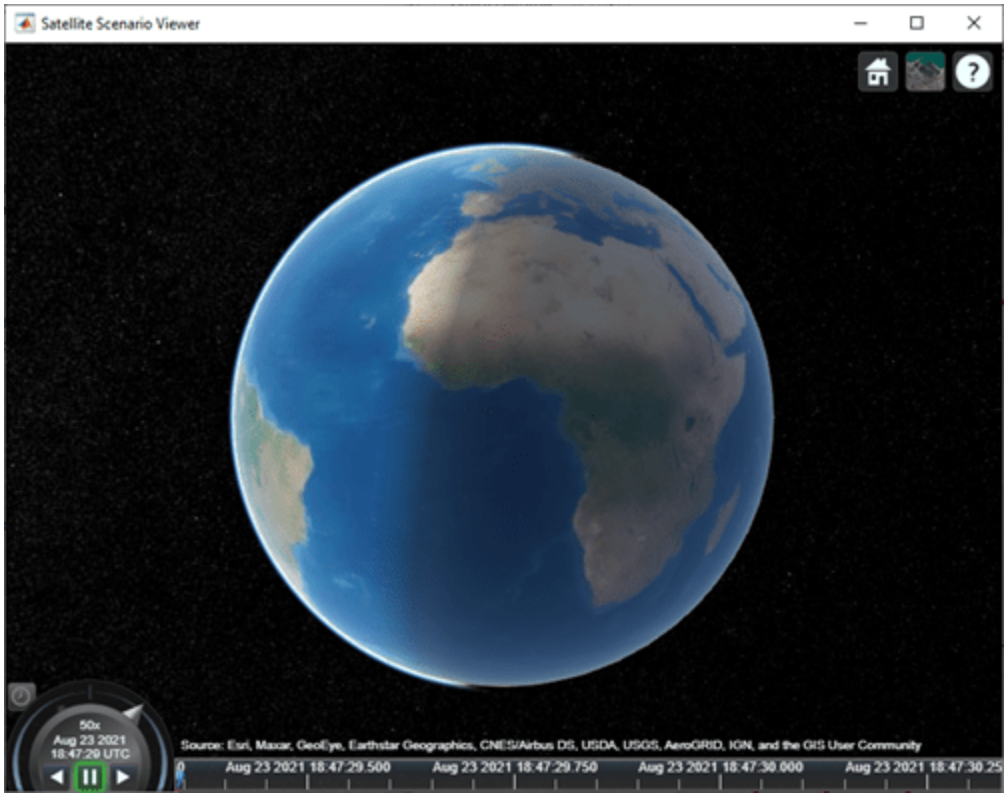
Set Camera Pitch Angle of Satellite Scenario Viewer

Create a satellite scenario object.

```
sc = satelliteScenario;
```

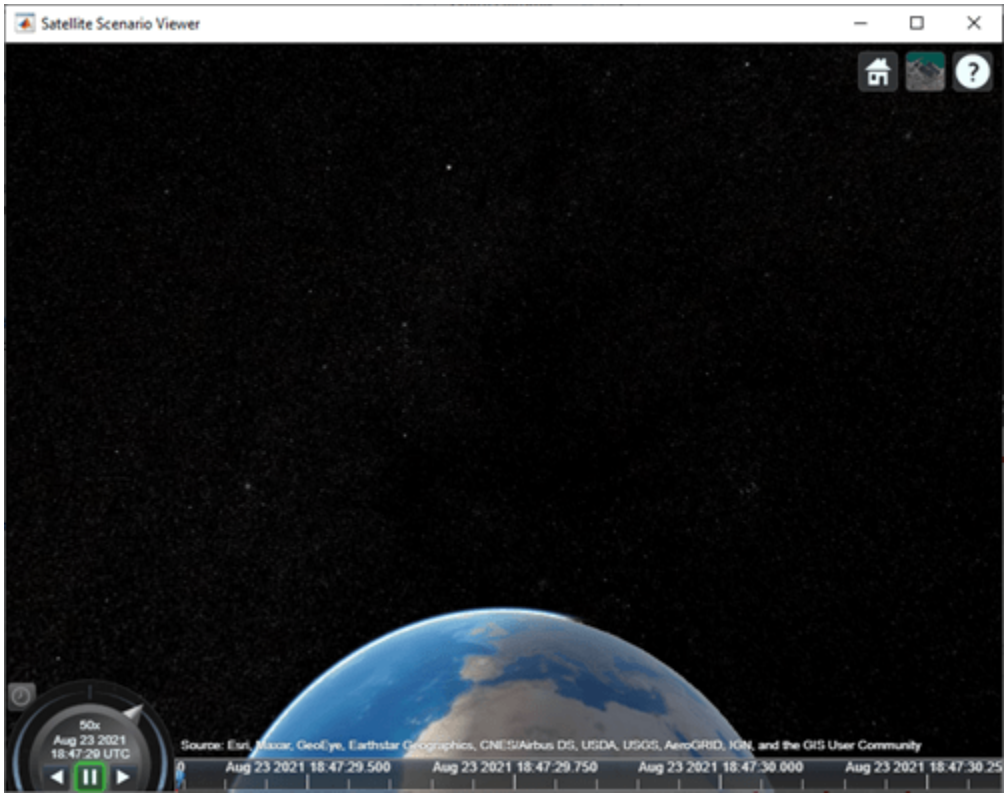
Launch the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```



In the Satellite Scenario Viewer, set the pitch angle of the camera to −60 degrees.

```
pitch = -60;      % degrees
campitch(v,pitch);
```



Input Arguments

`viewer` — Satellite Scenario Viewer

`satelliteScenarioViewer` object

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object.[4]

`pitch` — Pitch angle of camera

-90 (default) | scalar the in the range [–90, 90]

Pitch angle of the camera in degrees, specified as a scalar in the range [–90, 90]. By default, the pitch angle is –90 degrees, which means that camera points directly toward the surface of the globe.

Output Arguments

`outPitch` — Camera pitch angle

numeric scalar in the range [-90, 90]

Camera pitch angle in degrees, returned as a numeric scalar.

4 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Tips

- When the pitch angle is near –90 (the default value) or 90 degrees, the camera loses one rotational degree of freedom. As a result, when you change the roll angle, the heading angle might change instead. This phenomenon is called gimbal lock. To avoid the effects of gimbal lock, call the `camheading` function instead of the `camroll` function.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `camroll` | `campos` | `hideAll` | `camtarget` | `camheading`

Topics

“Satellite Scenario Key Concepts” on page 2-65

campos

Package: `matlabshared.satellitescenario`

Set or get position of camera for Satellite Scenario Viewer

Syntax

```
campos(viewer,lat,lon)
campos(viewer,lat,lon,height)
campos(viewer)
[latOut,lonOut,heightOut] = campos( ___ )
```

Description

`campos(viewer,lat,lon)` sets the latitude and longitude of the camera for the specified Satellite Scenario Viewer.

`campos(viewer,lat,lon,height)` sets the latitude, longitude, and ellipsoidal height of the camera. If you want to set only the height of the camera, use the `camheight` function instead.

`campos(viewer)` displays the latitude, longitude, and ellipsoidal height of the camera as a three-element vector. `satelliteScenarioViewer` objects use the WGS84 reference ellipsoid.

`[latOut,lonOut,heightOut] = campos( ___ )` sets the position and then returns the latitude, longitude, and height of the camera. Specify any input argument combinations from previous syntaxes.

Examples

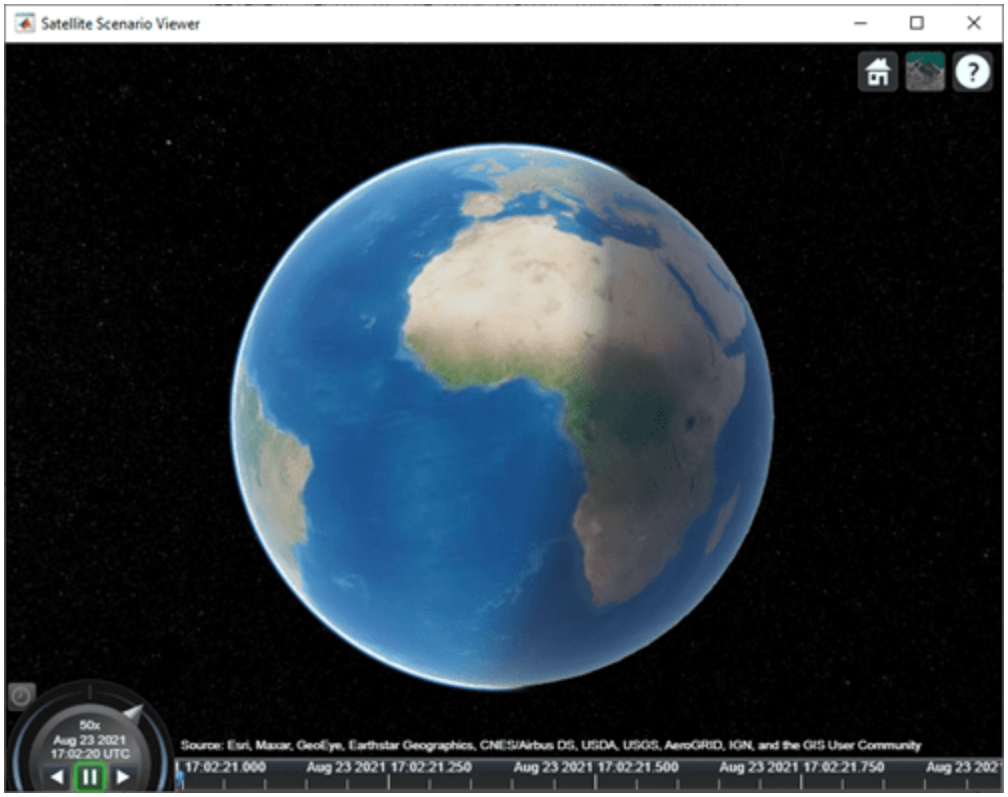
Reposition Camera of Satellite Scenario Viewer

Create a satellite scenario object.

```
sc = satelliteScenario;
```

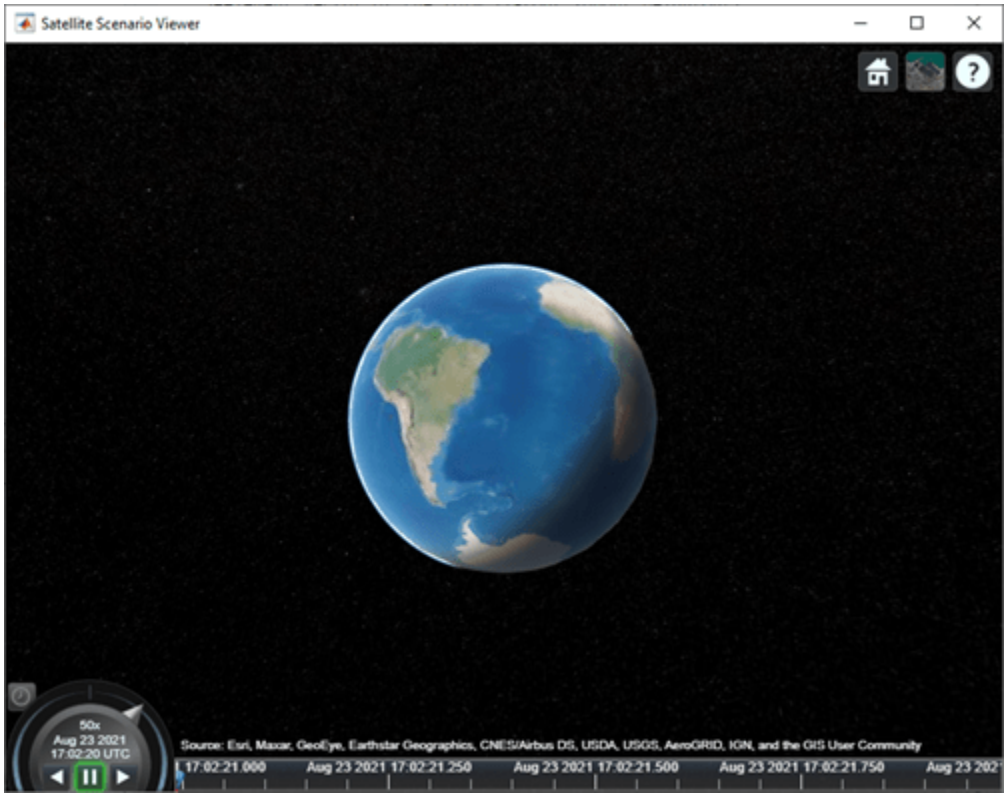
Launch the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```



In the Satellite Scenario Viewer, set the latitude and longitude of the camera to −30 degrees and the height of the camera to 30,000 km.

```
latitude = -30;                    % degrees
longitude = -30;                   % degrees
height = 30000000;                 % meters
campos(v,latitude,longitude,height)
```



Input Arguments

`viewer` — Satellite Scenario Viewer

`satelliteScenarioViewer` object

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object.[5]

`lat` — Geodetic latitude of camera

`0` (default) | scalar in the range [-90, 90].

Geodetic latitude of the camera in degrees, specified as a scalar in the range [–90, 90].

`lon` — Geodetic longitude of camera

`0` (default) | scalar in the range [-360, 360].

Geodetic longitude of the camera in degrees, specified as a scalar in the range [–360, 360].

`height` — Ellipsoidal height of camera

`15000000` (default) | numeric scalar

Ellipsoidal height of the camera in meters, specified as a numeric scalar.

If you specify the height such that the camera is level with or below the surface of the Earth, then the `campos` function sets the height to a value one meter above the surface.

5 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Output Arguments

`latOut` — Geodetic latitude of camera
numeric scalar

Geodetic latitude of the camera in degrees, returned as a numeric scalar.

`lonOut` — Geodetic longitude of camera
numeric scalar

Geodetic longitude of the camera in degrees, returned as a numeric scalar.

`heightOut` — Ellipsoidal height of camera
numeric scalar

Ellipsoidal height of the camera in meters, returned as a numeric scalar. For more information about ellipsoidal height, see “Geodetic Coordinates” on page 2-65.

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `play` | `hide` | `camroll` | `campitch` | `hideAll` | `camtarget` | `camheight` | `camheading`

Topics
“Satellite Scenario Key Concepts” on page 2-65

camroll

Package: `matlabshared.satellitescenario`

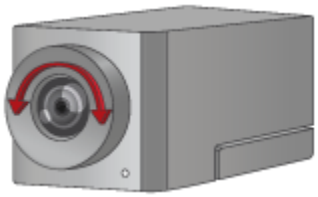
Set or get roll angle of camera for Satellite Scenario Viewer

Syntax

```
camroll(viewer,roll)
outRoll = camroll(viewer,___)
```

Description

`camroll(viewer,roll)` sets the roll angle of the camera for the Satellite Scenario Viewer. Setting the roll angle rotates the camera around its *x*-axis.



`outRoll = camroll(viewer,___)` returns the roll angle of the camera. If the second input is `roll`, then the function sets the output equal to the input roll.

Examples

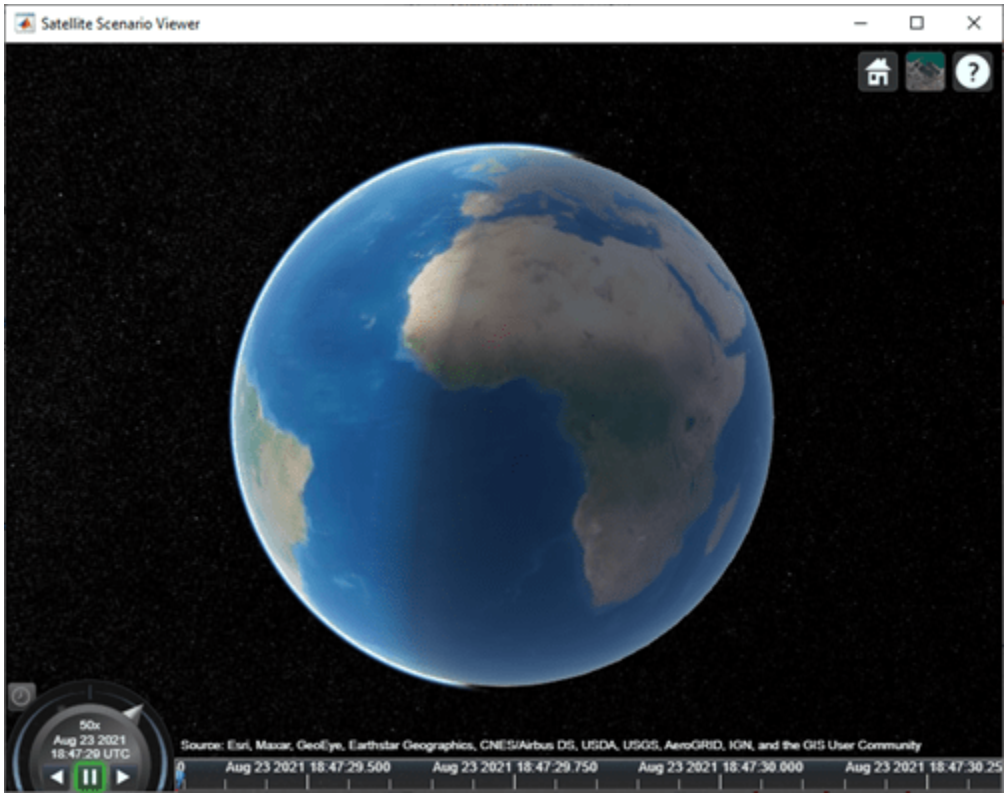
Set Camera Roll Angle of Satellite Scenario Viewer

Create a satellite scenario object.

```
sc = satelliteScenario;
```

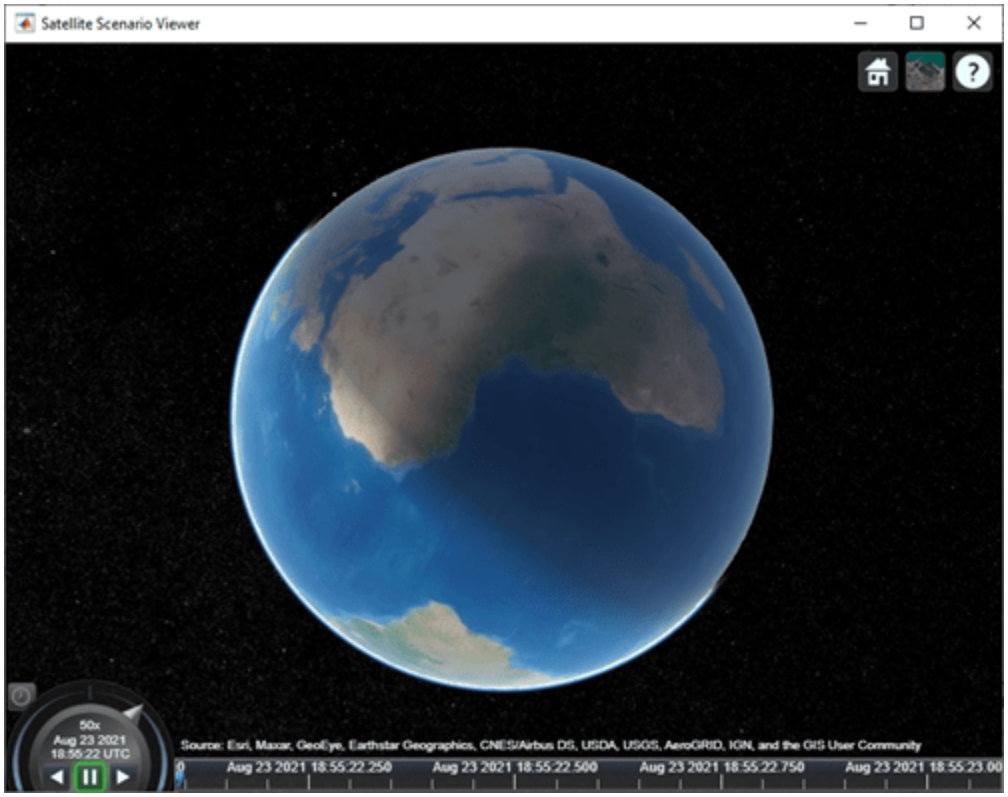
Launch the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```



Set the roll angle of the camera in the Satellite Scenario Viewer to 60 degrees.

```
roll = 60;      % degrees
camroll(v,roll);
```



Input Arguments

`viewer` — Satellite Scenario Viewer

`satelliteScenarioViewer` object

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object.[6]

`roll` — Roll angle of camera

numeric scalar in the range [–360, 360]

Roll angle of the camera in degrees, specified as a scalar in the range [–360, 360].

Output Arguments

`outRoll` — Camera roll angle

numeric scalar in the range [-360, 360]

Camera roll angle in degrees, returned as a numeric scalar.

Tips

- When the pitch angle is near –90 (the default value) or 90 degrees, the camera loses one rotational degree of freedom. As a result, when you change the roll angle, the heading angle might

6 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

change instead. This phenomenon is called gimbal lock. To avoid the effects of gimbal lock, call the `camheading` function instead of the `camroll` function.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `campitch` | `campos` | `hideAll` | `camtarget` | `camheight` | `camheading`

Topics

“Satellite Scenario Key Concepts” on page 2-65

camtarget

Package: `matlabshared.satellitescenario`

Set camera target for Satellite Scenario Viewer

Syntax

```
camtarget(viewer,target)
```

Description

`camtarget(viewer,target)` focuses the camera on the input satellite or ground station. The camera follows `target`. You can reset the camera target by using `camtarget` with a different satellite or ground station or by double-clicking anywhere in the map.

Examples

Set Camera Target to Satellite

Create a satellite scenario object.

```
sc = satelliteScenario;
```

Add a satellite to the scenario.

```
semiMajorAxis = 10000000;                                % meters
eccentricity = 0;
inclination = 0;                                         % degrees
rightAscensionOfAscendingNode = 0;                       % degrees
argumentOfPeriapsis = 0;                                 % degrees
trueAnomaly = 0;                                         % degrees
sat = satellite(sc,semiMajorAxis,eccentricity, ...
      inclination,rightAscensionOfAscendingNode, ...
      argumentOfPeriapsis,trueAnomaly);
```

Launch the Satellite Scenario Viewer.

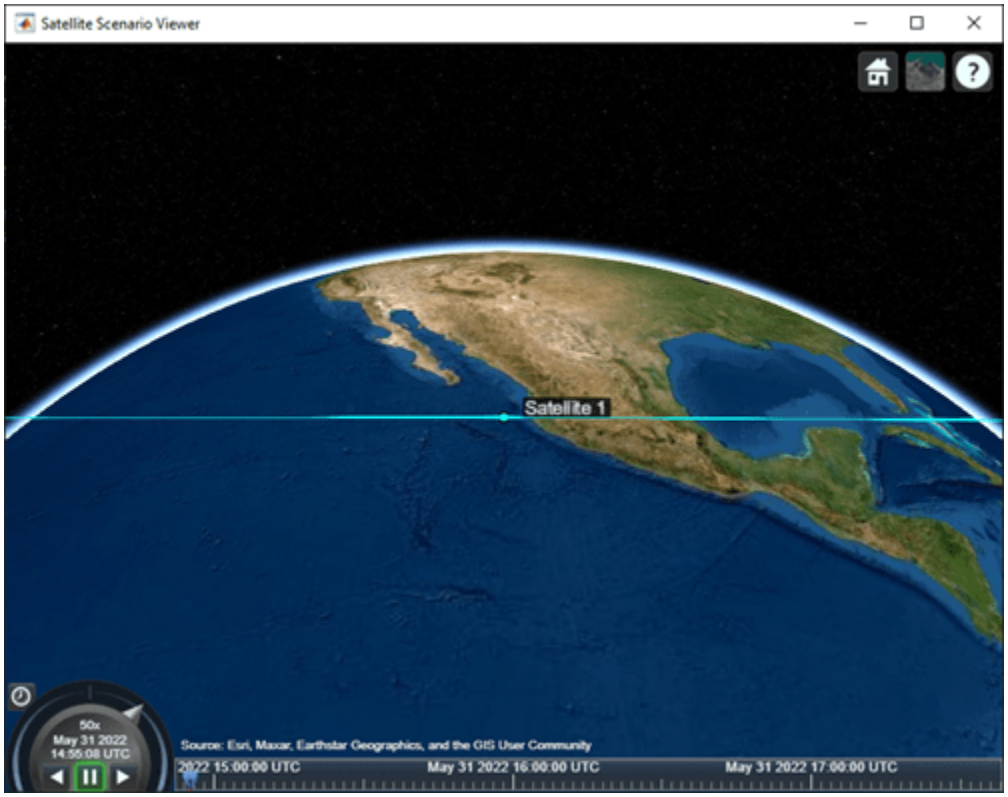
```
v = satelliteScenarioViewer(sc);
```

Play the scenario in the viewer.

```
play(sc,"Viewer",v);
```

Set the camera target to the satellite.

```
camtarget(v,sat);
```



Input Arguments

`viewer` — Satellite Scenario Viewer

`satelliteScenarioViewer` object

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object.[7]

`target` — Camera target

`Satellite` object | `GroundStation` object

Camera target, specified as a scalar `Satellite` or `GroundStation` object.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `camroll` | `campitch` | `campos` | `hideAll` | `camheight` | `camheading`

7 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Topics
“Satellite Scenario Key Concepts” on page 2-65

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')
```

Version History

Introduced in R2008b

See Also

`SetTimer`

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) | on/off logical value

Visibility of the climb indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`. 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.

`ContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object created using the `uicontextmenu` function. Use this property to display a context menu when you right-click on a component.

`Enable` — Operational state of climb indicator

`'on'` (default) | on/off logical value

Operational state of climb indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- 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

Element	Description
`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 = uigridlayout([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 “Callbacks in App Designer”.

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 “Callbacks in App Designer”.

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) | on/off logical value

Callback interruption, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value

of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

This 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.

MATLAB determines callback interruption behavior whenever it executes a command that processes the callback queue. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines if the interruption occurs:

- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the object that owns the interrupting callback determines if the interrupting callback is discarded or added to the callback queue.
- If the value of `Interruptible` is `'on'`, then the interruption occurs. The next time MATLAB processes the callback queue, it stops the execution of the running callback and executes the interrupting callback. After the interrupting callback completes, MATLAB then resumes executing the running callback.

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.
- If the interrupting callback is owned by a `Timer` object, then the callback executes according to schedule regardless of the `Interruptible` property value.

Note 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.

The BusyAction property determines callback queuing behavior only when both of these conditions are met:

- The running callback contains a command that processes the callback queue, such as drawnow, figure, uifigure, getframe, waitfor, or pause.
- The value of the Interruptible property of the object that owns the running callback is 'off'.

Under these conditions, the BusyAction property of the object that owns the interrupting callback determines how MATLAB handles the interrupting callback. 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

on/off logical value

This property is read-only.

Deletion status, returned as an on/off logical value of type matlab.lang.OnOffSwitchState.

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”.

Version History

Introduced in R2018b

See Also

`uiaeroclimb`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

conicalSensor

Package: `matlabshared.satellitescenario`

Add conical sensor to satellite scenario

Syntax

```
conicalSensor(parent)
conicalSensor(parent,Name=Value)
sensor = conicalSensor( ___ )
```

Description

`conicalSensor(parent)` adds a `ConicalSensor` object to each parent in the vector `parent` using default parameters. `parent` can be `Satellite` , `GroundStation` , or `Gimbal`.

`conicalSensor(parent,Name=Value)` adds conical sensors to the parents in `parent` using additional parameters specified by optional name-value arguments. For example, `'MaxViewAngle'=90` specifies a field of view angle of 90 degrees.

`sensor = conicalSensor( ___ )` returns added conical sensors as a row vector `sensor`. Specify any input argument combination from previous syntaxes.

Examples

Calculate Maximum Revisit Time of Satellite

Create a satellite scenario with a start time of 15-June-2021 8:55:00 AM UTC and a stop time of five days later. Set the simulation sample time to `60` seconds.

```
startTime = datetime(2021,6,21,8,55,0);
stopTime = startTime + days(5);
sampleTime = 60;                                        % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc =
  satelliteScenario with properties:

         StartTime: 21-Jun-2021 08:55:00
          StopTime: 26-Jun-2021 08:55:00
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add a satellite to the scenario using Keplerian orbital elements.

```
semiMajorAxis = 7878137;                                                                % me
eccentricity = 0;
inclination = 50;                                                                       % deg
rightAscensionOfAscendingNode = 0;                                                      % deg
argumentOfPeriapsis = 0;                                                                % deg
trueAnomaly = 50;                                                                       % deg
sat = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly)

sat =
  Satellite with properties:

              Name:  Satellite 1
                ID:  1
     ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
            Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
       Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
          Receivers:  [1x0 satcom.satellitescenario.Receiver]
           Accesses:  [1x0 matlabshared.satellitescenario.Access]
         GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
              Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
    OrbitPropagator:  sgp4
        MarkerColor:  [0.059 1 1]
         MarkerSize:  6
          ShowLabel:  true
     LabelFontColor:  [1 1 1]
      LabelFontSize:  15
```

Add a ground station, which represents the location to be photographed, to the scenario.

```
gs = groundStation(sc,Name="Location to Photograph", ...
    Latitude=42.3001,Longitude=-71.3504)                % degrees

gs =
  GroundStation with properties:

                Name:  Location to Photograph
                  ID:  2
            Latitude:  42.3 degrees
           Longitude:  -71.35 degrees
            Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
          MarkerColor:  [1 0.4118 0.1608]
           MarkerSize:  6
            ShowLabel:  true
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
```

Add a gimbal to the satellite. You can steer this gimbal independently of the satellite.

```
g = gimbal(sat)
```

```
g =
  Gimbal with properties:

               Name:  Gimbal 3
                 ID:  3
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
```

Track the location to be photographed using the gimbal.

```
pointAt(g,gs);
```

Add a conical sensor to the gimbal. This sensor represents the camera. Set the field of view to 60 degrees.

```
camSensor = conicalSensor(g,MaxViewAngle=60)
```

```
camSensor =
  ConicalSensor with properties:

               Name:  Conical sensor 4
                 ID:  4
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
        MaxViewAngle:  60 degrees
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         FieldOfView:  [0x0 matlabshared.satellitescenario.FieldOfView]
```

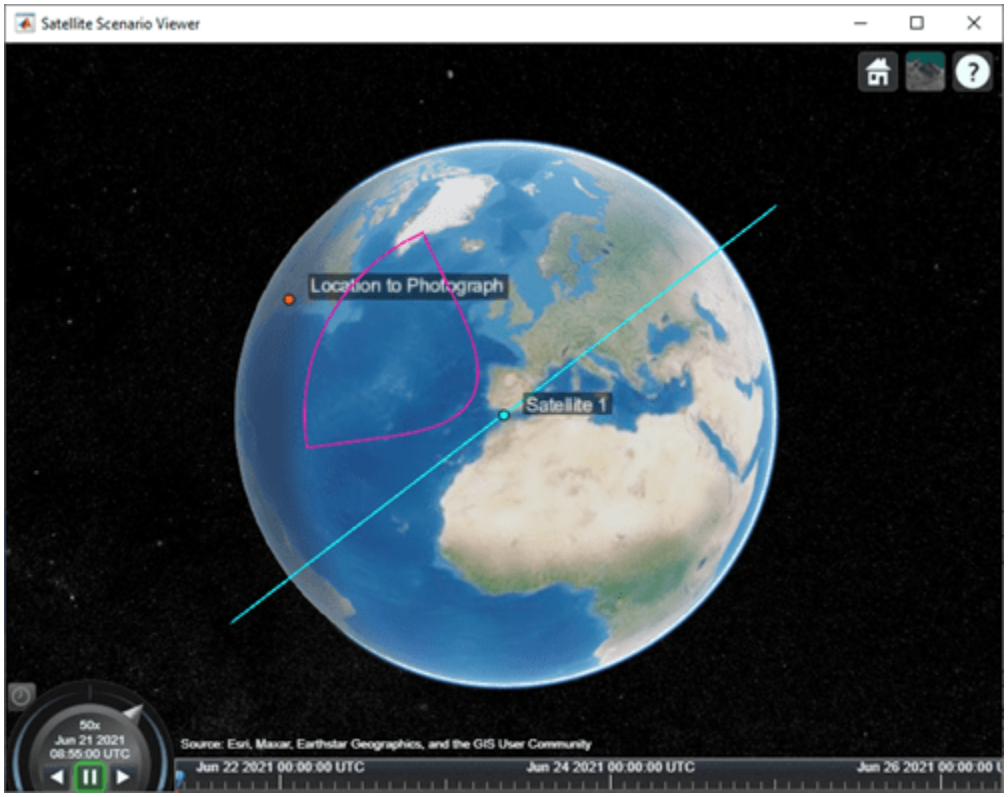
Add access analysis to the conical sensor between the camera and the location to be photographed.

```
ac = access(camSensor,gs)
```

```
ac =
  Access with properties:

    Sequence:  [4 2]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

Visualize the field of view of the camera by using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
fieldOfView(camSensor);
```



Determine the intervals during which the camera can see the geographical site.

```
t = accessIntervals(ac)
```

```
t=35×8 table
          Source                 Target            IntervalNumber        StartTime
    ________________    ________________________    ______________    ____________________

    "Conical sensor 4"    "Location to Photograph"          1          21-Jun-2021 10:38:00
    "Conical sensor 4"    "Location to Photograph"          2          21-Jun-2021 12:36:00
    "Conical sensor 4"    "Location to Photograph"          3          21-Jun-2021 14:37:00
    "Conical sensor 4"    "Location to Photograph"          4          21-Jun-2021 16:41:00
    "Conical sensor 4"    "Location to Photograph"          5          21-Jun-2021 18:44:00
    "Conical sensor 4"    "Location to Photograph"          6          21-Jun-2021 20:46:00
    "Conical sensor 4"    "Location to Photograph"          7          21-Jun-2021 22:50:00
    "Conical sensor 4"    "Location to Photograph"          8          22-Jun-2021 09:51:00
    "Conical sensor 4"    "Location to Photograph"          9          22-Jun-2021 11:46:00
    "Conical sensor 4"    "Location to Photograph"         10          22-Jun-2021 13:46:00
    "Conical sensor 4"    "Location to Photograph"         11          22-Jun-2021 15:50:00
    "Conical sensor 4"    "Location to Photograph"         12          22-Jun-2021 17:53:00
    "Conical sensor 4"    "Location to Photograph"         13          22-Jun-2021 19:55:00
    "Conical sensor 4"    "Location to Photograph"         14          22-Jun-2021 21:58:00
    "Conical sensor 4"    "Location to Photograph"         15          23-Jun-2021 10:56:00
    "Conical sensor 4"    "Location to Photograph"         16          23-Jun-2021 12:56:00
      ⋮
```

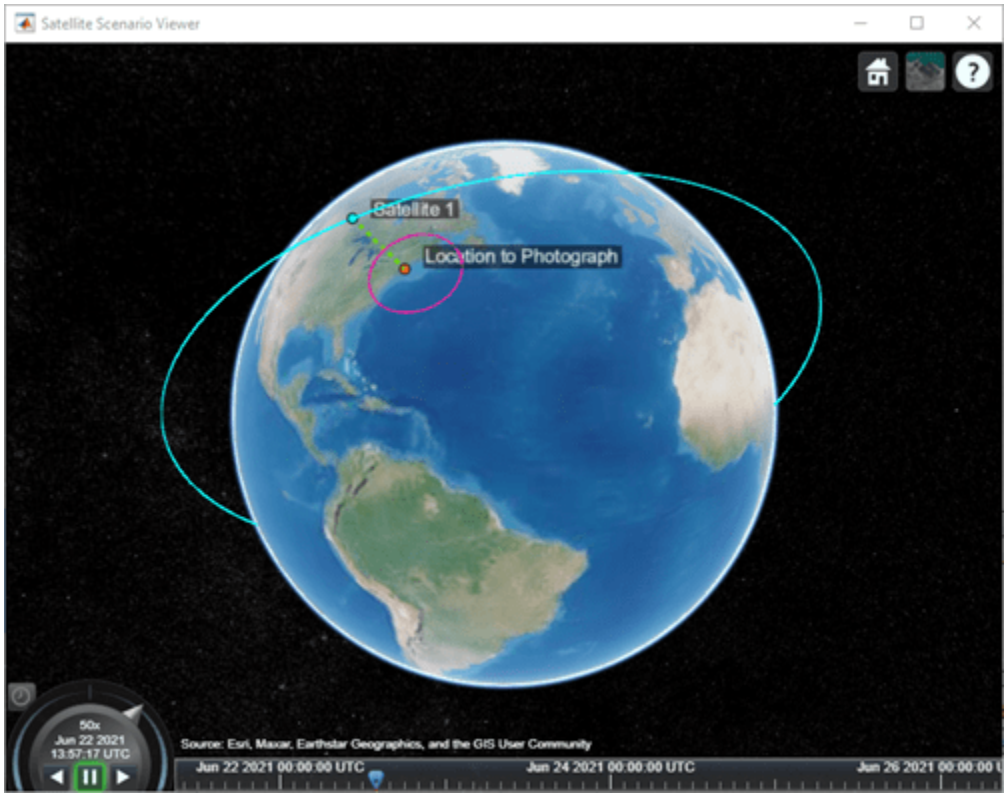
Calculate the maximum revisit time in hours.

```
startTimes = t.StartTime;
endTimes = t.EndTime;
revisitTimes = hours(startTimes(2:end) - endTimes(1:end-1));
maxRevisitTime = max(revisitTimes)                                % hours
```

```
maxRevisitTime = 12.6667
```

Visualize the revisit times that the camera photographs of the location.

```
play(sc);
```



Input Arguments

`parent` — Element of scenario to which conical sensor is added

scalar | vector

Element of scenario in which to add the conical sensor, specified as a scalar or vector of satellites, ground stations, or gimbals. The number of conical sensors to add is determined by the size of the inputs.

- If `parent` is a scalar, all conical sensors are added to the parent.
- If `parent` is a vector of parents and the number of conical sensors specified is one, that conical sensor is added to each parent.
- If `parent` is a vector of parents and the number of conical sensors specified is more than one, the number of conical sensors must equal the number of parents in `parent` and each parent gets one conical sensor.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MountingAngle'=[20;35;10]` sets the yaw, pitch, and roll angles of the conical sensor to 20, 35, and 10 degrees, respectively.

Note The size of the name-value arguments defines the number of conical sensors that you can specify. To understand how to specify multiple conical sensors, refer to these properties.

`Name` — Conical sensor name

`"Conical sensor idx"` (default) | string scalar | string vector | character vector | cell array of character vectors

You can set this property only when calling the `conicalSensor` function. After you call the `conicalSensor` function, this property is read-only.

Conical sensor name, specified as a name-value argument consisting of `'Name'` and a string scalar, string vector, character vector, or a cell array of character vectors.

- If you are adding only one conical sensor, specify `Name` as a string scalar or a character vector.
- If you are adding multiple conical sensors, specify `Name` as a string scalar, character vector, string vector, or a cell array of character vectors. All conical sensors that you add as a string scalar or a character vector are assigned the same specified name. The number of elements in the string vector or cell array of character vectors must equal the number of conical sensors that you are adding. Each conical sensor is assigned the corresponding name from the vector or cell array.

In the default value, *idx* is the ID assigned by the satellite scenario.

Data Types: `char` | `string`

`MountingLocation` — Mounting location with respect to parent

`[0; 0; 0]` (default) | three-element vector | matrix

Mounting location with respect to the parent object in meters, specified as a three-element vector or a matrix. The position vector is specified in the body frame of the input `parent`.

- If you are adding one conical sensor, `MountingLocation` is a three-element vector. The elements specify the *x*, *y*, and *z* components of the Cartesian coordinates in the body frame of conical sensor.
- If you are adding multiple conical sensors, `MountingLocation` can be a three-element vector or a matrix. When specified as a vector, the same set of mounting locations are assigned to all specified conical sensors. When specified as a matrix, `MountingLocation` must contain three rows and the same number of columns as the conical sensors. The columns correspond to the mounting location of each specified conical sensor and the rows correspond to the mounting location coordinates in the parent body frame.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify the `MountingLocation` property only when `SimulationStatus` is `NotStarted`. You can use the

`restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Data Types: `double`

`MountingAngles` — Mounting orientation with respect to parent object
`[0; 0; 0]` (default) | three-element row vector of positive numbers | matrix

Mounting orientation with respect to parent object in degrees, specified as a three-element row vector of positive numbers. The elements of the vector correspond to yaw, pitch, and roll, in that order. Yaw, pitch, and roll are positive rotations about the *z*-axis, intermediate *y*-axis, and intermediate *x*-axis of the parent.

- If you are adding one conical sensor, `MountingAngles` is a three-element vector.
- If you are adding multiple conical sensors, `MountingAngles` can be a three-element vector or a matrix. When specified as a vector, the same set of mounting angles are assigned to all specified conical sensors. When specified as a matrix, `MountingAngles` must contain three rows and the same number of columns as the conical sensors. The columns correspond to the mounting angles of each specified conical sensor and the rows correspond to the yaw, pitch, and roll angles in the parent body frame.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify the `MountingAngles` property only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Example: `[0; 30; 60]`

Data Types: `double`

`MaxViewAngle` — Field of view angle
`30` (default) | scalar in the range [0, 180] | vector

Field of view angle in degrees, specified as a scalar in the range [0, 180] or a vector.

- If you add one conical sensor, `MaxViewAngle` must be a scalar.
- If you add multiple conical sensors, `MaxViewAngle` can be a scalar or a vector. When `MaxViewAngle` is a scalar, the same field of view angle is assigned to all conical sensors that you are adding. When `MaxViewAngle` is a vector, the length of `MaxViewAngle` must equal the number of conical sensors in the parent. Each element of `MaxViewAngle` is assigned to the specified corresponding conical sensor.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify `MaxViewAngle` while the `SimulationStatus` is `NotStarted` or `InProgress`.

Data Types: `double`

Output Arguments

`sensor` — Conical sensor
`row vector` object

Conical sensors attached to `parent`, returned as a row vector.

When the `AutoSimulate` property of the satellite scenario is `false`, you can call `conicalSensor` function only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `groundStation` | `access` | `gimbal` | `satellite`

Topics

“Satellite Scenario Key Concepts” on page 2-65

ConicalSensor

Conical sensor object belonging to satellite scenario

Description

`ConicalSensor` defines a conical sensor object belonging to a satellite scenario.

Creation

You can create the `ConicalSensor` object using the `conicalSensor` object function of the `Satellite`, `GroundStation`, or `Gimbal` objects.

Properties

`Name` — Conical sensor name
`"Conical sensor idx"` (default) | string scalar | string vector | character vector | cell array of character vectors

You can set this property only when calling the `conicalSensor` function. After you call the `conicalSensor` function, this property is read-only.

Conical sensor name, specified as a name-value argument consisting of `'Name'` and a string scalar, string vector, character vector, or a cell array of character vectors.

- If you are adding only one conical sensor, specify `Name` as a string scalar or a character vector.
- If you are adding multiple conical sensors, specify `Name` as a string scalar, character vector, string vector, or a cell array of character vectors. All conical sensors that you add as a string scalar or a character vector are assigned the same specified name. The number of elements in the string vector or cell array of character vectors must equal the number of conical sensors that you are adding. Each conical sensor is assigned the corresponding name from the vector or cell array.

In the default value, *idx* is the ID assigned by the satellite scenario.

Data Types: `char` | `string`

`ID` — ConicalSensor ID assigned by simulator
real positive scalar

This property is set internally by the simulator and is read-only.

ConicalSensor ID assigned by the simulator, specified as a positive scalar.

`MountingLocation` — Mounting location with respect to parent
`[0; 0; 0]` (default) | three-element vector | matrix

Mounting location with respect to the parent object in meters, specified as a three-element vector or a matrix. The position vector is specified in the body frame of the input `parent`.

- If you are adding one conical sensor, `MountingLocation` is a three-element vector. The elements specify the *x*, *y*, and *z* components of the Cartesian coordinates in the body frame of conical sensor.
- If you are adding multiple conical sensors, `MountingLocation` can be a three-element vector or a matrix. When specified as a vector, the same set of mounting locations are assigned to all specified conical sensors. When specified as a matrix, `MountingLocation` must contain three rows and the same number of columns as the conical sensors. The columns correspond to the mounting location of each specified conical sensor and the rows correspond to the mounting location coordinates in the parent body frame.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify the `MountingLocation` property only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Data Types: `double`

`MountingAngles` — Mounting orientation with respect to parent object
`[0; 0; 0]` (default) | three-element row vector of positive numbers | matrix

Mounting orientation with respect to parent object in degrees, specified as a three-element row vector of positive numbers. The elements of the vector correspond to yaw, pitch, and roll, in that order. Yaw, pitch, and roll are positive rotations about the *z*-axis, intermediate *y*-axis, and intermediate *x*-axis of the parent.

- If you are adding one conical sensor, `MountingAngles` is a three-element vector.
- If you are adding multiple conical sensors, `MountingAngles` can be a three-element vector or a matrix. When specified as a vector, the same set of mounting angles are assigned to all specified conical sensors. When specified as a matrix, `MountingAngles` must contain three rows and the same number of columns as the conical sensors. The columns correspond to the mounting angles of each specified conical sensor and the rows correspond to the yaw, pitch, and roll angles in the parent body frame.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify the `MountingAngles` property only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Example: `[0; 30; 60]`

Data Types: `double`

`MaxViewAngle` — Field of view angle
`30` (default) | scalar in the range [0, 180] | vector

Field of view angle in degrees, specified as a scalar in the range [0, 180] or a vector.

- If you add one conical sensor, `MaxViewAngle` must be a scalar.
- If you add multiple conical sensors, `MaxViewAngle` can be a scalar or a vector. When `MaxViewAngle` is a scalar, the same field of view angle is assigned to all conical sensors that you are adding. When `MaxViewAngle` is a vector, the length of `MaxViewAngle` must equal the number of conical sensors in the parent. Each element of `MaxViewAngle` is assigned to the specified corresponding conical sensor.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify `MaxViewAngle` while the `SimulationStatus` is `NotStarted` or `InProgress`.

Data Types: `double`

`Accesses` — Access analysis objects
row vector of `Access` objects

You can set this property only when calling `ConicalSensor`. After you call `ConicalSensor`, this property is read-only.

Access analysis objects, specified as a row vector of `Access` objects.

`CoordinateAxes` — Coordinate axes triad graphic object
`CoordinateAxes` object (default)

You can set this property only when calling `coordinateAxes`. After you call `coordinateAxes`, this property is read-only.

Coordinate axes triad graphic object, specified as `CoordinateAxes` object.

`FieldOfView` — Field of view objects
row vector of `FieldOfView` objects

You can set this property only when calling `ConicalSensor`. After you call `ConicalSensor`, this property is read-only.

Field of view objects, specified as a scalar of `FieldOfView` objects.

Note The properties `Name`, `MountingLocation` , `MountingAngles`, and `MaxViewAngle` can be specified as name-value arguments in the `conicalSensor` object function. The size of specified name-value pairs determines the number of conical sensors that you can specify. Refer to these properties to understand how they must be defined when specifying multiple conical sensors.

Object Functions

aer	Calculate azimuth angle, elevation angle, and range of another satellite or ground station in NED frame
access	Add access analysis objects to satellite scenario
fieldOfView	Visualize field of view of conical sensor
coordinateAxes	Visualize coordinate axes triad of satellite scenario assets

Examples

Calculate Maximum Revisit Time of Satellite

Create a satellite scenario with a start time of 15-June-2021 8:55:00 AM UTC and a stop time of five days later. Set the simulation sample time to `60` seconds.

```
startTime = datetime(2021,6,21,8,55,0);
stopTime = startTime + days(5);
sampleTime = 60;                                        % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc =
  satelliteScenario with properties:

         StartTime: 21-Jun-2021 08:55:00
          StopTime: 26-Jun-2021 08:55:00
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add a satellite to the scenario using Keplerian orbital elements.

```
semiMajorAxis = 7878137;                                                                    % me
eccentricity = 0;
inclination = 50;                                                                           % deg
rightAscensionOfAscendingNode = 0;                                                          % deg
argumentOfPeriapsis = 0;                                                                    % deg
trueAnomaly = 50;                                                                           % deg
sat = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly)
```

```
sat =
  Satellite with properties:

              Name:  Satellite 1
                ID:  1
     ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
            Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
       Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
          Receivers:  [1x0 satcom.satellitescenario.Receiver]
           Accesses:  [1x0 matlabshared.satellitescenario.Access]
         GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
              Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
    OrbitPropagator:  sgp4
        MarkerColor:  [0.059 1 1]
         MarkerSize:  6
          ShowLabel:  true
     LabelFontColor:  [1 1 1]
      LabelFontSize:  15
```

Add a ground station, which represents the location to be photographed, to the scenario.

```
gs = groundStation(sc,Name="Location to Photograph", ...
    Latitude=42.3001,Longitude=-71.3504)                % degrees
```

```
gs =
  GroundStation with properties:

                Name:  Location to Photograph
                  ID:  2
            Latitude:  42.3 degrees
           Longitude:  -71.35 degrees
            Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
```

```
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         MarkerColor:  [1 0.4118 0.1608]
          MarkerSize:  6
           ShowLabel:  true
      LabelFontColor:  [1 1 1]
       LabelFontSize:  15
```

Add a gimbal to the satellite. You can steer this gimbal independently of the satellite.

```
g = gimbal(sat)
```

```
g =
  Gimbal with properties:

                Name:  Gimbal 3
                  ID:  3
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
```

Track the location to be photographed using the gimbal.

```
pointAt(g,gs);
```

Add a conical sensor to the gimbal. This sensor represents the camera. Set the field of view to 60 degrees.

```
camSensor = conicalSensor(g,MaxViewAngle=60)
```

```
camSensor =
  ConicalSensor with properties:

                Name:  Conical sensor 4
                  ID:  4
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
        MaxViewAngle:  60 degrees
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         FieldOfView:  [0x0 matlabshared.satellitescenario.FieldOfView]
```

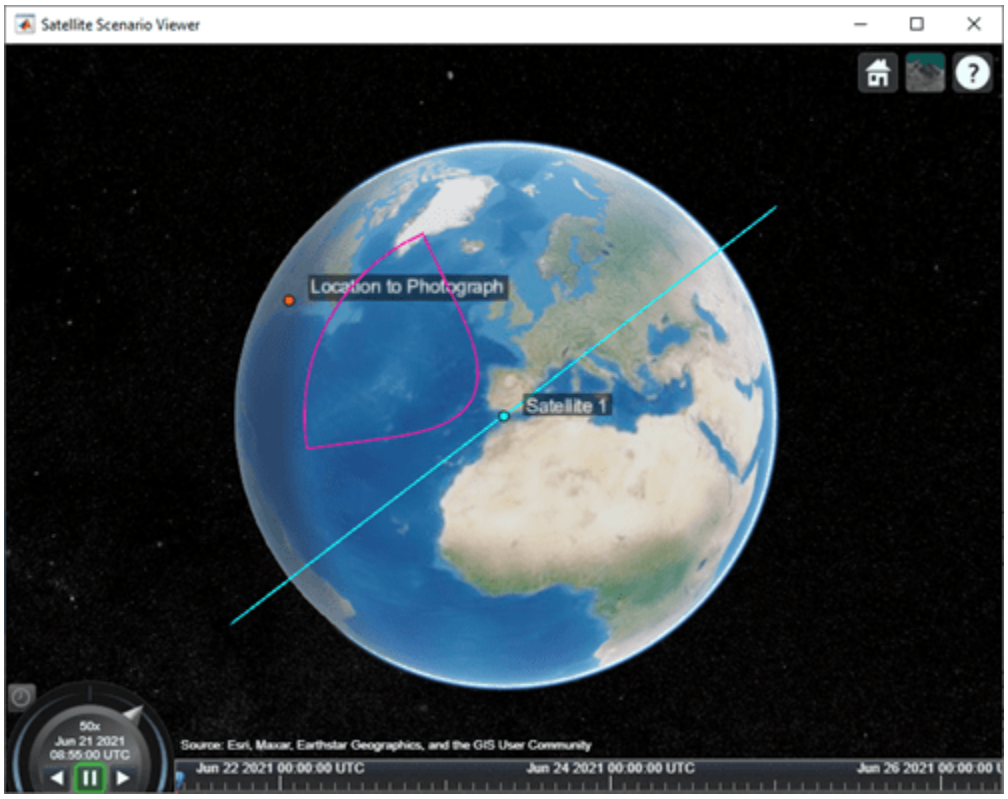
Add access analysis to the conical sensor between the camera and the location to be photographed.

```
ac = access(camSensor,gs)
```

```
ac =
  Access with properties:

    Sequence:  [4 2]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

Visualize the field of view of the camera by using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
fieldOfView(camSensor);
```



Determine the intervals during which the camera can see the geographical site.

```
t = accessIntervals(ac)
```

```
t=35×8 table
          Source                  Target              IntervalNumber        StartTime
    ____________________    ____________________    ______________    ____________________

    "Conical sensor 4"      "Location to Photograph"         1          21-Jun-2021 10:38:00
    "Conical sensor 4"      "Location to Photograph"         2          21-Jun-2021 12:36:00
    "Conical sensor 4"      "Location to Photograph"         3          21-Jun-2021 14:37:00
    "Conical sensor 4"      "Location to Photograph"         4          21-Jun-2021 16:41:00
    "Conical sensor 4"      "Location to Photograph"         5          21-Jun-2021 18:44:00
    "Conical sensor 4"      "Location to Photograph"         6          21-Jun-2021 20:46:00
    "Conical sensor 4"      "Location to Photograph"         7          21-Jun-2021 22:50:00
    "Conical sensor 4"      "Location to Photograph"         8          22-Jun-2021 09:51:00
    "Conical sensor 4"      "Location to Photograph"         9          22-Jun-2021 11:46:00
    "Conical sensor 4"      "Location to Photograph"        10          22-Jun-2021 13:46:00
    "Conical sensor 4"      "Location to Photograph"        11          22-Jun-2021 15:50:00
    "Conical sensor 4"      "Location to Photograph"        12          22-Jun-2021 17:53:00
    "Conical sensor 4"      "Location to Photograph"        13          22-Jun-2021 19:55:00
    "Conical sensor 4"      "Location to Photograph"        14          22-Jun-2021 21:58:00
    "Conical sensor 4"      "Location to Photograph"        15          23-Jun-2021 10:56:00
    "Conical sensor 4"      "Location to Photograph"        16          23-Jun-2021 12:56:00
```

Calculate the maximum revisit time in hours.

```
startTimes = t.StartTime;
endTimes = t.EndTime;
revisitTimes = hours(startTimes(2:end) - endTimes(1:end-1));
maxRevisitTime = max(revisitTimes)                                    % hours
```

```
maxRevisitTime = 12.6667
```

Visualize the revisit times that the camera photographs of the location.

```
play(sc);
```

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `access` | `groundStation`

Topics
“Satellite Scenario Key Concepts” on page 2-65

convacc

Convert from acceleration units to specified 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. The function then applies the conversion factor to the input to produce the output in the specified units.

Examples

Convert Accelerations

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 = 1×3

    0.9144    3.0480    6.0960
```

Input Arguments

`valuesToConvert` — Values to convert
floating-point array of size *m*-by-*n*

Values to convert, specified as a floating-point array of size *m*-by-*n* values. All values must have the same unit conversions from `inputAccelUnits` to `outputAccelUnits`.

Data Types: `double`

`inputAccelUnits` — Input acceleration units
`'ft/s^2'` | `'m/s^2'` | `'km/s^2'` | `'in/s^2'` | `'km/h-s'` | `'mph/s'` | `'G''s'`

Input acceleration units, specified as one of these values.

`'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-force (G's) acceleration

Data Types: `char` | `string`

`outputAccelUnits` — Output acceleration units
`'ft/s^2'` | `'m/s^2'` | `'km/s^2'` | `'in/s^2'` | `'km/h-s'` | `'mph/s'` | `'G''s'`

Output acceleration units, specified as one of these values.

`'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

Data Types: `char` | `string`

Output Arguments

`convertedValues` — Converted values
floating-point array of size *m*-by-*n*

Converted values, returned as a floating-point array of size *m*-by-*n*.

Version History

Introduced in R2006b

See Also

`convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

convang

Convert from angle units to specified 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. The function then applies the conversion factor to the input to produce the output in the specified units.

Examples

Convert Angles

Convert three angles from degrees to radians.

```
a = convang([3 10 20],'deg','rad')
```

```
a = 1×3

    0.0524    0.1745    0.3491
```

Input Arguments

`valuesToConvert` — Values to convert
floating-point array of size *m*-by-*n*

Values to convert, specified as a floating-point array of size *m*-by-*n*. All values must have the same unit conversions from `inputAngleUnits` to `outputAngleUnits`.

Data Types: `double`

`inputAngleUnits` — Input angle units
`'deg'` | `'rad'` | `'rev'`

Input angle units, specified as one of these values.

`'deg'`	Degrees
`'rad'`	Radians
`'rev'`	Revolutions

Data Types: `string`

`outputAngleUnits` — Output angle units
`'deg'` | `'rad'` | `'rev'`

Output angle units, specified as one of these values.

`'deg'`	Degrees
`'rad'`	Radians
`'rev'`	Revolutions

Data Types: `string`

Output Arguments

`convertedValues` — Converted values
floating-point array of size *m*-by-*n*

Converted values, returned as a floating-point array of size *m*-by-*n*.

Version History

Introduced in R2006b

See Also

`convacc` | `convangacc` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

convangacc

Convert from angular acceleration units to specified angular acceleration units

Syntax

```
convertedValues = convangacc(valuesToConvert,inputAngularAccelUnits,
outputAngularAccelUnits)
```

Description

`convertedValues = convangacc(valuesToConvert,inputAngularAccelUnits, outputAngularAccelUnits)` computes the conversion factor from specified input angular acceleration units to specified output angular acceleration units. The function then applies the conversion factor to the input to produce the output in the specified units.

Examples

Convert Angular Acceleration

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 = 1×3

    0.0052    0.0017    0.0087
```

Input Arguments

`valuesToConvert` — Values to convert
floating-point array of size *m*-by-*n*

Values to convert, specified as a floating-point array of size *m*-by-*n*. All values must have the same unit conversions from `inputAngularAccelUnits` to `outputAngularAccelUnits`.

Data Types: `double`

`inputAngularAccelUnits` — Input angular acceleration units
`'deg/s^2'` | `'rad/s^2'` | `'rpm/s'`

Input angular acceleration units, specified as one of these values.

`'deg/s^2'`	Degrees per second squared
`'rad/s^2'`	Radians per second squared
`'rpm/s'`	Revolutions per minute per second

Data Types: `string`

`outputAngularAccelUnits` — Output angular acceleration units
`'deg/s^2'` | `'rad/s^2'` | `'rpm/s'`

Output angular acceleration units, specified as one of these values.

`'deg/s^2'`	Degrees per second squared
`'rad/s^2'`	Radians per second squared
`'rpm/s'`	Revolutions per minute per second

Data Types: `string`

Output Arguments

`convertedValues` — Converted values
floating-point array of size *m*-by-*n*

Converted values, returned as a floating-point array of size *m*-by-*n*.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

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. The function then applies the conversion factor to the input to produce the output in the specified units.

Examples

Convert Angular Velocity

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 = 1×3

    0.0052    0.0017    0.0087
```

Input Arguments

`valuesToConvert` — Values to convert
floating-point array of size *m*-by-*n*

Values to convert, specified as a floating-point array of size *m*-by-*n*. All values must have the same unit conversions from `inputAngularVelocityUnits` to `outputAngularVelocityUnits`.

Data Types: `double`

`inputAngularVelocityUnits` — Input angular velocity units
`'deg/s'` | `'rad/s'` | `'rpm'`

Input angular velocity units, specified as one of these values.

`'deg/s'`	Degrees per second
`'rad/s'`	Radians per second
`'rpm'`	Revolutions per minute

Data Types: `string`

`outputAngularVelocityUnits` — Output angular velocity units
`'deg/s'` | `'rad/s'` | `'rpm'`

Output angular velocity units, specified as one of these values.

`'deg/s'`	Degrees per second
`'rad/s'`	Radians per second
`'rpm'`	Revolutions per minute

Data Types: `string`

Output Arguments

`convertedValues` — Converted values
floating-point array of size *m*-by-*n*

Converted values, returned as a floating-point array of size *m*-by-*n*.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangacc` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

convdensity

Convert from density units to specified 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. The function then applies the conversion factor to the input to produce the output in the specified units.

Examples

Convert Density

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 = 1×3

    4.8055    1.6018    8.0092
```

Input Arguments

`valuesToConvert` — Values to convert
floating-point array of size *m*-by-*n*

Values to convert, specified as a floating-point array of size *m*-by-*n*. All values must have the same unit conversions from `inputDensityUnits` to `outputDensityUnits`.

Data Types: `double`

`inputDensityUnits` — Input mass units
`'lbm/ft^3'` | `'kg/m^3'` | `'slug/ft^3'` | `'lbm/in^3'`

Input mass units, specified as one of these.

`'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

Data Types: `string`

`outputDensityUnits` — Output mass units
`'lbm/ft^3'` | `'kg/m^3'` | `'slug/ft^3'` | `'lbm/in^3'`

Output mass units, specified as one of these.

`'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

Data Types: `string`

Output Arguments

`convertedValues` — Converted values
floating-point array of size *m*-by-*n*

Converted values, returned as a floating-point array of size *m*-by-*n*.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

convforce

Convert from force units to specified 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. The function then applies the conversion factor to the input to produce the output in the specified units.

Examples

Convert Forces

Convert three forces from pound force to newtons.

```
a = convforce([120 1 5],'lbf','N')
```

```
a = 1×3

  533.7866    4.4482   22.2411
```

Input Arguments

`valuesToConvert` — Values to convert
floating-point array of size *m*-by-*n*

Values to convert, specified as a floating-point array of size *m*-by-*n*. All values must have the same unit conversions from `inputForceUnits` to `outputForceUnits`.

Data Types: `double`

`inputForceUnits` — Input force units
`'lbf'` | `'N'`

Input force units, specified as one of these.

`'lbf'`	Pound force
`'N'`	Newton

Data Types: `string`

`outputForceUnits` — Output force units
`'lbf'` | `'N'`

Output force units, specified as one of these.

`'lbf'`	Pound force
`'N'`	Newton

Data Types: `string`

Output Arguments

`convertedValues` — Converted values
floating-point array of size *m*-by-*n*

Converted values, returned as a floating-point array of size *m*-by-*n*.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

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.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convmass` | `convpres` | `convtemp` | `convvel`

convmass

Convert from mass units to specified 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. The function then applies the conversion factor to the input to produce the output in the specified units.

Examples

Convert masses

Convert three masses from pound mass to kilograms.

```
a = convmass([3 1 5],'lbm','kg')
```

```
a = 1×3

    1.3608    0.4536    2.2680
```

Input Arguments

`valuesToConvert` — Values to convert

floating-point array of size *m*-by-*n*

Values to convert, specified as a floating-point array of size *m*-by-*n*. All values must have the same unit conversions from `inputMassUnits` to `outputMassUnits`.

Data Types: `double`

`inputMassUnits` — Input mass units

`'lbm'` | `'kg'` | `'slug'`

Input mass units, specified as one of these.

`'lbm'`	Pound mass
`'kg'`	Kilograms
`'slug'`	Slugs

Data Types: `string`

`outputMassUnits` — Output mass units
`'lbm'` | `'kg'` | `'slug'`

Output mass units, specified as one of these.

`'lbm'`	Pound mass
`'kg'`	Kilograms
`'slug'`	Slugs

Data Types: `string`

Output Arguments

`convertedValues` — Converted values
floating-point array of size *m*-by-*n*

Converted values, returned as a floating-point array of size *m*-by-*n*.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convpres` | `convtemp` | `convvel`

convpres

Convert from pressure units to specified 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. The function then applies the conversion factor to the input to produce the output in the specified units.

Examples

Convert Pressures

Convert two pressures from pound force per square inch to atmospheres.

```
a = convpres([14.696  35],'psi','atm')
```

```
a = 1×2

    1.0000    2.3816
```

Input Arguments

`valuesToConvert` — Values to convert

floating-point array of size *m*-by-*n*

Values to convert, specified as a floating-point array of size *m*-by-*n*. All values must have the same unit conversions from `inputPressureUnits` to `outputPressureUnits`.

`inputPressureUnits` — Input pressure units

`'psi'` | `'Pa'` | `'psf'` | `'atm'`

Input pressure units, specified as one of these.

`'psi'`	Pound force per square inch
`'Pa'`	Pascal
`'psf'`	Pound force per square foot
`'atm'`	Atmosphere

`outputPressureUnits` — Output pressure units
`'psi'` | `'Pa'` | `'psf'` | `'atm'`

Output pressure units, specified as one of these.

`'psi'`	Pound force per square inch
`'Pa'`	Pascal
`'psf'`	Pound force per square foot
`'atm'`	Atmosphere

Output Arguments

`convertedValues` — Converted values
floating-point array of size *m*-by-*n*

Converted values, returned as a floating-point array of size *m*-by-*n*.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convtemp` | `convvel`

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.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convvel`

Topics
“Floating-Point Numbers”

convvel

Convert from current velocity units to desired velocity units

Syntax

```
convertedValues = convvel(valuesToConvert,inputVelocityUnits,
outputVelocityUnits)
```

Description

`convertedValues = convvel(valuesToConvert,inputVelocityUnits, outputVelocityUnits)` converts all velocity values from input velocity units to output velocity units. All values have the same unit conversions from `inputVelocityUnits` to `outputVelocityUnits`.

Examples

Convert Three Velocities

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
```

Input Arguments

`valuesToConvert` — Velocity values to convert

floating-point array of size *m*-by-*n*

Velocity values to convert, specified as floating-point array of size *m*-by-*n*.

Data Types: `double`

`inputVelocityUnits` — Input velocity units

`'ft/s'` | `'m/s'` | `'km/s'` | `'in/s'` | `'km/h'` | `'mph'` | `'kts'` | `'ft/min'`

Input velocity units, specified as one of these values.

`'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

Data Types: `string`

`outputVelocityUnits` — Output velocity units
`'ft/s'` | `'m/s'` | `'km/s'` | `'in/s'` | `'km/h'` | `'mph'` | `'kts'` | `'ft/min'`

Output velocities, specified as one of these values.

`'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

Data Types: `string`

Output Arguments

`convertedValues` — Output velocities
scalar | array

Output velocities, returned as an array or scalar in specified units.

Version History

Introduced in R2006b

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp`

CoordinateAxes

Coordinate axes triad graphic object

Description

`CoordinateAxes` object defines a coordinate axes triad graphic that corresponds to the body frame of a satellite scenario asset.

Creation

You can create a `CoordinateAxes` object using the `coordinateAxes` object function of the `Satellite`, `GroundStation`, `Gimbal`, or `ConicalSensor`.

Properties

`Scale` — Scale size of coordinate axes triad graphic
1 (default) | positive scalar

Scale size of coordinate axes triad, specified as a positive scalar.

Data Types: `double`

`VisibilityMode` — Visibility of body coordinate axes triad
`'auto'` (default) | `'inherit'` | `'manual'`

Visibility of coordinate axes triad graphic relative to the parent in the Satellite Scenario Viewer, specified as:

- `'auto'` — Hide coordinate axes triad.
- `'inherit'` — Inherit visibility from the asset.
- `'manual'` — Manually specify visibility. In this mode, the visibility of the coordinate axes triad is independent of the visibility of the asset.

Data Types: `string`

Object Functions

show	Show object in satellite scenario viewer
hide	Hide satellite scenario entity from viewer

Examples

Visualize Coordinate Axes Triad of Satellite

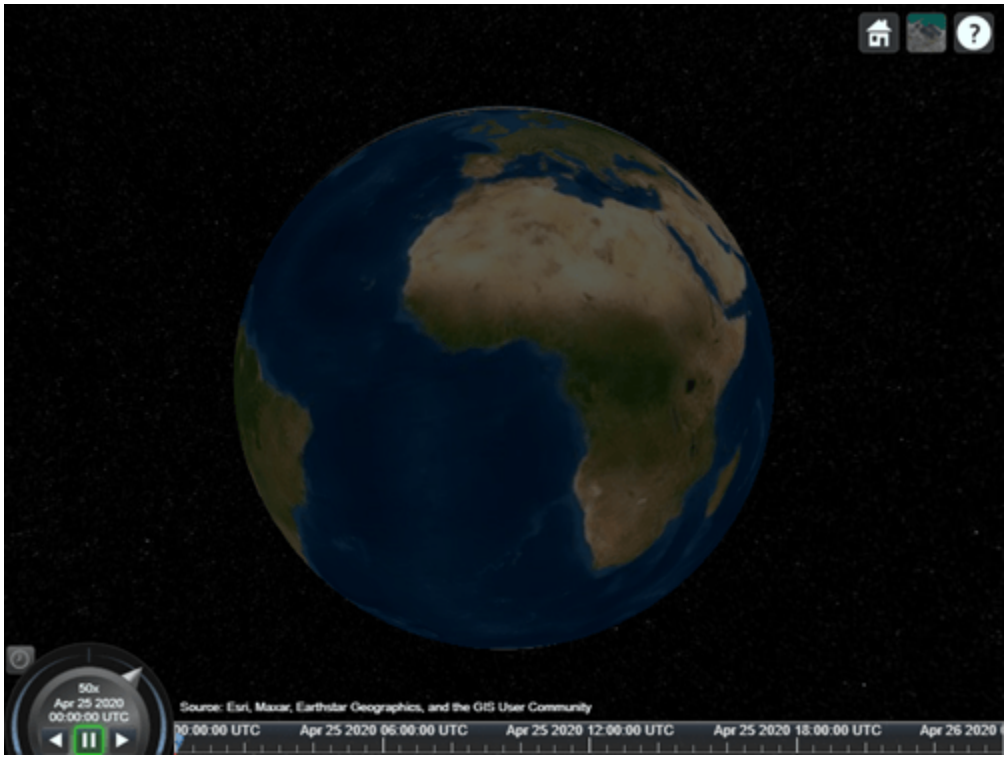
Create a `satelliteScenario` object.

```
startTime = datetime(2020,4,25); % April 25, 2020, 12:00 AM UTC
stopTime = datetime(2020,4,26);  % April 26, 2020, 12:00 AM UTC
```

```
sampleTime = 60;                % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

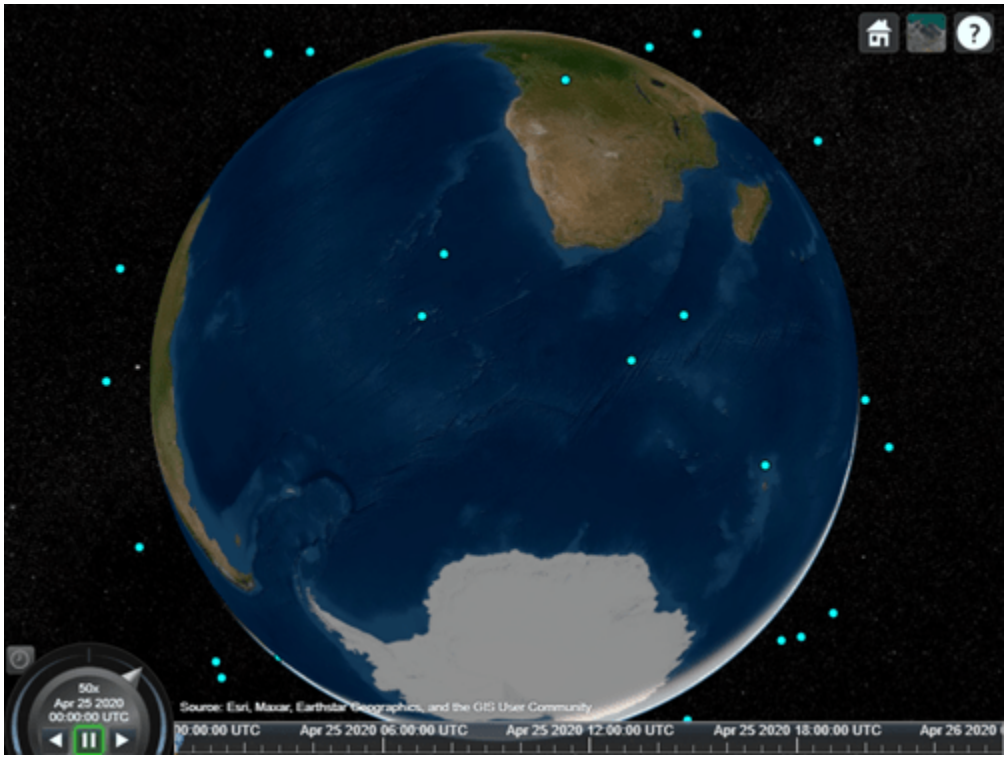
Launch a satellite scenario viewer with `ShowDetails` set to false.

```
v = satelliteScenarioViewer(sc,ShowDetails=false);
```



Add a low-earth orbit satellite constellation to the scenario from a TLE file.

```
tleFile = "leoSatelliteConstellation.tle";
sat = satellite(sc,tleFile);
```



Add a ground station to the scenario.

```
gs = groundStation(sc);
gs.ShowLabel = true;
```

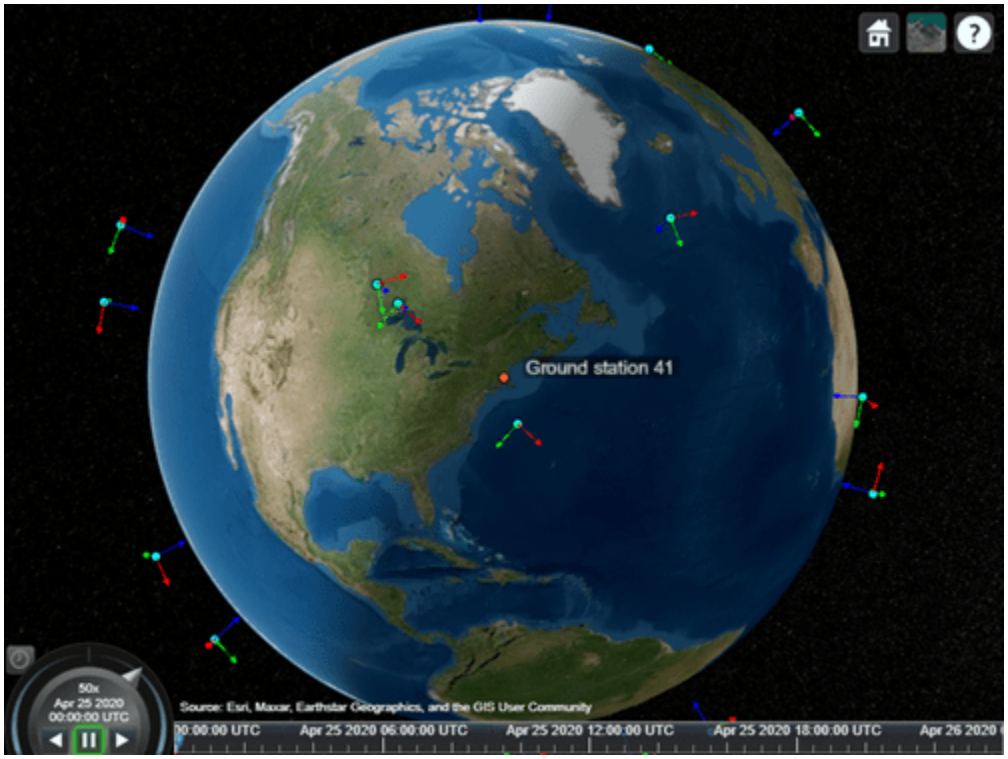
Visualize the coordinate axes triads of the satellites.

```
ax = coordinateAxes(sat)
```

```
ax=1×40 object
  1×40 CoordinateAxes array with properties:

    Scale
    VisibilityMode
```

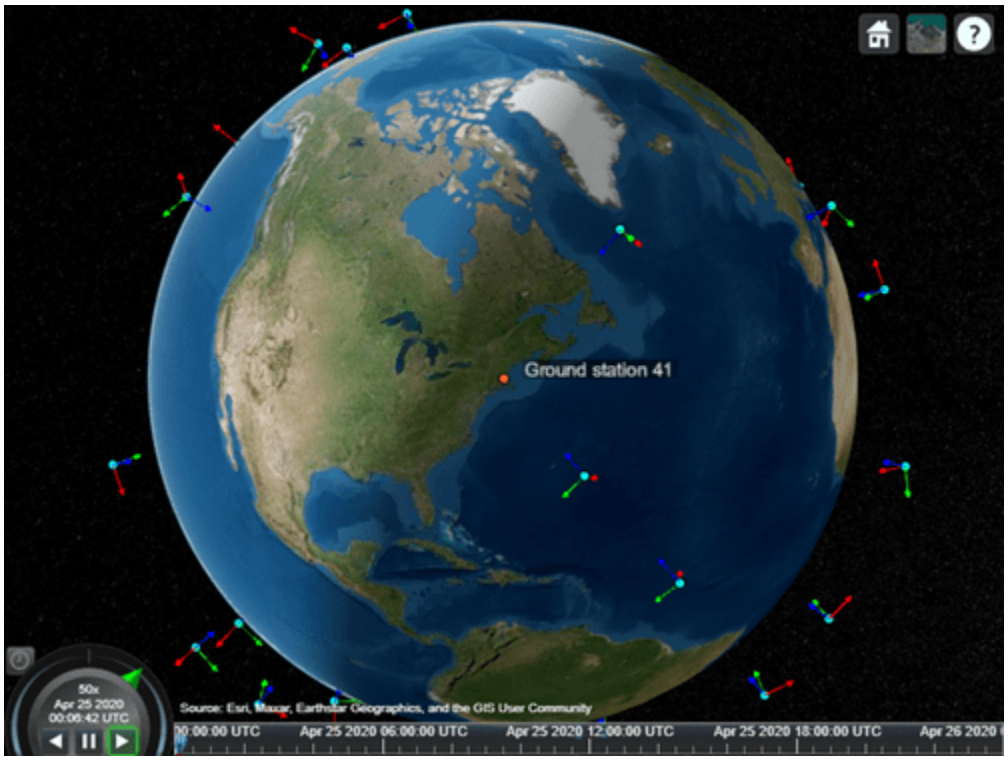
Point the satellites towards the ground station. The coordinate axes triad object updates to represent the new direction of the satellites.

```
pointAt(sat,gs);
```



Play the scenario.

```
play(sc);
```



Version History

Introduced in R2023a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`coordinateAxes` | `show` | `play` | `hide` | `satellite` | `access` | `groundStation` | `conicalSensor`

Topics

“Satellite Scenario Key Concepts” on page 2-65

coordinateAxes

Package: `matlabshared.satellitescenario`

Visualize coordinate axes triad of satellite scenario assets

Syntax

```
coordinateAxes(asset)
coordinateAxes(asset,Name=Value)
ax = coordinateAxes( ___ )
```

Description

`coordinateAxes(asset)` adds a coordinate axes triad graphic visualization (`CoordinateAxes` object) for an asset. In the visualization, the *x*-axis (roll) is red in color, the *y*-axis (pitch) is green, and *z*-axis (yaw) is blue.

`coordinateAxes(asset,Name=Value)` specifies additional arguments using one or more name-value.

`ax = coordinateAxes( ___ )` returns the `CoordinateAxes` object.

Examples

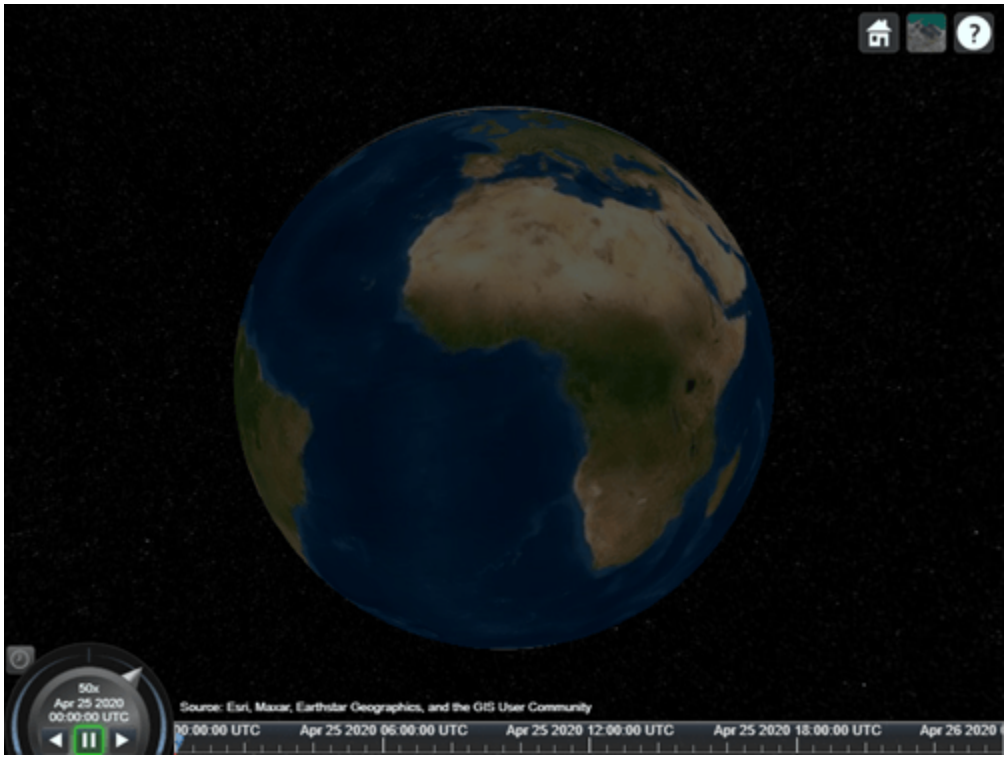
Visualize Coordinate Axes Triad of Satellite

Create a `satelliteScenario` object.

```
startTime = datetime(2020,4,25); % April 25, 2020, 12:00 AM UTC
stopTime = datetime(2020,4,26);  % April 26, 2020, 12:00 AM UTC
sampleTime = 60;                 % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

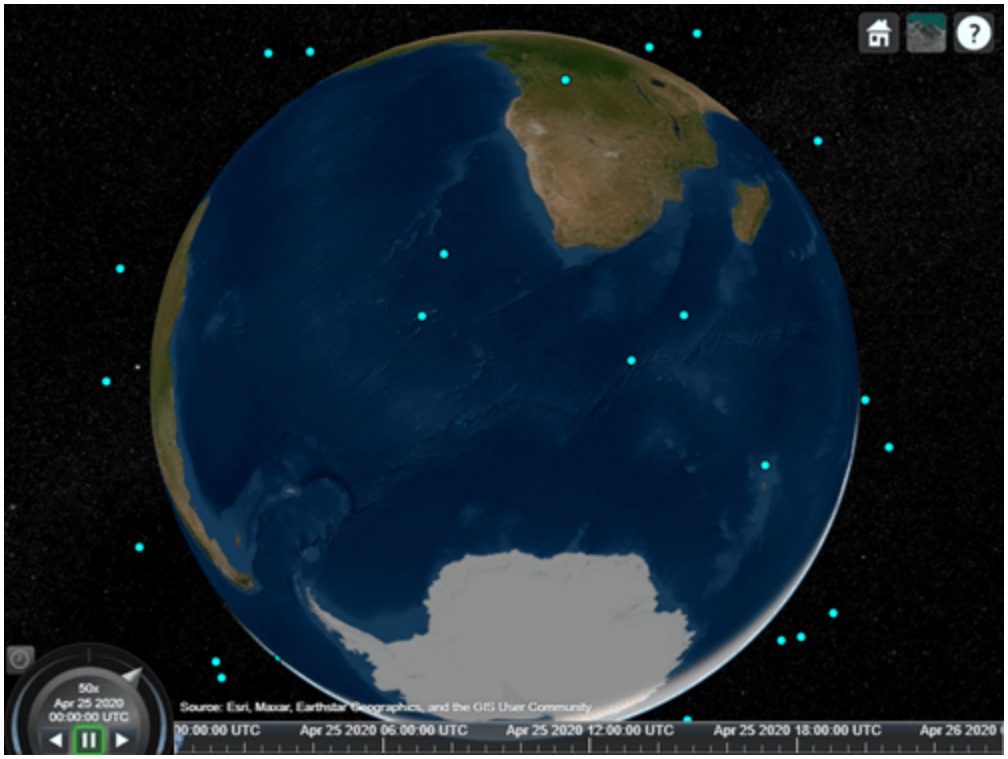
Launch a satellite scenario viewer with `ShowDetails` set to false.

```
v = satelliteScenarioViewer(sc,ShowDetails=false);
```



Add a low-earth orbit satellite constellation to the scenario from a TLE file.

```
tleFile = "leoSatelliteConstellation.tle";
sat = satellite(sc,tleFile);
```



Add a ground station to the scenario.

```
gs = groundStation(sc);
gs.ShowLabel = true;
```

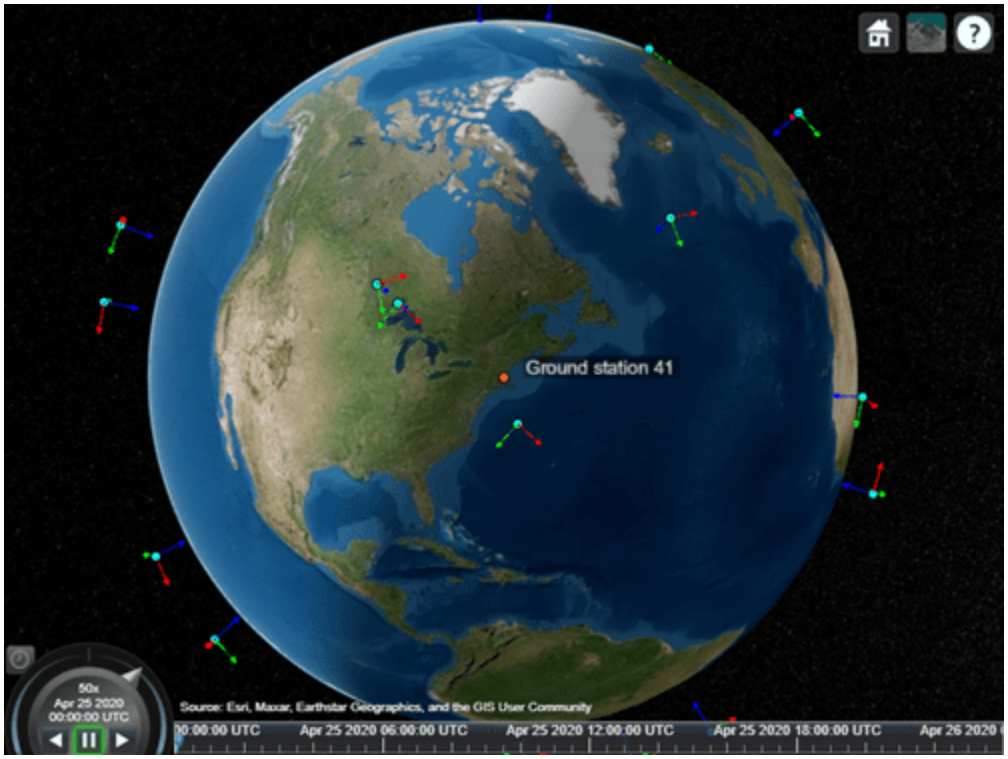
Visualize the coordinate axes triads of the satellites.

```
ax = coordinateAxes(sat)
```

```
ax=1×40 object
  1×40 CoordinateAxes array with properties:

    Scale
    VisibilityMode
```

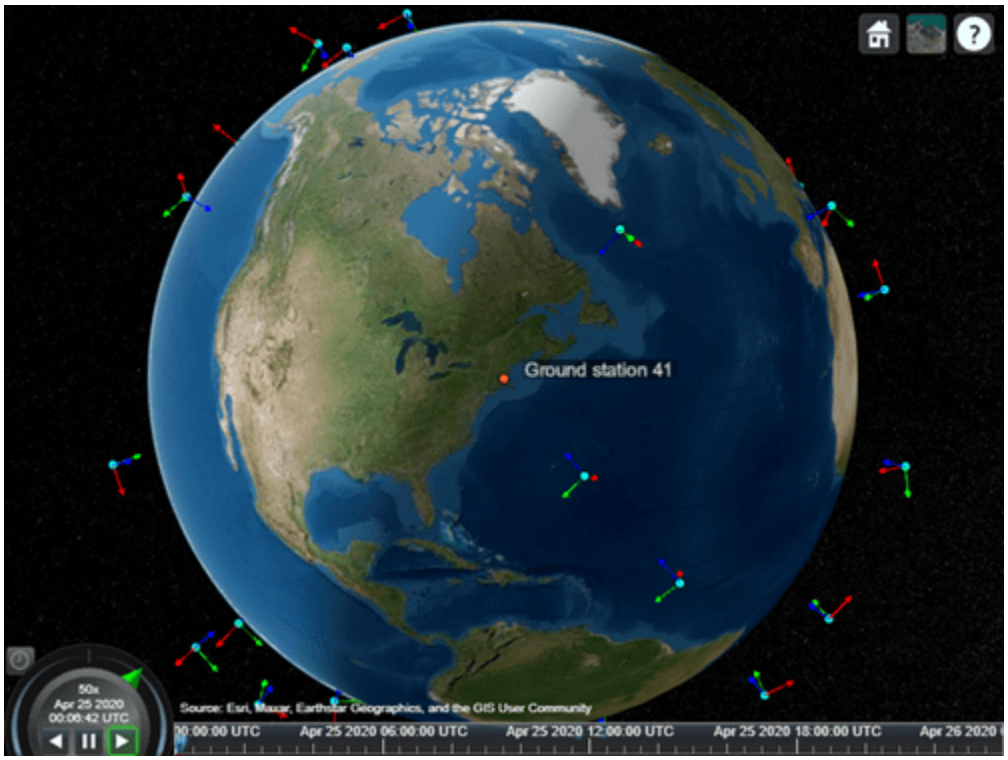
Point the satellites towards the ground station. The coordinate axes triad object updates to represent the new direction of the satellites.

```
pointAt(sat,gs);
```



Play the scenario.

```
play(sc);
```



Input Arguments

`asset` — Satellites, ground stations, gimbals, conical sensors, transmitters, or receivers

vector

Satellites, ground stations, gimbals, conical sensors, transmitters, or receivers, specified as a vector.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Scale =2` scales the size of the coordinate axes triad visualization by 2.

`Viewer` — Satellite scenario viewer

vector

Satellite scenario viewer, specified as a vector. You view the coordinate axes triad in the viewer specified in this argument.

`Scale` — Scale of coordinate axes triad graphic

1 (default) | positive scalar

Scale of the coordinate axes triad graphic, specified as a positive scalar.

Data Types: `double`

Output Arguments

`ax` — Coordinate axes triad

scalar | vector

Coordinate axes triad, returned as an object.

Version History

Introduced in R2023a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`play` | `show` | `groundStation` | `access` | `hide`

Topics

“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

correctairspeed

Convert airspeeds between 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 then 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.

Examples

Convert Three Airspeeds from True Airspeed at 1000 Meters

Convert three airspeeds from true airspeed to equivalent airspeed at 1000 meters using method `'TableLookup'`.

```
ain = [25.7222; 10.2889; 3.0867];
as = correctairspeed(ain,336.4,89874.6,'TAS','CAS','TableLookup')
```

```
as = 3×1

   24.5077
    9.8024
    2.9407
```

Convert Three Airspeeds from True Airspeed at 1000 Meters and 0 Meters

Convert airspeeds from true airspeed to equivalent airspeed at 1000 meters and 0 meters.

```
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 = 3×1

   25.7199
   10.2889
    3.0867
```

Convert Three Airspeeds from True Airspeed to Equivalent Airspeed at 15000 Meters

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 = 946.2572
```

Input Arguments

`inputAirspeed` — Input airspeed

floating-point array of size *m*-by-1

Input airspeed, specified as a floating-point array of size *m*-by-1, in meters per second. All values in the array must have the same airspeed conversion factor.

Data Types: `double`

`speedOfSound` — Speeds of sound

floating-point array of size *m*-by-1

Speeds of sound, specified as a floating-point array of size *m*-by-1 in meters per second.

Data Types: `double`

`pressure0` — Air pressures

floating-point array of size *m*-by-1

Air pressures, specified as a floating-point array of size *m*-by-1, in pascal.

Data Types: `double`

`inputAirspeedType` — Input airspeed type

`'TAS'` | `'CAS'` | `'EAS'`

Input airspeed type, specified as one of these.

Airspeed Type	Description
`'TAS'`	True airspeed
`'CAS'`	Calibrated airspeed
`'EAS'`	Equivalent airspeed

Data Types: `char` | `string`

`outputAirspeedType` — Output airspeed type
`'TAS'` | `'CAS'` | `'EAS'`

Output airspeed type, specified as one of these.

Airspeed Type	Description
`'TAS'`	True airspeed
`'CAS'`	Calibrated airspeed
`'EAS'`	Equivalent airspeed

Data Types: `char` | `string`

`method` — Airspeed conversion method
`'TableLookup'` (default) | `'Equation'`

Airspeed conversion method, specified as one of these.

Conversion Method	Description
`'TableLookup'`	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.

Dependencies

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).

Data Types: `char` | `string`

Output Arguments

`outputAirpseed` — Output airspeed
floating-point array of size *m*-by-1

Output airspeed, returned as a floating-point array of size *m*-by-1, in meters per second.

Limitations

This function assumes that air flow is compressible dry air with constant specific heat ratio (gamma).

Version History

Introduced in R2006b

References

[1] Lowry, J.T. *Performance of Light Aircraft*. Washington, DC: AIAA Education Series, 1999.

[2] Pratt & Whitney Aircraft. *Aeronautical Vestpocket Handbook*. United Technologies, August 1986.

[3] Gracey, William. *Measurement of Aircraft Speed and Altitude*. Washington, DC: NASA Reference Publication 1046, 1980.

See Also

`airspeed` | `atmoscoesa` | `atmosisa` | `atmoslapse` | `atmosnonstd`

Topics

“Compute Indicated Airspeed for Pitot-Static Airspeed Indicator” on page 5-221
“Differences Between Corrected Airspeeds” on page 5-224

createBody

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

Create body and its associated patches in animation

Syntax

```
idx = createBody(h,bodyDataSrc)
idx = createBody(h,bodyDataSrc,geometrysource)
```

Description

`idx = createBody(h,bodyDataSrc)` creates a new body using the body data source file `bodyDataSrc`, makes patches for the body, and adds the body to the animation object `h`. This command assumes a default geometry source type set to `Auto`. It returns `idx` as the body index.

`idx = createBody(h,bodyDataSrc,geometrysource)` creates a new body, makes patches for the body, and adds the body to the animation object `h`. `geometrysource` is the geometry source type for the body.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

bodyDataSrc — Body data source file

character vector | string scalar | structure containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables

Body data source file, specified as a character vector, string scalar, or structure containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables.

geometrysource — Geometry source type for body

`'Auto'` (default) | `'Variable'` | `'MatFile'` | `'Ac3d'` | `'Custom'`

Geometry source type for body, specified as one of these values.

Value	Description
`'Auto'`	Recognize `.mat` extensions as MAT-files, `.ac` extensions as AC3D files, and structures containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables.
`'Variable'`	Recognize structures containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables.

Value	Description
`'MatFile'`	Recognize `.mat` extensions as MAT-files.
`'Ac3d'`	Recognize `.ac` extensions as AC3D files.
`'Custom'`	Recognize custom extensions.

Output Arguments

`idx` — Body index

scalar

Body index, returned as a scalar.

Examples

Create Body for Animation Object

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

Version History

Introduced in R2007a

See Also

`Aero.Animation`

Aero.FixedWing.criteriaTable

Class: `Aero.FixedWing`
Package: `Aero`

Construct criteria table for fixed-wing static stability analysis

Syntax

```
criteriatable = Aero.FixedWing.criteriaTable()
```

Description

`criteriatable = Aero.FixedWing.criteriaTable()` constructs a criteria table for fixed-wing static stability analysis.

Output Arguments

`criteriatable` — Criteria table

6-by-*N* table

Criteria table, returned as a 6-by-*N* table where *N* is number of variables. By default, this table appears as follows:

	U	V	W	Alpha	Beta	P	Q	R
FX	"<"	''	''	''	''	''	''	''
FY	''	'<'	''	''	''	''	''	''
FZ	''	''	'<'	''	''	''	''	''
L	''	''	''	''	''	'<'	'<'	''
M	'>'	''	''	'<'	''	''	'<'	''
N	''	''	''	''	'>'	''	''	'<'

Examples

Calculate Static Stability of Cessna C182

Calculate the static stability of a Cessna C182.

```
[C182, CruiseState] = astC182();
stability = staticStability(C182, CruiseState)
```

```
stability =

  6×8 table

            U          V          W          Alpha      Beta       P          Q          R
          _______    _______    _______    _______    _______    _______    _______    _______

    FX    "Stable"   ""         ""         ""         ""         ""         ""         ""
    FY    ""         "Stable"   ""         ""         ""         ""         ""         ""
```

```
FZ    ""          ""          "Stable"    ""          ""          ""          ""          ""
L     ""          ""          ""          ""          "Stable"    "Stable"    ""          ""
M     "Stable"    ""          ""          "Stable"    ""          ""          "Stable"    ""
N     ""          ""          ""          ""          "Stable"    ""          ""          "Stable"
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `staticStability`

datcomimport

Bring DATCOM file into MATLAB environment

Syntax

```
aero = datcomimport(file)
aero = datcomimport(file,usenan)
aero = datcomimport(file,usenan,verbose)
aero = datcomimport(file,usenan,verbose,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(file,usenan,verbose,filetype)` imports a DATCOM of a particular USAF Digital DATCOM 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 4-337 • "Fields for 1999 Version (File Type 6)" on page 4-349 • "Fields for 2007, 2008, 2011, and 2014 Versions (File Type 6)" on page 4-353
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 4-357
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 4-362

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 4-366, 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'`
`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`

Field	Description	Default
`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`
`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

Field	Description	Default
`ndelta`	Number of control surface deflections: `delta`, `deltal`, or `deltar`.	`0`
`delta`	Array of control-surface streamwise deflection angles.	`[]`
`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`
`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`
`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`

Field	Matrix of...	Function of...
`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
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.

Field	Matrix of...	Stored with Indices of...
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)
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)

Field	Matrix of...	Stored with Indices of...
`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`)
`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`
`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`

Field	Matrix of...	Stored with Indices of...
`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.	`[]`
`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`

Field	Description	Default
`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`
`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`

Field	Description	Default
`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 = [];`

`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`
`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`

Field	Matrix of...	Function of...
`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`
`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'`
`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`

Field	Description	Default
`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`
`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`

Version History

Introduced in R2006b

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

“Import from USAF Digital DATCOM Files” on page 5-2

datcomToFixedWing

Class: `Aero.FixedWing`
Package: `Aero`

Construct fixed-wing aircraft from Digital DATCOM structure

Syntax

```
aircraft = datcomToFixedWing(aircraft,datcomstruct)
[aircraft,state] = datcomToFixedWing(aircraft,datcomstruct)
[aircraft,state] = datcomToFixedWing( ___ ,Name,Value)
```

Description

`aircraft = datcomToFixedWing(aircraft,datcomstruct)` returns a modified fixed-wing aircraft, `aircraft`, constructed from the fields of a Digital DATCOM structure, `datcomstruct`. To create a DATCOM structure, see `datcomimport`.

`[aircraft,state] = datcomToFixedWing(aircraft,datcomstruct)` returns an array of `Aero.FixedWing.State` objects in addition to the modified fixed-wing aircraft..

`[aircraft,state] = datcomToFixedWing( ___ ,Name,Value)` returns the modified fixed-wing aircraft using additional options specified by one or more `Name,Value` pair arguments. Specify the `Name,Value` argument as the last input argument followed by the input argument combination in the previous syntax.

Input Arguments

`aircraft` — `Aero.FixedWing` aircraft

scalar

Aero.FixedWing aircraft, specified as a scalar. To construct an empty aircraft, use `Aero.FixedWing(0)`.

`datcomstruct` — Digital DATCOM structure

scalar

Digital DATCOM structure, specified as a scalar. To create the digital DATCOM structure, use the `datcomimport` function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'StateMode','Exhaustive'`

`Build` — Build run dimension
scalar

Build run dimension to use, specified as a scalar greater than or equal to 1 or less than or equal to `datcomStruct.build`. The default is the value of `datcomStruct.build`.

Data Types: `double`

`Atmosphere` — Standard atmosphere model
`'ISA'` (default) | `'COESA'`

Standard atmosphere model when calculating the environment properties temperature, pressure, speed of sound, density, and aircraft speed, specified as `'ISA'` or `'COESA'`.

Data Types: `char` | `string`

`StateMode` — Source for constructing aircraft states
`'Scalar'` (default) | `'Exhaustive'`

Source for constructing aircraft states, specified as

- `'Scalar'` — `datcomToFixedWing` returns a scalar template state with the unit systems and control names derived from the DATCOM file. All other state fields retain their default values.
- `'Exhaustive'` — All supported fields from the DATCOM file are combined in an exhaustive state array. This option might take several minutes to execute.

Data Types: `char` | `string`

Output Arguments

`aircraft` — `Aero.FixedWing` object
scalar

`Aero.FixedWing` object, returned as a scalar. The method defines the aircraft coefficients of the object as `Simulink.LookupTable` objects derived from the coefficient fields in the Digital DATCOM structure `datcomStruct`.

`state` — Array of `Aero.FixedWing.State` objects
array

`Aero.FixedWing.State` objects, returned as an array. The value depends on the `StateMode` value.

Examples

Construct Fixed-Wing Aircraft from Digital DATCOM File

Construct a fixed-wing aircraft from an imported Digital DATCOM file.

```
datcomStruct = datcomimport('astdatcom.out');
aircraft = Aero.FixedWing();
aircraft.Properties.Name = "MyPlane";
aircraft = datcomToFixedWing(aircraft, datcomStruct{1})
```

```
aircraft =
```

```
  FixedWing with properties:

        ReferenceArea: 225.8000
        ReferenceSpan: 41.1500
      ReferenceLength: 5.7500
         Coefficients: [1×1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1×0 Aero.FixedWing.Surface]
              Thrusts: [1×0 Aero.FixedWing.Thrust]
          AspectRatio: 7.4992
           UnitSystem: "English (ft/s)"
          AngleSystem: "Degrees"
    TemperatureSystem: "Rankine"
           Properties: [1×1 Aero.Aircraft.Properties]
```

Construct Fixed-Wing Aircraft from Digital DATCOM File with Build Number

Construct fixed-wing aircraft from Digital DATCOM file, specifying the build number.

```
datcomStruct = datcomimport('astdatcom.out');
aircraft = Aero.FixedWing();
aircraft.Properties.Name = "MyPlane";
aircraft = datcomToFixedWing(aircraft,datcomStruct{1},'Build',1)
```

```
aircraft =

  FixedWing with properties:

        ReferenceArea: 225.8000
        ReferenceSpan: 41.1500
      ReferenceLength: 5.7500
         Coefficients: [1×1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1×0 Aero.FixedWing.Surface]
              Thrusts: [1×0 Aero.FixedWing.Thrust]
          AspectRatio: 7.4992
           UnitSystem: "English (ft/s)"
          AngleSystem: "Degrees"
    TemperatureSystem: "Rankine"
           Properties: [1×1 Aero.Aircraft.Properties]
```

Construct Fixed-Wing Aircraft from Digital DATCOM File with 'Exhaustive' StateMode

Construct a fixed-wing aircraft from an imported Digital DATCOM file and return the exhaustive state array.

```
datcomStruct = datcomimport('astdatcom.out');
aircraft = Aero.FixedWing();
aircraft.Properties.Name = "MyPlane";
[aircraft,state] = datcomToFixedWing(aircraft,datcomStruct{1},'StateMode','Exhaustive')
```

```
aircraft =

  FixedWing with properties:

        ReferenceArea: 225.8000
        ReferenceSpan: 41.1500
```

```
      ReferenceLength: 5.7500
         Coefficients: [1×1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1×0 Aero.FixedWing.Surface]
              Thrusts: [1×0 Aero.FixedWing.Thrust]
          AspectRatio: 7.4992
           UnitSystem: "English (ft/s)"
          AngleSystem: "Degrees"
    TemperatureSystem: "Rankine"
           Properties: [1×1 Aero.Aircraft.Properties]


state =

  5×2×2 State array with properties:

    Alpha
    Beta
    AlphaDot
    BetaDot
    Mass
    Inertia
    CenterOfGravity
    CenterOfPressure
    AltitudeMSL
    GroundHeight
    XN
    XE
    XD
    U
    V
    W
    Phi
    Theta
    Psi
    P
    Q
    R
    Weight
    AltitudeAGL
    Airspeed
    GroundSpeed
    MachNumber
    BodyVelocity
    GroundVelocity
    Ur
    Vr
    Wr
    FlightPathAngle
    CourseAngle
    InertialToBodyMatrix
    BodyToInertialMatrix
    BodyToWindMatrix
    WindToBodyMatrix
    DynamicPressure
    Environment
    UnitSystem
    AngleSystem
```

```
TemperatureSystem
ControlStates
OutOfRangeAction
DiagnosticAction
Properties
```

Limitations

- This method supports only Digital DATCOM, which is the 1976 version of DATCOM.
- This fields `alpha`, `mach`, `alt`, `grndht`, and `delta` must be strictly monotonically increasing.
- This method requires a Simulink license.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `datcomimport` | `Simulink.LookupTable`

dcm2alphabeta

Convert direction cosine matrix to angle of attack and sideslip angle

Syntax

```
[alpha beta] = dcm2alphabeta(dcm)
[alpha beta] = dcm2alphabeta(dcm,action)
[alpha beta] = dcm2alphabeta( ___ ,tolerance)
```

Description

`[alpha beta] = dcm2alphabeta(dcm)` calculates the angle of attack (`alpha`) and sideslip angle (`beta`) for the direction cosine matrix, `dcm`. The function transforms the coordinates from a vector in body axes into a vector in wind axes.

`[alpha beta] = dcm2alphabeta(dcm,action)` performs `action` if the `dcm` is not orthogonal.

`[alpha beta] = dcm2alphabeta( ___ ,tolerance)` uses a `tolerance` level to evaluate if the direction cosine matrix, `dcm`, is orthogonal. Specify `tolerance` after all other input arguments.

Examples

Determine Angle of Attack and Sideslip Angle from Direction Cosine Matrix

Determine the angle of attack and sideslip angle from a 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 Angle of Attack and Sideslip Angle from Multiple Direction Cosine Matrices

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 Angle of Attack and Sideslip Angle from Multiple Direction Cosine Matrices within Tolerance

Determine the angle of attack and sideslip angle from multiple direction cosine matrices. Return a warning if the `dcm` exceeds a tolerance of `0.1`:

```
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
```

Input 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. `dcm` contains *m* orthogonal direction cosine matrices.

`action` — Action

`'None'` (default) | `'Error'` | `'Warning'`

Action for invalid direction cosine matrices, specified as:

- `'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.

Data Types: `char` | `string`

`tolerance` — Relative tolerance
`eps(2)` (default) | scalar

Tolerance level to evaluate if the direction cosine matrix, `dcm`, is orthogonal, 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

`alpha` — Angles of attack
array

Angles of attack, returned as an array of *m* angles of attack, in radians.

`beta` — Sideslip angles
array

Sideslip angles, returned as an *m* array of sideslip angles, in radians.

Version History

Introduced in R2006b

See Also

`angle2dcm` | `dcm2angle` | `dcmbody2wind`

dcm2angle

Create rotation angles from direction cosine matrix

Syntax

```
[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm)
[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm,rotationSequence)

[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm,rotationSequence,
lim)
[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm,rotationSequence,
lim,action)
[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm,rotationSequence,
lim,action,tolerance)
```

Description

Basic Syntax

`[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm)` calculates the rotation angles, `rotationAng1`, `rotationAng2`, `rotationAng3`, for a direction cosine matrix, `dcm`. The rotation used in this function is a passive transformation between two coordinate systems.

`[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm,rotationSequence)` calculates the rotation angles for a specified rotation sequence, `rotationSequence`.

Constraint, Action, and Tolerance Syntax

`[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm,rotationSequence, lim)` calculates the rotation angles for a specified angle constraint, `lim`. Specify `lim` after all other input arguments.

`[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm,rotationSequence, lim,action)` calculates the rotation angles and performs an action if the direction cosine matrix is not orthogonal. Specify `action` after all other input arguments.

`[rotationAng1 rotationAng2 rotationAng3] = dcm2angle(dcm,rotationSequence, lim,action,tolerance)` calculates the rotation angles and uses a `tolerance` level to evaluate if the direction cosine matrix is orthogonal. Specify `tolerance` after all other input arguments.

Examples

Determine Rotation Angles from Direction Cosine Matrix

Determine the rotation angles from the 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 Rotation Angles from Multiple Direction Cosine Matrices

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','Default','None',0.1)
```

```
pitch =
          0
     0.3000

roll =
          0
     0.1000

yaw =
          0
     0.5000
```

Determine Rotation Angles from Multiple Direction Cosine Matrices and Angle Constraint

Calculate the rotation angles from direction cosine matrix and specify the rotation order and angle constraint.

```
dcm = [1 0 0; 0 1 0; 0 0 1];
[yaw, pitch, roll] = dcm2angle( dcm, 'zyx', 'robust')
```

```
yaw =
     0

pitch =
     0

roll =
     0
```

Determine Rotation Angles from Multiple Direction Cosine Matrices, Angle Constraint, and Action

Calculate the rotation angles from direction cosine matrix, specifying the rotation order, angle constraint, and action.

```
dcm = [1 0 0; 0 1 0; 0 0 1];
[yaw, pitch, roll] = dcm2angle( dcm, 'zyx', 'robust', 'warning')
```

```
yaw =
     0

pitch =
     0

roll =
     0
```

Determine Rotation Angles from Multiple Direction Cosine Matrices, Angle Constraint, Action, and Tolerance

Calculate the rotation angles from direction cosine matrix, specifying the rotation order, angle constraint, action, and tolerance.

```
dcm = [1 0 0; 0 1 0; 0 0 1];
[yaw, pitch, roll] = dcm2angle( dcm, 'zyx', 'robust', 'warning', 0.01)
```

```
yaw =
     0

pitch =
     0

roll =
     0
```

Input 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. The direction cosine matrices must be orthogonal with determinant +1.

`rotationSequence` — Rotation sequence

`'ZYX'` (default) | `'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XZY'` | `'XYX'` | `'XZX'`

Rotation sequence, specified as:

- `'ZYX'`
- `'ZYZ'`
- `'ZXY'`
- `'ZXZ'`
- `'YXZ'`
- `'YXY'`
- `'YZX'`
- `'YZY'`
- `'XYZ'`

- `'XZY'`
- `'XYX'`
- `'XZX'`

where `rotationAng1` is the *z*-axis rotation, `rotationAng2` is the *y*-axis rotation, and `rotationAng3` is the *x*-axis rotation.

Data Types: `char` | `string`

`lim` — Angle constraint
`'Default'` (default) | `'ZeroR3'` | `'Robust'`

Angle constraint, specified as:

- `'Default'` — Returns the default case of R1, R2, and R3. In the event of a gimbal lock, use `'ZeroR3'` or `'Robust'`.
- `'ZeroR3'` — In the event of gimbal lock, sets R3 to `0` and solves for R1 and R2.
- `'Robust'` — Returns R1, R2, and R3 from either the `'Default'` or `'ZeroR3'` case that produces a rotation matrix that most closely matches the input matrix.

For more information on angle constraints, see “Limitations” on page 4-380.

`action` — Action for invalid direction cosine matrix
`'None'` (default) | `'Error'` | `'Warning'`

Action for invalid direction cosine matrix, specified as:

- `'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.

Valid direction cosine matrices are orthogonal and proper when:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance `tolerance` (`transpose(dcm)*dcm == 1±tolerance`)
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(dcm) == 1±tolerance`).

Data Types: `char` | `string`

`tolerance` — Relative tolerance
`eps(2)` (default) | scalar

Relative tolerance level to evaluate if the direction cosine matrix, `dcm`, is orthogonal, specified as a scalar.

Data Types: `char` | `string`

Output Arguments

`rotationAng1` — First rotation angles
m-by-1 array

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

`rotationAng2` — Second rotation angles
m-by-1 array

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

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

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

Limitations

- The `'Default'` limitations for the `'ZYX'`, `'ZXY'`, `'YXZ'`, `'YZX'`, `'XYZ'`, and `'XZY'` implementations generate an `rotationAng2` angle that lies between ±90 degrees, and `rotationAng1` and `rotationAng3` angles that lie between ±180 degrees.
- The `'Default'` limitations for the `'ZYZ'`, `'ZXZ'`, `'YXY'`, `'YZY'`, `'XYX'`, and `'XZX'` implementations generate a `rotationAng2` angle that lies between 0–180 degrees, and `rotationAng1` and `rotationAng3` angles that lie between ±180 degrees.
- The `'ZeroR3'` limitations for the `'ZYX'`, `'ZXY'`, `'YXZ'`, `'YZX'`, `'XYZ'`, and `'XZY'` implementations generate a `rotationAng2` angle that lies between ±90 degrees, and `rotationAng1` and `rotationAng3` angles that lie between ±180 degrees. However, when `rotationAng2` is ±90 degrees, `rotationAng3` is set to 0 degrees.
- The `'ZeroR3'` limitations for the `'ZYZ'`, `'ZXZ'`, `'YXY'`, `'YZY'`, `'XYX'`, and `'XZX'` implementations generate a `rotationAng2` angle that lies between 0–180 degrees, and `rotationAng1` and `rotationAng3` angles that lie between ±180 degrees. However, when `rotationAng2` is 0 or ±180 degrees, `rotationAng3` is set to 0 degrees.

Version History

Introduced in R2006b

See Also

`angle2dcm` | `dcmbody2stability` | `dcm2quat` | `quat2dcm` | `quat2angle`

dcm2latlon

Convert direction cosine matrix to geodetic latitude and longitude

Syntax

```
[lat lon] = dcm2latlon(dcm)
[lat lon] = dcm2latlon(dcm,action)
[lat lon] = dcm2latlon(dcm,action,tolerance)
```

Description

`[lat lon] = dcm2latlon(dcm)` calculates the geodetic latitude and longitude `lat` and `lon` for the direction cosine matrix `dcm`.

`[lat lon] = dcm2latlon(dcm,action)` performs an, `action`, if the direction cosine matrix is invalid, that is, not orthogonal.

`[lat lon] = dcm2latlon(dcm,action,tolerance)` uses a tolerance level, `tolerance`, to evaluate if the direction cosine matrix is within tolerance.

Examples

Determine Geodetic Latitude and Longitude from Direction Cosine Matrix

Determine the geodetic latitude and longitude from direction cosine matrix `dcm`.

```
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 Geodetic Latitude and Longitude with Multiple Direction Cosine Matrices

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 = 2×1
```

```
   44.9995
   37.5028


lon = 2×1

 -122.0005
  -84.9975
```

Determine Geodetic Latitude and Longitude from Direction Cosine Matrix Within Tolerance

Determine the geodetic latitude and longitude from the direction cosine matrix `dcm` within tolerance.

```
dcm = [ 0.3747    0.5997    0.7071; ...
        0.8480   -0.5299         0; ...
        0.3747    0.5997   -0.7071];
[lat, lon] = dcm2latlon(dcm,'Warning',0.1)
```

```
lat = 44.9995

lon = -122.0005
```

Input Arguments

`dcm` — Direction cosine matrix

3-by-3-by-*M* matrix

Direction cosine matrix, specified as a 3-by-3-by-*M* containing *M* orthogonal direction cosine matrices. `dcm` performs the coordinate transformation of a vector in Earth-centered Earth-fixed (ECEF) axes into a vector in North-East-down (NED) axes.

Data Types: `double`

`action` — Function behavior

`'None'` (default) | `'Error'` | `'Warning'`

Function behavior when direction cosine matrix is invalid, that is 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.

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(dcm)*dcm == 1±tolerance`).

- The determinant of the direction cosine matrix equals 1 within the specified tolerance (`det(dcm) == 1±tolerance`).

Data Types: `double`

Output Arguments

`lat` — Geodetic latitude

M array

Geodetic latitude, returned as an *M* array in degrees.

`lon` — Geodetic longitude

M array

Geodetic longitude, returned as a *M* array in degrees.

Version History

Introduced in R2006b

See Also

`angle2dcm` | `dcm2angle` | `dcmecef2ned`

dcm2quat

Convert direction cosine matrix to quaternion

Syntax

```
q = dcm2quat(dcm)
q = dcm2quat(dcm,action)
q = dcm2quat(dcm,action,tolerance)
```

Description

`q = dcm2quat(dcm)` calculates the quaternion `q` for a given direction cosine matrix, `dcm`.

`q = dcm2quat(dcm,action)` performs an action, `action`, if the direction cosine matrix is invalid, that is not orthogonal.

`q = dcm2quat(dcm,action,tolerance)` uses a tolerance level, `tolerance`, to evaluate if the direction cosine matrix `dcm` is within tolerance.

Examples

Determine Quaternion from Direction Cosine Matrix

Determine the quaternion from a direction cosine matrix.

```
dcm = [0 1 0; 1 0 0; 0 0 -1];
q = dcm2quat(dcm)
```

```
q = 1×4

         0    0.7071    0.7071         0
```

Determine Quaternions from Multiple Direction Cosine Matrices

This example shows how to 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 = 2×4

         0    0.7071    0.7071         0
    0.8224    0.2006    0.5320    0.0223
```

Determine Quaternion from Direction Cosine Matrix Within Tolerance

Determine the quaternion from direction cosine matrix `dcm` within tolerance.

```
dcm = [0 1 0; 1 0 0; 0 0 -1];
q = dcm2quat(dcm, 'Warning', 0.01)
```

```
q = 1×4

         0    0.7071    0.7071         0
```

Input Arguments

`dcm` — Direction cosine matrix

3-by-3-by-*M* matrix

Direction cosine matrix, specified as a 3-by-3-by-*M* matrix.

Data Types: `double`

`action` — Function behavior

`'None'` (default) | `'Error'` | `'Warning'`

Function behavior when direction cosine matrix is invalid, that is, 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.

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(dcm)*dcm == 1±tolerance`).
- The determinant of the direction cosine matrix equals 1 within the specified tolerance (`det(dcm) == 1±tolerance`).

Data Types: `double`

Output Arguments

`q` — Quaternion

m-by-4 matrix

Quaternion, returned in an *m*-by-4 matrix. `q` has a scalar number as the first column.

Version History

Introduced in R2006b

See Also

`angle2dcm` | `dcm2angle` | `angle2quat` | `quat2dcm` | `quat2angle`

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 $\vec{b}$ 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.

Version History

Introduced in R2017a

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` | `rod2quat` | `rod2angle` | `rod2dcm`

dcmbody2stability

Convert body frame to stability frame transformation matrix

Syntax

```
dcm = dcmbody2stability(anglesOfAttack)
```

Description

`dcm = dcmbody2stability(anglesOfAttack)` calculates the direction cosine matrix `dcm` for given set of angles of attack `anglesOfAttack`.

Examples

Determine Direction Cosine Matrix from One Angle of Attack

This example shows how to determine the direction cosine matrix from one angle of attack.

```
alpha = 0.4363;
dcm = dcmbody2stability(alpha)
```

```
dcm = 3×3

    0.9063         0    0.4226
         0    1.0000         0
   -0.4226         0    0.9063
```

Determine Direction Cosine Matrix from Multiple Angles of Attack

This example shows how to determine the direction cosine matrix from multiple angles of attack.

```
alpha = [0.4363 0.1745];
dcm = dcmbody2stability(alpha)
```

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

    0.9063         0    0.4226
         0    1.0000         0
   -0.4226         0    0.9063


dcm(:,:,2) =

    0.9848         0    0.1736
         0    1.0000         0
   -0.1736         0    0.9848
```

Input Arguments

`anglesOfAttack` — Angles of attack

vector

Angles of attack, specified as a vector, in radians.

Data Types: `double`

Output Arguments

`dcm` — Direction cosine matrices

3-by-3-by-*m* matrix

Direction cosine matrices, returned as a 3-by-3-by-*m* matrix, where *m* is the number of direction cosine matrices. The direction cosine matrix performs the coordinate transformation of a vector in inertial axes to a vector in body axes.

Version History

Introduced in R2022a

See Also

Topics

`angle2dcm`
`dcm2alphabeta`
`dcm2angle`
`dcmbody2wind`

dcmbody2wind

Convert angle of attack and sideslip angle to direction cosine matrix

Syntax

```
dcm = dcmbody2wind(alpha,beta)
```

Description

`dcm = dcmbody2wind(alpha,beta)` calculates the direction cosine matrix `dcm` for given angles of attack `alpha` and sideslip angles `beta`.

Examples

Determine Direction Cosine Matrix from Angle of Attack and Sideslip Angle

Determine the direction cosine matrix `dcm` from the angle of attack and sideslip angle.

```
alpha = 0.4363;
beta = 0.1745;
dcm = dcmbody2wind(alpha, beta)
```

```
dcm = 3×3

    0.8926    0.1736    0.4162
   -0.1574    0.9848   -0.0734
   -0.4226         0    0.9063
```

Determine Direction Cosine Matrix from Multiple Angles of Attack and Sideslip Angles

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 =
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
```

Input Arguments

`alpha` — Angles of attack

array

Angles of attack, specified as an array of *m* angles of attack, in radians. `dcm` defines the transformation of the body frame to the stability frame.

Data Types: `double`

`beta` — Sideslip angles

array

Sideslip angles, specified as an *m* array of sideslip angles, in radians.

Data Types: `double`

Output Arguments

`dcm` — Direction cosine matrices

3-by-3-by-*m* matrix

Direction cosine matrices, returned as a 3-by-3-by-*m* matrix, where *m* is the number of direction cosine matrices. `dcm` performs the coordinate transformation of a vector in body-axes into a vector in wind-axes.

Version History

Introduced in R2006b

See Also

`angle2dcm` | `dcm2alphabeta` | `dcmbody2stability` | `dcm2angle`

dcmecef2ned

Convert geodetic latitude and longitude to direction cosine matrix

Syntax

```
dcm = dcmecef2ned(lat,lon)
```

Description

`dcm = dcmecef2ned(lat,lon)` calculates the direction cosine matrix `dcm` for a given set of geodetic latitude `lat` and longitude `lon`.

Examples

Determine Direction Cosine Matrix from Geodetic Latitude and Longitude

Determine the direction cosine matrix from the geodetic latitude and longitude.

```
lat = 45;
lon = -122;
dcm = dcmecef2ned(lat, lon)
```

```
dcm = 3×3

    0.3747    0.5997    0.7071
    0.8480   -0.5299         0
    0.3747    0.5997   -0.7071
```

Determine Direction Cosine Matrix from Multiple Geodetic Latitudes and Longitudes

Determine the direction cosine matrix from multiple geodetic latitudes and longitudes.

```
lat = [45 37.5];
lon = [-122 -85];
dcm = dcmecef2ned(lat, lon)
```

```
dcm =
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
```

Input Arguments

`lat` — Geodetic latitude

M array

Geodetic latitude, specified as an *M* array in degrees. Latitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `double`

`lon` — Geodetic longitude

M array

Geodetic longitude, specified as a *M* array in degrees. Longitude values can be any value.

Data Types: `double`

Output Arguments

`dcm` — Direction cosine matrix

3-by-3-by-*M* matrix

Direction cosine matrix, returned as a 3-by-3-by-*M* matrix, where *M* is the number of orthogonal direction cosine matrices. `dcm` performs the coordinate transformation of a vector in Earth-centered Earth-fixed (ECEF) axes into a vector in North-East-down (NED) axes.

Version History

Introduced in R2006b

See Also

`angle2dcm` | `dcm2angle` | `dcm2latlon`

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 Coordinated Universal 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
```

Convert Using IAU-76/FK5 Reduction and `datetime` Arrays

Convert Earth-centered inertial (ECI) to Earth-centered Earth-fixed (ECEF) coordinates using the IAU-2000/2006 reduction, for two UTC dates represented as datetime arrays `utcDT`. All other parameters default to null arrays.

```
utcDT = datetime([2000 1 12 4 52 12.4;2000 1 12 4 52 13])
DCM = dcmeci2ecef('IAU-2000/2006', utcDT)
```

```
utcDT =
  2×1 datetime array

   12-Jan-2000 04:52:12
   12-Jan-2000 04:52:13

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
```

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` — Coordinated Universal Time

scalar | 1-by-6 array | *M*-by-6 matrix | scalar 1-by-1 `datetime` array | *M*-by-1 `datetime` array

Coordinated Universal 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-1 array

  Specify a 1-row-by-6-column array of datetime arrays.
- *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.
- 1-by-1 array

  Specify a 1-row-by-1-column array of `datatime` arrays. To create the array, use the `datetime` function.
- *M*-by-1 array

  Specify an *M*-by-1 array of `datatime` arrays for *M* transformation matrices, one for each UTC date. To create the array, use the `datetime` function.

Example: `[2000 1 12 4 52 12.4]` 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]` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2013b

See Also

`ecef2lla` | `ecef2eci` | `eci2ecef` | `geoc2geod` | `geod2geoc` | `lla2ecef` | `datetime`

Topics
https://www.iers.org

decyear

Decimal year calculator

Syntax

```
dy = decyear(datetime)
dy = decyear(dateVector)
dy = decyear(dateCharacterVector,format)

dy = decyear(year,month,day)
dy = decyear([year,month,day])
dy = decyear(year,month,day,hour,minute,second)
dy = decyear([year,month,day,hour,minute,second])
```

Description

`dy = decyear(datetime)` converts one or more `datetime` arrays to decimal year, `dy`. Some applications require decimal years for better precision. For more information on decimal years and how `decyear` calculates them, see “Algorithms” on page 4-406.

`dy = decyear(dateVector)` converts one or more date vectors, `dateVector`, into decimal year, `dy`.

`dy = decyear(dateCharacterVector,format)` converts one or more date character vectors, `dateCharacterVector`, to decimal year using format `format`.

`dy = decyear(year,month,day)` and `dy = decyear([year,month,day])` return the decimal year for corresponding elements of the `year,month,day` arrays.

`dy = decyear(year,month,day,hour,minute,second)` and `dy = decyear([year,month,day,hour,minute,second])` return the decimal year for corresponding elements of the `year,month,day,hour,minute,second` arrays. Specify the six arguments as one-dimensional arrays of the same length or as scalar values.

Examples

Calculate Decimal Year Using `datetime` Array

Calculate the decimal year for February 4, 2016 from `datetime` array.

```
dt = datetime('04-02-2016','InputFormat','dd-MM-yyyy')
```

```
dt = datetime
   04-Feb-2016
```

```
dy = decyear(dt)
```

```
dy = 2.0161e+03
```

Calculate Decimal Year Using Data Character Vector and `dd-mm-yyyy` Format

Calculate decimal year for May 24, 2005 using data character vector and `dd-mm-yyyy` format.

```
dy = decyear('24-May-2005','dd-mmm-yyyy')
```

```
dy = 2.0054e+03
```

Calculate Decimal Year Using Year, Month, and Day Inputs

Calculate the decimal year for December 19, 2006 from year, month, and day inputs.

```
dy = decyear(2006,12,19)
```

```
dy = 2.0070e+03
```

Calculate Decimal Year from Year, Month, Day, Hour, Minute, and Second Inputs

Calculate the decimal year for October 10, 2004, at 12:21:00 p.m. from year, month, day, hour, month, and second inputs.

```
dy = decyear(2004,10,10,12,21,0)
```

```
dy = 2.0048e+03
```

Input Arguments

`datetime` — `datetime` array

m-by-1 array | 1-by-*m* array

`datetime` array, specified as an *m*-by-1 array or 1-by-*m* array.

`dateVector` — Full or partial date vector

m-by-6 matrix | *m*-by-3 matrix | positive double-precision number

Full or partial date vector, specified as an *m*-by-6 or *m*-by-3 matrix containing *m* full or partial date vectors, respectively:

- Full date vector — Contains six elements specifying the year, month, day, hour, minute, and second
- Partial date vector — Contains three elements specifying the year, month, and day

Data Types: `double`

`dateCharacterVector` — Date character vector

character array | one-dimensional cell array of character vectors

Date character vector, specified as a character array, where each row corresponds to one date, or a one-dimensional cell array of character vectors.

Data Types: `char` | `string`

`format` — Date format

`-1` (default) | character vector | string scalar | integer

Date format, specified as a character vector, string scalar, or integer. All dates in `dateCharacterVector` must have the same format and use the same date format symbols as the `datenum` function.

`decyear` does not accept formats containing the letter `Q`.

If `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.

Data Types: `char` | `string`

`year` — Year

current year (default) | scalar | one-dimensional array

Year, specified as a scalar or one-dimensional array.

Dates with two character years are interpreted to be within 100 years of the current year.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as one-dimensional arrays of the same length or scalar values.

Data Types: `double`

`month` — Month

1 (default) | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `10` | `11` | `12` | one-dimensional array

Month, specified as a scalar or one-dimensional array from `1` to `12`.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as one-dimensional arrays of the same length or scalar values.

Data Types: `double`

`day` — Day

1 (default) | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `10` | `11` | `12` | `13` | `14` | `15` | `16` | `17` | `18` | `19` | `20` | `21` | `22` | `23` | `24` | `25` | `26` | `27` | `28` | `29` | `30` | `31` | one-dimensional array

Day, specified as a scalar or one-dimensional array from 1 to 31.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as one-dimensional arrays of the same length or scalar values.

Data Types: `double`

`hour` — Date format

`0` (default) | double, whole number, `0` to `24`

Hour, specified as a scalar from `0` to `24`.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as one-dimensional arrays of the same length or scalar values.

Data Types: `double`

`minute` — Minute

`0` (default) | double, whole number, `0` to `60`

Minute, specified as a double, whole number from `0` to `60`.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as one-dimensional arrays of the same length or scalar values.

Data Types: `double`

`second` — Second

`0` (default) | double, whole number, `0` to `60`

Second, specified as a double, whole number from `0` to `60`.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as one-dimensional arrays of the same length or scalar values.

Data Types: `double`

Output Arguments

`dy` — Decimal year

m-by-6 column vector | *m*-by-3 column vector | row vector | column vector

Decimal year, returned as a row or column vector.

- *m*-by-6 column vector — Contains six elements specifying the year, month, day, hour, minute, and second
- *m*-by-3 column vector — Contains three elements specifying the year, month, and day
- Row or column vector — Contains *m* decimal years

Dependencies

The output format depends on the input format:

Input Syntax	dy Format
`dy = decyear(datetime)`	Column or row vector of *m* decimal years

Input Syntax	dy Format
`dy = decyear(dateVector)`	*m*-by-6 column vector or *m*-by-3 column vector of *m* decimal years
`dy = decyear(dateCharacterVector,format)`	Column vector of *m* decimal years, where *m* is the number of character vectors in `dateCharacterVector`

Algorithms

The `decyear` function calculates a decimal year as a representation of time expressed as the current year plus the time elapsed since the beginning of the current year as a fraction of the whole year. Since there are 365 days in one common year, a tenth of the common year is equal to 36.5 days (36 days and 12 hours).

Consider 2022.1. The .1 correlates to 36.5 days into the year 2022, which is February 6, 2022 at 12:00:00 (31 days in January + 6.5 days in February). Similarly, 2022.2 correlates to 73 days into the year 2022 (36.5 x 2), or March 15, 2022 at 00:00:00.

In a leap year, one tenth of a leap year equals 36.6 days. Thus, 2020.1 corresponds with February 6, 2020 at 14:24:00.

Version History

Introduced in R2006b

See Also

`juliandate` | `leapyear` | `mjuliandate` | `datenum` | `datestr` | `datetime`

delete

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

Destroy animation object

Syntax

```
delete(h)
```

Description

`delete(h)` destroys the animation object `h`. This function also destroys the animation object figure, and any objects that the animation object contains, for example, bodies, camera, and geometry.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

Examples

Delete Animation Object

```
h=Aero.Animation;
delete(h);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

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;
```

Version History

Introduced in R2007a

See Also

`initialize`

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;
```

Version History

Introduced in R2007b

See Also

`initialize`

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 Coordinated Universal 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

R2020b: Updated `aeroiersdata.mat` file

Behavior changed in R2020b

The contents of the `aeroiersdata.mat` file have been updated. Correspondingly, the output of this function will have different results when using the default value (`'aeroiersdata.mat'`) as the value of `Source`. The results reflect more accurate external data from the International Earth Rotation and Reference Systems Service (IERS).

See Also

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

Topics

“Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-96

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2017b

R2020b: Updated `aeroiersdata.mat` file
Behavior changed in R2020b

The contents of the `aeroiersdata.mat` file have been updated. Correspondingly, the output of this function will have different results when using the default value (`'aeroiersdata.mat'`) as the value of `Source`. The results reflect more accurate external data from the International Earth Rotation and Reference Systems Service (IERS).

See Also

`aeroReadIERSData` | `dcmeci2ecef` | `lla2eci` | `eci2lla` | `eci2aer` | `mjuliandate`

Topics
“Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-96

dpressure

Compute dynamic pressure using velocity and density

Syntax

```
dynamic_pressure = dpressure(velocity,rho)
```

Description

`dynamic_pressure = dpressure(velocity,rho)` computes dynamic pressure, `dynamic_pressure`, from an *m*-by-3 array of Cartesian velocity vectors, `velocity`, and a 1-D array of densities, `rho`. `velocity` and `rho` must have the same length units.

Examples

Determine Dynamic Pressure for Velocity in Feet per Second and Density in Slugs per Feet Cubed

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

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

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+03 *

    1.7983
    0.0053
```

Input Arguments

`velocity` — Cartesian velocity vectors

m-by-3 array | vector

Cartesian velocity vectors, specified as an *m*-by-3 array. `velocity` and `rho` must have the same length.

Data Types: `double`

`rho` — Density

array

Density, specified as an array of *m* densities. `velocity` and `rho` must have the same number of rows.

Data Types: `double`

Output Arguments

`dynamic_pressure` — Dynamic pressure

scalar | array

Dynamic pressure, returned as a scalar or array of *m* pressures.

Version History

Introduced in R2006b

See Also

`airspeed` | `machnumber`

earthNutation

Implement Earth nutation

Syntax

```
angles = earthNutation(ephemerisTime)
angles = earthNutation(ephemerisTime,ephemerisModel)
angles = earthNutation(ephemerisTime,ephemerisModel,action)

[angles,rates] = earthNutation( ___ )
```

Description

Implement Earth Nutation Angles

`angles = earthNutation(ephemerisTime)` implements the International Astronomical Union (IAU) 1980 nutation series for `ephemerisTime`, expressed in Julian days. The function 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.

Implement Earth Nutation Angles and Rates

`[angles,rates] = earthNutation( ___ )` implements the International Astronomical Union (IAU) 1980 nutation series using any combination of the input arguments in the previous syntaxes. The function 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 positions are calculated, specified as these values:

- 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: `double`

`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 and model simulation continues.
`'Error'`	MATLAB returns an exception and model simulation stops.

Data Types: `char` | `string`

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.

Version History

Introduced in R2013a

References

[1] Folkner, W. M., J. G. Williams, and D. H. Boggs. "The Planetary and Lunar Ephemeris DE 421." *JPL Interplanetary Network Progress Report 24-178*, 2009.

[2] Vallado, David A. *Fundamentals of Astrodynamics and Applications*. McGraw-Hill, 1997.

See Also

`juliandate` | `moonLibration` | `planetEphemeris`

External Websites

https://ssd.jpl.nasa.gov/planets/eph_export.html

ecef2eci

Position and velocity vectors in Earth-centered inertial mean-equator mean-equinox

Syntax

```
[r_eci] = ecef2eci(utc,r_ecef)
[r_eci,v_eci] = ecef2eci( ___ ,v_ecef)
[r_eci,v_eci,a_eci] = ecef2eci( ___ ,a_ecef)
[r_eci,v_eci,a_eci] = ecef2eci( ___ ,Name,Value)
```

Description

`[r_eci] = ecef2eci(utc,r_ecef)` calculates the position vector in the Earth-centered inertial (ECI) coordinate system for a given position vector in the Earth-centered Earth-fixed (ECEF) coordinate system at a specific Coordinated Universal Time (UTC). For more information on the Earth-centered Earth-fixed coordinate system, see “Algorithms” on page 4-426.

`[r_eci,v_eci] = ecef2eci( ___ ,v_ecef)` calculates the position and velocity vectors for given position and velocity vectors.

`[r_eci,v_eci,a_eci] = ecef2eci( ___ ,a_ecef)` calculates the position, velocity, acceleration vectors for given position, velocity, and acceleration vectors.

`[r_eci,v_eci,a_eci] = ecef2eci( ___ ,Name,Value)` calculates the position, velocity, and acceleration vectors at a higher precision using Earth orientation parameters. If Earth orientation parameters are not specified, the function sets them to `0`.

Examples

Convert ECEF Position and Velocity to ECI

Convert the ECEF position and velocity to ECI at 12:00 on January 4, 2019.

```
r_ecef = [-5762640 -1682738 3156028];
v_ecef = [3832 -4024 4837];
utc = [2019 1 4 12 0 0];
[r_eci, v_eci] = ecef2eci(utc, r_ecef, v_ecef);
```

```
r_eci =
   1.0e+06 *
   -2.9818
    5.2070
    3.1616

v_eci =
   1.0e+03 *
   -3.3837
   -4.8870
    4.8430
```

Convert ECEF Position to ECI Including Polar Motion Effects

Convert the ECEF position to ECI at 12:00 on January 4, 2019, including the effects of polar motion.

```
r_ecef = [-5762640 -1682738 3156028];
utc = [2019 1 4 12 0 0];
mjd = mjuliandate(utc);
pm = polarMotion(mjd, 'action', 'none')*180/pi;
r_eci = ecef2eci(utc, r_ecef, 'pm', pm);
```

```
r_eci =
   1.0e+06 *
   -2.9818
   5.2070
   3.1616
```

Convert ECEF Position to ECI Using `datetime` Arrays

Convert ECEF position and velocity to ECI at 12:00 on January 4, 2019 with `datetime` array `utcDT`.

```
r_ecef = [-5762640 -1682738 3156028];
v_ecef = [3832 -4024 4837];
utcDT = datetime(2019, 1, 4, 12, 0, 0)
[r_eci, v_eci] = ecef2eci(utcDT, r_ecef, v_ecef)
```

```
utcDT =
  datetime

   04-Jan-2019 12:00:00

r_eci =
   1.0e+06 *

   -2.9818
    5.2070
    3.1616

v_eci =
   1.0e+03 *

   -3.3837
   -4.8870
    4.8430
```

Input Arguments

`utc` — Coordinated Universal Time

1-by-6 array | 1-by-6 matrix | scalar `datetime` array

Coordinated Universal Time (UTC) specified as one of these:

- 1-by-6 array of UTC values in the order year, month, day, hour, minutes, and seconds:

Time Value	Enter
Year	Double value that is a whole number greater than 1, such as `2013`.
Month	Double value that is a whole number greater than 0, within the range `1` to `12`.
Day	Double value that is a whole number greater than 0, within the range `1` to `31`.
Hour	Double value that is a whole number greater than 0, within the range `1` to `24`.
Minute and second	Double value that is a whole number greater than 0, within the range `1` to `60`.

- Scalar `datetime` array. To create the array, use the `datetime` function.

Example: `[2000 1 12 4 52 12.4]`

Data Types: `double`

`r_ecef` — Position components
3-by-1 array

Array of ECEF position components, specified as a 3-by-1 array.

Data Types: `double`

`v_ecef` — Velocity components
3-by-1 array

ECEF velocity components, specified as a 3-by-1 array.

Data Types: `double`

`a_ecef` — Acceleration components
3-by-1 array

ECEF acceleration components, specified as a 3-by-1 array.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'dUT1',0.234`

`dAT` — Difference between TAI and UTC
`0` (default) | scalar

Difference between International Atomic Time (TAI) and UTC, specified as a scalar, in seconds.

Example: 32

Data Types: `double`

`dUT1` — Difference between UTC and Universal Time
`0` (default) | scalar

Difference between UTC and Universal Time (UT1), specified as a scalar, in seconds.

Example: 0.234

Data Types: `double`

`pm` — Polar displacement
array of zeroes (default) | 1-by-2 array

Polar displacements due to the motion of Earth crust along the *x*- and *y*-axis, in degrees.

Tip To calculate the displacement, use the `polarMotion` function.

Example: `pm = polarMotion(mjd, 'action', 'none')*180/pi;`

Data Types: `double`

`dCIP` — Adjustment to the CIP location
1-by-2 array

Adjustment to the location of the Celestial Intermediate Pole (CIP), in degrees, specified as a comma-separated pair consisting of `dCIP` and an *M*-by-2 array. This location (*dDeltaX*, *dDeltaY*) is along the *x*- and *y*- axes. By default, this function assumes a 1-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 an *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`

`lod` — Excess length of day
`0` (default) | scalar

Excess length of day (difference between astronomically determined duration of day and 86400 SI seconds), specified as a scalar, in seconds.

Example: 32

Data Types: `double`

Output Arguments

`r_eci` — Position components
3-by-1 array

ECI position components, specified as a 3-by-1 array.

`v_eci` — Velocity components
3-by-1 array

ECI velocity components, specified as a 3-by-1 array.

`a_eci` — Acceleration components
3-by-1 array

ECI acceleration components, specified as a 3-by-1 array.

Algorithms

The `ecef2eci` function uses these Earth-centric coordinate systems:

- Earth Centered Inertial Frame (ECI) — The inertial frame used is the International Celestial Reference Frame (ICRF). This frame can be treated as equal to the ECI coordinate system realized at J2000 (Jan 1 2000 12:00:00 TT). For more information, see “ECI Coordinates” on page 2-7.
- Earth-centered Earth-fixed Frame (ECEF) — The fixed-frame used is the International Terrestrial Reference Frame (ITRF). This reference frame is realized by the IAU2000/2006 reduction from the ICRF coordinate system. For more information, see “ECEF Coordinates” on page 2-8.

Version History

Introduced in R2019a

References

[1] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. alg. 4. New York: McGraw-Hill, 1997.

[2] 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.

[3] 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.

[4] 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.

[5] Seidelmann, P.K., Archinal, B.A., A'hearn, M.F. et al. "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006." *Celestial Mech Dyn Astr* 98, 155–180 (2007).

See Also

`eci2ecef` | `dcmeci2ecef` | `aeroReadIERSData` | `deltaCIP` | `polarMotion` | `deltaUT1` | `siderealTime` | `datetime` | CubeSat Vehicle

ecef2lla

Convert Earth-centered Earth-fixed (ECEF) coordinates to geodetic coordinates

Syntax

```
lla = ecef2lla(ecef)
lla = ecef2lla(ecef,model)
lla = ecef2lla(ecef,f,Re)
```

Description

`lla = ecef2lla(ecef)` converts the *m*-by-3 array of ECEF coordinates, `ecef`, to an *m*-by-3 array of geodetic coordinates (latitude, longitude and altitude), `lla`.

`lla = ecef2lla(ecef,model)` converts the coordinates for a specific ellipsoid planet.

`lla = ecef2lla(ecef,f,Re)` converts the coordinates for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`.

Examples

Determine Latitude, Longitude, and Altitude at One ECEF Coordinate

Determine latitude, longitude, and altitude at an ECEF coordinate:

```
lla = ecef2lla([4510731 4510731 0])
```

```
lla =

         0   45.0000  999.9564
```

Determine Latitude, Longitude, and Altitude at Multiple ECEF Coordinates

Determine latitude, longitude, and altitude at multiple ECEF coordinates with the 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 ECEF Coordinates with Custom Ellipsoid Model

Determine latitude, longitude, and altitude at multiple coordinates with the custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
lla = ecef2lla([4510731 4510731 0; 0 4507609 4498719], f, Re)

lla =

   1.0e+06 *

         0    0.0000    2.9821
    0.0000    0.0001    2.9801
```

Input Arguments

`ecef` — ECEF coordinates

m-by-3 array | vector

ECEF coordinates, specified as an *m*-by-3 array.

Data Types: `double`

`model` — Ellipsoid planet model

`'WGS84'` (default)

Ellipsoid planet model, specified as `'WGS84'`.

Data Types: `double`

`f` — Flattening

scalar

Flattening at each pole, specified as a scalar.

Data Types: `double`

`Re` — Equatorial radius

scalar

Equatorial radius, specified as a scalar in meters.

Data Types: `double`

Output Arguments

`lla` — Geodetic coordinates

m-by-3 array

Geodetic coordinates (latitude, longitude, and altitude), returned as an *m*-by-3 array in [degrees degrees meters].

Version History

Introduced in R2006b

See Also

`geoc2geod` | `geod2geoc` | `lla2ecef`

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 Coordinated Universal 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 Coordinated Universal 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` — Coordinated Universal Time

1-by-6 array of whole numbers | *M*-by-6 matrix of whole numbers

Coordinated Universal 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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`

Version History

Introduced in R2015a

See Also

`dcmeci2ecef` | `ecef2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef` | `lla2eci`

eci2ecef

Position, velocity, and acceleration vectors in Earth-centered Earth-fixed (ECEF) coordinate system

Syntax

```
[r_ecef,v_ecef,a_ecef] = eci2ecef(utc,r_eci,v_eci,a_eci)
[r_ecef,v_ecef,a_ecef] = eci2ecef(utc,r_eci,v_eci,a_eci,Name,Value)
```

Description

`[r_ecef,v_ecef,a_ecef] = eci2ecef(utc,r_eci,v_eci,a_eci)` calculates position, velocity, and acceleration vectors in Earth-centered Earth-fixed (ECEF) coordinate system for given position, velocity, and acceleration vectors in the Earth-centered inertial (ECI) coordinate system at a specific Coordinated Universal Time (UTC). For more information on the Earth-centered Earth-fixed coordinate system, see “Algorithms” on page 4-440.

`[r_ecef,v_ecef,a_ecef] = eci2ecef(utc,r_eci,v_eci,a_eci,Name,Value)` calculates the position, velocity, and acceleration vectors at a higher precision using Earth orientation parameters. If Earth orientation parameters are not specified, the function sets them to `0`.

Examples

Convert ECI Position and Velocity to ECEF

Convert ECI position and velocity to ECEF at 12:00 on January 4, 2019.

```
r_eci = [-2981784 5207055 3161595];
v_eci = [-3384 -4887 4843];
utc = [2019 1 4 12 0 0];
[r_ecef, v_ecef] = eci2ecef(utc, r_eci, v_eci)
```

```
r_ecef =
   1.0e+06 *
   -5.7627
   -1.6827
    3.1560

v_ecef =
   1.0e+03 *
    3.8319
   -4.0243
    4.8370
```

Convert ECI Position to ECEF Including Polar Motion Effects

Convert ECI position to ECEF at 12:00 on January 4, 2019 including effects of polar motion.

```
r_eci = [-2981784 5207055 3161595];
utc = [2019 1 4 12 0 0];
```

```
mjd = mjuliandate(utc);
pm = polarMotion(mjd, 'action', 'none')*180/pi;
r_ecef = eci2ecef(utc, r_eci, 'pm', pm)

r_ecef =
   1.0e+06 *
   -5.7627
   -1.6827
    3.1560
```

Convert ECI Position to ECEF Using `datetime` Arrays

Convert ECI position and velocity to ECEF at 12:00 on January 4, 2019 with `datetime` array `utcDT`.

```
r_eci = [-2981784 5207055 3161595];
v_eci = [-3384 -4887 4843];
utcDT = datetime(2019, 1, 4, 12, 0, 0)
[r_ecef, v_ecef] = eci2ecef(utcDT, r_eci, v_eci)

utcDT =
  datetime

   04-Jan-2019 12:00:00

r_ecef =
   1.0e+06 *

   -5.7627
   -1.6827
    3.1560

v_ecef =
   1.0e+03 *

    3.8319
   -4.0243
    4.8370
```

Input Arguments

`utc` — Coordinated Universal Time

scalar | 1-by-6 array | 1-by-6 matrix | scalar `datetime` array

Coordinated Universal Time (UTC) specified as one of these:

- 1-by-6 array of UTC values in the order year, month, day, hour, minutes, and seconds:

Time Value	Enter
Year	Double value that is a whole number greater than 1, such as 2013.
Month	Double value that is a whole number greater than 0, within the range 1 to 12.

Time Value	Enter
Day	Double value that is a whole number greater than 0, within the range 1 to 31.
Hour	Double value that is a whole number greater than 0, within the range 1 to 24.
Minute and second	Double value that is a whole number greater than 0, within the range 1 to 60.

- Scalar `datetime` array. To create the array, use the `datetime` function.

Example: `[2000 1 12 4 52 12.4]` is a one row-by-6 column array of UTC values.

Data Types: `double`

`r_eci` — Position components
3-by-1 array

ECI position components, specified as a 3-by-1 array.

Data Types: `double`

`v_eci` — Velocity components
3-by-1 array

ECI velocity components, specified as a 3-by-1 array.

Data Types: `double`

`a_eci` — Acceleration components
3-by-1 array

ECI acceleration components, specified as a 3-by-1 array.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'dUT1',0.234`

`dAT` — Difference between TAI and UTC
`0` (default) | scalar

Difference between International Atomic Time (TAI) and UTC, specified as a scalar, in seconds.

Example: 32

Data Types: `double`

`dUT1` — Difference between UTC and Universal Time
`0` (default) | scalar

Difference between UTC and Universal Time (UT1), specified as a scalar, in seconds.

Example: 0.234

Data Types: `double`

`pm` — Polar displacement

array of zeroes (default) | 1-by-2 array

Polar displacements due to the motion of Earth crust along the *x*- and *y*-axis, in degrees.

Tip To calculate the displacement, use the `polarMotion` function.

Example: `pm = polarMotion(mjd, 'action', 'none')*180/pi;`

Data Types: `double`

`dCIP` — Adjustment to the CIP

1-by-2 array

Adjustment to the location of the Celestial Intermediate Pole (CIP), in degrees, specified as a comma-separated pair consisting of `dCIP` and an *M*-by-2 array. This location (*dDeltaX*, *dDeltaY*) is along the *x*- and *y*- axes. By default, this function assumes a 1-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 **Data/Products/Tools** > **Earth orientation data**.

- *M*-by-2 array

  Specify a *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`

`lod` — Excess length of day

`0` (default) | scalar

Excess length of day (difference between astronomically determined duration of day and 86400 SI seconds), specified as a scalar, in seconds.

Example: 32

Data Types: `double`

Output Arguments

`r_ecef` — Position components

3-by-1 array

ECEF position components, specified as a 3-by-1 array.

`v_ecef` — Velocity components

3-by-1 array

ECEF velocity components, specified as a 3-by-1 array.

a_ecef — Acceleration components
3-by-1 array

ECEF acceleration components, specified as a 3-by-1 array.

Algorithms

The `eci2ecef` function uses these Earth-centric coordinate systems:

- Earth Centered Inertial Frame (ECI) — The inertial frame used is the International Celestial Reference Frame (ICRF). This frame can be treated as equal to the ECI coordinate system realized at J2000 (Jan 1 2000 12:00:00 TT). For more information, see “ECI Coordinates” on page 2-7.
- Earth-centered Earth-fixed Frame (ECEF) — The fixed-frame used is the International Terrestrial Reference Frame (ITRF). This reference frame is realized by the IAU2000/2006 reduction from the ICRF coordinate system. For more information, see “ECEF Coordinates” on page 2-8.

Version History

Introduced in R2019a

References

[1] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. alg. 4. New York: McGraw-Hill, 1997.

[2] 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.

[3] 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.

[4] 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.

[5] Seidelmann, P.K., Archinal, B.A., A'hearn, M.F. et al. "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006." *Celestial Mech Dyn Astr* 98, 155–180 (2007).

See Also

`ecef2eci` | `dcmeci2ecef` | `aeroReadIERSData` | `deltaCIP` | `polarMotion` | `deltaUT1` | `siderealTime` | `datetime` | CubeSat Vehicle

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 Coordinated Universal 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 Coordinated Universal 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` — Coordinated Universal Time

1-by-6 array | *M*-by-6 matrix

Coordinated Universal 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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`

Version History

Introduced in R2014a

See Also

`dcmeci2ecef` | `ecef2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef` | `lla2eci`

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) | 1-by-n string array | 1-by-n cell array | n-by-3 array of RGB triplets | hexadecimal color code | ...

Scale colors, specified as one of the following arrays:

- A 1-by-n string array of color options, such as `["blue" "green" "red"]`.

- An n-by-3 array of RGB triplets, such as `[0 0 1;1 1 0]`.
- A 1-by-n cell array containing RGB triplets, hexadecimal color codes, or named color options. For example, `{'#EDB120','#7E2F8E','#77AC30'}`.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- 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]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	

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 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) | on/off logical value

Visibility of the EGT indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`. 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.

`ContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object created using the `uicontextmenu` function. Use this property to display a context menu when you right-click on a component.

`Enable` — Operational state of EGT indicator

`'on'` (default) | on/off logical value

Operational state of EGT indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- 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 = uigridlayout([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 “Callbacks in App Designer”.

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 “Callbacks in App Designer”.

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) | on/off logical value

Callback interruption, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

This 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.

MATLAB determines callback interruption behavior whenever it executes a command that processes the callback queue. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines if the interruption occurs:

- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the object that owns the interrupting callback determines if the interrupting callback is discarded or added to the callback queue.
- If the value of `Interruptible` is `'on'`, then the interruption occurs. The next time MATLAB processes the callback queue, it stops the execution of the running callback and executes the interrupting callback. After the interrupting callback completes, MATLAB then resumes executing the running callback.

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.
- If the interrupting callback is owned by a `Timer` object, then the callback executes according to schedule regardless of the `Interruptible` property value.

Note 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.

The `BusyAction` property determines callback queuing behavior only when both of these conditions are met:

- The running callback contains a command that processes the callback queue, such as `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause`.
- The value of the `Interruptible` property of the object that owns the running callback is `'off'`.

Under these conditions, the `BusyAction` property of the object that owns the interrupting callback determines how MATLAB handles the interrupting callback. 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

on/off logical value

This property is read-only.

Deletion status, returned as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

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

`'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”.

Version History

Introduced in R2018b

See Also

`uiaeroegt`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

fieldOfView

Package: `matlabshared.satellitescenario`

Visualize field of view of conical sensor

Syntax

```
fieldOfView(sensor)
fieldOfView(sensor,Name,Value)
fov = fieldOfView( ___ )
```

Description

`fieldOfView(sensor)` adds a `FieldOfView` object to the specified conical sensor, and draws contours on the Earth. Each contour represents the field of view of a conical sensor in `sensor` based on the current state of the scenario.

Locations inside the contour are inside the field of view. The field of view contours are drawn on all open satellite scenario viewers. The contours are the lines of intersection of the surface of the earth and the field of view cone. The half angle of the field of view cone equals the `MaxViewAngle` property of the conical sensor, and the axis of the cone is the *z*-axis (or boresight) of the conical sensor. The vertex of the cone is located at the position of the conical sensor. The cone becomes wider along the positive body *z*-axis of the conical sensor.

`fieldOfView(sensor,Name,Value)` specifies options by using one or more name-value arguments.

`fov = fieldOfView( ___ )` returns a vector of handles to the added field of view graphic objects. Specify any input combination from previous syntaxes.

Examples

Calculate Maximum Revisit Time of Satellite

Create a satellite scenario with a start time of 15-June-2021 8:55:00 AM UTC and a stop time of five days later. Set the simulation sample time to `60` seconds.

```
startTime = datetime(2021,6,21,8,55,0);
stopTime = startTime + days(5);
sampleTime = 60;                                          % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc =
  satelliteScenario with properties:

         StartTime: 21-Jun-2021 08:55:00
          StopTime: 26-Jun-2021 08:55:00
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
```

```
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add a satellite to the scenario using Keplerian orbital elements.

```
semiMajorAxis = 7878137;                                                                % me
eccentricity = 0;
inclination = 50;                                                                       % deg
rightAscensionOfAscendingNode = 0;                                                      % deg
argumentOfPeriapsis = 0;                                                                % deg
trueAnomaly = 50;                                                                       % deg
sat = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly)

sat =
  Satellite with properties:

               Name:  Satellite 1
                 ID:  1
     ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
            Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
       Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
          Receivers:  [1x0 satcom.satellitescenario.Receiver]
           Accesses:  [1x0 matlabshared.satellitescenario.Access]
         GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
               Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
    OrbitPropagator:  sgp4
        MarkerColor:  [0.059 1 1]
         MarkerSize:  6
          ShowLabel:  true
     LabelFontColor:  [1 1 1]
      LabelFontSize:  15
```

Add a ground station, which represents the location to be photographed, to the scenario.

```
gs = groundStation(sc,Name="Location to Photograph", ...
    Latitude=42.3001,Longitude=-71.3504)                % degrees

gs =
  GroundStation with properties:

                  Name:  Location to Photograph
                    ID:  2
              Latitude:  42.3 degrees
             Longitude:  -71.35 degrees
              Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
           MarkerColor:  [1 0.4118 0.1608]
            MarkerSize:  6
             ShowLabel:  true
        LabelFontColor:  [1 1 1]
```

```
        LabelFontSize:  15
```

Add a gimbal to the satellite. You can steer this gimbal independently of the satellite.

```
g = gimbal(sat)
```

```
g =
  Gimbal with properties:

                Name:  Gimbal 3
                  ID:  3
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
```

Track the location to be photographed using the gimbal.

```
pointAt(g,gs);
```

Add a conical sensor to the gimbal. This sensor represents the camera. Set the field of view to 60 degrees.

```
camSensor = conicalSensor(g,MaxViewAngle=60)
```

```
camSensor =
  ConicalSensor with properties:

                Name:  Conical sensor 4
                  ID:  4
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
        MaxViewAngle:  60 degrees
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         FieldOfView:  [0x0 matlabshared.satellitescenario.FieldOfView]
```

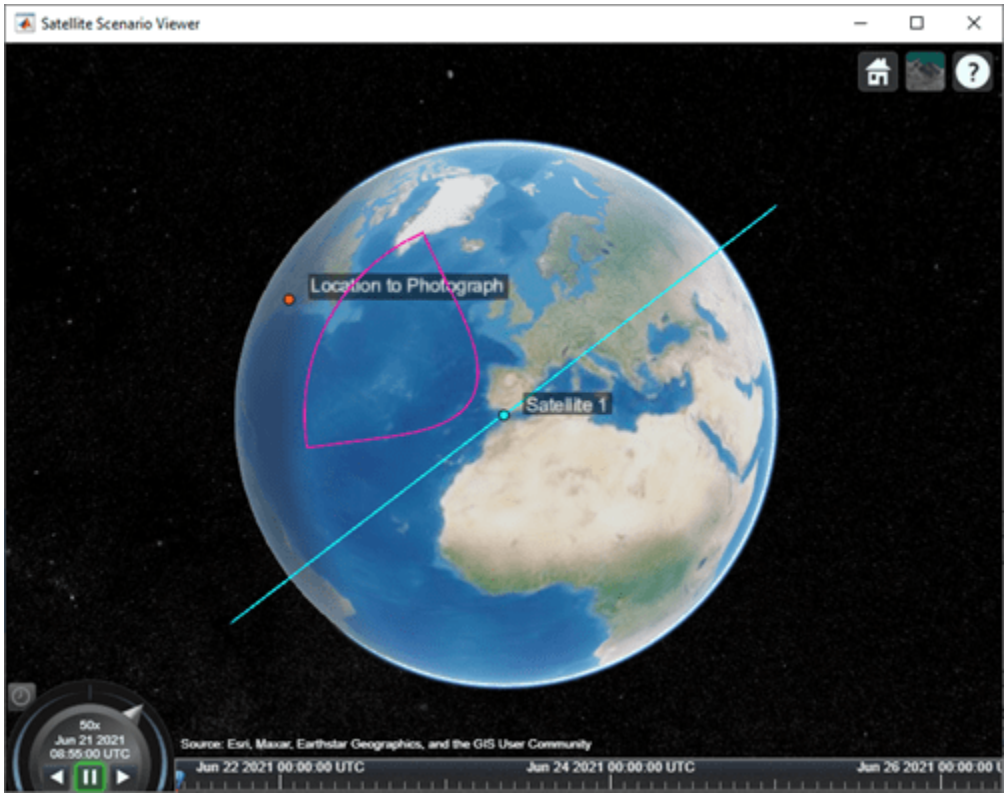
Add access analysis to the conical sensor between the camera and the location to be photographed.

```
ac = access(camSensor,gs)
```

```
ac =
  Access with properties:

    Sequence:  [4 2]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

Visualize the field of view of the camera by using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
fieldOfView(camSensor);
```



Determine the intervals during which the camera can see the geographical site.

```
t = accessIntervals(ac)
```

```
t=35×8 table
          Source                 Target            IntervalNumber        StartTime
    ________________    ________________________    ______________    ____________________

    "Conical sensor 4"    "Location to Photograph"          1          21-Jun-2021 10:38:00
    "Conical sensor 4"    "Location to Photograph"          2          21-Jun-2021 12:36:00
    "Conical sensor 4"    "Location to Photograph"          3          21-Jun-2021 14:37:00
    "Conical sensor 4"    "Location to Photograph"          4          21-Jun-2021 16:41:00
    "Conical sensor 4"    "Location to Photograph"          5          21-Jun-2021 18:44:00
    "Conical sensor 4"    "Location to Photograph"          6          21-Jun-2021 20:46:00
    "Conical sensor 4"    "Location to Photograph"          7          21-Jun-2021 22:50:00
    "Conical sensor 4"    "Location to Photograph"          8          22-Jun-2021 09:51:00
    "Conical sensor 4"    "Location to Photograph"          9          22-Jun-2021 11:46:00
    "Conical sensor 4"    "Location to Photograph"         10          22-Jun-2021 13:46:00
    "Conical sensor 4"    "Location to Photograph"         11          22-Jun-2021 15:50:00
    "Conical sensor 4"    "Location to Photograph"         12          22-Jun-2021 17:53:00
    "Conical sensor 4"    "Location to Photograph"         13          22-Jun-2021 19:55:00
    "Conical sensor 4"    "Location to Photograph"         14          22-Jun-2021 21:58:00
    "Conical sensor 4"    "Location to Photograph"         15          23-Jun-2021 10:56:00
    "Conical sensor 4"    "Location to Photograph"         16          23-Jun-2021 12:56:00
      ⋮
```

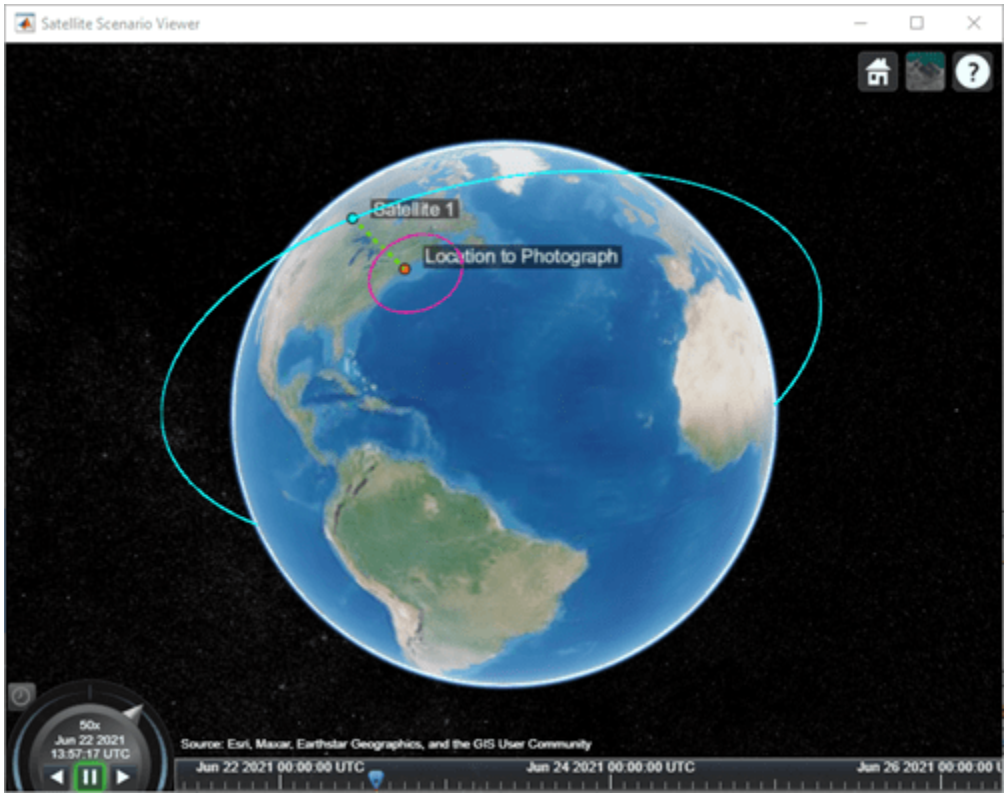
Calculate the maximum revisit time in hours.

```
startTimes = t.StartTime;
endTimes = t.EndTime;
revisitTimes = hours(startTimes(2:end) - endTimes(1:end-1));
maxRevisitTime = max(revisitTimes)                                  % hours
```

```
maxRevisitTime = 12.6667
```

Visualize the revisit times that the camera photographs of the location.

```
play(sc);
```



Input Arguments

`sensor` — Conical sensor

`ConicalSensor` object

Conical sensor, specified as a `ConicalSensor` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'LineWidth',2.5` sets the line width of the field of view to 2.5 pixels.

`Viewer` — Satellite scenario viewer

vector of `satelliteScenarioViewer` objects (default) | scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Satellite scenario viewer, specified as a scalar, vector, or array of `satelliteScenarioViewer` objects. If the `AutoSimulate` property of the scenario is `false`, adding a satellite to the scenario disables any previously available timeline and playback widgets.

`NumContourPoints` — Number of contour points

`40` (default) | integer greater than or equal to 4

Number of contour points used to draw the contour of the field of view, specified as an integer greater than or equal to 4.

Data Types: `double`

`LineWidth` — Visual width of field of view contour

`1` (default) | scalar in the range (0 10]

Visual width of the field of view contour in pixels, specified as a scalar in the range (0 10].

The line width cannot be thinner than the width of a pixel. If you set the line width to a value that is less than the width of a pixel on your system, the line displays as one pixel wide.

`LineColor` — Color of field of view contour

`[0 1 0]` (default) | RGB triplet | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Color of field of view contour, specified as an RGB triplet, hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
"yellow"	"y"	[1 1 0]	"#FFFF00"	
"black"	"k"	[0 0 0]	"#000000"	
"white"	"w"	[1 1 1]	"#FFFFFF"	
"none"	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	"#0072BD"	
[0.8500 0.3250 0.0980]	"#D95319"	
[0.9290 0.6940 0.1250]	"#EDB120"	
[0.4940 0.1840 0.5560]	"#7E2F8E"	
[0.4660 0.6740 0.1880]	"#77AC30"	
[0.3010 0.7450 0.9330]	"#4DBEEE"	
[0.6350 0.0780 0.1840]	"#A2142F"	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

Output Arguments

fov — Field of view of conical sensor

row vector of FieldOfView objects

Field of view of conical sensor, returned as a row vector of FieldOfView objects.

Note When the AutoSimulate property is set to false, the SimulationStatus must equal NotStarted to call the fieldOfView function. Otherwise, use the restart function to reset the SimulationStatus to NotStarted, which erases the simulation data.

Version History

Introduced in R2021a

See Also

Objects

satelliteScenario | satelliteScenarioViewer

Functions

show | play | hide | access | groundStation | conicalSensor

Topics
“Satellite Scenario Key Concepts” on page 2-65

FieldOfView

Field of view object belonging to satellite scenario

Description

The FieldOfView object defines a field of view object belonging to a satellite scenario.

Creation

You can create a `FieldOfView` object using the `fieldOfView` object function of the `ConicalSensor` object.

Properties

`LineWidth` — Visual width of field of view contour
1 (default) | scalar in the range (0 10]

Visual width of the field of view contour in pixels, specified as a scalar in the range (0 10].

The line width cannot be thinner than the width of a pixel. If you set the line width to a value that is less than the width of a pixel on your system, the line displays as one pixel wide.

`LineColor` — Color of field of view contour
`[0 1 0]` (default) | RGB triplet | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Color of field of view contour, specified as an RGB triplet, hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
"blue"	"b"	[0 0 1]	"#0000FF"	
"cyan"	"c"	[0 1 1]	"#00FFFF"	
"magenta"	"m"	[1 0 1]	"#FF00FF"	
"yellow"	"y"	[1 1 0]	"#FFFF00"	
"black"	"k"	[0 0 0]	"#000000"	
"white"	"w"	[1 1 1]	"#FFFFFF"	
"none"	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	"#0072BD"	
[0.8500 0.3250 0.0980]	"#D95319"	
[0.9290 0.6940 0.1250]	"#EDB120"	
[0.4940 0.1840 0.5560]	"#7E2F8E"	
[0.4660 0.6740 0.1880]	"#77AC30"	
[0.3010 0.7450 0.9330]	"#4DBEEE"	
[0.6350 0.0780 0.1840]	"#A2142F"	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

VisibilityMode — Visibility mode of field of view contour
'inherit' (default) | 'manual'

Visibility mode of the field of view contour, specified as one of these values:

- 'inherit' — Visibility of the graphic matches that of the parent
- 'manual' — Visibility of the graphic is not inherited and is independent of that of the parent

Object Functions

show Show object in satellite scenario viewer
hide Hide satellite scenario entity from viewer

Examples

Calculate Maximum Revisit Time of Satellite

Create a satellite scenario with a start time of 15-June-2021 8:55:00 AM UTC and a stop time of five days later. Set the simulation sample time to 60 seconds.

```
startTime = datetime(2021,6,21,8,55,0);
stopTime = startTime + days(5);
sampleTime = 60;                                          % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime)

sc =
  satelliteScenario with properties:

         StartTime: 21-Jun-2021 08:55:00
          StopTime: 26-Jun-2021 08:55:00
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add a satellite to the scenario using Keplerian orbital elements.

```
semiMajorAxis = 7878137;                                                                    % me
eccentricity = 0;
inclination = 50;                                                                           % deg
rightAscensionOfAscendingNode = 0;                                                          % deg
argumentOfPeriapsis = 0;                                                                    % deg
trueAnomaly = 50;                                                                           % deg
sat = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly)

sat =
  Satellite with properties:

              Name:  Satellite 1
                ID:  1
     ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
            Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
       Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
          Receivers:  [1x0 satcom.satellitescenario.Receiver]
           Accesses:  [1x0 matlabshared.satellitescenario.Access]
         GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
              Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
    OrbitPropagator:  sgp4
        MarkerColor:  [0.059 1 1]
         MarkerSize:  6
          ShowLabel:  true
     LabelFontColor:  [1 1 1]
      LabelFontSize:  15
```

Add a ground station, which represents the location to be photographed, to the scenario.

```
gs = groundStation(sc,Name="Location to Photograph", ...
    Latitude=42.3001,Longitude=-71.3504)                % degrees

gs =
  GroundStation with properties:

                Name:  Location to Photograph
                  ID:  2
```

```
            Latitude:  42.3 degrees
           Longitude:  -71.35 degrees
            Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
          MarkerColor:  [1 0.4118 0.1608]
           MarkerSize:  6
            ShowLabel:  true
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
```

Add a gimbal to the satellite. You can steer this gimbal independently of the satellite.

```
g = gimbal(sat)
```

```
g =
  Gimbal with properties:

                Name:  Gimbal 3
                  ID:  3
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
```

Track the location to be photographed using the gimbal.

```
pointAt(g,gs);
```

Add a conical sensor to the gimbal. This sensor represents the camera. Set the field of view to 60 degrees.

```
camSensor = conicalSensor(g,MaxViewAngle=60)
```

```
camSensor =
  ConicalSensor with properties:

                Name:  Conical sensor 4
                  ID:  4
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
        MaxViewAngle:  60 degrees
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         FieldOfView:  [0x0 matlabshared.satellitescenario.FieldOfView]
```

Add access analysis to the conical sensor between the camera and the location to be photographed.

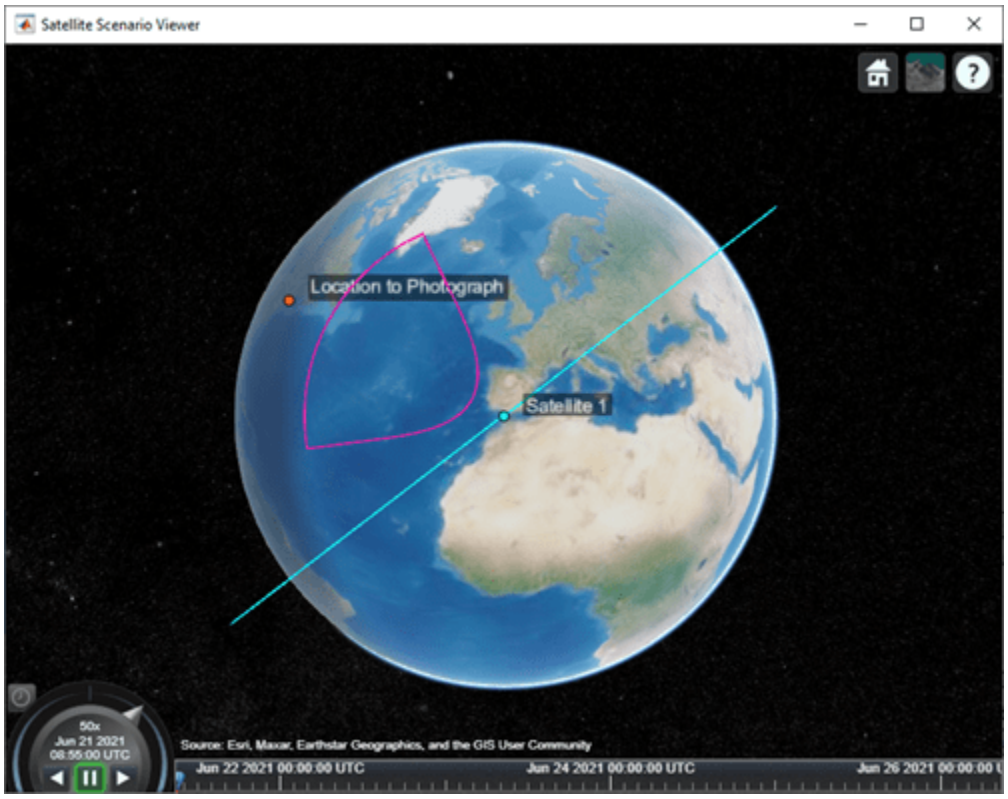
```
ac = access(camSensor,gs)
```

```
ac =
  Access with properties:
```

```
    Sequence:  [4 2]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

Visualize the field of view of the camera by using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
fieldOfView(camSensor);
```



Determine the intervals during which the camera can see the geographical site.

```
t = accessIntervals(ac)
```

t=*35×8 table*

Source	Target	IntervalNumber	StartTime
"Conical sensor 4"	"Location to Photograph"	1	21-Jun-2021 10:38:00
"Conical sensor 4"	"Location to Photograph"	2	21-Jun-2021 12:36:00
"Conical sensor 4"	"Location to Photograph"	3	21-Jun-2021 14:37:00
"Conical sensor 4"	"Location to Photograph"	4	21-Jun-2021 16:41:00
"Conical sensor 4"	"Location to Photograph"	5	21-Jun-2021 18:44:00
"Conical sensor 4"	"Location to Photograph"	6	21-Jun-2021 20:46:00
"Conical sensor 4"	"Location to Photograph"	7	21-Jun-2021 22:50:00
"Conical sensor 4"	"Location to Photograph"	8	22-Jun-2021 09:51:00
"Conical sensor 4"	"Location to Photograph"	9	22-Jun-2021 11:46:00
"Conical sensor 4"	"Location to Photograph"	10	22-Jun-2021 13:46:00
"Conical sensor 4"	"Location to Photograph"	11	22-Jun-2021 15:50:00

```
    "Conical sensor 4"    "Location to Photograph"          12          22-Jun-2021 17:53:00    2
    "Conical sensor 4"    "Location to Photograph"          13          22-Jun-2021 19:55:00    2
    "Conical sensor 4"    "Location to Photograph"          14          22-Jun-2021 21:58:00    2
    "Conical sensor 4"    "Location to Photograph"          15          23-Jun-2021 10:56:00    2
    "Conical sensor 4"    "Location to Photograph"          16          23-Jun-2021 12:56:00    2
      ⋮
```

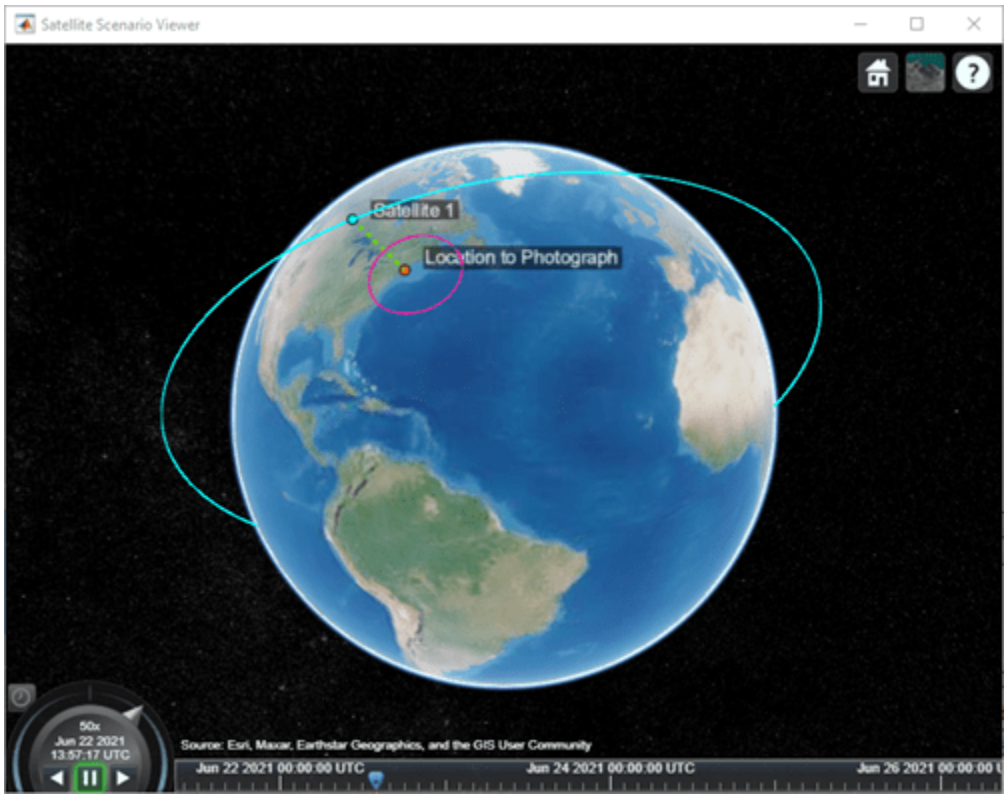
Calculate the maximum revisit time in hours.

```
startTimes = t.StartTime;
endTimes = t.EndTime;
revisitTimes = hours(startTimes(2:end) - endTimes(1:end-1));
maxRevisitTime = max(revisitTimes)                                % hours
```

```
maxRevisitTime = 12.6667
```

Visualize the revisit times that the camera photographs of the location.

```
play(sc);
```



Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `play` | `hide` | `groundStation` | `access`

Topics
“Satellite Scenario Key Concepts” on page 2-65

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
```

Version History

Introduced in R2007a

See Also

`Aero.FlightGearAnimation`

findstartstoptimes

Class: `Aero.Body`
Package: `Aero`

Return start and stop times of time series data

Syntax

```
[tstart,tstop] = findstartstoptimes(h,tsdata)
```

Description

`[tstart,tstop] = findstartstoptimes(h,tsdata)` returns the start and stop times of time series data `tsdata` for the animation body object `h`.

Input Arguments

`h` — Aerospace body object

`Aero.Body` object

Aerospace body object, specified as an `Aero.Body` object.

`tsdata` — Time series data for animation object

`timeseries` object | `timetable` object | `StructureWithTime` | double precision array

Time series data for animation object, specified as a `timeseries` object, `timetable` object, Simulink, or double precision array.

Dependencies

The TimeSeriesSource property affects the interpretation of this argument.

Output Arguments

`tstart` — Start time

scalar

Start time, returned as a scalar.

`tstop` — Stop time

scalar

Stop time, returned as a scalar.

Examples

Find Start and Stop Times

Find the start and stop times of the timeseries 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)
```

```
tstart = 0
```

```
tstop = 10
```

Version History

Introduced in R2007a

See Also

Aero.Animation | Aero.Body

findstartstoptimes (Aero.Node)

Return start and stop times for time series data

Syntax

```
[tstart,tstop] = findstartstoptimes(h,tsdata)
[tstart,stop] = h.findstartstoptimes(tsdata)
```

Description

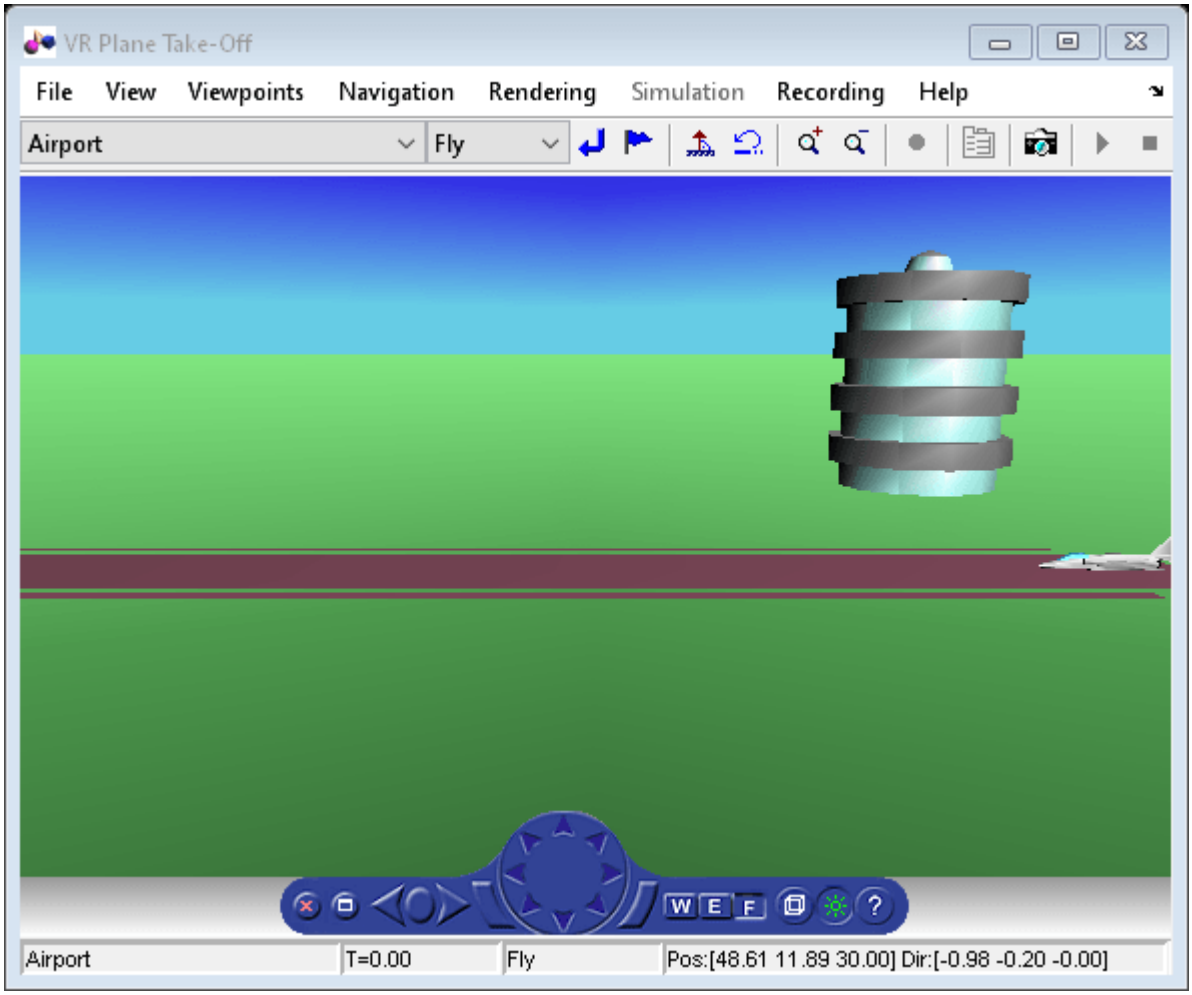
`[tstart,tstop] = findstartstoptimes(h,tsdata)` and `[tstart,stop] = h.findstartstoptimes(tsdata)` return the start and stop times of time series data `tsdata` for the virtual reality animation object `h`.

Examples

Find Start and Stop Time for Virtual Reality Animation Object

This example shows how to find the start and stop times of the time series data, `takeoffData`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
```




```
load takeoffData;
h.Nodes{7}.TimeSeriesSource = takeoffData;
h.Nodes{7}.TimeSeriesSourceType = 'StructureWithTime';
[tstart,stop]=h.Nodes{7}.findstartstoptimes
```

```
tstart = 0
```

```
stop = 15
```

Version History

Introduced in R2007b

See Also

`Aero.Node`

fixedWingAircraft

Create fixed-wing aircraft

Syntax

```
aircraft = fixedWingAircraft(name)
aircraft = fixedWingAircraft(name,referencearea,referencespan,
referencelength)
aircraft = fixedWingAircraft(name,referencearea,referencespan,
referencelength,degreesoffreedom)
aircraft = fixedWingAircraft( ___ ,Name=Value)
```

Description

`aircraft = fixedWingAircraft(name)` returns a fixed-wing aircraft object, `aircraft`, specified by the aircraft name.

`aircraft = fixedWingAircraft(name,referencearea,referencespan, referencelength)` returns a fixed-wing aircraft object, `aircraft`, specified by the aircraft name, `name`, reference area, `referencearea`, reference span, `referencespan`, and reference length, `referencelength`.

`aircraft = fixedWingAircraft(name,referencearea,referencespan, referencelength,degreesoffreedom)` returns a fixed-wing aircraft object created with the specified degrees of freedom, `degreesoffreedom`.

`aircraft = fixedWingAircraft( ___ ,Name=Value)` returns a fixed-wing aircraft object created with one or more name-value arguments.

Examples

Create FIxed-Wing Aircraft Object

Create a fixed-wing aircraft object.

```
aircraft = fixedWingAircraft("MyPlane",174,36,4.9)
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 174
       ReferenceSpan: 36
     ReferenceLength: 4.9000
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: 7.4483
          Properties: [1×1 Aero.Aircraft.Properties]
```

```
          UnitSystem: "Metric"
         AngleSystem: "Radians"
    TemperatureSystem: "Kelvin"
```

Create Fourth Order Point Mass Fixed-Wing Aircraft Object

Create a fourth order point-mass fixed-wing aircraft using positional arguments.

```
aircraft = fixedWingAircraft("MyPlane",174,36,4.9,"PM4")
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 174
       ReferenceSpan: 36
     ReferenceLength: 4.9000
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "PM4"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: 7.4483
          Properties: [1×1 Aero.Aircraft.Properties]
          UnitSystem: "Metric"
         AngleSystem: "Radians"
    TemperatureSystem: "Kelvin"
```

Create Fixed-Wing Aircraft Object Unit System

Create a fixed-wing aircraft by specifying the unit system as a name-value argument.

```
aircraft = fixedWingAircraft("MyPlane",174,36,4.9,"UnitSystem","English (kts)")
```

```
aircraft =

  FixedWing with properties:

       ReferenceArea: 174
       ReferenceSpan: 36
     ReferenceLength: 4.9000
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×0 Aero.FixedWing.Surface]
             Thrusts: [1×0 Aero.FixedWing.Thrust]
         AspectRatio: 7.4483
          Properties: [1×1 Aero.Aircraft.Properties]
          UnitSystem: "English (kts)"
         AngleSystem: "Radians"
    TemperatureSystem: "Kelvin"
```

Input Arguments

`name` — Fixed-wing aircraft name

scalar string

Fixed-wing aircraft name, specified as a scalar string.

Data Types: `string`

`referencearea` — Reference area

0 (default) | scalar numeric

Reference area, specified as a scalar numeric, commonly denoted as `'S'`, in these units.

Units	UnitSystem
meters squared (m^2)	`'Metric'`
feet squared (ft^2)	`'English (kts)'` or `'English (ft/s)'`

Tip This argument also exists as the name-value argument `ReferenceArea`. If you specify the `ReferenceArea` name-value argument, its value supersedes the `referencearea` positional argument.

Data Types: `double`

`referencespan` — Reference span

0 (default) | scalar numeric

Reference span, specified as a scalar numeric, commonly denoted as `'b'`, in units of:

Units	UnitSystem
meters squared (m)	`'Metric'`
feet squared (ft)	`'English (kts)'` or `'English (ft/s)'`

Tip This argument also exists as the name-value argument `ReferenceSpan`. If you specify the `ReferenceSpan` name-value argument, its value supersedes the `referencespan` positional argument.

Data Types: `double`

`referencelength` — Reference length

0 (default) | scalar numeric

Reference length, specified as a scalar numeric, commonly denoted as `'c'`, in these units:

Units	UnitSystem
meters squared (m)	`'Metric'`
feet squared (ft)	`'English (kts)'` or `'English (ft/s)'`

Tip This argument also exists as the name-value argument `ReferenceLength`. If you specify the `ReferenceLength` name-value argument, its value supersedes the `referencelength` positional argument.

Data Types: `double`

`degreesoffreedom` — Degrees of freedom
`'6DOF'` (default) | `'3DOF'` | `'PM4'` | `'PM6'`

Degrees of freedom, specified as a string or character vector.

Degrees of Freedom	Description
`'6DOF'`	Six degrees of freedom. Describes translational and rotational movement in 3-D space.
`'3DOF'`	Three degrees of freedom. Describes translational and rotational movement in 2-D space.
`'PM4'`	Fourth order point-mass. Describes translational movement in 2-D space.
`'PM6'`	Sixth order point-mass. Describes translational movement in 3-D space.

Tip This argument also exists as the name-value argument `DegreesOfFreedom`. If you specify the `DegreesOfFreedom` name-value argument, its value supersedes the `degreesoffreedom` positional argument.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"UnitSystem","English (kts)"`

`UnitSystem` — Unit system
`'Metric'` (default) | `'English (kts)'` | `'English (ft/s)'` | scalar | character vector

Unit system, specified as `'Metric'`, `'English (kts)'`, or `'English (ft/s)'`.

`AngleSystem` — Angle system
`'Radians'` (default) | `'Degrees'`

Angle system, specified as `'Radians'` or `'Degrees'`.

`TemperatureSystem` — Temperature system
`'Kelvin'` (default) | `'Celsius'` | `'Rankine'` | `'Fahrenheit'`

Temperature system, specified as `'Kelvin'`, `'Celsius'`, `'Rankine'`, or `'Fahrenheit'`.

`ReferenceArea` — Reference area
0 (default) | scalar numeric

Reference area, specified as a scalar numeric, commonly denoted as `'S'`, in these units.

Units	UnitSystem
meters squared (m^2)	`'Metric'`
feet squared (ft^2)	`'English (kts)'` or `'English (ft/s)'`

Tip This argument also exists as the `referencearea` positional argument. If you specify the `ReferenceArea` name-value argument, its value supersedes the `referencearea` positional argument.

Data Types: `double`

`ReferenceSpan` — Reference span
0 (default) | scalar numeric

Reference span, specified as a scalar numeric, commonly denoted as `'b'`, in units of:

Units	UnitSystem
meters squared (m)	`'Metric'`
feet squared (ft)	`'English (kts)'` or `'English (ft/s)'`

Tip This argument also exists as the `referencespan` positional argument. If you specify the `ReferenceSpan` name-value argument, its value supersedes the `referencespan` positional argument.

Data Types: `double`

`ReferenceLength` — Reference length
0 (default) | scalar numeric

Reference length, specified as a scalar numeric, commonly denoted as `'c'`, in units of:

Units	UnitSystem
meters squared (m)	`'Metric'`
feet squared (ft)	`'English (kts)'` or `'English (ft/s)'`

Tip This argument also exists as the `referencelength` positional argument. If you specify the `ReferenceLength` name-value argument, its value supersedes the `referencelength` positional argument.

Data Types: `double`

`Coefficients` — `Aero.FixedWing.Coefficients` class instance
scalar

`Aero.FixedWing.Coefficients` class instance, specified as a scalar that contains the coefficients defining the fixed-wing aircraft. This object ignores this property if no value is set.

`DegreesOfFreedom` — Degrees of freedom
`'6DOF'` (default) | `'3DOF'` | `'PM4'` | `'PM6'`

Degrees of freedom, specified as a string or character vector.

Degrees of Freedom	Description
`'6DOF'`	Six degrees of freedom. Describes translational and rotational movement in 3-D space.
`'3DOF'`	Three degrees of freedom. Describes translational and rotational movement in 2-D space.
`'PM4'`	Fourth order point-mass. Describes translational movement in 2-D space.
`'PM6'`	Sixth order point-mass. Describes translational movement in 3-D space.

Tip This argument also exists as the `degreesoffreedom` positional argument. If you specify the `DegreesOfFreedom` name-value argument, its value supersedes the `degreesoffreedom` positional argument.

`Surfaces` — `Aero.FixedWing.Surface` definitions
vector

`Aero.FixedWing.Surface` definitions, specified as a vector that contains the definitions of the surfaces on the fixed-wing aircraft. The object ignores this property if no value is set.

`Thrusts` — `Aero.FixedWing.Thrust` definitions
vector

`Aero.FixedWing.Thrust` definitions, specified as a vector that contains the definitions of the thrust on the fixed-wing aircraft. The object ignores this property if no value is set.

Data Types: `double`

Output Arguments

`aircraft` — Fixed-wing aircraft
scalar

Fixed-wing aircraft, returned as a scalar.

Version History

Introduced in R2021b

See Also

`aircraftEnvironment` | `aircraftProperties` | `fixedWingCoefficient` | `fixedWingState` | `fixedWingSurface` | `fixedWingThrust`

fixedWingCoefficient

Define numeric coefficients of fixed-wing aircraft

Syntax

```
coefficient = fixedWingCoefficient( )
coefficient = fixedWingCoefficient(statevariables)
coefficient = fixedWingCoefficient(statevariables,referenceframe)
coefficient = fixedWingCoefficient(statevariables,referenceframe,
multiplystatevariables)
coefficient = fixedWingCoefficient(statevariables,multiplystatevariables,
nondimensional)
coefficient = fixedWingCoefficient( ___ ,Name=Value)
```

Description

`coefficient = fixedWingCoefficient( )` returns a fixed-wing coefficient object with default properties.

`coefficient = fixedWingCoefficient(statevariables)` returns a fixed-wing coefficient object with the specified state variables, `statevariables`.

`coefficient = fixedWingCoefficient(statevariables,referenceframe)` returns a fixed-wing coefficient object with the specified reference frame, `referenceframe`.

`coefficient = fixedWingCoefficient(statevariables,referenceframe, multiplystatevariables)` returns a fixed-wing coefficient object with the specified multiply state variables switch, `multiplystatevariables`.

`coefficient = fixedWingCoefficient(statevariables,multiplystatevariables, nondimensional)` returns a fixed-wing coefficient object with the specified nondimensional switch, `nondimensional`.

`coefficient = fixedWingCoefficient( ___ ,Name=Value)` returns a fixed-wing coefficient object created with one or more Name=Value arguments.

Examples

Create Fixed-Wing Coefficient Object

Create a fixed-wing coefficient object.

```
coeffs = fixedWingCoefficient( )
```

```
coeffs =

  Coefficient with properties:
```

```
                    Table: [6×1 table]
                   Values: {6×1 cell}
           StateVariables: "Zero"
              StateOutput: [6×1 string]
           ReferenceFrame: "Wind"
    MultiplyStateVariables: on
           NonDimensional: on
               Properties: [1×1 Aero.Aircraft.Properties]
```

Create Fixed-Wing Coefficient Object with Specified State Variables

Create a fixed-wing coefficient object in the body frame nondimensional coefficients and nondefault state variables.

```
coeffs = fixedWingCoefficient(["U","Alpha"],"body","on","off")
```

```
coeffs =

  Coefficient with properties:

                    Table: [6×2 table]
                   Values: {6×2 cell}
           StateVariables: ["U"    "Alpha"]
              StateOutput: [6×1 string]
           ReferenceFrame: "Body"
    MultiplyStateVariables: on
           NonDimensional: off
               Properties: [1×1 Aero.Aircraft.Properties]
```

Input Arguments

`statevariables` — State variable names

`'Zero'` (default) | 1-by-*N* of strings

State variable names, specified as a 1-by-*N* vector of strings. Each entry in this property corresponds to a column in the `Values` property. Each entry in `StateVariables` must be a valid property in the `Aero.FixedWing.State` object. Adding a state variable adds a column of zeros to the end of the `Values` cell array.

Tip This argument also exists as the `Name=Value` argument `StateVariables`. If you specify the `StateVariables` `Name=Value` argument, its value supersedes the `statevariables` argument.

Data Types: `string`

`referenceframe` — Reference frame for coefficients

`'Wind'` (default) | `'Body'`

Reference frame for coefficients, specified as `'Wind'` or `'Body'` with these outputs.

Reference Frame	Coefficient Output
Wind	Forces: • drag (CD) • Y (CY) • lift (CL)
	Moments: • L (Cl) • M (Cm) • N (Cn)
Body	Forces: • X (CX) • Y (CY) • Z (CZ)
	Moments: • L (Cl) • M (Cm) • N (Cn)

Example of `'Wind'` table:

Coefficient	State
`CD`	*state*
`CY`	*state*
`CL`	*state*
`Cl`	*state*
`Cm`	*state*
`Cn`	*state*

Example of `'Body'` table:

Coefficient	State
`CX`	*state*
`CY`	*state*
`CZ`	*state*
`Cl`	*state*
`Cm`	*state*
`Cn`	*state*

Tip This argument also exists as the name-value argument `ReferenceFrame`. If you specify the `ReferenceFrame` name-value argument, its value supersedes the `referenceframe` argument.

Data Types: `string`

`multiplystatevariables` — Option to multiply coefficients by state variables
`'on'` (default) | `'off'`

Option to multiply coefficients by state variables when calculating forces and moments. To multiply coefficients by state variables, set this property to `'on'`. Otherwise, set this property to `'off'`.

Tip This argument also exists as the name-value argument `MultiplyStateVariables`. If you specify the `MultiplyStateVariables` name-value argument, its value supersedes the `multiplystatevariables` argument.

Data Types: `string`

`nondimensional` — Option to specify nondimensional coefficients
`'on'` (default) | `'off'`

Option to specify nondimensional coefficients, specified as `'on'` or `'off'`. To specify nondimensional coefficients, set this property to `'on'`. Otherwise, set this property to `'off'`.

Data Types: `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ReferenceFrame','Body'`

`Table` — Coefficient values
6-by-*N* table

Coefficient values, specified in a 6-by-*N* table. Each row in the table must be a member of and in the same order as the “`StateOutput” on page 4-0` property.

Setting the `Table` property also sets the contents of the `Values` property and `StateVariables` to the `Table` property variables. To have a `Simulink.LookupTable` object and a constant value in the same column, use the `setCoefficient` function or set the desired content of the `Values` property. Setting the `Table` property does not set the `ReferenceFrame`.

Note Tables must have a single data type per column. If there are both constant values and `Simulink.LookupTable` objects in a given column, the `Table` property automatically converts the constants to `Simulink.LookupTable` objects.

Data Types: `double`

`Values` — Coefficient values

6-by-*N* cell array

Coefficient values, specified as a 6-by-*N* cell array. Each entry in the cell array must be a single coefficient value corresponding to the `StateOutput` (row) and `StateVariable` (column) properties. Each coefficient value must be a scalar numeric value or a `Simulink.LookupTable` object. If a value is a `Simulink.LookupTable` object, the `FieldName` of each breakpoint must be a valid property of the `Aero.FixedWing.State` object.

Note `Values` do need to be a single data type per column.

Data Types: `double`

`StateVariables` — State variable names

`'Zero'` (default) | 1-by-*N* vector of strings

State variable names, specified as a 1-by-*N* vector of strings. Each entry in this property corresponds to a column in the `Values` property. Each entry in `StateVariables` must be a valid property in the `Aero.FixedWing.State` object. Adding a state variable adds a column of zeros to the end of the `Values` cell array.

Tip This argument also exists as the `statevariables` argument. If you specify the `StateVariables` name-value argument, its value supersedes the `statevariables` argument.

Data Types: `char` | `string`

`ReferenceFrame` — Reference frame for coefficients

`'Wind'` (default) | `'Body'`

Reference frame for coefficients, specified as `'Wind'` or `'Body` with these outputs.

Reference Frame	Coefficient Output
Wind	Forces: • drag (CD) • Y (CY) • lift (CL)
	Moments: • L (Cl) • M (Cm) • N (Cn)
Body	Forces: • X (CX) • Y (CY) • Z (CZ)

Reference Frame	Coefficient Output
	Moments: • L (Cl) • M (Cm) • N (Cn)

Example of `'Wind'` table:

Coefficient	State
`CD`	*state*
`CY`	*state*
`CL`	*state*
`Cl`	*state*
`Cm`	*state*
`Cn`	*state*

Example of `'Body'` table:

Coefficient	State
`CX`	*state*
`CY`	*state*
`CZ`	*state*
`Cl`	*state*
`Cm`	*state*
`Cn`	*state*

Tip This argument also exists as the `referenceframe` argument. If you specify the `ReferenceFrame` name-value argument, its value supersedes the `referenceframe` argument.

Data Types: `char` | `string`

`MultiplyStateVariables` — Option to multiply coefficients by state variables
`'on'` (default) | `'off'`

Option to multiply coefficients by state variables when calculating forces and moments, specified as `'on'` or `'off'`. To multiply coefficients by state variables, set this property to `'on'`. Otherwise, set this property to `'off'`.

Tip This argument also exists as the `multiplystatvariables` argument. If you specify the `MultiplyStateVariables` name-value argument, its value supersedes the `multiplystatvariables` argument.

`NonDimensional` — Option to specify nondimensional coefficients
`'on'` (default) | `'off'`

Option to specify nondimensional coefficients, specified as `'on'` or `'off'`. To specify nondimensional coefficients, set this property to `'on'`. Otherwise, set this property to `'off'`.

Tip This argument also exists as the `nondimensional` argument. If you specify the `NonDimensional` name-value argument, its value supersedes the `nondimensional` argument.

`Properties` — `Aero.Aircraft.Properties` object
scalar

`Aero.Aircraft.Properties` object, specified as a scalar.

Output Arguments

`coefficient` — Fixed-wing aircraft
scalar

Fixed-wing aircraft, returned as a scalar.

Version History

Introduced in R2021b

See Also

`aircraftEnvironment` | `aircraftProperties` | `fixedWingAircraft` | `fixedWingState` | `fixedWingSurface` | `fixedWingThrust` | `atmoscira` | `atmoscoesa`

fixedWingState

Define fixed-wing aircraft state

Syntax

```
state = fixedWingState(aircraft)
state = fixedWingState(aircraft,environment)
state = fixedWingState( ___ ,Name=Value)
```

Description

`state = fixedWingState(aircraft)` returns a fixed-wing `state` object created from a fixed-wing aircraft, `aircraft`, using a default environment.

`state = fixedWingState(aircraft,environment)` returns a fixed-wing `state` object using a specified environment, `environment`.

`state = fixedWingState( ___ ,Name=Value)` returns a fixed-wing `state` object with an environment defined by `Name=Value` arguments.

Examples

Create Fixed-Wing Aircraft State Object and Default Environment

Create a fixed-wing aircraft state object from a fixed-wing aircraft object.

```
aircraft = astC182();
state = fixedWingState(aircraft)
```

```
state =

  State with properties:

                  Alpha: 0
                   Beta: 0
               AlphaDot: 0
                BetaDot: 0
                   Mass: 0
                Inertia: [3×3 table]
        CenterOfGravity: [0 0 0]
       CenterOfPressure: [0 0 0]
            AltitudeMSL: 0
           GroundHeight: 0
                     XN: 0
                     XE: 0
                     XD: 0
                      U: 50
                      V: 0
                      W: 0
                    Phi: 0
                  Theta: 0
```

```
                    Psi: 0
                      P: 0
                      Q: 0
                      R: 0
                 Weight: 0
             AltitudeAGL: 0
                Airspeed: 50
             GroundSpeed: 50
              MachNumber: 0.0448
            BodyVelocity: [50 0 0]
          GroundVelocity: [50 0 0]
                      Ur: 50
                      Vr: 0
                      Wr: 0
       FlightPathAngle: 0
           CourseAngle: 0
  InertialToBodyMatrix: [3×3 double]
  BodyToInertialMatrix: [3×3 double]
      BodyToWindMatrix: [3×3 double]
      WindToBodyMatrix: [3×3 double]
       DynamicPressure: 2.9711
           Environment: [1×1 Aero.Aircraft.Environment]
         ControlStates: [1×4 Aero.Aircraft.ControlState]
      OutOfRangeAction: "Limit"
      DiagnosticAction: "Warning"
            Properties: [1×1 Aero.Aircraft.Properties]
            UnitSystem: "English (ft/s)"
           AngleSystem: "Radians"
     TemperatureSystem: "Fahrenheit"
```

Create Fixed-Wing Aircraft State Object from Fixed-Wing Aircraft Object and Mass

Create a fixed-wing aircraft state object from a fixed-wing aircraft object and specify the mass using positional arguments.

```
aircraft = astC182();
state = fixedWingState(aircraft,"Mass",500)
```

```
state =

  State with properties:

                  Alpha: 0
                   Beta: 0
               AlphaDot: 0
                BetaDot: 0
                   Mass: 500
                Inertia: [3×3 table]
         CenterOfGravity: [0 0 0]
        CenterOfPressure: [0 0 0]
             AltitudeMSL: 0
            GroundHeight: 0
                      XN: 0
                      XE: 0
                      XD: 0
                       U: 50
```

```
                     V: 0
                     W: 0
                   Phi: 0
                 Theta: 0
                   Psi: 0
                     P: 0
                     Q: 0
                     R: 0
                Weight: 1.6093e+04
            AltitudeAGL: 0
               Airspeed: 50
            GroundSpeed: 50
             MachNumber: 0.0448
           BodyVelocity: [50 0 0]
         GroundVelocity: [50 0 0]
                    Ur: 50
                    Vr: 0
                    Wr: 0
       FlightPathAngle: 0
           CourseAngle: 0
    InertialToBodyMatrix: [3×3 double]
    BodyToInertialMatrix: [3×3 double]
        BodyToWindMatrix: [3×3 double]
        WindToBodyMatrix: [3×3 double]
         DynamicPressure: 2.9711
             Environment: [1×1 Aero.Aircraft.Environment]
           ControlStates: [1×4 Aero.Aircraft.ControlState]
        OutOfRangeAction: "Limit"
        DiagnosticAction: "Warning"
              Properties: [1×1 Aero.Aircraft.Properties]
              UnitSystem: "English (ft/s)"
             AngleSystem: "Radians"
      TemperatureSystem: "Fahrenheit"
```

Create Fixed-Wing Aircraft State Object and Custom Environment

Create a fixed-wing aircraft state object from a fixed-wing aircraft object using a custom environment and `Name=Value` arguments.

```
aircraft = astC182();
state = fixedWingState(aircraft,aircraftEnvironment(aircraft,"COESA",1000))
```

```
state =

  State with properties:

                   Alpha: 0
                    Beta: 0
                AlphaDot: 0
                 BetaDot: 0
                    Mass: 0
                 Inertia: [3×3 table]
          CenterOfGravity: [0 0 0]
         CenterOfPressure: [0 0 0]
              AltitudeMSL: 0
             GroundHeight: 0
                       XN: 0
```

```
                    XE: 0
                    XD: 0
                     U: 50
                     V: 0
                     W: 0
                   Phi: 0
                 Theta: 0
                   Psi: 0
                     P: 0
                     Q: 0
                     R: 0
                Weight: 0
            AltitudeAGL: 0
               Airspeed: 50
            GroundSpeed: 50
             MachNumber: 0.0449
           BodyVelocity: [50 0 0]
         GroundVelocity: [50 0 0]
                    Ur: 50
                    Vr: 0
                    Wr: 0
       FlightPathAngle: 0
           CourseAngle: 0
  InertialToBodyMatrix: [3×3 double]
  BodyToInertialMatrix: [3×3 double]
      BodyToWindMatrix: [3×3 double]
      WindToBodyMatrix: [3×3 double]
       DynamicPressure: 2.8851
           Environment: [1×1 Aero.Aircraft.Environment]
         ControlStates: [1×4 Aero.Aircraft.ControlState]
      OutOfRangeAction: "Limit"
      DiagnosticAction: "Warning"
            Properties: [1×1 Aero.Aircraft.Properties]
            UnitSystem: "English (ft/s)"
           AngleSystem: "Radians"
     TemperatureSystem: "Fahrenheit"
```

Input Arguments

`aircraft` — Fixed-wing aircraft object

scalar

Fixed-wing aircraft object, specified as a scalar.

`environment` — Fixed-wing aircraft environment name

`aircraftEnvironment(aircraft, "ISA", 0)` (default) | scalar string

Fixed-wing aircraft environment name, specified as a scalar string.

Tip This argument also exists as the name-value argument `Environment`. If you specify the `Environment` name-value argument, its value supersedes the `environment` positional argument.

Data Types: `string`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"Mass",500`

`UnitSystem` — Unit system

`'Metric'` (default) | `'English (kts)'` | `'English (ft/s)'`

Unit system, specified as `'Metric'`, `'English (kts)'`, `'English (ft/s)'`.

`AngleSystem` — Angle system

`'Radians'` (default) | `'Degrees'`

Angle system, specified as `'Radians'` or `'Degrees'`.

`TemperatureSystem` — Temperature system

`'Kelvin'` (default) | `'Celsius'` | `'Rankine'` | `'Fahrenheit'`

Temperature system, specified as `'Kelvin'`, `'Celsius'`, `'Rankine'`, or `'Fahrenheit'`.

`Mass` — Fixed-wing aircraft mass

`0` (default) | scalar numeric

Fixed-wing aircraft mass, specified as a scalar numeric in these units.

Unit	Unit System
newtons (N)	`'Metric'`
slugs (slug)	`'English (kts)'` and `'English (ft/s)'`

Data Types: `double`

`Inertia` — Inertial matrix of aircraft

`3-by-3` table of numeric values (default) | scalar numeric

Inertial matrix of aircraft, specified as a 3-by-3 table of numeric values specifying the body in this matrix form.

	X	Y	Z
X	Ixx	Ixy	Ixz
Y	Iyx	Iyy	Iyz
Z	Izx	Izy	Izz

The matrix has these units.

Unit	Unit System
kilogram meters squared (kg m^2)	`'Metric'`
slug feet squared (slug ft^2)	`'English (kts)'` and `'English (ft/s)'`

Data Types: `double`

`CenterOfGravity` — Location of center of gravity
[0, 0, 0] (default) | three-element vector

Location of center of gravity on fixed-wing aircraft in body frame, specified as a three-element vector in these units.

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Data Types: `double`

`CenterOfPressure` — Location of center of pressure
[0, 0, 0] (default) | three-element vector

Location of center of pressure on fixed-wing aircraft in body frame, specified as a three-element vector in these units.

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Data Types: `double`

`AltitudeMSL` — Altitude above sea level
0 (default) | scalar numeric

Altitude above sea level, specified as a scalar numeric in these units.

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Data Types: `double`

`GroundHeight` — Ground height above sea level
0 (default) | scalar numeric

Ground height above sea level, specified as a scalar numeric in these units.

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Data Types: `double`

`XN` — North position of fixed-wing aircraft
0 (default) | scalar numeric

North position of fixed-wing aircraft, specified as a scalar numeric in these units.

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Data Types: `double`

`XE` — East position of fixed-wing aircraft
0 (default) | scalar numeric

East position of fixed-wing aircraft, specified as a scalar numeric in these units.

Unit	Unit System
Meters (m)	`'Metric'`
Feet (ft)	`'English (kts)'` and `'English (ft/s)'`

Data Types: `double`

`U` — Forward component of ground velocity
50 (default) | scalar numeric

Forward component of ground velocity, specified as a scalar numeric in these units.

Unit	Unit System
Meters per second (m/s)	`'Metric'`
Feet per second (ft/s)	`'English (kts)'`
Knots (kts)	`'English (ft/s)'`

Data Types: `double`

`V` — Side component of ground velocity
0 (default) | scalar numeric

Side component of ground velocity, specified as a scalar numeric in these units.

Unit	Unit System
Meters per second (m/s)	`'Metric'`
Feet per second (ft/s)	`'English (kts)'`
Knots (kts)	`'English (ft/s)'`

Data Types: `double`

`W` — Downward component of ground velocity
0 (default) | scalar numeric

Downward component of ground velocity, specified as a scalar numeric in these units.

Unit	Unit System
Meters per second (m/s)	`'Metric'`
Feet per second (ft/s)	`'English (kts)'`

Unit	Unit System
Knots (kts)	`'English (ft/s)'`

Data Types: `double`

`Phi` — Euler roll angle
0 (default) | scalar numeric

Euler roll angle, specified as a scalar numeric in units of radians or degrees, depending on the `AngleSystem` property.

Data Types: `double`

`Theta` — Euler pitch angle
0 (default) | scalar numeric

Euler pitch angle, specified as a scalar numeric in units of radians or degrees, depending on the `AngleSystem` property.

Data Types: `double`

`Psi` — Euler yaw angle
0 (default) | scalar numeric

Euler yaw angle, specified as a scalar numeric in units of radians or degrees, depending on the `AngleSystem` property.

Data Types: `double`

`P` — Body roll rate
0 (default) | scalar numeric

Body roll rate, specified as a scalar numeric in units of radians per second or degrees per second, depending on the `AngleSystem` property.

Data Types: `double`

`Q` — Body pitch rate
0 (default) | scalar numeric

Body pitch rate, specified as a scalar numeric in units of radians per second or degrees per second, depending on the `AngleSystem` property.

Data Types: `double`

`R` — Body yaw rate
0 (default) | scalar numeric

Body yaw rate, specified as a scalar numeric in units of radians per second or degrees per second, depending on the `AngleSystem` property.

Data Types: `double`

`AlphaDot` — Angle of attack rate on fixed-wing aircraft
0 (default) | scalar numeric

Angle of attack rate on fixed-wing aircraft, specified as a scalar numeric in units of radians per second or degrees per second, depending on the `AngleSystem` property.

Data Types: `double`

`BetaDot` — Angle of sideslip rate on fixed-wing aircraft
0 (default) | scalar numeric

Angle of sideslip rate on the fixed-wing aircraft, specified as a scalar numeric in units of radians per second or degrees per second, depending on the `AngleSystem` property.

Data Types: `double`

`ControlStates` — Current control state values
vector

Current control state values, specified as a vector.

- To set up control states, use `setupControlStates`.
- To set the control state positions, use `setState`.
- To get the control state positions, use `getState`.

Data Types: `double`

`Environment` — Definition of current environment
scalar

Definition of current environment, contained in an `Aero.Aircraft.Environment` object, specified as a scalar.

Tip This argument also exists as the `environment` positional argument. If you specify the `Environment` name-value argument, its value supersedes the `environment` positional argument.

Output Arguments

`state` — `Aero.FixedWing.State` objects
matrix

`Aero.FixedWing.State` objects, returned as a matrix the same size as `environment`.

Version History

Introduced in R2021b

See Also

`aircraftEnvironment` | `aircraftProperties` | `fixedWingAircraft` | `fixedWingCoefficient` | `fixedWingSurface` | `fixedWingThrust` | `atmoscira` | `atmoscoesa`

fixedWingSurface

Define aerodynamic or control surface on fixed-wing aircraft

Syntax

```
surface = fixedWingSurface(name)
surface = fixedWingSurface(name,controllable)
surface = fixedWingSurface(name,controllable,symmetry)
surface = fixedWingSurface(name,controllable,symmetry,bounds)
surface = fixedWingSurface(Name=Value)
```

Description

`surface = fixedWingSurface(name)` returns a fixed-wing aerodynamic surface object with a specified component, `name`.

`surface = fixedWingSurface(name,controllable)` returns a fixed-wing surface object specifying the controllability, `controllable`, of the surface.

`surface = fixedWingSurface(name,controllable,symmetry)` returns a fixed-wing surface object specifying the symmetry,`symmetry`, of the surface.

`surface = fixedWingSurface(name,controllable,symmetry,bounds)` returns a fixed-wing surface object specifying the bounds, `bounds`, of the surface.

`surface = fixedWingSurface(Name=Value)` returns a fixed-wing surface object with one or more `Name=Value` arguments.

Examples

Create Fixed-Wing Surface Object

Create a fixed-wing surface object named `MySurface`.

```
surface = fixedWingSurface("MySurface")
```

```
surface =

  Surface with properties:

           Surfaces: [1×0 Aero.FixedWing.Surface]
       Coefficients: [1×1 Aero.FixedWing.Coefficient]
       MaximumValue: Inf
       MinimumValue: -Inf
       Controllable: off
           Symmetry: "Symmetric"
    ControlVariables: [0×0 string]
         Properties: [1×1 Aero.Aircraft.Properties]
```

Create Fixed-Wing Asymmetric Control Surface

Create a fixed-wing asymmetric control surface named `MyCtrl` using arguments.

```
ctrlsurface = fixedWingSurface("MyCtrl","on","asymmetric")
```

```
ctrlsurface =

  Surface with properties:

           Surfaces: [1×0 Aero.FixedWing.Surface]
       Coefficients: [1×1 Aero.FixedWing.Coefficient]
       MaximumValue: Inf
       MinimumValue: -Inf
       Controllable: on
           Symmetry: "Asymmetric"
    ControlVariables: ["MyCtrl_1"    "MyCtrl_2"]
         Properties: [1×1 Aero.Aircraft.Properties]
```

Create Fixed-Wing Symmetric Control Surface with Specified Bounds

Create a fixed-wing symmetric control surface named `MyCtrl` with specified bounds and add it to an aerodynamic surface using a `Name=Value` argument.

```
ctrlsurface = fixedWingSurface("MyCtrl","on","symmetric",[-20, 20]);
surface = fixedWingSurface("MySurface","Surfaces",ctrlsurface)
```

```
surface =

  Surface with properties:

           Surfaces: [1×1 Aero.FixedWing.Surface]
       Coefficients: [1×1 Aero.FixedWing.Coefficient]
       MaximumValue: Inf
       MinimumValue: -Inf
       Controllable: off
           Symmetry: "Symmetric"
    ControlVariables: [0×0 string]
         Properties: [1×1 Aero.Aircraft.Properties]
```

Input Arguments

`name` — Fixed-wing aircraft surface name

scalar string

Fixed-wing aircraft surface name, specified as a scalar string.

Data Types: `string`

`controllable` — Option to control surface

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

Controllable control surface, specified as `'on'` or `'off'`. To control the control surface, set this property to `'on'`. Otherwise, set this property to `'off'`.

Data Types: `string`

`symmetry` — Symmetry of control surface
`Symmetric` (default) | `Asymmetric`

Symmetry of the control surface, specified as `Symmetric` or `Asymmetric`.

The `Asymmetric` option creates two control variables, denoted by the name on the properties and appended by `_1` and `_2`. These control variables can be independently controlled but also produce an effective control variable specified by the name on the properties. This equation defines the control variable:

name = (*name*_1-*name*_2)/2.

You cannot set this effective control variable.

Data Types: `string`

`bounds` — Lower and upper bounds
`[-inf,inf]` (default) | two-element numeric vector

Lower and upper bounds of a controllable surface, specified as a two-element numeric vector.

Data Types: `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"Surfaces",ctrlsurface`

`Surfaces` — `Aero.FixedWing.Surface` objects
vector

`Aero.FixedWing.Surface` objects providing nested control surfaces, specified as a vector.

`Coefficients` — `Aero.FixedWing.Coefficients` objects
scalar

`Aero.FixedWing.Coefficients` objects that define control surface, specified as a scalar.

`MaximumValue` — Maximum value of control surfaces
infinity (default) | scalar numeric

Maximum value of control surfaces, specified as a scalar numeric.

Dependencies

If `Symmetry` is set to `Asymmetric`, then this value applies to both control variables.

`MinimumValue` — Minimum value of control surface
negative infinity (default) | scalar numeric

Minimum value of control surface, specified as a scalar numeric.

Dependencies

If `Symmetry` is set to `Asymmetric`, then this value applies to both control variables.

`Controllable` — Controllable control surface
`'off'` (default) | `'on'`

Controllable control surface specified as `'on'` or `'off'`. To control the control surface, set this property to `'on'`. Otherwise, set this property to `'off'`.

`Symmetry` — Symmetry of control surface
`Symmetric` (default) | `Asymmetric`

Symmetry of the control surface, specified as `Symmetric` or `Asymmetric`.

The `Asymmetric` option creates two control variables, denoted by the name on the properties and appended by `_1` and `_2`. These control variables can be independently controlled but also produce an effective control variable specified by the name on the properties. This equation defines the control variable:

name = (*name*_1-*name*_2)/2.

You cannot set this effective control variable.

`Properties` — `Aero.Aircraft.Properties` object
scalar

`Aero.Aircraft.Properties` object, specified as a scalar.

Output Arguments

`surface` — `Aero.FixedWing.Surface` object
scalar

`Aero.FixedWing.State` object, returned as a scalar.

Version History

Introduced in R2021b

See Also

`aircraftEnvironment` | `aircraftProperties` | `fixedWingAircraft` | `fixedWingCoefficient` | `fixedWingState` | `fixedWingThrust`

fixedWingThrust

Define thrust vector on fixed-wing aircraft

Syntax

```
thrust = fixedWingThrust(name)
thrust = fixedWingThrust(name,controllable)
thrust = fixedWingThrust(name,controllable,symmetry)
thrust = fixedWingThrust(name,controllable,symmetry,bounds)
thrust = fixedWingThrust(Name=Value)
```

Description

`thrust = fixedWingThrust(name)` returns a fixed-wing thrust object with a specified component, `name`.

`thrust = fixedWingThrust(name,controllable)` returns a fixed-wing thrust object specifying the controllability, `controllable`, of the thrust.

`thrust = fixedWingThrust(name,controllable,symmetry)` returns a fixed-wing thrust object specifying the symmetry, `symmetry`, of the thrust.

`thrust = fixedWingThrust(name,controllable,symmetry,bounds)` returns a fixed-wing thrust object specifying the bounds, `bounds`, of the thrust.

`thrust = fixedWingThrust(Name=Value)` returns a fixed-wing thrust object with one or more `Name=Value` arguments.

Examples

Create Fixed-Wing Thrust Object

Create a fixed-wing thrust object named `MyThrust`.

```
thrust = fixedWingThrust("MyThrust")
```

```
thrust =

  Thrust with properties:

       Coefficients: [1×1 Aero.FixedWing.Coefficient]
       MaximumValue: 1
       MinimumValue: 0
       Controllable: on
           Symmetry: "Symmetric"
    ControlVariables: "MyThrust"
         Properties: [1×1 Aero.Aircraft.Properties]
```

Create Asymmetric Thrust Object

Create a asymmetric fixed-wing thrust object named `MyThrust` using arguments.

```
thrust = fixedWingThrust("MyThrust","on","asymmetric")
```

```
thrust =

  Thrust with properties:

        Coefficients: [1×1 Aero.FixedWing.Coefficient]
        MaximumValue: 1
        MinimumValue: 0
        Controllable: on
            Symmetry: "Asymmetric"
    ControlVariables: ["MyThrust_1"    "MyThrust_2"]
          Properties: [1×1 Aero.Aircraft.Properties]
```

Input Arguments

`name` — Fixed-wing aircraft thrust name

scalar string

Fixed-wing aircraft thrust name, specified as a scalar string.

Data Types: `char` | `string`

`controllable` — Controllable thrust value

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

Controllable thrust value, specified as `'on'` or `'off'`. To control the control thrust, set this property to `'on'`. Otherwise, set this property to `'off'`.

Data Types: `string`

`symmetry` — Symmetry of thrust control

`Symmetric` (default) | `Asymmetric`

Symmetry of thrust control, specified as `Symmetric` or `Asymmetric`.

The `Asymmetric` option creates two control variables, denoted by the name on the properties and appended by `_1` and `_2`. These control variables can be independently controlled, but also produce an effective control variable specified by the name on the properties. This equation defines the control variable:

name = (*name*_1-*name*_2)/2.

You cannot set this effective control variable.

Data Types: `string`

`bounds` — Lower and upper bounds

`[-inf,inf]` (default) | two-element numeric vector

Lower and upper bounds of controllable thrust, specified as a two-element numeric vector.

Data Types: `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MaximumValue','500'`

`Coefficients` — `Aero.FixedWing.Coefficients` object
scalar

`Aero.FixedWing.Coefficients` object, specified as a scalar, that defines the thrust vector.

`MaximumValue` — Maximum thrust value
`1` (default) | scalar | scalar numeric

Maximum thrust value, specified as a scalar numeric.

Dependencies

If `Symmetry` is set to `Asymmetric`, then this value applies to both control variables.

Data Types: `double`

`MinimumValue` — Minimum thrust value
`0` (default) | scalar numeric

Minimum thrust value, specified as a scalar numeric.

Dependencies

If `Symmetry` is set to `Asymmetric`, then this value applies to both control variables.

Data Types: `double`

`Controllable` — Controllable thrust value
`on` (default) | `off`

To control the thrust value, set this property to `on`. Otherwise, set this property to `off`.

Data Types: `double`

`Symmetry` — Symmetry of thrust control
`Symmetric` (default) | `Asymmetric`

Symmetry of the thrust control, specified as `Symmetric` or `Asymmetric`.

The `Asymmetric` option creates two control variables, denoted by the name on the properties and appended by `_1` and `_2`. These control variables can be independently controlled, but also produce an effective control variable specified by the name on the properties. This equation defines the control variable:

name = (*name*_1-*name*_2)/2.

You cannot set this effective control variable.

Data Types: `char` | `string`

`Properties` — `Aero.Aircraft.Properties` object
scalar

`Aero.Aircraft.Properties` object, specified as a scalar.

Output Arguments

`thrust` — `Aero.FixedWing.Thrust` object
scalar

`Aero.FixedWing.Thrust` object, returned as a scalar.

Version History

Introduced in R2021b

See Also

`aircraftEnvironment` | `aircraftProperties` | `fixedWingAircraft` | `fixedWingCoefficient` | `fixedWingState` | `fixedWingSurface`

flat2lla

Convert from flat Earth position to array of geodetic coordinates

Syntax

```
lla = flat2lla(flatearth_pos,llo,psio,href)
lla = flat2lla( ___ , ellipsoidModel)
lla = flat2lla( ___ ,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 you define with `llo`, `psio`, and `href`.

`lla = flat2lla( ___ , ellipsoidModel)` estimates the coordinates for a specific ellipsoid planet.

`lla = flat2lla( ___ ,flattening,equatorialRadius)` estimates the coordinates for a custom ellipsoid planet defined by `flattening` and `equatorialRadius`.

Examples

Estimate Latitude, Longitude, and Altitude at Single Coordinate

Estimate latitude, longitude, and altitude at a single 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 with WGS84 Ellipsoid Model

Estimate latitudes, longitudes, and altitudes at multiple coordinates with the WGS84 ellipsoid model:

```
lla = flat2lla( [ 4731 4511 120; 0 5074 4498 ], [0 45], 5, -100, 'WGS84' )
```

```
lla =
   1.0e+03 *

    0.0000    0.0450   -0.0200
   -0.0000    0.0450   -4.3980
```

Estimate Latitudes, Longitudes, and Altitudes at Multiple Coordinates with Custom Ellipsoid Model:

Estimate latitudes, longitudes, and altitudes at multiple coordinates with 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+03 *

    0.0001    0.0451   -0.0200
   -0.0000    0.0451   -4.3980
```

Input Arguments

`flatearth_pos` — Flat Earth position coordinates

m-by-3 array

Flat Earth position coordinates, specified as *m*-by-3 array, in meters.

Data Types: `double`

`llo` — Latitude and longitude of reference location

2-element vector

Latitude and longitude of reference location , specified as a 2-element vector in degrees, for the origin of the estimation and the origin of the flat Earth coordinate system.

Data Types: `double`

`psio` — Angular direction of flat Earth

scalar

Angular direction of flat Earth *x*-axis, specified as a scalar. The angular direction is the degrees clockwise from the north, which is the angle in degrees used for converting flat Earth *x* and *y* coordinates to the north and east coordinates.

Data Types: `double`

`href` — Reference height

scalar

Reference height from the surface of the Earth to the flat Earth frame with respect to the flat Earth frame, specified as a scalar, in meters.

Data Types: `double`

`ellipsoidModel` — Ellipsoid planet model

`'WGS84'` (default)

Ellipsoid planet model. `'WGS84'` is the only option.

Data Types: `char` | `string`

`flattening` — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Data Types: `double`

`equatorialRadius` — Planetary equatorial radius
scalar

Planetary equatorial radius, specified as a scalar, in meters.

Data Types: `double`

Output Arguments

`lla` — Geodetic coordinates
m-by-3 array

Geodetic coordinates (latitude, longitude, and altitude), returned as an *m*-by-3 array, in [degrees, degrees, meters].

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 the curvature in the prime vertical (R_N) and the radius of the curvature in the meridian (R_M). (R_N) and (R_M) are defined by the following relationships:

$$R_N = \frac{R}{\sqrt{1 - (2f - f^2)\sin^2\mu_0}},$$

and

$$R_M = R_N \frac{1 - (2f - f^2)}{1 - (2f - f^2)\sin^2\mu_0},$$

where (R) is the equatorial radius of the planet and$\left(\overline{f}\right)$ 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

$$d\mu = \text{atan}\left(\frac{1}{R_M}\right) dN$$

$$d\imath = \text{atan}\left(\frac{1}{R_N \cos\mu}\right) dE$$

and

$$d\iota = \text{atan}\left(\frac{1}{R_N \cos\mu}\right) dE\,.$$

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}).

$$h = -p_z - h_{ref}$$

Version History

Introduced in R2011a

References

[1] Etkin, B., *Dynamics of Atmospheric Flight*. New York: John Wiley & Sons, 1972.

[2] Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, 2nd ed. New York: John Wiley & Sons, 2003.

See Also

`lla2flat`

flowfanno

Fanno line flow relations

Syntax

```
[mach,T,P,rho,velocity,P0,fanno] = flowfanno(gamma,fanno_flow)

[mach,T,P,rho,velocity,P0,fanno] = flowfanno( ___ ,mtype)
```

Description

Default Input Mode

`[mach,T,P,rho,velocity,P0,fanno] = flowfanno(gamma,fanno_flow)` returns an array for each Fanno line flow relation. This function calculates the arrays for a given set of specific heat ratios (`gamma`) for the Mach input mode.

Specify Input Mode

`[mach,T,P,rho,velocity,P0,fanno] = flowfanno( ___ ,mtype)` uses any one of the Fanno flow types `mtype`. Specify `mtype` types after all other input arguments.

Examples

Calculate Fanno Line Flow Relations for Subsonic Fanno Parameter

Calculate the Fanno line flow relations for air (`gamma` = 1.4) for subsonic Fanno parameter 1.2. This example 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')
```

```
mach =
    0.4849

T =
    1.1461

P =
    2.2080

rho =
    1.9265

velocity =
    0.5191

P0 =
    1.3699
```

```
fanno =
    1.2000
```

Calculate Fanno Line Flow Relations for Gases with Specific Heat Ratios

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. This example 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)
```

```
mach =

    0.5000    0.5000    0.5000    0.5000


T =

    1.1084    1.1188    1.1429    1.2318


P =

    2.1056    2.1155    2.1381    2.2198


rho =

    1.8997    1.8908    1.8708    1.8020


velocity =

    0.5264    0.5289    0.5345    0.5549


P0 =

    1.3479    1.3454    1.3398    1.3201


fanno =

    1.1724    1.1397    1.0691    0.8549
```

Calculate Fanno Line Flow Relations for Specific Heat Ratio and range of Temperature Ratios

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. This example 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')
```

```
mach =
    0.6742         0

T =
    1.1000    1.2000

P =
    1.5556       Inf

rho =
    1.4142       Inf

velocity =
    0.7071         0

P0 =
    1.1144       Inf

fanno =
    0.2630       Inf
```

Input Arguments

gamma — Specific heat ratios

scalar | array | real numbers greater than 1

Specific heat ratios, specified as an array or scalar of *N* specific heat ratios.

Dependencies

`gamma` must be a real, finite scalar greater than 1 for these input modes:

- Subsonic total pressure ratio
- Supersonic total pressure ratio
- Subsonic Fanno parameter
- Supersonic Fanno parameter

Data Types: `double`

fanno_flow — One Fanno flow

array of real numerical values

One Fanno flow, specified as an array of real numerical values. This argument can be one of these types.

Fanno Flow Type	Description
Mach numbers	Mach numbers, specified as a scalar or 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.

Fanno Flow Type	Description
Temperature ratios	Temperature ratios on page 4-515, specified as an array or scalar 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'`.
Pressure ratios	Pressure ratios on page 4-515, specified as an array or scalar 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'`.
Density ratios	Density ratios on page 4-515, specified as an array or scalar 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'`.
Velocity ratios	Velocity ratios on page 4-515, specified as an array or scalar of *N* 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'`.
Total pressure ratio	Total pressure ratio on page 4-516, specified as a scalar greater than or equal to 1. Use `flow_fanno` with `mtype` values `'totalp'` and `'totalpsup'`.

Fanno Flow Type	Description
Fanno parameter scalar	"Fanno Parameter" on page 4-516, specified as a scalar. 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'`.

Data Types: `double`

`mtype` — Input mode of Fanno flow
`'mach'` (default) | `'temp'` | `'pres'` | `'dens'` | `'velo'` | `'totalpsub'` | `'totalpsup'` | `'fannosub'` | `'fannosup'`

Input mode of Fanno flow, specified as one of these values.

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
`'fannosup'`	Supersonic Fanno parameter

Data Types: `double`

Output Arguments

All outputs are the same size as the array inputs. If there are no array inputs, all outputs are scalars.

`mach` — Mach numbers
array

Mach numbers, returned as an array.

`T` — Temperature ratios
array

Temperature ratios on page 4-515, returned as an array.

`P` — Pressure ratios
array

Pressure ratios on page 4-515, returned as an array.

`rho` — Density ratios
array

Density ratios on page 4-515, returned as an array.

`velocity` — Velocity ratios
array

Velocity ratios on page 4-515, returned as an array.

`P0` — Stagnation pressure ratios
array

Stagnation (total) pressure ratios on page 4-516, returned as an array.

`fanno` — Fanno parameters
array

Fanno parameters on page 4-516, returned as an array.

Limitations

- This function assumes that variables vary only in one dimension. 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. For thermally perfect gas correction factors, see [2]. If the temperature is so high that molecules dissociate and ionize (static temperature 5000 K for air), you cannot assume a perfect gas.

More About

Pressure Ratio

Calculated as local static pressure over the reference static pressure for sonic flow.

Temperature Ratio

Calculated as local static temperature over the reference static temperature for sonic flow.

Density Ratio

Calculated as local density over the reference density for sonic flow.

Velocity Ratio

Calculated as local velocity over the reference velocity for sonic flow.

Total Pressure Ratio

Calculated as local total pressure over the reference total pressure for sonic flow.

Fanno Parameter

This function uses Fanno variables given by the equation: $F = f^*L/D$, where:

- F is the Fanno parameter.
- 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.

Version History

Introduced in R2010a

References

[1] James, John E. A. *Gas Dynamics*. 2nd ed. Boston: Allyn and Bacon 1984.

[2] Ames Research Staff. *NACA Technical Report 1135*. Moffett Field, CA: National Advisory Committee on Aeronautics, 1953. 667–671.

See Also

`flowisentropic` | `flownormalshock` | `flowprandtlmeyer` | `flowrayleigh`

flowisentropic

Isentropic flow ratios

Syntax

```
[mach,T,P,rho,area] = flowisentropic(gamma,flow)

[mach,T,P,rho,area] = flowisentropic( ___ ,mtype)
```

Description

Default Input Mode

`[mach,T,P,rho,area] = flowisentropic(gamma,flow)` returns an array that 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`) for the Mach input mode.

Specify Input Mode

`[mach,T,P,rho,area] = flowisentropic( ___ ,mtype)` uses any one of the isentropic flow types `mtype`. Specify `mtype` types after all other input arguments.

Examples

Calculate Isentropic Flow Relations for Gases with Specific Heat Ratios

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 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)

mach =
    0.5000    0.5000    0.5000    0.5000

T =
    0.9639    0.9604    0.9524    0.9227

P =
    0.8525    0.8497    0.8430    0.8183

rho =
    0.8845    0.8847    0.8852    0.8869

area =
    1.3479    1.3454    1.3398    1.3201
```

Calculate Isentropic Flow Relations for Air

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')
```

```
mach =
    0.5500

T =
    0.9430

P =
    0.8142

rho =
    0.8634

area =
    1.2550
```

Calculate Isentropic Flow Relations for Gases with Specific Heat Ratio and Density Ratio

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')
```

```
mach =
    2.3724    0.4639

T =
    0.5422    0.9587

P =
    0.0705    0.8629

rho =
    0.1300    0.9000

area =
    2.5769    1.4155
```

Calculate Isentropic Flow Relations for Specific Heat Ratio

Calculate the isentropic flow relations for a specific heat ratio of 1.4, and 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')
```

```
mach =
    2.7386
    2.2361
    1.8257
    1.4639

T =
    0.4000
    0.5000
    0.6000
    0.7000

P =
    0.0405
    0.0884
    0.1673
    0.2870

rho =
    0.1012
    0.1768
    0.2789
    0.4100

area =
    3.3018
    2.0704
    1.4674
    1.1526
```

Input Arguments

`gamma` — Specific heat ratios

scalar | array | real numbers greater than 1

Specific heat ratios, specified as an array or scalar of *N* specific heat ratios.

Dependencies

`gamma` must be a real, finite scalar greater than 1 for these input modes:

- Subsonic area ratio
- Supersonic area ratio

Data Types: `double`

`flow` — One isentropic flow relation

array | real numerical

One isentropic flow relation, specified as an array of real numerical values. This argument can be one of these types:

Isentropic Flow Type	Description
Mach numbers	Mach numbers, specified as a scalar or 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.
Temperature ratios	Temperature ratios on page 4-522, specified as an array or scalar 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'`.
Pressure ratios	Pressure ratios on page 4-522, specified as an array or scalar of real numbers greater than or equal to 0. • 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'`.
Density ratios	Density ratios on page 4-522, specified as an array or scalar 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 arrays, they must be the same size. Use `flow` with `mtype` value `'dens'`.
Area ratios	Area ratios on page 4-522, specified as a scalar real value greater than or equal to 1. Use `flow` with `mtype` value `'sup'`.

Data Types: `double`

`mtype` — Input mode of Fanno flow

`'mach'` (default) | `'temp'` | `'pres'` | `'dens'` | `'velo'` | `'totalpsub'` | `'totalpsup'` | `'fannosub'` | `'fannosup'`

Input mode of Fanno flow, specified as one of these values.

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
`'fannosup'`	Supersonic Fanno parameter

Data Types: `double`

Output Arguments

`mach` — Mach numbers

array

Mach numbers, returned as an array.

`T` — Temperature ratios

array

Temperature ratios on page 4-522, returned as an array.

`P` — Pressure ratios

array

Pressure ratios on page 4-522, returned as an array.

`rho` — Density ratios

array

Density ratios on page 4-522, returned as an array.

`area` — Density ratios

array

Area ratios on page 4-522, returned as an array.

Limitations

- This function assumes that variables vary only in one dimension. 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. For thermally perfect gas correction factors, see [2]. If the temperature is so high that molecules dissociate and ionize (static temperature 5000 K for air), you cannot assume a perfect gas.

More About

Temperature Ratio

Calculated as local static temperature over the stagnation temperature.

Pressure Ratio

Calculated as local static pressure over the stagnation pressure.

Density Ratio

Calculated as local density over the stagnation density.

Area Ration

Calculated as local stream tube area over the reference stream tube area for sonic conditions.

Version History

Introduced in R2010a

See Also

`flowfanno` | `flownormalshock` | `flowprandtlmeyer` | `flowrayleigh`

flownormalshock

Normal shock relations

Syntax

```
[mach,T,P,rho,downstream_mach,P0,P1] = flownormalshock(gamma,
normal_shock_relations,mtype)

[mach,T,P,rho,downstream_mach,P0,P1] = flownormalshock( ___ ,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 and downstream to be after or behind the shock.

`[mach,T,P,rho,downstream_mach,P0,P1] = flownormalshock( ___ ,mtype)` uses any one of the normal shock relations `mtype`. Specify `mtype` types after all other input arguments.

Examples

Calculate Normal Shock Relations for Gases with Specific Heat Ratios

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. This example 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)
```

```
mach =
    1.5000    1.5000    1.5000    1.5000

T =
    1.2473    1.2697    1.3202    1.4968

P =
    2.4130    2.4270    2.4583    2.5637

rho =
    1.9346    1.9116    1.8621    1.7128

downstream_mach =
    0.6942    0.6964    0.7011    0.7158

P0 =
```

```
    0.9261    0.9272    0.9298    0.9381

P1 =
    0.3062    0.3021    0.2930    0.2628
```

Calculate Normal Shock Relations for Air

Calculate the normal shock relations for air (`gamma` = 1.4) for a total pressure ratio of 0.61. This example 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')
```

```
mach =
    2.2401

T =
    1.8925

P =
    5.6875

rho =
    3.0053

downstream_mach =
    0.5418

P0 =
    0.6100

P1 =
    0.1440
```

Calculate Normal Shock Relations for Specific Heat Ratio and Range of Density Ratios

Calculate the normal shock relations for a specific heat ratio of 1.4 and a range of density ratios from 2.40 to 2.70 in increments of 0.10. This example 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')
```

```
mach =
    1.8257
    1.8898
    1.9554
    2.0226

T =
    1.5509
    1.6000
    1.6516
    1.7059

P =
```

```
    3.7222
    4.0000
    4.2941
    4.6061

rho =
    2.4000
    2.5000
    2.6000
    2.7000

downstream_mach =
    0.6108
    0.5976
    0.5852
    0.5735

P0 =
    0.8012
    0.7720
    0.7417
    0.7103

P1 =
    0.2088
    0.1964
    0.1847
    0.1737
```

Calculate Normal Shock Relations for Gases with Specific Heat Ratio and Downstream Mach Number Combinations

Calculate the normal shock relations for gases with a specific heat ratio and downstream Mach number combinations as shown. This 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')
```

```
mach =
   60.2773    2.7745

T =
  536.6972    2.4233

P =
   1.0e+03 *

    4.1071    0.0088

rho =
    7.6526    3.6374
```

```
downstream_mach =
    0.3400    0.4900

P0 =
    0.0000    0.3979

P1 =
    0.0002    0.0963
```

Input Arguments

gamma — Specific heat ratios

scalar | array | real numbers greater than 1

Specific heat ratios, specified as an array or scalar of *N* specific heat ratios.

Dependencies

`gamma` must be a real, finite scalar greater than 1 for these input modes:

- Temperature ratio
- Total pressure ratio
- Rayleigh-Pitot ratio

Data Types: `double`

normal_shock_relations — One normal shock relation

array | scalar

One normal shock relation, specified as an array or scalar of real numerical values. This argument can be one of these types:

Normal Shock Relation Types	Description
Mach numbers	Mach numbers, specified as a scalar or 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 the `mtype` value `'mach'`. Because `'mach'` is the default of `mtype`, `mtype` is optional when this array is the input mode.
Temperature ratio	Temperature ratios on page 4-529, specified as a scalar or array of real numbers. `normal_shock_relations` must be a real scalar greater than or equal to 1. Use `normal_shock_relations` with `mtype` value `'temp'`.

Normal Shock Relation Types	Description
Pressure ratios	Pressure ratios on page 4-529, specified as an array or scalar. `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'`.
Density ratios	Density ratios on page 4-529, specified as an array or scalar of real numbers that are: • Greater than or equal to 1 (at Mach number equal 1) • Less than or equal to 1 `(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'`.
Downstream Mach numbers	Mach numbers, specified as a 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 `flow` with `mtype` value `'down'`.
Total pressure ratio	Total pressure ratios on page 4-529, specified as a scalar. `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'`.

Normal Shock Relation Types	Description
Rayleigh-Pitot ratio	Rayleigh-Pitot ratio on page 4-529, specified as a scalar. `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)

Data Types: `double`

`mtype` — Input mode for normal shock relations
`'mach'` (default) | `'temp'` | `'pres'` | `'dens'` | `'down'` | `'totalp'` | `'pito'`

Input mode for normal shock relations, specified as one of these values.

Type	Description
`'mach'`	Default. Mach number.
`'temp'`	Temperature ratio.
`'pres'`	Pressure ratio.
`'dens'`	Density ratio.
`'down'`	Downstream Mach number.
`'totalp'`	Total pressure ratio.
`'pito'`	Rayleigh-Pitot ratio.

Data Types: `string`

Output Arguments

`mach` — Mach numbers
array

Mach numbers, returned as an array.

`P` — Pressure ratios
array

Pressure ratios on page 4-529, returned as an array.

`T` — Temperature ratios
array

Temperature ratios on page 4-529, returned as an array.

`rho` — Density ratios
array

Density ratios on page 4-529, returned as an array.

`downstream_mach` — Downstream Mach numbers
array

Downstream Mach numbers, returned as an array.

`P0` — Total pressure ratios
array

Total pressure ratios on page 4-529, returned as an array.

`P1` — Rayleigh-Pitot ratios
array

Rayleigh-Pitot ratios on page 4-529, returned as an array.

Limitations

- This function assumes that:
  - The medium is a calorically perfect gas.
  - The flow is frictionless and adiabatic.
  - The flow variables vary in one dimension only.
  - 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. For thermally perfect gas correction factors, see [2]. If the temperature is so high that molecules dissociate and ionize (static temperature 5000 K for air), you cannot assume a perfect gas.

More About

Pressure Ratio

Calculated as the static pressure downstream of the shock over the static pressure upstream of the shock.

Temperature Ratio

Calculated as the static temperature downstream of the shock over the static temperature upstream of the shock.

Density Ratio

Calculated as the fluid density downstream of the shock over the density upstream of the shock.

Total Pressure Ratio

Calculated as static pressure downstream of the shock over the static pressure upstream of the shock.

Rayleigh-Pitot Ratio

Static pressure upstream of the shock over the total pressure downstream of the shock.

Version History

Introduced in R2010a

References

[1] James, John E. A. *Gas Dynamics*. 2nd ed. Boston: Allyn and Bacon 1984.

[2] Ames Research Staff. *NACA Technical Report 1135*. Moffett Field, CA: National Advisory Committee on Aeronautics, 1953. 667–671.

See Also

`flowfanno` | `flowisentropic` | `flowprandtlmeyer` | `flowrayleigh`

flowprandtlmeyer

Calculate Prandtl-Meyer functions for expansion waves

Syntax

```
[mach,nu,mu] = flowprandtlmeyer(gamma,prandtlmeyer_array)

[mach,nu,mu] = flowprandtlmeyer( ___ ,mtype)
```

Description

Default Input Mode

`[mach,nu,mu] = flowprandtlmeyer(gamma,prandtlmeyer_array)` returns an array containing Mach numbers `mach`, Prandtl-Meyer angles `nu`, and Mach angles `mu`. `flowprandtlmeyer` calculates these arrays for a given set of specific heat ratios, `gamma`, for the Mach input mode.

Specify Input Mode

`[mach,nu,mu] = flowprandtlmeyer( ___ ,mtype)` uses any one of the isentropic flow types `mtype`. Specify `mtype` types after all other input arguments.

Examples

Calculate Prandtl-Meyer Functions for Gases with Specific Heat Ratios

Calculate the Prandtl-Meyer functions for gases with specific heat ratios. This example 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)
```

```
mach =
    1.5000    1.5000    1.5000    1.5000

nu =
   12.6928   12.4455   11.9052   10.2042

mu =
   41.8103   41.8103   41.8103   41.8103
```

Calculate Prandtl-Meyer Relations for Air

Calculate the Prandtl-Meyer relations for air (`gamma` = 1.4) for Prandtl-Meyer angle 61 degrees. This example returns a scalar for `mach`, `nu`, and `mu`.

```
[mach,nu,mu] = flowprandtlmeyer(1.4,61,'nu')
```

```
mach =
    3.6600
```

```
nu =
    61

mu =
   15.8564
```

Calculate Prandtl-Meyer Angles for Specific Heat Ratio and Range of Mach Angles

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 and returns a 4 x 1 column array for `mach`, `nu`, and `mu`.

```
[mach,nu,mu] = flowprandtlmeyer(1.4,(40:10:70)','mu')
```

```
mach =
    1.5557
    1.3054
    1.1547
    1.0642

nu =
   13.5505
    6.3185
    2.4868
    0.7025

mu =
    40
    50
    60
    70
```

Calculate Prandtl-Meyer Relations for Gases with Specific Heat Ratio and Mach Number Combinations

Calculate the Prandtl-Meyer relations for gases with specific heat ratio and Mach number combinations as shown. This example returns a 1 x 2 arrayeach for `nu` and `mu`, 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)
```

```
mach =
    1.1300    9.0000

nu =
    2.0405   99.3181
```

```
mu =
   62.2461    6.3794
```

Input Arguments

gamma — Specific heat ratios

scalar | array | real numbers greater than 1

Specific heat ratios, specified as an array or scalar of *N* specific heat ratios.

Data Types: `double`

prandtlmeyer_array — Prandtl-Meyer types

array | real number

Prandtl-Meyer types, specified as an array of one of these types.

Prandtl-Meyer Type	Description
Mach numbers	Mach numbers, specified as a scalar or array of *N* real numbers greater *N* real numbers greater than or equal to 1. If `prandtlmeyer_array` and `gamma` are arrays, they must be the same size. Mach angle inputs must be real numbers between 0 and 90 degrees, inclusive. Use `prandtlmeyer_array` with the `mtype` value `'mach'`. Because `'mach'` is the default of `mtype`, `mtype` is optional when this array is the input mode.
Prandtl-Meyer angle	Prandtl-Meyer angle on page 4-534, specified as a scalar or array of *N* real numbers greater than or equal to 0 in degrees. `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'`.
Mach angles	Mach angles on page 4-534, specified as a scalar or array of *N* in degrees. A Mach angle is a function of Mach number only.

Data Types: `double`

mtype — Input mode of Isentropic flow

`'mach'` (default) | `'nu'` | `'mu'`

Input mode of Isentropic flow, specified as one of these types.

Type	Description
`'mach'`	Mach number.

Type	Description
`'nu'`	Prandtl-Meyer angle.
`'mu'`	Mach angle.

Data Types: `double`

Output Arguments

`mach` — Mach numbers

array

Mach numbers, returned as an array.

`nu` — Prandtl-Meyer angles

array

Prandtl-Meyer angles on page 4-534, returned as an array.

`mu` — Mach angles

array

Mach angles on page 4-534, returned as an array.

Limitations

- 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.
- This function assumes that the environment is a perfect gas. 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.
  - 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. For thermally perfect gas correction factors, see [2].
  - The local static temperature is so high that molecules might dissociate and ionize (static temperature 5000 K for air). In this case, you cannot assume a calorically or thermally perfect gas.

More About

Prandtl-Meyer Angle

Angle change required for a Mach 1 flow to achieve a given Mach number after expansion.

Mach angle

Angle between the flow direction and the lines of pressure disturbance caused by supersonic motion in degrees.

Version History

Introduced in R2010a

References

[1] James, John E. A. *Gas Dynamics*. 2nd ed. Boston: Allyn and Bacon 1984.

[2] Ames Research Staff. *NACA Technical Report 1135*. Moffett Field, CA: National Advisory Committee on Aeronautics, 1953. 667–671.

See Also

`flowfanno` | `flowisentropic` | `flownormalshock` | `flowrayleigh`

flowrayleigh

Rayleigh line flow relations

Syntax

```
[mach,T,P,rho,velocity,T0,P0] = flowrayleigh(gamma,rayleigh_flow)

[mach,T,P,rho,velocity,T0,P0] = flowrayleigh(gamma,rayleigh_flow,mtype)
```

Description

`[mach,T,P,rho,velocity,T0,P0] = flowrayleigh(gamma,rayleigh_flow)` returns an array for each Rayleigh line flow relation. This function calculates these arrays for a given set of specific heat ratios (`gamma`) for the Mach input mode.

`[mach,T,P,rho,velocity,T0,P0] = flowrayleigh(gamma,rayleigh_flow,mtype)` uses any one of the Rayleigh flow types `mtype`. Specify `mtype` types after all other input arguments.

Examples

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 this 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 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
```

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
```

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.

Input Arguments

gamma — Specific heat ratios

array | scalar | real numbers greater than 1

Specific heat ratios, specified an array or scalar of *N* real numbers greater than 1.

Dependencies

`gamma` must be a real, finite scalar greater than 1 for these input modes:

- Low-speed temperature ratio
- High-speed temperature ratio
- Subsonic total temperature
- Supersonic total temperature

- Subsonic total pressure
- Supersonic total pressure

Data Types: `double`

`rayleigh_flow` — One Rayleigh line flow
array | real numerical

One Rayleigh line flow, specified as an array of real numerical values. This argument can be one of these types.

Normal Shock Relation Types	Description
Mach numbers	Mach numbers, specified as a scalar or array of *N* real numbers greater than or equal to 1. If `rayleigh_flow` and `gamma` are arrays, they must be the same size. Use `rayleigh_flow` with the `mtype` value `'mach'`. Because `'mach'` is the default of `mtype`, `mtype` is optional when this array is the input mode.
Temperature ratio	Temperature ratios on page 4-542, specified as a scalar of real numbers: • 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'`.
Pressure ratios	Pressure ratios on page 4-542, specified as an array or scalar. `normal_shock_relations` 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. . If `rayleigh_flow` and `gamma` are arrays, they must be the same size. Use `rayleigh_flow` with `mtype` value `'pres'`.
Density ratios	Density ratios on page 4-542, specified as an array or scalar of real numbers that are 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'`.

Normal Shock Relation Types	Description
Velocity ratios	Velocity ratios on page 4-542, specified as an array or scalar of *N* real numbers: • Greater than or equal to 0 • Less than or equal to `(gamma+1)/gamma` (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'`.

Total temperature ratio	Total temperature ratios on page 4-543, specified as a real scalar:
	• 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'`.

Total pressure ratio	Total pressure ratios on page 4-543, specified as a scalar:
	• 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'`.

Data Types: `double`

`mtype` — Input mode for Rayleigh
`'mach'` (default) | `'templo'` | `'temphi'` | `'pres'` | `'dens'` | `'velo'` | `'totaltsub'` | `'totaltsup'` | `'totalpsub'` | `'totalpsup'`

Input mode for the Rayleigh flow in `rayleigh_flow`, specified as one of these types.

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.

Data Types: `string`

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` — Mach numbers
array

Mach numbers, returned as an array.

`T` — Temperature ratios
array

Temperature ratios on page 4-515, returned as an array.

`P` — Pressure ratios
array

Pressure ratios on page 4-542, returned as an array.

`rho` — Density ratios
array

Density ratios on page 4-542, returned as an array.

`velocity` — Velocity ratios
array

Velocity ratios on page 4-542, returned as an array.

`T0` — Total temperature ratios
array

Total temperature ratios on page 4-543, returned as an array.

`P0` — Total pressure ratios
array

Total pressure ratios on page 4-543, returned as an array.

Limitations

- This function assumes that:
  - The medium is a calorically perfect gas in a constant area duct.
  - The flow is steady, frictionless, and one dimensional.
  - 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.
  - 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. For thermally perfect gas correction factors, see [2]. 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.

More About

Temperature Ratio

Calculated as the local static temperature over the reference static temperature for sonic flow.

Pressure Ratio

Calculated as the static pressure downstream of the shock over the static pressure upstream of the shock.

Density Ratio

Calculated as the fluid density downstream of the shock over the density upstream of the shock.

Velocity Ratio

Calculated as local velocity over the reference velocity for sonic flow.

Total Temperature Ratio

Calculated as the static temperature downstream of the shock over the static temperature upstream of the shock.

Total Pressure Ratio

Calculated as static pressure downstream of the shock over the static pressure upstream of the shock..

Version History

Introduced in R2010a

References

[1] James, John E. A. *Gas Dynamics*. 2nd ed. Boston: Allyn and Bacon 1984.

[2] Ames Research Staff. *NACA Technical Report 1135*. Moffett Field, CA: National Advisory Committee on Aeronautics, 1953. 667–671.

See Also

`flowisentropic` | `flownormalshock` | `flowprandtlmeyer` | `flowfanno`

fluxSolarAndGeomagnetic

Extract solar flux and geomagnetic field data

Syntax

```
[f107average,f107daily,magneticIndex] = fluxSolarAndGeomagnetic(year,
dayOfYear,UTseconds,matFile)
[___] = fluxSolarAndGeomagnetic(___,Name=Value)
```

Description

`[f107average,f107daily,magneticIndex] = fluxSolarAndGeomagnetic(year, dayOfYear,UTseconds,matFile)` extracts the solar flux and geomagnetic data from the MAT-file `matFile` that is generated by the `aeroReadSpaceWeatherData` function.

The `fluxSolarAndGeomagnetic` function returns the 81-day average of F10.7 solar flux `f107average` (centered on the day of year), daily F10.7 flux for previous day `f107daily`, and geomagnetic index information `magneticIndex` corresponding to the times specified in `year`, `dayOfYear`, and `UTseconds`. Use this output to specify the `f107average` ,`f107daily`, and `magneticIndex` values for the `atmosnrlmsise00` function.

`[___] = fluxSolarAndGeomagnetic(___,Name=Value)` specifies additional options using one or more name-value arguments. For example, to specify that the elements of `magneticIndex` are extrapolated to the nearest data point available in the MAT-file, set `MagneticIndexExtrapMethod` to `'none-clip'`.

Input Arguments

`year` — Year

m-by-1 array

Year, specified as an *m*-by-1 array. This function ignores the value of `year`.

Data Types: `double`

`dayOfYear` — Day of year

m-by-1 array

Day of year, specified as an *m*-by-1 array.

Data Types: `double`

`UTseconds` — Universal time

m-by-1 array

Universal time (UT), specified as an *m*-by-1 array in seconds.

Data Types: `double`

`matFile` — MAT-file of space weather data

character vector | string scalar

MAT-file of space weather data, specified as a character vector or string scalar. This file is the output from the `aeroReadSpaceWeatherData` function. If the file is not on the MATLAB path, specify the full pathname. `matFile` must contain these variables from the space weather data file:

- `YEAR`
- `MONTH`
- `DAY`
- `AP1`
- `AP2`
- `AP3`
- `AP4`
- `AP5`
- `AP6`
- `AP7`
- `AP8`
- `AP_AVG`
- `F107_OBS`
- `F107_DATA_TYPE`
- `F107_OBS_CENTER81`

For more information, see `aeroReadSpaceWeatherData`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `MagneticIndexExtrapMethod='none-clip'` specifies that the elements of `magneticIndex` are extrapolated to the nearest data point available in the MAT-file.

`F107ExtrapMethod` — Extrapolation method for `f107Average` and `f107Daily`
`'none-clip'` (default) | `'constant'` | `'least-squares-fit'`

Extrapolation method for `f107Average` and `f107Daily` for times outside the range of the MAT-file data, specified as one of these values.

Method	Description
`'none-clip'`	Set `f107Average` and `f107Daily` to the nearest data point available in the MAT-file.
`'constant'`	Set `f107Average` and `f107Daily` to a constant value specified by the `F107ExtrapolationValue` name-value argument.

Method	Description
`'least-squares-fit'`	Approximate `f107Average` and `f107Daily` using a least-squares fit of the space weather data from October 1, 1957, to December 1, 2040. This method uses a trigonometric function of the form: `a + b*cos(c*t + d*sin(e*t)` where *t* is time.

Data Types: `char` | `string`

`F107ExtrapValue` — Extrapolation value to assign to `f107Average` and to calculate `f107Daily`
scalar

Extrapolation value to assign to `f107Average` and to calculate `f107Daily`, specified as a scalar.

Dependencies

This value is assigned when:

- `F107ExtrapMethod` is set to `'constant'`.
- Time specified by `year`, `dayOfYear`, and `UTseconds` is outside the range of the data in the MAT-file.

Data Types: `double`

`MagneticIndexExtrapMethod` — Extrapolate `magneticIndex`
`'none-clip'` (default) | `'constant'` | `'igrf'`

Extrapolation method for `magneticIndex` values for times outside the range of the MAT-file data, specified as one of these values.

Method	Description
`'none-clip'`	Set elements of `magneticIndex` to the nearest data point available in the MAT-file.
`'constant'`	Set elements of `magneticIndex` to a constant value specified by the `F107ExtrapolationValue` name-value argument.
`'igrf'`	Calculate the elements of `magneticIndex` using the International Geomagnetic Reference Field. Because this model is defined for times between January 1, 1900, 12:00 AM UTC and January 1, 2025, 12:00 AM UTC, the predictions for times outside this range are clipped to values at these time limits.

Data Types: `char` | `string`

`MagneticIndexExtrapValue` — Extrapolation value used to calculate `magneticIndex` elements
scalar

Extrapolation value used to calculate `magneticIndex` elements, specified as a scalar.

Dependencies

This value is assigned when:

- `MagneticIndexExtrapMethod` is set to `'constant'`.
- Time specified by `year`, `dayOfYear`, and `UTseconds` is outside the range of the data in the MAT-file.

Data Types: `double`

Output Arguments

`f107average` — 81-day average of F10.7 flux
m-by-1 array

81-day average of F10.7 flux, centered on day of year (`dayOfYear`), specified as an *m*-by-1 array. These values correspond to the 10.7 cm radio flux at the actual distance of the Earth from the Sun. They do not correspond to the radio flux at 1 AU.

`f107daily` — Daily F10.7 flux for previous day
m-by-1 array

Daily F10.7 flux for previous day, specified as an *m*-by-1 array. These values correspond to the 10.7 cm radio flux at the actual distance of the Earth from the Sun. They do not correspond to the radio flux at 1 AU.

`magneticIndex` — Magnetic index information
m-by-7 array

Magnetic index information, specified as an *m*-by-7 array. 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

For more information, see “Limitations” on page 4-547.

Limitations

When specifying `MagneticIndexExtrapvalue` as `'igrf'`, the `fluxSolarAndGeomagnetic` function predicts the elements of `magneticIndex` outside the MAT-file data range using the International Geomagnetic Reference Field. Because this model is defined for times between January

1, 1900, 12:00 AM UTC and January 1, 2025, 12:00 AM UTC, the predictions for times outside this range are clipped to values at these time limits.

Algorithms

The default space weather data file `aeroSpaceWeatherData.mat` installed with the Aerospace Toolbox was created using the `aeroReadSpaceWeatherData` function. `aeroReadSpaceWeatherData` reads space weather data from a CelesTrak consolidated space weather data file into a MAT-file.

- Geomagnetic data - Provided by the Geomagnetic Observatory Niemegk, GFZ German Research Center for Geosciences, and licensed under the CC-BY-4.0 license. Matzka, J., Stolle, C., Yamazaki, Y., Bronkalla, O. and Morschhauser, A., 2021. The geomagnetic Kp index and derived indices of geomagnetic activity. Space Weather, `https://doi.org/10.1029/2020SW002641`.
- Solar Radio Flux Data - Provided by Dominion Radio Astrophysical Observatory and Natural Resources Canada.
- 45-Day and Monthly predictions - Provided by the Space Weather Prediction center of the National Oceanic and Atmospheric Administration (NOAA).

To generate an up-to-date space weather data MAT-file to use with the `fluxSolarAndGeomagnetic` function, use the `aeroReadSpaceWeatherData` function.

Version History

Introduced in R2023a

See Also

`aeroReadSpaceWeatherData` | `atmosnrlmsise00`

forcesAndMoments

Class: `Aero.FixedWing`
Package: `Aero`

Calculate forces and moments of fixed-wing aircraft

Syntax

```
[F,M] = forcesAndMoments(aircraft,state)
[F,M] = forcesAndMoments(aircraft,state,OutputReferenceFrame=outputReference)
```

Description

`[F,M] = forcesAndMoments(aircraft,state)` calculates the forces and moments of a fixed-wing aircraft, `aircraft`, based around a state `state`.

`[F,M] = forcesAndMoments(aircraft,state,OutputReferenceFrame=outputReference)` calculates the forces and moments using an output reference.

Input Arguments

`aircraft` — `Aero.FixedWing` object
scalar

`Aero.FixedWing` object, specified as a scalar.

`state` — `Aero.FixedWing.State` object
scalar

`Aero.FixedWing.State` object, specified as a scalar.

`OutputReferenceFrame` — Output reference frame
`"Body"` (default) | `"Wind"` | `"Stability"`

Output reference frame of the forces and moments calculation, specified as:

- `"Body"`
- `"Wind"`
- `"Stability"`

Example: `OutputReferenceFrame="Stability"`

Output Arguments

`F` — Forces
three-element vector

Forces output in `OutputReferenceFrame`, returned as a three-element vector.

`M` — Moments
three-element vector

Moments output in `OutputReferenceFrame`, returned as a three-element vector.

Examples

Calculate Forces and Moments of a Cessna 182

Calculate the forces and moments of a Cessna 182.

```
[C182,CruiseState] = astC182();
[F,M] = forcesAndMoments(C182, CruiseState)
```

```
F =
 -233.0908
         0
   -0.3300

M =
   1.0e+03 *

         0
    1.8739
         0
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `linearize` | `nonlinearDynamics`

Topics

“Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121

generatePatches

Class: `Aero.Body`
Package: `Aero`

Generate patches for body with loaded face, vertex, and color data

Syntax

```
generatePatches(h,ax)
```

Description

`generatePatches(h,ax)` generates patches for the animation body object `h` using the loaded face, vertex, and color data in axes object, `ax`.

Input Arguments

h — Aerospace body object

`Aero.Body` object

Aerospace body object, specified as an `Aero.Body` object.

ax — Axes object

`Axes` object

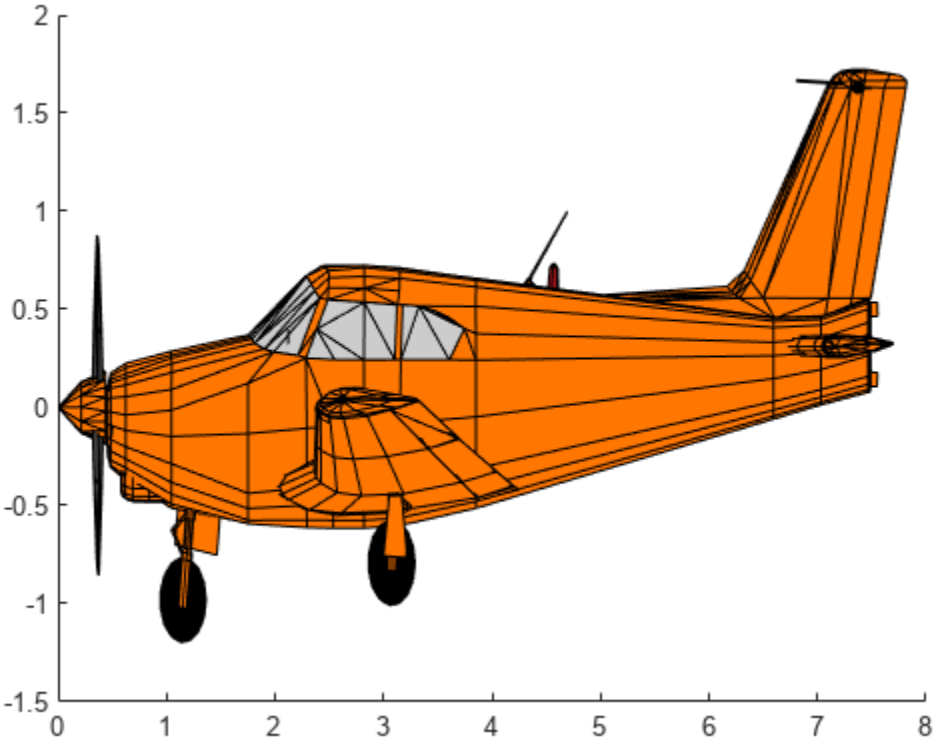
`Axes` object.

Examples

Generate Patches

Generate patches for the body object `b` using the axes, `ax`.

```
b=Aero.Body;
b.load('pa24-250_orange.ac','Ac3d');
f = figure;
ax = axes;
b.generatePatches(ax);
```



Version History

Introduced in R2007a

See Also

`Aero.Body` | `Aero.Animation`

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 2-44.

`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.
`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)
```

Version History

Introduced in R2007a

See Also

`initialize` | `play` | `update`

geoc2geod

Convert geocentric latitude to geodetic latitude

Syntax

```
geodeticLatitude = geoc2geod(geocentricLatitude,radii)
[geodeticLatitude,height] = geoc2geod(geocentricLatitude,radii)

geodeticLatitude = geoc2geod(geocentricLatitude,radii,model)
[geodeticLatitude,height] = geoc2geod(geocentricLatitude,radii,model)

geodeticLatitude = geoc2geod(geocentricLatitude,radii,flattening, Re)
[geodeticLatitude,height] = geoc2geod(geocentricLatitude,radii,flattening,Re)
```

Description

WGS84 Ellipsoid Planet

`geodeticLatitude = geoc2geod(geocentricLatitude,radii)` converts an array of geocentric latitudes and an array of radii from the center of the planet into an array of geodetic latitudes.

`[geodeticLatitude,height] = geoc2geod(geocentricLatitude,radii)` returns an array of altitudes above the planetary ellipsoid.

Specific Ellipsoid Planet

`geodeticLatitude = geoc2geod(geocentricLatitude,radii,model)` returns an array of radii, `radii`.

`[geodeticLatitude,height] = geoc2geod(geocentricLatitude,radii,model)` returns an array of altitude above planetary ellipsoid.

Custom Ellipsoid Planet

`geodeticLatitude = geoc2geod(geocentricLatitude,radii,flattening, Re)` returns an array of radii, `radii`.

`[geodeticLatitude,height] = geoc2geod(geocentricLatitude,radii,flattening,Re)` returns an array of altitude above planetary ellipsoid.

Examples

Determine Geodetic Latitude and Ellipsoidal Altitude

Determine geodetic latitude and ellipsoidal altitude.

```
[gd,h] = geoc2geod(45,6379136)
```

```
gd =
   45.1921
```

```
h =
   1.1718e+04
```

Determine Geodetic Latitude and Ellipsoidal Altitude at Multiple Geocentric Latitudes Using Radius and WGS84 Ellipsoid Model

Determine geodetic latitude and ellipsoidal altitude at multiple geocentric latitudes and radii that specify a WGS84 ellipsoid model.

```
[gd,h] = geoc2geod([0 45 90],6379136,'WGS84')
```

```
gd =
         0   45.1921   90.0000
h =
   1.0e+04 *
    0.0999    1.1718    2.2384
```

Determine Geodetic Latitude and Ellipsoidal Altitudes at Multiple Geocentric Latitudes Using Radii that Specify Custom Ellipsoid Model

Determine geodetic latitude at multiple geocentric latitudes given radii that specify a custom ellipsoid model.

```
f = 1/196.877360;
Re = 3397000;
[gd,h] = geoc2geod([0 45 90],6379136,f,Re)
```

```
gd =
         0   45.1550   90.0000
h =
   1.0e+06 *
    2.9821    2.9908    2.9994
```

Input Arguments

`geocentricLatitude` — Geocentric latitudes

array

Geocentric latitudes, specified as an array in degrees. Latitude values can be any value. However, values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `double`

`radii` — Radii from center of planet

array

Radii from center of planet, specified as an array in meters.

Data Types: `double`

`model` — Specific ellipsoid planet model

`'WGS84'`

Specific ellipsoid planet model, specified as `'WGS84'`.

Data Types: `char` | `string`

`flattening` — Flattening
scalar

Flattening at each pole, specified as a scalar.

Data Types: `double`

`Re` — Equatorial radius
scalar

Equatorial radius, specified as a scalar in meters.

Data Types: `double`

Output Arguments

`geodeticLatitude` — Geocentric latitudes
array

Geocentric latitudes, returned as an array in degrees.

`height` — Altitude above planetary ellipsoid
array

Altitude above planetary ellipsoid, returned as an array in meters.

Limitations

This function generates a geocentric latitude that lies between ±90 degrees.

Version History

Introduced in R2006b

See Also

`ecef2lla` | `geod2geoc` | `lla2ecef` | `lla2eci` | `eci2lla`

geocradius

Convert from geocentric latitude to radius of ellipsoid planet

Syntax

```
r = geocradius(lambda)
r = geocradius(lambda,model)

r = geocradius(lambda,f,Re)
```

Description

WGS84 Ellipsoid Planet

`r = geocradius(lambda)` estimates the radius, `r`, of an ellipsoid planet at a particular geocentric latitude, `lambda`.

`r = geocradius(lambda,model)` estimates the radius for a specific ellipsoid planet.

Custom Ellipsoid Planet

`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

Determine radius at 45 degrees latitude.

```
r = geocradius(45)
```

```
r =

  6.3674e+006
```

Determine radius at multiple latitudes

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 with WGS84 ellipsoid model

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 with custom ellipsoid model

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
```

Input Arguments

`lambda` — Geocentric latitude

scalar

Geocentric latitude, specified as a double, in degrees.

Data Types: `double`

`model` — Ellipsoid planet model

`'WGS84'` (default)

Ellipsoid planet model, specified as `'WGS84'`.

Data Types: `double`

`f` — Flattening

scalar

Flattening at each pole, specified as a scalar.

Data Types: `double`

`Re` — Equatorial radius

scalar

Equatorial radius, specified as a scalar in meters.

Data Types: `double`

Output Arguments

`r` — Radius
vector

Radius of an ellipsoid planet, returned as a double, in meters.

Version History

Introduced in R2021b

References

[1] Stevens, Brian L., Frank L. Lewis, *Aircraft Control and Simulation*, John Wiley & Sons, New York, NY, 1992.

[2] Zipfel, Peter H., and D. E. Penny, *Modeling and Simulation of Aerospace Vehicle Dynamics*. Second Edition. Reston, VA: AIAA Education Series, 2000.

See Also

`geoc2geod` | `geod2geoc`

geod2geoc

Convert geodetic latitude to geocentric latitude

Syntax

```
geocentricLatitude = geod2geoc(geodeticLatitude,height)
[geocentricLatitude,radii] = geod2geoc(geodeticLatitude,height)

geocentricLatitude = geod2geoc(geodeticLatitude,height,model)
[geocentricLatitude,radii] = geod2geoc(geodeticLatitude,height,model)

geocentricLatitude = geod2geoc(geodeticLatitude,height,flattening,Re)
[geocentricLatitude,radii] = geod2geoc(geodeticLatitude,height,flattening,Re)
```

Description

WGS84 Ellipsoid Planet

`geocentricLatitude = geod2geoc(geodeticLatitude,height)` converts an array of geodetic latitudes, `geodeticLatitude`, and an array of heights above the planetary ellipsoid, `height`, into an array of geocentric latitudes, `geocentricLatitude`.

`[geocentricLatitude,radii] = geod2geoc(geodeticLatitude,height)` returns an array of radii `radii` from the center of the planet to the center of gravity.

Specific Ellipsoid Planet

`geocentricLatitude = geod2geoc(geodeticLatitude,height,model)` converts from geodetic to geocentric latitude for a specific ellipsoid planet.

`[geocentricLatitude,radii] = geod2geoc(geodeticLatitude,height,model)` returns the radius `radii` from the center of the planet to the center of gravity.

Custom Ellipsoid Planet

`geocentricLatitude = geod2geoc(geodeticLatitude,height,flattening,Re)` converts from geodetic to geocentric latitude for a custom ellipsoid planet defined by flattening, `flattening`, and the equatorial radius, `equatorialRadius`.

`[geocentricLatitude,radii] = geod2geoc(geodeticLatitude,height,flattening,Re)` returns the radius `radii` from the center of the planet to the center of gravity.

Examples

Determine Geocentric Latitude and Radius with Geodetic Latitude and Altitude

Determine geocentric latitude given a geodetic latitudes and ellipsoidal altitudes.

```
[gc,r] = geod2geoc(45,1000)
```

```
gc =
   44.8076
```

```
r =
   6.3685e+06
```

Determine Geocentric Latitudes and Radii at Multiple Geodetic Latitudes and Altitudes, Specifying WGS84 Ellipsoid Model

Determine geocentric latitude at multiple geodetic latitudes and ellipsoidal altitudes, specifying WGS84 ellipsoid model.

```
[gc,r] = geod2geoc([0 45 90],[1000 0 2000],'WGS84')
```

```
gc =
         0   44.8076   90.0000

r =

   1.0e+06 *
    6.3791    6.3675    6.3588
```

Determine Geocentric Latitudes and Radii at Multiple Geodetic Latitudes, Given Altitude and Custom Ellipsoid Model

Determine geocentric latitude at multiple geodetic latitudes, given multiple geodetic latitudes that specify a custom ellipsoid model.

```
f = 1/196.877360;
Re = 3397000;
[gc,r] = geod2geoc([0 45 90],2000,f,Re)
```

```
gc =
         0   44.7084   90.0000

r =
   1.0e+06 *
    3.3990    3.3904    3.3817
```

Input Arguments

`geodeticLatitude` — Geodetic latitude

array

Geodetic latitude, specified as an array in degrees.

Data Types: `double`

`height` — Height above the planetary ellipsoid

scalar

Height above the planetary ellipsoid, specified as a scalar in meters.

Data Types: `double`

`model` — Specific ellipsoid planet model
`'WGS84'`

Specific ellipsoid planet model, specified as `'WGS84'`.

Data Types: `double`

`flattening` — Flattening
scalar

Flattening at each pole, specified as a scalar.

Data Types: `double`

`Re` — Equatorial radius
scalar

Equatorial radius, specified as a scalar in meters.

Data Types: `double`

Output Arguments

`geocentricLatitude` — Geocentric latitudes
array

Geocentric latitudes, returned as an array 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` — Radii from center of planet
array

Radii from the center of the planet, returned as an array in meters.

Limitations

This function generates a geocentric latitude that lies between ±90 degrees.

Version History
Introduced in R2006b

References

[1] Stevens, Brian L., Frank L. Lewis, *Aircraft Control and Simulation*, Wiley–Interscience, 1992.

See Also
`ecef2lla` | `geoc2geod` | `lla2ecef` | `lla2eci` | `eci2lla`

geoidegm96

Calculate geoid height as determined from EGM96 Geopotential Model

Syntax

```
N = geoidegm96(latitude,longitude)
N = geoidegm96(latitude,longitude,action)
```

Description

`N = geoidegm96(latitude,longitude)` calculates the geoid height as determined from the EGM96 Geopotential Model. 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(latitude,longitude,action)` performs `action` if latitude `latitude` or longitude `longitude` are out of range.

Examples

Calculate Geoid Height at 42.4 Degrees N Latitude and 71.0 Degrees E Longitude

Calculate the geoid height at 42.4 degrees N latitude and 71.0 degrees E longitude

```
N = geoidegm96( 42.4, 71.0)
```

```
N =
  -36.5900
```

Calculate the Geoid Height at Two Different Locations with Actions

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])
```

```
N =
  -33.0100   25.5500
```

Calculate Geoid Height with Latitude Wrapping with No Warnings

Calculate the geoid height with latitude wrapping, with out-of-range actions displaying no warnings.

```
N = geoidegm96(100,150,'None')
```

```
N =
   36.4100
```

Input Arguments

`latitude` — Geocentric latitudes

array of *m* values

Geocentric latitudes, specified as an array of *m* values in degrees, where north latitude is positive and south latitude is negative. If `lat` is not within the range -90 to 90, inclusive, this function wraps the value to be within the range.

Data Types: `double` | `single`

`longitude` — Geocentric longitudes

array of *m* values

Geocentric longitudes, specified as an array of *m* values in degrees, where east longitude is positive and west longitude is negative. If `long` is not within the range 0 to 360 inclusive, this function wraps the value to be within the range.

Data Types: `double` | `single`

`action` — Action

`'Warning'` (default) | `'Error'` | `'None'`

Action for out-of-range input, specified as:

- `'Error'` — Displays warning and indicates that the input is out of range
- `'Warning'` — Displays error and indicates that the input is out of range
- `'None'` — Does not display warning or error

Data Types: `char` | `string`

Output Arguments

`N` — Geoid height

array

Geoid height, returned as an array in meters. The function calculates geoid heights to 0.01 meters.

Version History

Introduced in R2007b

R2010b: Use `geoidheight` Instead

Behavior change in future release

`geoidegm96` will be removed in a future version. Use `geoidheight` instead.

References

[1] NIMA TR8350.2: "Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems."

[2] NASA/TP-1998-206861: "The Development of the Joint NASA GSFC and NIMA Geopotential Model EGM96."

See Also

`gravitywgs84`

External Websites

Office of Geomatics

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( ___ ,action)
```

Description

`N = geoidheight(latitude,longitude)` calculates the geoid height using the EGM96 Geopotential Model. For this geopotential model, the function calculates the geoid heights to an accuracy of 0.01 m and interpolates an array of *m* geoid heights at *m* geodetic latitudes, `latitude`, and *m* longitudes, `longitude`.

`N = geoidheight(latitude,longitude,modelname)` calculates the geoid height using the geopotential model, `modelname`.

`N = geoidheight(latitude,longitude,action)` calculates the geoid height and performs `action` if `latitude` or `longitude` are out of range.

`N = geoidheight(latitude,longitude,modelname,action)` calculates the geoid height using `modelname` and performs action if latitude or longitude are out of range.

`N = geoidheight(latitude,longitude,'custom',datafile)` calculates the geoid height using the custom model specified by `datafile`.

`N = geoidheight( ___ ,action)` calculates the geoid height using the custom geopotential model and performs function performs `action` if `latitude` or `longitude` are out of range. Specify `action` as the last input argument preceded by any of the input argument combinations in the previous syntaxes.

Examples

Calculate EGM96 Geoid Height with Warning

Calculate the EGM96 geoid height at 42.4 degrees N latitude and 71.0 degrees W longitude. A warning, enabled by default, is returned for the out-of-range longitude value:

```
N = geoidheight(42.4,-71.0)
```

```
Warning: One or more longitude values exceed [0,360] range. Wrapping out of
range longitude values within 0 degrees and 360 degrees and continuing.
> In geoidheight>@()warning(message('aero:geoidheight:warnLongitudeWrap')) (line 324)
In geoidheight/checklongitude (line 328)
In geoidheight (line 166)
```

```
N =

  -28.3700
```

Calculate EGM2008 Geoid height at Two Locations with Error Action

Calculate the EGM2008 geoid height at two different locations. The function returns an error if the results are out of range:

```
N = geoidheight([39.3, 33.4],[77.2,36.5],'egm2008','error')
```

```
N =

  -49.9440   23.6110
```

Calculate Custom Geoid Height at Two Locations

Calculate a custom geoid height at two different locations:

```
N = geoidheight([39.3,33.4],[-77.2,36.5],'custom',...
'geoidegm96grid','none')
```

```
N =

  -33.0100   25.5500
```

Input Arguments

`latitude` — Geodetic latitudes

array

Geodetic latitudes, specified as an array of *`m`* geodetic latitudes, in degrees, where north latitude is positive and south latitude is negative.

If *`latitude`* is not within the range –90 to 90, inclusive, this function wraps the value to be within the range when `action` is set to `'None'` or `'Warning'`. It does not wrap when `action` is set to `'Error'`.

Data Types: `double` | `single`

`longitude` — Longitudes

array

Longitudes, specified as an array of *`m`* longitudes, in degrees, where east longitude is positive and west longitude is negative. If *`longitude`* is not within the range 0 to 360 inclusive, this function wraps the value to be within the range.

If *`longitude`* is not within the range –90 to 90, inclusive, this function wraps the value to be within the range when `action` is set to `'None'` or `'Warning'`. It does not wrap when `action` is set to `'Error'`.

Data Types: `double` | `single`

`modelname` — Geopotential model
`'EGM96'` (default) | `'EGM2008'` | `'custom'`

Geopotential model, specified as:

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`.

Data Types: `char` | `string`

`datafile` — Custom geopotential model definitions
scalar

Custom geopotential model definitions, specified as a scalar file of definitions for a custom geopotential model.

This file must contain these variables:

Variable	Description
`'latbp'`	Array of geodetic latitude breakpoints.
`'lonbp'`	Array of longitude breakpoints.
`'grid'`	Table of geoid height values.
`'windowSize'`	Even integer scalar greater than 2 for the number of interpolation points.

Data Types: `char` | `string`

`action` — Action
`'Warning'` (default) | `'Error'` | `'None'`

Action for out-of-range input, specified as:

- `'Error'` — Displays warning and indicates that the input is out of range
- `'Warning'` — Displays error and indicates that the input is out of range
- `'None'` — Does not display warning or error

Data Types: `char` | `string`

Output Arguments

`N` — Geoid heights
array | same data type as `latitude` argument

Geoid heights, returned as an array of *M* geoid heights, in meters.

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.

Version History

Introduced in R2010b

References

[1] Vallado, D. A. "Fundamentals of Astrodynamics and Applications." McGraw-Hill, New York, 1997.

[2] NIMA TR8350.2: "Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems."

See Also

`gravitywgs84` | `gravitysphericalharmonic`

External Websites

Office of Geomatics

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.

Version History

Introduced in R2007a

See Also

`Aero.Geometry`

getCoefficient

Class: `Aero.FixedWing`
Package: `Aero`

Get coefficient value for `Aero.FixedWing` object

Syntax

```
value = getCoefficient(aircraft,stateOutput,stateVariable)
value = getCoefficient( ___ ,Name,Value)
```

Description

`value = getCoefficient(aircraft,stateOutput,stateVariable)` gets the coefficient value `value` from the coefficient specified by `stateOutput` and `stateVariable`.

`value = getCoefficient( ___ ,Name,Value)` gets the coefficient value using one or more `Name,Value` pairs.

Input Arguments

`aircraft` — `Aero.FixedWing` object

scalar

`Aero.FixedWing` object, specified as a scalar.

`stateOutput` — Valid state output

vector of strings | character array

Valid state output, specified in a vector of strings or character array. For more information, see `Aero.FixedWing.Coefficient`.

Data Types: `string` | `char`

`stateVariable` — Valid state variable

vector

Valid state variable, specified in a vector of strings or character array. Valid state variables depend on the coefficients defined on the object. For more information, see `Aero.FixedWing.Coefficient`.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Component','Hello'`

`State` — `Aero.FixedWing.State` object to calculate numeric values of `Simulink.LookupTable` objects
scalar

`Aero.FixedWing.State` object to calculate numeric values of `Simulink.LookupTable` objects, specified as a scalar.

Data Types: `double`

`Component` — Valid component name
scalar string

Valid component name, specified as a scalar string. Valid component names depend on the `'Name'` property of an object and all its subcomponents. The default component is the current object.

Data Types: `char` | `string`

Output Arguments

`value` — Coefficient values
vector

Coefficient values, returned as a value of the same size as `stateOutput` and `stateVariable`. Vector contents depend on the type of coefficients in the vector.

Type of Coefficients in Vector	Vector
All numeric constants	Numeric vector
`Simulink.LookupTable` objects	Vector of `Simulink.LookupTable` objects
Mix of numeric constants and `Simulink.LookupTable` objects	Vector of cells
`Simulink.LookupTable` objects with state included	Numeric vector

Examples

Get Coefficient for Angle of Attack

Get a `CD_alpha` on an `Aero.FixedWing` object.

```
C182 = astC182();
CD_alpha = getCoefficient(C182, "CD", "Alpha")
```

```
CD_alpha =
    0.1210
```

Get Vector of Coefficient Values

Get a vector of coefficient values on a component within an `Aero.FixedWing` object.

```
C182 = astC182();
coeffs = getCoefficient(C182,{'CY'; 'Cm'},{'Aileron';'Aileron'},'Component','Aileron')
```

Get Simulink.LookupTable Coefficient

Get a `Simulink.LookupTable` coefficient from an `Aero.FixedWing` object.

```
SkyHogg = astSkyHogg();
Cl_zero = getCoefficient(SkyHogg,"Cl","Zero")
```

```
Cl_zero =
     0
```

Get Simulink.LookupTable Coefficient with State

Get a `Simulink.LookupTable` coefficient from an `Aero.FixedWing` object and include a state.

```
[SkyHogg, CruiseState] = astSkyHogg();
Cl_zero = getCoefficient(SkyHogg, "Cl", "Zero", "State", CruiseState)
```

```
Cl_zero =
     0
```

Limitations

- Each vector of inputs `stateOutput` and `stateVariable` must be the same length.
- When used with `Simulink.LookupTable` objects, this method requires a Simulink license.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `setCoefficient` | `Simulink.LookupTable`

getCoefficient

Class: `Aero.FixedWing.Coefficient`
Package: `Aero`

Get coefficient values from fixed-wing coefficient object

Syntax

```
value = getCoefficient(fixedWingCoefficient,stateOutput,stateVariable)
value = getCoefficient( ___ ,Name,Value)
```

Description

`value = getCoefficient(fixedWingCoefficient,stateOutput,stateVariable)` gets the coefficient value `value` from the coefficient specified by `stateOutput` and `stateVariable`.

`value = getCoefficient( ___ ,Name,Value)` gets the coefficient value using one or more `Name,Value` pairs.

Input Arguments

`fixedWingCoefficient` — `Aero.FixedWing.Coefficient` object for which to get coefficient

scalar

`Aero.FixedWing.Coefficient` object for which to get coefficient, specified as a scalar.

`stateOutput` — State output

6-by-1 vector

State output, specified as a 6-by-1 vector where each entry is a valid state output. For more information on state outputs, see `Aero.FixedWing.Coefficient`.

Data Types: `char` | `string`

`stateVariable` — State variable

vector

State variable, specified as a vector where each entry is a valid state variable. Valid state variables depend on the coefficients defined on the object. For more information on fixed-wing states, see `Aero.FixedWing.State`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'State','on'`

`State` — `Aero.FixedWing.State` object
scalar | `Aero.FixedWing.State` object

`Aero.FixedWing.State` object, specified as a scalar, that calculates the numeric values of any `Simulink.LookupTable` objects. Including an `Aero.FixedWing.State` guarantees `value` is a numeric vector.

Data Types: `char` | `string`

`Component` — Component name
string

Component name, specified as a string. Valid component names depend on the object properties and all subcomponents on the object. The default component name is the current object.

Data Types: `char` | `string`

Output Arguments

`value` — Coefficient values
vector

Coefficient values, returned as a vector of the same size as `stateOutput` and `stateVariable`. Vector contents depend on the type of coefficients in the vector.

Type of Coefficients in Vector	Vector
All numeric constants	Numeric vector
`Simulink.LookupTable` objects	Vector of `Simulink.LookupTable` objects
Mix of numeric constants and `Simulink.LookupTable` objects	Vector of cells
`Simulink.LookupTable` objects with state included	Numeric vector

Examples

Get `CD_alpha` on Fixed-Wing Coefficient Object

Get a `CD_alpha` on a fixed-wing coefficient object.

```
C182 = astC182();
CD_alpha = getCoefficient(C182, "CD", "Alpha")
```

```
CD_alpha =

    0.1210
```

Get Vector of Coefficient Values

Get a vector of coefficient values on a component within a fixed-wing coefficient object.

```
C182 = astC182();
coeffs = getCoefficient(C182, ["CY"; "Cm"], ["Aileron"; "Aileron"], "Component", "Aileron")
```

```
coeffs =

     0     0
```

Get Simulink.LookupTable Coefficient

Get a `Simulink.LookupTable` coefficient from a fixed-wing coefficient object.

```
SkyHogg = astSkyHogg();
Cl_zero = getCoefficient(SkyHogg, "Cl", "Zero")
```

```
Cl_zero =

     0
```

Get Simulink.LookupTable Coefficient and Include State

Get a `Simulink.LookupTable` coefficient from a fixed-wing coefficient object and include a state.

```
[SkyHogg, CruiseState] = astSkyHogg();
Cl_zero = getCoefficient(SkyHogg, "Cl", "Zero", "State", CruiseState)
```

```
Cl_zero =

     0
```

Limitations

- The vectors for the `stateOutput`, `stateVariable`, and `value` arguments must be the same length.
- When used with `Simulink.LookupTable` objects, this method requires a Simulink license.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `getCoefficient` | `setCoefficient`

getCoefficient

Class: `Aero.FixedWing.Surface`
Package: `Aero`

Get coefficient for fixed-wing surface object

Syntax

```
value = getCoefficient(fixedWingSurface,stateOutput,stateVariable)
value = getCoefficient( ___ ,Name,Value)
```

Description

`value = getCoefficient(fixedWingSurface,stateOutput,stateVariable)` gets the coefficient value `value` from the coefficient specified by `stateOutput` and `stateVariable`.

`value = getCoefficient( ___ ,Name,Value)` gets the coefficient value using one or more `Name,Value` pairs.

Input Arguments

`fixedWingSurface` — `Aero.FixedWing.Surface` object for which to get coefficient

scalar

`Aero.FixedWing.Surface` object for which to get coefficient, specified as a scalar.

`stateOutput` — State output

6-by-1 vector

State output, specified as a 6-by-1 vector where each entry is a valid state output. For more information on state outputs, see `Aero.FixedWing.Coefficient`.

Data Types: `char` | `string`

`stateVariable` — State variable

vector

State variable, specified as a vector where each entry is a valid state variable. Valid state variables depend on the coefficients defined on the object. For more information on fixed-wing states, see `Aero.FixedWing.State`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Component','Hello'`

`State` — `Aero.FixedWing.State` object to calculate numeric values of `Simulink.LookupTable` objects
scalar

`Aero.FixedWing.State` object to calculate numeric values of `Simulink.LookupTable` objects, specified as a scalar.

Data Types: `double`

`Component` — Valid component `Name`
scalar

Valid component name, specified as a scalar string. Valid component names depend on the `'Name'` property of an object and all its subcomponents. The default component is the current object.

Data Types: `char` | `string`

Output Arguments

`value` — Coefficient values
vector

Coefficient values, returned as a vector of the same size as `stateOutput` and `stateVariable`. Vector contents depend on the type of coefficients in the vector.

Type of Coefficients in Vector	Vector
All numeric constants	Numeric vector
`Simulink.LookupTable` objects	Vector of `Simulink.LookupTable` objects
Mix of numeric constants and `Simulink.LookupTable` objects	Vector of cells
`Simulink.LookupTable` objects with state included	Numeric vector

Version History
Introduced in R2021a

See Also
`Aero.FixedWing` | `Aero.FixedWing.Coefficient` | `Aero.FixedWing.Surface` | `Aero.FixedWing.Thrust` | `setCoefficient`

getCoefficient

Class: `Aero.FixedWing.Thrust`
Package: `Aero`

Get coefficient for fixed-wing thrust object

Syntax

```
value = getCoefficient(fixedWingThrust,stateOutput,stateVariable)
value = getCoefficient( ___ ,Name,Value)
```

Description

`value = getCoefficient(fixedWingThrust,stateOutput,stateVariable)` gets the coefficient value `value` from the coefficient specified by `stateOutput` and `stateVariable`.

`value = getCoefficient( ___ ,Name,Value)` gets the coefficient value using one or more `Name,Value` pairs.

Input Arguments

`fixedWingThrust` — `Aero.FixedWing.Thrust` object for which to get coefficient

scalar

`Aero.FixedWing.Thrust` object for which to get coefficient, specified as a scalar.

`stateOutput` — State output

6-by-1 vector

State output, specified as a 6-by-1 vector where each entry is a valid state output. For more information on state outputs, see `Aero.FixedWing.Coefficient`.

Data Types: `char` | `string`

`stateVariable` — State variable

vector

State variable, specified as a vector where each entry is a valid state variable. Valid state variables depend on the coefficients defined on the object. For more information on fixed-wing states, see `Aero.FixedWing.State`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Component','Hello'`

`State` — `Aero.FixedWing.State` object to calculate numeric values of `Simulink.LookupTable` objects
scalar

`Aero.FixedWing.State` object to calculate numeric values of `Simulink.LookupTable` objects, specified as a scalar.

Data Types: `double`

`Component` — Valid component name
scalar

Valid component name, specified as a scalar string. Valid component names depend on the `'Name'` property of an object and all its subcomponents. The default component is the current object.

Data Types: `char` | `string`

Output Arguments

`value` — Coefficient values
vector

Coefficient values, specified as a vector of the same size as `stateOutput` and `stateVariable`. Vector contents depend on the type of coefficients in the vector.

Type of Coefficients in Vector	Vector Type
All numeric constants	Numeric vector
`Simulink.LookupTable` objects	Vector of `Simulink.LookupTable` objects
Mix of numeric constants and `Simulink.LookupTable` objects	Vector of cells
`Simulink.LookupTable` objects with state included	Numeric vector

Version History
Introduced in R2021a

See Also

Topics
`Aero.FixedWing`
`Aero.FixedWing.Thrust`
`setCoefficient`
`Simulink.LookupTable`

getControlStates

Class: Aero.FixedWing
Package: Aero

Get control states for Aero.FixedWing object

Syntax

```
control_states = getControlStates(aircraft)
```

Description

control_states = getControlStates(aircraft) gets Aero.Aircraft.ControlState control states for the Aero.FixedWing object aircraft.

Input Arguments

aircraft — Aero.FixedWing coefficient object

scalar | Aero.FixedWing | Aero.FixedWing.Surface | Aero.FixedWing.Control | Aero.FixedWing.Thrust

Aero.FixedWing coefficient object, specified as a scalar of type Aero.FixedWing, Aero.FixedWing.Surface, Aero.FixedWing.Control, or Aero.FixedWing.Thrust.

Output Arguments

control_states — Aero.Aircraft.ControlState objects

vector

Aero.Aircraft.ControlState objects, returned as a vector.

Examples

Get Aero.Aircraft.ControlState Control States

Get the control states for an Aero.FixedWing object.

```
C182 = astC182();
ctrlStates = getControlStates(C182)
```

```
ctrlStates =
  1×4 ControlState array with properties:

    Position
    MaximumValue
    MinimumValue
    DependsOn
    Settable
```

```
DeflectionAngle
Properties
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Aero.Aircraft.ControlState` | `Aero.FixedWing.State`

getControlStates

Class: `Aero.FixedWing.Surface`
Package: `Aero`

Get control states for `Aero.FixedWing.Surface` object

Syntax

```
control_states = getControlStates(surface)
```

Description

`control_states = getControlStates(surface)` gets control states, `control_states`, for the `Aero.FixedWing.Surface` object, `surface`.

Input Arguments

`surface` — `Aero.FixedWing.Surface` object
scalar

`Aero.FixedWing.Surface` object, specified as a scalar.

Output Arguments

`control_states` — `Aero.Aircraft.ControlState` objects
vector

`Aero.Aircraft.ControlState` objects, returned as a vector.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Aero.Aircraft.ControlState` | `Aero.FixedWing.State` | `getControlStates`

getControlStates

Class: `Aero.FixedWing.Thrust`
Package: `Aero`

Get control states for `Aero.FixedWing.Thrust` object

Syntax

```
control_states = getControlStates(thrust)
```

Description

`control_states = getControlStates(thrust)` gets control states, `control_states`, for the `Aero.FixedWing.Thrust` object, `thrust`.

Input Arguments

`thrust` — `Aero.FixedWing.Thrust` object
scalar

`Aero.FixedWing.Thrust` object, specified as a scalar.

Output Arguments

`control_states` — `Aero.Aircraft.ControlState` objects
vector

`Aero.Aircraft.ControlState` objects, returned as a vector.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Aero.Aircraft.ControlState` | `Aero.FixedWing.State` | `getControlStates`

getState

Class: `Aero.FixedWing.State`
Package: `Aero`

Get state value

Syntax

```
value = getState(state,statename)
```

Description

`value = getState(state,statename)` gets the state `value` from the state name `statename`.

Input Arguments

`state` — `Aero.FixedWing.State` object

scalar

`Aero.FixedWing.State` object, specified as a scalar.

`statename` — State name

vector

State names, specified as a vector. For more information on state names, see the `Aero.FixedWing.State` “Properties” on page 4-79.

Data Types: `char` | `string`

Output Arguments

`value` — State values

vector

State values, returned as a vector.

- If the states are all scalar constants, `value` is a numeric vector.
- If one of more states are not scalar constants, `value` is a cell vector.

Examples

Get Angle of Attack from Cruise State

Get the angle of attack from a cruise state.

```
[C182, CruiseState] = astC182();
alpha = getState(CruiseState, 'Alpha')
```

```
alpha =

     0
```

Get U, V, and W Velocity Components

Get the *U*, *V*, and *W* velocity components from a cruise state.

```
[C182, CruiseState] = astC182();
uvw = getState(CruiseState, {'U', 'V', 'W'})
```

```
uvw =
  220.1000         0         0
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing.State` | `setState` | `setupControlStates`

gimbal

Add gimbal to satellite or ground station

Syntax

```
gimbal(parent)
gimbal(parent,Name=Value)
gimbal( ___ )
```

Description

`gimbal(parent)` adds a default `Gimbal` object to each parent in the `parent` vector, which can be a satellite or a ground station. A gimbal can be added to satellites and ground stations, and dynamically change orientation independent of the parent. Transmitters, receivers, and conical sensors can be mounted on the gimbals.

`gimbal(parent,Name=Value)` adds gimbals to parents in `parent` using additional parameters specified by optional name-value pairs.

`gim = gimbal( ___ )` returns the added gimbals in the row vector `gim`.

Examples

Calculate Maximum Revisit Time of Satellite

Create a satellite scenario with a start time of 15-June-2021 8:55:00 AM UTC and a stop time of five days later. Set the simulation sample time to `60` seconds.

```
startTime = datetime(2021,6,21,8,55,0);
stopTime = startTime + days(5);
sampleTime = 60;                                          % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc =
  satelliteScenario with properties:

         StartTime: 21-Jun-2021 08:55:00
          StopTime: 26-Jun-2021 08:55:00
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add a satellite to the scenario using Keplerian orbital elements.

```
semiMajorAxis = 7878137;                                                    % me
eccentricity = 0;
inclination = 50;                                                           % deg
```

```
rightAscensionOfAscendingNode = 0;                                                  % deg
argumentOfPeriapsis = 0;                                                            % deg
trueAnomaly = 50;                                                                   % deg
sat = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly)
```

```
sat =
  Satellite with properties:

              Name:  Satellite 1
                ID:  1
     ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
            Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
       Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
          Receivers:  [1x0 satcom.satellitescenario.Receiver]
           Accesses:  [1x0 matlabshared.satellitescenario.Access]
         GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
              Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
    OrbitPropagator:  sgp4
        MarkerColor:  [0.059 1 1]
         MarkerSize:  6
          ShowLabel:  true
     LabelFontColor:  [1 1 1]
      LabelFontSize:  15
```

Add a ground station, which represents the location to be photographed, to the scenario.

```
gs = groundStation(sc,Name="Location to Photograph", ...
    Latitude=42.3001,Longitude=-71.3504)                % degrees
```

```
gs =
  GroundStation with properties:

                Name:  Location to Photograph
                  ID:  2
            Latitude:  42.3 degrees
           Longitude:  -71.35 degrees
            Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
          MarkerColor:  [1 0.4118 0.1608]
           MarkerSize:  6
            ShowLabel:  true
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
```

Add a gimbal to the satellite. You can steer this gimbal independently of the satellite.

```
g = gimbal(sat)
```

```
g =
  Gimbal with properties:
```

```
                Name:  Gimbal 3
                  ID:  3
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
```

Track the location to be photographed using the gimbal.

```
pointAt(g,gs);
```

Add a conical sensor to the gimbal. This sensor represents the camera. Set the field of view to 60 degrees.

```
camSensor = conicalSensor(g,MaxViewAngle=60)
```

```
camSensor =
  ConicalSensor with properties:

                Name:  Conical sensor 4
                  ID:  4
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
        MaxViewAngle:  60 degrees
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         FieldOfView:  [0x0 matlabshared.satellitescenario.FieldOfView]
```

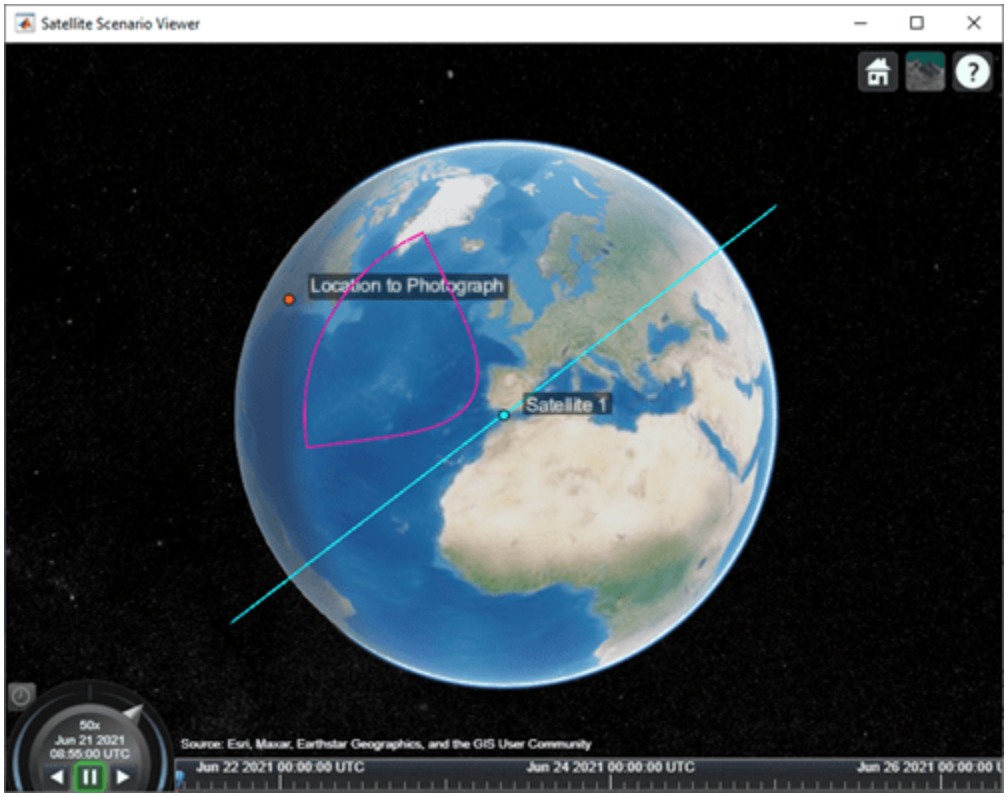
Add access analysis to the conical sensor between the camera and the location to be photographed.

```
ac = access(camSensor,gs)
```

```
ac =
  Access with properties:

    Sequence:  [4 2]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

Visualize the field of view of the camera by using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
fieldOfView(camSensor);
```



Determine the intervals during which the camera can see the geographical site.

```
t = accessIntervals(ac)
```

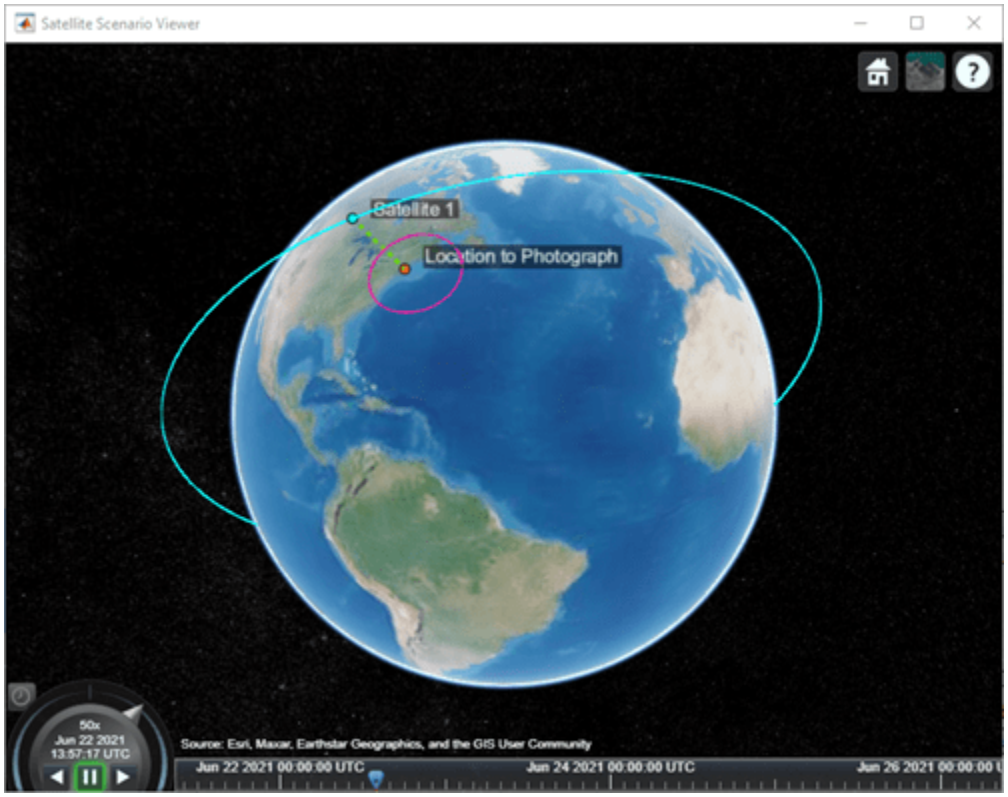
```
t=35×8 table
          Source                 Target            IntervalNumber        StartTime
    ________________    ________________________    ______________    ____________________

    "Conical sensor 4"    "Location to Photograph"          1          21-Jun-2021 10:38:00
    "Conical sensor 4"    "Location to Photograph"          2          21-Jun-2021 12:36:00
    "Conical sensor 4"    "Location to Photograph"          3          21-Jun-2021 14:37:00
    "Conical sensor 4"    "Location to Photograph"          4          21-Jun-2021 16:41:00
    "Conical sensor 4"    "Location to Photograph"          5          21-Jun-2021 18:44:00
    "Conical sensor 4"    "Location to Photograph"          6          21-Jun-2021 20:46:00
    "Conical sensor 4"    "Location to Photograph"          7          21-Jun-2021 22:50:00
    "Conical sensor 4"    "Location to Photograph"          8          22-Jun-2021 09:51:00
    "Conical sensor 4"    "Location to Photograph"          9          22-Jun-2021 11:46:00
    "Conical sensor 4"    "Location to Photograph"         10          22-Jun-2021 13:46:00
    "Conical sensor 4"    "Location to Photograph"         11          22-Jun-2021 15:50:00
    "Conical sensor 4"    "Location to Photograph"         12          22-Jun-2021 17:53:00
    "Conical sensor 4"    "Location to Photograph"         13          22-Jun-2021 19:55:00
    "Conical sensor 4"    "Location to Photograph"         14          22-Jun-2021 21:58:00
    "Conical sensor 4"    "Location to Photograph"         15          23-Jun-2021 10:56:00
    "Conical sensor 4"    "Location to Photograph"         16          23-Jun-2021 12:56:00
      ⋮
```

Calculate the maximum revisit time in hours.

```
startTimes = t.StartTime;
endTimes = t.EndTime;
revisitTimes = hours(startTimes(2:end) - endTimes(1:end-1));
maxRevisitTime = max(revisitTimes)                                    % hours

maxRevisitTime = 12.6667
```

Visualize the revisit times that the camera photographs of the location.

```
play(sc);
```



Input Arguments

`parent` — Element of scenario to which you add gimbal

scalar | vector

Element of scenario to which you add the gimbal, specified as a scalar or vector of satellites or ground stations. The number of gimbals specified is determined by the size of the inputs.

- If `parent` is a scalar, all gimbals are added to the parent.
- If `parent` is a vector and the number of gimbals specified is one, that gimbal is added to each parent.
- If `parent` is a vector and the number of gimbals specified is more than one, the number of gimbals must equal the number of `parent`s and each `parent` gets one gimbal.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MountingAngle'=[20;35;10]` sets the yaw, pitch, and roll angles of the gimbal to 20, 35, and 10 degrees, respectively.

Note The size of the name-value arguments defines the number of gimbals that you can specify. To understand how to specify multiple gimbals, refer to these properties.

`Name` — Gimbal name

`"Gimbal idx"` (default) | string scalar | string vector | character vector | cell array of character vectors

You can set this property only when calling the `gimbal` function. After you call the `gimbal` function, this property is read-only.

Gimbal name, specified as a name-value argument consisting of `'Name'` and a string scalar, string vector, character vector, or a cell array of character vectors.

- If you are adding only one gimbal, specify `Name` as a string scalar or a character vector.
- If you are adding multiple gimbals, specify `Name` as a string scalar, character vector, string vector, or a cell array of character vectors. All gimbals that you add as a string scalar or a character vector are assigned the same specified name. The number of elements in the string vector or cell array of character vectors must equal the number of gimbals that you are adding. Each gimbal is assigned the corresponding name from the vector or cell array.

In the default value, *idx* is the ID assigned by the satellite scenario.

Data Types: `char` | `string`

`MountingLocation` — Mounting location with respect to parent

`[0; 0; 0]` (default) | three-element vector | matrix

Mounting location with respect to the parent object in meters, specified as a three-element vector or a matrix. The position vector is specified in the body frame of the input `parent`.

- If you are adding one gimbal, `MountingLocation` is a three-element vector. The elements specify the *x*, *y*, and *z* components of the Cartesian coordinates in the body frame of gimbal.
- If you are adding multiple gimbals, `MountingLocation` can be a three-element vector or a matrix. When specified as a vector, the same set of mounting locations are assigned to all specified gimbals. When specified as a matrix, `MountingLocation` must contain three rows and the same number of columns as the gimbals. The columns correspond to the mounting location of each specified gimbal and the rows correspond to the mounting location coordinates in the parent body frame.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify the `MountingLocation` property only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Data Types: `double`

`MountingAngles` — Mounting orientation with respect to parent object
`[0; 0; 0]` (default) | three-element row vector of positive numbers | matrix

Mounting orientation with respect to parent object in degrees, specified as a three-element row vector of positive numbers. The elements of the vector correspond to yaw, pitch, and roll, in that order. Yaw, pitch, and roll are positive rotations about the *z*-axis, intermediate *y*-axis, and intermediate *x*-axis of the parent.

- If you are adding one gimbal, the `MountingAngles` property is a three-element vector.
- If you are adding multiple gimbals the `MountingAngles` property can be a three-element vector or a matrix. When specified as a vector, the same set of mounting angles are assigned to all specified gimbals. When specified as a matrix, `MountingAngles` must contain three rows and the same number of columns as the gimbals. The columns correspond to the mounting angles of each specified gimbal and the rows correspond to the yaw, pitch, and roll angles in the parent body frame.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify the `MountingAngles` property only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Example: `[0; 30; 60]`

Data Types: `double`

Output Arguments

`gim` — Gimbal
scalar | vector

Gimbal object attached to `parent`, returned as either a scalar or a vector.

When the `AutoSimulate` property of the satellite scenario is `false`, you can call the `gimbal` function only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `play` | `access` | `groundStation` | `satellite` | `conicalSensor` | `hide`

Topics
“Satellite Scenario Key Concepts” on page 2-65

Gimbal

Gimbal object belonging to satellite scenario

Description

The `Gimbal` defines a gimbal object belonging to a satellite scenario.

Creation

You can create a `Gimbal` object using the `gimbal` object function of the `Satellite` or `GroundStation`.

Properties

`Name` — Gimbal name

`"Gimbal idx"` (default) | string scalar | string vector | character vector | cell array of character vectors

You can set this property only when calling the `Gimbal` function. After you call the `gimbal` function, this property is read-only.

Gimbal name, specified as a name-value argument consisting of `'Name'` and a string scalar, string vector, character vector, or a cell array of character vectors.

- If you are adding only one Gimbal, specify `Name` as a string scalar or a character vector.
- If you are adding multiple gimbals, specify `Name` as a string scalar, character vector, string vector, or a cell array of character vectors. All gimbals that you add as a string scalar or a character vector are assigned the same specified name. The number of elements in the string vector or cell array of character vectors must equal the number of gimbals that you are adding. Each gimbal is assigned the corresponding name from the vector or cell array.

In the default value, *idx* is the ID assigned by the satellite scenario.

Data Types: `char` | `string`

`ID` — Gimbal ID assigned by simulator

real positive scalar

This property is set internally by the simulator and is read-only.

Gimbal ID assigned by the simulator, specified as a positive scalar.

`MountingLocation` — Mounting location with respect to parent

`[0; 0; 0]` (default) | three-element vector | matrix

Mounting location with respect to the parent object in meters, specified as a three-element vector or a matrix. The position vector is specified in the body frame of the input `parent`.

- If you are adding one gimbal, the `MountingLocation` property is a three-element vector. The elements specify the *x*, *y*, and *z* components of the Cartesian coordinates in the body frame of gimbal.
- If you are adding multiple gimbals, the `MountingLocation` property can be a three-element vector or a matrix. When specified as a vector, the same set of mounting locations are assigned to all specified gimbals. When specified as a matrix, `MountingLocation` must contain three rows and the same number of columns as the gimbals. The columns correspond to the mounting location of each specified gimbal and the rows correspond to the mounting location coordinates in the parent body frame.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify the `MountingLocation` property only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Data Types: `double`

`MountingAngles` — Mounting orientation with respect to parent object
`[0; 0; 0]` (default) | three-element row vector of positive numbers | matrix

Mounting orientation with respect to parent object in degrees, specified as a three-element row vector of positive numbers. The elements of the vector correspond to yaw, pitch, and roll, in that order. Yaw, pitch, and roll are positive rotations about the *z*-axis, intermediate *y*-axis, and intermediate *x*-axis of the parent.

- If you are adding one gimbal, the `MountingAngles` property is a three-element vector.
- If you are adding multiple gimbals the `MountingAngles` property can be a three-element vector or a matrix. When specified as a vector, the same set of mounting angles are assigned to all specified gimbals. When specified as a matrix, `MountingAngles` must contain three rows and the same number of columns as the gimbals. The columns correspond to the mounting angles of each specified gimbal and the rows correspond to the yaw, pitch, and roll angles in the parent body frame.

When the `AutoSimulate` property of the satellite scenario is `false`, you can modify the `MountingAngles` property only when `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Example: `[0; 30; 60]`

Data Types: `double`

`ConicalSensors` — Conical sensors
row vector of conical sensors

You can set this property only when calling the `conicalSensor`. After you call the `conicalSensor` function, this property is read-only.

Conical sensors attached to the Gimbal, specified as a row vector of conical sensors.

`CoordinateAxes` — Coordinate axes triad graphic object
`CoordinateAxes` object (default)

You can set this property only when calling `coordinateAxes`. After you call `coordinateAxes`, this property is read-only.

Coordinate axes triad graphic object, specified as `CoordinateAxes` object.

Object Functions

aer	Calculate azimuth angle, elevation angle, and range of another satellite or ground station in NED frame
conicalSensor	Add conical sensor to satellite scenario
gimbalAngles	Steering angles of gimbal
pointAt	Point satellite at target
CoordinateAxes	Coordinate axes triad graphic object

Examples

Calculate Maximum Revisit Time of Satellite

Create a satellite scenario with a start time of 15-June-2021 8:55:00 AM UTC and a stop time of five days later. Set the simulation sample time to `60` seconds.

```
startTime = datetime(2021,6,21,8,55,0);
stopTime = startTime + days(5);
sampleTime = 60;                                        % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc =
  satelliteScenario with properties:

         StartTime: 21-Jun-2021 08:55:00
          StopTime: 26-Jun-2021 08:55:00
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add a satellite to the scenario using Keplerian orbital elements.

```
semiMajorAxis = 7878137;                                                                    % me
eccentricity = 0;
inclination = 50;                                                                           % deg
rightAscensionOfAscendingNode = 0;                                                          % deg
argumentOfPeriapsis = 0;                                                                    % deg
trueAnomaly = 50;                                                                           % deg
sat = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly)
```

```
sat =
  Satellite with properties:

              Name:  Satellite 1
                ID:  1
     ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
            Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
       Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
          Receivers:  [1x0 satcom.satellitescenario.Receiver]
```

```
          Accesses:  [1x0 matlabshared.satellitescenario.Access]
        GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
             Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
    OrbitPropagator:  sgp4
        MarkerColor:  [0.059 1 1]
         MarkerSize:  6
          ShowLabel:  true
     LabelFontColor:  [1 1 1]
      LabelFontSize:  15
```

Add a ground station, which represents the location to be photographed, to the scenario.

```
gs = groundStation(sc,Name="Location to Photograph", ...
    Latitude=42.3001,Longitude=-71.3504)                % degrees
```

```
gs =
  GroundStation with properties:

                Name:  Location to Photograph
                  ID:  2
            Latitude:  42.3 degrees
           Longitude:  -71.35 degrees
            Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
          MarkerColor:  [1 0.4118 0.1608]
           MarkerSize:  6
            ShowLabel:  true
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
```

Add a gimbal to the satellite. You can steer this gimbal independently of the satellite.

```
g = gimbal(sat)
```

```
g =
  Gimbal with properties:

               Name:  Gimbal 3
                 ID:  3
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
```

Track the location to be photographed using the gimbal.

```
pointAt(g,gs);
```

Add a conical sensor to the gimbal. This sensor represents the camera. Set the field of view to 60 degrees.

```
camSensor = conicalSensor(g,MaxViewAngle=60)
```

```
camSensor =
  ConicalSensor with properties:

               Name:  Conical sensor 4
                 ID:  4
    MountingLocation:  [0; 0; 0] meters
      MountingAngles:  [0; 0; 0] degrees
        MaxViewAngle:  60 degrees
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         FieldOfView:  [0x0 matlabshared.satellitescenario.FieldOfView]
```

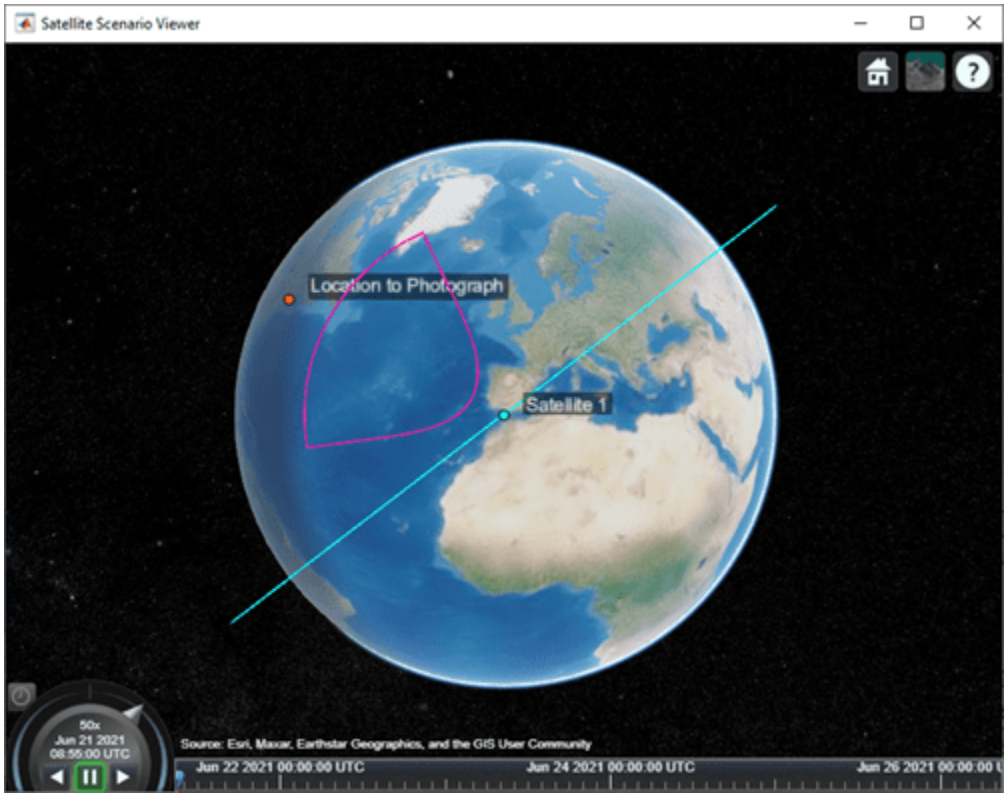
Add access analysis to the conical sensor between the camera and the location to be photographed.

```
ac = access(camSensor,gs)
```

```
ac =
  Access with properties:

    Sequence:  [4 2]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

Visualize the field of view of the camera by using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
fieldOfView(camSensor);
```



Determine the intervals during which the camera can see the geographical site.

```
t = accessIntervals(ac)
```

```
t=35×8 table
          Source                  Target             IntervalNumber         StartTime
    ________________    ________________________    ______________    ____________________

    "Conical sensor 4"    "Location to Photograph"           1          21-Jun-2021 10:38:00
    "Conical sensor 4"    "Location to Photograph"           2          21-Jun-2021 12:36:00
    "Conical sensor 4"    "Location to Photograph"           3          21-Jun-2021 14:37:00
    "Conical sensor 4"    "Location to Photograph"           4          21-Jun-2021 16:41:00
    "Conical sensor 4"    "Location to Photograph"           5          21-Jun-2021 18:44:00
    "Conical sensor 4"    "Location to Photograph"           6          21-Jun-2021 20:46:00
    "Conical sensor 4"    "Location to Photograph"           7          21-Jun-2021 22:50:00
    "Conical sensor 4"    "Location to Photograph"           8          22-Jun-2021 09:51:00
    "Conical sensor 4"    "Location to Photograph"           9          22-Jun-2021 11:46:00
    "Conical sensor 4"    "Location to Photograph"          10          22-Jun-2021 13:46:00
    "Conical sensor 4"    "Location to Photograph"          11          22-Jun-2021 15:50:00
    "Conical sensor 4"    "Location to Photograph"          12          22-Jun-2021 17:53:00
    "Conical sensor 4"    "Location to Photograph"          13          22-Jun-2021 19:55:00
    "Conical sensor 4"    "Location to Photograph"          14          22-Jun-2021 21:58:00
    "Conical sensor 4"    "Location to Photograph"          15          23-Jun-2021 10:56:00
    "Conical sensor 4"    "Location to Photograph"          16          23-Jun-2021 12:56:00
      ⋮
```

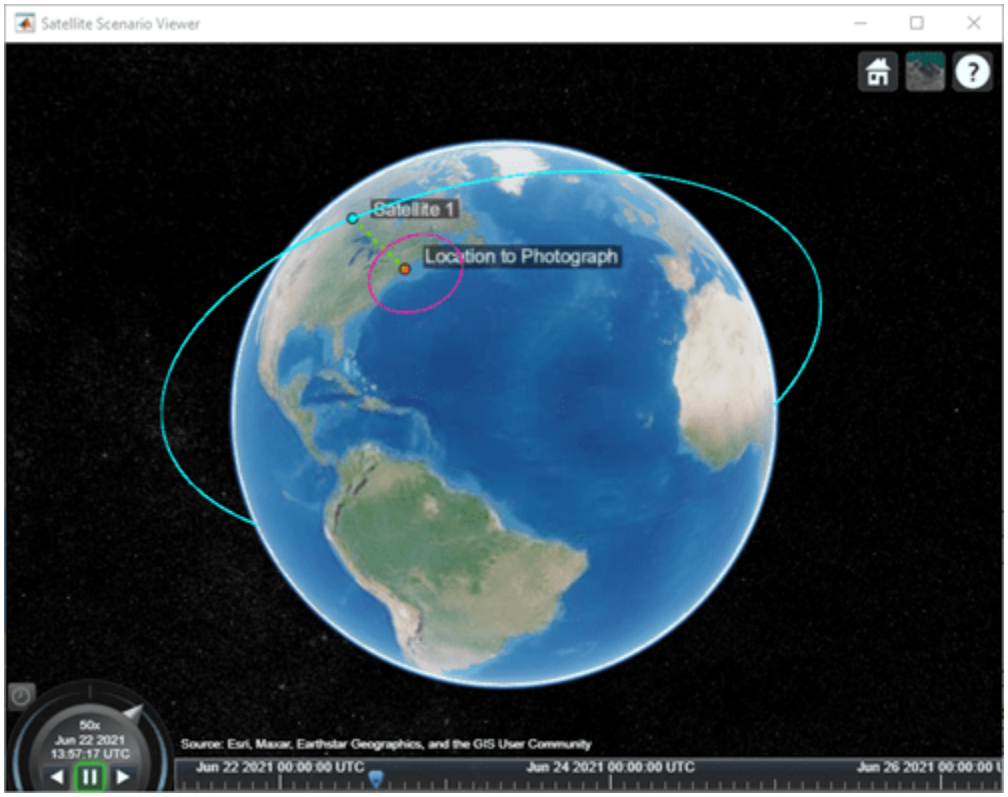
Calculate the maximum revisit time in hours.

```
startTimes = t.StartTime;
endTimes = t.EndTime;
revisitTimes = hours(startTimes(2:end) - endTimes(1:end-1));
maxRevisitTime = max(revisitTimes)                              % hours
```

```
maxRevisitTime = 12.6667
```

Visualize the revisit times that the camera photographs of the location.

```
play(sc);
```



Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `satellite` | `access` | `groundStation` | `conicalSensor`

Topics

“Satellite Scenario Key Concepts” on page 2-65

gimbalAngles

Steering angles of gimbal

Syntax

```
az = gimbalAngles(gimbal)
[az,el] = gimbalAngles(gimbal)
[ ___ ] = gimbalAngles(gimbal,timeIn)
[az,el,timeOut] = gimbalAngles(gimbal)
```

Description

`az = gimbalAngles(gimbal)` returns an array of gimbal azimuth `az` histories of the gimbals defined in the vector `gimbal`.

`[az,el] = gimbalAngles(gimbal)` returns an array of gimbal azimuth `az`and gimbal elevation `el` in the vector `gimbal`.

`[ ___ ] = gimbalAngles(gimbal,timeIn)` returns column vectors of gimbal azimuth and gimbal elevation of gimbals defined in the vector `gimbal` at the specified time `timeIn`, depending on the specified output arguments.

`[az,el,timeOut] = gimbalAngles(gimbal)` returns gimbal azimuth, gimbal elevation, and corresponding time in UTC.

Examples

Retrieve Gimbal Angles at Specific Time

Create a satellite scenario object.

```
startTime = datetime(2020,10,10);                    % 10 October 2020, 12:00 AM UTC
stopTime = datetime(2020,10,11);                     % 11 October 2020, 12:00 AM UTC
sampleTime = 60;                                     % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add a satellite to the scenario.

```
semiMajorAxis = 10000000;                            % meters
eccentricity = 0;
inclination = 10;                                    % degrees
rightAscensionOfAscendingNode = 0;                   % degrees
argumentOfPeriapsis = 0;                             % degrees
trueAnomaly = 0;                                     % degrees
sat = satellite(sc,semiMajorAxis,eccentricity, ...
    inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly);
```

Add a gimbal to the satellite.

```
g = gimbal(sat);
```

Point the gimbal at 0 degree latitude and longitude.

```
pointAt(g,[0; 0; 0]);
```

Get the gimbal azimuth and gimbal elevation corresponding to October 10, 2020, 20:54 PM UTC.

```
time = datetime(2020,10,10,20,54,0);
[az,el] = gimbalAngles(g,time)
```

```
az = -5.4268
```

```
el = 19.0368
```

Input Arguments

`gimbal` — Gimbal

scalar | vector

Gimbal object whose steering angle is being calculated, specified as either a scalar or a vector.

`timeIn` — Time at which output is calculated

`datetime` scalar

Time at which the output is calculated, specified as a `datetime` scalar. If no time zone is specified in `timeIn`, the time zone is assumed to be UTC.

Output Arguments

`az` — Gimbal azimuth

array

Gimbal azimuth histories of gimbals in degrees, returned as an array in the range [-180,180]. Each row corresponds to a gimbal in `gimbal`, and each column corresponds to a time sample. This represents the angle of rotation of the gimbal about its *y*-axis.

If `AutoSimulate` of the satellite scenario is `true`, `az` returns the gimbal azimuth history from `StartTime` to `StopTime`. Otherwise the gimbal azimuth history is returned from `StartTime` to `SimulationStatus`.

`el` — Gimbal elevation

array

Gimbal elevation histories of gimbals in degree, returned as an array in the range [0,180]. This represents the angle of rotation of the gimbal about its *y*-axis. Each row corresponds to a gimbal in `gimbal`, and each column corresponds to a time sample. This represents the angle of rotation of the gimbal about its x-axis.

If `AutoSimulate` of the satellite scenario is `true`, `el` returns the gimbal elevation history from `StartTime` to `StopTime`. Otherwise the gimbal elevation history is returned from `StartTime` to `SimulationStatus`.

`timeOut` — Time samples between start and stop time of scenario

scalar | vector

Time samples between start and stop time of the scenario, returned as a scalar or vector. If `az` and `el` histories are returned, `timeOut` is a row vector.

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `play` | `hide` | `groundStation` | `conicalSensor`

Topics
“Satellite Scenario Key Concepts” on page 2-65

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',value)
```

Description

Planetary Gravitational Potential Based on Planetary Rotation Rate

`[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. The function performs these calculations using `planet_coordinates`. You use centrifugal force in rotating or noninertial coordinate systems. Gravity centrifugal effect values are greatest at the equator of a planet.

Planetary Gravitational Potential Based on Specified Planetary Model

`[gx gy gz] = gravitycentrifugal(planet_coordinates,model)` implements the mathematical representation of centrifugal effect based on planetary gravitational potential for the planetary model, `model`.

Planetary Gravitational Potential Based on Custom Rotational Rate

`[gx gy gz] = gravitycentrifugal(planet_coordinates,'Custom',value)` implements the mathematical representation of centrifugal effect based on planetary gravitational potential using the custom rotational rate, `rotational_rate`.

Examples

Calculate Centrifugal Effect of Earth Gravity

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])
```

```
gx =
   -0.0339
```

Calculate Centrifugal Effect of Mars Gravity

Calculate the centrifugal effect of Mars gravity at 15,000 m over the equator and 11,000 m over the North Pole.

```
p  = [2412.648e3 -2412.648e3 0; 0 0 3376.2e3]
[gx, gy, gz] = gravitycentrifugal( p, 'Mars' )

p =
     2412648    -2412648           0
           0           0     3376200
gx =
    0.0121
         0
gy =
   -0.0121
         0
gz =
     0
     0
```

Calculate Precessing Centrifugal Effect of Gravity for Earth

Calculate the precessing centrifugal effect of gravity for Earth at 15,000 m over the equator and 11,000 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)

gx =
    0.0128
         0

gy =
   -0.0128
         0

gz =
     0
     0
```

Input Arguments

`planet_coordinates` — Planet-Centered Planet-Fixed coordinates

M-by-3 array

Planet-Centered Planet-Fixed coordinates, specified as an *M*-by-3 array in meters. The *z*-axis is positive toward the North Pole. If `model` is `'Earth'`, the planet coordinates are ECEF coordinates.

Data Types: `double`

`model` — Planetary model

`'Earth'` (default) | `'Mercury'` | `'Venus'` | `'Moon'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'`

Planetary model, specified as:

- `'Mercury'`
- `'Venus'`
- `'Earth'`
- `'Moon'`
- `'Mars'`
- `'Jupiter'`
- `'Saturn'`
- `'Uranus'`
- `'Neptune'`

Data Types: `double`

`'Custom',value` — Custom planetary model
name-value argument | scalar | planetary model

Custom planetary model, specified as a name-value argument, where the value specifies the planetary rotational rate in radians per second.

Example: `'Custom',Omega`

Data Types: `double`

Output Arguments

`gx` — Gravity values in *x*-axis
array

Gravity values in *x*-axis of the Planet-Centered Planet-Fixed coordinates, returned as an array of M gravity values in meters per second squared (m/s^2).

`gy` — Gravity values in *y*-axis
array

Gravity values in *y*-axis of the Planet-Centered Planet-Fixed coordinates, returned as an array of M gravity values in meters per second squared (m/s^2).

`gz` — Gravity values in *z*-axis
array

Gravity values in *z*-axis of the Planet-Centered Planet-Fixed coordinates, returned as an array of M gravity values in meters per second squared (m/s^2).

Version History

Introduced in R2010a

See Also

`gravitywgs84` | `gravitysphericalharmonic`

gravityspherical harmonic

Implement spherical harmonic representation of planetary gravity

Syntax

```
[gx gy gz] = gravityspherical harmonic(planet_coordinates)
[gx gy gz] = gravityspherical harmonic(planet_coordinates,degree)

[gx gy gz] = gravityspherical harmonic(planet_coordinates,model)
[gx gy gz] = gravityspherical harmonic(planet_coordinates,model,degree)
[gx gy gz] = gravityspherical harmonic(planet_coordinates,model,degree,action)

[gx gy gz] = gravityspherical harmonic(planet_coordinates,'Custom',degree,{
datafile dfreader},action)
```

Description

Default Planetary Model

`[gx gy gz] = gravityspherical harmonic(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. The function performs these calculations using `planet_coordinates`, an *M*-by-3 array of Planet-Centered Planet-Fixed coordinates.

`[gx gy gz] = gravityspherical harmonic(planet_coordinates,degree)` uses the degree and order that `degree` specifies.

Specified Planetary Model

`[gx gy gz] = gravityspherical harmonic(planet_coordinates,model)` implements the mathematical representation for the planetary model, `model`.

`[gx gy gz] = gravityspherical harmonic(planet_coordinates,model,degree)` uses the degree and order that `degree` specifies. `model` specifies the planetary model.

`[gx gy gz] = gravityspherical harmonic(planet_coordinates,model,degree,action)` uses the specified `action` when input is out of range.

Custom Planetary Model

`[gx gy gz] = gravityspherical harmonic(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`.

Examples

Calculate Gravity in *x*-Axis at Equator on Earth Surface

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 = gravityspherical harmonic([-6378.137e3 0 0])

gx =

    9.8143
```

Calculate Gravity at 25,000 Meters Over South Pole of Earth

Calculate the gravity at 25,000 m over the south pole of Earth. This example uses the 70 degree model of EGM96 with error actions.

```
[gx, gy, gz] = gravityspherical harmonic([0 0 -6381.751e3],'EGM96','Error')

gx =
     0

gy =
     0

gz =
    9.7552
```

Calculate Gravity at 15,000 Meters Over Equator and 11,000 Meters Over North Pole with GMM2B Mars Model

Calculate the gravity at 15,000 m over the equator and 11,000 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] = gravityspherical harmonic(p,'GMM2B',30,'Warning')

gx =
   -2.6085
         0

gy =
    2.6073
         0

gz =
    0.0000
   -3.6895
```

Calculate Gravity at 25,000 Meters Over South Pole and with 120th Order EIGEN-GL04C Earth Model

Calculate the gravity at 25,000 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] = gravityspherical harmonic( p,'EIGENGL04C', ...
120,'Warning')
```

```
gx =
     0

gy =
     0

gz =
    9.7552
```

Calculate Gravity at 15,000 Meters Over Equator and 11,000 Meters Over North Pole with Custom Planetary Model

Calculate the gravity at 15,000 m over the equator and 11,000 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')
```

```
gx =
   -2.6079
         0

gy =
    2.6067
         0

gz =
    0.0002
   -3.6902
```

Input Arguments

`planet_coordinates` — Planet coordinates

M-by-3 array

Planet coordinates, specified as an *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.

Data Types: `double`

`model` — Planetary model

`'EGM2008'` (default) | `'EGM96'` | `'LP100K'` | `'LP165P'` | `'GMM2B'` | `'Custom'` | `'EIGENGL04C'`

Planetary model, specified as one of these values.

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.

Planetary Model	Planet
`'GMM2B'`	Goddard Mars model 2B.

`'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.
	
	For more information, see “'Custom'” on page 4-0 .

`'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”.

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”.

Data Types: `char` | `string`

degree — Degree and order of harmonic gravity

scalar

Degree and order of harmonic gravity, specified as a scalar.

Planetary Model	Degree and Order
`'EGM2008'`	Maximum degree and order are 2159. Default degree and order are 120.
`'EGM96'`	Maximum degree and order are 360. Default degree and order are 70.
`'LP100K'`	Maximum degree and order are 100. Default degree and order are 60.
`'LP165P'`	Maximum degree and order are 165. Default degree and order are 60.
`'GMM2B'`	Maximum degree and order are 80. Default degree and order are 60.
`'Custom'`	Maximum degree is default degree and order. For more information, see “'Custom'” on page 4-0 .

Planetary Model	Degree and Order
`'EIGENGL04C'`	Maximum degree and order are 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”.

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”.

Data Types: `char` | `string`

`'Custom'` — Custom planetary model
`'Custom'`

Custom planetary model definitions, specified as `'Custom'`. Specify the planetary model definitions with a definitions data file and accompanying reader. For more information, see “datafile dfreader” on page 4-0 .

Data Types: `char` | `string`

`datafile dfreader` — Custom planetary model definitions file and reader
vector

Custom planetary model definitions file and reader, specified as a vector. `datafile` must contain these variables.

Variable	Description
Re	Scalar of planet equatorial radius in meters (m)
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*

To read `datafile`, specify a MATLAB function in the `dfreader` parameter. 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.

Example: `{'GMM2BC80_SHA.txt' @astReadSHAFile}`

Data Types: `double`

`action` — Action
`'Warning'` (default) | `'Error'` | `'None'`

Action for out-of-range input, specified as:

- `'Error'` — Displays warning and indicates that the input is out of range.
- `'Warning'` — Displays error and indicates that the input is out of range.
- `'None'` — Does not display warning or error.

Data Types: `char` | `string`

Output Arguments

gx — Gravity values in *x*-axis
array

Gravity values in *x*-axis of Planet-Centered Planet-Fixed coordinates, returned as an array of *M* gravity values in meters per second squared (m/s^2).

gy — Gravity values in *y*-axis
array

Gravity values in *y*-axis of Planet-Centered Planet-Fixed coordinates, returned as an array of *M* gravity values in meters per second squared (m/s^2).

gz — Gravity values in *z*-axis
array

Gravity values in *z*-axis of Planet-Centered Planet-Fixed coordinates, returned as an array of *M* gravity values in meters per second squared (m/s^2).

Limitations

- The function excludes the centrifugal effects of planetary rotation, and the effects of a precessing reference frame.
- The 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.

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”.
- 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”.

Version History

Introduced in R2010a

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*. Houston: NASA Lyndon B. Johnson Space Center, February 1993.

[2] Vallado, David A. *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

[3] Defense Mapping Agency. *Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems*. TR 8350.2, 2nd ed. Fairfax, VA: DMA, September 1, 1991.

[4] Konopliv, A. S., S. W. Asmar, E. Carranza, W. L. Sjogen, and D. N. Yuan. "Recent Gravity Models as a Result of the Lunar Prospector Mission, Icarus" 150, no. 1 (2001): 1–18.

[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* 106, no. E10 (October 25, 2001): 23359–23376.

[6] Kenyon S., J. Factor, N. Pavlis, and S. Holmes. "Towards the Next Earth Gravitational Model." Paper presented at the 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." Paper presented at the 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." Paper presented at the IAG International Symposium on Gravity, Geoid & Earth Observation, Chania, Greece, June 23–27, 2008.

[9] Förste, C., Flechtner et al, "A Mean Global Gravity Field Model From the Combination of Satellite Mission and Altimetry/Gravmetry Surface Data - EIGEN-GL04C." *Geophysical Research Abstracts* 8, 03462, 2006.

[10] Hill, K. A. "Autonomous Navigation in Libration Point Orbits." PhD diss. University of Colorado, Boulder, 2007.

[11] Colombo, Oscar L. *Numerical Methods for Harmonic Analysis on the Sphere*. Report no. 310. Columbus: Department of Geodetic Science at Ohio State University, 1981.

[12] Colombo, Oscar L. "The Global Mapping of Gravity with Two Satellites." Netherlands Geodetic Commission 7, no 3, Delft, The Netherlands, 1984., Reports of the Department of Geodetic Science. Report No. 310. Columbus: Ohio State University, March 1981.

[13] Jones, Brandon A. "Efficient Models for the Evaluation and Estimation of the Gravity Field." PhD diss. University of Colorado, Boulder, 2010.

[14] *Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 1991*.

See Also

`gravitywgs84` | `gravitycentrifugal` | `gravityzonal`

gravitywgs84

Implement 1984 World Geodetic System (WGS84) representation of Earth 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 altitude `h` and geodetic latitude `lat`.

`g = gravitywgs84(h,lat,lon,method,[noatm,nocent,prec,jd],action)` uses both latitude and longitude, as well as other optional inputs. `method` must be `'CloseApprox'`, `'Exact'`, or `TaylorSeries`.

`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 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 surface.

Examples

Normal Gravity with Taylor Series

Calculate the normal gravity at 5000 meters and 55 degrees latitude using the Taylor Series approximation method and return errors for out-of-range inputs:

```
g = gravitywgs84(5000,55,'TaylorSeries','Error')
```

```
g =

    9.7997
```

Normal Gravity with Close Approximation

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 precession. A warning, enabled by default, is returned for out-of-range inputs.

```
g = gravitywgs84(15000,45,120,'CloseApprox')
```

```
g =

    9.7601
```

Normal and Tangential Gravity

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 precession. A warning, enabled by default, is returned for out-of-range inputs.

```
[gn, gt] = gravitywgs84(1000,0,20,'Exact')
```

```
gn =
    9.7772

gt =
     0
```

Normal and Tangential Gravity with Latitude, Longitude, and Exact Method

Calculate the normal and tangential gravity at 1000 meters, 0 degrees latitude, and 20 degrees longitude, and the normal and tangential gravity at 11,000 meters, 30 degrees latitude, and 50 degrees longitude using the Exact method with atmosphere, centrifugal effects, and no precession. Do not return 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
```

Normal Gravity with Latitude, Longitude, and Close Approximation Method

Calculate the normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude, and the normal gravity at 5000 meters, 55 degrees latitude, and 100 degrees longitude using the Close Approximation method with atmosphere, no centrifugal effects, and no precession. A warning, enabled by default, is returned 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
```

Normal and Tangential Gravity with Latitude, Longitude, Julian date, and Exact Method

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 precession at Julian date 2451545. Return warnings for out-of-range inputs.

```
[gn, gt] = gravitywgs84(1000,0,20,'Exact', ...
              [false false true 2451545],'Warning')
```

```
gn =

    9.7772


gt =

     0
```

Normal Gravity with Julian Date, No Atmosphere, and Close Method

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 precession at Julian date 2451545. Return errors for out-of-range inputs.

```
g = gravitywgs84(15000,45,120,'CloseApprox', ...
        [true false true 2451545],'Error')
```

```
g =

    9.7601
```

Normal Gravity with Julian Date, No Atmosphere, and Exact Method

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 precession at Julian date 2451545. Return errors for out-of-range inputs.

```
gn = gravitywgs84(15000,45,120,'Exact', ...
        [true false true 2451545],'Error')
```

```
gn =

    9.7601
```

Input Arguments

h — Altitudes

array

Altitudes, specified as an array of *m* values, with respect to the WGS84 ellipsoid, in meters.

Data Types: `double`

`lat` — Geodetic latitudes
array

Geodetic latitudes, specified as an array of *m* latitudes in degrees, where the north latitude is positive, and south latitude is negative.

Data Types: `double`

`lon` — Geodetic longitudes
array

Geodetic longitudes, specified as an array of *m* longitudes, in degrees, where the east longitude is positive, and west longitude is negative.

Only use this input when you specify `method` as `'CloseApprox'` or `'Exact'`.

Data Types: `double`

`method` — Gravity calculation method
`'TaylorSeries'` (default) | `'CloseApprox'` | `'Exact'`

Gravity calculation method, specified as:

- `'TaylorSeries'` — Medium gravity precision
- `'CloseApprox'` — Close gravity precision
- `'Exact'` — Exact gravity precision

For more information, see “Limitations” on page 4-620.

Data Types: `double`

`noatm` — Earth atmosphere
`false` (default) | `true`

Exclude or include Earth atmosphere, specified as `true` or `false`:

- `false` — Include the mass of the atmosphere in the value for the Earth gravitational field.
- `true` — Exclude the mass of the atmosphere in the value for the Earth gravitational field.

Only use this input when you specify `method` as `'CloseApprox'` or `'Exact'`.

Data Types: `logical`

`nocent` — Atmosphere
`false` (default) | `true`

Remove or include centrifugal effects, specified as:

- `false` — Calculate gravity including the centrifugal force resulting from the Earth angular velocity; the centrifugal contribution is included.
- `true` — Calculate gravity based on pure attraction resulting from the normal gravitational potential; the centrifugal contribution is excluded.

Only use this input when you specify `method` as `'CloseApprox'` or `'Exact'`.

Data Types: `logical`

`prec` — Atmosphere
`false` (default) | `true`

Include or exclude a precession reference frame.

- `false` — Calculate gravity using the angular velocity of the Earth as the value of the standard Earth rotating at a constant angular velocity.
- `true` — Calculate gravity using the International Astronomical Union (IAU) value of the Earth angular velocity and the precession rate in right ascension. For the precession rate in right ascension, this option calculates Julian centuries from Epoch J2000.0 using the Julian date, `jd`.

Only use this input when you specify `method` as `'CloseApprox'` or `'Exact'`.

Data Types: `logical`

`jd` — Julian date
0 (default) | `scalar`

Julian date, specified as a scalar, to calculate Julian centuries from Epoch J2000.0. The `prec` option uses this option to calculate Julian centuries from Epoch J2000.0 for the precession rate in right ascension.

Only use this input when you specify `method` as `'CloseApprox'` or `'Exact'`.

Data Types: `double`

`action` — Action
`'Warning'` (default) | `'Error'` | `'None'`

Action for out-of-range input, specified as:

- Warning — Displays warning and indicates that the input is out-of-range.
- Error — Displays error and indicates that the input is out-of-range.
- None — Does not display warning or error.

Data Types: `char` | `string`

Output Arguments

`g` — Gravity values normal to Earth surface at specific longitude and latitude
array

Gravity values normal to the Earth surface at specific longitude and latitude, returned as an array of *m* gravity values in the direction normal to the Earth surface. A positive value indicates a downward direction.

`gn` — Total gravity values normal to Earth surface
array

Total gravity values normal to the Earth surface at a specific `lat` `lon` location, returned as an array of `m` gravity values. A positive value indicates a downward direction.

Dependencies

This output is available only with `method` specified as `'Exact'`. When `method` is `'TaylorSeries'` or `'CloseApprox'`, the function assumes that `gn` equals `g`.

`gt` — Gravity values tangential to Earth surface

array

An array of *m* gravity values in the direction tangential to the Earth surface at a specific `lat lon` location. A positive value indicates a northward direction.

Dependencies

This output is available only with `method` specified as `'Exact'`.

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.
- Limit use of the WGS84 Taylor Series model 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.
- Limit use of the WGS84 Close Approximation model to a geodetic height of 20,000.0 meters (approximately 65,620.0 feet). Below this height, the function gives results with submicrogal precision.
- To predict and determine a satellite orbit with high accuracy, instead of the `gravitywgs84` function, use the `gravityspericalharmonics` function with the `EGM96` option and degree and order 70.

Version History

Introduced in R2006b

References

[1] National Imagery and Mapping Agency (NIMA). “Department of Defense World Geodetic System 1984: Its Definition, and Relationship with Local Geodetic Systems, TR8350.2, Third Ed.” Department of Defense, Washington, DC: 1997.

gravityzonal

Implement zonal harmonic representation of planetary gravity

Syntax

```
[gravityXcoord gravityYcoord,gravityZcoord] = gravityzonal(planetCoord)
[gravityXcoord,gravityYcoord,gravityZcoord] = gravityzonal(planetCoord,
action)

[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',Re,planetaryGravitional,zonalHarmonicCoeff)
[gravityXcoord,gravityYcoord,gravityZcoord] = gravityzonal(planetCoord,
'Custom',Re,planetaryGravitional,zonalHarmonicCoeff,action)
```

Description

Default Degree of Harmonic and Planetary Model

`[gravityXcoord gravityYcoord,gravityZcoord] = gravityzonal(planetCoord)` implements the mathematical representation of zonal harmonic planetary gravity based on planetary gravitational potential. The function 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.

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.

`[gravityXcoord,gravityYcoord,gravityZcoord] = gravityzonal(planetCoord, action)` specifies the action for out-of-range input.

Degree of Harmonic Model and Planetary Model

`[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.

Custom Planetary Model

`[gravityXcoord,gravityYcoord,gravityZcoord] = gravityzonal(planetCoord, 'Custom',Re,planetaryGravitional,zonalHarmonicCoeff)` uses the equatorial radius, planetary gravitational parameter, and zonal harmonic coefficients for the custom planetary model.

`[gravityXcoord,gravityYcoord,gravityZcoord] = gravityzonal(planetCoord, 'Custom',Re,planetaryGravitional,zonalHarmonicCoeff,action)` specifies the action for out-of-range input.

Examples

Calculate Gravity in the *x*-Axis at Equator on the Surface of Earth Using Fourth Degree Model

Calculate the gravity in *x*-axis at the equator on surface of Earth using the fourth degree model with no warning actions.

```
gx = gravityzonal([-6378.1363e3 0 0])
```

```
gx =
    9.8142
```

Calculate Gravity Using Close Approximation Method at 100 Meters over Geographic South Pole of Earth

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 Gravity at 15,000 Meters over Equator and 11,000 Meters Over Geographic North Pole and Mars Model

Calculate the gravity at 15,000 m over the equator and 11,000 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')
```

```
gx =
   -2.6224
         0
```

```
gy =
    2.6224
         0

gz =
         0
   -3.7542
```

Calculate Gravity at 15,000 Meters Over Equator and 11,000 Meters Over Geographic North Pole with Custom Planetary Model

Calculate the gravity at 15,000 m over the equator and 11,000 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')
```

```
gx =
   -2.6090
         0

gy =
    2.6090
         0

gz =
    0.0002
   -3.7352
```

Input Arguments

`planetCoord` — Planet-centered planet-fixed coordinates

m-by-3 matrix

Planet-centered planet-fixed coordinates from center of planet, specified as an *m*-by-3 matrix in meters. If `planetModel` has a value of `'Earth'`, this matrix contains Earth-centered Earth-fixed (ECEF) coordinates.

Data Types: `double`

`planetModel` — Planetary model

`'Earth'` (default) | `'Mercury'` | `'Venus'` | `'Moon'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Custom'`

Planetary model, specified as:

- `'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.

Data Types: double

degreeGravityModel — Degree of harmonic model
scalar

Degree of harmonic model, specified as a scalar of one of these values.

Degree	Description	Default WhenplanetModel Is
4	Fourth degree, J4	• 'Earth • 'Jupiter' • 'Saturn' • 'Custom'
2	Second degree, J2	• 'Mercury' • 'Venus' • 'Moon' • 'Uranus' • 'Neptune'
3	Third degree, J3	'Mars'

Data Types: double

Re — Equatorial radius
scalar

Equatorial radius, specified as a scalar in meters.

Data Types: double

planetaryGravitional — Planetary gravitational parameter
scalar

Planetary gravitational parameter, specified as a scalar in meters cubed per second squared.

Data Types: double

zonalHarmonicCoeff — Zonal harmonic coefficients
3-element array

Zonal harmonic coefficients to calculate zonal harmonics planetary gravity, specified as a 3-element array.

Data Types: `double`

`action` — Action
`'None'` (default) | `'Error'` | `'Warning'`

Action for out-of-range input, specified as:

- `'Error'` — Displays warning and indicates that the input is out of range.
- `'Warning'` — Displays error and indicates that the input is out of range.
- `'None'` — Does not display warning or error.

Data Types: `char` | `string`

Output Arguments

`gravityXcoord` — Gravity values in the *x*-axis
array

Gravity values in the *x*-axis, returned as an array of *m* gravity values of the planet-centered planet-fixed coordinates in meters per second squared.

`gravityYcoord` — Gravity values in the *y*-axis
array

Gravity values in the *y*-axis, returned as an array of *m* gravity values of the planet-centered planet-fixed coordinates in meters per second squared.

`gravityZcoord` — Gravity values in the *z*-axis
array

Gravity values in the *z*-axis, returned as an array of *m* gravity values of the planet-centered planet-fixed coordinates in meters per second squared.

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]

Planet	Equatorial Radius (Re) in Meters	Gravitational Parameter (GM) in m^3/s^2	Zonal Harmonic Coefficients (J Values)
Uranus	25559.e3	5.794e15	0.012
Venus	6052.0e3	3.257e14	0.000027

Version History

Introduced in R2009b

References

[1] Vallado, David A. *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

[2] Fortescue, Peter, Graham Swinerd, and John Stark, eds. *Spacecraft Systems Engineering*, 3rd ed. West Sussex: Wiley & Sons, 2003.

[3] Tewari, Ashish. *Atmospheric and Space Flight Dynamics Modeling and Simulation with MATLAB and Simulink*. Boston. Birkhäuser, 2007.

See Also

`gravitywgs84` | `geoidheight`

greenwichsrt

Greenwich mean and apparent sidereal times

Note `greenwichsrt` is not recommended. For replacement functionality, see the `siderealTime` function.

Syntax

```
[thGMST,thGAST] = greenwichSRT(utcJD)
[thGMST,thGAST] = greenwichSRT(utcJD,dUT1,dAT)
```

Description

`[thGMST,thGAST] = greenwichSRT(utcJD)` calculates Greenwich mean and apparent sidereal times at a specific Coordinated Universal Time (UTC). Mean sidereal time accounts only for secular motion (precession). Apparent sidereal time includes secular and periodic contributions.

`[thGMST,thGAST] = greenwichSRT(utcJD,dUT1,dAT)`calculates Greenwich mean and apparent sidereal times at a higher precision using Earth orientation parameters.

Examples

Calculate Greenwich Sidereal Times

Calculate Greenwich sidereal times at 12:00 on January 4, 2019.

```
jd = juliandate([2019 1 4 12 0 0]);
[thGMST, thGAST] = greenwichSRT(jd)
```

```
thGMST =
  283.8103

thGAST =
  283.8065
```

Input Arguments

`utcJD` — UTC as Julian date

scalar

Coordinated Universal Time (UTC) as a Julian date, specified as a scalar.

Tip To calculate the Julian date for a particular date, use the `juliandate` function.

Data Types: `double`

dUT1 — Difference between CUT and UT1
`0` (default) | scalar

Difference between the Coordinated Universal Time (UTC) and Universal Time (UT1), specified as a scalar, in seconds.

dAT — Difference between TAI and UTC
`0` (default) | scalar

Difference between International Atomic Time (TAI) and Coordinated Universal Time (UTC), specified as a scalar, in seconds.

Output Arguments

thGMST — Greenwich mean sidereal time
scalar

Greenwich mean sidereal time, specified as a scalar, in seconds.

thGAST — Greenwich apparent sidereal time
scalar

Greenwich apparent sidereal time, specified as a scalar, in seconds.

Limitations

This function requires the Mapping Toolbox™ license.

Version History

Introduced in R2019a

References

[1] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. alg. 1 and eqs. 1-63. New York: McGraw-Hill, 1997.

See Also

`ecef2eci` | `eci2ecef` | `dcmeci2ecef` | CubeSat Vehicle

groundStation

Package: `matlabshared.satellitescenario`

Add ground station to satellite scenario

Syntax

```
groundStation(scenario)
groundStation(scenario,lat,lon)
groundStation( ___ ,Name,Value)
gs = groundStation( ___ )
```

Description

`groundStation(scenario)` adds a default `GroundStation` object to the specified satellite scenario.

`groundStation(scenario,lat,lon)` sets the Latitude and Longitude properties of the ground station to `lat` and `lon`, respectively. `lat` and `lon` must be of the same length. This length specifies the number of ground stations that the function adds to the input scenario. Together, `lat` and `lon` indicate the locations of the ground stations.

`groundStation( ___ ,Name,Value)` sets options using one or more name-value arguments in addition to any input argument combination from previous syntaxes. For example, `'MinElevationAngle',10` specifies a minimum elevation angle of 10 degrees.

`gs = groundStation( ___ )` returns a vector of handles to the added ground stations. Specify any input argument combination from previous syntaxes.

Examples

Add Ground Stations to Scenario and Visualize Access Intervals

Create a satellite scenario and add ground stations from latitudes and longitudes.

```
startTime = datetime(2020,5,1,11,36,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
lat = 10;
lon = -30;
gs = groundStation(sc,lat,lon);
```

Add satellites using Keplerian elements.

```
semiMajorAxis = 10000000;
eccentricity = 0;
inclination = 10;
rightAscensionOfAscendingNode = 0;
argumentOfPeriapsis = 0;
```

```
trueAnomaly = 0;
sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
        rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly);
```

Add access analysis to the scenario and obtain the table of intervals of access between the satellite and the ground station.

```
ac = access(sat,gs);
intvls = accessIntervals(ac)
```

```
intvls=8×8 table
       Source              Target          IntervalNumber         StartTime                EndTi
    ____________    ________________    ______________    ____________________    ___________

    "Satellite 2"    "Ground station 1"          1           01-May-2020 11:36:00    01-May-2020
    "Satellite 2"    "Ground station 1"          2           01-May-2020 14:20:00    01-May-2020
    "Satellite 2"    "Ground station 1"          3           01-May-2020 17:27:00    01-May-2020
    "Satellite 2"    "Ground station 1"          4           01-May-2020 20:34:00    01-May-2020
    "Satellite 2"    "Ground station 1"          5           01-May-2020 23:41:00    02-May-2020
    "Satellite 2"    "Ground station 1"          6           02-May-2020 02:50:00    02-May-2020
    "Satellite 2"    "Ground station 1"          7           02-May-2020 05:59:00    02-May-2020
    "Satellite 2"    "Ground station 1"          8           02-May-2020 09:06:00    02-May-2020
```

Play the scenario to visualize the ground stations.

```
play(sc)
```

Input Arguments

`scenario` — Satellite scenario

`satelliteScenario` object

Satellite scenario, specified as a `satelliteScenario` object.

`lat`, `lon` — Latitude and longitude

real-valued scalar | real-valued vector

Latitude and longitude of the ground station, specified as a real-valued scalar or real-valued vector.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MinElevationAngle',10` specifies a minimum elevation angle of 10 degrees.

`Viewer` — Satellite scenario viewer

vector of `satelliteScenarioViewer` objects (default) | scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Satellite scenario viewer, specified as a scalar, vector, or array of `satelliteScenarioViewer` objects. If the `AutoSimulate` property of the scenario is `false`, adding a satellite to the scenario disables any previously available timeline and playback widgets.

`Name` — groundStation name

"groundStation *idx*" (default) | string scalar | string vector | character vector | cell array of character vectors

You can set this property only when calling the `groundStation` function. After you call `groundStation` function, this property is read-only.

groundStation name, specified as a comma-separated pair consisting of `'Name'` and a string scalar, string vector, character vector or a cell array of character vectors.

- If only one groundStation is added, specify `Name` as a string scalar or a character vector.
- If multiple groundStations are added, specify `Name` as a string scalar, character vector, string vector or a cell array of character vectors. All groundStations added as a string scalar or a character vector are assigned the same specified name. The number of elements in the string vector or cell array of character vector must equal the number of groundStations being added. Each groundStation is assigned the corresponding name from the vector or cell array.

In the default value, *idx* is the ID assigned by satellite scenario.

Data Types: `char` | `string`

`Latitude` — Geodetic latitude of ground stations

`42.3001` (default) | scalar | row vector

You can set this property only when calling groundStation. After you call groundStation, this property is read-only.

Geodetic latitude of ground stations, specified as a scalar. Values must be in the range [-90, 90].

- If you add only one ground station, specify Latitude as a scalar double.
- If you add multiple ground stations, specify Latitude as a vector double whose length is equal to the number of ground stations being added.

When latitude and longitude are specified as `lat, lon` inputs to groundStation, Latitude specified as a name-value argument takes precedence.

Data Types: `double`

`Longitude` — Geodetic longitude of ground stations
`-71.3504` (default) | scalar | row vector

You can set this property only when calling groundStation. After you call groundStation, this property is read-only.

Geodetic longitude of ground stations, specified as a scalar or a vector. Values must be in the range [-180, 180].

- If you add only one ground station, specify longitude as a scalar.
- If you add multiple ground stations, specify longitude as a vector whose length is equal to the number of ground stations being added.

When longitude and longitude are specified as `lat, lon` inputs to groundStation, longitude specified as a name-value argument takes precedence.

Data Types: `double`

`Altitude` — Altitude of ground station
`0` m (default) | scalar | vector

You can set this property only when calling groundStation. After you call groundStation, this property is read-only.

Altitude of ground stations, specified as a scalar or a vector.

- If you specify `Altitude` as a scalar, the value is assigned to each ground station in the groundStation.
- If you specify `Altitude` as a vector, the vector length must be equal to the number of ground stations in the groundStation.

When latitude and longitude are specified as `lat, lon` inputs to groundStation, Latitude specified as a name-value argument takes precedence.

Data Types: `double`

`MinElevationAngle` — Minimum elevation angle
`0` (default) | scalar | vector

Minimum elevation angle of a satellite for the satellite to be visible from the ground station, and for the ground station to be visible from the satellite in degrees, specified as a scalar or row vector. Values must be in the range [–90, 90]. For access and link closure to be possible, the elevation angle must be at least equal to the value specified in `MinElevationAngle`.

- If you specify `MinElevationAngle` as a scalar, the value is assigned to each ground station in the groundStation.
- If you specify `MinElevationAngle` as a vector, the vector length must be equal to the number of ground stations in the groundStation.

When the `AutoSimulate` property of the satellite scenario is `false`, `MinElevationAngle` property can be modified while the `SimulationStatus` is `NotStarted` or `InProgress`.

Data Types: `double`

Output Arguments

`gs` — Ground station in scenario

`GroundStation` object

Ground station in the scenario, returned as a `GroundStation` object belonging to the satellite scenario specified by the input `scenario`.

You can modify the `GroundStation` object by changing its property values. The name-value arguments used when calling this function correspond to property names.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `satellite` | `access`

Topics

“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

GroundStation

Ground station object belonging to satellite scenario

Description

The `GroundStation` object defines a ground station object belonging to a satellite scenario.

Creation

You can create `GroundStation` object using the `groundStation` object function of the `satelliteScenario` object.

Properties

`Name` — GroundStation name
`"GroundStation idx"` (default) | string scalar | string vector | character vector | cell array of character vectors

You can set this property only when calling the `groundStation` function. After you call `groundStation` function, this property is read-only.

GroundStation name, specified as a comma-separated pair consisting of `'Name'` and a string scalar, string vector, character vector or a cell array of character vectors.

- If only one GroundStation is added, specify `Name` as a string scalar or a character vector.
- If multiple GroundStations are added, specify `Name` as a string scalar, character vector, string vector or a cell array of character vectors. All GroundStations added as a string scalar or a character vector are assigned the same specified name. The number of elements in the string vector or cell array of character vector must equal the number of GroundStations being added. Each GroundStation is assigned the corresponding name from the vector or cell array.

In the default value, *idx* is the ID assigned by satellite scenario.

Data Types: `char` | `string`

`ID` — GroundStation ID assigned by simulator
real positive scalar

This property is set internally by the simulator and is read-only.

GroundStation ID assigned by the simulator, specified as a positive scalar.

`Latitude` — Geodetic latitude of ground stations
`42.3001` (default) | scalar | row vector

You can set this property only when calling GroundStation. After you call GroundStation, this property is read-only.

Geodetic latitude of ground stations, specified as a scalar. Values must be in the range [-90, 90].

- If you add only one ground station, specify Latitude as a scalar double.
- If you add multiple ground stations, specify Latitude as a vector double whose length is equal to the number of ground stations being added.

When latitude and longitude are specified as `lat, lon` inputs to GroundStation, Latitude specified as a name-value argument takes precedence.

Data Types: `double`

`Longitude` — Geodetic longitude of ground stations
`-71.3504` (default) | scalar | row vector

You can set this property only when calling GroundStation. After you call GroundStation, this property is read-only.

Geodetic longitude of ground stations, specified as a scalar or a vector. Values must be in the range [-180, 180].

- If you add only one ground station, specify longitude as a scalar.
- If you add multiple ground stations, specify longitude as a vector whose length is equal to the number of ground stations being added.

When longitude and longitude are specified as `lat, lon` inputs to GroundStation, longitude specified as a name-value argument takes precedence.

Data Types: `double`

`Altitude` — Altitude of ground station
`0` m (default) | scalar | vector

You can set this property only when calling GroundStation. After you call GroundStation, this property is read-only.

Altitude of ground stations, specified as a scalar or a vector.

- If you specify `Altitude` as a scalar, the value is assigned to each ground station in the GroundStation.
- If you specify `Altitude` as a vector, the vector length must be equal to the number of ground stations in the GroundStation.

When latitude and longitude are specified as `lat, lon` inputs to GroundStation, Latitude specified as a name-value argument takes precedence.

Data Types: `double`

`CoordinateAxes` — Coordinate axes triad graphic object
`CoordinateAxes` object (default)

You can set this property only when calling `coordinateAxes`. After you call `coordinateAxes`, this property is read-only.

Coordinate axes triad graphic object, specified as `CoordinateAxes` object.

`MinElevationAngle` — Minimum elevation angle
`0` (default) | scalar | vector

Minimum elevation angle of a satellite for the satellite to be visible from the ground station, and for the ground station to be visible from the satellite in degrees, specified as a scalar or row vector. Values must be in the range [–90, 90]. For access and link closure to be possible, the elevation angle must be at least equal to the value specified in `MinElevationAngle`.

- If you specify `MinElevationAngle` as a scalar, the value is assigned to each ground station in the GroundStation.
- If you specify `MinElevationAngle` as a vector, the vector length must be equal to the number of ground stations in the GroundStation.

When the `AutoSimulate` property of the satellite scenario is `false`, `MinElevationAngle` property can be modified while the `SimulationStatus` is `NotStarted` or `InProgress`.

Data Types: `double`

`Accesses` — Access analysis objects
row vector of `Access` objects

You can set this property only when calling `GroundStation`. After you call `GroundStation`, this property is read-only.

Access analysis objects, specified as a row vector of `Access` objects.

`ConicalSensors` — Conical sensors
row vector of conical sensors

You can set this property only when calling the `conicalSensor`. After you call the `conicalSensor` function, this property is read-only.

Conical sensors attached to the GroundStation, specified as a row vector of conical sensors.

`Gimbals` — Gimbals
row vector of `Gimbal` objects

You can set this property only when calling `gimbal`. After you call `gimbal`, this property is read-only.

Gimbals attached to the GroundStation, specified as the comma-separated pair consisting of `'Gimbals'` and a row vector of `Gimbal` objects.

`MarkerColor` — Color of marker
`[1 0 0]` (default) | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Color of the marker, specified as a comma-separated pair consisting of `'MarkerColor'` and either an RGB triplet or a string or character vector of a color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	
`"none"`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	

`MarkerSize` — Size of marker
`10` (default) | positive scalar less than 30

Size of the marker, specified as a comma-separated pair consisting of `'MarkerSize'` and a real positive scalar less than 30. The unit is in pixels.

`ShowLabel` — State of GroundStation label visibility
`true` or `1` (default) | `false` or `0`

State of GroundStation label visibility, specified as a comma-separated pair consisting of `'ShowLabel'` and numerical or logical value of `1` (`true`) or `0` (`false`).

Data Types: `logical`

`LabelFontSize` — Font size of GroundStation label
`15` (default) | positive scalar in the range [6 30]

Font size of the GroundStation label, specified as a comma-separated pair consisting of `'LabelFontSize'` and a positive scalar in the range [6 30].

`LabelFontColor` — Font color of GroundStation label
`[1,0,0]` (default) | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Font color of the GroundStationlabel, specified as a comma-separated pair consisting of `'LabelFontColor'` and either an RGB triplet or a string or character vector of a color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	
`"none"`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	

Object Functions

access	Add access analysis objects to satellite scenario
conicalSensor	Add conical sensor to satellite scenario
gimbal	Add gimbal to satellite or ground station
coordinateAxes	Visualize coordinate axes triad of satellite scenario assets
show	Show object in satellite scenario viewer
aer	Calculate azimuth angle, elevation angle, and range of another satellite or ground station in NED frame
hide	Hide satellite scenario entity from viewer

Examples

Add Ground Stations to Scenario and Visualize Access Intervals

Create a satellite scenario and add ground stations from latitudes and longitudes.

```
startTime = datetime(2020,5,1,11,36,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
lat = 10;
lon = -30;
gs = groundStation(sc,lat,lon);
```

Add satellites using Keplerian elements.

```
semiMajorAxis = 10000000;
eccentricity = 0;
inclination = 10;
rightAscensionOfAscendingNode = 0;
argumentOfPeriapsis = 0;
trueAnomaly = 0;
sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
        rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly);
```

Add access analysis to the scenario and obtain the table of intervals of access between the satellite and the ground station.

```
ac = access(sat,gs);
intvls = accessIntervals(ac)
```

```
intvls=8×8 table
       Source             Target          IntervalNumber         StartTime              EndTi
    ______________    ___________________    ______________    ____________________    _____

    "Satellite 2"    "Ground station 1"          1           01-May-2020 11:36:00    01-May-2020
    "Satellite 2"    "Ground station 1"          2           01-May-2020 14:20:00    01-May-2020
    "Satellite 2"    "Ground station 1"          3           01-May-2020 17:27:00    01-May-2020
    "Satellite 2"    "Ground station 1"          4           01-May-2020 20:34:00    01-May-2020
    "Satellite 2"    "Ground station 1"          5           01-May-2020 23:41:00    02-May-2020
    "Satellite 2"    "Ground station 1"          6           02-May-2020 02:50:00    02-May-2020
    "Satellite 2"    "Ground station 1"          7           02-May-2020 05:59:00    02-May-2020
    "Satellite 2"    "Ground station 1"          8           02-May-2020 09:06:00    02-May-2020
```

Play the scenario to visualize the ground stations.

```
play(sc)
```

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `satellite` | `access` | `groundStation` | `conicalSensor`

Topics

“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

groundTrack

Package: `matlabshared.satellitescenario`

Add ground track object to satellite in scenario

Syntax

```
groundTrack(sat)
groundTrack( ___ ,Name=Value)
```

Description

`groundTrack(sat)` adds ground track visualization for each satellite in `sat` based on their current positions. The ground track begins at the scenario StartTime, and ends at the StopTime. The spacing between samples that make up the ground track visualization is determined by the scenario `SampleTime`. If no viewer is open, a new viewer is launched, and the ground track is displayed. If a viewer is already open, the ground track is added to that viewer. By default, ground tracks will be displayed in 2-D.

`groundTrack( ___ ,Name=Value)` adds a `groundTrack` object by using one or more name-value pairs. Enclose each property name in quotes.

Examples

Add Ground Track to Satellite in Geosynchronous Orbit

Create a satellite scenario object.

```
startTime = datetime(2020,5,10);
stopTime = startTime + days(5);
sampleTime = 60;                                          % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Calculate the semimajor axis of the geosynchronous satellite.

```
earthAngularVelocity = 0.0000729211585530;                                                % rad/s
orbitalPeriod = 2*pi/earthAngularVelocity;                                                % seconds
earthStandardGravitationalParameter = 398600.4418e9;                                      % m^3/s^2
semiMajorAxis = (earthStandardGravitationalParameter*((orbitalPeriod/(2*pi))^2))^(1/3);
```

Define the remaining orbital elements of the geosynchronous satellite.

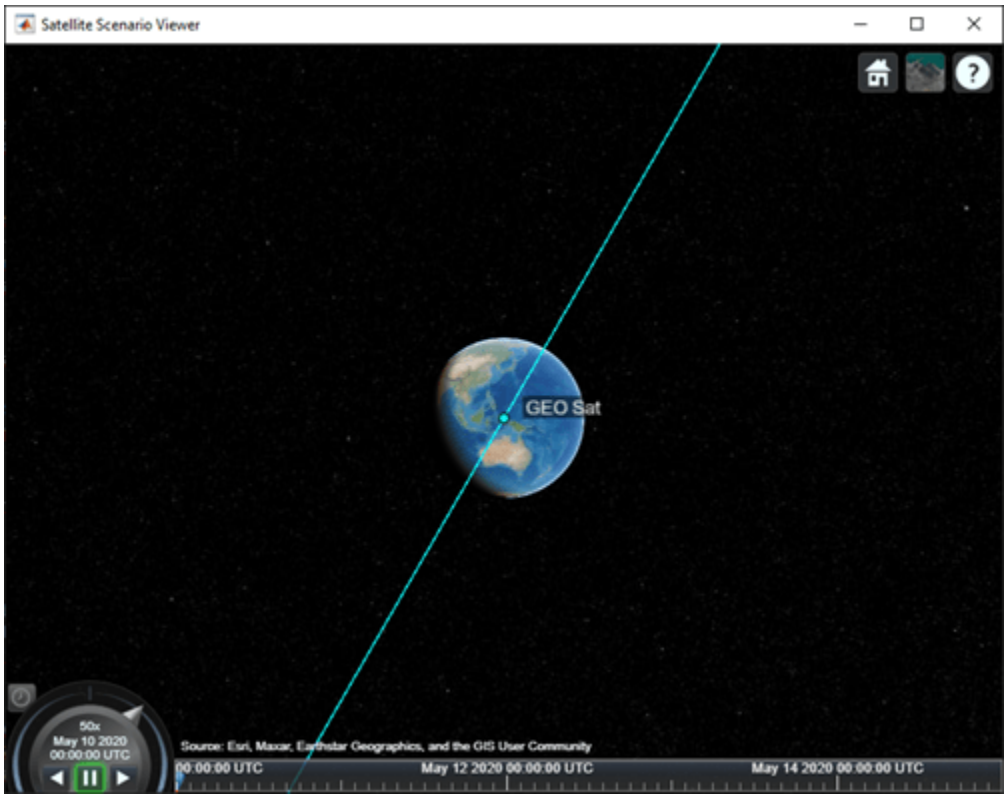
```
eccentricity = 0;
inclination = 60;                   % degrees
rightAscensionOfAscendingNode = 0; % degrees
argumentOfPeriapsis = 0;            % degrees
trueAnomaly = 0;                    % degrees
```

Add the geosynchronous satellite to the scenario.

```
sat = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode,...
        argumentOfPeriapsis,trueAnomaly,"OrbitPropagator","two-body-keplerian","Name","GEO Sat");
```

Visualize the scenario using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```



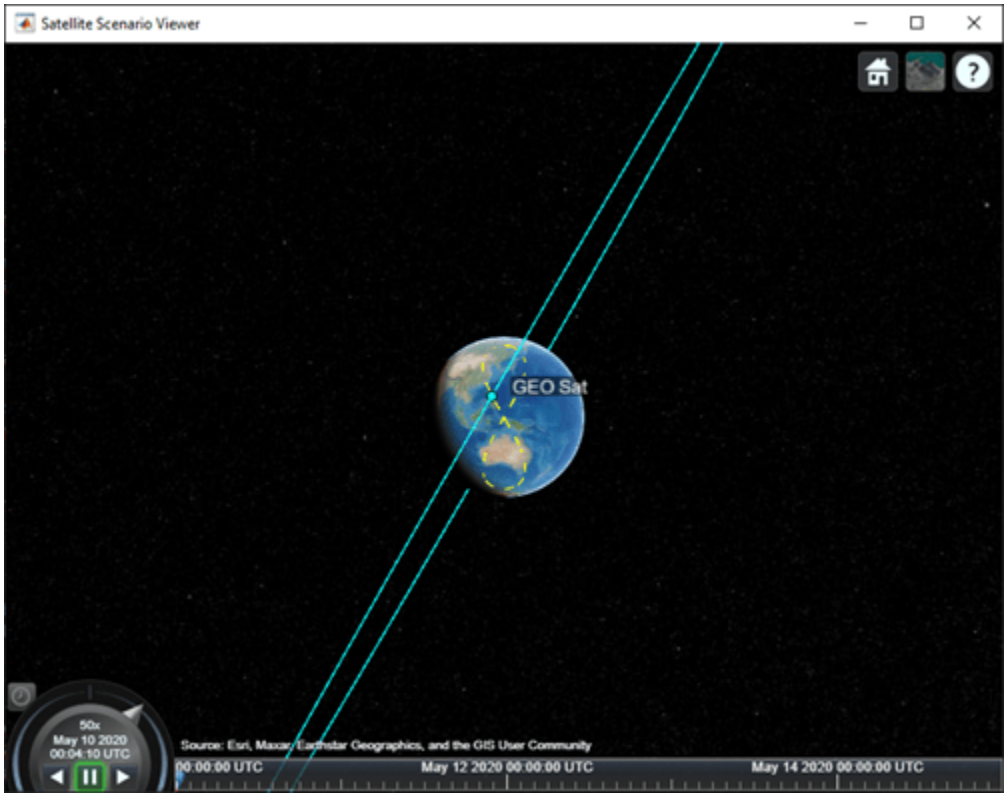
Add a ground track of the satellite to the visualization and adjust how much of the future and history of the ground track to display.

```
leadTime = 2*24*3600;                                          % seconds
trailTime = leadTime;
gt = groundTrack(sat,"LeadTime",leadTime,"TrailTime",trailTime)
```

```
gt =
  GroundTrack with properties:

          LeadTime: 172800
         TrailTime: 172800
         LineWidth: 1
     LeadLineColor: [1 1 0.0670]
    TrailLineColor: [1 1 0.0670]
    VisibilityMode: 'inherit'
```

Visualize the satellite movement and its trace on the ground. The satellite covers the area around Japan during one half of the day and Australia during the other half.

```
play(sc);
```



Input Arguments

`sat` — Satellite

row vector of `Satellite` objects

Satellite, specified as a row vector of `Satellite` objects.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `LeadTime=3600` sets the lead time of the ground track to 3600 seconds upon creation.

`Viewer` — Satellite scenario viewer

vector of `satelliteScenarioViewer` objects (default) | scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Satellite scenario viewer, specified as a scalar, vector, or array of `satelliteScenarioViewer` objects. If the `AutoSimulate` property of the scenario is `false`, adding a satellite to the scenario disables any previously available timeline and playback widgets.

`LeadTime` — Period of ground track to be visualized

`StartTime` to `StopTime` (default) | positive scalar

Period of the ground track to be visualized in the satellite scenario viewer, specified as `'LeadTime'` and a positive scalar in seconds.

`TrailTime` — Period of ground track history to be visualized
`StartTime` to `StopTime` (default) | positive scalar

Period of the ground track history to be visualized in `Viewer`, specified as `'TrailTime'` and a positive scalar in seconds.

`LineWidth` — Visual width of ground track
1 (default) | scalar in the range (0 10]

Visual width of the ground track in pixels, specified as `'LineWidth'` and a scalar in the range (0 10].

The line width cannot be thinner than the width of a pixel. If you set the line width to a value that is less than the width of a pixel on your system, the line displays as one pixel wide.

`LeadLineColor` — Color of future ground track line
`[1 0 1]` (default) | RGB triplet | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Color of the future ground track line, specified as `'LeadLineColor'` and an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	
`"none"`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	"#0072BD"	
[0.8500 0.3250 0.0980]	"#D95319"	
[0.9290 0.6940 0.1250]	"#EDB120"	
[0.4940 0.1840 0.5560]	"#7E2F8E"	
[0.4660 0.6740 0.1880]	"#77AC30"	
[0.3010 0.7450 0.9330]	"#4DBEEE"	
[0.6350 0.0780 0.1840]	"#A2142F"	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

TrailLineColor — Color of ground track line history
[1 0.5 0] (default) | RGB triplet | RGB triplet | string scalar of color name | character vector of color name

Color of the ground track line history, specified as 'TrailLineColor' and an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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].
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Therefore, the color codes "#FF8800", "#ff8800", "#F80", and "#f80" are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
"red"	"r"	[1 0 0]	"#FF0000"	
"green"	"g"	[0 1 0]	"#00FF00"	
"blue"	"b"	[0 0 1]	"#0000FF"	
"cyan"	"c"	[0 1 1]	"#00FFFF"	
"magenta"	"m"	[1 0 1]	"#FF00FF"	
"yellow"	"y"	[1 1 0]	"#FFFF00"	
"black"	"k"	[0 0 0]	"#000000"	
"white"	"w"	[1 1 1]	"#FFFFFF"	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
"none"	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	"#0072BD"	
[0.8500 0.3250 0.0980]	"#D95319"	
[0.9290 0.6940 0.1250]	"#EDB120"	
[0.4940 0.1840 0.5560]	"#7E2F8E"	
[0.4660 0.6740 0.1880]	"#77AC30"	
[0.3010 0.7450 0.9330]	"#4DBEEE"	
[0.6350 0.0780 0.1840]	"#A2142F"	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `groundStation` | `access` | `hide` | `satellite`

Topics

“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

GroundTrack

Ground track object belonging to satellite in scenario

Description

`GroundTrack` defines a ground track object belonging to a satellite in a scenario.

Creation

You can create a `GroundTrack` object using the `groundTrack` object function of the `Satellite` object.

Properties

`LeadTime` — Period of ground track to be visualized
`StartTime` to `StopTime` (default) | positive scalar

Period of the ground track to be visualized in the satellite scenario viewer, specified as `'LeadTime'` and a positive scalar in seconds.

`TrailTime` — Period of ground track history to be visualized
`StartTime` to `StopTime` (default) | positive scalar

Period of the ground track history to be visualized in `Viewer`, specified as `'TrailTime'` and a positive scalar in seconds.

`LineWidth` — Visual width of ground track
`1` (default) | scalar in the range (0 10]

Visual width of the ground track in pixels, specified as `'LineWidth'` and a scalar in the range (0 10].

The line width cannot be thinner than the width of a pixel. If you set the line width to a value that is less than the width of a pixel on your system, the line displays as one pixel wide.

`LeadLineColor` — Color of future ground track line
`[1 0 1]` (default) | RGB triplet | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Color of the future ground track line, specified as `'LeadLineColor'` and an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case

sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	
`"none"`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`TrailLineColor` — Color of ground track line history

`[1 0.5 0]` (default) | RGB triplet | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Color of the ground track line history, specified as `'TrailLineColor'` and an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.

- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Therefore, the color codes "#FF8800", "#ff8800", "#F80", and "#f80" are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
"red"	"r"	[1 0 0]	"#FF0000"	
"green"	"g"	[0 1 0]	"#00FF00"	
"blue"	"b"	[0 0 1]	"#0000FF"	
"cyan"	"c"	[0 1 1]	"#00FFFF"	
"magenta"	"m"	[1 0 1]	"#FF00FF"	
"yellow"	"y"	[1 1 0]	"#FFFF00"	
"black"	"k"	[0 0 0]	"#000000"	
"white"	"w"	[1 1 1]	"#FFFFFF"	
"none"	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	"#0072BD"	
[0.8500 0.3250 0.0980]	"#D95319"	
[0.9290 0.6940 0.1250]	"#EDB120"	
[0.4940 0.1840 0.5560]	"#7E2F8E"	
[0.4660 0.6740 0.1880]	"#77AC30"	
[0.3010 0.7450 0.9330]	"#4DBEEE"	
[0.6350 0.0780 0.1840]	"#A2142F"	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

VisibilityMode — Visibility mode of ground track
'inherit' (default) | 'manual'

Visibility mode of the ground track, specified as either one of these values:

- 'inherit' — Visibility of the graphic matches that of the parent.
- 'manual' — Visibility of the graphic is not inherited and is independent of that of the parent.

Object Functions

show Show object in satellite scenario viewer
hide Hide satellite scenario entity from viewer

Examples

Add Ground Track to Satellite in Geosynchronous Orbit

Create a satellite scenario object.

```
startTime = datetime(2020,5,10);
stopTime = startTime + days(5);
sampleTime = 60;                                          % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Calculate the semimajor axis of the geosynchronous satellite.

```
earthAngularVelocity = 0.0000729211585530;                                                % rad/s
orbitalPeriod = 2*pi/earthAngularVelocity;                                                % seconds
earthStandardGravitationalParameter = 398600.4418e9;                                      % m^3/s^2
semiMajorAxis = (earthStandardGravitationalParameter*((orbitalPeriod/(2*pi))^2))^(1/3);
```

Define the remaining orbital elements of the geosynchronous satellite.

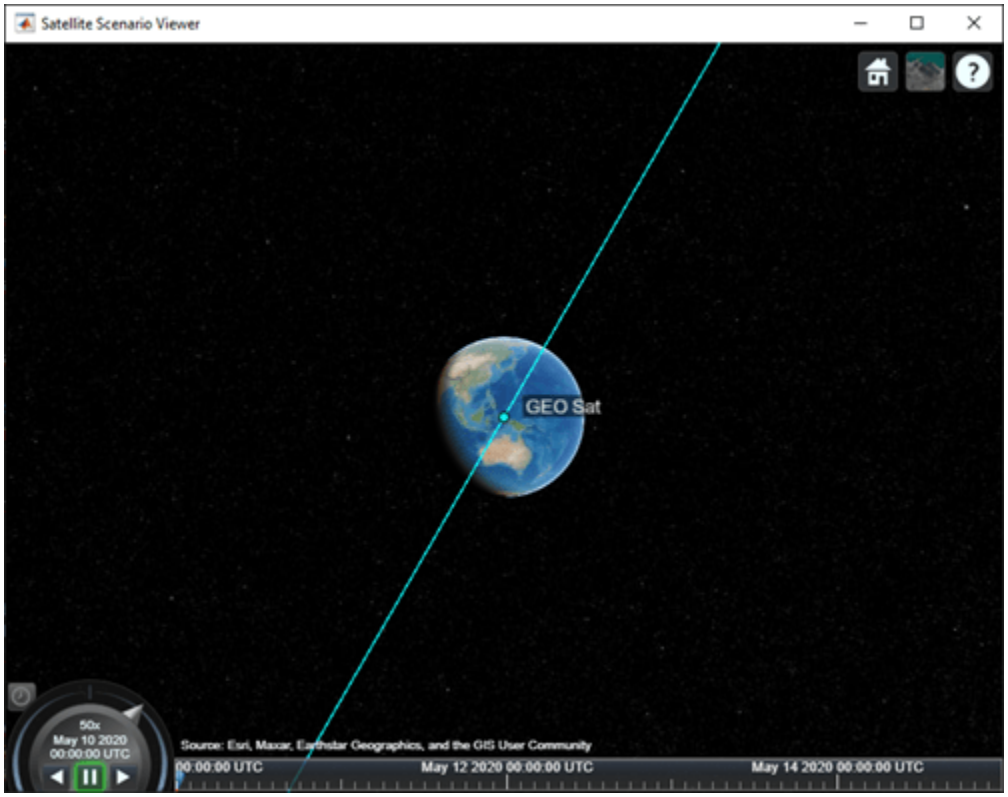
```
eccentricity = 0;
inclination = 60;                   % degrees
rightAscensionOfAscendingNode = 0; % degrees
argumentOfPeriapsis = 0;            % degrees
trueAnomaly = 0;                    % degrees
```

Add the geosynchronous satellite to the scenario.

```
sat = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode,...
        argumentOfPeriapsis,trueAnomaly,"OrbitPropagator","two-body-keplerian","Name","GEO Sat");
```

Visualize the scenario using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```



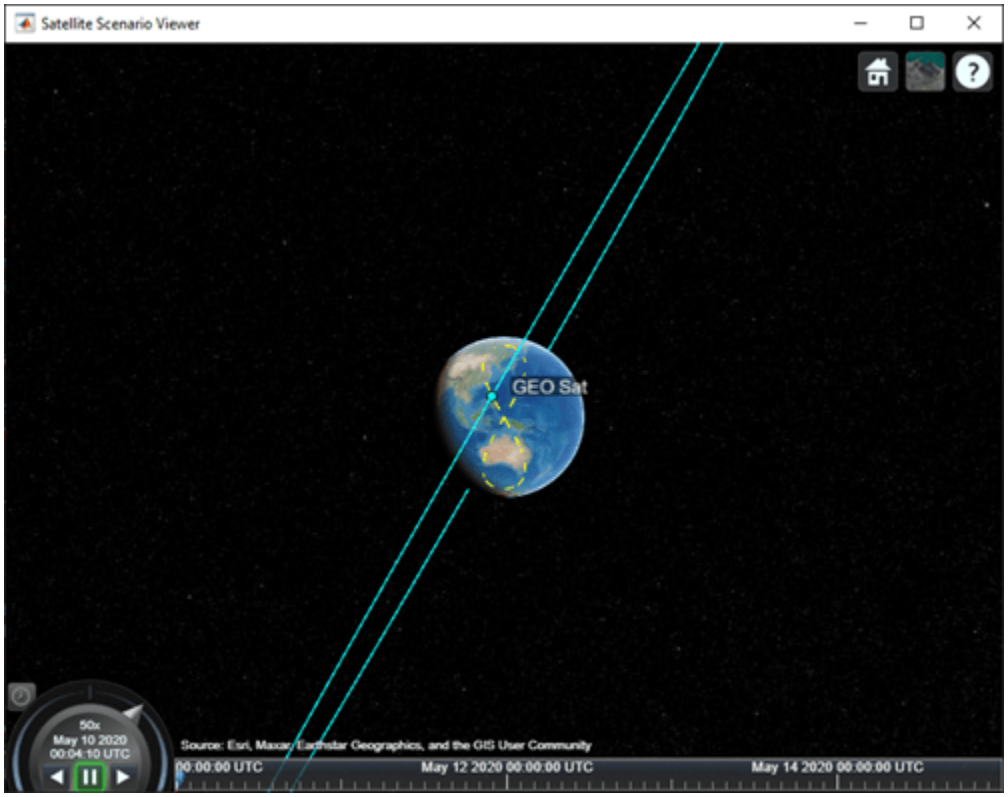
Add a ground track of the satellite to the visualization and adjust how much of the future and history of the ground track to display.

```
leadTime = 2*24*3600;                                          % seconds
trailTime = leadTime;
gt = groundTrack(sat,"LeadTime",leadTime,"TrailTime",trailTime)
```

```
gt =
  GroundTrack with properties:

         LeadTime: 172800
        TrailTime: 172800
        LineWidth: 1
     LeadLineColor: [1 1 0.0670]
    TrailLineColor: [1 1 0.0670]
    VisibilityMode: 'inherit'
```

Visualize the satellite movement and its trace on the ground. The satellite covers the area around Japan during one half of the day and Australia during the other half.

```
play(sc);
```



Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `groundStation` | `access` | `hide` | `satellite`

Topics

“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

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) | on/off logical value

Visibility of the heading indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`. 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.

`ContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object created using the `uicontextmenu` function. Use this property to display a context menu when you right-click on a component.

`Enable` — Operational state of header indicator

`'on'` (default) | on/off logical value

Operational state of header indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- 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 = uigridlayout([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 “Callbacks in App Designer”.

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 “Callbacks in App Designer”.

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) | on/off logical value

Callback interruption, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

This 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.

MATLAB determines callback interruption behavior whenever it executes a command that processes the callback queue. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines if the interruption occurs:

- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the object that owns the interrupting callback determines if the interrupting callback is discarded or added to the callback queue.
- If the value of `Interruptible` is `'on'`, then the interruption occurs. The next time MATLAB processes the callback queue, it stops the execution of the running callback and executes the interrupting callback. After the interrupting callback completes, MATLAB then resumes executing the running callback.

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.
- If the interrupting callback is owned by a `Timer` object, then the callback executes according to schedule regardless of the `Interruptible` property value.

Note 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.

The `BusyAction` property determines callback queuing behavior only when both of these conditions are met:

- The running callback contains a command that processes the callback queue, such as `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause`.
- The value of the `Interruptible` property of the object that owns the running callback is `'off'`.

Under these conditions, the `BusyAction` property of the object that owns the interrupting callback determines how MATLAB handles the interrupting callback. 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

on/off logical value

This property is read-only.

Deletion status, returned as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

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”.

Version History

Introduced in R2018b

See Also

`uiaeroheading`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

hide

Package: `matlabshared.satellitescenario`

Hide satellite scenario entity from viewer

Syntax

```
hide(item)
hide(item,viewer)
```

Description

`hide(item)` hides the specified satellite scenario entity from all open instances of the Satellite Scenario Viewer.

`hide(item,viewer)` hides the specified satellite scenario entity on the Satellite Scenario Viewer specified by `viewer`.

Examples

Hide Satellite from Satellite Scenario Viewer

Create a satellite scenario object.

```
sc = satelliteScenario;
```

Add a satellite to the scenario.

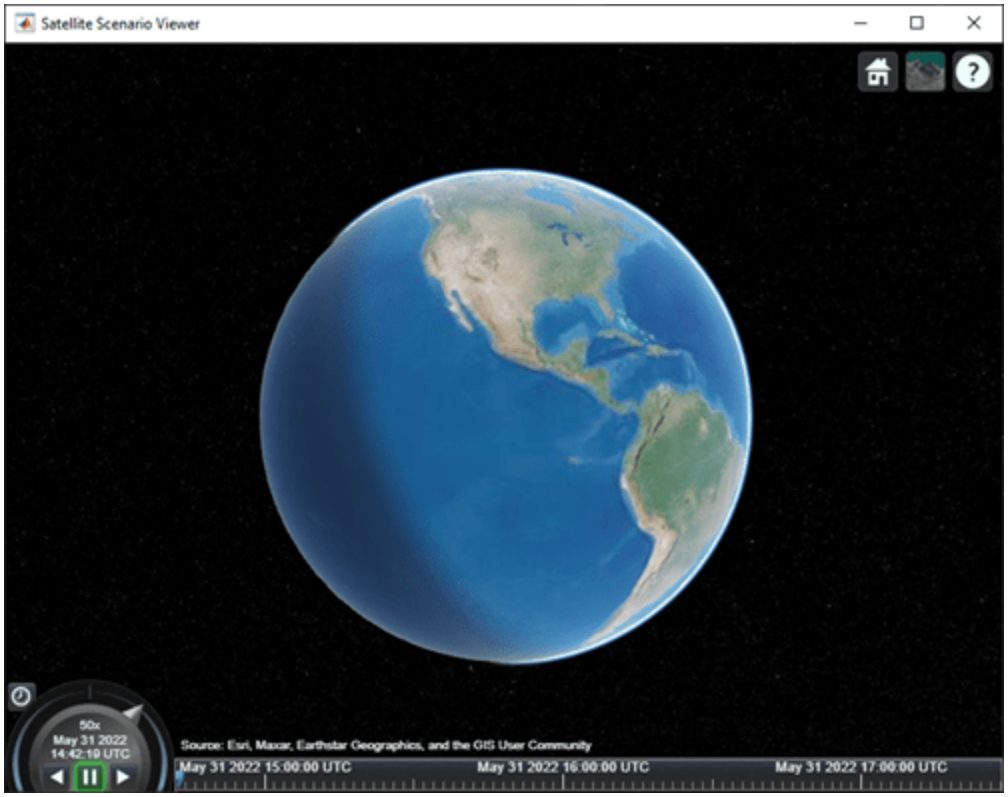
```
semiMajorAxis = 10000000;                           % meters
eccentricity = 0;
inclination = 0;                                    % degrees
rightAscensionOfAscendingNode = 0;                  % degrees
argumentOfPeriapsis = 0;                            % degrees
trueAnomaly = 0;                                    % degrees
sat = satellite(sc,semiMajorAxis,eccentricity, ...
      inclination,rightAscensionOfAscendingNode, ...
      argumentOfPeriapsis,trueAnomaly);
```

Visualize the satellite by using the Satellite Scenario Viewer.

```
viewer = satelliteScenarioViewer(sc);
```

Hide the satellite from the viewer.

```
hide(sat,viewer);
```



Input Arguments

`item` — Item

`Satellite` object | `GroundStation` object | `ConicalSensor` object | `GroundTrack` object | `FieldofView` object | `Access` object

The item, specified as a `Satellite`, `GroundStation`, `ConicalSensors`, `GroundTrack`, `FieldOfView`, or `Access` object. These objects must belong to the same `satelliteScenario`, object.

`viewer` — Satellite Scenario Viewer

row vector of all `satelliteScenarioViewer` objects (default) | scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object or a vector or array of `satelliteScenarioViewer` objects.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `hideAll` | `showAll`

Topics
“Satellite Scenario Key Concepts” on page 2-65

hide

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

Hide animation figure

Syntax

```
hide(h)
```

Description

`hide(h)` hides (closes) the figure for the animation object `h`. Use the `show` function to display the animation object figure.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

Examples

Hide Animation Object

Hide the animation object figure that the `Aero.Animation.show` method displays.

```
h=Aero.Animation;
show(h);
hide(h);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation` | `show`

hideAll

Package: `matlabshared.satellitescenario`

Hide all graphics in satellite scenario viewer

Syntax

```
hideAll(viewer)
```

Description

`hideAll(viewer)` hides all graphics in the specified satellite scenario viewer.

Examples

Hide All Graphics from Satellite Scenario Viewer

Create a satellite scenario object.

```
sc = satelliteScenario;
```

Add satellites to the scenario.

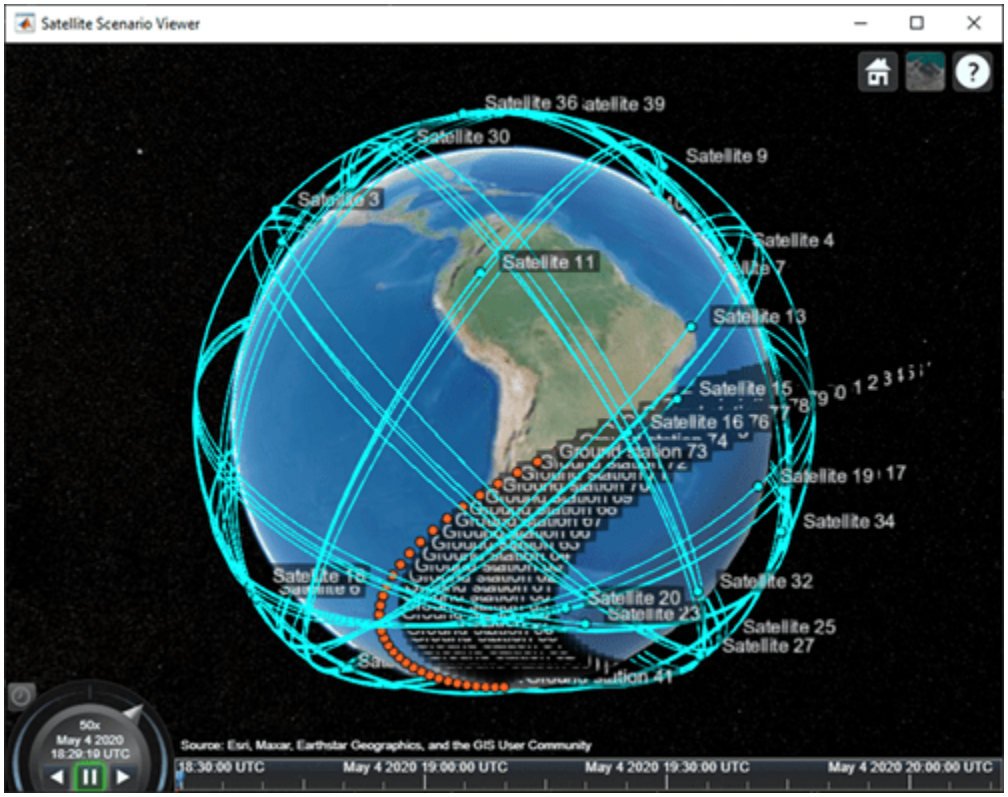
```
tleFile = "leoSatelliteConstellation.tle";
sats = satellite(sc,tleFile);
```

Add a hundred ground stations to the scenario.

```
latitudes = linspace(-90,90,100);            % degrees
longitudes = linspace(-180,180,100);         % degrees
gss = groundStation(sc,latitudes,longitudes);
```

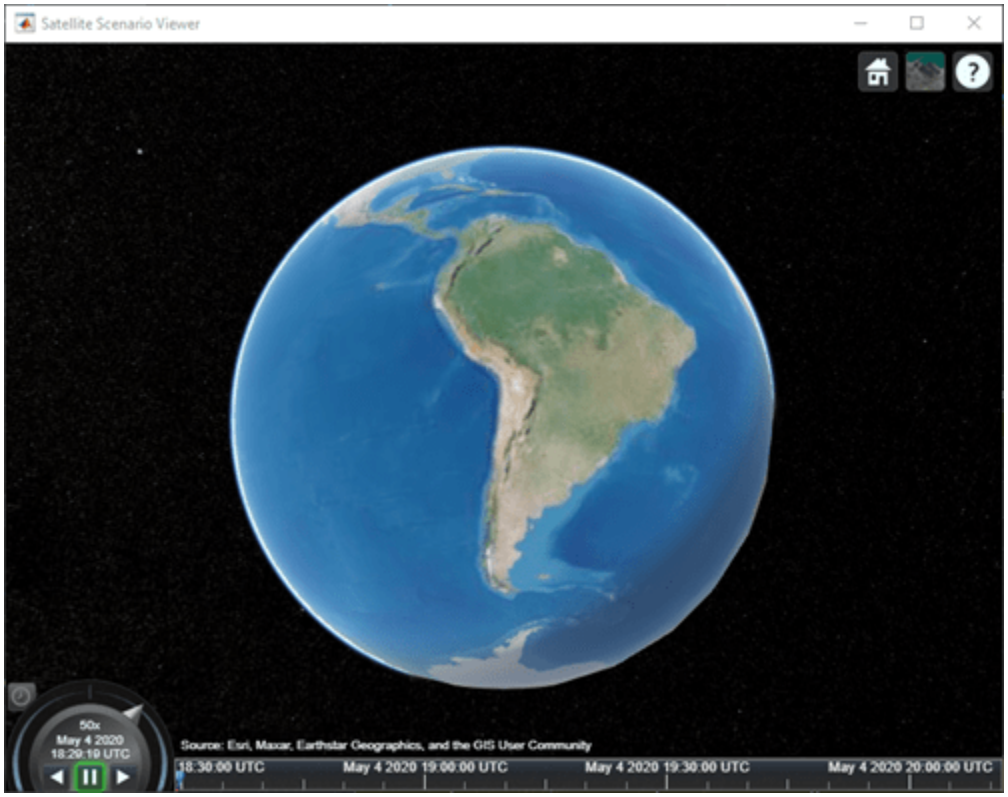
Visualize the scenario using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```



Hide all graphics in the viewer.

```
hideAll(v);
```



Input Arguments

`viewer` — Satellite Scenario Viewer

`satelliteScenarioViewer` object

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object.[8]

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide` | `campos` | `camroll` | `campitch` | `camheading` | `camheight` | `camtarget` | `access` | `groundStation` | `conicalSensor` | `showAll`

Topics

“Satellite Scenario Key Concepts” on page 2-65

8 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

igrfmagm

Calculate Earth magnetic field and secular variation using International Geomagnetic Reference Field

Syntax

```
[XYZ,H,D,I,F,DXDYDZ,DH,DD,DI,DF] = igrfmagm(height,latitude,longitude,
decimalYear)
[XYZ,H,D,I,F,DXDYDZ,DH,DD,DI,DF] = igrfmagm(height,latitude,longitude,
decimalYear,generation)
```

Description

`[XYZ,H,D,I,F,DXDYDZ,DH,DD,DI,DF] = igrfmagm(height,latitude,longitude, decimalYear)` calculates the Earth magnetic field and the secular variation at a specific location and time using the International Geomagnetic Reference Field generation 13 (IGRF-13).

`[XYZ,H,D,I,F,DXDYDZ,DH,DD,DI,DF] = igrfmagm(height,latitude,longitude, decimalYear,generation)` optionally uses different generations of the International Geomagnetic Reference Field (IGRF-13, IGRF-12, and IGRF-11).

Examples

Calculate the Magnetic Model

Calculate the magnetic model 1000 meters over Natick, Massachusetts on July 4, 2015 using IGRF-13.

```
[XYZ,H,D,I,F] ...
= igrfmagm(1000,42.283,-71.35,decyear(2015,7,4),13)

XYZ =
   1.0e+04 *

    1.9471   -0.5086    4.8177

H =
   2.0124e+04

D =
  -14.6381

I =
   67.3295

F =
   5.2212e+04
```

Calculate the Magnetic Model with Matrix inputs

Calculate the magnetic model at 0 and 10000 km over Lawrence, Kansas on May 15, 2018 using IGRF-13.

```
h = [0,10000000]
lat = [38.957114,38.957114]
lon = [-95.253997,-95.253997]
dyear = [decyear(2018,5,14), decyear(2018,5,14)]
[XYZ,H,D,I,F] = igrfmagm(h,lat,lon,dyear,13)
```

```
h =
           0    10000000

lat =
   38.9571   38.9571

lon =
  -95.2540  -95.2540

dyear =
   1.0e+03 *
    2.0184    2.0184

XYZ =
   1.0e+04 *

    2.0655    0.0783    4.7990
    0.1192    0.0046    0.2571

H =
   1.0e+04 *

    2.0670
    0.1193

D =
    2.1714
    2.1968

I =
   66.6981
   65.1016

F =
   1.0e+04 *

    5.2252
    0.2834
```

Input Arguments

`height` — Distance

matrix | scalar | vector

Distance from the surface of the Earth, specified as a matrix, scalar, or vector, in meters.

Data Types: `double`

`latitude` — Geodetic latitude
scalar | vector | matrix

Geodetic latitude, specified as a matrix, scalar, or vector, in degrees. North latitude is positive and south latitude is negative.

This function accepts latitude values greater than 90 and less than -90.

Data Types: `double`

`longitude` — Geodetic longitude
matrix | scalar | vector

Geodetic longitude specified as a matrix, scalar, or vector, in degrees. East longitude is positive and west longitude is negative. This function accepts ranges greater than 180 and less than -180.

Data Types: `double`

`decimalYear` — Year
matrix | scalar | vector

Year, in decimal format, specified as a matrix. This value can have any fraction of the year that has already passed.

Data Types: `double`

`generation` — Generation version of International Geomagnetic Reference Field
13 (default) | 12 | 11 | scalar numeric

Generation version of the International Geomagnetic Reference Field, specified as 13, 12, or 11.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

Output Arguments

`XYZ` — Magnetic field vector
vector | matrix

Magnetic field vector, in nanotesla (nT), returned as a vector or matrix the same size as the input matrix with an additional dimension, the last dimension. The last dimension of the matrix is of size 3, specifying the *X*, *Y*, and *Z* components of the magnetic field. *Z* is the vertical component (+ve down). The components of this vector are in the north-east-down (NED) reference frame.

Data Types: `double`

`H` — Horizontal intensity
scalar | vector | matrix

Horizontal intensity, returned as a scalar, vector, or matrix, in nanotesla (nT), the same size as the input matrix.

Data Types: `double`

`D` — Declination
scalar | vector | matrix

Declination, returned as a scalar, in degrees (+ve east), the same size as the input matrix.

Data Types: `double`

`I` — Inclination
scalar | vector | matrix

Inclination, returned as a scalar, in degrees (+ve down), the same size as the input matrix.

Data Types: `double`

`F` — Total intensity
scalar | vector | matrix

Total intensity, returned as a scalar, in nanotesla (nT), the same size as the input matrix.

Data Types: `double`

`DXDYDZ` — Secular variation in magnetic field vector
vector | matrix

Secular variation in magnetic field vector, returned as a vector or matrix, in nT/year, the same size ans the input matrix with an additional dimension, the last dimension. The last dimension of the matrix is of size 3, specifying the X, Y, and Z components of the magnetic field. Z is the vertical component (+ve down).

Data Types: `double`

`DH` — Secular variation in horizontal intensity
scalar | vector | matrix

Secular variation in horizontal intensity, in nT/year, returned as a scalar, the same size as the input matrix.

Data Types: `double`

`DD` — Secular variation in declination
scalar | vector | matrix

Secular variation in declination, in minutes/year (+ve east), returned as a scalar, the same size as the input matrix.

Data Types: `double`

`DI` — Secular variation in inclination
scalar | vector | matrix

Secular variation in inclination, in minutes/year (+ve down), returned as a scalar, the same size as the input matrix.

Data Types: `double`

`DF` — Secular variation in total intensity
scalar | vector | matrix

Secular variation in total intensity, in nT/year, returned as a scalar, the same size as the input matrix.

Data Types: `double`

Limitation

- This function is valid for these year ranges:
  - IGRF-13 model — 1900 and 2025
  - IGRF-12 model — 1900 and 2020
  - IGRF-11 model — 1900 and 2015
- This function is valid between the heights of -1000 m and 5.6 Earth radii (35,717,567.2 m).
- The `height`, `latitude`, `longitude`, and `decimalYear` arguments must all be the same size (matrix, scalar, and so forth).

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`.

Version History

Introduced in R2015b

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

ijk2keplerian

Keplerian orbit elements using position and velocity vectors

Syntax

```
[a,ecc,incl,RAAN,argp,nu,truelon,arglat,lonper] = ijk2keplerian(r_ijk, v_ijk)
```

Description

`[a,ecc,incl,RAAN,argp,nu,truelon,arglat,lonper] = ijk2keplerian(r_ijk, v_ijk)` calculates Keplerian orbit elements for given position and velocity vectors in the geocentric equatorial coordinate system.

Examples

Convert IJK Position and Velocity

Convert the geocentric equatorial coordinate system (IJK) position and velocity to Keplerian orbital elements.

```
r_ijk = [-2981784 5207055 3161595];
  v_ijk = [-3384 -4887 4843];
  [a, ecc, incl, RAAN, argp, nu, truelon, arglat, lonper] =...
   ijk2keplerian(r_ijk, v_ijk)

a =
   6.7845e+06

ecc =
   9.1950e-04

incl =
   51.7528

RAAN =
   95.2570

argp =
  106.4005

nu =
  290.0096

truelon =
   NaN

arglat =
   NaN
```

```
lonper =
   NaN
```

Input Arguments

`r_ijk` — Position component

`0` (default) | 3-by-1 array

Geocentric equatorial position components, specified as a 3-by-1 array, in meters.

Data Types: `double`

`v_ijk` — Velocity component

`0` (default) | 3-by-1 array

Geocentric equatorial velocity components, specified as a 3-by-1 array, in m/s.

Data Types: `double`

Output Arguments

`a` — Semi-major axis

scalar

Semimajor axis (half of the longest diameter) of the orbit, returned as a scalar, in meters.

Data Types: `double`

`ecc` — Orbit eccentricity

scalar value greater than or equal to 0

Orbit eccentricity (deviation of orbital curve from circular), returned as a scalar.

Data Types: `double`

`incl` — Inclination

scalar value from `0` to `180`

Inclination (tilt angle) of the orbit, in degrees.

Data Types: `double`

`RAAN` — Right ascension of ascending node

scalar value from `0` to `360`

Angle in the equatorial plane from the *x*-axis to the location of the ascending node (point at which the satellite crosses the equator from south to north), in degrees.

Data Types: `double`

`argp` — Angle between CubeSat ascending node and periapsis

scalar value from `0` to `360`

Angle between the CubeSat ascending node and periapsis (closest point of orbit to Earth), in degrees.

Data Types: `double`

`nu` — Angle between periapsis and current position of CubeSat
scalar value from `0` to `360`

Angle between periapsis and current position of CubeSat, in degrees.

Data Types: `double`

`truelon` — Angle between *x*-axis and CubeSat position vector
scalar value from `0` to `360`

Angle between the *x*-axis and CubeSat position vector, in degrees.

Data Types: `double`

`arglat` — Angle between ascending node and CubeSat position vector
scalar value from `0` to `360`

Angle between the ascending node and the CubeSat position vector, in degrees.

Data Types: `double`

`lonper` — Angle between *x*-axis and eccentricity vector
scalar value from `0` to `360`

Angle between the *x*-axis and the eccentricity vector, in degrees.

Data Types: `double`

Version History

Introduced in R2019a

References

[1] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. alg. 5. McGraw-Hill, 1997.

See Also

`ecef2eci` | `eci2ecef` | `dcmeci2ecef` | `aeroReadIERSData` | `deltaCIP` | `polarMotion` | `deltaUT1` | `keplerian2ijk` | `siderealTime` | CubeSat Vehicle

initialize

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

Create animation object figure and axes and build patches for bodies

Syntax

```
initialize(h)
```

Description

`initialize(h)` creates and initializes a figure and axes for the animation object `h` and builds patches for the bodies associated with the animation object. If an existing figure is present, 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 — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

Examples

Initialize `Aero.Animation` Object

```
h = Aero.Animation;
initialize(h);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

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();
```

Version History

Introduced in R2007a

See Also

`delete` | `play` | `GenerateRunScript` | `update`

initialize (Aero.VirtualRealityAnimation)

Create and populate virtual reality animation object

Syntax

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

Description

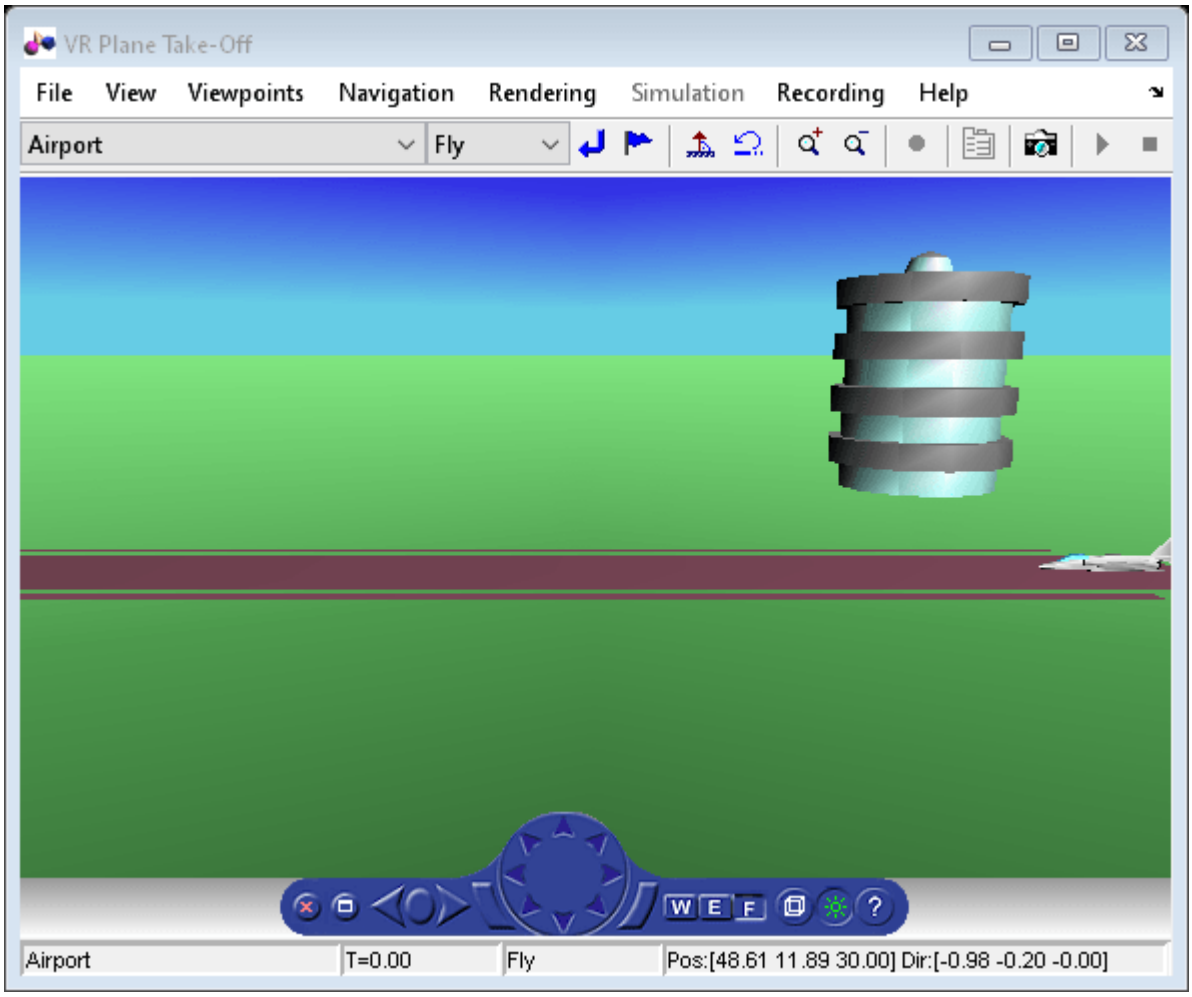
`initialize(h)` and `h.initialize` create a virtual reality animation world and populate the virtual reality animation object `h`. If a previously initialized virtual reality animation object exists, and that object has user-specified data, this function saves the previous object to be reset after the initialization.

Examples

Initialize Virtual Animation Object

This example shows how to initialize the virtual reality animation object, `h`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
```



Version History

Introduced in R2007b

See Also

`delete` | `play`

initIfNeeded

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

Initialize animation graphics if needed

Syntax

```
initIfNeeded(h)
```

Description

`initIfNeeded(h)` initializes animation object graphics, if necessary.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

Examples

Initialize Animation Object as Needed

```
h=Aero.Animation;
initIfNeeded(h);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

juliandate

Julian date calculator

Syntax

```
jd = juliandate(datetime)
jd = juliandate(dateVector)
jd = juliandate(dateCharacterVector,format)

jd = juliandate(year,month,day)
jd = juliandate([year,month,day],'Action'=value)
jd = juliandate(year,month,day,hour,minute,second,'Action'=value)
jd = juliandate([year,month,day,hour,minute,second],'Action'=value)
```

Description

`jd = juliandate(datetime)` converts one or more `datetime` arrays to Julian date, `jd`.

`jd = juliandate(dateVector)` converts one or more date vectors, `dateVector`, to Julian date, `jd`.

`jd = juliandate(dateCharacterVector,format)` converts one or more date character vectors, `dateCharacterVector`, to Julian date, `jd`, using format `format`.

`jd = juliandate(year,month,day)` and `jd = juliandate([year,month,day], 'Action'=value)` return the Julian date for corresponding elements of the `year,month,day` arrays.

`jd = juliandate(year,month,day,hour,minute,second,'Action'=value)` and `jd = juliandate([year,month,day,hour,minute,second],'Action'=value)` return the Julian date for corresponding elements of the `year,month,day,hour,minute,second` arrays. Specify the six arguments as 1-D arrays of the same length or scalar values.

Examples

Calculate Julian Date Using `datetime` Array

Calculate the Julian date for February 4, 2016 from datetime array.

```
dt = datetime('04-02-2016','InputFormat','dd-MM-yyyy')
```

```
dt = datetime
   04-Feb-2016
```

```
jd = juliandate(dt)
```

```
jd = 2.4574e+06
```

Calculate Julian Date Using Date Character Version and dd-mm-yyyy Format

Calculate Julian date for May 24, 2005 using date character version and `dd-mm-yyyy` format.

```
jd = juliandate('24-May-2005','dd-mmm-yyyy')
```

```
jd = 2.4535e+06
```

Calculate Julian Date Using Year, Month, and Day Inputs

Calculate Julian date for December 19, 2006 from year, month, and day inputs.

```
jd = juliandate(2006,12,19)
```

```
jd = 2.4541e+06
```

Calculate Julian Date from Year, Month, Day, Hour, Minute, and Second Inputs

Calculate Julian date for October 10, 2004, at 12:21:00 p.m. from year, month, day, hour, month, and second inputs.

```
jd = juliandate(2004,10,10,12,21,0)
```

```
jd = 2.4533e+06
```

Input Arguments

`datetime` — `datetime` array
m-by-1 array | 1-by-*m* array

`datetime` array, specified as an *m*-by-1 array or 1-by-*m* array.

`dateVector` — Full or partial date vector
m-by-6 matrix | *m*-by-3 matrix | positive double-precision number

Full or partial date vector, specified as an *m*-by-6 or *m*-by-3 matrix containing *m* full or partial date vectors, respectively:

- Full date vector — Contains six elements specifying the year, month, day, hour, minute, and second
- Partial date vector — Contains three elements specifying the year, month, and day

Data Types: `double`

`dateCharacterVector` — Date character vector
character array | 1-D cell array of character vectors

Date character vector, specified as a character array, where each row corresponds to one date, or a 1-D cell array of character vectors.

Data Types: `char` | `string`

`format` — Date format

-1 (default) | character vector | string scalar | integer

Date format, specified as a character vector, string scalar, or integer. All dates in `dateCharacterVector` must have the same format and use the same date format symbols as the `datenum` function.

`juliandate` does not accept formats containing the letter *`Q`*.

Data Types: `char` | `string`

`year` — Year

positive scalar | 1-D array | decimal

Year, specified as a positive scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `char` | `string`

`month` — Month

positive or negative scalar | 1-D array | decimal

Month, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`day` — Day

positive or negative scalar | 1-D array | decimal

Day, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`hour` — Date format

positive or negative scalar | 1-D array | decimal

Hour, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`minute` — Minute

positive or negative scalar | 1-D array | decimal

Minute, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`second` — Second

positive or negative scalar | 1-D array | decimal

Second, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`'Action'=value` — Action

`'Warning'` (default) | `'Error'` | `'None'`

Action for noninteger decimal date input for `year`, `month`, `day`, `hour`, and `minute`, specified as:

- `'Error'` — Displays error and total number of noninteger date values.
- `'Warning'` — Displays warning and total number of adjusted noninteger date values. The function adjusts the noninteger value to its integer equivalent before calculating the value.
- `'None'` — Does not display warning or error. The function adjusts the noninteger value to its integer equivalent before calculating the value.

Data Types: `char` | `string`

Output Arguments

`jd` — Julian date

m-by-6 column vector | *m*-by-3 column vector | row vector | column vector

Julian date, returned as a column vector of *m* Julian dates, which are the number of days and fractions since noon Universal Time on January 1, 4713 BCE.

- *m*-by-6 column vector — Contains six elements specifying the year, month, day, hour, minute, and second
- *m*-by-3 column vector — Contains three elements specifying the year, month, and day
- Row or column vector — Contains *m* Julian dates

Dependencies

The output format depends on the input format:

Input Syntax	dy Format
`jd = juliandate(dateVector)`	*m*-by-6 column vector or *m*-by-3 column vector of *m* Julian dates.
`jd = juliandate(dateCharacterVector,format)`	Column vector of *m* Julian dates, where *m* is the number of character vectors in `dateCharacterVector`.

Limitations

The calculation of Julian date does not take into account leap seconds.

Version History

Introduced in R2006b

R2022b: `juliandate` accepts decimal values

Behavior changed in R2022b

The `juliandate` function now accepts decimal values for these input arguments:

- `year`
- `month`
- `day`
- `hour`
- `minute`

See Also

`decyear` | `leapyear` | `mjuliandate` | `datenum` | `datestr`

keplerian2ijk

Position and velocity vectors in geocentric equatorial coordinate system using Keplerian orbit elements

Syntax

```
[r_ijk,v_ijk] = keplerian2ijk(a,ecc,incl,RAAN,argp,nu)
[r_ijk,v_ijk] = keplerian2ijk( ___ ,Name,Value)
```

Description

`[r_ijk,v_ijk] = keplerian2ijk(a,ecc,incl,RAAN,argp,nu)` calculates the position and velocity vectors in the geocentric equatorial coordinate system (IJK) for given Keplerian orbit elements of noncircular, inclined orbits.

`[r_ijk,v_ijk] = keplerian2ijk( ___ ,Name,Value)` specifies orbit element properties using one or more name-value pair arguments. For example, `'truelon','17'` specifies the angle between the *x*-axis and CubeSat position vector. Specify name-value pair arguments after all other input arguments.

Examples

Convert Keplerian Orbital Elements

Convert Keplerian orbital elements to geocentric equatorial coordinate system (IJK) position and velocity.

```
a = 6786230;
ecc = .01;
incl = 52;
RAAN = 95;
argp = 93;
nu = 300;
[r_ijk, v_ijk] = keplerian2ijk(a, ecc, incl, RAAN, argp, nu)
```

```
r_ijk =
   1.0e+06 *
   -2.7489
    5.4437
    2.8977

v_ijk =
   1.0e+03 *
   -3.5694
   -4.5794
    5.0621
```

Convert Keplerian Orbital Elements for Equatorial Orbit

Convert Keplerian orbital elements to geocentric equatorial coordinate system (IJK) position and velocity for equatorial orbit.

```
a = 6786230;
ecc = .1;
incl = 0;
RAAN = 95;
argp = 93;
nu = 300;
lonper = 45;
[r_ijk, v_ijk] = keplerian2ijk(a, ecc, incl, RAAN, argp, nu, 'lonper', lonper)
```

```
r_ijk =

   1.0e+06 *
    6.1804
   -1.6560
         0

v_ijk =

   1.0e+03 *
    1.4489
    7.9848
         0
```

Input Arguments

a — Semi-major axis

scalar

Semimajor axis (half of the longest diameter) of the orbit, specified as a scalar, in meters.

Data Types: `double`

ecc — Orbit eccentricity

`0` (default) | scalar value greater than or equal to 0

Orbit eccentricity (deviation of orbital curve from circular), specified as a scalar.

Data Types: `double`

incl — Inclination

`0` (default) | scalar value from `0` to `180`

Inclination (tilt angle) of the orbit, in degrees.

Data Types: `double`

RAAN — Right ascension of ascending node

`0` (default) | scalar value from `0` to `360`

Angle in the equatorial plane from the *x*-axis to the location of the ascending node, point at which the satellite crosses the equator from south to north, in degrees. The function does not use this value for equatorial orbits.

Data Types: `double`

`argp` — Angle between CubeSat ascending node and periapsis
`0` (default) | scalar value from `0` to `360`

Angle between the CubeSat ascending node and the periapsis (closest point of orbit to Earth), in degrees. The function does not use this value for circular and equatorial orbits.

Data Types: `double`

`nu` — Angle between periapsis and current position of CubeSat
`0` (default) | scalar value from `0` to `360`

Angle between the periapsis and the current position of CubeSat, in degrees. The function does not use this value for circular orbits.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: 45

`truelon` — Angle between *x*-axis and CubeSat position vector
`0` (default) | scalar value from `0` to `360`

Angle between the *x*-axis and the CubeSat position vector, in degrees. The function uses this value only for circular equatorial orbits (where eccentricity and inclination are zero).

Data Types: `double`

`arglat` — Angle between ascending node and CubeSat position vector
`0` (default) | scalar value from `0` to `360`

Angle between the ascending node and the CubeSat position vector, in degrees. The function uses this value only for circular inclined orbits (where eccentricity is zero and inclination is nonzero).

Data Types: `double`

`lonper` — Angle between *x*-axis and eccentricity vector
`0` | scalar value from `0` to `360`

Angle between the *x*-axis and the eccentricity vector, in degrees. The function uses this value only for noncircular equatorial orbits (where eccentricity is nonzero and inclination is zero).

Data Types: `double`

Output Arguments

`r_ijk` — Position component
3-by-1 array

Geocentric equatorial position components, returned as a 3-by-1 array, in meters.

`v_ijk` — Velocity component
3-by-1 array

Geocentric equatorial velocity components, returned as a 3-by-1 array, in m/s.

Version History
Introduced in R2019a

References

[1] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. alg. 5. McGraw-Hill, 1997.

See Also
`ecef2eci` | `eci2ecef` | `dcmeci2ecef` | `aeroReadIERSData` | `deltaCIP` | `polarMotion` | `deltaUT1` | `ijk2keplerian` | `siderealTime` | CubeSat Vehicle

leapyear

Determine leap year

Syntax

```
leapyear = leapyear(year)
```

Description

`leapyear = leapyear(year)` determines whether one or more years are leap years.

Examples

Determine if 2005 is Leap Year

Determine whether 2005 is a leap year.

```
ly = leapyear(2005)
```

```
ly =
  logical

   0
```

Determine if Array of Years are Leap Years

Determine if 2000, 2005, and 2020 are leap years.

```
ly = leapyear([2000 2005 2020])
```

```
ly =
  1×3 logical array

   1   0   1
```

Input Arguments

year — Year

scalar | array | numeric

Year to be evaluated, specified as an array or scalar. The function floors non-integer values to the nearest integer value.

Data Types: `double`

Output Arguments

`leapyear` — Leap year determination

scalar | array | logical value

Leap year determination, returned as a scalar or array as a logical value.

Limitations

The determination of leap years is done by Gregorian calendar rules.

Version History

Introduced in R2006b

See Also

`decyear` | `juliandate` | `mjuliandate` | `tdbjuliandate`

linearize

Class: `Aero.FixedWing`
Package: `Aero`

Return linear state-space model

Syntax

```
linsys = linearize(aircraft,state)
linsys = linearize( ___ ,Name,Value)
```

Description

`linsys = linearize(aircraft,state)` returns a linear state-space representation of a fixed-wing `aircraft` linearized around a point given by `state`.

`linsys = linearize( ___ ,Name,Value)` returns the linear system using additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`aircraft` — `Aero.FixedWing` object

scalar

`Aero.FixedWing` object, specified as a scalar.

`state` — `Aero.FixedWing.State` object

scalar

`Aero.FixedWing.State` object, specified as a scalar.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'RelativePerturbation','1e-5'`

`RelativePerturbation` — Relative perturbation

`1e-5` (default) | scalar numeric

Relative perturbation of the system, specified as a scalar numeric. This perturbation takes the form of:

Perturbation Type	Definition
System State perturbation	`statePert = RelativePerturbation +1e-3*RelativePerturbation*|baseValue|`
System input perturbation	`ctrlPert = RelativePerturbation +1e-3*RelativePerturbation*|baseValue|`

To calculate the Jacobian of the system, `linearize` uses the result of these equations in conjunction with the `DifferentialMethod` property.

Example: `'RelativePerturbation',1e-5`

Data Types: `double`

`DifferentialMethod` — Direction while perturbing model
`'Forward'` (default) | `'Backward'` | `'Ceentral'`

Direction while perturbing model, specified as:

Direction	Description
`'Forward'`	Forward difference method that adds `statePert` and `ctrlPert` to the base states and inputs, respectively.
`'Backward'`	Backward difference method that adds `statePert` and `ctrlPert` to the base states an inputs, respectively.
`'Central'`	Central difference method that adds and subtracts `statePert` and `ctrlPert` to and from the base states and inputs, respectively.

Example: `'DifferentialMethod','Backward'`

Data Types: `char` | `string`

Output Arguments

`linsys` — Linear state-space model
scalar

Linear state-space model, returned as a scalar. The inputs and outputs of the state-space model depend on the degrees of freedom of the fixed-wing model and the number of control states on the model.

Examples

Calculate Linear State-Space Model

Calculate the linear state-space model of a Cessna 182 during cruise.

```
[C182, CruiseState] = astC182();
linSys = linearize(C182, CruiseState)
```

```
linSys =

  A =
                       XN          XE          XD           U           V           W
    XN                  0           0           0           1           0           0
    XE                  0           0           0           0           1           0
    XD                  0           0           0           0           0           1
    U                   0           0           0    -0.02574  -6.661e-10     0.08865
    V                   0           0           0           0     -0.1873           0
    W                   0           0           0     -0.2926  -7.183e-09      -2.115
    P                   0           0           0           0     -0.1375           0
    Q                   0           0           0     0.01265   3.331e-10    -0.07866
    R                   0           0           0           0     0.04268           0
    RollAngle           0           0           0           0           0           0
    PitchAngle          0           0           0           0           0           0
    YawAngle            0           0           0           0           0           0

                        P           Q           R   RollAngle  PitchAngle    YawAngle
    XN                  0           0           0           0     -0.0011     -0.0011
    XE                  0           0           0           0           0       220.1
    XD                  0           0           0           0      -220.1           0
    U                   0           0           0           0       -32.2           0
    V              -7.867           0      -197.7        32.2           0           0
    W                   0        -189           0   -0.000161   -0.000161           0
    P              -158.7           0       26.16           0           0           0
    Q                   0        -388           0           0           0           0
    R               -4.37           0      -14.87           0           0           0
    RollAngle           1           0           0           0           0           0
    PitchAngle          0           1           0           0           0           0
    YawAngle            0           0           1           0           0           0

  B =
                  Aileron   Elevator     Rudder  Propeller
    XN                  0          0          0          0
    XE                  0          0          0          0
    XD                  0          0          0          0
    U                   0          0          0       2215
    V                   0      19.62          0          0
    W                   0          0     -45.11          0
    P               75.07      4.819          0          0
    Q                   0          0     -42.84          0
    R              -7.963     -12.78          0          0
    RollAngle           0          0          0          0
    PitchAngle          0          0          0          0
    YawAngle            0          0          0          0

  C =
                       XN          XE          XD           U           V           W
    XN                  1           0           0           0           0           0
    XE                  0           1           0           0           0           0
    XD                  0           0           1           0           0           0
    U                   0           0           0           1           0           0
    V                   0           0           0           0           1           0
    W                   0           0           0           0           0           1
    P                   0           0           0           0           0           0
    Q                   0           0           0           0           0           0
    R                   0           0           0           0           0           0
    RollAngle           0           0           0           0           0           0
    PitchAngle          0           0           0           0           0           0
    YawAngle            0           0           0           0           0           0

                        P           Q           R   RollAngle  PitchAngle    YawAngle
    XN                  0           0           0           0           0           0
    XE                  0           0           0           0           0           0
    XD                  0           0           0           0           0           0
    U                   0           0           0           0           0           0
    V                   0           0           0           0           0           0
    W                   0           0           0           0           0           0
    P                   1           0           0           0           0           0
    Q                   0           1           0           0           0           0
    R                   0           0           1           0           0           0
    RollAngle           0           0           0           1           0           0
    PitchAngle          0           0           0           0           1           0
    YawAngle            0           0           0           0           0           1

  D =
                  Aileron   Elevator     Rudder  Propeller
    XN                  0          0          0          0
    XE                  0          0          0          0
    XD                  0          0          0          0
    U                   0          0          0          0
```

```
   V                 0          0          0          0
   W                 0          0          0          0
   P                 0          0          0          0
   Q                 0          0          0          0
   R                 0          0          0          0
   RollAngle         0          0          0          0
   PitchAngle        0          0          0          0
   YawAngle          0          0          0          0

Continuous-time state-space model.
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `forcesAndMoments` | `nonlinearDynamics` | `staticStability`

lla2ecef

Convert geodetic coordinates to Earth-centered Earth-fixed (ECEF) coordinates

Syntax

```
ecef = lla2ecef(lla)
ecef = lla2ecef(lla,model)
ecef = lla2ecef(lla,f,Re)
```

Description

`ecef = 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, `ecef`.

`ecef = lla2ecef(lla,model)` converts the coordinates for a specific ellipsoid planet.

`ecef = lla2ecef(lla,f,Re)` converts the coordinates for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters.

Examples

Determine ECEF Coordinates at Latitude, Longitude, and Altitude

Determine ECEF coordinates at a latitude, longitude, and altitude:

```
p = lla2ecef([0 45 1000])
```

```
p =
   1.0e+06 *

    4.5107    4.5107         0
```

Determine ECEF Coordinates at Multiple Latitudes, Longitudes, and Altitudes with WGS84 Ellipsoid Model

Determine ECEF coordinates at multiple latitudes, longitudes, and altitudes using the WGS84 ellipsoid model:

```
p = lla2ecef([0 45 1000; 45 90 2000], 'WGS84')
```

```
p =
   1.0e+06 *

    4.5107    4.5107         0
         0    4.5190    4.4888
```

Determine ECEF Coordinates at Multiple Latitudes, Longitudes, and Altitudes with Custom Ellipsoid Model

Determine ECEF coordinates at multiple latitudes, longitudes, and altitudes using the custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
p = lla2ecef([0 45 1000; 45 90 2000], f, Re)
```

```
p =
   1.0e+06 *

    2.4027    2.4027         0
         0    2.4096    2.3852
```

Input Arguments

`lla` — Geodetic coordinates

m-by-3 array

Geodetic coordinates (latitude, longitude and altitude), specified as an *m*-by-3 array 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. Altitude is above the planetary ellipsoid.

Data Types: `double`

`model` — Ellipsoid planet model

`'WGS84'` (default)

Ellipsoid planet model, specified as `'WGS84'`.

Data Types: `char` | `string`

`f` — Flattening

scalar

Flattening at each pole, specified as a scalar.

Data Types: `double`

`Re` — Planetary equatorial radius

scalar

Equatorial radius, specified as a scalar, in meters.

Data Types: `double`

Output Arguments

`ecef` — ECEF coordinates

m-by-3 array | vector

ECEF coordinates, returned as an *m*-by-3 array of ECEF coordinates.

Version History

Introduced in R2006b

See Also

`ecef2lla` | `geoc2geod` | `geod2geoc`

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 Coordinated Universal 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 Coordinated Universal 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` — Coordinated Universal Time

1-by-6 array | *M*-by-6 matrix

Coordinated Universal 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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`

Version History

Introduced in R2014a

See Also

`dcmeci2ecef` | `ecef2lla` | `eci2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef`

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 you define with `llo`, `psio`, and `href`.

`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`.

Examples

Estimate Coordinates at Latitude, Longitude, and Altitude

Estimate the coordinates at a latitude, longitude, and altitude:

```
p = lla2flat( [ 0.1 44.95 1000 ], [0 45], 5, -100 )
```

```
p =
   1.0e+04 *

    1.0530   -0.6509   -0.0900
```

Estimate Coordinates at Multiple Latitudes, Longitudes, and Altitudes with the WGS84 Ellipsoid Model

Estimate coordinates at multiple latitudes, longitudes, and altitudes with the WGS84 ellipsoid model:

```
p = lla2flat( [ 0.1 44.95 1000; -0.05 45.3 2000 ], [0 45], 5, -100, 'WGS84' )
```

```
p =
   1.0e+04 *
```

```
    1.0530   -0.6509   -0.0900
   -0.2597    3.3751   -0.1900
```

Estimate Coordinates at Multiple Latitudes, Longitudes, and Altitudes with a Custom Ellipsoid Model

Estimate coordinates at multiple latitudes, longitudes, and altitudes using 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+04 *

    0.5588   -0.3465   -0.0900
   -0.1373    1.7975   -0.1900
```

Input Arguments

`lla` — Geodetic coordinates

m-by-3 array

Geodetic coordinates (latitude, longitude, and altitude), specified as an *m*-by-3 array 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.

Data Types: `double`

`llo` — Reference location

m-by-2 array

Reference location of latitude and longitude, specified as an *m*-by-2 array, in degrees, for the origin of the estimation and the origin of the flat Earth coordinate system.

Data Types: `double`

`psio` — Angular direction of flat Earth

scalar

Angular direction of the flat Earth *x*-axis, specified as a scalar. The angular direction is the 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.

Data Types: `double`

`href` — Reference height

scalar

Reference height from the surface of the Earth to the flat Earth frame with regard to the flat Earth frame, specified as a scalar, in meters. `href` is positive down.

Data Types: `double`

`ellipsoidModel` — Ellipsoid planet model

`'WGS84'` (default)

Ellipsoid planet model, specified as `'WGS84'`.

Data Types: `char` | `string`

`flattening` — Flattening
scalar

Flattening at each pole, specified as a scalar.

Data Types: `double`

`equatorialRadius` — Planetary equatorial radius
scalar

Planetary equatorial radius, specified as a scalar, in meters.

Data Types: `double`

Output Arguments

`flatearth_pos` — Flat Earth position coordinates
3-element vector

Flat Earth position coordinates, specified as 3-element vector, in meters.

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 function 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 function 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:

$$R_N = \frac{R}{\sqrt{1-(2f-f^2)\sin^2\mu_0}},$$

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

$$dN = \frac{d\mu}{\text{atan}\left(\frac{1}{R_M}\right)},$$

and

$$dE = \frac{d\iota}{\text{atan}\left(\frac{1}{R_N \cos\mu_0}\right)}.$$

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

(ψ)

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}):

$$p_z = -h - h_{ref}.$$

Version History

Introduced in R2011a

References

[1] Etkin, B., *Dynamics of Atmospheric Flight*. New York: John Wiley & Sons, 1972.

See Also

Topics

`flat2lla`

load

Class: `Aero.Body`
Package: `Aero`

Get geometry data from source

Syntax

```
load(h,bodyDataSrc)
load(h,bodyDataSrc,geometrysource)
```

Description

`load(h,bodyDataSrc)` loads the graphics data from the body graphics file `bodyDataSrc`. This command assumes a default geometry source type set to `Auto`.

`load(h,bodyDataSrc,geometrysource)` loads the graphics data from the body graphics file, `bodyDataSrc`, into the face, vertex, and color data of the animation body object `h`. `geometrysource` specifies the body graphics file type. When axes `ax` is available, you can use this data to generate patches with `generatePatches`.

Input Arguments

`h` — Aerospace body object

`Aero.Body` object

Aerospace body object, specified as an `Aero.Body` object.

`bodyDataSrc` — Body graphics file

character vector | string

Body graphics file, specified as a character vector or string.

Data Types: `char` | `string`

`geometrysource` — Geometry source type for body

`Auto` (default) | `Variable` | `MatFile` | `Ac3d` | `Custom`

Geometry source type for body, specified as:

- `Auto`
- `Variable`
- `MatFile`
- `Ac3d`
- `Custom`

Setting this argument to `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.

Data Types: `char` | `string`

Examples

Load Graphic Data

Load the graphic data from the graphic data file, `pa24-250_orange.ac`, into body object, `b`.

```
b=Aero.Body;
b.load('pa24-250_orange.ac','Ac3d');
```

Version History

Introduced in R2007a

See Also

Classes

`Aero.Animation` | `Aero.Body`

Functions

`generatePatches`

machnumber

Compute Mach number using velocity and speed of sound

Syntax

```
mach = machnumber(velocities,speed_of_sound)
```

Description

`mach = machnumber(velocities,speed_of_sound)` computes Mach numbers, `mach`, from an *m*-by-3 array of Cartesian velocity vectors, `velocities`, and an array of *m* speeds of sound, `speed_of_sound`.

Examples

Determine Mach Number for Velocity and Speed of Sound in Feet per Second

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 Mach Number for Velocity and Speed of Sound in Meters per Second

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 Mach Number for Velocity and Speed of Sound in Knots

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
```

Input Arguments

`velocities` — Cartesian velocity vectors

m-by-3 array | vector

Cartesian velocity vectors, specified as an *m*-by-3 array. `velocities` and `speed_of_sound` must have the same length.

Data Types: `double`

`speed_of_sound` — Speed of sound

array

Speed of sound, specified as an array of *m* speeds of sound. `velocities` and `speed_of_sound` must have the same length.

Data Types: `double`

Output Arguments

`mach` — Mach numbers

scalar | array

Mach numbers, returned as a scalar or array of *m* Mach numbers.

Version History

Introduced in R2006b

See Also

`airspeed` | `alphabeta` | `dpressure`

mjuliandate

Modified Julian date calculator

Syntax

```
mjd = mjuliandate(datetime)
mjd = mjuliandate(dateVector)
mjd = mjuliandate(dateCharacterVector,format)

mjd = mjuliandate(year,month,day,'Action'=value)
mjd = mjuliandate([year,month,day],'Action'=value)
mjd = mjuliandate(year,month,day,hour,minute,second,'Action'=value)
mjd = mjuliandate([year,month,day,hour,minute,second],'Action'=value)
```

Description

`mjd = mjuliandate(datetime)` converts one or more `datetime` arrays to modified Julian date, `mjd`. Modified Julian dates begin at midnight rather than noon, and the first two digits of its corresponding Julian date are removed.

`mjd = mjuliandate(dateVector)` converts one or more date vectors, `dateVector`, to modified Julian date, `mjd`.

`mjd = mjuliandate(dateCharacterVector,format)` converts one or more date character vectors, `dateCharacterVector`, to modified Julian date, `mjd`, using format `format`.

`mjd = mjuliandate(year,month,day,'Action'=value)` and `mjd = mjuliandate([year,month,day],'Action'=value)` return the modified Julian date for corresponding elements of the `year,month,day` arrays.

`mjd = mjuliandate(year,month,day,hour,minute,second,'Action'=value)` and `mjd = mjuliandate([year,month,day,hour,minute,second],'Action'=value)` return the modified Julian date for corresponding elements of the `year,month,day,hour,minute,second` arrays. Specify the six arguments as 1-D arrays of the same length or scalar values.

Examples

Calculate Modified Julian Date Using `datetime` Array

Calculate the modified Julian date for February 4, 2016 from `datetime` array.

```
dt = datetime('04-02-2016','InputFormat','dd-MM-yyyy')
jd = mjuliandate(dt)
```

```
dt =
  datetime

   04-Feb-2016
```

```
jd =
       57422
```

Calculate Modified Julian Date Using Date Character Version and dd-mm-yyyy Format

Calculate the modified Julian date for May 24, 2005 using date character version and `dd-mm-yyyy` format:

```
mjd = mjuliandate('24-May-2005','dd-mmm-yyyy')
```

```
mjd =
       53514
```

Calculate Modified Julian Date Using Year, Month, and Day Inputs

Calculate modified Julian date for December 19, 2006 from year, month, and day inputs:

```
mjd = mjuliandate(2006,12,19)
```

```
mjd =
       54088
```

Calculate Modified Julian Date Using Year, Month, Day, Hour, Minute, and Second Inputs

Calculate the modified Julian date for October 10, 2004, at 12:21:00 p.m. using year, month, day, hour, month, and second inputs:

```
mjd = mjuliandate(2004,10,10,12,21,0)
```

```
mjd =
  5.3289e+004
```

Input Arguments

`datetime` — `datetime` array

m-by-1 array | 1-by-*m* array

`datetime` array, specified as an *m*-by-1 array or 1-by-*m* array.

`dateVector` — Full or partial date vector

m-by-6 matrix | *m*-by-3 matrix | positive double-precision number

Full or partial date vector, specified as an *m*-by-6 or *m*-by-3 matrix containing *m* full or partial date vectors, respectively:

- Full date vector — Contains six elements specifying the year, month, day, hour, minute, and second
- Partial date vector — Contains three elements specifying the year, month, and day

Data Types: `double`

`dateCharacterVector` — Date character vector
character array | 1-D array cell array of character vectors

Date character vector, specified as a character array, where each row corresponds to one date, or a 1-D array cell array of character vectors.

Data Types: `char` | `string`

`format` — Date format
`-1` (default) | character vector | string scalar | integer

Date format, specified as a character vector, string scalar, or integer. All dates in `dateCharacterVector` must have the same format and use the same date format symbols as the `datenum` function.

`mjuliandate` 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.

Data Types: `char` | `string`

`year` — Year
positive scalar | 1-D array | decimal

Year, specified as a positive scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `char` | `string`

`month` — Month
positive or negative scalar | 1-D array | decimal

Month, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`day` — Day
positive or negative scalar | 1-D array | decimal

Day, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`hour` — Date format

positive or negative scalar | 1-D array | decimal

Hour, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`minute` — Minute

positive or negative scalar | 1-D array | decimal

Minute, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`second` — Second

positive or negative scalar | 1-D array | decimal

Second, specified as a positive or negative scalar or 1-D array.

Dependencies

Depending on the syntax, specify `year`, `month`, and `day` or `year`, `month`, `day`, `hour`, `minute`, and `second` as 1-D arrays of the same length or scalar values.

Data Types: `double`

`'Action'=value` — Action

`'Warning'` (default) | `'Error'` | `'None'`

Action for noninteger decimal date input for `year`, `month`, `day`, `hour`, and `minute`, specified as:

- `'Error'` — Displays error and total number of noninteger date values.
- `'Warning'` — Displays warning and total number of adjusted noninteger date values. The function adjusts the noninteger value to its integer equivalent before calculating the value.
- `'None'` — Does not display warning or error. The function adjusts the noninteger value to its integer equivalent before calculating the value.

Data Types: `char` | `string`

Output Arguments

`mjd` — Modified Julian date

row vector | column vector

Modified Julian date, returned as a row vector or column vector of modified Julian dates. Modified Julian dates are the number of days and fractions since midnight Universal Time on November 17, 1858.

Dependencies

The output format depends on the input format:

Input Syntax	dy Format
`mjd = mjuliandate(dateVector)`	Row vector.
`mjd = mjuliandate(dateCharacterVector,format)`	Column vector.

Limitations

The calculation of modified Julian date does not take into account leap seconds.

Version History

Introduced in R2006b

R2022b: `mjuliandate` accepts decimal values

Behavior changed in R2022b

The `mjuliandate` function now accepts decimal values for these input arguments:

- `year`
- `month`
- `day`
- `hour`
- `minute`

See Also

`decyear` | `juliandate` | `leapyear` | `datenum` | `datestr`

moonLibration

Moon librations

Syntax

```
angles = moonLibration(ephemerisTime)
angles = moonLibration(ephemerisTime,ephemerisModel)
angles = moonLibration(ephemerisTime,ephemerisModel,action)

[angles,rates] = moonLibration( ___ )
```

Description

Implement Moon Libration Angles

`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.

Implement Moon Libration Angles and Rates

`[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: `double`

`action` — Function behavior
`'Error'` (default) | `'None'` | `'Warning'`

Function behavior when inputs are out of range, specified as one of these values, specified as one of these values.

Value	Description
`'None'`	No action.
`'Warning'`	Warning in the MATLAB Command Window and model simulation continues.
`'Error'`	MATLAB returns an exception and model simulation stops.

Data Types: `char` | `string`

Output Arguments

`angles` — Moon libration angles
M-by-3 numeric array

Moon libration angles, returned 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, returned 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.

Version History

Introduced in R2013a

See Also

`juliandate` | `earthNutation` | `planetEphemeris`

External Websites

https://ssd.jpl.nasa.gov/planets/eph_export.html

move

Class: `Aero.Body`
Package: `Aero`

Change animation body position and orientation

Syntax

```
move(h,translation,rotation)
```

Description

`move(h,translation,rotation)` sets a new position and orientation for the body object `h`.

Input Arguments

`h` — Aerospace body object

`Aero.Body` object

Aerospace body object, specified as an `Aero.Body` object.

`translation` — Translation matrix

1-by-3 matrix

Translation matrix in the aerospace body *x*-*y*-*z* coordinate system, specified as a 1-by-3 matrix.

Example:

Data Types: `double`

`rotation` — Body rotation

1-by-3 matrix

Body rotation, specified is a 1-by-3 matrix, in radians, that specifies the rotations about the right-hand *x*-*y*-*z* coordinate axes. The order of application of the rotation is *z*-*y*-*x* (yaw, pitch, and roll (*Y*-*P*-*R*)).

Data Types: `double`

Examples

Change Animation Body Position

Change the 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);
```

Version History

Introduced in R2007a

See Also

Classes
`Aero.Animation` | `Aero.Body`

Functions
`load`

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.

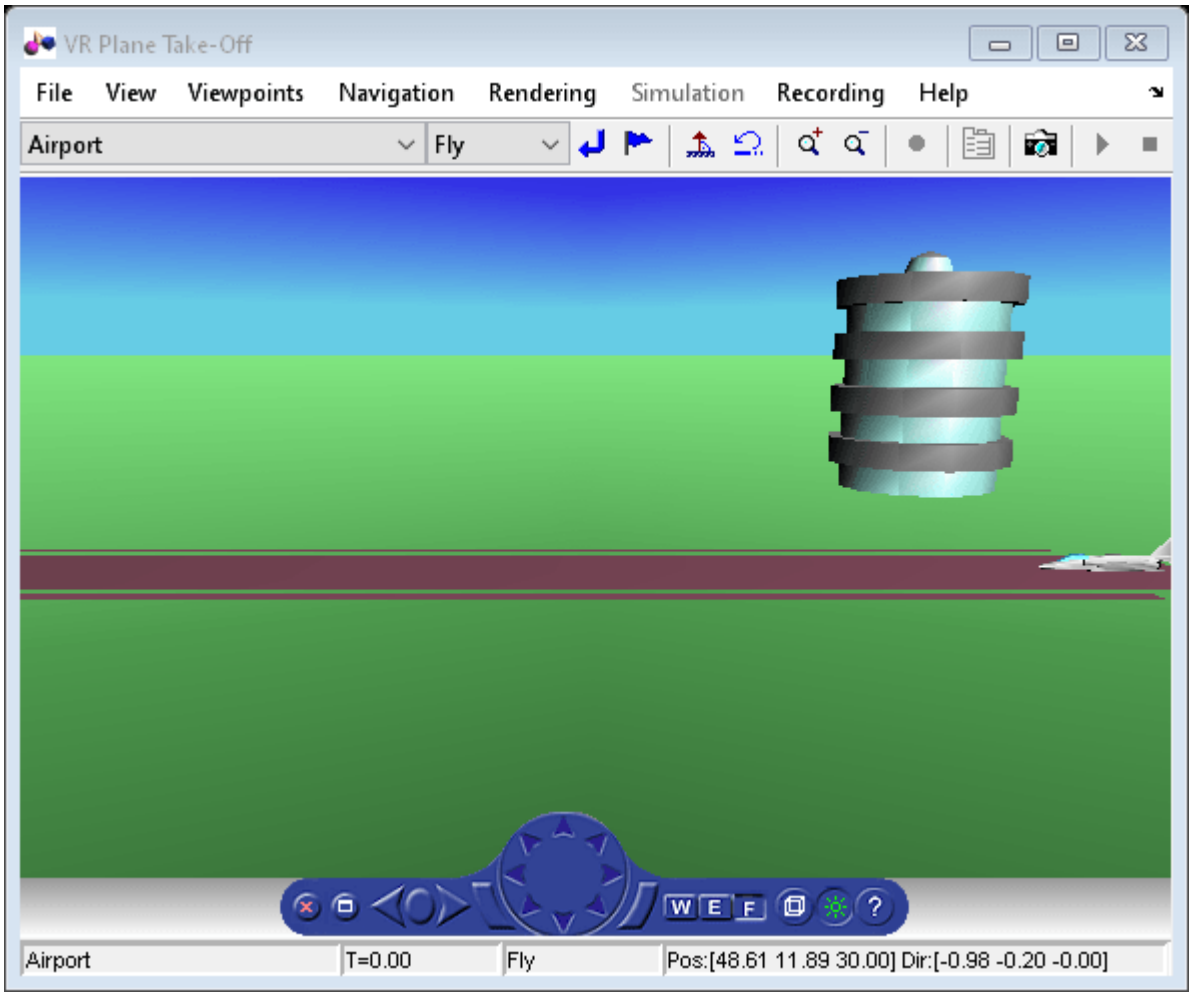
`rotation` is a 1-by-3 matrix specified as an Euler angle, in radians, that specifies the rotations about the right-hand *x*-*y*-*z* coordinate axes. The order of application of the rotation is *z*-*y*-*x* (yaw, pitch, and roll (*Y*-*P*-*R*)). This function uses the `CoordTransformFcn` to apply the translation 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 Lynx Body

Use the Simulink® 3D Animation™ `vrnode/getfield` function to retrieve the translation and rotation. These coordinates are those used in the animation software.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
```




```
newtrans = getfield(h.Nodes{2}.VRNode,'translation') - [0 40 6];
newrot = [0.4 0.3 0.1];
h.Nodes{2}.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`).

Version History

Introduced in R2007b

See Also

`addNode`

moveBody

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

Move body in animation object

Syntax

```
moveBody(h,idx,translation,rotation)
```

Description

`moveBody(h,idx,translation,rotation)` sets a new position and attitude for the body specified with the index `idx` in the animation object `h`. `translation` is a 1-by-3 vector in the aerospace body coordinate system. `rotation` is a 1-by-3 vector, in radians, that specifies the rotations about the right-hand *x*-*y*-*z* coordinate axes. The order of application of the rotation is *z*-*y*-*x* (yaw, pitch, and roll (*Y*-*P*-*R*)).

Input Arguments

`h` — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

`idx` — Body index

scalar

Body index, specified as a scalar.

`translation` — Body translation

1-by-3 vector

Body translation, specified as a 1-by-3 vector in the aerospace body coordinate system.

`rotation` — Body rotation

1-by-3 vector

Body rotation, specified as a 1-by-3 vector, in radians, that specifies the rotations about the right-hand *x*-*y*-*z* coordinate axes. The order of application of the rotation is *z*-*y*-*x* (*Y*-*P*-*R*).

Examples

Move `Aero.Animation` Body

Move an `Aero.Animation` body with the index 1 to the position offset from the original by `+ [0 0 -3]`.

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

```
pos1 = h.Bodies{1}.Position;
rot1 = h.Bodies{1}.Rotation;
moveBody(h,1,pos1 + [0 0 -3],rot1);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

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.

Version History

Introduced in R2007b

See Also

`Aero.Node`

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:

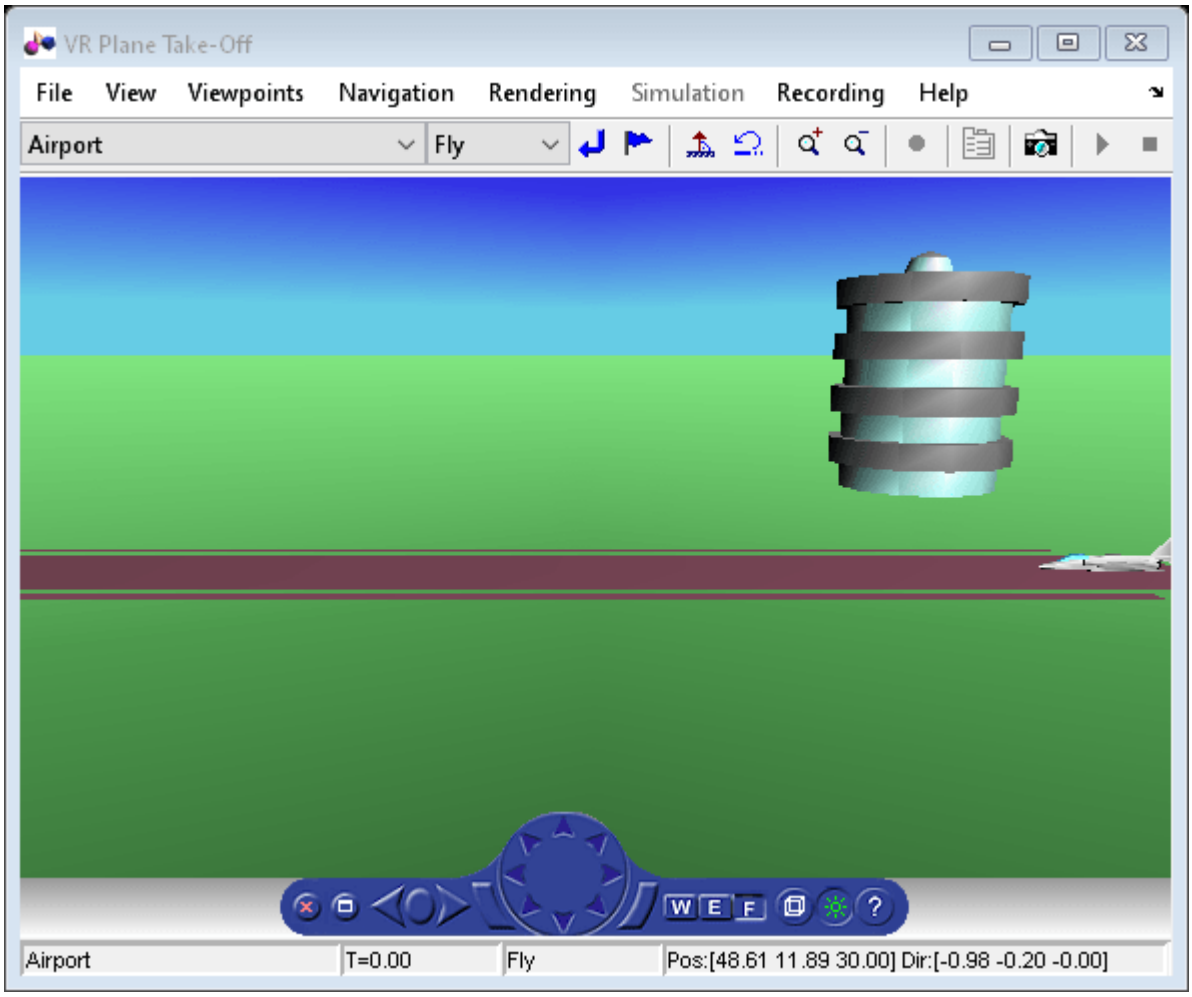
```
N{1,n} = Node Index
N{2,n} = Node Name
N{3,n} = Node Type
```

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.

ExamplesCreate List of Nodes

This example shows how to create a list of nodes associated with the virtual reality animation object, `h`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
```




```
h.nodeInfo;
```

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

Version History

Introduced in R2007b

See Also

`Aero.VirtualRealityAnimation` | `addNode`

nonlinearDynamics

Class: `Aero.FixedWing`
Package: `Aero`

Calculate dynamics of fixed-wing aircraft

Syntax

```
state_derivatives = nonlinearDynamics(aircraft,state)
```

Description

`state_derivatives = nonlinearDynamics(aircraft,state)` returns the column vector of `state_derivatives` of the fixed-wing aircraft `aircraft` from the initial state `state`.

Input Arguments

`aircraft` — `Aero.FixedWing` object

scalar

`Aero.FixedWing` object, specified as a scalar.

`state` — `Aero.FixedWing.State` object

scalar

`Aero.FixedWing.State` specified as a scalar.

Output Arguments

`state_derivatives` — State derivatives

vector

State derivatives with respect to time, returned as a vector. The rate vector size depends on the degrees of freedom, and is defined in the following form:

```
4th order point mass:
  DYDT(1) = dXN/dt
  DYDT(2) = dXD/dt
  DYDT(3) = dU/dt
  DYDT(4) = dW/dt

6th order point mass:
  DYDT(1) = dXN/dt
  DYDT(2) = dXE/dt
  DYDT(3) = dXD/dt
  DYDT(4) = dU/dt
  DYDT(5) = dV/dt
  DYDT(6) = dW/dt

3 DOF:
```

```
  DYDT(1) = dXN/dt
  DYDT(2) = dXD/dt
  DYDT(3) = dU/dt
  DYDT(4) = dW/dt
  DYDT(5) = dQ/dt
  DYDT(6) = dTheta/dt

6 DOF:
  DYDT(1) = dXN/dt
  DYDT(2) = dXE/dt
  DYDT(3) = dXD/dt
  DYDT(4) = dU/dt
  DYDT(5) = dV/dt
  DYDT(6) = dW/dt
  DYDT(7) = dP/dt
  DYDT(8) = dQ/dt
  DYDT(9) = dR/dt
  DYDT(10) = dPhi/dt
  DYDT(11) = dTheta/dt
  DYDT(12) = dPsi/dt
```

Examples

Calculate Dynamics of a Cessna 182:

Calculate the dynamics of Cessna 182.

```
[C182, CruiseState] = astC182();
dydt = nonlinearDynamics(C182, CruiseState)
```

```
dydt =

  220.1000
         0
         0
   -2.8323
         0
   -0.0040
         0
    1.3922
         0
         0
         0
         0
```

Limitations

When used with `Simulink.LookupTable` objects, this method requires a Simulink license.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `forcesAndMoments` | `linearize`

orbitalElements

Package: `matlabshared.satellitescenario`

Orbital elements of satellites in scenario

Syntax

```
elements = orbitalElements(sat)
```

Description

`elements = orbitalElements(sat)` returns the orbital elements of the specified satellite `sat`.

Examples

Retrieve Orbital Elements of Satellite

Create a satellite scenario object.

```
sc = satelliteScenario;
```

Add a satellite to the scenario.

```
tleFile = "eccentricOrbitSatellite.tle";
sat1 = satellite(sc,tleFile);
```

Retrieve the orbital elements of `sat1`.

```
elements1 = orbitalElements(sat1)
```

```
elements1 = struct with fields:
                      MeanMotion: 0.0083
                    Eccentricity: 0.7415
                     Inclination: 60.0000
    RightAscensionOfAscendingNode: 30.0000
              ArgumentOfPeriapsis: 280
                      MeanAnomaly: 289.4697
                           Period: 43200
                            Epoch: 05-May-2020 13:51:55
                            BStar: 0
```

Add a satellite from Keplerian elements to the scenario.

```
semiMajorAxis = 6878137;                                % meters
eccentricity = 0;
inclination = 20;                                       % degrees
rightAscensionOfAscendingNode = 0;                      % degrees
argumentOfPeriapsis = 0;                                % degrees
trueAnomaly = 0;                                        % degrees
sat2 = satellite(sc,semiMajorAxis,eccentricity, ...
       inclination,rightAscensionOfAscendingNode, ...
```

```
        argumentOfPeriapsis,trueAnomaly, ...
        "OrbitPropagator","two-body-keplerian", ...
        "Name","Sat2");
```

Retrieve the orbital elements of `sat2`.

```
elements2 = orbitalElements(sat2)
```

```
elements2 = struct with fields:
                    SemiMajorAxis: 6878137
                     Eccentricity: 0
                      Inclination: 20
    RightAscensionOfAscendingNode: 0
              ArgumentOfPeriapsis: 0
                      TrueAnomaly: 0
                           Period: 5.6770e+03
```

Input Arguments

`sat` — Satellite

scalar

Satellite object, specified as a scalar.

Output Arguments

`elements` — Orbital elements

structure

Orbital elements of the input `sat`, returned as a structure. The fields of the structure depend on the value of the `OrbitPropagator` property of the `satelliteScenario` object.

For more information on orbital elements, see “Orbital Elements” on page 2-70.

`Two-Body Keplerian` — Two-Body Keplerian orbit propagator fields

structure

The orbital elements are defined in the Geocentric Celestial Reference Frame (GCRF).

Field	Description
`SemiMajorAxis`	Semimajor axis, in meters
`Eccentricity`	Eccentricity
`Inclination`	Inclination angle, in degrees
`RightAscensionOfAscendingNode`	Right ascension of Ascending node, in degrees
`ArgumentOfPeriapsis`	Argument of periapsis, in degrees
`TrueAnomaly`	True anomaly at reference time, in degrees
`Period`	Orbital period, in seconds

Data Types: `struct`

`SGP4 and SDP4` — SGP4 and SDP4 orbit propagator fields
structure

The orbital elements represent general perturbation mean elements.

Field	Description
`MeanMotion`	General perturbation mean motion, in degrees per second
`Eccentricity`	Mean eccentricity
`Inclination`	Mean inclination angle, in degrees
`RightAscensionOfAscendingNode`	Mean right ascension of Ascending node, in degrees
`ArgumentOfPeriapsis`	Mean argument of periapsis, in degrees
`MeanAnomaly`	Mean mean anomaly at the reference time, in degrees
`Epoch`	Epoch
`BStar`	Drag term, in per EarthRadius
`Period`	Mean orbital period, in seconds

Data Types: `struct`

`Ephemeris` — Ephemeris orbit propagator fields
structure

Field	Description
`EphemerisStartTime`	Ephemeris start time
`EphemerisStopTime`	Ephemeris stop time
`PositionTimeTable`	Position timetable
`VelocityTimeTable`	Velocity timetable

Data Types: `struct`

`GPS` — GPS orbit propagator fields
structure

The orbital elements are derived from the SEM almanac file or RINEX GPS navigation message, and defined in the Earth-Centered-Earth-Fixed (ECEF) frame.

Field	Description
`PRN`	Pseudorandom noise number
`SatelliteHealth`	Satellite health
`GPSWeekNumber`	GPS week number, with time of ephemeris.
`GPSTimeOfApplicability`	Time of Applicability, in seconds
`SemiMajorAxis`	Semimajor axis, in meters
`Eccentricity`	Eccentricity

Field	Description
`Inclination`	Inclination angle at reference time, in degrees
`GeographicLongitudeOfOrbitalPlane`	Longitude of ascending node of orbit plane at weekly epoch, in degrees
`RateOfRightAscension`	Reference rate of right ascension, in degrees per second
`ArgumentOfPerigee`	Argument of perigee, in degrees
`MeanAnomaly`	Mean anomaly at the reference time, in degrees
`Period`	Orbital period, in seconds

Data Types: `struct`

`Galileo` — Galileo orbit propagator fields
structure

The orbital elements are derived from the RINEX Galileo navigation message, and defined in the Earth-Centered-Earth-Fixed (ECEF) frame.

Field	Description
`SatelliteID`	Satellite system number
`GALWeekNumber`	GAL week number
`TimeofEphemeris`	Time of ephemeris, in seconds
`SatelliteHealth`	Satellite health
`SemiMajorAxis`	Semimajor axis, in meters
`Eccentricity`	Eccentricity
`Inclination`	Inclination angle at reference time, in degrees
`GeographicLongitudeOfOrbitalPlane`	Longitude of ascending node of orbit plane at weekly epoch, in degrees
`RateOfRightAscension`	Reference rate of right ascension, in degrees per second
`ArgumentOfPerigee`	Argument of perigee, in degrees
`MeanAnomaly`	Mean anomaly at the reference time, in degrees
`Period`	Orbital period, in seconds

Data Types: `struct`

Data Types: `struct`

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`groundStation` | `conicalSensor` | `satellite`

Topics
“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

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

Implement Planet Ephemeris Position

`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.

Implement Planet Ephemeris Position and Velocity

`[position,velocity] = planetEphemeris( ___ )` implements the position and velocity of 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 positions are calculated, specified as one of these values:

- 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 (Julian epoch date) and the second column contains the elapsed days (elapsed day pairs).

Specify the Julian dates in Barycentric Dynamical Time (TDB).

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 position and velocity, specified as `'Sun'`, `'Mercury'`, `'Venus'`, `'Earth'`, `'Moon'`, `'Mars'`, `'Jupiter'`, `'Saturn'`, `'Uranus'`, `'Neptune'`, `'Pluto''SolarSystem'`, or `'EarthMoon'`.

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 position and velocity measurement, specified as `'Sun'`, `'Mercury'`, `'Venus'`, `'Earth'`, `'Moon'`, `'Mars'`, `'Jupiter'`, `'Saturn'`, `'Uranus'`, `'Neptune'`, `'Pluto''SolarSystem'`, or `'EarthMoon'`.

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, specified as one of these values.

Value	Description
`'None'`	No action.
`'Warning'`	Warning in the MATLAB Command Window and model simulation continues.
`'Error'`	MATLAB returns an exception and model simulation stops.

Data Types: `char`

Output Arguments

`position` — Position of target

M-by-3 vector

Position of the `target` object relative to the `center` object, returned as an *M*-by-3 vector, where *M* is the number of Julian dates. The 3 columns contain 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` — Velocity of target

M-by-3 vector

Velocity of the `target` object relative to 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.

Version History

Introduced in R2013a

References

[1] Folkner, W. M., J. G. Williams, and 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 (1998): 516–546.

[3] Vallado, David A., *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

See Also

`juliandate` | `moonLibration` | `earthNutation`

Topics

“Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-96
“Marine Navigation Using Planetary Ephemerides” on page 5-88

External Websites
https://ssd.jpl.nasa.gov/planets/eph_export.html

play

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

Animate `Aero.Animation` object given position/angle time series

Syntax

```
play(h)
```

Description

`play(h)` animates the loaded geometry in the animation object `h` for the current `TimeseriesDataSource` at the specified rate given by the `Aero.Animation` `'TimeScaling'` property in seconds of animation data per second of wall-clock time. The animation is created at a certain number of frames per second using the `'FramesPerSecond'` property. For more information on how the method uses timeseries data, see “Algorithms” on page 4-745.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

Examples

Record Animation Object Simulation

Simulate and record flight data. Use the data to create an animation object.

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

Control the frame display rate.

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

Set the time-scaling property `TimeScaling` on the animation object to specify the data per second.

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

The combination of the `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 = createBody(h,'pa24-250_orange.ac','Ac3d');
```

Load simulated flight trajectory data, `simdata`.

```
load simdata;
```

Set the time series data for the body.

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

Create a figure object for the animation object.

```
show(h);
```

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]
                 TCurrent: 0
              VideoRecord: 'on'
            VideoFileName: 'temp'
         VideoCompression: 'Motion JPEG AVI'
```

```
              VideoQuality: 50
               VideoTStart: NaN
               VideoTFinal: NaN
               TimeScaling: 5
          FramesPerSecond: 10
                   TStart: NaN
                   TFinal: NaN
```

```
h.VideoFilename = 'astMotion_JPEG';
```

Play the animation.

```
play(h);
```

Verify that a file named `astMotion_JPEG.avi` was created in the current folder.

Wait

Wait for the animation to stop playing before the modifying the object.

```
wait(h);
```

Disable recording to preserve the file.

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

Algorithms

The `Aero.Animation.play` interprets its time series data according to the `Aero.Body` `'TimeSeriesSourceType'` property, which can be one of these values.

`'Timeseries'`	MATLAB time series data with six values per time: `x y z phi theta psi` *x*, *y*, and *z* are Cartesian position coordinates. *phi*, *theta*, and *psi* are roll, pitch, and yaw angles. 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 the `interp1` function.

`'Array6DoF'`	A double-precision array in *n* rows and seven columns for 6DOF data: `time x y z phi theta psi`. If a double-precision array of eight or more columns is in `'TimeSeriesSource'`, the first seven columns are used as 6DOF data.
`'Array3DoF'`	A double-precision array in *n* rows and four columns for 3DOF data: `time x z theta`. If a double-precision array of five or more columns is in `'TimeSeriesSource'`, the first four columns are used as 3DOF data.
`'Custom'`	Position and angle data is retrieved from `'TimeSeriesSource'` by the currently registered `'TimeseriesReadFcn'`.

The `TStart` and `TFinal` values have these limitations:

- `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 `Aero.Animation.play` method uses a time advancement algorithm based on animation frames counted by ticks:

```
ticks = ticks + 1;
time  = tstart + ticks*FramesPerSecond*TimeScaling;
```

where

`TimeScaling`	Specifies the seconds of animation data per second of wall-clock time.
`FramesPerSecond`	Specifies 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. All angles are in radians.

Note If there is a 15% difference between the expected time advance and the actual time advance, this method generates this 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.
```

Version History

Introduced in R2007a

See Also

Aero.Animation

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')
```

Version History

Introduced in R2007a

See Also

`GenerateRunScript` | `initialize` | `update`

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.

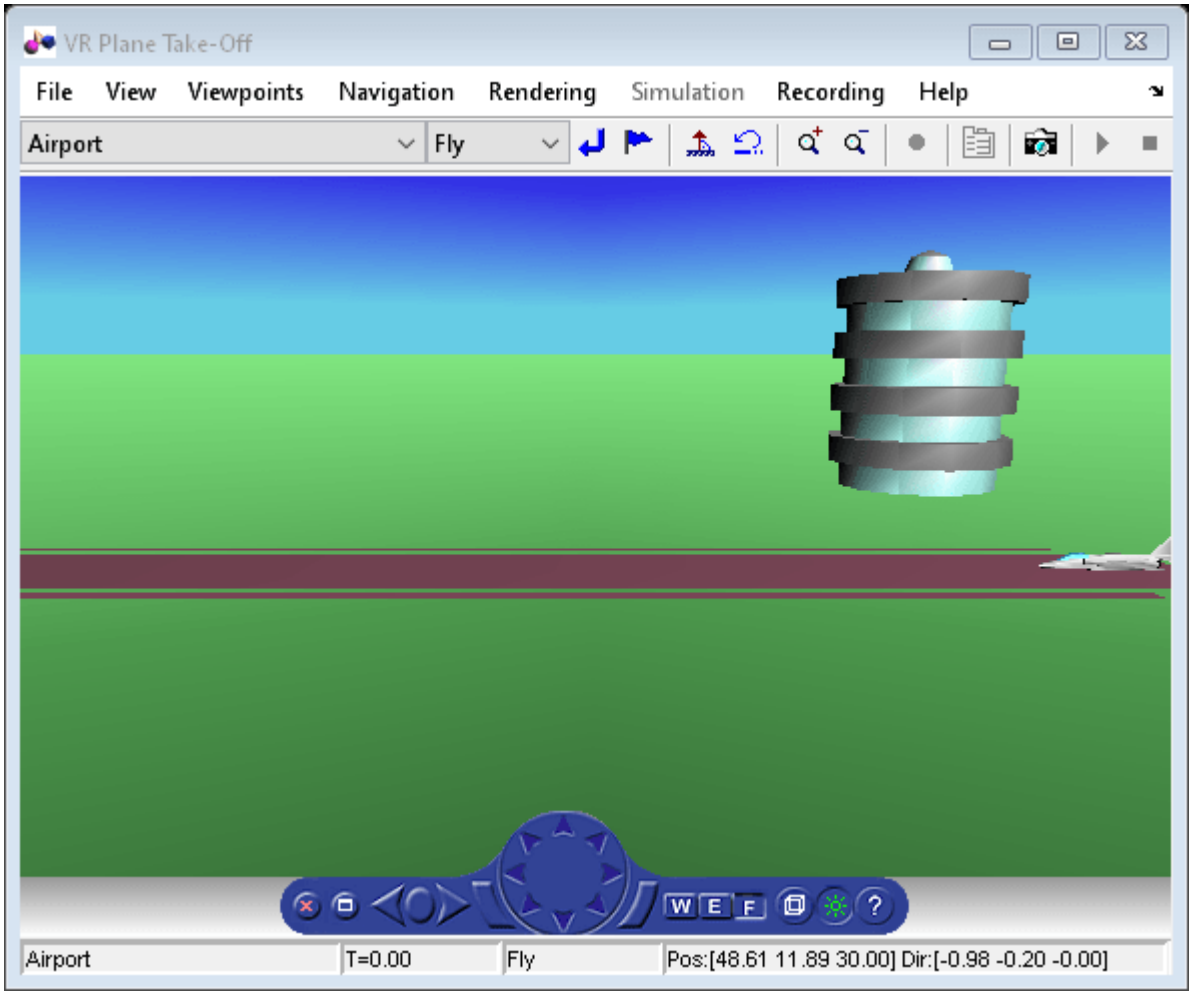
Time is in seconds, position values are in the same units as the geometry data loaded into the animation object, and all angles are in radians.

Examples

Animate Virtual Reality World

This example shows how to animate the virtual reality world, `asttkoff`.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = '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();
```

Version History

Introduced in R2007b

See Also

`initialize`

play

Package: `matlabshared.satellitescenario`

Play satellite scenario simulation results on viewer

Syntax

```
play(scenario)
play(viewer)
play(scenario,Name=Value)
```

Description

`play(scenario)` plays simulation results of the satellite scenario, `scenario`, on the Satellite Scenario Viewer. When the `AutoSimulate` property of the satellite scenario is `true`, the simulation is automatically performed from `StartTime` to `StopTime` using a step size specified by `SampleTime`, and the results are played on the viewer. Otherwise, the results calculated for `SimulationTime` are played on the viewer. Calling the `play` function enables the widgets on the viewer.

`play(viewer)` plays the satellite scenario simulation results on the Satellite Scenario Viewer specified by `viewer`.

`play(scenario,Name=Value)` specifies additional options using one or more name-value arguments.

Examples

Add Satellites to Scenario Using Keplerian Elements

Create a satellite scenario with a start time of 02-June-2020 8:23:00 AM UTC, and the stop time set to one day later. Set the simulation sample time to 60 seconds.

```
startTime = datetime(2020,6,02,8,23,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add two satellites to the scenario using their Keplerian elements.

```
semiMajorAxis = [10000000;15000000];
eccentricity = [0.01;0.02];
inclination = [0;10];
rightAscensionOfAscendingNode = [0;15];
argumentOfPeriapsis = [0;30];
trueAnomaly = [0;20];

sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
    rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly)
```

```
sat =
  1x2 Satellite array with properties:

    Name
    ID
    ConicalSensors
    Gimbals
    Transmitters
    Receivers
    Accesses
    GroundTrack
    Orbit
    OrbitPropagator
    MarkerColor
    MarkerSize
    ShowLabel
    LabelFontColor
    LabelFontSize
```

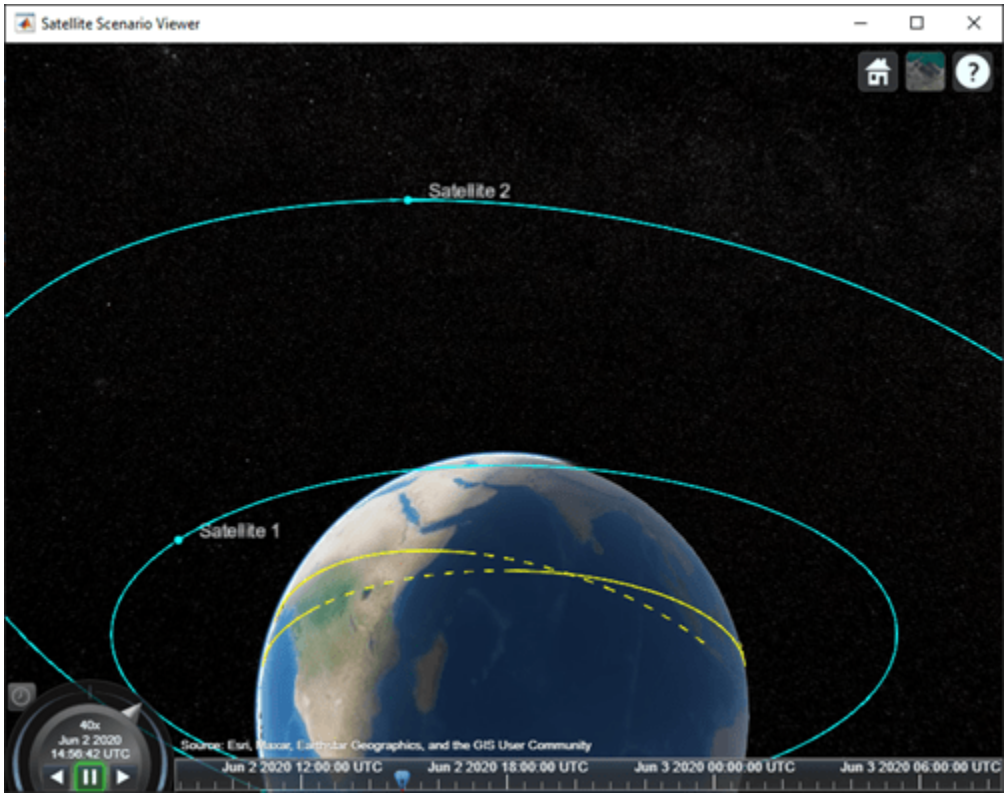
View the satellites in orbit and the ground tracks over one hour.

```
show(sat)
groundTrack(sat,LeadTime=3600)
```

```
ans=1×2 object
  1×2 GroundTrack array with properties:

    LeadTime
    TrailTime
    LineWidth
    LeadLineColor
    TrailLineColor
    VisibilityMode
```

Play the scenario and set the animation speed of the simulation to 40.

```
play(sc,PlaybackSpeedMultiplier=40)
```



Input Arguments

`scenario` — Satellite scenario

`satelliteScenario` object

Satellite scenario, specified as a `satelliteScenario` object.

`viewer` — Viewer playing simulation results

scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Viewer playing the simulation results, specified as a scalar `satelliteScenarioViewer` object or an array of `satelliteScenarioViewer` objects.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `PlaybackSpeedMultiplier=30` plays the animation 30 times faster than real time.

`Viewer` — Satellite Scenario Viewer

all viewers associated with `satelliteScenarioViewer` (default) | scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object or an array of `satelliteScenarioViewer` objects.

`PlaybackSpeedMultiplier` — Speed of animation
`50` (default) | positive scalar

Speed of animation in the viewer relative to real time, specified as a positive scalar.

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`hide` | `show` | `satellite` | `access` | `groundStation` | `restart`

Topics
“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

pointAt

Package: `matlabshared.satellitescenario`

Point satellite at target

Syntax

```
pointAt(sat,coordinates)
pointAt(sat,target)
pointAt(sat,'nadir')
pointAt(sat,attitudetable)
pointAt(sat,attitudetable,Name=Value)
pointAt(sat,attitudetimeseries)
pointAt(sat,attitudetimeseries,Name=Value)

pointAt(gimbal,'none')
pointAt(gimbal,coordinates)
pointAt(gimbal,target)
pointAt(gimbal,'nadir')
pointAt(gimbal,steeringtable)
pointAt(gimbal,steeringtimeseries)
```

Description

Satellite Object

`pointAt(sat,coordinates)` steers the satellites in the vector `sat` toward the geographical coordinates [latitude; longitude; altitude] specified by `coordinates`.

`pointAt(sat,target)` steers the satellites specified by `sat` toward the specified `target`. The input `target` can be another satellite or ground station.

`pointAt(sat,'nadir')` steers the satellites specified by the row vector `sat` toward the nadir direction. `'nadir'` is the default pointing direction.

`pointAt(sat,attitudetable)` sets the attitude of the satellite `sat` such that it follows the attitudes provided in `attitudetable`, which is a MATLAB `timetable` object.

`pointAt(sat,attitudetable,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous `attitudetable` syntax. For example, to interpret the provided attitude values as the rotation from the Geocentric Celestial Reference Frame (GCRF) to the body frame, set `CoordinateFrame` to `inertial`.

`pointAt(sat,attitudetimeseries)` sets the attitudes of the satellite `sat` such that it follows the attitude provided in `attitudetimeseries`, which is a MATLAB `timeseries` object.

`pointAt(sat,attitudetimeseries,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous `attitudetimeseries` syntax. For example, to interpret the provided attitude values as the rotation from the GCRF to the body frame, set `CoordinateFrame` to `inertial`.

Gimbal Object

`pointAt(gimbal,'none')` sets the gimbal angles (gimbal azimuth and gimbal elevation) of the gimbals in the vector `gimbal` to zero. This setting is the default.

`pointAt(gimbal,coordinates)` steers the gimbals in the vector `gimbal` toward the geographical coordinates [latitude; longitude; altitude] specified by `coordinates`.

`pointAt(gimbal,target)` steers the gimbals in the vector `gimbal` toward the specified `target`.

`pointAt(gimbal,'nadir')` steers the gimbals specified by the row vector `gimbal` toward the nadir direction of their parents, namely, their parent's latitude, longitude, and 0 meter altitude.

`pointAt(gimbal,steeringtable)` sets the orientation of the gimbals to align with the azimuth and elevation angles provided in `steeringtable`, which is a MATLAB `timetable` object.

`pointAt(gimbal,steeringtimeseries)` sets the orientation of the gimbals to align with the azimuth and elevation angles provided in `steeringtimeseries`, which is MATLAB `timeseries` object.

Examples

Steer Ground Station Gimbal to Point at Satellite

Create a satellite scenario object.

```
startTime = datetime(2021,6,10);                        % 10 June 2021, 12:00 AM UTC
stopTime = datetime(2021,6,11);                         % 11 June 2021, 12:00 AM UTC
sampleTime = 60;                                        % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add a satellite to the scenario.

```
semiMajorAxis = 10000000;                               % meters
eccentricity = 0;
inclination = 10;                                       % degrees
rightAscensionOfAscendingNode = 0;                      % degrees
argumentOfPeriapsis = 0;                                % degrees
trueAnomaly = 0;                                        % degrees
sat = satellite(sc,semiMajorAxis,eccentricity, ...
    inclination,rightAscensionOfAscendingNode, ...
    argumentOfPeriapsis,trueAnomaly);
```

Add a ground station to the scenario.

```
latitude = 42.3501;                          % degrees
longitude = -71.3504;                        % degrees
gs = groundStation(sc,latitude,longitude);
```

Add a gimbal to the ground station.

```
g = gimbal(gs,MountingLocation=[0; 0; -1],MountingAngles=[0; 180; 0]);
```

Add a conical sensor to the gimbal.

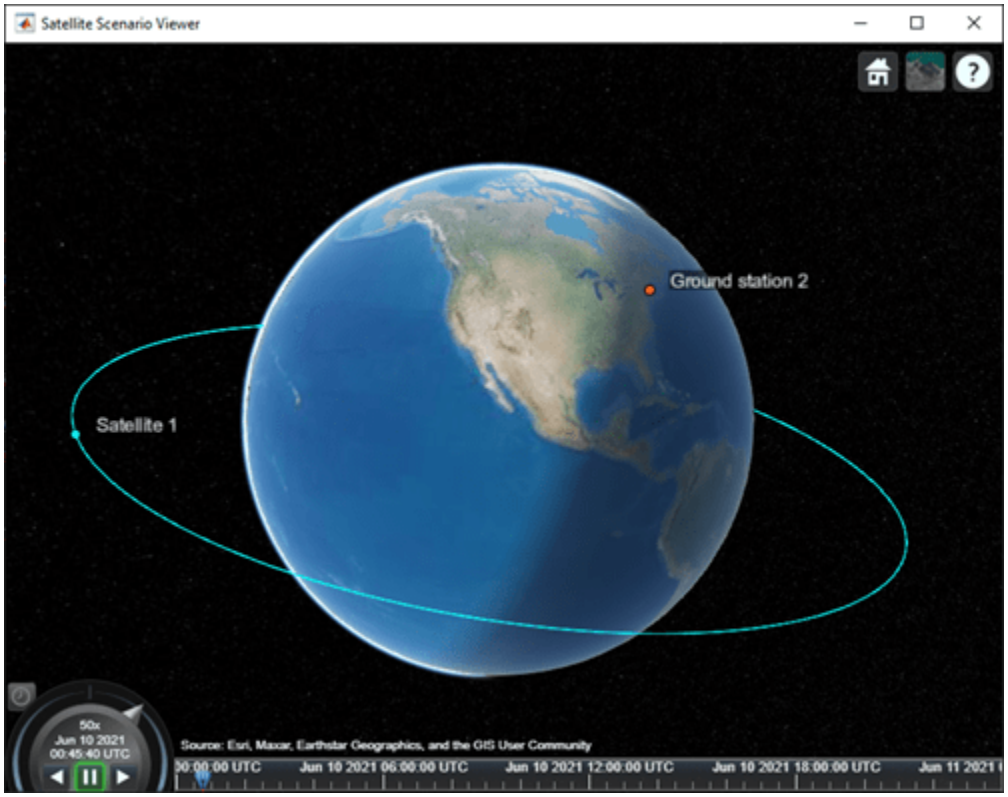
```
c = conicalSensor(g,MountingLocation=[0; 0; 0.5]);
```

Point the gimbal at the satellite.

```
pointAt(g,sat);
```

Visualize the scenario by using the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```

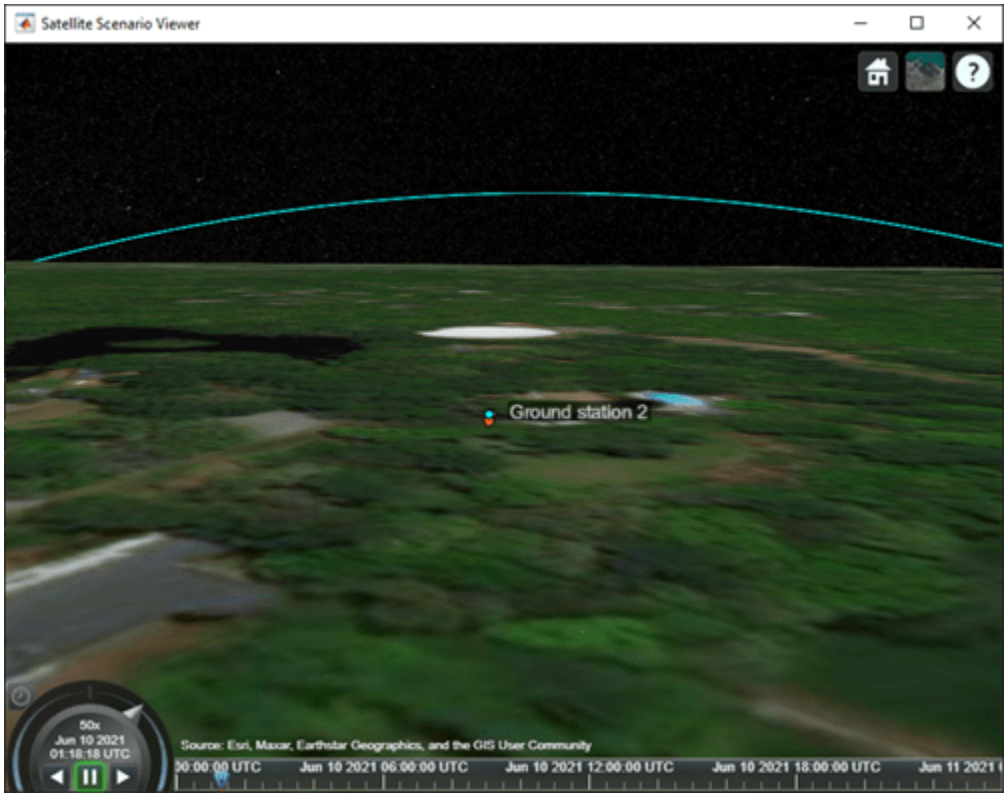


Play the scenario.

```
play(sc);
```

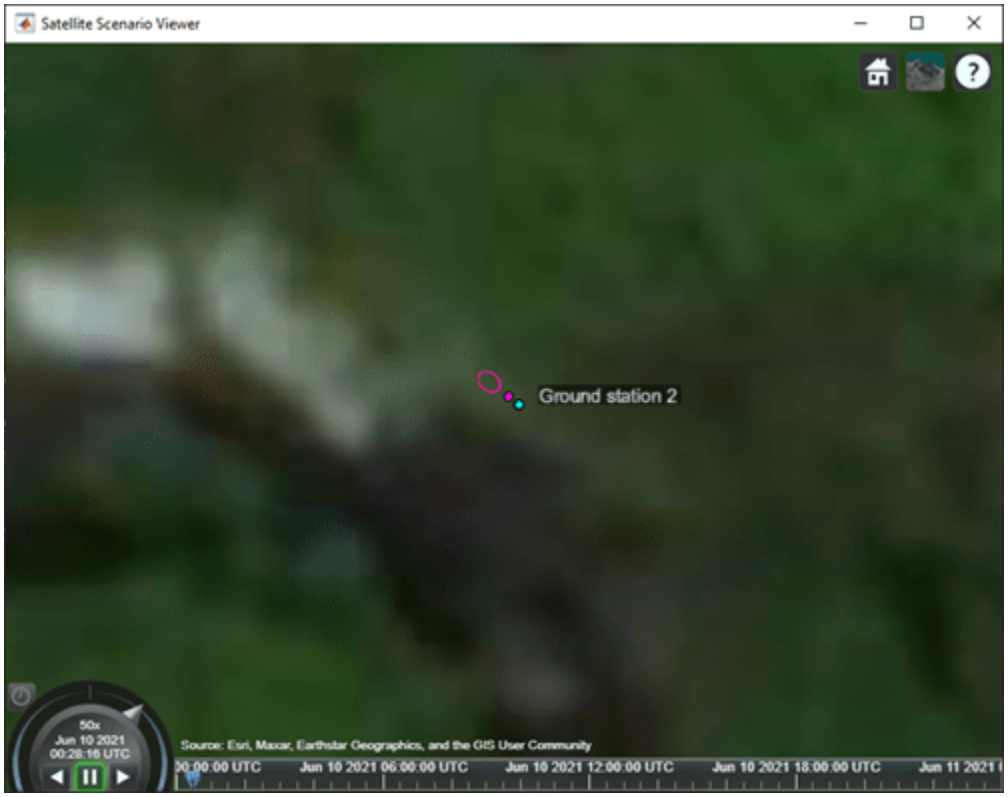
Set the ground station as the camera target.

```
camtarget(v,gs);
```



Visualize the field of view of the conical sensor and observe the change in orientation of the conical sensor.

```
fieldOfView(c);
```



Input Arguments

`sat` — Satellite

scalar | vector

Satellite object, specified as either a scalar or a vector.

`gimbal` — Gimbal

scalar | vector

Gimbal object, specified as either a scalar or a vector.

`coordinates` — Geographical coordinates of satellite or gimbal target

three-element vector | 2-D array

Geographical coordinates of the satellite or gimbal target, specified as a three-element vector or a 2-D array.

- When the coordinates are a three-element vector, the elements of the vector correspond to the latitude, longitude, and altitude, in that order, and all satellites or gimbals are steered to point at this location.
- When the coordinates are a 2-D array, the number of rows must equal 3 and the number of columns must equal the number of satellites in `satellite` or the number of gimbals in `gimbal`. The rows correspond to the latitude, longitude, and altitude, in that order, and each column represents the pointing coordinates of the corresponding satellite in the vector `satellite` or

gimbal in the vector `gimbal`. The latitudes and longitudes are specified in degrees and the altitudes are specified in meters, which represent the height above the surface of the Earth.

`target` — Target
scalar | vector

Target at which input `satellite` or `gimbal` is pointed, specified as a scalar or a vector. The input `target` can be another satellite or a ground station.

- When `target` is a scalar, all satellites or gimbals point to the specified `target`.
- When `target` is a vector, the length of `target` must equal the number of satellites in `sat` or the number of gimbals in `gimbal`. Each element in `target` represents the pointing target of a satellite in `sat` or a gimbal in `gimbal`.

`attitudetable` — MATLAB `timetable` for satellite attitude
`timetable` object

MATLAB `timetable` with exactly one monotonically increasing column of `rowTimes` (datetime or duration).

- If `sat` contains a single satellite, the table must contain one data column of scalar-first quaternions [1-by-4] or ZYX Euler angles [1-by-3].
- If `sat` is an array of satellites, each data row must contain either:
  - Multiple columns, where each column contains data for an individual satellite over time.
  - One column of 2-D data, where the length of one dimension must equal 3 or 4, depending on whether Euler angles or quaternions are used, and the remaining dimension must have length equal to the number of satellites in `sat`.
  - One column of 3-D data, where the length of one dimension must equal 3 or 4, depending on whether Euler angles or quaternions are used, one dimension is a singleton, and the remaining dimension must have length equal to the number of satellites in `sat`.

Euler angles represent passive, intrinsic rotations in degrees, using the ZYX rotation order. If the provided `rowTimes` are of type `duration`, time values are measured relative to the current scenario `StartTime` property.

The function assumes that satellite attitudes represent the transformation from the GCRF to the body frame, unless you specify a `CoordinateFrame` name-value argument. For scenario timesteps outside of the time range of `attitudetable`, the function uses `nadir` by default, unless you specify an `ExtrapolationMethod` name-value argument.

`attitudetimeseries` — MATLAB `timeseries` for satellite attitude
`timeseries` object

MATLAB `timeseries` containing scalar-first quaternions or ZYX Euler angles.

- If the `Data` property of `timeseries` has two dimensions, the length of one dimension must equal 3 or 4, depending on whether Euler angles or quaternions are used, and the other dimension must align with the orientation of the time vector.
- If `sat` is an array of satellites, the `Data` property of `timeseries` must have three dimensions where the length of one dimension must equal 3 or 4, depending on whether Euler angles or quaternions are used, either the first or the last dimension must align with the orientation of the time vector, and the remaining dimension must align with the number of satellites in `sat`.

Euler angles represent passive, intrinsic rotations in degrees, using the ZYX rotation order. When `timeseries.TimeInfo.StartDate` is empty, time values are measured relative to the current scenario `StartTime` property.

The function assumes that satellite attitudes represent the transformation from the Geocentric Celestial Reference Frame (GCRF) to the body frame, unless you specify a `CoordinateFrame` name-value argument. For scenario timesteps outside of the time range of `attitudetable`, the function uses `nadir` by default, unless you specify an `ExtrapolationMethod` name-value argument.

`steeringtable` — MATLAB `timetable` for satellite steering
`timetable` object

MATLAB `timetable` with exactly one monotonically increasing column of `rowTimes` (datetime or duration).

- If `gimbal` contains a single gimbal, the table must contain one data column of azimuth and elevation angles in degrees [1-by-2].
- If `gimbal` is an array of gimbals, each data row must contain either:
  - Multiple columns, where each column contains data for an individual gimbal over time.
  - One column of 2-D data, where the length of one dimension must equal 2 and the remaining dimension must have length equal to the number of gimbals in `gimbal`.
  - One column of 3-D data, where the length of one dimension must equal 2, one dimension is a singleton, and the remaining dimension must have length equal to the number of gimbals in `gimbal`.

Specify the azimuth and elevation angles in degrees. If the provided `rowTimes` are of type `duration`, time values are measured relative to the current scenario `StartTime` property.

`steeringtimeseries` — MATLAB `timeseries`for satellite steering
`timeseries` object

MATLAB `timeseries` timeseries containing azimuth and elevation in degrees [1-by-2].

- If the `Data` property of `timeseries` has two dimensions, the length of one dimension must equal 2 and the other dimension must align with the orientation of the time vector.
- If `gimbal` is an array of gimbals, the `Data` property of `timeseries` must have three dimensions where:
  - The length of one dimension must equal 2.
  - Either the first or the last dimension must align with the orientation of the time vector.
  - The remaining dimension must align with the number of gimbals in `gimbal`.

When `timeseries.TimeInfo.StartDate` is empty, time values are measured relative to the current scenario `StartTime` property.

Name-Value argument Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `pointAt(sat,attTT,CoordinateFrame="inertial")` interprets the provided attitude values as the rotation from the Geocentric Celestial Reference Frame (GCRF) to the body frame.

`CoordinateFrame` — Coordinate frame of custom attitude inputs
`inertial` (default) | `ecef` | `ned`

Coordinate frame of custom attitude inputs, specified as one of these options.

- `inertial` — Interprets the provided attitude values as the rotation from the GCRF to the body frame.
- `ecef` — Interprets the provided attitude values as the rotation from the Earth-Centered-Earth-Fixed (ECEF) frame to the body frame.
- `ned` — Interprets the provided attitude values as the rotation from the North-East-Down (NED) frame to the body frame.

Data Types: `char` | `string`

`ExtrapolationMethod` — Default behavior for attitude
`nadir` (default) | `fixed`

Default behavior for attitude, specified as:

- `nadir` — Sets the attitude of the satellite `sat` such that the yaw axis points in the nadir direction.
- `fixed` — Keeps the attitude constant with respect to the GCRF at the closest time value for which data is provided in the custom attitude data.

The scenario uses this setting for scenario time steps that lie outside the provided custom attitude time range. If you do not provide `ExtrapolationMethod`, the function returns a warning when the scenario time is out of range of the custom attitude time range.

Data Types: `char` | `string`

`Format` — Format of attitude data provided
`quaternion` (default) | `euler`

Format of attitude data provided, specified as one of these options.

- `quaternion` — Interprets the provided attitude values as scalar-first quaternions. Quaternions represent passive rotations from `CoordinateFrame` to the body frame.
- `euler` — Interprets the provided attitude values as Euler angles, in degrees. Euler angles represent passive, intrinsic rotations from `CoordinateFrame` to the body frame using the ZYX rotation order and are provided in that order.

Data Types: `char` | `string`

Note When the `AutoSimulate` property of the satellite scenario is `false`, you can call the `pointAt` function as long as the `SimulationStatus` is `NotStarted` or `InProgress`.

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`show` | `play` | `hide` | `access` | `groundStation` | `conicalSensor`

Topics
“Modeling Custom Satellite Attitude and Gimbal Steering” on page 5-204
“Satellite Scenario Key Concepts” on page 2-65

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 Coordinated Universal 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

R2020b: Updated `aeroiersdata.mat` file
Behavior changed in R2020b

The contents of the `aeroiersdata.mat` file have been updated. Correspondingly, the output of this function will have different results when using the default value (`'aeroiersdata.mat'`) as the value of `Source`. The results reflect more accurate external data from the International Earth Rotation and Reference Systems Service (IERS).

See Also

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

Topics
“Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-96

quat2angle

Convert quaternion to rotation angles

Syntax

```
[rotationAng1 rotationAng2 rotationAng3] = quat2angle(q)
[rotationAng1 rotationAng2 rotationAng3] = quat2angle(q,s)
```

Description

`[rotationAng1 rotationAng2 rotationAng3] = quat2angle(q)` calculates the set of rotation angles, `rotationAng1`, `rotationAng2`, `rotationAng3`, for a given quaternion, `q`. The rotation used in this function is a passive transformation between two coordinate systems.

`[rotationAng1 rotationAng2 rotationAng3] = quat2angle(q,s)` calculates the set of rotation angles `rotationAng1`, `rotationAng2`, `rotationAng3` for a given quaternion, `q`, and a specified rotation sequence, `s`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention. This function normalizes all quaternion inputs.

Examples

Determine Rotation Angles from Quaternion

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 Rotation Angles from Multiple Quaternions and Rotation Order

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
```

Input Arguments

q — Quaternion

m-by-4 matrix | each element of *q* must be real number

Quaternion, specified as an *m*-by-4 matrix containing *m* quaternions. `q` has its scalar number as the first column.

Data Types: `double`

s — Rotation order

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

Rotation order for three rotation angles, where `Z` is in the *z*-axis, `Y` is in the *y*-axis, and `X` is in the *x*-axis.

Data Types: `char` | `string`

Output Arguments

rotationAng1 — First rotation angles

m-array

First rotation angles, returned as an *m*-array, in radians.

rotationAng2 — Second rotation angles

m-array

Second rotation angles, returned as an *m*-array, in radians.

rotationAng3 — Third rotation angles

m-array

Third rotation angles, returned as an *m*-array, in radians.

Limitations

- The `'ZYX'`, `'ZXY'`, `'YXZ'`, `'YZX'`, `'XYZ'`, and `'XZY'` implementations generate a `rotationAng2` angle that lies between ±90 degrees, and `rotationAng1` and `rotationAng3` angles that lie between ±180 degrees.
- The `'ZYZ'`, `'ZXZ'`, `'YXY'`, `'YZY'`, `'XYX'`, and `'XZX'` implementations generate a `rotationAng2` angle that lies between 0 and 180 degrees, and `rotationAng1` and `rotationAng3` angles that lie between ±180 degrees.

Version History

Introduced in R2007b

See Also

angle2dcm | angle2quat | dcm2angle | dcm2quat | quat2dcm

quat2dcm

Convert quaternion to direction cosine matrix

Syntax

```
dcm = quat2dcm(q)
```

Description

`dcm = quat2dcm(q)` calculates the direction cosine matrix, `n`, for a given quaternion, `q`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention. This function normalizes all quaternion inputs.

Examples

Determine Direction Cosine Matrix from Single Quaternion

Determine the direction cosine matrix from the single quaternion q = [1 0 1 0].

```
dcm = quat2dcm([1 0 1 0])
```

```
dcm = 3×3

         0         0   -1.0000
         0    1.0000         0
    1.0000         0         0
```

Determine Direction Cosine Matrices from Multiple Quaternions

Determine the direction cosine matrices from multiple quaternions.

```
q = [1 0 1 0; 1 0.5 0.3 0.1];
dcm = quat2dcm(q)
```

```
dcm =
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
```

Input Arguments

q — Quaternion

m-by-4 matrix

Quaternion, specified as an *m*-by-4 matrix containing *m* quaternions. Each element of *q* must be a real number.

Data Types: `double`

Output Arguments

dcm — Direction cosine matrices

3-by-3-by-*m* matrix

Direction cosine matrices, returned as a 3-by-3-by-*m* matrix, where *m* is the number of direction cosine matrices. The direction cosine matrix performs the coordinate transformation of a vector in inertial axes to a vector in body axes.

Version History

Introduced in R2006b

See Also

`angle2dcm` | `dcm2angle` | `dcm2quat` | `angle2quat` | `quat2angle` | `quatrotate`

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`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention. This function normalizes all quaternion inputs.

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.

Version History

Introduced in R2017a

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` | `rod2quat` | `rod2angle` | `rod2dcm`

quatconj

Calculate conjugate of quaternion

Syntax

```
n = quatconj(q)
```

Description

`n = quatconj(q)` calculates the conjugate `n` for a given quaternion, `q`. For more information on the quaternion and quaternion conjugate forms, see “Algorithms” on page 4-777.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

Examples

Determine Conjugate

Determine the conjugate of `q = [1 0 1 0]`.

```
conj = quatconj([1 0 1 0])
```

```
conj = 1×4

     1     0    -1     0
```

Input Arguments

q — quaternion matrix

m-by-4 matrix of real numbers

Quaternion matrix, specified in an *m*-by-4 matrix of real numbers containing *m* quaternions.

Example: `[1 0 1 0]`

Data Types: `double`

Output Arguments

n — conjugate matrix

m-by-4 matrix

Conjugate matrix, returned in an *m*-by-4 matrix.

Algorithms

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

$$q^{'} = q_0 - \mathbf{i}q_1 - \mathbf{j}q_2 - \mathbf{k}q_3 .$$

Version History

Introduced in R2006b

References

[1] Stevens, Brian L. and Frank L. Lewis. *Aircraft Control and Simulation*. 2nd ed. Wiley–Interscience, 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

`quatdivide` | `quatinv` | `quatmod` | `quatmultiply` | `quatnorm` | `quatnormalize` | `quatrotate`

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`. For more information on the input and output quaternion forms, see “Algorithms” on page 4-780.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

Examples

Determine Division of Two 1-by-4 Quaternions

Divide one 1-by-4 quaternions by another 1-by-4 quaternion.

```
q = [1 0 1 0];
r = [1 0.5 0.5 0.75];
d = quatdivide(q, r)
```

```
d = 1×4

    0.7273    0.1212    0.2424   -0.6061
```

Determine Division of 2-by-4 Quaternion

Divide 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 = 2×4

    0.7273    0.1212    0.2424   -0.6061
    1.2727    0.0121   -0.7758   -0.4606
```

Input Arguments

q — Numerator quaternion

m-by-4 matrix of real numbers | 1-by-4 matrix of real numbers

Numerator quaternion, specified in a *m*-by-4 matrix of real numbers containing *m* quaternions or a 1-by-4 matrix of reall numbers containing one quaternion.

Example: `[1 0 1 0]`

Data Types: `double`

r — Denominator quaternion

m-by-4 matrix of real numbers | 1-by-4 matrix of real numbers

Denominator quaternion, specified in a *m*-by-4 matrix of real numbers containing *m* quaternions or a 1-by-4 matrix of real numbers containing one quaternion.

Example: `[1 0.5 0.5 0.75]`

Data Types: `double`

Output Arguments

n — Quaternion quotients

m-by-4 matrix of real numbers

Quaternion quotients, returned in an *m*-by-4 matrix of real numbers.

Algorithms

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

$$t_0 = \frac{(r_0 q_0 + r_1 q_1 + r_2 q_2 + r_3 q_3)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$

$$t_1 = \frac{(r_0 q_1 - r_1 q_0 - r_2 q_3 + r_3 q_2)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$

$$t_2 = \frac{(r_0 q_2 + r_1 q_3 - r_2 q_0 - r_3 q_1)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$

$$t_3 = \frac{(r_0 q_3 - r_1 q_2 + r_2 q_1 - r_3 q_0)}{r_0^2 + r_1^2 + r_2^2 + r_3^2} .$$

Version History

Introduced in R2006b

References

[1] Stevens, Brian L. and Frank L. Lewis. *Aircraft Control and Simulation*. 2nd ed. Wiley–Interscience, 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` | `quatinv` | `quatmod` | `quatmultiply` | `quatnorm` | `quatnormalize` | `quatrotate`

quatexp

Exponential of quaternion

Syntax

```
qe=quatexp(q)
```

Description

`qe=quatexp(q)` calculates the exponential, `qe`, for the specified quaternion, `q`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

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.

Version History

Introduced in R2016a

References

[1] Dam, Erik B., Martin Koch, Martin Lillholm. "Quaternions, Interpolation, and Animation." University of Copenhagen, København, Denmark, 1998.

See Also

`quatinterp` | `quatlog` | `quatpower` | `quatconj` | `quatdivide` | `quatinv` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatrotate`

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.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

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 [1].

- `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.

Version History

Introduced in R2016a

References

[1] Dam, Erik B., Martin Koch, Martin Lillholm. "Quaternions, Interpolation, and Animation." University of Copenhagen, København, Denmark, 1998.

See Also

`quatlog` | `quatexp` | `quatpower` | `quatconj` | `quatdivide` | `quatinv` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatrotate`

quatinv

Calculate inverse of quaternion

Syntax

```
n = quatinv(q)
```

Description

`n = quatinv(q)` calculates the inverse, `n`, for a given quaternion, `q`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention. For more information on quaternion forms, see “Algorithms” on page 4-787.

Examples

Determine Inverse

Determine the inverse of `q = [1 0 1 0]`:

```
qinv = quatinv([1 0 1 0])
```

```
qinv =

    0.5000         0   -0.5000         0
```

Input Arguments

q — Quaternion

m-by-4 matrix | real number

Quaternion, specified as an *m*-by-4 matrix containing *m* quaternions.

Data Types: `double`

Output Arguments

n — Inverse of quaternion

m-by-4 matrix

Inverse of quaternion, returned as an *m*-by-4 matrix.

Algorithms

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

$$q^{-1} = \frac{q_0 - \mathbf{i}q_1 - \mathbf{j}q_2 - \mathbf{k}q_3}{q_0^2 + q_1^2 + q_2^2 + q_3^2}.$$

Version History

Introduced in R2006b

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` | `quatconj` | `quatmod` | `quatmultiply` | `quatnorm` | `quatrotate` | `quatnormalize`

quatlog

Natural logarithm of quaternion

Syntax

```
ql=quatlog(q)
```

Description

`ql=quatlog(q)` calculates the natural logarithm, `ql`, for a normalized quaternion, `q`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

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.

Version History

Introduced in R2016a

References

[1] Dam, Erik B., Martin Koch, Martin Lillholm. "Quaternions, Interpolation, and Animation." University of Copenhagen, København, Denmark, 1998.

See Also

`quatinterp` | `quatexp` | `quatpower` | `quatconj` | `quatdivide` | `quatinv` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatrotate`

quatmod

Calculate modulus of quaternion

Syntax

```
n = quatmod(q)
```

Description

`n = quatmod(q)` calculates the modulus `n` for a given quaternion, `q`. For more information on the quaternion and quaternion modulus forms, see “Algorithms” on page 4-791.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

Examples

Determine Modulus of Quaternion

Determine the modulus of q = [1 0 0 0].

```
mod = quatmod([1 0 0 0])
```

```
mod = 1
```

Input Arguments

q — Quaternion

m-by-4 matrix of real numbers

Quaternion, specified in a *m*-by-4 matrix of real numbers containing *m* quaternions.

Example: `[1 0 0 0]`

Data Types: `double`

Output Arguments

n — Moduli

column vector of `m` elements

Moduli, returned as a column vector of `m` elements.

Algorithms

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

$$|q| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}.$$

Version History

Introduced in R2006b

References

[1] Stevens, Brian L. and Frank L. Lewis. *Aircraft Control and Simulation*. 2nd ed. Wiley–Interscience, 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` | `quatmultiply` | `quatnorm` | `quatnormalize` | `quatrotate`

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`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

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

$$n_0 = (r_0q_0 - r_1q_1 - r_2q_2 - r_3q_3)$$
$$n_1 = (r_0q_1 + r_1q_0 - r_2q_3 + r_3q_2)$$
$$n_2 = (r_0q_2 + r_1q_3 + r_2q_0 - r_3q_1)$$
$$n_3 = (r_0q_3 - r_1q_2 + r_2q_1 + r_3q_0)$$

Version History

Introduced in R2006b

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`

quatnorm

Calculate norm of quaternion

Syntax

```
norm = quatnorm(q)
```

Description

`norm = quatnorm(q)` calculates the norm `norm` for a given quaternion, `q`. For more information on the quaternion and quaternion norm forms, see “Algorithms” on page 4-796.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

Examples

Determine Norm of Quaternion

Determine the norm of `q = [.5 -.5 .5 0]`.

```
norm=quatnorm([.5 -.5 .5 0])
```

```
norm = 0.7500
```

Input Arguments

q — quaternion matrix
m-by-4 matrix of real numbers

Quaternion matrix, specified in an *m*-by-4 matrix of real numbers containing *m* quaternions.

Example: `[1 0 0 0]`

Data Types: `double`

Output Arguments

norm — Norms
column vector

Norms, returned as a column vector of *m* norms.

Algorithms

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

$$norm(q) = q_0^2 + q_1^2 + q_2^2 + q_3^2 .$$

Version History

Introduced in R2006b

References

[1] Stevens, Brian L. and Frank L. Lewis. *Aircraft Control and Simulation*. 2nd ed. Wiley–Interscience, 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` | `quatmultiply` | `quatnormalize` | `quatrotate`

quatnormalize

Normalize quaternion

Syntax

```
normalized_q = quatnormalize(q)
```

Description

`normalized_q = quatnormalize(q)` calculates the normalized quaternion, normalized `n`, for a given quaternion, `q`. For more information on the quaternion and normalized quaternion forms, see “Algorithms” on page 4-796.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

Examples

Normalize Quaternion

Normalize q = [1 0 1 0].

```
normal = quatnormalize([1 0 1 0])
```

```
normal = 1×4

    0.7071         0    0.7071         0
```

Input Arguments

q — quaternion matrix

m-by-4 matrix of real numbers

Quaternion matrix, specified in an *m*-by-4 matrix of real numbers containing *m* quaternions.

Example: `[1 0 0 0]`

Data Types: `double`

Output Arguments

normalized_q — Normalized quaternions

m-by-4 matrix

Normalized quaternions, returned in an *m*-by-4 matrix.

Algorithms

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

$$normal(q) = \frac{q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3}{\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}} .$$

Version History

Introduced in R2006b

References

[1] Stevens, Brian L. and Frank L. Lewis. *Aircraft Control and Simulation*. 2nd ed. Wiley–Interscience, 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.

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`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

$q^t = \exp(t \cdot \log(q))$, 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.

Version History

Introduced in R2016a

References

[1] Dam, Erik B., Martin Koch, Martin Lillholm. "Quaternions, Interpolation, and Animation." University of Copenhagen, København, Denmark, 1998.

See Also

`quatexp` | `quatinterp` | `quatlog` | `quatconj` | `quatdivide` | `quatinv` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatrotate`

quatrotate

Rotate vector by quaternion

Syntax

```
n = quatrotate(q,r)
```

Description

`n = quatrotate(q,r)` calculates the resulting vector following the passive rotation of initial vector `r` by quaternion `q` and returns a final vector `n`. If quaternions are not yet normalized, the function normalizes them.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention. This function normalizes all quaternion inputs.

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 — Initial vector

m-by-3 matrix | 1-by-3 array

Initial 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 — Final vector

m-by-3 matrix

Final vector, returned as an *m*-by-3 matrix.

More About

q

The normalized 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:

$$v = \mathbf{i}v_1 + \mathbf{j}v_2 + \mathbf{k}v_3$$

The Aerospace Toolbox defines a passive quaternion rotation of the form:

$$v' = q^{-1} \otimes \left[\frac{0}{v}\right] \otimes q,$$

where Ⓧ is the operator of a quaternion multiplication.

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.

Version History

Introduced in R2006b

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`

Topics

“Passive Transformations” on page 2-3

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');
```

Version History

Introduced in R2007a

removeBody

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

Remove one body from animation

Syntax

```
h = removeBody(h,idx)
```

Description

`h = removeBody(h,idx)` removes the body specified by the index `idx` from the animation object `h`.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

idx — Body index

scalar

Body index, specified as a scalar.

Examples

Remove `Aero.Animation` Body

Remove the body identified by the index, `1`.

```
h = Aero.Animation;
idx1 = createBody(h,'pa24-250_orange.ac','Ac3d');
h = removeBody(h,1);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

removeNode (Aero.VirtualRealityAnimation)

Remove node from virtual reality animation object

Syntax

```
removeNode(h,node)
h.removeNode(node)
```

Description

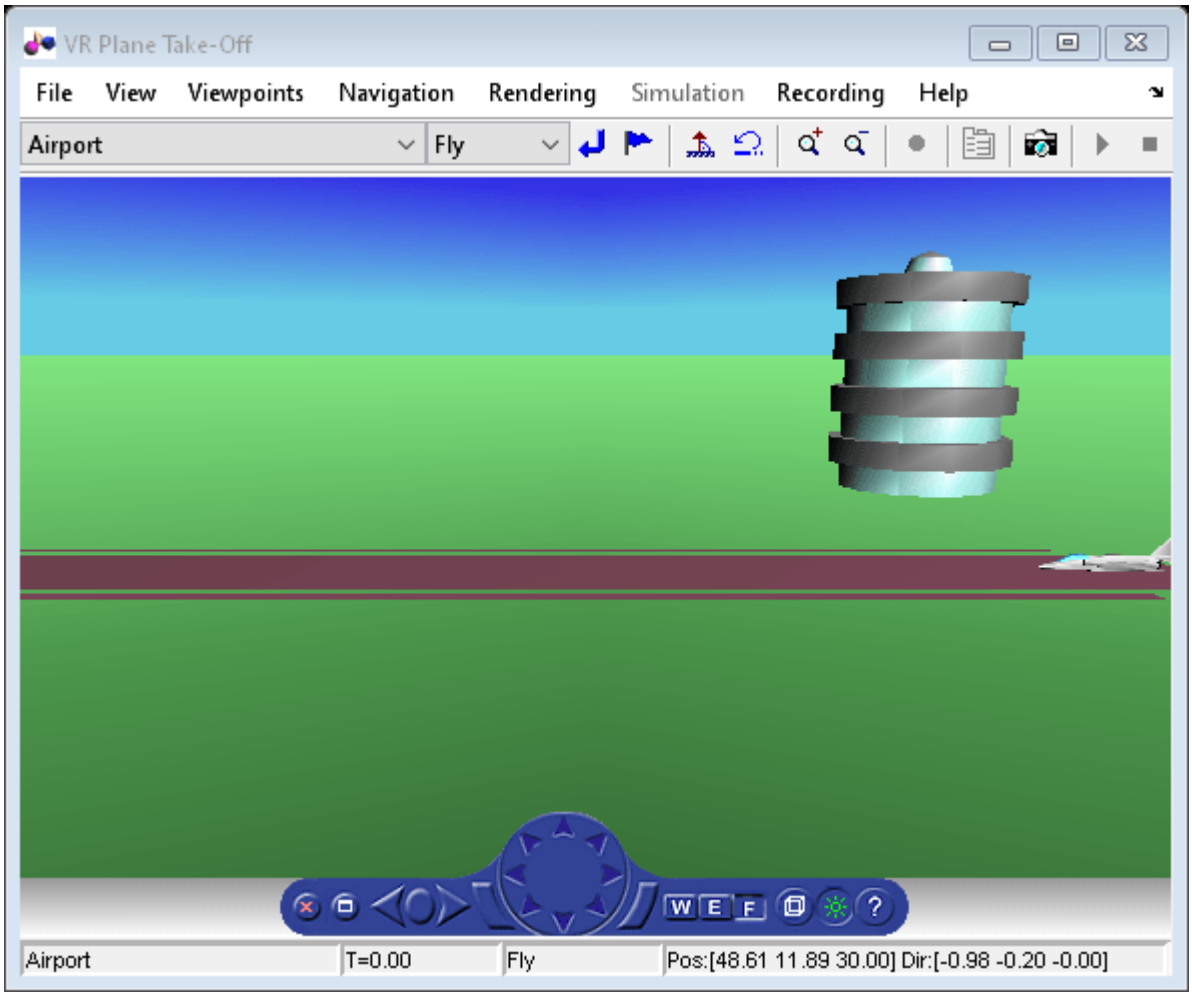
`removeNode(h,node)` and `h.removeNode(node)` remove the node specified by `node` from the virtual reality animation object `h`. `node` can be either the node name or the node index. This function can remove only one node at a time.

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.

ExamplesRemove Node

This example shows how to remove the node, `Lynx1`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
h.addNode('Lynx1','chaseHelicopter.wrl');
h.removeNode('Lynx1');
```



Version History

Introduced in R2007b

See Also

`Aero.VirtualRealityAnimation` | `addNode`

removeViewpoint (Aero.VirtualRealityAnimation)

Remove viewpoint node from virtual reality animation

Syntax

```
removeViewpoint(h,viewpoint)
h.removeViewpoint(viewpoint)
```

Description

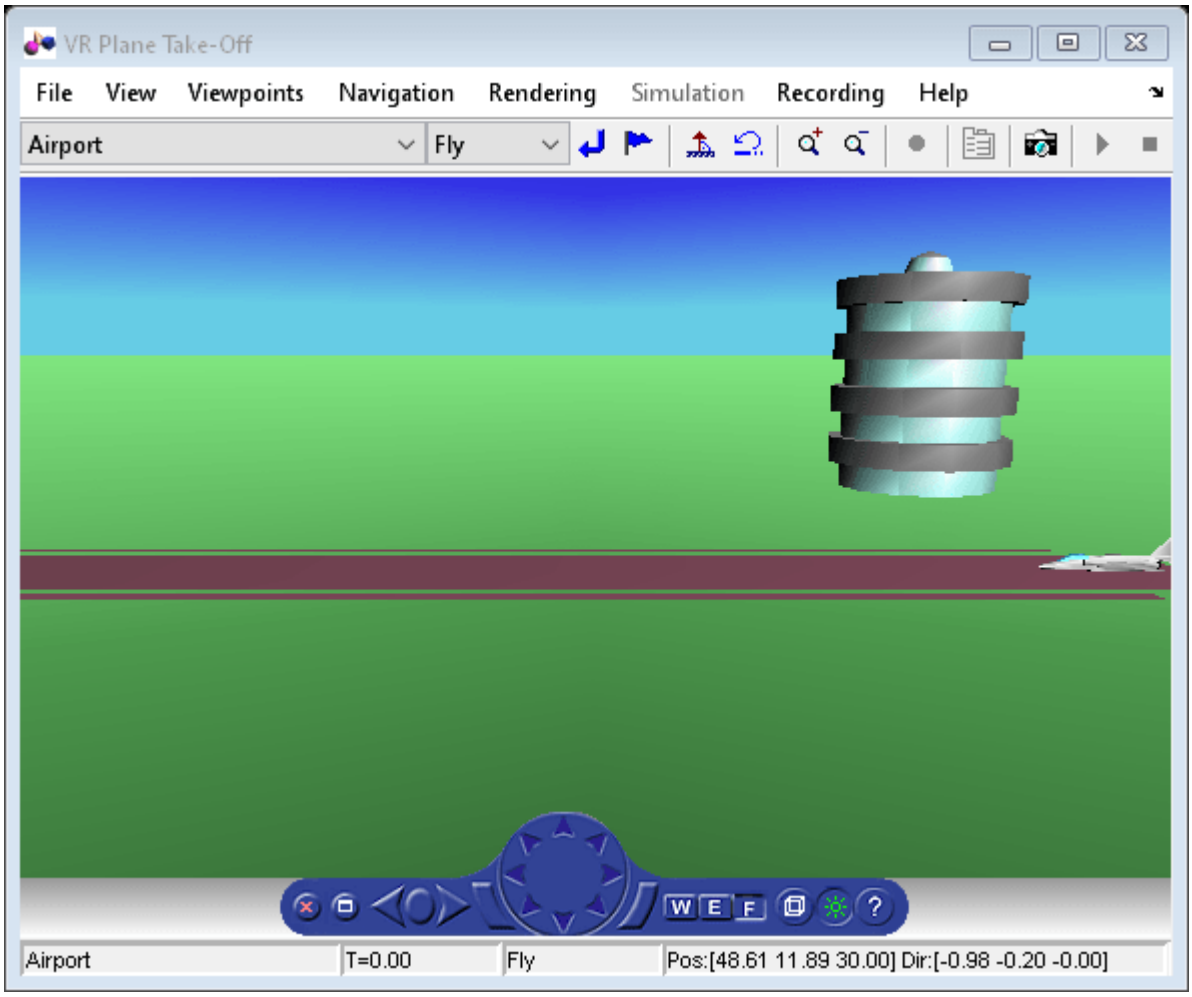
`removeViewpoint(h,viewpoint)` and `h.removeViewpoint(viewpoint)` remove the viewpoint specified by `viewpoint` from the virtual reality animation object `h`. `viewpoint` can be either the viewpoint name or the viewpoint index. This function can remove only one viewpoint at a time.

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.

ExamplesRemove Viewpoint, Lynx1

This example shows how to remove the viewpoint, `Lynx1`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
h.addViewpoint(h.Nodes{2}.VRNode,'children','chaseView','View From Helicopter');
h.removeViewpoint('chaseView');
```



Version History

Introduced in R2007b

See Also

`Aero.VirtualRealityAnimation` | `addViewpoint`

restart

Restart simulation from beginning

Syntax

```
restart(sc)
```

Description

`restart(sc)` resets the satellite scenario `sc` to the initial start time.

Examples

Manual Simulation of Satellite Scenario

Create a satellite scenario object and set the `AutoSimulate` property to `false` to enable manual simulation of the satellite scenario.

```
startTime = datetime(2022,1,12);
stopTime = startTime + days(0.5);
sampleTime = 60;                                % Seconds
sc = satelliteScenario('AutoSimulate',false);
```

Add a GPS satellite constellation to the scenario.

```
sat = satellite(sc,"gpsAlmanac.txt");
```

Simulate the scenario using the `advance` function.

```
while advance(sc)
end
```

Obtain the satellite position histories.

```
p = states(sat);
```

`AutoSimulate` is `false`, so restart the scenario before adding a ground station.

```
restart(sc);
```

Add a ground station to the scenario.

```
gs = groundStation(sc);
```

Add access analysis between each satellite and ground station.

```
ac = access(sat,gs);
```

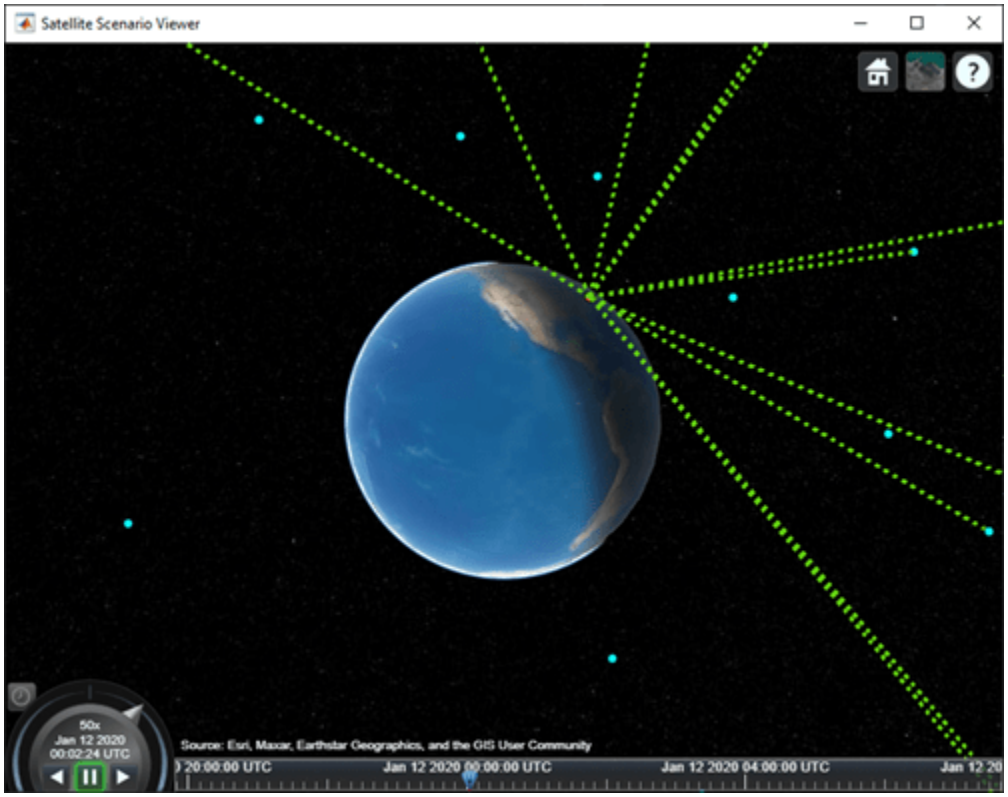
Simulate the scenario and determine the access intervals.

```
while advance(sc)
end
intvls1 = accessIntervals(ac)
```

```
intvls1=35×8 table
     Source           Target           IntervalNumber         StartTime              EndTime
    ________    ____________________    ______________    ____________________    ____________

    "PRN:1"     "Ground station 32"           1          11-Jan-2020 23:20:25    12-Jan-2020 05:1
    "PRN:2"     "Ground station 32"           1          12-Jan-2020 04:03:16    12-Jan-2020 07:4
    "PRN:3"     "Ground station 32"           1          11-Jan-2020 19:50:06    11-Jan-2020 21:3
    "PRN:3"     "Ground station 32"           2          12-Jan-2020 01:52:43    12-Jan-2020 06:4
    "PRN:4"     "Ground station 32"           1          11-Jan-2020 19:50:06    12-Jan-2020 00:1
    "PRN:4"     "Ground station 32"           2          12-Jan-2020 04:54:02    12-Jan-2020 07:4
    "PRN:5"     "Ground station 32"           1          12-Jan-2020 05:52:03    12-Jan-2020 07:4
    "PRN:6"     "Ground station 32"           1          12-Jan-2020 02:43:29    12-Jan-2020 07:4
    "PRN:7"     "Ground station 32"           1          11-Jan-2020 21:09:52    12-Jan-2020 03:2
    "PRN:8"     "Ground station 32"           1          11-Jan-2020 20:33:36    12-Jan-2020 03:1
    "PRN:9"     "Ground station 32"           1          11-Jan-2020 19:50:06    12-Jan-2020 00:4
    "PRN:9"     "Ground station 32"           2          12-Jan-2020 05:08:32    12-Jan-2020 07:4
    "PRN:10"    "Ground station 32"           1          12-Jan-2020 00:32:56    12-Jan-2020 01:5
    "PRN:11"    "Ground station 32"           1          11-Jan-2020 22:15:09    12-Jan-2020 04:3
    "PRN:12"    "Ground station 32"           1          12-Jan-2020 04:32:16    12-Jan-2020 07:4
    "PRN:13"    "Ground station 32"           1          12-Jan-2020 00:03:56    12-Jan-2020 02:5
      ⋮
```

Visualize the simulation results.

```
v = satelliteScenarioViewer(sc,'ShowDetails',false);
play(sc);
```



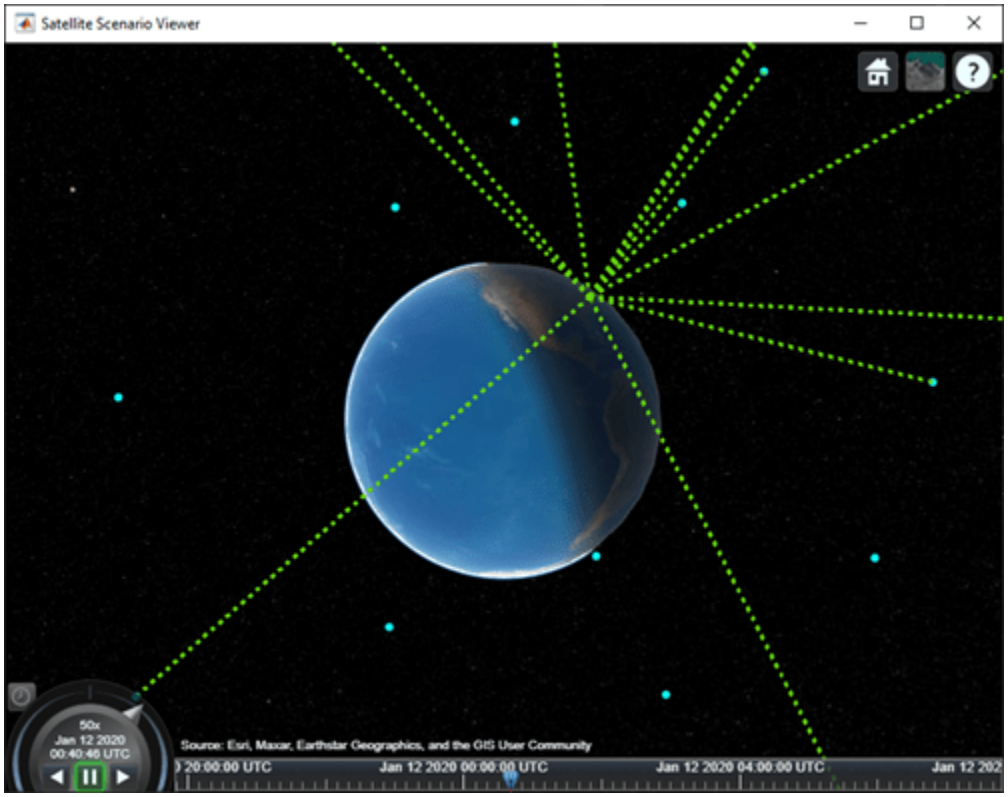
Verify that the access intervals are the same when you set the `AutoSimulate` property to `true`.

```
sc.AutoSimulate = true;
intvls2 = accessIntervals(ac)
```

```
intvls2=35×8 table
     Source          Target           IntervalNumber        StartTime              EndTime
    _______    ___________________    ______________    ___________________    ___________________

    "PRN:1"     "Ground station 32"          1           11-Jan-2020 23:20:25    12-Jan-2020 05:
    "PRN:2"     "Ground station 32"          1           12-Jan-2020 04:03:16    12-Jan-2020 07:
    "PRN:3"     "Ground station 32"          1           11-Jan-2020 19:50:06    11-Jan-2020 21:
    "PRN:3"     "Ground station 32"          2           12-Jan-2020 01:52:43    12-Jan-2020 06:
    "PRN:4"     "Ground station 32"          1           11-Jan-2020 19:50:06    12-Jan-2020 00:
    "PRN:4"     "Ground station 32"          2           12-Jan-2020 04:54:02    12-Jan-2020 07:
    "PRN:5"     "Ground station 32"          1           12-Jan-2020 05:52:03    12-Jan-2020 07:
    "PRN:6"     "Ground station 32"          1           12-Jan-2020 02:43:29    12-Jan-2020 07:
    "PRN:7"     "Ground station 32"          1           11-Jan-2020 21:09:52    12-Jan-2020 03:
    "PRN:8"     "Ground station 32"          1           11-Jan-2020 20:33:36    12-Jan-2020 03:
    "PRN:9"     "Ground station 32"          1           11-Jan-2020 19:50:06    12-Jan-2020 00:
    "PRN:9"     "Ground station 32"          2           12-Jan-2020 05:08:32    12-Jan-2020 07:
    "PRN:10"    "Ground station 32"          1           12-Jan-2020 00:32:56    12-Jan-2020 01:
    "PRN:11"    "Ground station 32"          1           11-Jan-2020 22:15:09    12-Jan-2020 04:
    "PRN:12"    "Ground station 32"          1           12-Jan-2020 04:32:16    12-Jan-2020 07:
    "PRN:13"    "Ground station 32"          1           12-Jan-2020 00:03:56    12-Jan-2020 02:
      ⋮
```

Visualize the scenario.

```
play(sc);
```



Input Arguments

`sc` — Satellite scenario

`satelliteScenario` object

Satellite scenario, specified as a `satelliteScenario` object. The argument applies only if the `AutoSimulate` property of the `sc` object is `false`.

The timeline and playback widgets on the open satellite scenario viewers that were previously made available after calling the `play` function become unavailable for interaction.

Version History

Introduced in R2022a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`play` | `satellite` | `groundStation` | `advance`

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`. The rotation used in this function is a passive transformation between two coordinate systems.

`[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.

Version History

Introduced in R2017a

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`

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 $\vec{b}$ 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.

Version History

Introduced in R2017a

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`

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`.

Aerospace Toolbox uses quaternions that are defined using the scalar-first convention.

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.

Version History

Introduced in R2017a

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`

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

two-element finite, real, and scalar numeric array | read-only

Minimum and maximum indicator scale values, specified as a two-element numeric array. This value is read-only.

`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) | 1-by-n string array | 1-by-n cell array | n-by-3 array of RGB triplets | hexadecimal color code | ...

Scale colors, specified as one of the following arrays:

- A 1-by-n string array of color options, such as `["blue" "green" "red"]`.
- An n-by-3 array of RGB triplets, such as `[0 0 1;1 1 0]`.

- A 1-by-n cell array containing RGB triplets, hexadecimal color codes, or named color options. For example, `{'#EDB120','#7E2F8E','#77AC30'}`.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- 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]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
"red"	"r"	[1 0 0]	"#FF0000"	
"green"	"g"	[0 1 0]	"#00FF00"	
"blue"	"b"	[0 0 1]	"#0000FF"	
"cyan"	"c"	[0 1 1]	"#00FFFF"	
"magenta"	"m"	[1 0 1]	"#FF00FF"	
"yellow"	"y"	[1 1 0]	"#FFFF00"	
"black"	"k"	[0 0 0]	"#000000"	
"white"	"w"	[1 1 1]	"#FFFFFF"	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	"#0072BD"	
[0.8500 0.3250 0.0980]	"#D95319"	
[0.9290 0.6940 0.1250]	"#EDB120"	
[0.4940 0.1840 0.5560]	"#7E2F8E"	
[0.4660 0.6740 0.1880]	"#77AC30"	
[0.3010 0.7450 0.9330]	"#4DBEEE"	
[0.6350 0.0780 0.1840]	"#A2142F"	

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) | on/off logical value

Visibility of the RPM indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`. 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.

`ContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object created using the `uicontextmenu` function. Use this property to display a context menu when you right-click on a component.

`Enable` — Operational state of RPM indicator

`'on'` (default) | on/off logical value

Operational state of RPM indicator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- 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 “Callbacks in App Designer”.

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 “Callbacks in App Designer”.

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) | on/off logical value

Callback interruption, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

This 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.

MATLAB determines callback interruption behavior whenever it executes a command that processes the callback queue. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines if the interruption occurs:

- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the object that owns the interrupting callback determines if the interrupting callback is discarded or added to the callback queue.
- If the value of `Interruptible` is `'on'`, then the interruption occurs. The next time MATLAB processes the callback queue, it stops the execution of the running callback and executes the interrupting callback. After the interrupting callback completes, MATLAB then resumes executing the running callback.

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.
- If the interrupting callback is owned by a `Timer` object, then the callback executes according to schedule regardless of the `Interruptible` property value.

Note 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.

The `BusyAction` property determines callback queuing behavior only when both of these conditions are met:

- The running callback contains a command that processes the callback queue, such as `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause`.
- The value of the `Interruptible` property of the object that owns the running callback is `'off'`.

Under these conditions, the `BusyAction` property of the object that owns the interrupting callback determines how MATLAB handles the interrupting callback. 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

on/off logical value

This property is read-only.

Deletion status, returned as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

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”.

Version History

Introduced in R2018b

See Also

`uiaerorpm`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

rrdelta

Compute relative pressure ratio

Syntax

```
d = rrdelta(p0,mach,g)
```

Description

`d = rrdelta(p0,mach,g)` computes the relative pressure ratio `d` from the static pressures `p0`, Mach numbers, `mach`, and specific heat ratios, `g`.

Examples

Determine Relative Pressure Ratio for Three Pressures

Determine the relative pressure ratio for three pressures.

```
delta = rrdelta([101325 22632.0672 4328.1393], 0.5, 1.4)
```

```
delta = 1×3

    1.1862    0.2650    0.0507
```

Determine Relative Pressure Ratio for Three Pressures and Heat Ratios

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 = 1×3

    1.1862    0.2635    0.0507
```

Determine Relative Pressure Ratio for Three Pressures at Different Conditions

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 = 1×3

    1.1862    0.4161    0.3342
```

Input Arguments

`p0` — Static pressure

scalar | vector

Static pressures, specified as a scalar or vector in pascals.

Example: `[1.4 1.35. 1.4]`

Data Types: `double`

`mach` — Mach numbers

scalar | vector

Mach numbers, specified as a scalar or vector.

Example: `[0.5 1 2]`

Data Types: `double`

`g` — Specific heat ratios

scalar | vector

Specific heat ratios, specified as a scalar or vector.

Example: `[0.5 1 2]`

Data Types: `double`

Output Arguments

`d` — Relative pressure ratio

scalar | vector

Relative pressure ratio, returned as a scalar or vector.

Limitations

For cases in which total pressure ratio is desired, that is, the Mach number is nonzero, the function assumes perfect gas — with constant molecular weight, constant pressure specific heat, and constant specific heat ratio — and dry air when calculating total pressures.

Version History

Introduced in R2006b

References

[1] Pratt & Whitney Aircraft. *Aeronautical Vestpocket Handbook*. United Technologies, August 1986.

See Also

`rrsigma` | `rrtheta`

rrsigma

Compute relative density ratio

Syntax

```
s = rrsigma(rho,mach,g)
```

Description

`s = rrsigma(rho,mach,g)` computes the relative density ratio `s`, from the static densities `rho`, Mach numbers, `mach`, and specific heat ratios, `g`.

Examples

Determine Relative Density Ratio for Given Densities

Determine the relative density ratio for three densities.

```
sigma = rrsigma([1.225 0.3639 0.0953], 0.5, 1.4)
```

```
sigma = 1×3

    1.1297    0.3356    0.0879
```

Determine Relative Density Ratio Given Densities and Heat Ratios

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 = 1×3

    1.1297    0.3357    0.0879
```

Determine Relative Density Ratio for Densities at Different Conditions

This example shows how to 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 = 1×3

    1.1297    0.4709    0.3382
```

Input Arguments

`rho` — Static densities

scalar | vector

M static densities, specified as a scalar or vector in kilograms per meter cubed.

Example: `[1.225 0.3639 0.0953]`

Data Types: `double`

`mach` — Mach numbers

scalar | vector

M mach numbers, specified as a scalar or vector.

Example: `[0.5 1 2]`

Data Types: `double`

`g` — Specific heat ratios

scalar | vector

M specific heat ratios, specified as a scalar or vector.

Example: `[1.4 1.35 1.4]`

Data Types: `double`

Output Arguments

`s` — Density relative ratios

scalar | vector

M density relative ratios, returned as a scalar or vector.

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.

Version History

Introduced in R2006b

References

[1] Pratt & Whitney Aircraft. *Aeronautical Vestpocket Handbook*. United Technologies, August 1986.

See Also

`rrdelta` | `rrtheta`

rrtheta

Compute relative temperature ratio

Syntax

```
th = rrtheta(t0,mach,g)
```

Description

`th = rrtheta(t0,mach,g)` computes temperature relative ratios, `th`, from static temperatures `t0`, Mach numbers `mach`, and specific heat ratios `g`. `t0` must be in kelvin.

Examples

Determine Relative Temperature Ratio for Given Temperatures

Determine the relative temperature ratio for three temperatures.

```
th = rrtheta([273.15 310.9278 373.15], 0.5, 1.4)
```

```
th = 1×3

    0.9953    1.1330    1.3597
```

Determine Relative Temperature Ratio Given Temperatures at Different Conditions

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 = 1×3

    0.9953    1.2679    2.3310
```

Determine Relative Temperature Ratio Given Temperatures and Different Heat Ratios

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 = 1×3

    0.9953    1.1263    1.3597
```

Input Arguments

`t0` — Static temperatures

scalar | vector

Static temperatures, specified as a scalar or vector in kelvin.

Example: `[1.4 1.35. 1.4]`

Data Types: `double`

`mach` — Mach numbers

scalar | vector

Mach numbers, specified as a scalar or vector.

Example: `[0.5 1 2]`

Data Types: `double`

`g` — Specific heat ratios

scalar | vector

Specific heat ratios, specified as a scalar or vector.

Example: `[0.5 1 2]`

Data Types: `double`

Output Arguments

`th` — Relative temperature ratios

scalar | vector

Relative temperature ratios, returned as a scalar or vector.

Limitations

For cases in which total temperature ratio is desired, that is, the Mach number is nonzero, the total temperature is calculated assuming perfect gas (with constant molecular weight — with constant pressure specific heat, and constant specific heat ratio — and dry air.

Version History

Introduced in R2006b

References

[1] Pratt & Whitney Aircraft. *Aeronautical Vestpocket Handbook*. United Technologies, August 1986.

See Also

`rrdelta` | `rrsigma`

Satellite

Satellite in satellite scenario

Description

Satellite defines a satellite in satellite scenario object.

Creation

You can create Satellite objects using the `satellite` function of `satelliteScenario` object.

Properties

`Name` — Satellite name

string scalar | string vector | character vector | cell array of character vectors

You can set this property only when calling the `satellite` function. After you call `satellite` function, this property is read-only.

Satellite name, specified as a comma-separated pair consisting of `'Name'` and a string scalar, string vector, character vector or a cell array of character vectors.

- If only one satellite is added, specify `Name` as a string scalar or a character vector.
- If multiple satellites are added, specify `Name` as a string scalar, character vector, string vector or a cell array of character vectors. All satellites added as a string scalar or a character vector are assigned the same specified name. The number of elements in the string vector or cell array of character vector must equal the number of satellites being added. Each satellite is assigned the corresponding name from the vector or cell array.

The default value when satellite is added to the satellite scenario using

- Keplerian orbital elements, TLE file, timeseries, or timetable — "Satellite ID", where `ID` is assigned by the satellite scenario.
- SEM almanac file or RINEX GPS navigation data — "PRN:prnValue", where prnValue is an integer denoting the pseudorandom noise code of the satellite as specified in the SEM almanac file.
- RINEX Galileo navigation data — "GAL Sat IF: id", where "id" is the satellite ID of the Galileo satellite defined in the RINEX navigation data.

Data Types: `string`

`ID` — Satellite ID assigned by simulator

real positive scalar

This property is set internally by the simulator and is read-only.

Satellite ID assigned by the simulator, specified as a positive scalar.

`ConicalSensors` — Conical sensors
row vector of conical sensors

You can set this property only when calling the `conicalSensor`. After you call the `conicalSensor` function, this property is read-only.

Conical sensors attached to the Satellite, specified as a row vector of conical sensors.

`Gimbals` — Gimbals
row vector of `Gimbal` objects

You can set this property only when calling `gimbal`. After you call `gimbal`, this property is read-only.

Gimbals attached to the Satellite, specified as the comma-separated pair consisting of `'Gimbals'` and a row vector of `Gimbal` objects.

`Accesses` — Access analysis objects
row vector of `Access` objects

You can set this property only when calling `Satellite`. After you call `Satellite`, this property is read-only.

Access analysis objects, specified as a row vector of `Access` objects.

`CoordinateAxes` — Coordinate axes triad graphic object
`CoordinateAxes` object (default)

You can set this property only when calling `coordinateAxes`. After you call `coordinateAxes`, this property is read-only.

Coordinate axes triad graphic object, specified as `CoordinateAxes` object.

`Orbit` — Orbit graphic
`Orbit` object

Orbit object parameters for a satellite, specified as an `orbit` object. Only these object properties are relevant for this function.

`LineColor` — Color of orbit
`[1,0,0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'`

Color of the orbit, specified as an RGB triplet, hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	
`"none"`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineWidth` — Visual width of orbit

1 (default) | scalar in the range (0, 10)

Visual width of orbit in pixels, specified as a scalar in the range (0, 10).

The line width cannot be thinner than the width of a pixel. If you set the line width to a value that is less than the width of a pixel on your system, the line displays as one pixel wide.

`VisibilityMode` — Visibility mode of orbit graphic

`'inherit'` (default) | `'manual'`

Visibility mode of orbit graphic, specified as one of these values:

- `'inherit'` — Visibility of the graphic matches that of the parent

- `'manual'` — Visibility of the graphic is not inherited and is independent of that of the parent

Data Types: `char` | `string`

`OrbitPropagator` — Name of orbit propagator
`"sgp4"` | `"sdp4"` | `"two-body-keplerian"` | `"ephemeris"` | `"gps"` | `"galileo"`

You can set this property on `satellite` object creation and then this property becomes read-only.

Name of the orbit propagator used for propagating the satellite position and velocity, specified as one of these options.

- If you specify the satellite using timetable, table, `timeseries`, or `tscollection`, the `OrbitPropagator` value is `"ephemeris"`.
- If you specify the satellite using a SEM almanac file or RINEX data containing a GPS navigation message, the `OrbitPropagator` value can take one of these options.
  - `"gps"` (default)
  - `"sgp4"`
  - `"sdp4"`
  - `"two-body-keplerian"`
- If you specify the satellite using the RINEX data containing a Galileo navigation message, the `OrbitPropagator` value can take one of these options.
  - `"galileo"` (default)
  - `"sgp4"`
  - `"sdp4"`
  - `"two-body-keplerian"`
- If you specify the satellite using a TLE file, the `OrbitPropagator` value can take one of these options.
  - `"two-body-keplerian"`
  - `"sgp4"`
  - `"sdp4"`

  If the orbital period is less than 225 minutes, the default `OrbitPropagator` value is `"sgp4"`. Otherwise, the default `OrbitPropagator` value is `"sdp4"`.
- If you specify the satellite using `Keplerian` elements, the `OrbitPropagator` value can take one of these options.
  - `"two-body-keplerian"`
  - `"sgp4"`
  - `"sdp4"`

If the RINEX data contains both valid GPS and Galileo navigation messages, you cannot specify `OrbitPropagator` as `"gps"` or `"galileo"` using a name-value argument. However, you can still specify it as `"two-body-keplerian"`, `"sgp4"`, or `"sdp4"`.

`MarkerColor` — Color of marker
`[1 0 0]` (default) | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Color of the marker, specified as a comma-separated pair consisting of `'MarkerColor'` and either an RGB triplet or a string or character vector of a color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	
`"none"`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	

`MarkerSize` — Size of marker

`10` (default) | positive scalar less than 30

Size of the marker, specified as a comma-separated pair consisting of `'MarkerSize'` and a real positive scalar less than 30. The unit is in pixels.

ShowLabel — State of Satellite label visibility

`true` or `1` (default) | `false` or `0`

State of Satellite label visibility, specified as a comma-separated pair consisting of `'ShowLabel'` and numerical or logical value of `1` (`true`) or `0` (`false`).

Data Types: `logical`

LabelFontColor — Font color of Satellite label

`[1,0,0]` (default) | `RGB triplet` | `string scalar of color name` | `character vector of color name`

Font color of the Satellitelabel, specified as a comma-separated pair consisting of `'LabelFontColor'` and either an RGB triplet or a string or character vector of a color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- 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]`.
- A hexadecimal color code is a string scalar or character vector that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	
`"none"`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4940 0.1840 0.5560]	"#7E2F8E"	
[0.4660 0.6740 0.1880]	"#77AC30"	
[0.3010 0.7450 0.9330]	"#4DBEEE"	
[0.6350 0.0780 0.1840]	"#A2142F"	

`LabelFontSize` — Font size of Satellite label
15 (default) | positive scalar in the range [6 30]

Font size of the Satellite label, specified as a comma-separated pair consisting of `'LabelFontSize'` and a positive scalar in the range [6 30].

Object Functions

access	Add access analysis objects to satellite scenario
aer	Calculate azimuth angle, elevation angle, and range of another satellite or ground station in NED frame
conicalSensor	Add conical sensor to satellite scenario
gimbal	Add gimbal to satellite or ground station
groundTrack	Add ground track object to satellite in scenario
orbitalElements	Orbital elements of satellites in scenario
coordinateAxes	Visualize coordinate axes triad of satellite scenario assets
pointAt	Point satellite at target
states	Obtain position and velocity of satellite
show	Show object in satellite scenario viewer
hide	Hide satellite scenario entity from viewer

Examples

Visualize Line of Sight Between Two Satellites

Create a satellite scenario object.

```
startTime = datetime(2020,5,5,0,0,0);
stopTime = startTime + days(1);
sampleTime = 60;                                          %seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add a satellite from a TLE file to the scenario.

```
tleFile = "eccentricOrbitSatellite.tle";
sat1 = satellite(sc,tleFile,"Name","Sat1")
```

```
sat1 =
  Satellite with properties:

               Name:  Sat1
                 ID:  1
    ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
           Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
      Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
         Receivers:  [1x0 satcom.satellitescenario.Receiver]
```

```
          Accesses:  [1x0 matlabshared.satellitescenario.Access]
       GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
             Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
   OrbitPropagator:  sdp4
       MarkerColor:  [0.059 1 1]
        MarkerSize:  6
         ShowLabel:  true
    LabelFontColor:  [1 1 1]
     LabelFontSize:  15
```

Add a satellite from Keplerian elements to the scenario and specify its orbit propagator to be "two-body-keplerian".

```
semiMajorAxis = 6878137;                                                                    %m
eccentricity = 0;
inclination = 20;                                                                           %degr
rightAscensionOfAscendingNode = 0;                                                          %degr
argumentOfPeriapsis = 0;                                                                    %degr
trueAnomaly = 0;                                                                            %degr
sat2 = satellite(sc,semiMajorAxis,eccentricity,inclination,rightAscensionOfAscendingNode,...
    argumentOfPeriapsis,trueAnomaly,"OrbitPropagator","two-body-keplerian","Name","Sat2")
```

```
sat2 =
  Satellite with properties:

              Name:  Sat2
                ID:  2
    ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
           Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
      Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
         Receivers:  [1x0 satcom.satellitescenario.Receiver]
          Accesses:  [1x0 matlabshared.satellitescenario.Access]
       GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
             Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
   OrbitPropagator:  two-body-keplerian
       MarkerColor:  [0.059 1 1]
        MarkerSize:  6
         ShowLabel:  true
    LabelFontColor:  [1 1 1]
     LabelFontSize:  15
```

Add access analysis between the two satellites.

```
ac = access(sat1,sat2);
```

Determine the times when there is line of sight between the two satellites.

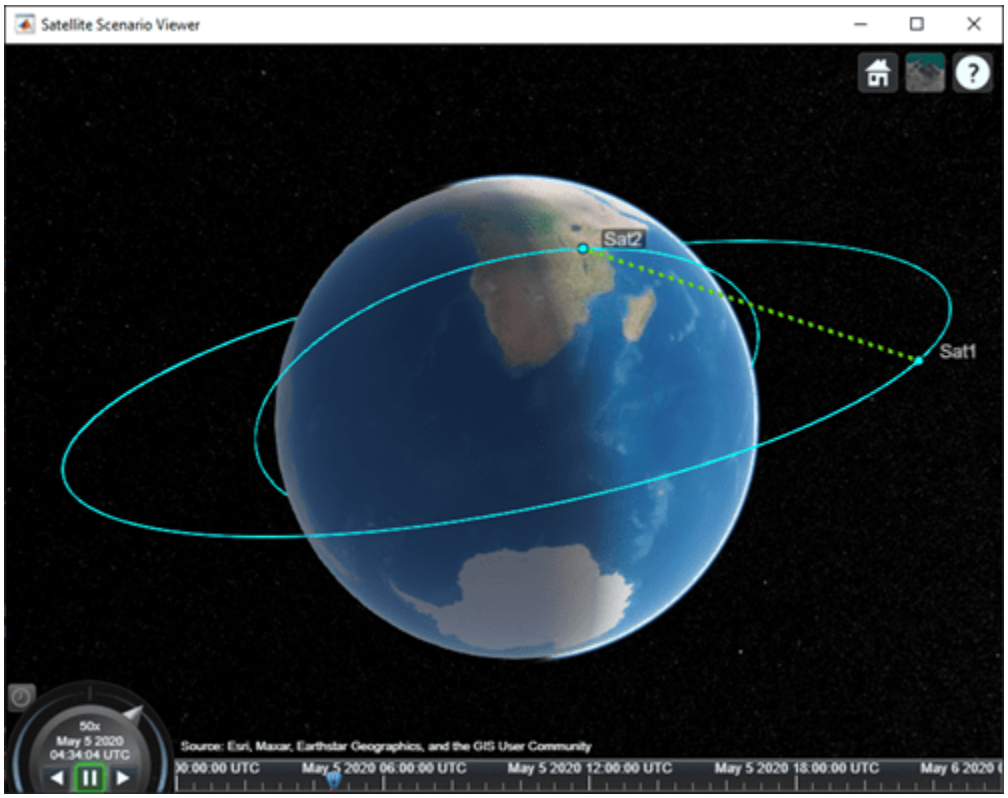
```
accessIntervals(ac)
```

```
ans=15×8 table
    Source    Target    IntervalNumber         StartTime                EndTime            Duratio
    ______    ______    ______________    ____________________    ____________________    _______

    "Sat1"    "Sat2"           1          05-May-2020 00:09:00    05-May-2020 01:08:00      3540
    "Sat1"    "Sat2"           2          05-May-2020 01:50:00    05-May-2020 02:47:00      3420
    "Sat1"    "Sat2"           3          05-May-2020 03:45:00    05-May-2020 04:05:00      1200
    "Sat1"    "Sat2"           4          05-May-2020 04:32:00    05-May-2020 05:26:00      3240
```

```
    "Sat1"    "Sat2"          5          05-May-2020 06:13:00    05-May-2020 07:10:00      3420
    "Sat1"    "Sat2"          6          05-May-2020 07:52:00    05-May-2020 08:50:00      3480
    "Sat1"    "Sat2"          7          05-May-2020 09:30:00    05-May-2020 10:29:00      3540
    "Sat1"    "Sat2"          8          05-May-2020 11:09:00    05-May-2020 12:07:00      3480
    "Sat1"    "Sat2"          9          05-May-2020 12:48:00    05-May-2020 13:46:00      3480
    "Sat1"    "Sat2"         10          05-May-2020 14:31:00    05-May-2020 15:27:00      3360
    "Sat1"    "Sat2"         11          05-May-2020 17:12:00    05-May-2020 18:08:00      3360
    "Sat1"    "Sat2"         12          05-May-2020 18:52:00    05-May-2020 19:49:00      3420
    "Sat1"    "Sat2"         13          05-May-2020 20:30:00    05-May-2020 21:29:00      3540
    "Sat1"    "Sat2"         14          05-May-2020 22:08:00    05-May-2020 23:07:00      3540
    "Sat1"    "Sat2"         15          05-May-2020 23:47:00    06-May-2020 00:00:00       780
```

Visualize the line of sight between the satellites.

```
play(sc);
```



Visualize GPS Constellation

Set up the satellite scenario.

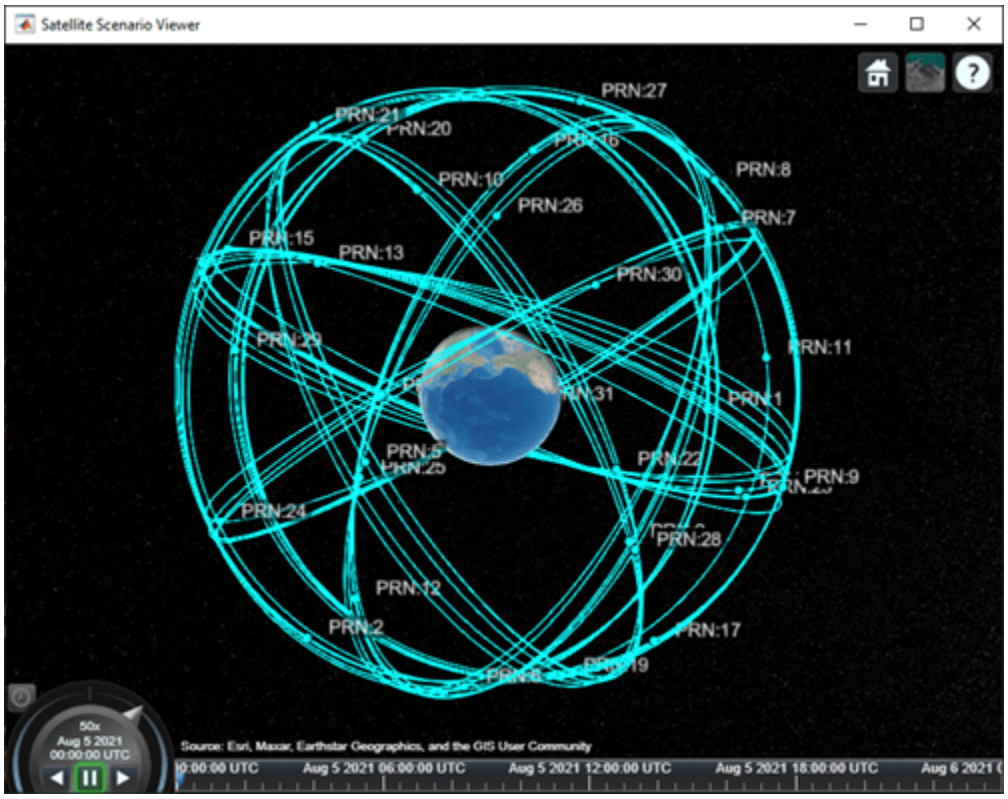
```
startTime = datetime(2021,8,5);
stopTime = startTime + days(1);
sampleTime = 60;                                        % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add satellites to the scenario from a SEM almanac file.

```
sat = satellite(sc,"gpsAlmanac.txt","OrbitPropagator","gps");
```

Visualize the GPS constellation.

```
v = satelliteScenarioViewer(sc);
```



Version History

Introduced in R2021a

References

[1] Hoots, Felix R., and Ronald L. Roehrich. *Models for propagation of NORAD element sets*. Aerospace Defense Command Peterson AFB CO Office of Astrodynamics, 1980.

See Also

Objects

`satelliteScenario` | `groundStation` | `access` | `satelliteScenarioViewer`

Functions

`show` | `play` | `hide`

Topics

“Model, Visualize, and Analyze Satellite Scenario” on page 2-77

“Satellite Scenario Key Concepts” on page 2-65

satellite

Add satellites to satellite scenario

Syntax

```
satellite(scenario,file)
satellite(scenario,RINEXdata)
satellite(scenario,semimajoraxis,eccentricity,inclination,RAAN,
argofperiapsis,trueanomaly)
satellite(scenario,positiontable)
satellite(scenario,positiontable,velocitytable)
satellite(scenario,positiontimeseries)
satellite(scenario,positiontimeseries,velocitytimeseries)
satellite( ___ ,Name,Value)
sat = satellite( ___ )
```

Description

`satellite(scenario,file)` adds `Satellite` objects from `file` to the satellite scenario specified by `scenario`. The yaw (z) axes of the satellites point toward nadir and the roll (x) axes of the satellites align with their respective inertial velocity vectors.

`satellite(scenario,RINEXdata)` adds `Satellite` objects from `RINEXdata` to the satellite scenario specified by `scenario`.

`satellite(scenario,semimajoraxis,eccentricity,inclination,RAAN,
argofperiapsis,trueanomaly)` adds a `Satellite` object from Keplerian elements defined in the Geocentric Celestial Reference Frame (GCRF) to the satellite scenario.

`satellite(scenario,positiontable)` adds a `Satellite` object from position data specified in `positiontable` (`timetable` object) to the scenario. This function creates a `Satellite` with `OrbitPropagator="ephemeris"`.

`satellite(scenario,positiontable,velocitytable)` adds a `Satellite` object from position data specified in `positiontable` (`timetable` object) and velocity data specified in `velocitytable` (`timetable` object) to the scenario. This function creates a `Satellite` with `OrbitPropagator="ephemeris"`.

`satellite(scenario,positiontimeseries)` adds a `Satellite` object from position data specified in `positiontimeseries` to the `scenario`. This function creates a `Satellite` with `OrbitPropagator="ephemeris"`.

`satellite(scenario,positiontimeseries,velocitytimeseries)` adds a `Satellite` object to the `scenario` from position (in meters) data specified in `positiontimeseries` (`timeseries` object) and velocity (in meters/second) data specified in `velocitytimeseries` (`timeseries` object). This function creates a `Satellite` with `OrbitPropagator="ephemeris"`.

`satellite( ___ ,Name,Value)` specifies options using one or more name-value arguments in addition to any input argument combination from previous syntaxes.

`sat = satellite( ___ )` returns a vector of handles to the added satellites. Specify any input argument combination from previous syntaxes.

Note When the `AutoSimulate` property of the `satelliteScenario` is `false`, you can modify the `satellite` only when the `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Examples

Visualize Satellite Trajectories

Create a satellite scenario object.

```
sc = satelliteScenario;
```

Load the satellite ephemeris timetable in the Earth Centered Earth Fixed (ECEF) coordinate frame.

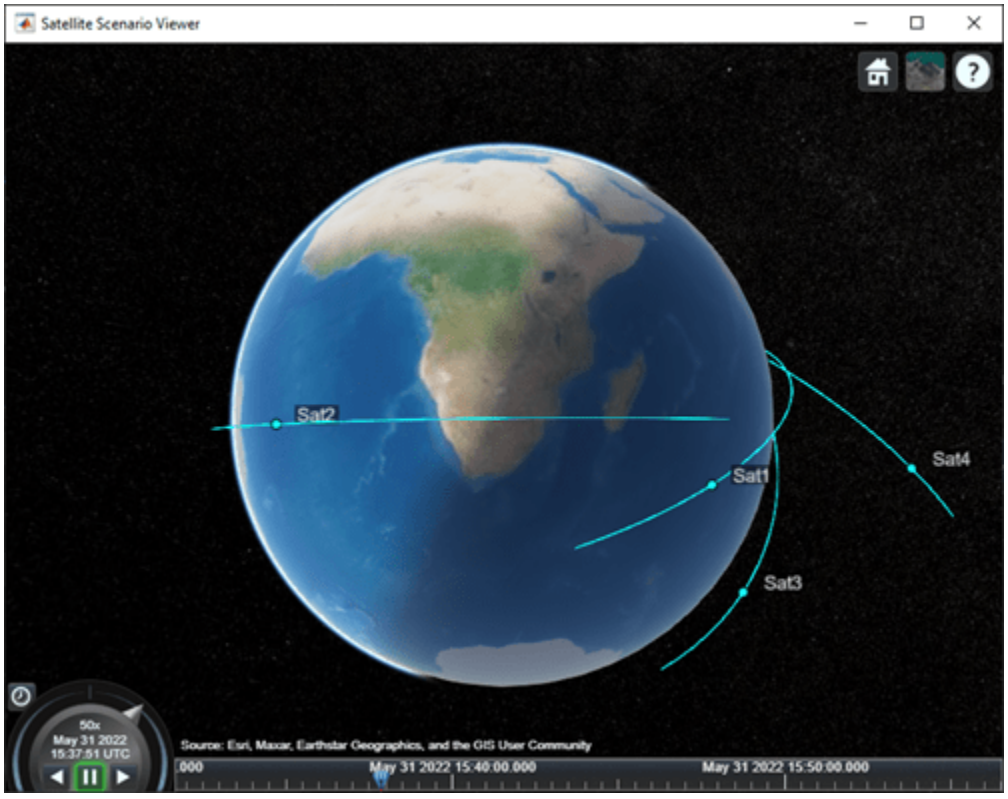
```
load("timetableSatelliteTrajectory.mat","positionTT","velocityTT");
```

Add four satellites to the scenario.

```
sat = satellite(sc,positionTT,velocityTT,"CoordinateFrame","ecef");
```

Visualize the trajectories of the satellites.

```
play(sc);
```



Add Ground Stations to Scenario and Visualize Access Intervals

Create a satellite scenario and add ground stations from latitudes and longitudes.

```
startTime = datetime(2020,5,1,11,36,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
lat = 10;
lon = -30;
gs = groundStation(sc,lat,lon);
```

Add satellites using Keplerian elements.

```
semiMajorAxis = 10000000;
eccentricity = 0;
inclination = 10;
rightAscensionOfAscendingNode = 0;
argumentOfPeriapsis = 0;
trueAnomaly = 0;
sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
        rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly);
```

Add access analysis to the scenario and obtain the table of intervals of access between the satellite and the ground station.

```
ac = access(sat,gs);
intvls = accessIntervals(ac)
```

```
intvls=8×8 table
       Source              Target          IntervalNumber         StartTime                EndTi
    _____________    ___________________    _______________    ____________________    ___________

    "Satellite 2"    "Ground station 1"          1           01-May-2020 11:36:00    01-May-2020
    "Satellite 2"    "Ground station 1"          2           01-May-2020 14:20:00    01-May-2020
    "Satellite 2"    "Ground station 1"          3           01-May-2020 17:27:00    01-May-2020
    "Satellite 2"    "Ground station 1"          4           01-May-2020 20:34:00    01-May-2020
    "Satellite 2"    "Ground station 1"          5           01-May-2020 23:41:00    02-May-2020
    "Satellite 2"    "Ground station 1"          6           02-May-2020 02:50:00    02-May-2020
    "Satellite 2"    "Ground station 1"          7           02-May-2020 05:59:00    02-May-2020
    "Satellite 2"    "Ground station 1"          8           02-May-2020 09:06:00    02-May-2020
```

Play the scenario to visualize the ground stations.

```
play(sc)
```

Add Satellites to Scenario Using Keplerian Elements

Create a satellite scenario with a start time of 02-June-2020 8:23:00 AM UTC, and the stop time set to one day later. Set the simulation sample time to 60 seconds.

```
startTime = datetime(2020,6,02,8,23,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add two satellites to the scenario using their Keplerian elements.

```
semiMajorAxis = [10000000;15000000];
eccentricity = [0.01;0.02];
inclination = [0;10];
rightAscensionOfAscendingNode = [0;15];
argumentOfPeriapsis = [0;30];
trueAnomaly = [0;20];

sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
    rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly)
```

```
sat =
  1x2 Satellite array with properties:

    Name
    ID
```

```
    ConicalSensors
    Gimbals
    Transmitters
    Receivers
    Accesses
    GroundTrack
    Orbit
    OrbitPropagator
    MarkerColor
    MarkerSize
    ShowLabel
    LabelFontColor
    LabelFontSize
```

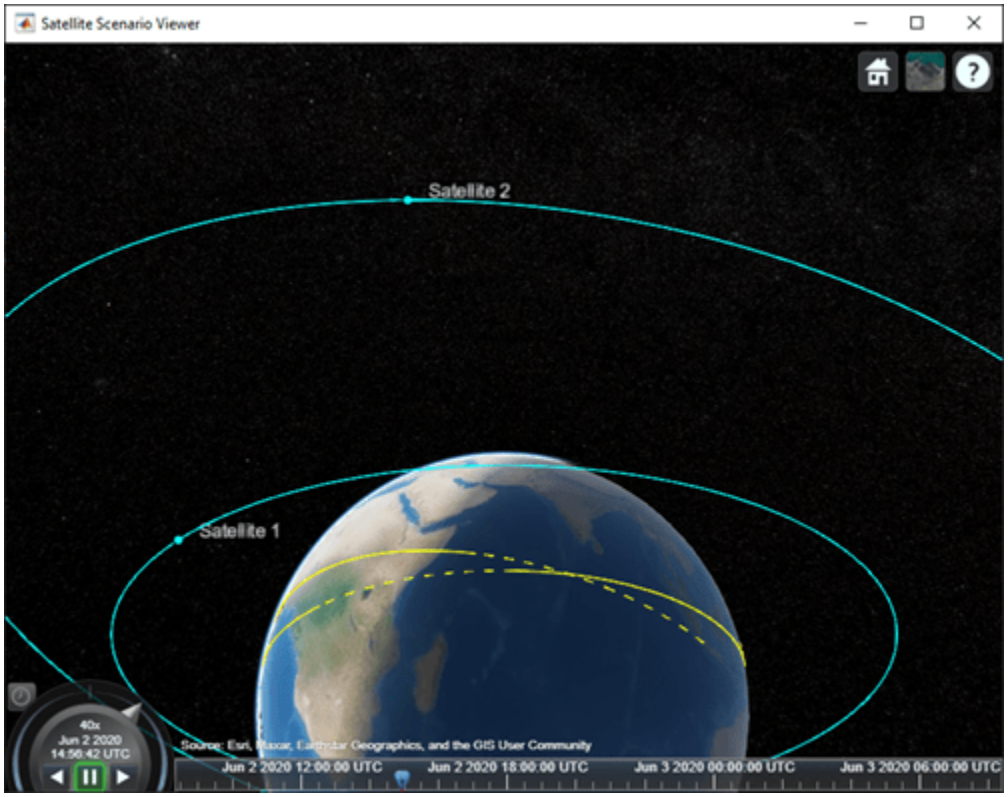
View the satellites in orbit and the ground tracks over one hour.

```
show(sat)
groundTrack(sat,LeadTime=3600)
```

```
ans=1×2 object
  1×2 GroundTrack array with properties:

    LeadTime
    TrailTime
    LineWidth
    LeadLineColor
    TrailLineColor
    VisibilityMode
```

Play the scenario and set the animation speed of the simulation to 40.

```
play(sc,PlaybackSpeedMultiplier=40)
```



Visualize GPS Constellation

Set up the satellite scenario.

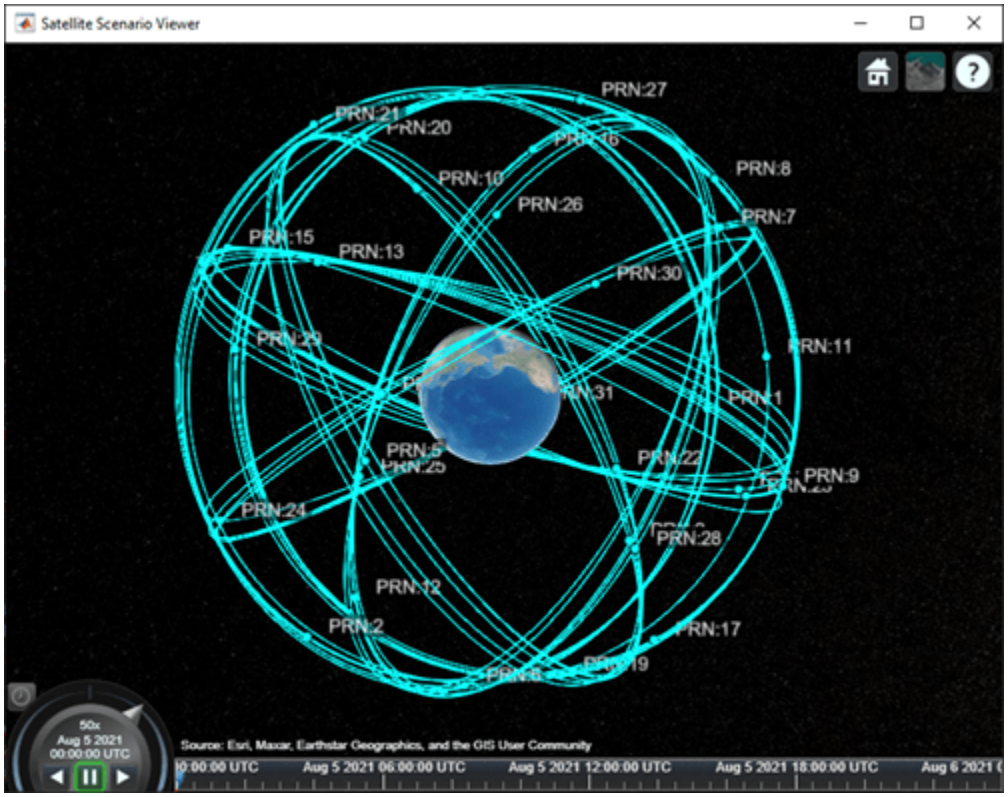
```
startTime = datetime(2021,8,5);
stopTime = startTime + days(1);
sampleTime = 60;                                          % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add satellites to the scenario from a SEM almanac file.

```
sat = satellite(sc,"gpsAlmanac.txt","OrbitPropagator","gps");
```

Visualize the GPS constellation.

```
v = satelliteScenarioViewer(sc);
```



Input Arguments

`scenario` — Satellite scenario

`satelliteScenario` object

Satellite scenario, specified as a `satelliteScenario` object.

`file` — TLE or SEM almanac file

character vector | string scalar

TLE or SEM almanac file, specified as a character vector or a string scalar. The file must exist in the current folder, in a folder on the MATLAB path, or it must include a full or relative path to a file.

For more information on TLE files, see “Two Line Element (TLE) Files” on page 2-72.

Data Types: `char` | `string`

`RINEXdata` — RINEX navigation message

structure

RINEX navigation message output from `rinexread`, specified as a structure. The navigation message must belong to GPS or Galileo constellation.

If `RINEXdata` contains multiple entries for the same satellite and `StartTime` of `scenario` is using default values, the entry corresponding to the earliest time is used. If `StartTime` is explicitly specified, the entry that is closest to `StartTime` is used.

Data Types: `struct`

Note This argument requires Navigation Toolbox™.

`semimajoraxis`, `eccentricity`, `inclination`, `RAAN`, `argofperiapsis`, `trueanomaly` — Keplerian elements defined in GCRF

comma-separated list of vectors

Keplerian elements defined in the GCRF, specified as a comma-separated list of vectors. The Keplerian elements are:

- `semimajoraxis` – This vector defines the semimajor axis of the orbit of the satellite. Each value is equal to half of the longest diameter of the orbit.
- `eccentricity` – This vector defines the shape of the orbit of the satellite.
- `inclination` – This vector defines the angle between the orbital plane and the *xy*-plane of the GCRF for each satellite in the range [0,180].
- `RAAN` (right ascension of ascending node) – This element defines the angle between the *xy*-plane of the GCRF and the direction of the ascending node, as seen from the Earth's center of mass for each satellite in the range [0,360). The ascending node is the location where the orbit crosses the *xy*-plane of the GCRF and goes above the plane.
- `argofperiapsis` (argument of periapsis) – This vector defines the angle between the direction of the ascending node and the periapsis, as seen from the Earth's center of mass in the range [0,360). Periapsis is the location on the orbit that is closest to the Earth's center of mass for each satellite.
- `trueanomaly` – This vector defines the angle between the direction of the periapsis and the current location of the satellite, as seen from the Earth's center of mass for each satellite in the range [0,360).

Note All angles defined outside the specified range is automatically converted to the corresponding value within the acceptable range.

For more information on Keplerian elements, see “Orbital Elements” on page 2-70.

`positiontable` — Position data

timetable | table

Position data in meters, specified as a timetable created using the `timetable` function or `table` function. The `positiontable` has exactly one monotonically increasing column of *rowTimes* (`datetime` or `duration`) and either:

- One or more columns of variables, where each column contains data for an individual satellite over time.
- One column of 2-D data, where the length of one dimension must equal 3 and the remaining dimension defines the number of satellites in the ephemeris.
- One column of 3-D data, where the length of one dimension must equal 3, one dimension is a singleton, and the remaining dimension defines the number of satellites in the ephemeris.

If *rowTimes* values are of type `duration`, time values are measured relative to the current scenario `StartTime` property. The timetable `VariableNames` property are used by default if no names are

provided as an input. Satellite states are assumed to be in the GCRF unless a `CoordinateFrame` name-value argument is provided. States are held constant in GCRF for scenario timesteps outside of the time range of `positiontable`.

Data Types: `table` | `timetable`

`velocitytable` — Velocity data
timetable | table

Velocity data in meters/second, specified as a timetable created using the `timetable` function or the `table` function. The `velocitytable` has exactly one monotonically increasing column of *rowTimes* (`datetime` or `duration`), and either:

- One or more columns of variables, where each column contains data for an individual satellite over time.
- One column of 2-D data, where the length of one dimension must equal 3 and the remaining dimension defines the number of satellites in the ephemeris.
- One column of 3-D data, where the length of one dimension must equal 3, one dimension is a singleton, and the remaining dimension defines the number of satellites in the ephemeris.

If *rowTimes* values are of type `duration`, time values are measured relative to the current scenario `StartTime` property. The timetable `VariableNames` are used by default if no names are provided as an input. Satellite states are assumed to be in the GCRF unless a `CoordinateFrame` name-value argument is provided. States are held constant in GCRF for scenario timesteps outside of the time range of `velocitytable`.

Data Types: `table` | `timetable`

`positiontimeseries` — Position data
`timeseries` object | `tscollection` object

Position data in meters, specified as a `timeseries` object or a `tscollection` object.

- If the `Data` property of the `timeseries` or `tscollection` object has two dimensions, one dimension must equal 3, and the other dimension must align with the orientation of the time vector.
- If the `Data` property of the `timeseries` or `tscollection` has three dimensions, one dimension must equal 3, either the first or the last dimension must align with the orientation of the time vector, and the remaining dimension defines the number of satellites in the ephemeris.

  When `timeseries.TimeInfo.StartDate` is empty, time values are measured relative to the current scenario `StartTime` property. The timeseries `Name` property (if defined) is used by default if no names are provided as inputs. Satellite states are assumed to be in the GCRF unless a `CoordinateFrame` name-value pair is provided. States are held constant in GCRF for scenario timesteps outside of the time range of `positiontimeseries`.

Data Types: `timeseries` | `tscollection`

`velocitytimeseries` — Velocity data
`timeseries` object | `tscollection` object

Velocity data in meters/second, specified as a `timeseries` object or a `tscollection` object.

- If the `Data` property of the `timeseries` or `tscollection` object has two dimensions, one dimension must equal 3, and the other dimension must align with the orientation of the time vector.
- If the `Data` property of the `timeseries` or `tscollection` has three dimensions, one dimension must equal 3, either the first or the last dimension must align with the orientation of the time vector, and the remaining dimension defines the number of satellites in the ephemeris.

  When `timeseries.TimeInfo.StartDate` is empty, time values are measured relative to the current scenario `StartTime` property. The timeseries `Name` property (if defined) is used by default if no names are provided as inputs. Satellite states are assumed to be in the GCRF unless a `CoordinateFrame` name-value pair is provided. States are held constant in GCRF for scenario timesteps outside of the time range of `velocitytimeseries`.

Data Types: `timeseries` | `tscollection`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Name = 'MySatellite'` sets the satellite name to `'MySatellite'`.

`CoordinateFrame` — Satellite state coordinate frame

`"inertial"` (default) | `"ecef"` | `"geographic"`

Satellite state coordinate frame, specified as the comma-separated pair consisting of `'CoordinateFrame'` and one of these values:

- `"inertial"` — For `timeseries` or `timetable` data, specifying this value accepts the position and velocity in the GCRF frame.
- `"ecef"` — For `timeseries` or `timetable` data, specifying this value accepts the position and velocity in the ECEF frame.
- `"geographic"` — For `timeseries` or `timetable` data, specifying this value accepts the position [*lat*, *lon*, *altitude*], where *lat* and *lon* are latitude and longitude in degrees, and *altitude* is the height above the World Geodetic System 84 (WGS 84) ellipsoid in meters.

  Velocity is in the local NED frame.

Dependencies

To enable this name value argument, ephemeris data inputs (`timetable` or `timeseries`).

Data Types: `string` | `char`

`GPSweekepoch` — GPS week epoch

date string

GPS week epoch, specified as a date string in "dd-Mmm-yyyy" or 'dd-Mmm-yyyy' format. The GPS week number specifies the reference date that the function uses when counting weeks defined in the SEM almanac file. If you do not specify `GPSweekepoch`, the function uses the date that coincides with the latest GPS week number rollover date before the start time.

This argument applies only if you use a SEM almanac file. If you specify `GPSweekepoch` and you are not using a SEM almanac file, the function ignores the argument value.

Data Types: `string` | `char`

`Viewer` — Satellite scenario viewer
vector of `satelliteScenarioViewer` objects (default) | scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Satellite scenario viewer, specified as a scalar, vector, or array of `satelliteScenarioViewer` objects. If the `AutoSimulate` property of the scenario is `false`, adding a satellite to the scenario disables any previously available timeline and playback widgets.

`Name` — Satellite name
string scalar | string vector | character vector | cell array of character vectors

You can set this property only when calling the `satellite` function. After you call `satellite` function, this property is read-only.

Satellite name, specified as a comma-separated pair consisting of `'Name'` and a string scalar, string vector, character vector or a cell array of character vectors.

- If only one satellite is added, specify `Name` as a string scalar or a character vector.
- If multiple satellites are added, specify `Name` as a string scalar, character vector, string vector or a cell array of character vectors. All satellites added as a string scalar or a character vector are assigned the same specified name. The number of elements in the string vector or cell array of character vector must equal the number of satellites being added. Each satellite is assigned the corresponding name from the vector or cell array.

The default value when satellite is added to the satellite scenario using

- Keplerian orbital elements, TLE file, timeseries, or timetable — "Satellite ID", where `ID` is assigned by the satellite scenario.
- SEM almanac file or RINEX GPS navigation data — "PRN:prnValue", where prnValue is an integer denoting the pseudorandom noise code of the satellite as specified in the SEM almanac file.
- RINEX Galileo navigation data — "GAL Sat IF: id", where "id" is the satellite ID of the Galileo satellite defined in the RINEX navigation data.

Data Types: `string`

`OrbitPropagator` — Name of orbit propagator
`"sgp4"` | `"sdp4"` | `"two-body-keplerian"` | `"ephemeris"` | `"gps"` | `"galileo"`

You can set this property on `satellite` object creation and then this property becomes read-only.

Name of the orbit propagator used for propagating the satellite position and velocity, specified as one of these options.

- If you specify the satellite using timetable, table, `timeseries`, or `tscollection`, the `OrbitPropagator` value is `"ephemeris"`.
- If you specify the satellite using a SEM almanac file or RINEX data containing a GPS navigation message, the `OrbitPropagator` value can take one of these options.
  - `"gps"` (default)
  - `"sgp4"`
  - `"sdp4"`

  - `"two-body-keplerian"`
- If you specify the satellite using the RINEX data containing a Galileo navigation message, the `OrbitPropagator` value can take one of these options.
  - `"galileo"` (default)
  - `"sgp4"`
  - `"sdp4"`
  - `"two-body-keplerian"`
- If you specify the satellite using a TLE file, the `OrbitPropagator` value can take one of these options.
  - `"two-body-keplerian"`
  - `"sgp4"`
  - `"sdp4"`

  If the orbital period is less than 225 minutes, the default `OrbitPropagator` value is `"sgp4"`. Otherwise, the default `OrbitPropagator` value is `"sdp4"`.
- If you specify the satellite using `Keplerian` elements, the `OrbitPropagator` value can take one of these options.
  - `"two-body-keplerian"`
  - `"sgp4"`
  - `"sdp4"`

If the RINEX data contains both valid GPS and Galileo navigation messages, you cannot specify `OrbitPropagator` as `"gps"` or `"galileo"` using a name-value argument. However, you can still specify it as `"two-body-keplerian"`, `"sgp4"`, or `"sdp4"`.

Output Arguments

`sat` — Satellite in the scenario

`Satellite` object

Satellite in the scenario, returned as a `Satellite` object belonging to the satellite scenario specified by `scenario`.

You can modify the `Satellite` object by changing its property values.

Version History

Introduced in R2021a

R2022b: RINEXdata argument added to the function

You can now add satellites via RINEX navigation data to the scenario using the `RINEXdata` input argument.

R2022a: SEM almanac file added to the file argument

You can now add satellites via SEM almanac file to the scenario using the `file` input argument.

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`access` | `show` | `play` | `hide` | `orbitalElements`

Topics
“Satellite Scenario Key Concepts” on page 2-65

satelliteScenario

Satellite scenario

Description

The `satelliteScenario` object represents a 3D arena consisting of satellites, ground stations, and the interactions between them. Use this object to model satellite constellations, model ground station networks, perform access analyses between the satellites and the ground stations, and visualize the results.

Creation

Syntax

```
sc = satelliteScenario
sc = satelliteScenario(startTime,stopTime,sampleTime)
sc = satelliteScenario( ___ ,AutoSimulate=false)
```

Description

`sc = satelliteScenario` creates a default satellite scenario object.

`sc = satelliteScenario(startTime,stopTime,sampleTime)` sets the `StartTime`, `StopTime`, and `SampleTime` properties to the values of `startTime`, `stopTime`, and `sampleTime`, respectively.

`sc = satelliteScenario( ___ ,AutoSimulate=false)` sets the `AutoSimulate` property to a specified value.

Properties

`StartTime` — Start time of satellite scenario simulation in UTC

`datetime` scalar

Start time of the satellite scenario simulation in UTC, specified as a `datetime` scalar.

The default `StartTime` is the current UTC time if no satellites are present in the scenario. Otherwise, it is the earliest value among

- the current UTC time.
- the epoch defined in the TLE files.
- the reference time deduced from the GPS week number defined in SEM files.
- the earliest time deduced from RINEX navigation data for GPS and Galileo.
- the initial time in the timetable and timeseries.

If the `StartTime`, `StopTime`, or `SampleTime` properties are explicitly specified, the `StartTime` property no longer updates with further additions of satellites.

When the `AutoSimulate` property is `false`, you can modify the `StartTime` property only when the `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Data Types: `datetime`

`StopTime` — Stop time of satellite scenario simulation in UTC
`datetime` scalar

Stop time of the satellite scenario simulation in UTC, specified as a `datetime` scalar. The default `StopTime` is `StartTime` + longest orbital period among the satellites in the scenario. If no satellites are added to the scenario, the default `StopTime` is the same as the default `StartTime`. If the `StartTime`, `StopTime`, or `SampleTime` properties are explicitly specified, the `StopTime` property no longer updates with further additions of satellites.

When the `AutoSimulate` property is `false`, you can modify the `StopTime` property only when the `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Data Types: `datetime`

`SampleTime` — Sample time of satellite scenario simulation
scalar

Sample time of the satellite scenario simulation, specified as a real-valued scalar. The default sample time is set such that there are 100 samples between `StartTime` and `StopTime`. If the default `StartTime` and `StopTime` are the same, which is the case when no satellites are added to the scenario, the default `SampleTime` is 60 seconds. If the `StartTime`, `StopTime`, or `SampleTime` properties are explicitly specified, the `SampleTime` property no longer updates with further additions of satellites.

When the `AutoSimulate` property is `false`, you can modify the `SampleTime` property only when the `SimulationStatus` is `NotStarted`. You can use the `restart` function to reset `SimulationStatus` to `NotStarted`, but doing so erases the simulation data.

Data Types: `double`

`SimulationTime` — Simulation time of satellite scenario in UTC
current UTC time (default) | `datetime` scalar

This property is read-only.

Current simulation time of the satellite scenario simulation in UTC, specified as a `datetime` scalar. Call the `restart` function to reset the time to `StartTime`.

Dependencies

To enable this property, set `AutoSimulate` to `false`.

Data Types: `datetime`

`SimulationStatus` — Simulation status
`'NotStarted'` (default) | `'InProgress'` | `'Completed'`

This property is read-only.

Simulation status of the satellite scenario, specified as one of the following:

- `'NotStarted'` — No call to the `advance` function has been made
- `'InProgress'` — Simulation is running
- `'Completed'` — Simulation is finished

The simulation starts when the first call to the `advance` function is made. The simulation continues until one of the following occurs:

- The simulation reaches the `StopTime`.
- A new asset is added to the satellite scenario.
- Certain properties of the asset (satellites, ground stations, gimbals, conical sensors, and so on) have been modified, such as `MountingLocation` or `MountingAngles`. Refer to the properties to determine if modifying them can stop the simulation.

Call the `restart` function to restart the simulation, erase the simulation data, and set `SimulationStatus` to `NotStarted`.

Dependencies

To enable this property, set `AutoSimulate` to `false`.

`AutoSimulate` — Option to simulate satellite scenario automatically

`true` or `1` (default) | `false` or `0`

Option to simulate the satellite scenario automatically, specified as one of these numeric or logical values.

- `1` (`true`) — Simulate the satellite scenario automatically on any call to an analysis function, such as `states` or `accessIntervals`.
- `0` (`false`)— Simulate the satellite scenario only by calling the `advance` function.

Changing the `AutoSimulate` value erases the previous simulation data.

Data Types: `double`

`Satellites` — Satellites in the scenario

row vector of `Satellite` objects

Satellites in the scenario, returned as a row vector of `Satellite` objects. To create a `Satellite` object and add it to the satellite scenario, use the `satellite` object function. After adding satellites to the scenario, this property is read-only.

`GroundStations` — Ground stations in scenario

row vector of `GroundStation` objects

Ground stations in the scenario, returned as a row vector of `GroundStation` objects. To create a `GroundStation` object and add it to the satellite scenario, use the `groundStation` object function. After adding ground stations to the scenario, this property is read-only.

`AutoShow` — Option to automatically show graphics

`true` or `1` (default) | `false` or `0`

Option to automatically show graphics, specified as a logical `1` (`true`) or `0` (`false`). This property determines if entities added to the scenario are automatically shown in an open `satelliteScenarioViewer` window.

Object Functions

groundStation	Add ground station to satellite scenario
satellite	Add satellites to satellite scenario
satelliteScenarioViewer	Create viewer for satellite scenario
advance	Move simulation forward by one sample time
restart	Restart simulation from beginning
play	Play satellite scenario simulation results on viewer
walkerDelta	Create Walker-Delta constellation in satellite scenario
walkerStar	Create Walker-Star constellation in satellite scenario

Examples

Create Satellite Scenario with Custom Start and Stop Times

Specify the start time in the current time zone as yesterday. The simulation lasts for half a day.

```
startTime = datetime("yesterday","TimeZone","local");
stopTime = startTime + days(0.5);
```

Specify the sample time as 60 seconds. Create a satellite scenario object, specifying the start time, stop time, and sample time.

```
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc =
  satelliteScenario with properties:

         StartTime: 02-Mar-2023 05:00:00
          StopTime: 02-Mar-2023 17:00:00
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1x0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1x0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0x0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add Satellites to Scenario Using Keplerian Elements

Create a satellite scenario with a start time of 02-June-2020 8:23:00 AM UTC, and the stop time set to one day later. Set the simulation sample time to 60 seconds.

```
startTime = datetime(2020,6,02,8,23,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add two satellites to the scenario using their Keplerian elements.

```
semiMajorAxis = [10000000;15000000];
eccentricity = [0.01;0.02];
```

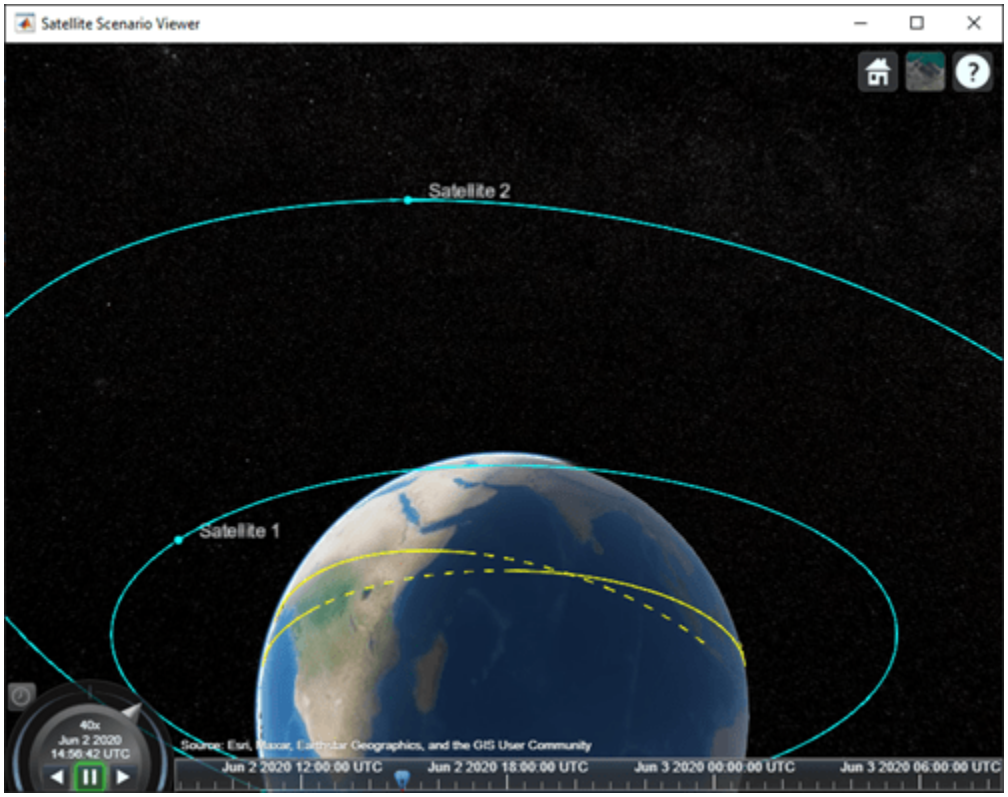
```
inclination = [0;10];
rightAscensionOfAscendingNode = [0;15];
argumentOfPeriapsis = [0;30];
trueAnomaly = [0;20];

sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
    rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly)

sat =
  1x2 Satellite array with properties:

    Name
    ID
    ConicalSensors
    Gimbals
    Transmitters
    Receivers
    Accesses
    GroundTrack
    Orbit
    OrbitPropagator
    MarkerColor
    MarkerSize
    ShowLabel
    LabelFontColor
    LabelFontSize
```

View the satellites in orbit and the ground tracks over one hour.

```
show(sat)
groundTrack(sat,LeadTime=3600)

ans=1×2 object
  1×2 GroundTrack array with properties:

    LeadTime
    TrailTime
    LineWidth
    LeadLineColor
    TrailLineColor
    VisibilityMode
```

Play the scenario and set the animation speed of the simulation to 40.

```
play(sc,PlaybackSpeedMultiplier=40)
```



Manual Simulation of Satellite Scenario

Create a satellite scenario object and set the `AutoSimulate` property to `false` to enable manual simulation of the satellite scenario.

```
startTime = datetime(2022,1,12);
stopTime = startTime + days(0.5);
sampleTime = 60;                                % Seconds
sc = satelliteScenario('AutoSimulate',false);
```

Add a GPS satellite constellation to the scenario.

```
sat = satellite(sc,"gpsAlmanac.txt");
```

Simulate the scenario using the `advance` function.

```
while advance(sc)
end
```

Obtain the satellite position histories.

```
p = states(sat);
```

`AutoSimulate` is `false`, so restart the scenario before adding a ground station.

```
restart(sc);
```

Add a ground station to the scenario.

```
gs = groundStation(sc);
```

Add access analysis between each satellite and ground station.

```
ac = access(sat,gs);
```

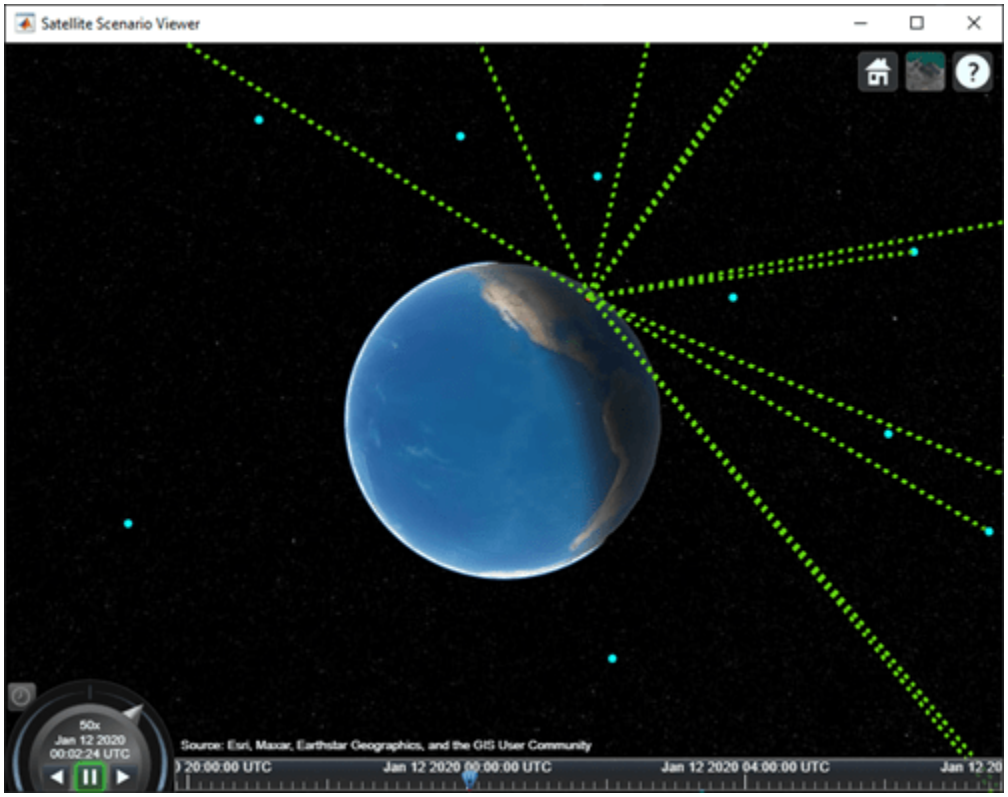
Simulate the scenario and determine the access intervals.

```
while advance(sc)
end
intvls1 = accessIntervals(ac)
```

```
intvls1=35×8 table
     Source           Target           IntervalNumber         StartTime              EndTime
    ________    ____________________    ______________    ____________________    ____________

    "PRN:1"     "Ground station 32"           1          11-Jan-2020 23:20:25    12-Jan-2020 05:1
    "PRN:2"     "Ground station 32"           1          12-Jan-2020 04:03:16    12-Jan-2020 07:4
    "PRN:3"     "Ground station 32"           1          11-Jan-2020 19:50:06    11-Jan-2020 21:3
    "PRN:3"     "Ground station 32"           2          12-Jan-2020 01:52:43    12-Jan-2020 06:4
    "PRN:4"     "Ground station 32"           1          11-Jan-2020 19:50:06    12-Jan-2020 00:1
    "PRN:4"     "Ground station 32"           2          12-Jan-2020 04:54:02    12-Jan-2020 07:4
    "PRN:5"     "Ground station 32"           1          12-Jan-2020 05:52:03    12-Jan-2020 07:4
    "PRN:6"     "Ground station 32"           1          12-Jan-2020 02:43:29    12-Jan-2020 07:4
    "PRN:7"     "Ground station 32"           1          11-Jan-2020 21:09:52    12-Jan-2020 03:2
    "PRN:8"     "Ground station 32"           1          11-Jan-2020 20:33:36    12-Jan-2020 03:1
    "PRN:9"     "Ground station 32"           1          11-Jan-2020 19:50:06    12-Jan-2020 00:4
    "PRN:9"     "Ground station 32"           2          12-Jan-2020 05:08:32    12-Jan-2020 07:4
    "PRN:10"    "Ground station 32"           1          12-Jan-2020 00:32:56    12-Jan-2020 01:5
    "PRN:11"    "Ground station 32"           1          11-Jan-2020 22:15:09    12-Jan-2020 04:1
    "PRN:12"    "Ground station 32"           1          12-Jan-2020 04:32:16    12-Jan-2020 07:4
    "PRN:13"    "Ground station 32"           1          12-Jan-2020 00:03:56    12-Jan-2020 02:5
      ⋮
```

Visualize the simulation results.

```
v = satelliteScenarioViewer(sc,'ShowDetails',false);
play(sc);
```



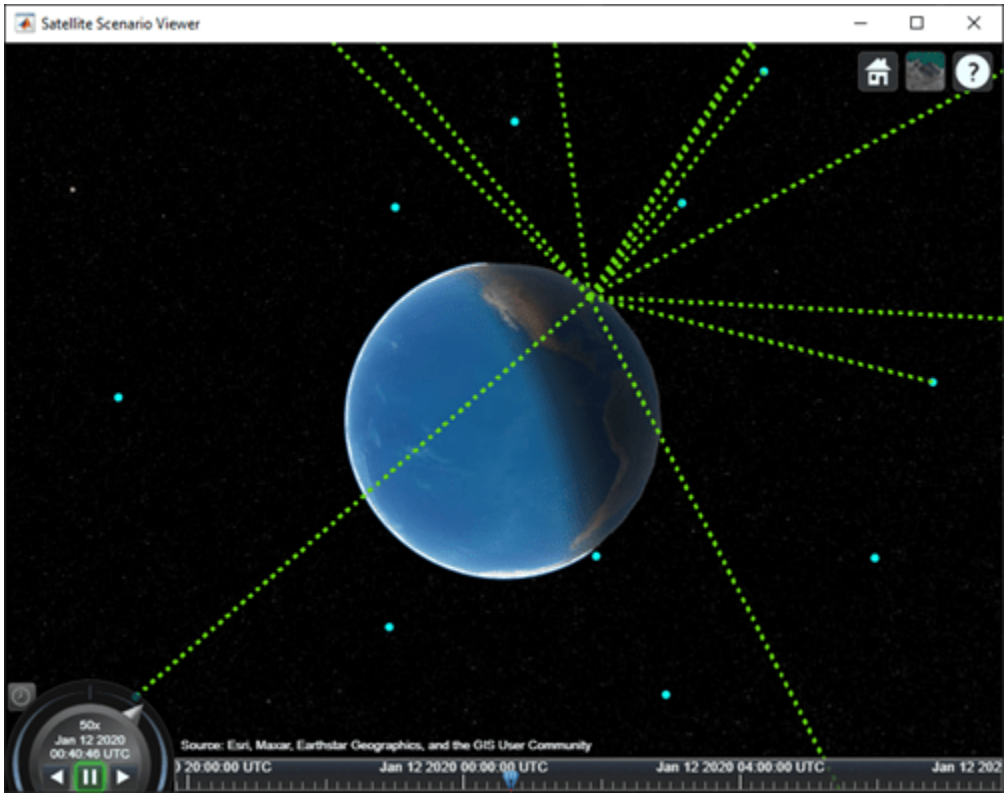
Verify that the access intervals are the same when you set the `AutoSimulate` property to `true`.

```
sc.AutoSimulate = true;
intvls2 = accessIntervals(ac)
```

```
intvls2=35×8 table
     Source           Target           IntervalNumber        StartTime              EndTime
    ________    ____________________    ______________    ____________________    ____________

    "PRN:1"     "Ground station 32"          1           11-Jan-2020 23:20:25    12-Jan-2020 05:1
    "PRN:2"     "Ground station 32"          1           12-Jan-2020 04:03:16    12-Jan-2020 07:4
    "PRN:3"     "Ground station 32"          1           11-Jan-2020 19:50:06    11-Jan-2020 21:3
    "PRN:3"     "Ground station 32"          2           12-Jan-2020 01:52:43    12-Jan-2020 06:4
    "PRN:4"     "Ground station 32"          1           11-Jan-2020 19:50:06    12-Jan-2020 00:1
    "PRN:4"     "Ground station 32"          2           12-Jan-2020 04:54:02    12-Jan-2020 07:4
    "PRN:5"     "Ground station 32"          1           12-Jan-2020 05:52:03    12-Jan-2020 07:4
    "PRN:6"     "Ground station 32"          1           12-Jan-2020 02:43:29    12-Jan-2020 07:4
    "PRN:7"     "Ground station 32"          1           11-Jan-2020 21:09:52    12-Jan-2020 03:2
    "PRN:8"     "Ground station 32"          1           11-Jan-2020 20:33:36    12-Jan-2020 03:1
    "PRN:9"     "Ground station 32"          1           11-Jan-2020 19:50:06    12-Jan-2020 00:4
    "PRN:9"     "Ground station 32"          2           12-Jan-2020 05:08:32    12-Jan-2020 07:4
    "PRN:10"    "Ground station 32"          1           12-Jan-2020 00:32:56    12-Jan-2020 01:5
    "PRN:11"    "Ground station 32"          1           11-Jan-2020 22:15:09    12-Jan-2020 04:1
    "PRN:12"    "Ground station 32"          1           12-Jan-2020 04:32:16    12-Jan-2020 07:4
    "PRN:13"    "Ground station 32"          1           12-Jan-2020 00:03:56    12-Jan-2020 02:5
      ⋮
```

Visualize the scenario.

```
play(sc);
```



Tips

- When saving the satellite scenario, either save the entire workspace containing the scenario object or save the scenario object itself.

Version History

Introduced in R2021a

See Also

Objects

`satellite` | `satelliteScenarioViewer`

Functions

`play` | `show` | `hide` | `advance` | `restart` | `access` | `groundStation`

Topics

“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

satelliteScenarioViewer

Create viewer for satellite scenario

Description

The `satelliteScenarioViewer` object represents a 3D visualization of the `satelliteScenario` object. Use this object to focus the camera onto the satellite or the ground station by double-clicking the satellite or ground station. Once focused, you can rotate the camera and it will orbit around the selected satellite/ground station and not be allowed to pan out. To unfocus, you must double-click elsewhere in the viewer or press the home button.

Creation

Syntax

```
satelliteScenarioViewer(scenario)
v = satelliteScenarioViewer(scenario)
satelliteScenarioViewer(scenario,Name=Value)
```

Description

`satelliteScenarioViewer(scenario)` creates a 3-D or 2-D satellite scenario viewer for the specified satellite scenario. Satellite Scenario Viewer is a 3-D map display and requires hardware graphics support for WebGL™.

`v = satelliteScenarioViewer(scenario)` returns the handle to the satellite scenario viewer.

`satelliteScenarioViewer(scenario,Name=Value)` sets properties on page 4-870 using one or more optional name-value arguments. For example, `satelliteScenarioViewer(scenario,Position = [1 1 500 500])` sets the position to `[1 1 500 500]`.

Properties

`Name` — Name of viewer window

`'Satellite Scenario Viewer'` (default) | string scalar | character vector

Name of the viewer window, specified as a comma-separated pair consisting of `'Name'` and either a string scalar or a character vector.

Data Types: `char` | `string`

`Position` — Viewer window position

center of the screen (default) | row vector of four elements

Size and location of the satellite scenario window in pixels, specified as a row vector of four elements. The elements of the vector are `[left bottom width height]`. In the default case, width and height are 800 and 600 pixels, respectively.

Basemap — Map on which scenario is visualized

`'satellite'` (default) | `'topographic'` | `'streets'` | `'streets-light'` | `'streets-dark'` | `'darkwater'` | `'grayland'` | `'bluegreen'` | `'colorterrain'` | `'grayterrain'` | `'landcover'`

Map on which scenario is visualized, specified as a comma-separated pair consisting of `'Basemap'` and one of the values specified in this table:

All basemaps except `'darkwater'` require Internet access. The `'darkwater'` basemap is included with MATLAB and Aerospace Toolbox.

If you do not have consistent access to the Internet, you can download the basemaps created using Natural Earth onto your local system by using the Add-On Explorer. The basemaps hosted by Esri are not available for download.

Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by The MathWorks®.

Data Types: `char` | `string`

`PlaybackSpeedMultiplier` — Speed of animation
`50` (default) | positive scalar

Speed of the animation for the input `scenario` used by the `play` function, specified as a comma-separated pair consisting of `'PlaybackSpeedMultiplier'` and a positive scalar.

`CameraReferenceFrame` — Reference frame of camera
`'ECEF'` (default) | `'Inertial'`

Reference frame of the camera, specified as a comma-separated pair consisting of `'CameraReferenceFrame'` and one of these values:

- `'ECEF'` — Earth-Centered Earth-Fixed camera.
- `'Inertial'` — Inertially fixed camera.

When you specify `'Inertial'`, the globe rotates with respect to the camera. When you specify `'ECEF'`, the camera rotates with the globe.

Dependencies

To enable this name-value argument, set to `Dimension` to `'3-D'`.

`CurrentTime` — Current simulation time
`StartTime` of `satelliteScenario` (default) | `datetime` array

Current simulation time of the viewer, specified as a `datetime` array. This value changes over time when the animation is playing.

Dependencies

To enable this name-value argument, set `AutoSimulate` to `true`.

Data Types: `datetime`

`Dimension` — Dimension of viewer
`'3D'` (default) | `'2D'`

Dimension of the viewer, specified as a comma-separated pair consisting of `'Dimension'` and either `'3D'` or `'2D'`.

`ShowDetails` — Flag to show graphical details
`true` or `1` (default) | `false` or `0`

Flag to show the graphical details for Satellite Scenario Viewer, specified as one of these numeric or logical values1.

- `1` (`true`) — Show all graphical details of satellites and ground stations except those explicitly hidden.
- `0` (`false`) — Hide all graphical details of satellites and the ground stations, including orbits, fields of view, labels, and the ground track. Even when `ShowDetails` is `false`, clicking or pausing on satellites and ground stations reveals hidden graphical details or labels, respectively.

Data Types: `logical`

Object Functions

campos	Set or get position of camera for Satellite Scenario Viewer
camheight	Set or get height of camera for Satellite Scenario Viewer
camheading	Set or get heading angle of camera for Satellite Scenario Viewer
camroll	Set or get roll angle of camera for Satellite Scenario Viewer
campitch	Set or get pitch angle of camera for Satellite Scenario Viewer
camtarget	Set camera target for Satellite Scenario Viewer
play	Play satellite scenario simulation results on viewer
showAll	Show all graphics in viewer
hideAll	Hide all graphics in satellite scenario viewer

Examples

Create and Visualize Satellite Scenario

Create a satellite scenario object.

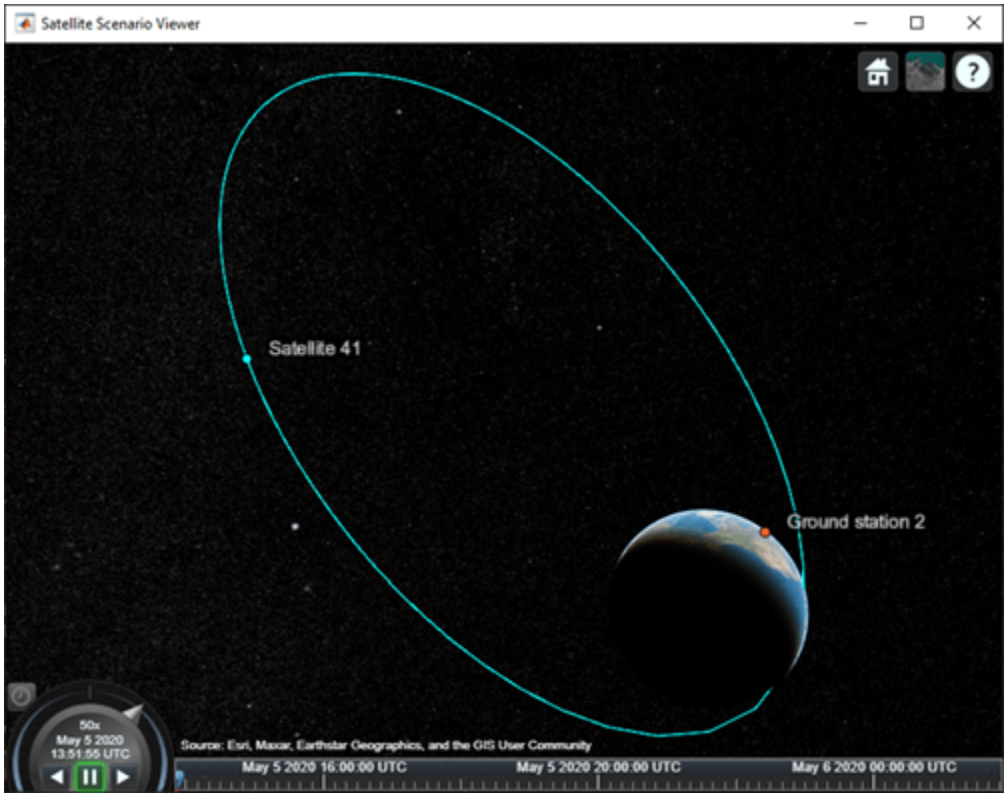
```
sc = satelliteScenario;
```

Add a satellite and ground station to the scenario. Additionally, add an access between the satellite and the ground station.

```
sat = satellite(sc,"eccentricOrbitSatellite.tle");
gs = groundStation(sc);
access(sat,gs);
```

Visualize the scenario at the start time defined in the TLE file by using the Satellite Scenario Viewer.

```
satelliteScenarioViewer(sc);
```



Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario`

Functions

`show` | `hide` | `access` | `groundStation` | `satellite`

Topics

“Satellite Scenario Key Concepts” on page 2-65

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.

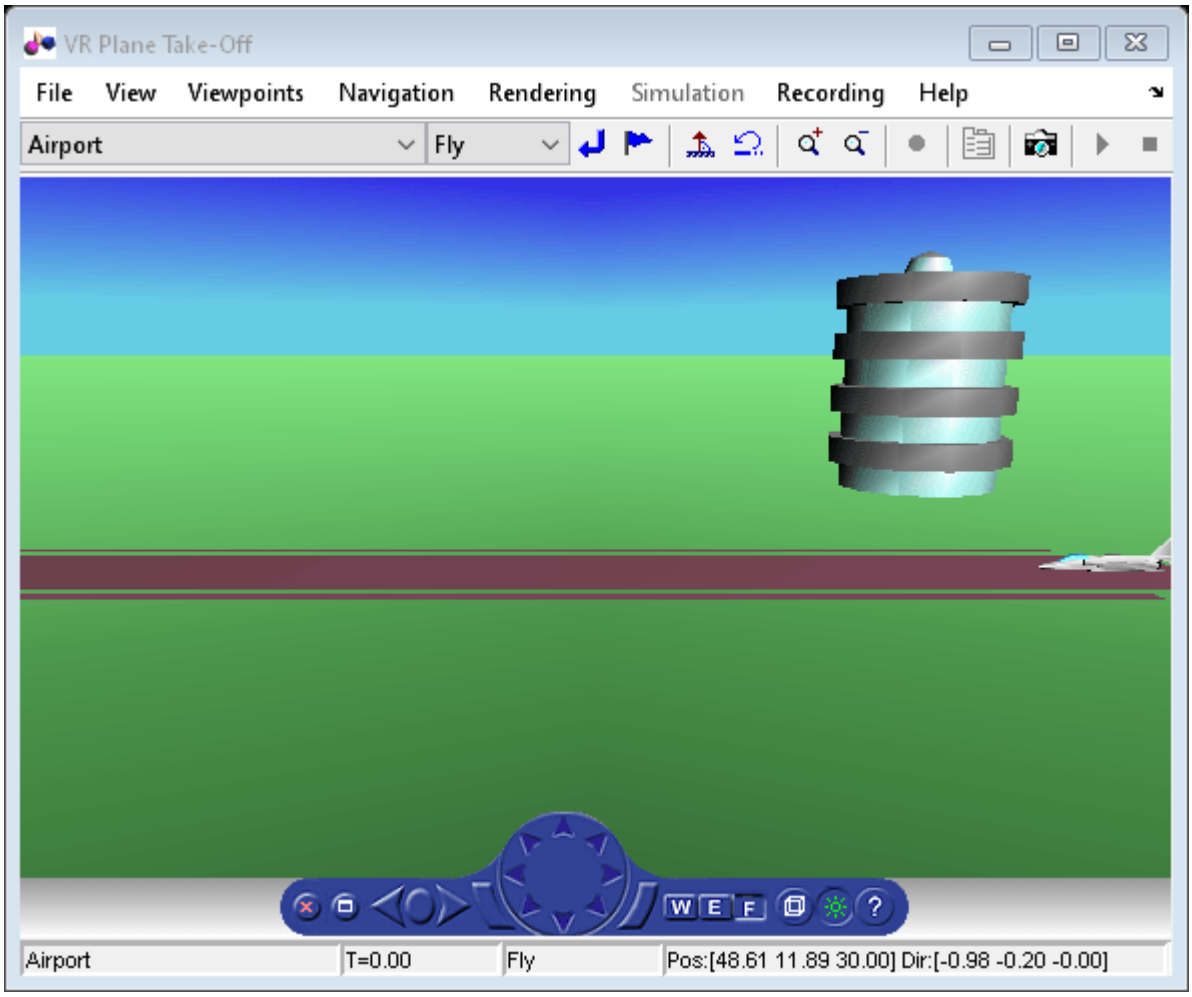
`saveas(h, filename, '-nothumbnail')` and `h.saveas(filename, '-nothumbnail')` suppress creating a thumbnail image used for virtual world preview.

Examples

Save Virtual World

This example shows how to save the world associated with `h`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
h.saveas([tempdir,'my_asttkoff.wrl']);
```



Version History

Introduced in R2007b

setCoefficient

Class: `Aero.FixedWing`
Package: `Aero`

Set coefficient value for `Aero.FixedWing` object

Syntax

```
aircraft = setCoefficient(aircraft,stateOutput,stateVariable,value)
aircraft = setCoefficient(__,Name,Value)
```

Description

`aircraft = setCoefficient(aircraft,stateOutput,stateVariable,value)` sets the coefficient value `value` to the coefficient specified by `stateOutput` and `stateVariable` and returns the modified object `aircraft`.

`aircraft = setCoefficient(__,Name,Value)` sets the coefficient value using one or more `Name,Value` pair arguments.

Input Arguments

`aircraft` — `Aero.FixedWing` object

scalar

`Aero.FixedWing` object, specified as a scalar.

`stateOutput` — Valid state output

vector of strings | character array

Valid state output, specified in a vector of strings or character array. For more information, see `Aero.FixedWing.Coefficient`.

Data Types: `string` | `char`

`stateVariable` — Valid state output

vector of strings | character array

Valid state variable, specified in a vector of strings or character array. Valid state variables depend on the coefficients defined on the object. For more information, see `Aero.FixedWing.State`.

Data Types: `string` | `char`

`value` — `Simulink.LookupTable` object or numeric constant

vector of cells

`Simulink.LookupTable` object or numeric constant, specified as a vector of cells where each cell is a `Simulink.LookupTable` object or numeric constant. For more information on coefficient values, see `Aero.FixedWing.Coefficient`.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'AddVariable','on'`

`Component` — Component name
string

Component name, specified as a string. Valid component names depend on the object properties and all subcomponents on the object. The default component name is the current object.

Data Types: `char` | `string`

`AddVariable` — Option to add state variable
`off` (default) | `on`

Option to add state variable if desired state variable is missing, specified as:

- `'on'` — Add a state variable.
- `'off'` — Do not add a state variable.

Data Types: `logical`

Output Arguments

`aircraft` — Modified `Aero.FixedWing` object
scalar

Modified `Aero.FixedWing` object with the modified coefficients at the specified locations, returned as a scalar.

Examples

Set Coefficient for `Aero.FixedWing` Object

Set a coefficient on a `Aero.FixedWing` object.

```
C182 = astC182();
C182 = setCoefficient(C182, "CD", "Alpha", {5})
```

```
C182 =

  FixedWing with properties:

       ReferenceArea: 174
       ReferenceSpan: 36
     ReferenceLength: 4.9000
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
```

```
            Surfaces: [1×3 Aero.FixedWing.Surface]
             Thrusts: [1×1 Aero.FixedWing.Thrust]
         AspectRatio: 7.4483
          UnitSystem: "English (ft/s)"
         AngleSystem: "Radians"
   TemperatureSystem: "Fahrenheit"
          Properties: [1×1 Aero.Aircraft.Properties]
```

Set Vector of Coefficients for `FixedWing.Control` Object

Set a vector of coefficients on a `FixedWing.Control` object.

```
C182 = astC182();
C182 = setCoefficient(C182, ["CY"; "Cm"], ["Zero"; "Alpha"], {5; Simulink.LookupTable})
```

```
C182 =

  FixedWing with properties:

       ReferenceArea: 174
       ReferenceSpan: 36
     ReferenceLength: 4.9000
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×3 Aero.FixedWing.Surface]
             Thrusts: [1×1 Aero.FixedWing.Thrust]
         AspectRatio: 7.4483
          UnitSystem: "English (ft/s)"
         AngleSystem: "Radians"
   TemperatureSystem: "Fahrenheit"
          Properties: [1×1 Aero.Aircraft.Properties]
```

Set Coefficient on Component in Aero.FixedWing Object

Set a coefficient on a component within an Aero.FixedWing object.

```
C182 = astC182();
C182 = setCoefficient(C182, "CD", "Elevator", {5}, "Component", "Elevator")
```

```
C182 =

  FixedWing with properties:

       ReferenceArea: 174
       ReferenceSpan: 36
     ReferenceLength: 4.9000
        Coefficients: [1×1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1×3 Aero.FixedWing.Surface]
             Thrusts: [1×1 Aero.FixedWing.Thrust]
         AspectRatio: 7.4483
          UnitSystem: "English (ft/s)"
         AngleSystem: "Radians"
```

```
    TemperatureSystem: "Fahrenheit"
           Properties: [1×1 Aero.Aircraft.Properties]
```

Limitations

When used with `Simulink.LookupTable` objects, this method requires a Simulink license.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `getCoefficient` | `Simulink.LookupTable`

setCoefficient

Class: `Aero.FixedWing.Coefficient`
Package: `Aero`

Set coefficient values for fixed-wing coefficient object

Syntax

```
fixedWingCoefficient = setCoefficient(fixedWingCoefficient,stateOutput,
stateVariable,value)
fixedWingCoefficient = setCoefficient( ___ ,Name,Value)
```

Description

`fixedWingCoefficient = setCoefficient(fixedWingCoefficient,stateOutput, stateVariable,value)` sets the coefficient value `value` to the coefficient specified by `stateOutput` and `stateVariable` and returns the modified `Aero.FixedWing.Coefficient` object.

`fixedWingCoefficient = setCoefficient( ___ ,Name,Value)` sets the coefficient value `value` to the coefficient specified by `stateOutput` and `stateVariable` and returns the modified object.

Input Arguments

`fixedWingCoefficient` — `Aero.FixedWing.Coefficient` object on which to set coefficient

scalar

`Aero.FixedWing.Coefficient` on which to set coefficient, specified as a scalar.

`stateOutput` — State output

6-by-1 vector

State output, specified as a 6-by-1 vector where each entry is a valid state output. For more information on state outputs, see `Aero.FixedWing.Coefficient`.

Data Types: `char` | `string`

`stateVariable` — State variable

vector

State variable, specified as a vector where each entry is a valid state variable. Valid state variables depend on the coefficients defined on the object. For more information on fixed-wing states, see `Aero.FixedWing.State`.

Data Types: `char` | `string`

`value` — State values

vector of cells

State values, specified as a vector of cells where each cell is a numeric constant or a `Simulink.LookupTable` object. For more information on coefficient values, see `Aero.FixedWing.Coefficient`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'AddVariable','on'`

`Component` — Component name
string

Component name, specified as a string. Valid component names depend on the object properties and all subcomponents on the object. The default component name is the current object.

Data Types: `char` | `string`

`AddVariable` — Option to add state variable
`off` (default) | `on`

Add state variable if desired state variable is missing, specified as:

- `'on'` — Add a state variable.
- `'off'` — Do not add a state variable.

Data Types: `logical`

Output Arguments

`fixedWingCoefficient` — Modified fixed-wing coefficient object
`Aero.FixedWing.Coefficient`

Modified fixed-wing coefficient object on which coefficient is set, returned as `Aero.FixedWing.Coefficient`.

Examples

Set Coefficient for `Aero.FixedWing` Object

Set a coefficient on an `Aero.FixedWing` object.

```
C182 = astC182();
C182 = setCoefficient(C182, "CD", "Alpha", {5})
```

```
C182 =

  FixedWing with properties:
```

```
        ReferenceArea: 174
        ReferenceSpan: 36
      ReferenceLength: 4.9000
         Coefficients: [1×1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1×3 Aero.FixedWing.Surface]
              Thrusts: [1×1 Aero.FixedWing.Thrust]
          AspectRatio: 7.4483
           UnitSystem: "English (ft/s)"
          AngleSystem: "Radians"
    TemperatureSystem: "Fahrenheit"
           Properties: [1×1 Aero.Aircraft.Properties]
```

Set Vector of Coefficients for `Aero.FixedWing.Control` Object

Set a vector of coefficient values on an `Aero.FixedWing.Control` object.

```
C182 = astC182();
C182 = setCoefficient(C182, ["CY"; "Cm"], ["Zero"; "Alpha"], {5; Simulink.LookupTable})
```

```
C182 =

  FixedWing with properties:

        ReferenceArea: 174
        ReferenceSpan: 36
      ReferenceLength: 4.9000
         Coefficients: [1×1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1×3 Aero.FixedWing.Surface]
              Thrusts: [1×1 Aero.FixedWing.Thrust]
          AspectRatio: 7.4483
           UnitSystem: "English (ft/s)"
          AngleSystem: "Radians"
    TemperatureSystem: "Fahrenheit"
           Properties: [1×1 Aero.Aircraft.Properties]
```

Limitations

The vectors for the `stateOutput`, `stateVariable`, and `value` arguments must be the same length.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `getCoefficient`

shortPeriodCategoryAPlot

Draw MIL-F-8785C short-period category A requirements plot

Syntax

```
shortPeriodCategoryAPlot(nalpha,omega)
shortPeriodCategoryAPlot(nalpha,omega,LineSpec)
shortPeriodCategoryAPlot(nalpha1,omega1,LineSpec1,...nalphan,omegan,
LineSpecn)

shortPeriodCategoryAPlot(___,Name,Value)
shortPeriodCategoryAPlot(ax,___)

[line,bline] = shortPeriodCategoryAPlot(___)
```

Description

Basic Syntax and Line Specification

`shortPeriodCategoryAPlot(nalpha,omega)` plots vector `omega` versus vector `nalpha`. If `nalpha` or `omega` is a matrix, then the function plots the vector versus the rows or columns of the matrix, whichever are aligned. If `nalpha` is a scalar and `omega` is a vector, the function creates the disconnected line objects and plots them as discrete points vertically at `nalpha`. This function is based on the MATLAB `plot` function.

`shortPeriodCategoryAPlot(nalpha,omega,LineSpec)` plots short-period category A requirements specified by the line specification `LineSpec`.

`shortPeriodCategoryAPlot(nalpha1,omega1,LineSpec1,...nalphan,omegan, LineSpecn)` combines the plots specified by the `nalpha`, `omega`, and `linespec`. It sets the line style, marker type, and color for each line. You can mix `nalpha`, `omega`, `LineSpec` triplets with `nalpha`, `omega` arguments, for example, `plot(nalpha1,omega1,nalpha2,omega2,LineSpec2,nalpha3,omega3)`.

Name-Value Arguments and Axes Specification

`shortPeriodCategoryAPlot(___,Name,Value)` plots an altitude envelope contour specified by one or more `Name,Value` arguments. Specify name-value arguments after all other input arguments.

`shortPeriodCategoryAPlot(ax,___)` draws an altitude contour plot onto the axes `ax`. Specify arguments as previously listed after the `ax` argument.

Return Line Objects

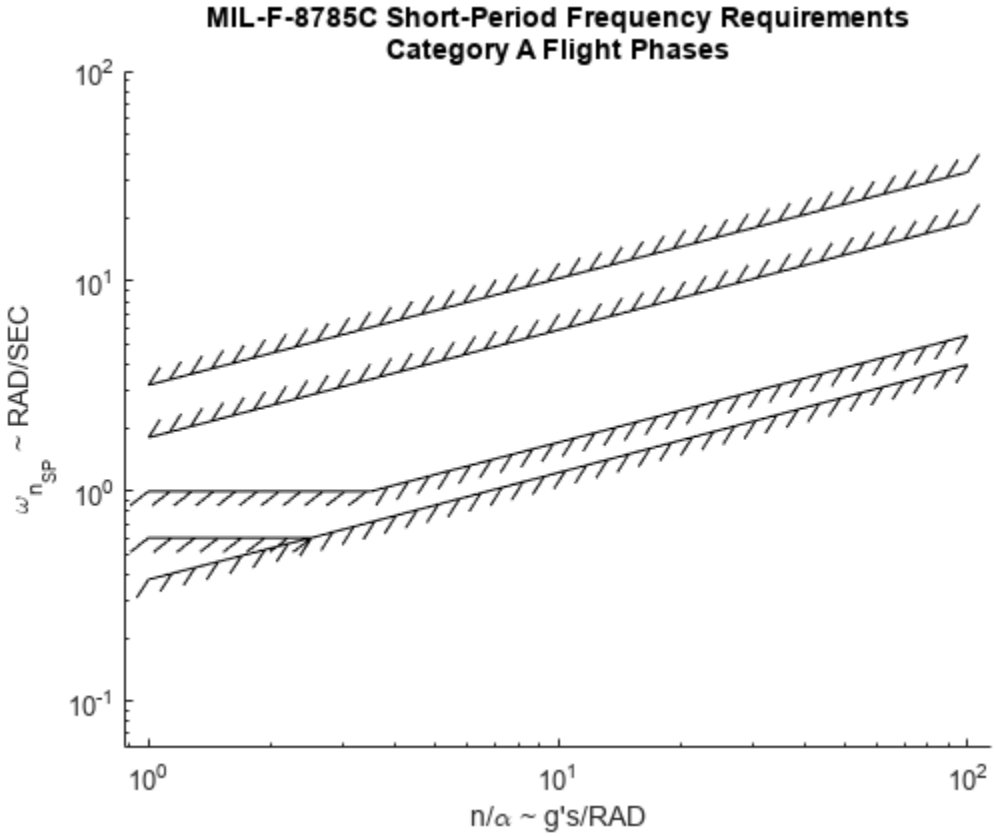
`[line,bline] = shortPeriodCategoryAPlot(___)` returns a vector of line objects `lineobjects` and a vector of `boundaryline` objects `boundary_lineobjects`. Use `lineobjects` and `boundary_lineobjects` to modify properties of a specific plot after it is created. Specify arguments as previously listed.

Examples

Plot MIL-F-8785C Short-Period Category A Requirements

Plot the reference MIL-F-8785C short-period category A requirements.

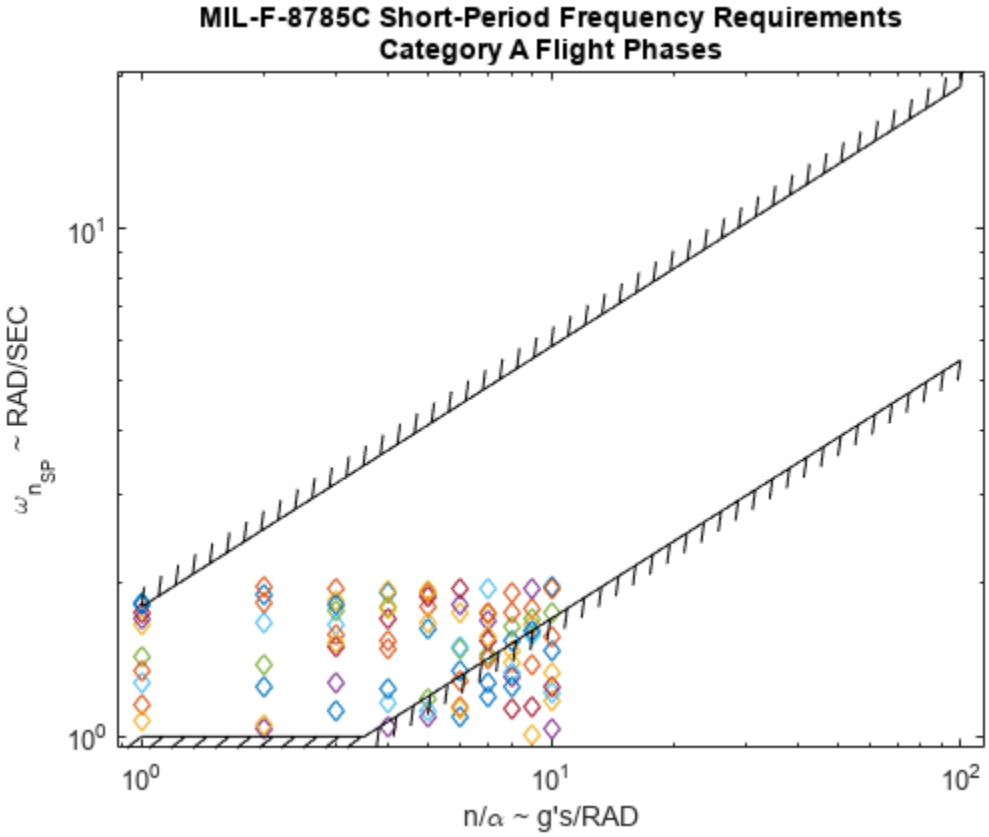
```
shortPeriodCategoryAPlot([])
```



Plot `nalpha` and `omega` Data Against Level 1 A Requirements

Plot `nalpha` and `omega` data against the level 1 short-period category A requirements using diamond markers.

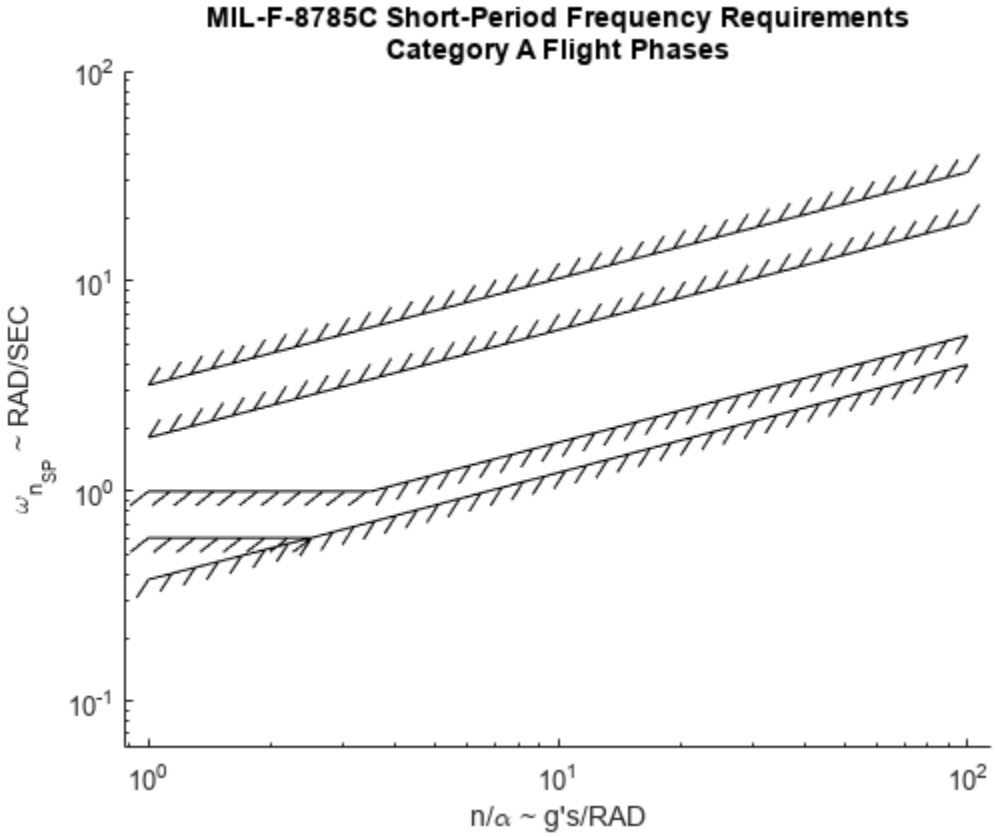
```
nalpha = 1:10;
omega = rand(10)+1;
shortPeriodCategoryAPlot(nalpha,omega,"d","Level","1")
```



Plot MIL-F-8785C Short-Period Category A Requirements and Return Vectors of Line Objects and Boundary Line Objects

Plot the reference MIL-F-8785C short-period category A requirements. Return line objects and boundary line objects in *h* and *b*.

```
[h,b] = shortPeriodCategoryAPlot([])
```




```
h =
  0x1 empty Line array.


b =
  6x1 BoundaryLine array:

  BoundaryLine    (Level 1)
  BoundaryLine    (Level 1)
  BoundaryLine    (Level 2)
  BoundaryLine    (Level 2)
  BoundaryLine    (Level 3)
  BoundaryLine    (Level 2 & 3)
```

Input Arguments

`nalpha` — Load factor per angle of attack

scalar | vector | matrix

Load factor per angle of attack *n*/α, specified as a scalar, vector, or matrix, in g's/radian.

Data Types: `double`

`omega` — Short-period undamped natural frequency response

scalar | vector | matrix

Short-period undamped natural frequency response ω_{nSP}, specified as a scalar, vector, or matrix, in radians/second.

Data Types: `double`

ax — Valid axes

scalar handle

Valid axes, specified as a scalar handle. By default, this function plots to the current axes, obtainable with the `gca` function.

Data Types: `double`

LineSpec — Line style, marker, and color

string scalar | character vector

Line style, marker, and color, specified as a string scalar or character vector containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `"--or"` is a red dashed line with circle markers.

Line Style	Description	Resulting Line
`"-"`	Solid line	
`"--"`	Dashed line	
`":"`	Dotted line	
`"-."`	Dash-dotted line	

Marker	Description	Resulting Marker
`"o"`	Circle	
`"+"`	Plus sign	
`"*"`	Asterisk	
`"."`	Point	
`"x"`	Cross	
`"_"`	Horizontal line	
`"|"`	Vertical line	
`"square"`	Square	
`"diamond"`	Diamond	

Marker	Description	Resulting Marker
"^"	Upward-pointing triangle	
"v"	Downward-pointing triangle	
">"	Right-pointing triangle	
"<"	Left-pointing triangle	
"pentagram"	Pentagram	
"hexagram"	Hexagram	

Color Name	Short Name	RGB Triplet	Appearance
"red"	"r"	[1 0 0]	
"green"	"g"	[0 1 0]	
"blue"	"b"	[0 0 1]	
"cyan"	"c"	[0 1 1]	
"magenta"	"m"	[1 0 1]	
"yellow"	"y"	[1 1 0]	
"black"	"k"	[0 0 0]	
"white"	"w"	[1 1 1]	

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Note These properties are only a subset. For a full list, see Line Properties.

Example: `"Level","1"`

`level` — Requirement level

`"All"` (default) | `"1"` | `"2"` | `"3"`

Requirement level to plot, specified as:

- `"All"`
- `"1"`
- `"2"`
- `"3"`

Data Types: `double`

Output Arguments

`line` — One or more line objects
scalar | vector

One or more line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific line. For a list of properties, see Line Properties.

`bline` — One or more boundary line objects
scalar | vector

One or more boundary line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line. For a list of properties, see Line Properties.

Version History

Introduced in R2021b

See Also

`altitudeEnvelopeContour` | `shortPeriodCategoryBPlot` | `shortPeriodCategoryCPlot` | `boundaryline` | `line` | Line Properties | `plot`

shortPeriodCategoryBPlot

Draw MIL-F-8785C short-period category B requirements plot

Syntax

```
shortPeriodCategoryBPlot(nalpha,omega)
shortPeriodCategoryBPlot(nalpha,omega,LineSpec)
shortPeriodCategoryBPlot(nalpha1,omega1,LineSpec1,...nalphan,omegan,
LineSpecn)

shortPeriodCategoryBPlot(___,Name,Value)
shortPeriodCategoryBPlot(ax,___)

[line,bline] = shortPeriodCategoryBPlot(___)
```

Description

Basic Syntax and Line Specification

`shortPeriodCategoryBPlot(nalpha,omega)` plots vector `omega` versus vector `nalpha`. If `nalpha` or `omega` is a matrix, then the function plots the vector versus the rows or columns of the matrix, whichever are aligned. If `nalpha` is a scalar and `omega` is a vector, the function creates the disconnected line objects and plots them as discrete points vertically at `nalpha`. This function is based on the MATLAB `plot` function.

`shortPeriodCategoryBPlot(nalpha,omega,LineSpec)` plots short-period category A requirements specified by the line specification `LineSpec`.

`shortPeriodCategoryBPlot(nalpha1,omega1,LineSpec1,...nalphan,omegan, LineSpecn)` combines the plots specified by the `nalpha`, `omega`, and `linespec`. It sets the line style, marker type, and color for each line. You can mix `nalpha`, `omega`, `LineSpec` triplets with `nalpha`, `omega` arguments, for example, `plot(nalpha1,omega1,nalpha2,omega2,LineSpec2,nalpha3,omega3)`.

Name-Value Arguments and Axes Specification

`shortPeriodCategoryBPlot(___,Name,Value)` plots an altitude envelope contour specified by one or more `Name,Value` arguments. Specify name-value arguments after all other input arguments.

`shortPeriodCategoryBPlot(ax,___)` draws an altitude contour plot onto the axes `ax`. Specify arguments as previously listed after the `ax` argument.

Return Line Objects

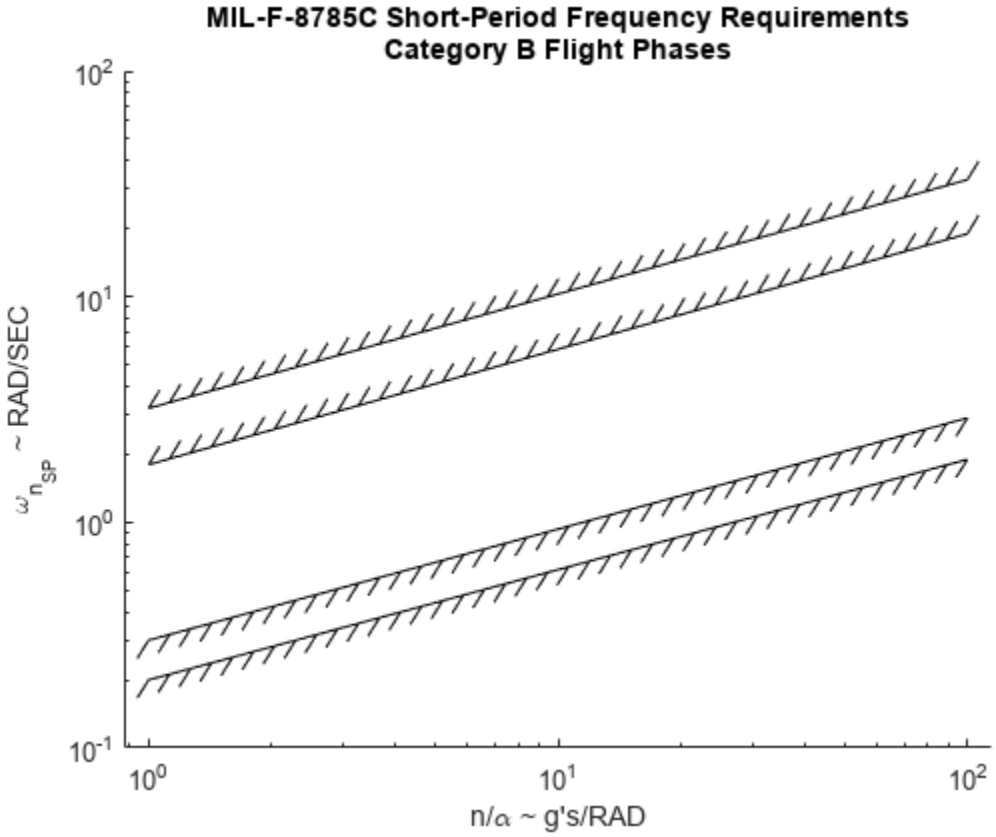
`[line,bline] = shortPeriodCategoryBPlot(___)` returns a vector of line objects `lineobjects` and a vector of `boundaryline` objects `boundary_lineobjects`. Use `lineobjects` and `boundary_lineobjects` to modify properties of a specific plot after it is created. Specify arguments as previously listed.

Examples

Plot MIL-F-8785C Short-Period Category B Requirements

Plot the reference MIL-F-8785C short-period category B requirements.

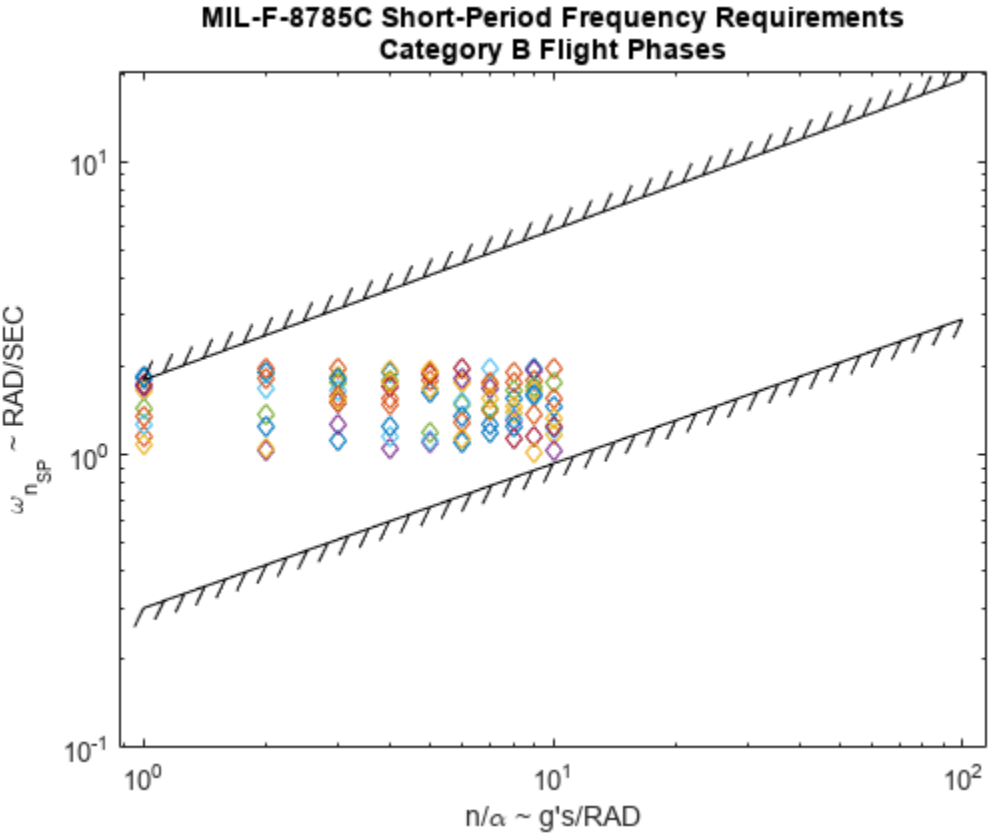
```
shortPeriodCategoryBPlot([])
```



Plot `nalpha` and `omega` Data Against Level 1 B Requirements

Plot `nalpha` and `omega` data against the level 1 short-period category B requirements using diamond markers.

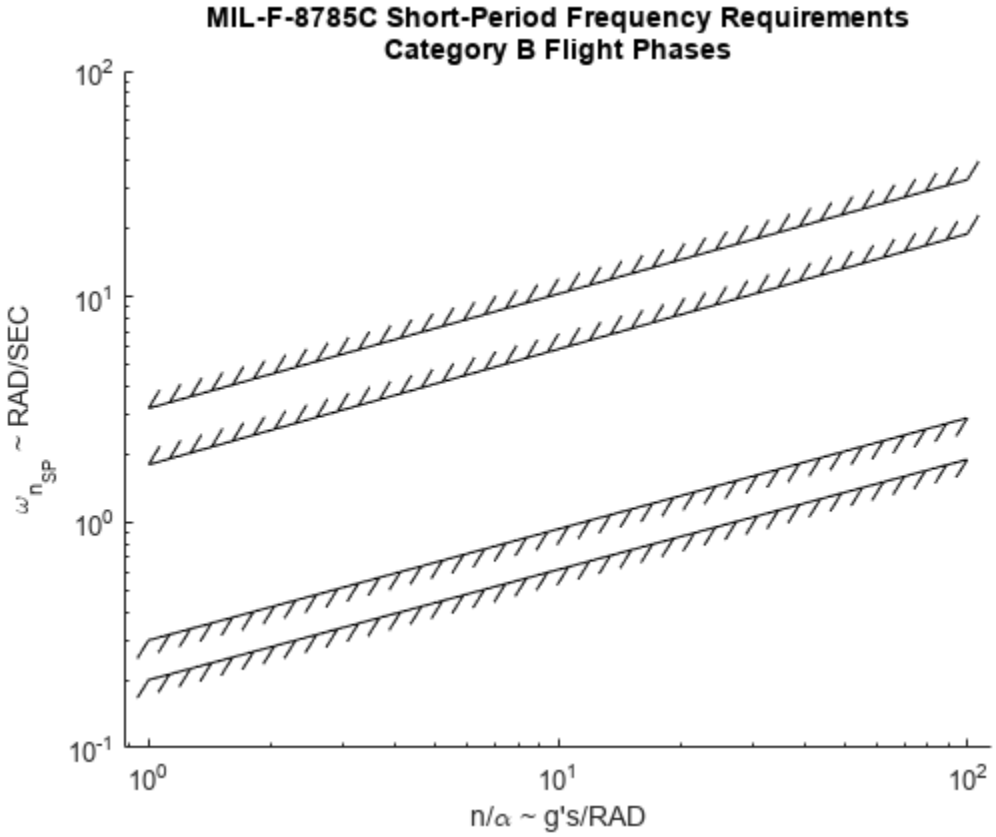
```
nalpha = 1:10;
omega = rand(10)+1;
shortPeriodCategoryBPlot(nalpha,omega,"d","Level","1")
```



Plot MIL-F-8785C Short-Period Category B Requirements and Return Vectors of Line Objects and Boundary Line Objects

Plot the reference MIL-F-8785C short-period category B requirements. Return line objects and boundary line objects in *h* and *b*.

```
[h,b] = shortPeriodCategoryBPlot([])
```




```
h =
  0x1 empty Line array.


b =
  4x1 BoundaryLine array:

  BoundaryLine    (Level 1)
  BoundaryLine    (Level 1)
  BoundaryLine    (Level 2)
  BoundaryLine    (Level 2 & 3)
```

Input Arguments

`nalpha` — Load factor per angle of attack

scalar | vector | matrix

Load factor per angle of attack *n*/α, specified as a scalar, vector, or matrix, in g's/radian.

Data Types: `double`

`omega` — Short-period undamped natural frequency response

scalar | vector | matrix

Short-period undamped natural frequency response ω_{nSP}, specified as a scalar, vector, or matrix, in radians/second.

Data Types: `double`

ax — Valid axes

scalar handle

Valid axes, specified as a scalar handle. By default, this function plots to the current axes, obtainable with the `gca` function.

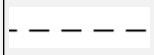
Data Types: `double`

LineSpec — Line style, marker, and color

string scalar | character vector

Line style, marker, and color, specified as a string scalar or character vector containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `"--or"` is a red dashed line with circle markers.

Line Style	Description	Resulting Line
`"-"`	Solid line	
`"--"`	Dashed line	
`":"`	Dotted line	
`"-."`	Dash-dotted line	

Marker	Description	Resulting Marker
`"o"`	Circle	
`"+"`	Plus sign	
`"*"`	Asterisk	
`"."`	Point	
`"x"`	Cross	
`"_"`	Horizontal line	
`"|"`	Vertical line	
`"square"`	Square	
`"diamond"`	Diamond	
`"^"`	Upward-pointing triangle	
`"v"`	Downward-pointing triangle	

Marker	Description	Resulting Marker
">"	Right-pointing triangle	
"<"	Left-pointing triangle	
"pentagram"	Pentagram	
"hexagram"	Hexagram	

Color Name	Short Name	RGB Triplet	Appearance
"red"	"r"	[1 0 0]	
"green"	"g"	[0 1 0]	
"blue"	"b"	[0 0 1]	
"cyan"	"c"	[0 1 1]	
"magenta"	"m"	[1 0 1]	
"yellow"	"y"	[1 1 0]	
"black"	"k"	[0 0 0]	
"white"	"w"	[1 1 1]	

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Note These properties are only a subset. For a full list, see Line Properties.

Example: `"Level","1"`

`level` — Requirement level
`"All"` (default) | `"1"` | `"2"` | `"3"`

Requirement level to plot, specified as:

- `"All"`
- `"1"`
- `"2"`
- `"3"`

Data Types: `double`

Output Arguments

`line` — One or more line objects
scalar | vector

One or more line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific line. For a list of properties, see Line Properties.

`bline` — One or more boundary line objects
scalar | vector

One or more boundary line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line. For a list of properties, see Line Properties.

Version History

Introduced in R2021b

See Also

`altitudeEnvelopeContour` | `shortPeriodCategoryAPlot` | `shortPeriodCategoryCPlot` | `boundaryline` | `line` | Line Properties | `plot`

shortPeriodCategoryCPlot

Draw MIL-F-8785C short-period category C requirements plot

Syntax

```
shortPeriodCategoryCPlot(nalpha,omega)
shortPeriodCategoryCPlot(nalpha,omega,LineSpec)
shortPeriodCategoryCPlot(nalpha1,omega1,LineSpec1,...nalphan,omegan,
LineSpecn)

shortPeriodCategoryCPlot(___,Name,Value)
shortPeriodCategoryCPlot(ax,___)

[line,bline] = shortPeriodCategoryCPlot(___)
```

Description

Basic Syntax and Line Specification

`shortPeriodCategoryCPlot(nalpha,omega)` plots vector `omega` versus vector `nalpha`. If `nalpha` or `omega` is a matrix, then the function plots the vector versus the rows or columns of the matrix, whichever are aligned. If `nalpha` is a scalar and `omega` is a vector, the function creates the disconnected line objects and plots them as discrete points vertically at `nalpha`. This function is based on the MATLAB `plot` function.

`shortPeriodCategoryCPlot(nalpha,omega,LineSpec)` plots short-period category A requirements specified by the line specification `LineSpec`.

`shortPeriodCategoryCPlot(nalpha1,omega1,LineSpec1,...nalphan,omegan, LineSpecn)` combines the plots specified by the `nalpha`, `omega`, and `linespec`. It sets the line style, marker type, and color for each line. You can mix `nalpha`, `omega`, `LineSpec` triplets with `nalpha`, `omega` arguments, for example, `plot(nalpha1,omega1,nalpha2,omega2,LineSpec2,nalpha3,omega3)`.

Name-Value Arguments and Axes Specification

`shortPeriodCategoryCPlot(___,Name,Value)` plots an altitude envelope contour specified by one or more `Name,Value` arguments. Specify name-value arguments after all other input arguments.

`shortPeriodCategoryCPlot(ax,___)` draws an altitude contour plot onto the axes `ax`. Specify arguments as previously listed after the `ax` argument.

Return Line Objects

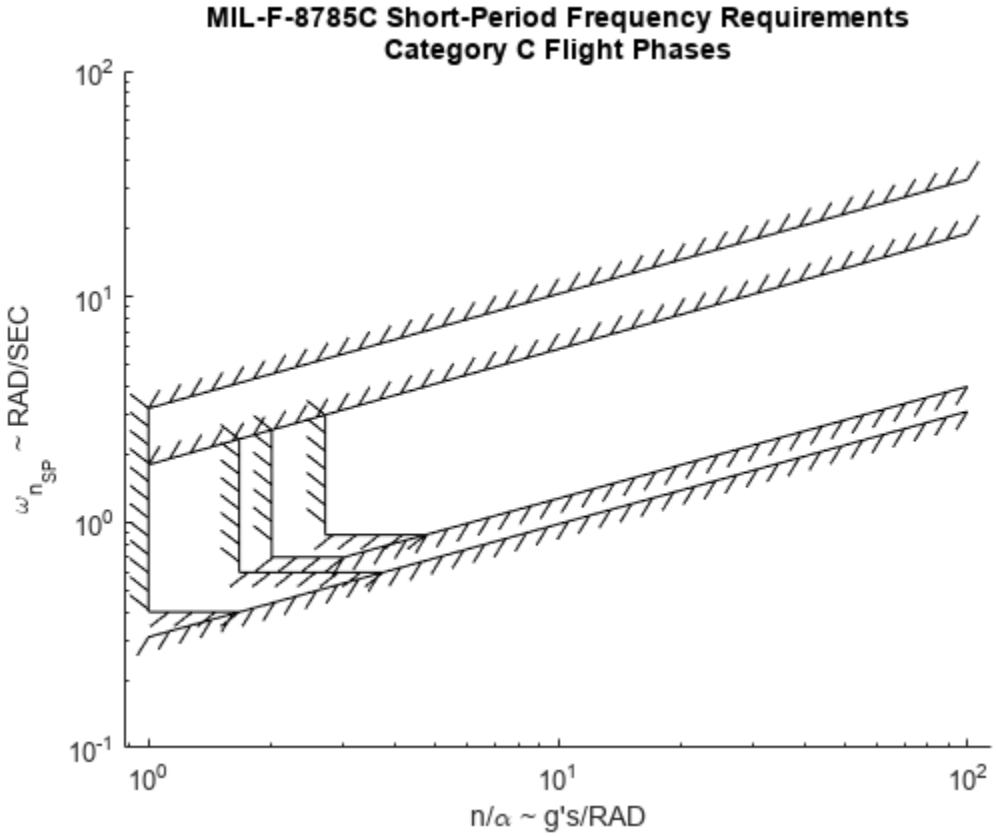
`[line,bline] = shortPeriodCategoryCPlot(___)` returns a vector of line objects `lineobjects` and a vector of `boundaryline` objects `boundary_lineobjects`. Use `lineobjects` and `boundary_lineobjects` to modify properties of a specific plot after it is created. Specify arguments as previously listed.

Examples

Plot MIL-F-8785C Short-Period Category C Requirements

Plot the reference MIL-F-8785C short-period category C requirements.

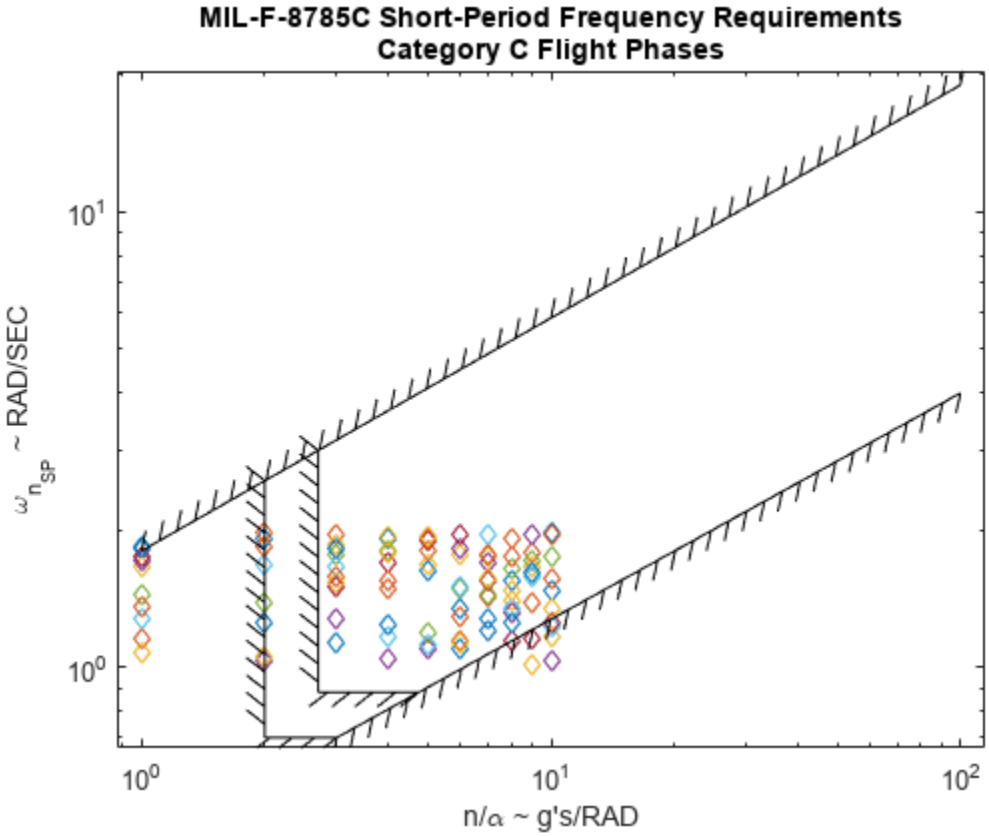
```
shortPeriodCategoryCPlot([])
```



Plot `nalpha` and `omega` Data Against Level 1 C Requirements

Plot `nalpha` and `omega` data against the level 1 short-period category C requirements using diamond markers.

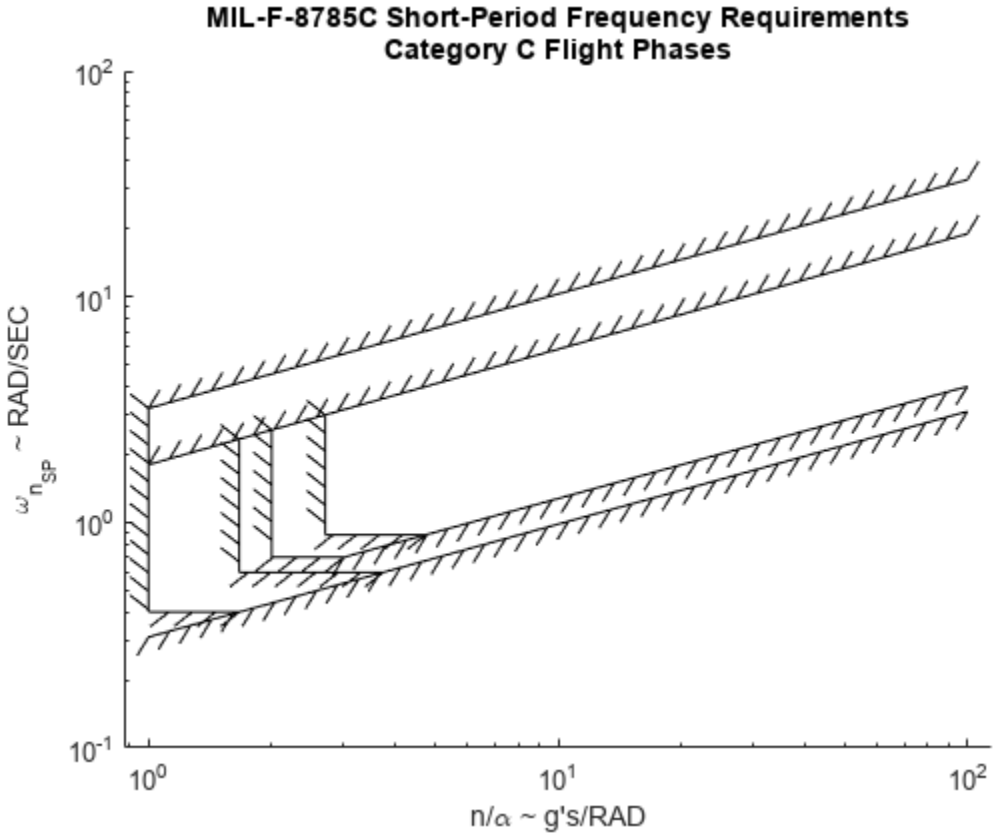
```
nalpha = 1:10;
omega = rand(10)+1;
shortPeriodCategoryCPlot(nalpha,omega,"d","Level","1")
```



Plot MIL-F-8785C Short-Period Category C Requirements and Return Vectors of Line Objects and Boundary Line Objects

Plot the reference MIL-F-8785C short-period category C requirements. Return line objects and boundary line objects in *h* and *b*.

```
[h,b] = shortPeriodCategoryCPlot([])
```




```
h =
  0x1 empty Line array.


b =
  8x1 BoundaryLine array:

  BoundaryLine    (Level 1)
  BoundaryLine    (Level 1)
  BoundaryLine    (Level 1 - Classes II-L, III)
  BoundaryLine    (Level 1 - Classes I, II-C, IV)
  BoundaryLine    (Level 2 - Classes II-L, III)
  BoundaryLine    (Level 2 - Classes I, II-C, IV)
  BoundaryLine    (Level 2)
  BoundaryLine    (Level 2 & 3)
```

Input Arguments

`nalpha` — Load factor per angle of attack

scalar | vector | matrix

Load factor per angle of attack *n*/α, specified as a scalar, vector, or matrix, in g's/radian.

Data Types: `double`

`omega` — Short-period undamped natural frequency response

scalar | vector | matrix

Short-period undamped natural frequency response ω_{nSP}, specified as a scalar, vector, or matrix, in radians/second.

Data Types: `double`

`ax` — Valid axes

scalar handle

Valid axes, specified as a scalar handle. By default, this function plots to the current axes, obtainable with the `gca` function.

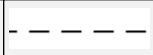
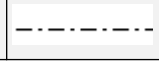
Data Types: `double`

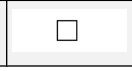
`LineSpec` — Line style, marker, and color

string scalar | character vector

Line style, marker, and color, specified as a string scalar or character vector containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `"--or"` is a red dashed line with circle markers.

Line Style	Description	Resulting Line
`"-"`	Solid line	
`"--"`	Dashed line	
`":"`	Dotted line	
`"-."`	Dash-dotted line	

Marker	Description	Resulting Marker
`"o"`	Circle	
`"+"`	Plus sign	
`"*"`	Asterisk	
`"."`	Point	
`"x"`	Cross	
`"_"`	Horizontal line	
`"|"`	Vertical line	
`"square"`	Square	

Marker	Description	Resulting Marker
"diamond"	Diamond	
"^"	Upward-pointing triangle	
"v"	Downward-pointing triangle	
">"	Right-pointing triangle	
"<"	Left-pointing triangle	
"pentagram"	Pentagram	
"hexagram"	Hexagram	

Color Name	Short Name	RGB Triplet	Appearance
"red"	"r"	[1 0 0]	
"green"	"g"	[0 1 0]	
"blue"	"b"	[0 0 1]	
"cyan"	"c"	[0 1 1]	
"magenta"	"m"	[1 0 1]	
"yellow"	"y"	[1 1 0]	
"black"	"k"	[0 0 0]	
"white"	"w"	[1 1 1]	

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Note These properties are only a subset. For a full list, see Line Properties.

Example: `"Level","1"`

`level` — Requirement level

`"All"` (default) | `"1"` | `"2"` | `"3"`

Requirement level to plot, specified as:

- `"All"`
- `"1"`
- `"2"`
- `"3"`

Data Types: `double`

Output Arguments

`line` — One or more line objects

scalar | vector

One or more line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific line. For a list of properties, see Line Properties.

`bline` — One or more boundary line objects

scalar | vector

One or more boundary line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line. For a list of properties, see Line Properties.

Version History

Introduced in R2021b

See Also

`altitudeEnvelopeContour` | `shortPeriodCategoryAPlot` | `shortPeriodCategoryBPlot` | `boundaryline` | `line` | Line Properties | `plot`

showAll

Package: `matlabshared.satellitescenario`

Show all graphics in viewer

Syntax

```
showAll(viewer)
```

Description

`showAll(viewer)` shows all graphics in the specified satellite scenario viewer.

Examples

Show All Hidden Satellite Scenario Objects

Create a satellite scenario object.

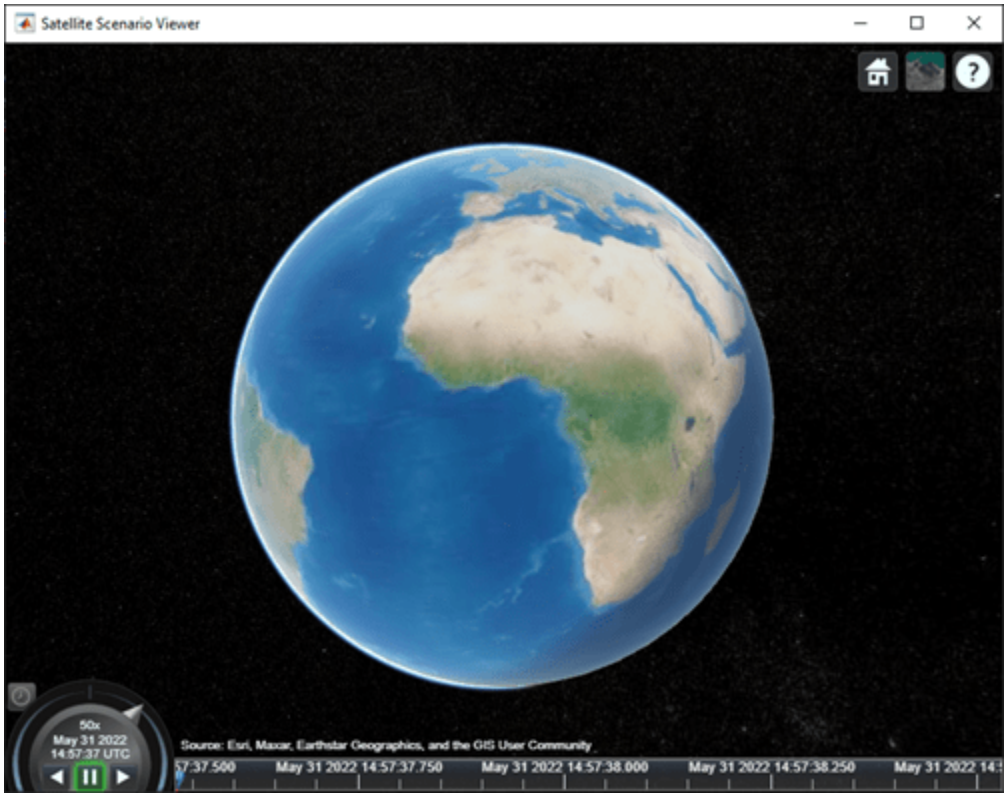
```
sc = satelliteScenario;
```

Specify not to automatically show scenario entities in an open Satellite Scenario Viewer.

```
sc.AutoShow = false;
```

Launch the Satellite Scenario Viewer.

```
v = satelliteScenarioViewer(sc);
```



Add a constellation of satellites to the scenario.

```
tleFile = "leoSatelliteConstellation.tle";
sat = satellite(sc,tleFile);
```

Add a ground station to the scenario.

```
gs = groundStation(sc);
```

Visualize the satellite scenario objects using the Satellite Scenario Viewer.

```
showAll(v);
```

Input Arguments

`viewer` — Satellite Scenario Viewer

`satelliteScenarioViewer` object

Satellite Scenario Viewer, specified as a scalar `satelliteScenarioViewer` object.[9]

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `access` | `groundStation` | `satelliteScenarioViewer` | `conicalSensor`

Functions

`show` | `play` | `hide` | `campos` | `camroll` | `campitch` | `camheading` | `camheight` | `camtarget`

Topics

“Satellite Scenario Key Concepts” on page 2-65

9 Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

siderealTime

Greenwich mean and apparent sidereal times

Syntax

```
thGMST = siderealTime(utcJD)
thGMST = siderealTime(utcJD, dUT1, dAT)
[thGMST,thGAST] = siderealTime(utcJD,dUT1,dAT)
```

Description

`thGMST = siderealTime(utcJD)` calculates mean sidereal time at a specific Coordinated Universal Time (UTC), specified as a Julian date.

`thGMST = siderealTime(utcJD, dUT1, dAT)` calculates mean sidereal time at a specific Coordinated Universal Time (UTC) at a higher precision using Earth orientation parameters.

`[thGMST,thGAST] = siderealTime(utcJD,dUT1,dAT)` calculates mean and apparent sidereal times.

Note Apparent sidereal time calculation requires that you download ephemeris data using the Add-On Explorer. To start the Add-On Explorer, in the MATLAB Command Window, type `aeroDataPackage`. on the MATLAB desktop toolstrip, click the **Add-Ons** button.

Examples

Calculate Greenwich Sidereal Times for Particular Date

Calculate Greenwich sidereal times at 12:00 on January 4, 2019.

```
jd = juliandate([2019 1 4 12 0 0]);
[thGMST, thGAST] = siderealTime(jd);
```

Calculate Greenwich Sidereal Times for Whole Month

Calculate Greenwich sidereal times at 12:00 for the month of January, 2019:

```
dates = datetime([2019 1 4 12 0 0]);
dates = dates + days(1:30)';
jdJan = juliandate(dates);
[thGMST, thGAST] = siderealTime(jdJan);
```

Input Arguments

`utcJD` — UTC as Julian date

scalar

Coordinated Universal Time (UTC) as a Julian date, specified as a scalar.

Tip To calculate the Julian date for a particular date, use the `juliandate` function.

Data Types: `double`

`dUT1` — Difference between CUT and UT1
`0` (default) | scalar

Difference between the Coordinated Universal Time (UTC) and Universal Time (UT1), specified as a scalar, in seconds.

`dAT` — Difference between TAI and UTC
`0` (default) | scalar

Difference between International Atomic Time (TAI) and Coordinated Universal Time (UTC), specified as a scalar, in seconds.

Output Arguments

`thGMST` — Greenwich mean sidereal time
scalar

Greenwich mean sidereal time, specified as a scalar, in seconds.

`thGAST` — Greenwich apparent sidereal time
scalar

Greenwich apparent sidereal time, specified as a scalar, in seconds.

Version History

Introduced in R2021a

References

[1] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. alg. 1 and eqs. 1-63. New York: McGraw-Hill, 1997.

See Also

`ecef2eci` | `eci2ecef` | `dcmeci2ecef` | CubeSat Vehicle

setCoefficient

Class: `Aero.FixedWing.Surface`
Package: `Aero`

Set coefficient values for `Aero.FixedWing.Surface` object

Syntax

```
fixedWingSurface = setCoefficient(fixedWingSurface,stateOutput,stateVariable,
value)
fixedWingSurface = setCoefficient( ___ ,Name,Value)
```

Description

`fixedWingSurface = setCoefficient(fixedWingSurface,stateOutput,stateVariable,
value)` sets the coefficient value `value` to the coefficient specified by `stateOutput` and `stateVariable` and returns the modified surface object.

`fixedWingSurface = setCoefficient( ___ ,Name,Value)` sets the coefficient value `value` to the coefficient specified by `stateOutput` and `stateVariable` and returns the modified surface object.

Input Arguments

`fixedWingSurface` — `Aero.FixedWing.Surface` object on which to set coefficient

scalar

`Aero.FixedWing.Surface` object on which to set coefficient, specified as a scalar.

`stateOutput` — State output

6-by-1 vector

State output, specified as a 6-by-1 vector where each entry is a valid state output. For more information on state outputs, see `Aero.FixedWing.Coefficient`.

Data Types: `char` | `string`

`stateVariable` — State variable

vector

State variable, specified as a vector where each entry is a valid state variable. Valid state variables depend on the coefficients defined on the object. For more information on fixed-wing states, see `Aero.FixedWing.State`.

Data Types: `char` | `string`

`value` — State values

vector of cells

State values, specified as a vector of cells where each cell is a numeric constant or a `Simulink.LookupTable` object. For more information on coefficient values, see `Aero.FixedWing.Coefficient`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'AddVariable','on'`

`Component` — Component name
string

Component name, specified as a string. Valid component names depend on the object properties and all subcomponents on the object. The default component name is the current object.

Data Types: `char` | `string`

`AddVariable` — Option to add state variable
`off` (default) | `on`

Option to add state variable if desired variable is missing, specified as:

- `'on'` — Add a state variable.
- `'off'` — Do not add a state variable.

Data Types: `logical`

Output Arguments

`fixedWingSurface` — Modified fixed-wing surface object
`Aero.FixedWing.Surface`

Modified fixed-wing surface object on which coefficient is set, returned as `Aero.FixedWing.Surface`.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Aero.FixedWing.Coefficient` | `Aero.FixedWing.Surface` | `Aero.FixedWing.Thrust` | `getCoefficient`

setCoefficient

Class: `Aero.FixedWing.Thrust`
Package: `Aero`

Set coefficient values for `Aero.FixedWing.Thrust` object

Syntax

```
fixedWingThrust = setCoefficient(fixedWingThrust,stateOutput,stateVariable,
value)
fixedWingThrust = setCoefficient( ___ ,Name,Value)
```

Description

`fixedWingThrust = setCoefficient(fixedWingThrust,stateOutput,stateVariable, value)` sets the coefficient value `value` to the coefficient specified by `stateOutput` and `stateVariable` and returns the modified thrust object.

`fixedWingThrust = setCoefficient( ___ ,Name,Value)` sets the coefficient value `value` to the coefficient specified by `stateOutput` and `stateVariable` and returns the modified thrust object.

Input Arguments

`fixedWingCoefficient` — `Aero.FixedWing.Thrust` object on which to set coefficient

`scalar`

`Aero.FixedWing.Thrust` object on which to set coefficient, specified as a scalar.

`stateOutput` — State output

6-by-1 vector

State output, specified as a 6-by-1 vector where each entry is a valid state output. For more information on state outputs, see `Aero.FixedWing.Coefficient`.

Data Types: `char` | `string`

`stateVariable` — State variable

vector

State variable, specified as a vector where each entry is a valid state variable. Valid state variables depend on the coefficients defined on the object. For more information on fixed-wing states, see `Aero.FixedWing.State`.

Data Types: `char` | `string`

`value` — State values

vector of cells

State values, specified as a vector of cells where each cell is a numeric constant or a `Simulink.LookupTable` object. For more information on coefficient values, see `Aero.FixedWing.Coefficient`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'AddVariable','on'`

`Component` — Component name
string

Component name, specified as a string. Valid component names depend on the object properties and all subcomponents on the object. The default component name is the current object.

Data Types: `char` | `string`

`AddVariable` — Option to add state variable
`off` (default) | `on`

Option to add state variable if desired state variable is missing, specified as:

- `'on'` — Add a state variable.
- `'off'` — Do not add a state variable.

Data Types: `logical`

Output Arguments

`fixedWingThrust` — Modified fixed-wing thrust object
`Aero.FixedWing.Thrust`

Modified fixed-wing thrust object on which coefficient is set, returned as `Aero.FixedWing.Thrust`.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing.Thrust` | `getCoefficient`

setState

Class: `Aero.FixedWing.State`
Package: `Aero`

Set state value to `Aero.FixedWing.State` object

Syntax

```
state = setState(state,statename,value)
```

Description

`state = setState(state,statename,value)` sets the state `value` to a specified state name value.

Input Arguments

`state` — `Aero.FixedWing.State` object

scalar

`Aero.FixedWing.State` object, specified as a scalar.

`statename` — State names

vector

State names, specified in a vector. You cannot set effective control variables created with asymmetric control surfaces. For more information on state names, see the `Aero.FixedWing.State` “Properties” on page 4-79.

Tip Each vector of `statename` and `value` must be the same length.

Data Types: `char` | `string`

`value` — State values

vector

State values, specified as a vector.

- If the states are all scalar constants, `value` is a numeric vector.
- If one of more states are not scalar constants, `value` is a cell vector.

Tip Each vector of `statename` and `value` must be the same length.

Output Arguments

`state` — Modified `Aero.FixedWing.State` object

vector

Modified input object with the modified states at the specified locations.

Examples

Set Pitch Angle of Cruise State

Set the pitch angle of a cruise state.

```
[C182, CruiseState] = astC182();
CruiseState = setState(CruiseState, "Alpha", 5)
```

```
CruiseState =

  State with properties:

                  Alpha: NaN
                   Beta: NaN
               AlphaDot: 0
                BetaDot: 0
                   Mass: 82.2981
                Inertia: [3×3 table]
         CenterOfGravity: [1.2936 0 0]
        CenterOfPressure: [1.2250 0 0]
             AltitudeMSL: 5000
            GroundHeight: 0
                      XN: 0
                      XE: 0
                      XD: -5000
                       U: 220.1000
                       V: 0
                       W: 0
                     Phi: 0
                   Theta: 0
                     Psi: 0
                       P: 0
                       Q: 0
                       R: 0
                  Weight: 2.6500e+03
             AltitudeAGL: 5000
                Airspeed: NaN
             GroundSpeed: 220.1000
              MachNumber: NaN
            BodyVelocity: [NaN NaN NaN]
          GroundVelocity: [220.1000 0 0]
                      Ur: NaN
                      Vr: NaN
                      Wr: NaN
       FlightPathAngle: 0
           CourseAngle: 0
    InertialToBodyMatrix: [3×3 double]
    BodyToInertialMatrix: [3×3 double]
        BodyToWindMatrix: [3×3 double]
        WindToBodyMatrix: [3×3 double]
         DynamicPressure: NaN
             Environment: [1×1 Aero.Aircraft.Environment]
              UnitSystem: "English (ft/s)"
```

```
           AngleSystem: "Radians"
      TemperatureSystem: "Fahrenheit"
          ControlStates: [1×4 Aero.Aircraft.ControlState]
       OutOfRangeAction: "Limit"
       DiagnosticAction: "Warning"
             Properties: [1×1 Aero.Aircraft.Properties]
```

Set U, V, and W Velocity Components

Set the *U*, *V*, and *W* velocity components of a cruise state.

```
[C182, CruiseState] = astC182();
CruiseState = setState(CruiseState, ["U", "V", "W"], [50, 1, 10])
```

```
CruiseState =

  State with properties:

                 Alpha: 0.1974
                  Beta: 0.0196
              AlphaDot: 0
               BetaDot: 0
                  Mass: 82.2981
               Inertia: [3×3 table]
        CenterOfGravity: [1.2936 0 0]
      CenterOfPressure: [1.2250 0 0]
           AltitudeMSL: 5000
          GroundHeight: 0
                    XN: 0
                    XE: 0
                    XD: -5000
                     U: 50
                     V: 1
                     W: 10
                   Phi: 0
                 Theta: 0
                   Psi: 0
                     P: 0
                     Q: 0
                     R: 0
                Weight: 2.6500e+03
            AltitudeAGL: 5000
              Airspeed: 51
            GroundSpeed: 51
             MachNumber: 0.0465
           BodyVelocity: [50 1 10]
         GroundVelocity: [50 1 10]
                    Ur: 50
                    Vr: 1
                    Wr: 10
       FlightPathAngle: 0.1974
           CourseAngle: 0.0200
   InertialToBodyMatrix: [3×3 double]
   BodyToInertialMatrix: [3×3 double]
       BodyToWindMatrix: [3×3 double]
       WindToBodyMatrix: [3×3 double]
```

```
  DynamicPressure: 2.6639
      Environment: [1×1 Aero.Aircraft.Environment]
       UnitSystem: "English (ft/s)"
      AngleSystem: "Radians"
TemperatureSystem: "Fahrenheit"
    ControlStates: [1×4 Aero.Aircraft.ControlState]
 OutOfRangeAction: "Limit"
 DiagnosticAction: "Warning"
       Properties: [1×1 Aero.Aircraft.Properties]
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing.State` | `getState` | `setupControlStates`

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')
```

Version History

Introduced in R2008b

See Also

`ClearTimer`

setupControlStates

Class: `Aero.FixedWing.State`
Package: `Aero`

Set up control states for `Aero.FixedWing.State` object

Syntax

```
state = setupControlStates(aircraft, state)
```

Description

`state = setupControlStates(aircraft, state)` sets up initial control states for the `Aero.FixedWing` object.

Input Arguments

`aircraft` — `Aero.FixedWing` object

scalar

`Aero.FixedWing` object, specified as a scalar.

Data Types: `double`

`state` — `Aero.FixedWing.State` object

scalar

`Aero.FixedWing.State` object, specified as a scalar.

Output Arguments

`state` — Modified `Aero.FixedWing.State` object

scalar

Modified `Aero.FixedWing.State` object, returned as a scalar.

Data Types: `double`

Examples

Initialize Control and Command States

Initialize the control and command states on a cruise state.

```
[C182, CruiseState] = astC182();
CruiseState = setupControlStates(CruiseState,C182)
```

```
CruiseState =
```

```
  State with properties:

                  Alpha: 0
                   Beta: 0
                AlphaDot: 0
                 BetaDot: 0
                    Mass: 82.2981
                 Inertia: [3×3 table]
          CenterOfGravity: [1.2936 0 0]
       CenterOfPressure: [1.2250 0 0]
            AltitudeMSL: 5000
           GroundHeight: 0
                     XN: 0
                     XE: 0
                     XD: -5000
                      U: 220.1000
                      V: 0
                      W: 0
                    Phi: 0
                  Theta: 0
                    Psi: 0
                      P: 0
                      Q: 0
                      R: 0
                 Weight: 2.6500e+03
             AltitudeAGL: 5000
                Airspeed: 220.1000
             GroundSpeed: 220.1000
              MachNumber: 0.2006
            BodyVelocity: [220.1000 0 0]
          GroundVelocity: [220.1000 0 0]
                      Ur: 220.1000
                      Vr: 0
                      Wr: 0
         FlightPathAngle: 0
             CourseAngle: 0
    InertialToBodyMatrix: [3×3 double]
    BodyToInertialMatrix: [3×3 double]
        BodyToWindMatrix: [3×3 double]
        WindToBodyMatrix: [3×3 double]
         DynamicPressure: 49.6149
             Environment: [1×1 Aero.Aircraft.Environment]
              UnitSystem: "English (ft/s)"
             AngleSystem: "Radians"
       TemperatureSystem: "Fahrenheit"
           ControlStates: [1×4 Aero.Aircraft.ControlState]
        OutOfRangeAction: "Limit"
        DiagnosticAction: "Warning"
              Properties: [1×1 Aero.Aircraft.Properties]
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing.State` | `getState` | `setState`

show

Package: `matlabshared.satellitescenario`

Show object in satellite scenario viewer

Syntax

```
show(item)
show(item,viewer)
```

Description

`show(item)` shows the item on all open Satellite Scenario Viewers.

`show(item,viewer)` shows the graphic on the Satellite Scenario Viewer specified by `v`.

Examples

Add Satellites to Scenario Using Keplerian Elements

Create a satellite scenario with a start time of 02-June-2020 8:23:00 AM UTC, and the stop time set to one day later. Set the simulation sample time to 60 seconds.

```
startTime = datetime(2020,6,02,8,23,0);
stopTime = startTime + days(1);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add two satellites to the scenario using their Keplerian elements.

```
semiMajorAxis = [10000000;15000000];
eccentricity = [0.01;0.02];
inclination = [0;10];
rightAscensionOfAscendingNode = [0;15];
argumentOfPeriapsis = [0;30];
trueAnomaly = [0;20];

sat = satellite(sc,semiMajorAxis,eccentricity,inclination, ...
    rightAscensionOfAscendingNode,argumentOfPeriapsis,trueAnomaly)
```

```
sat =
  1x2 Satellite array with properties:

    Name
    ID
    ConicalSensors
    Gimbals
    Transmitters
    Receivers
    Accesses
    GroundTrack
```

```
    Orbit
    OrbitPropagator
    MarkerColor
    MarkerSize
    ShowLabel
    LabelFontColor
    LabelFontSize
```

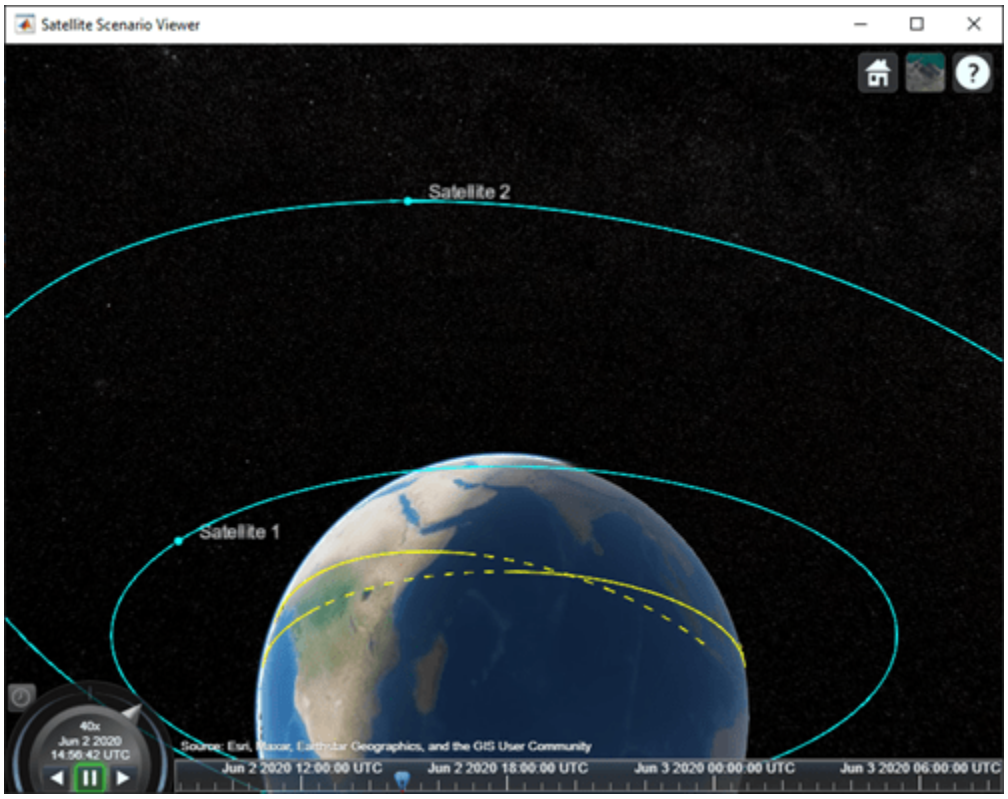
View the satellites in orbit and the ground tracks over one hour.

```
show(sat)
groundTrack(sat,LeadTime=3600)
```

```
ans=1×2 object
  1×2 GroundTrack array with properties:

    LeadTime
    TrailTime
    LineWidth
    LeadLineColor
    TrailLineColor
    VisibilityMode
```

Play the scenario and set the animation speed of the simulation to 40.

```
play(sc,PlaybackSpeedMultiplier=40)
```



Input Arguments

`item` — Item

`Satellite` object | `GroundStation` object | `ConicalSensor` object | `GroundTrack` object | `FieldofView` object | `Access` object

The item, specified as a `Satellite`, `GroundStation`, `ConicalSensors`, `GroundTrack`, `FieldOfView`, or `Access` object. These objects must belong to the same `satelliteScenario`, object.

Note If `item` is a satellite or a ground station, then the associated gimbals are also displayed on the viewer.

`viewer` — Satellite scenario viewer

row vector of all `satelliteScenarioViewer` objects (default) | scalar `satelliteScenarioViewer` object | array of `satelliteScenarioViewer` objects

Satellite scenario viewer, specified as a scalar, vector, or array of `satelliteScenarioViewer` objects.

Version History

Introduced in R2021a

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer`

Functions

`play` | `hide` | `access` | `groundStation` | `conicalSensor`

Topics

“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

show

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

Show animation object figure

Syntax

```
show(h)
```

Description

`show(h)` creates the figure graphics object for the animation object `h`. Use the `hide` function to close the figure.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

Examples

Show `Aero.Animation` Object

```
h = Aero.Animation;
idx1 = createBody(h,'pa24-250_orange.ac','Ac3d');
show(h);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

states

Package: `matlabshared.satellitescenario`

Obtain position and velocity of satellite

Syntax

```
pos = states(sat)
[pos,velocity] = states(sat)
[___] = states(sat,timeIn)
[___] = states(___,CoordinateFrame=C)
[pos,velocity,timeOut] = states(___)
```

Description

`pos = states(sat)` returns a 3-by-*n*-by-*m* array of the position history `pos` of each satellite in the vector `sat`, where *n* is the number of time samples and *m* is the number of satellites. The rows represent the *x*, *y*, and *z* coordinates of the satellite in the Geocentric Celestial Reference Frame (GCRF).

`[pos,velocity] = states(sat)` returns a 3-by-*n*-by-*m* array of the inertial velocity `velocity` of each satellite in the vector `sat` in the GCRF.

`[___] = states(sat,timeIn)` returns one or both of the outputs as 3-by-1-by-*m* arrays in addition to the position at the specified datetime `timeIn`. If no time zone is specified in `timeIn`, the time zone is assumed to be Coordinated Universal Time (UTC).

`[___] = states(___,CoordinateFrame=C)` returns the outputs in the coordinates specified by `C`.

`[pos,velocity,timeOut] = states(___)` returns the position and velocity history of the satellites and the corresponding datetime in UTC.

Examples

Obtain States of Satellite in ECEF Frame

Create a satellite scenario object.

```
startTime = datetime(2021,5,25);                        % May 25, 2021, 12:00 AM UTC
stopTime = datetime(2021,5,26);                         % May 26, 2021, 12:00 AM UTC
sampleTime = 60;                                        % In seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add a satellite to the scenario.

```
tleFile = "eccentricOrbitSatellite.tle";
sat = satellite(sc,tleFile);
```

Obtain the position and velocity of the satellite in the Earth-centered Earth-fixed (ECEF) frame corresponding to May 25, 2021, 10:30 PM UTC.

```
time = datetime(2021,5,25,22,30,0);
[position,velocity] = states(sat,time,"CoordinateFrame","ecef")
```

```
position = 3×1
10^7 ×

   -0.9431
   -3.0675
    2.7404


velocity = 3×1
10^3 ×

   -1.2166
    0.4198
   -1.6730
```

Input Arguments

`sat` — Satellite

row vector of `Satellite` objects

Satellite, specified as a row vector of `Satellite` objects.

`timeIn` — Time at which output is calculated

`datetime` scalar

Time at which the output is calculated, specified as a `datetime` scalar. If no time zone is specified in `timeIn`, the time zone is assumed to be UTC.

`C` — Coordinate frame

`'inertial'` | `'ecef'` | `'geographic'`

Coordinate frame in which the outputs are returned, specified as `'inertial'`, `'ecef'`, or `'geographic'`.

- The `'inertial'` option returns the position and velocity coordinates in the GCRF.
- The `'ecef'` option returns the position and velocity coordinates in the ECEF frame. For more information on ECEF frames, see “Earth-Centered Earth-Fixed Coordinates” on page 2-66.
- The `'geographic'` option returns the position as [*lat*; *lon*; *altitude*], where *lat* and *lon* are latitude and longitude in degrees and altitude is the height above the surface of the Earth in meters. The velocity returned is in the North-East-Down (NED) frame.

Output Arguments

`pos` — Position history

3-by-*n*-by-*m* array

Position history of the satellites in meters, returned as a 3-by-*n*-by-*m* array in the GCRF. *n* is the number of time samples and *m* is the number of satellites.

When the `AutoSimulate` property of the satellite scenario is `true`, the position history from `StartTime` to `StopTime` is returned. Otherwise, the position history from `StartTime` to `SimulationTime` is returned.

`velocity` — Velocity history
3-by-*n*-by-*m* array

Velocity history of the satellites in meters per second, returned as a 3-by-*n*-by-*m* array in the GCRF. *n* is the number of time samples and *m* is the number of satellites.

`timeOut` — Time samples of position and velocity
scalar | vector

Time samples of the position and velocity of the satellites, returned as a scalar or vector. If time histories of the position and velocity of the satellite are returned, `timeOut` is a row vector.

Version History

Introduced in R2021a

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer`

Functions
`satellite` | `groundStation`

Topics
“Model, Visualize, and Analyze Satellite Scenario” on page 2-77
“Satellite Scenario Key Concepts” on page 2-65

staticStability

Class: `Aero.FixedWing`
Package: `Aero`

Calculate static stability of fixed-wing aircraft

Syntax

```
stability = staticStability(aircraft,state)
stability = staticStability( ___ ,Name,Value)
[stability,derivatives] = staticStability( ___ )
```

Description

`stability = staticStability(aircraft,state)` calculates the static stability `stability` of a fixed-wing aircraft `aircraft` at an `Aero.FixedWing.State` `state`. This method calculates static stability from changes in body forces and moments due to perturbations of an aircraft state. By default, these states are airspeed, angle of attack, angle of side slip, and body roll rates. To change these states, see `criteriaTable`.

The `staticStability` method evaluates the changes in body forces and moments after a perturbation as either greater than, equal to, or less than 0 using the matching entry in the criteria table.

- If the evaluation of a criterion is met, the aircraft is statically stable at that condition.
- If the evaluation of a criterion is not met, the aircraft is statically unstable at that condition.
- If the perturbation value is set to `0`, the aircraft is statically neutral at that condition.

`stability = staticStability( ___ ,Name,Value)` calculates the static stability result with the specified `Name,Value` arguments. Specify any of the input argument combinations in the previous syntaxes followed by `Name,Value` pairs as the last input arguments.

`[stability,derivatives] = staticStability( ___ )` returns the body forces and moments derivatives table along with the static stability. Specify any of the input argument combinations in the previous syntaxes.

Input Arguments

`aircraft` — `Aero.FixedWing` object

scalar

`Aero.FixedWing` object, specified as a scalar.

Data Types: `double`

`state` — `Aero.FixedWing.State` object

scalar

`Aero.FixedWing.State` object, specified as a scalar.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'RelativePerturbation','1e-5'`

`CriteriaTable` — Static stability test criteria

6-by-8 table (default) | 6-by-*N* table

Static stability test criteria, specified as a 6-by-*N* table, where *N* is number of variables.

- If the value being evaluated is 0, it is neutral.
- If the value being evaluated does not meet the criteria, it is unstable.
- If the criterion is an empty string or is missing, then the stability result is an empty string.

The criteria table has these requirements:

- Each entry in the criteria table must be `'<'`, `'>'`, `''`, or missing.
- The table must have six rows: `'FX'`, `'FY'`, `'FZ'`, `'L'`, `'M'`, and `'N'`.
- *N* number of variables for columns.

By default, this table appears as:

	U	V	W	Alpha	Beta	P	Q	R
FX	'<'	''	''	''	''	''	''	''
FY	''	'<'	''	''	''	''	''	''
FZ	''	''	'<'	''	''	''	''	''
L	''	''	''	''	''	'<'	'<'	''
M	'>'	''	''	'<'	''	''	'<'	''
N	''	''	''	''	'>'	''	''	'<'

Data Types: `string`

`RelativePerturbation` — Relative perturbation

`1e-5` (default) | scalar numeric

Relative perturbation of the system, specified as a scalar numeric. This perturbation takes the form of:

Perturbation Type	Definition
System State perturbation	`statePert = RelativePerturbation +1e-3*RelativePerturbation*|baseValue|`
System input perturbation	`ctrlPert = RelativePerturbation +1e-3*RelativePerturbation*|baseValue|`

To calculate the Jacobian of the system, `linearize` uses the result of these equations in conjunction with the `'DifferentialMethod'` property.

Example: `'RelativePerturbation',1e-5`

Data Types: `double`

`DifferentialMethod` — Direction while perturbing model
`'Forward'` (default) | `'Backward'` | `'Central'`

Direction while perturbing, specified as `'Forward'`, `'Backward'`, or `'Central'`.

Direction	Description
`'Forward'`	Forward difference method that adds `statePert` and `ctrlPert` to the base states and inputs, respectively.
`'Backward'`	Backward difference method that adds `statePert` and `ctrlPert` to the base states an inputs, respectively.
`'Central'`	Central difference method that adds and subtracts `statePert` and `ctrlPert` to and from the base states and inputs, respectively.

Example: `'DifferentialMethod','Backward'`

Data Types: `char` | `string`

`OutputReferenceFrame` — Output reference
`"Body"` (default) | `"Wind"` | `"Stability"`

Output reference of the forces and moments calculation, specified as:

- `"Body"`
- `"Wind"`
- `"Stability"`

Example: `OutputReferenceFrame="Stability"`

Output Arguments

`stability` — Stability of fixed-wing aircraft
6-by-*N* table

Stability of fixed-wing aircraft, returned as a 6-by-*N* table.

`derivatives` — Forces and moments derivatives
6-by-*N* table

Forces and moments derivatives output in `OutputReferenceFrame`, returned as a 6-by-*N* table.

Examples

Calculate Static Stability of Cessna C182

Calculate the static stability of a Cessna C182.

```
[C182, CruiseState] = astC182();
stability = staticStability(C182, CruiseState)
```

```
stability =

  6×8 table

            U           V           W          Alpha        Beta         P           Q          ___
         ________    ________    ________    ________    ________    ________    ________    ___

    FX    "Stable"    ""          ""          ""          ""          ""          ""          ""
    FY    ""          "Stable"    ""          ""          ""          ""          ""          ""
    FZ    ""          ""          "Stable"    ""          ""          ""          ""          ""
    L     ""          ""          ""          ""          "Stable"    "Stable"    ""          ""
    M     "Stable"    ""          ""          "Stable"    ""          ""          "Stable"    ""
    N     ""          ""          ""          ""          "Stable"    ""          ""          "S
```

Calculate Static Stability of Cessna C182 with Custom Criteria Table

Calculate the static stability of a Cessna C182 with a custom criteria table.

```
[C182, CruiseState] = astC182();
CT = C182.criteriaTable()
CT{"FX", "U"} = ">"
 stability = staticStability(C182, CruiseState, "CriteriaTable", CT)
```

```
CT =

  6×8 table

           U      V      W     Alpha    Beta     P      Q      R
          ___    ___    ___    _____    ____    ___    ___    ___

    FX    "<"    ""     ""      ""      ""      ""     ""     ""
    FY    ""     "<"    ""      ""      ""      ""     ""     ""
    FZ    ""     ""     "<"     ""      ""      ""     ""     ""
    L     ""     ""     ""      ""      "<"     "<"    ""     ""
    M     ">"    ""     ""      "<"     ""      ""     "<"    ""
    N     ""     ""     ""      ""      ">"     ""     ""     "<"


CT =

  6×8 table

           U      V      W     Alpha    Beta     P      Q      R
          ___    ___    ___    _____    ____    ___    ___    ___

    FX    ">"    ""     ""      ""      ""      ""     ""     ""
    FY    ""     "<"    ""      ""      ""      ""     ""     ""
    FZ    ""     ""     "<"     ""      ""      ""     ""     ""
    L     ""     ""     ""      ""      "<"     "<"    ""     ""
    M     ">"    ""     ""      "<"     ""      ""     "<"    ""
```

```
    N     ""      ""      ""       ""       ">"      ""      ""      "<"
```

```
stability =

  6×8 table

            U           V           W          Alpha        Beta         P           Q          
        ____________    ________    ________    ________    ________    ________    ________    _

    FX  "Unstable"      ""          ""          ""          ""          ""          ""          "
    FY  ""              "Stable"    ""          ""          ""          ""          ""          "
    FZ  ""              ""          "Stable"    ""          ""          ""          ""          "
    L   ""              ""          ""          ""          "Stable"    "Stable"    ""          "
    M   "Stable"        ""          ""          "Stable"    ""          ""          "Stable"    "
    N   ""              ""          ""          ""          "Stable"    ""          ""          "
```

Calculate Static Stability of Cessna C182 with Central Differential Method

Calculate the static stability of a Cessna C182 using the central differential method.

```
[C182, CruiseState] = astC182();
stability = staticStability(C182, CruiseState, "DifferentialMethod", "Central")
```

```
stability =

  6×8 table

            U           V           W          Alpha        Beta         P           Q          
        ________    ________    ________    ________    ________    ________    ________    ___

    FX  "Stable"    ""          ""          ""          ""          ""          ""          ""
    FY  ""          "Stable"    ""          ""          ""          ""          ""          ""
    FZ  ""          ""          "Stable"    ""          ""          ""          ""          ""
    L   ""          ""          ""          ""          "Stable"    "Stable"    ""          ""
    M   "Stable"    ""          ""          "Stable"    ""          ""          "Stable"    ""
    N   ""          ""          ""          ""          "Stable"    ""          ""          "S
```

Calculate Static Stability and Derivatives of Cessna C182

Calculate the static stability and derivatives of a Cessna C182.

```
[C182, CruiseState] = astC182();
[stability,derivatives] = staticStability(C182, CruiseState)
```

```
stability =

  6×8 table

          U           V           W          Alpha        Beta         P           Q           R
      ________    ________    ________    ________    ________    ________    ________    ________

  FX  "Stable"    ""          ""          ""          ""          ""          ""          ""
  FY  ""          "Stable"    ""          ""          ""          ""          ""          ""
  FZ  ""          ""          "Stable"    ""          ""          ""          ""          ""
  L   ""          ""          ""          ""          "Stable"    "Stable"    ""          ""
  M   "Stable"    ""          ""          "Stable"    ""          ""          "Stable"    ""
  N   ""          ""          ""          ""          "Stable"    ""          ""          "Stable"
```

```
derivatives =

  6×8 table
```

	U	V	W	Alpha	Beta	P	Q	R
FX	-2.118	-5.4001e-08	7.2955	1606.1	-0.0023309	0	0	0
FY	0	-15.415	0	0	-3392.8	-647.47	0	1847.5
FZ	-24.083	-5.9117e-07	-174.03	-38305	-0.026503	0	-33669	0
L	0	-130.33	0	0	-28686	-1.5042e+05	0	24801
M	17.028	4.5475e-07	-105.88	-23303	0.018739	0	-5.2223e+05	0
N	0	83.944	0	0	18476	-8595.5	0	-29248

Version History

Introduced in R2021a

See Also

Aero.FixedWing | criteriaTable | forcesAndMoments | linearize

tdbjuliandate

Convert from Terrestrial time to Barycentric Dynamical Time (TDB) 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 the difference between Barycentric Dynamical Time and Terrestrial Time (TT-TDB) in seconds.

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.

Version History

Introduced in R2015a

References

[1] Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, New York: McGraw-Hill, 1997.

See Also

`dcmeci2ecef` | `ecef2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef` | `lla2eci`

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. Tilt angle values are limited to ±20 degrees. Slip values are limited to ±15 degrees.

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 rate value, specified as any finite and scalar numeric, in degrees. Input the turn rate value as the degrees of tilt of the aircraft symbol in the gauge. The standard rate turn marks are at angles of ±15 degrees. Tilt angle values are limited to ±20 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 `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 rate value indicates the aircraft heading rate of change by the degrees of tilt of the aircraft symbol.
- The slip value controls the direction of the inclinometer ball. A negative value moves the ball to the right, and a positive value moves the ball to the left.

Example: `[15 0]` indicates a coordinated, standard rate turn.

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) | on/off logical value

Visibility of the turn coordinator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`. 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.

`ContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object created using the `uicontextmenu` function. Use this property to display a context menu when you right-click on a component.

`Enable` — Operational state of turn coordinator

`'on'` (default) | on/off logical value

Operational state of turn coordinator, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- 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 “Callbacks in App Designer”.

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 “Callbacks in App Designer”.

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) | on/off logical value

Callback interruption, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

This 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.

MATLAB determines callback interruption behavior whenever it executes a command that processes the callback queue. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines if the interruption occurs:

- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the object that owns the interrupting callback determines if the interrupting callback is discarded or added to the callback queue.
- If the value of `Interruptible` is `'on'`, then the interruption occurs. The next time MATLAB processes the callback queue, it stops the execution of the running callback and executes the interrupting callback. After the interrupting callback completes, MATLAB then resumes executing the running callback.

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.
- If the interrupting callback is owned by a `Timer` object, then the callback executes according to schedule regardless of the `Interruptible` property value.

Note 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.

The `BusyAction` property determines callback queuing behavior only when both of these conditions are met:

- The running callback contains a command that processes the callback queue, such as `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause`.
- The value of the `Interruptible` property of the object that owns the running callback is `'off'`.

Under these conditions, the `BusyAction` property of the object that owns the interrupting callback determines how MATLAB handles the interrupting callback. 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

on/off logical value

This property is read-only.

Deletion status, returned as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

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
`'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”.

Version History

Introduced in R2018b

See Also

`uiaeroturn`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51

“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

uiaeroairspeed

Package: `Aero.ui.control`

Create airspeed indicator component

Syntax

```
airspeed = uiaeroairspeed
airspeed = uiaeroairspeed(parent)
airspeed = 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.

If the value of the input is under **Minimum**, the needle displays 5 degrees under the **Minimum** value. If the value exceeds the **Maximum** value, the needle displays 5 degrees over the maximum tick.

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.

`airspeed = uiaeroairspeed(parent)` specifies the object in which to create the airspeed indicator.

`airspeed = 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.

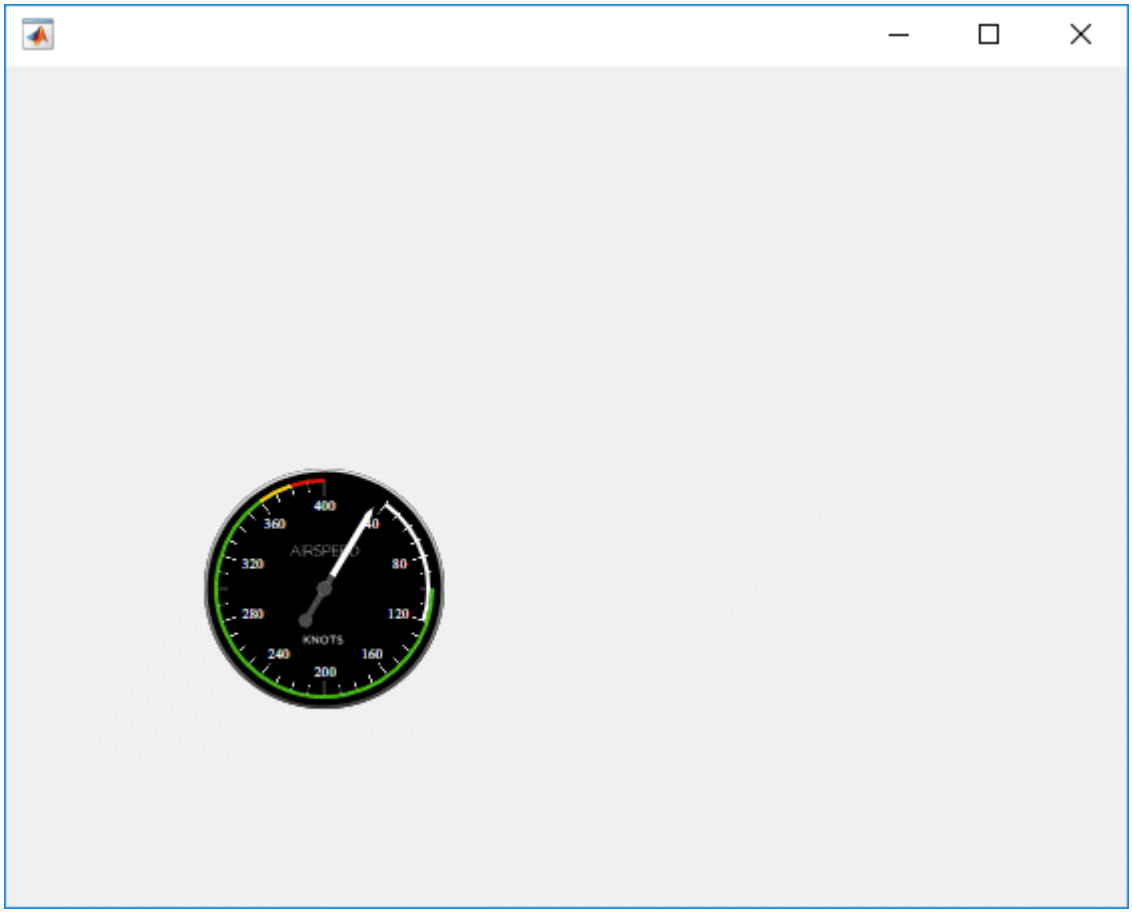
Examples

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) | `Tab` object | `Panel` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

AirspeedIndicator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

“Compute Indicated Airspeed for Pitot-Static Airspeed Indicator” on page 5-221
“Differences Between Corrected Airspeeds” on page 5-224

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 2000 feet and a major tick represents 10,000 feet.
  - For the second longest needle, a minor tick represents 20 feet and a major tick represents 100 feet.
  - For the shortest needle a minor tick represents 200 feet and a major tick represents 1,000 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 are 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.

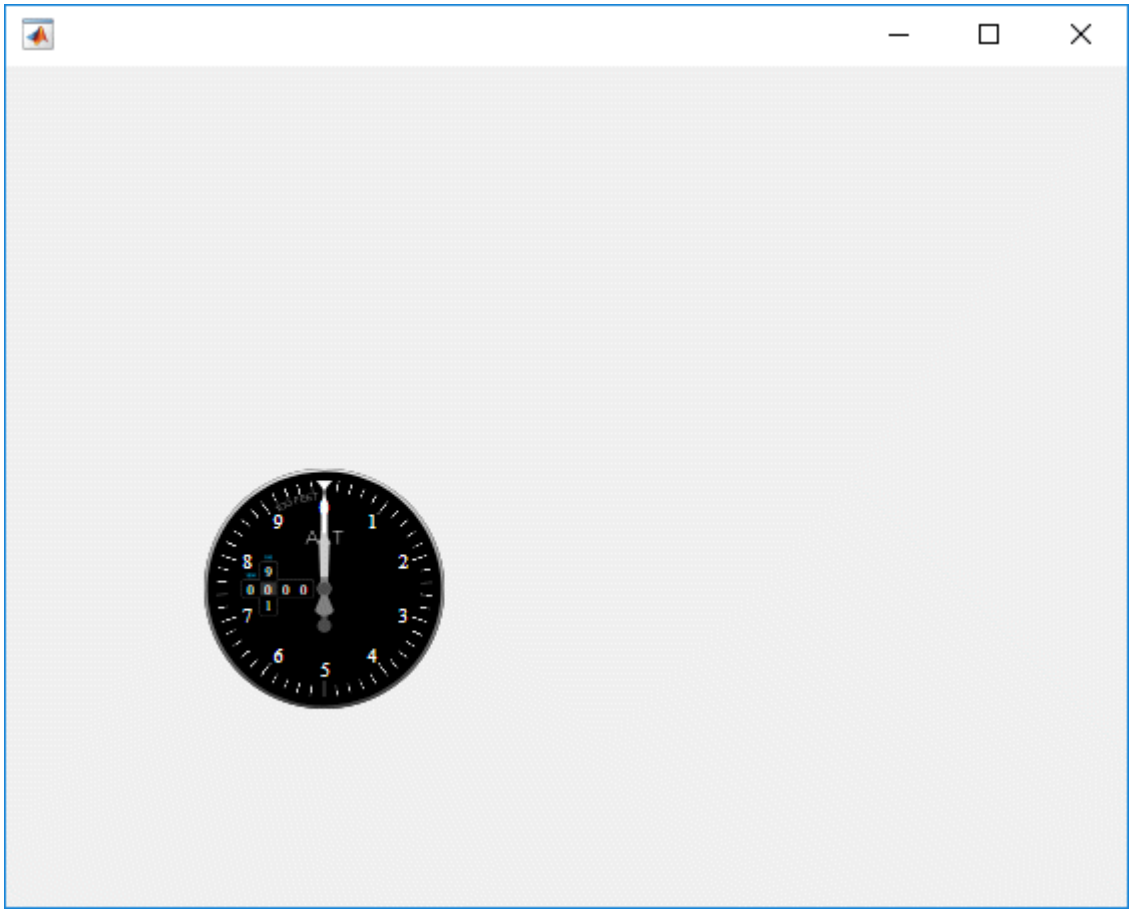
Examples

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) | `Tab` object | `Panel` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

Altimeter Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

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

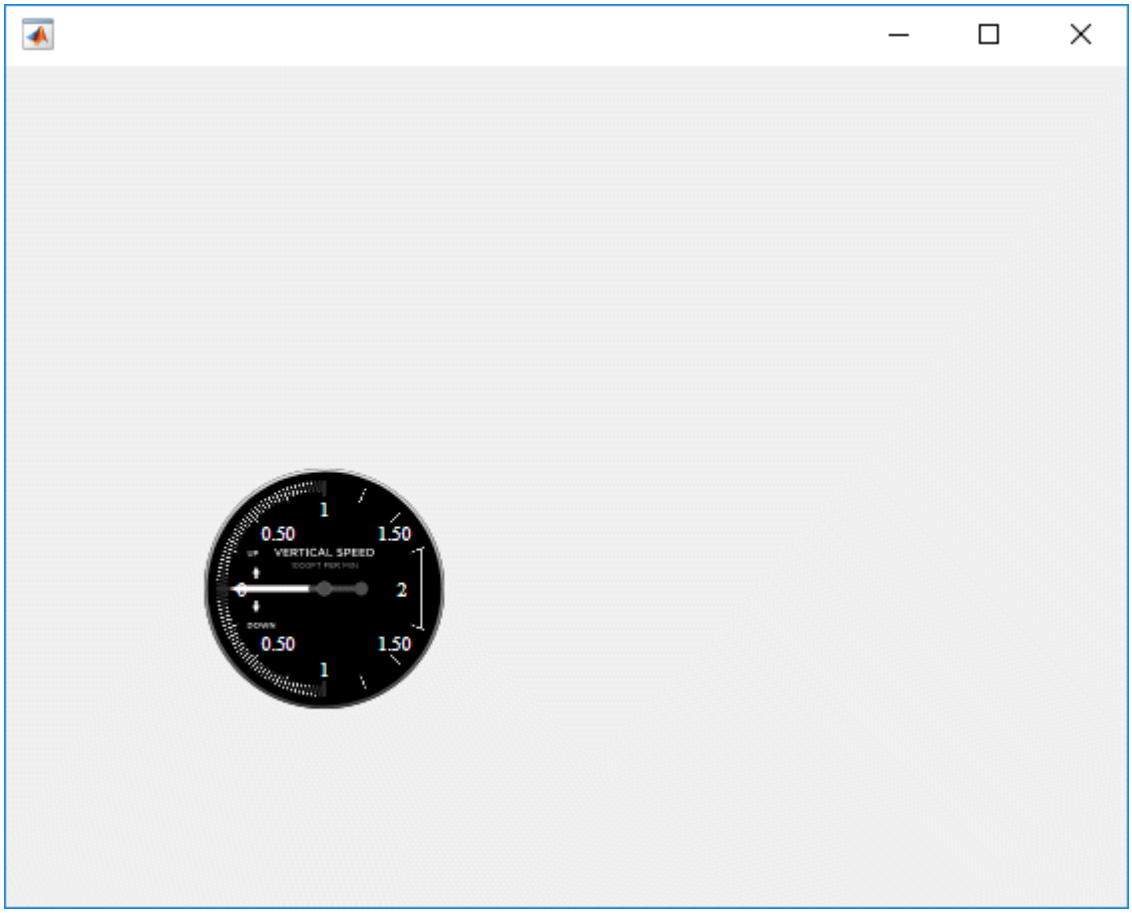
Create Climb Rate Indicator Component

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) | `Tab` object | `Panel` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

ClimbIndicator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

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 input 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.

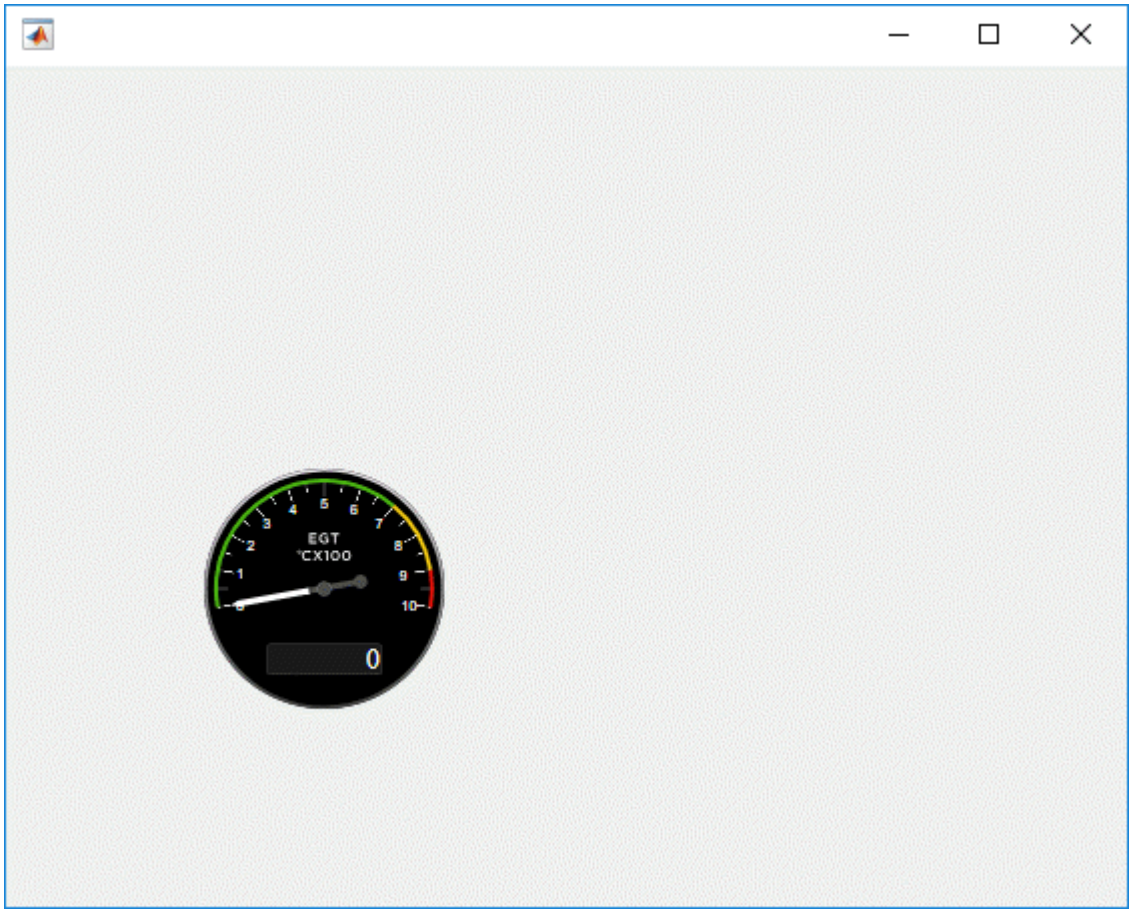
Examples

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) | `Tab` object | `Panel` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

EGTIndicator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

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.

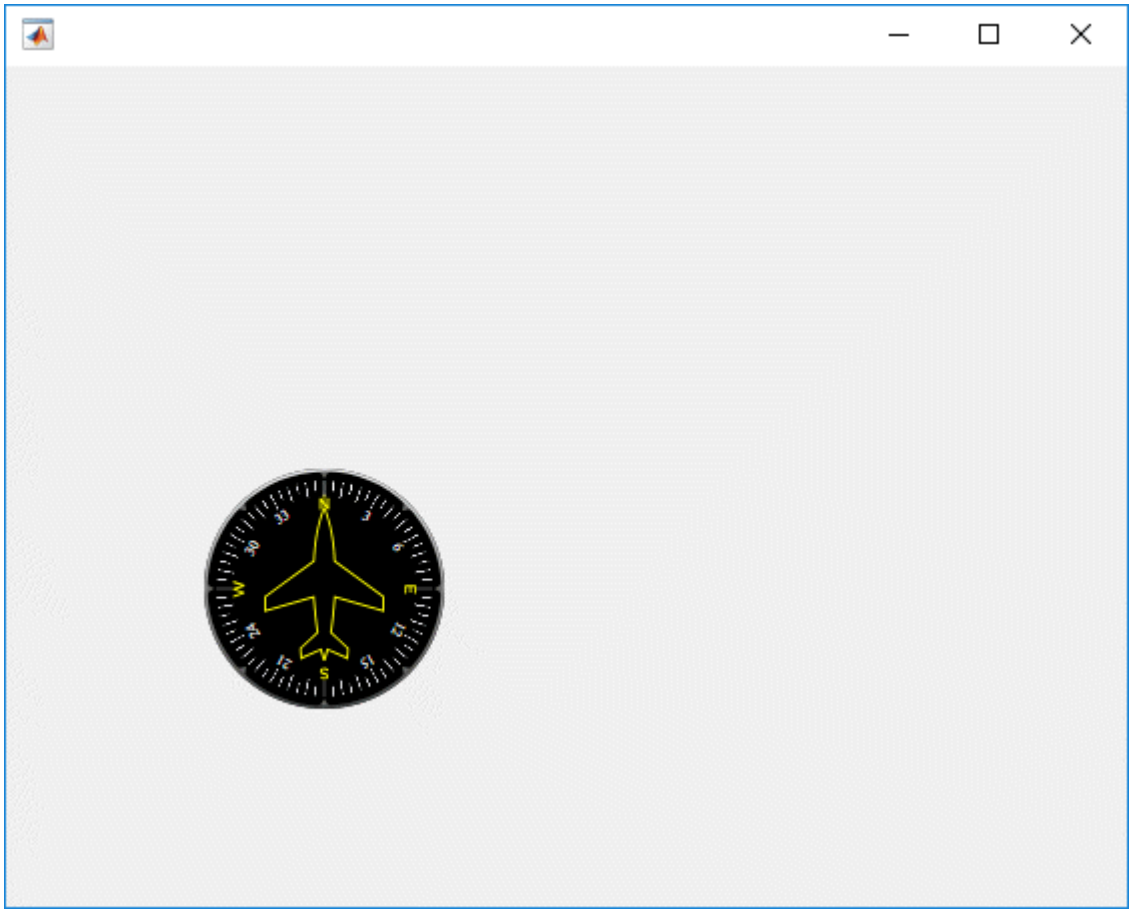
Examples

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) | `Tab` object | `Panel` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

HeadingIndicator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

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

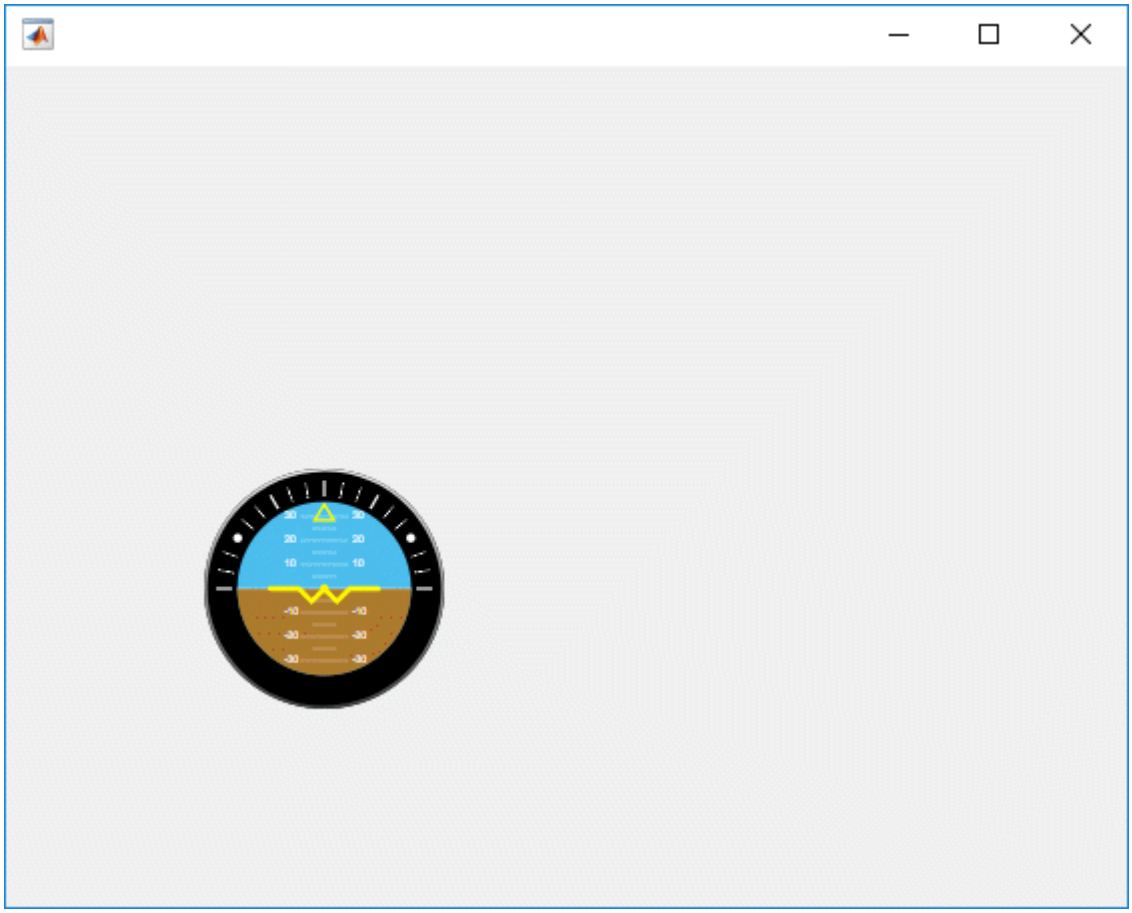
Create Artificial Horizon Component

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) | `Tab` object | `Panel` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

ArtificialHorizon Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

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.

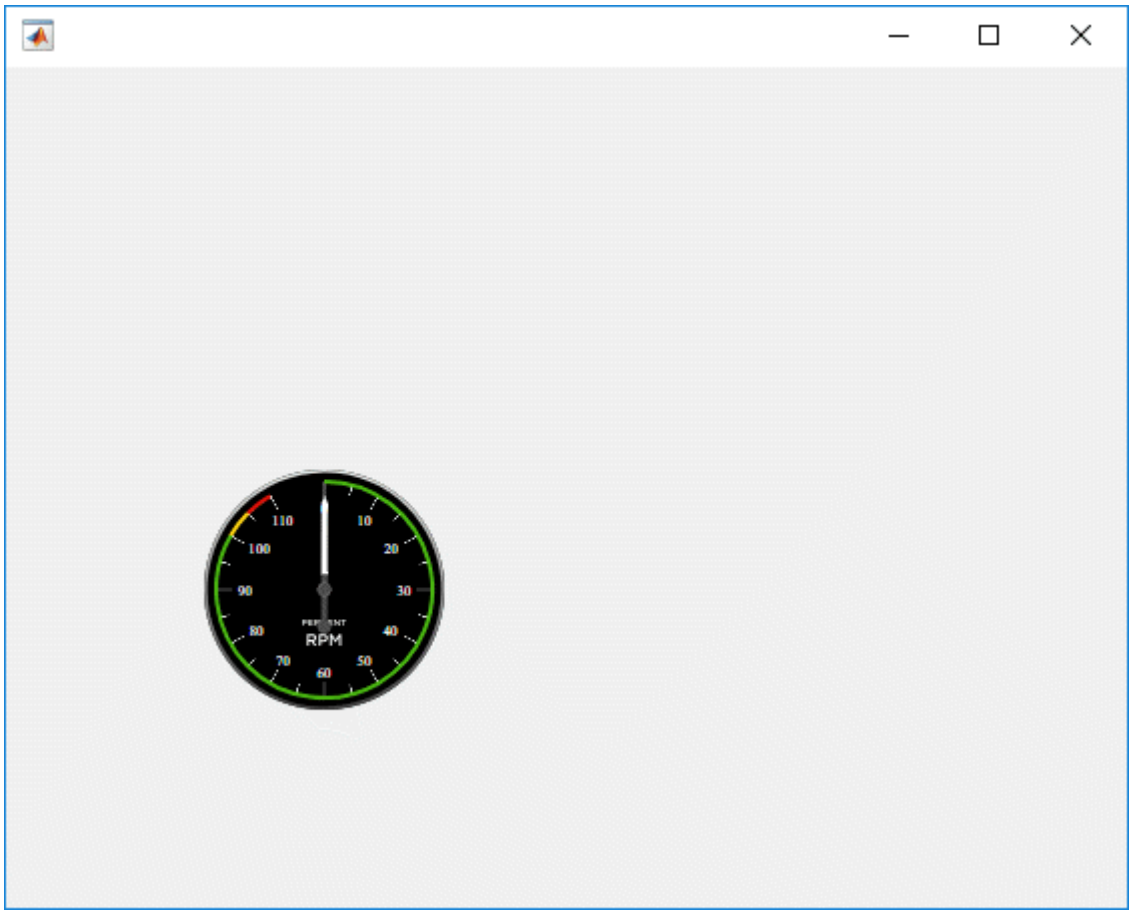
Examples

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) | `Tab` object | `Panel` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

RPMIndicator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

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 `uiaeroturn` function displays measurements on a gyroscopic turn rate instrument and on an inclinometer.

- The gyroscopic turn rate instrument shows the rate of heading change of the aircraft as a tilting of the aircraft symbol in the gauge.
- The inclinometer shows whether the turn is coordinated, slipping, or skidding by the position of the ball.

When the ball is centered, the turn is coordinated. When the ball is off center, the turn is slipping or skidding. The turn rate instrument has marks for wings level and for a standard rate turn. A standard rate turn is a heading change of 3 degrees per second, also known as a two minute turn.

The input for gyroscopic turn rate instruments and inclinometers is in degrees. The turn rate value is input as the degrees of tilt of the aircraft symbol in the gauge. The standard rate turn marks are at angles of ±15 degrees. Tilt angle values are limited to ±20 degrees, whereas inclinometer angles are limited to ±15 degrees.

For example, turn indicator and inclinometer values of `[15 0]` indicate a coordinated, standard rate turn.

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.

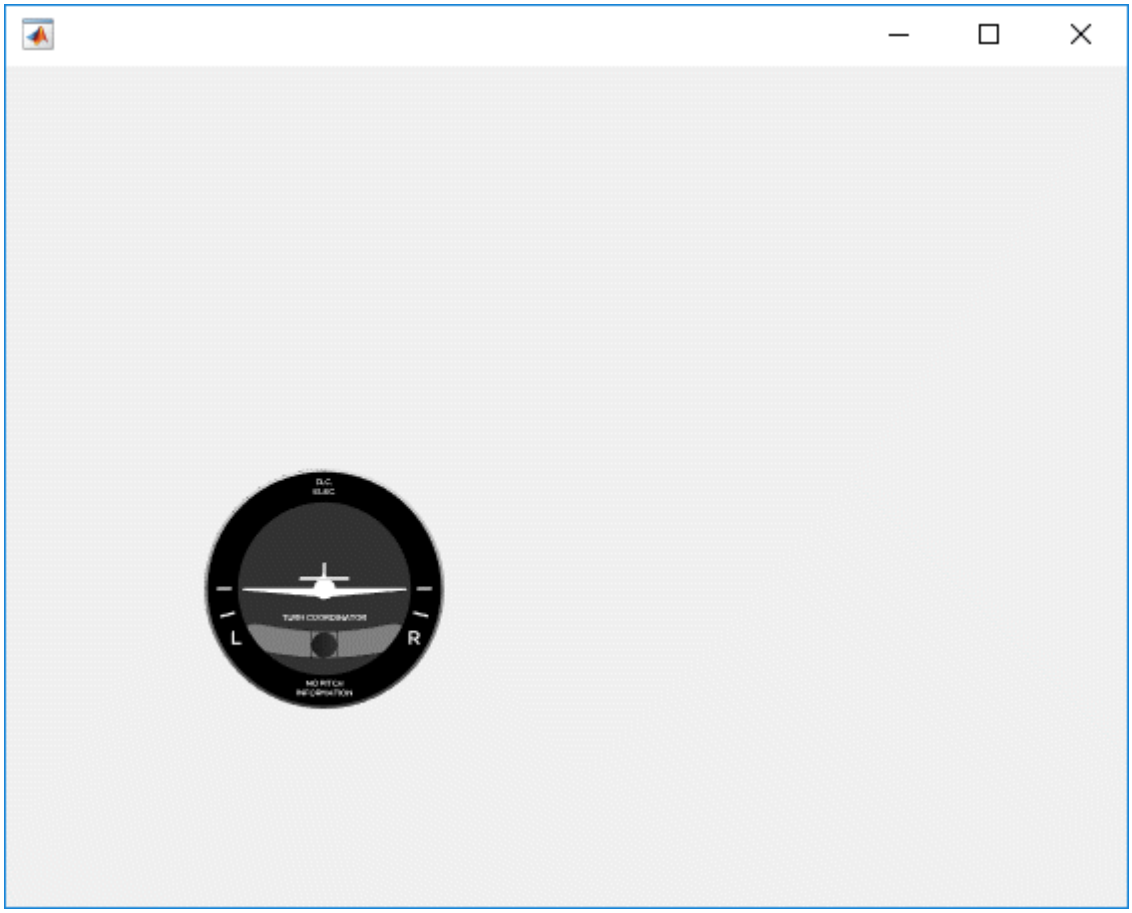
Examples

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) | `Tab` object | `Panel` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

TurnCoordinator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

update

Class: `Aero.FixedWing`
Package: `Aero`

Update `Aero.FixedWing` object

Syntax

```
aircraft = update(aircraft)
aircraft = update(aircraft,Rename,Value)
```

Description

`aircraft = update(aircraft)` returns the modified coefficient `Aero.FixedWing` object.

`aircraft = update(aircraft,Rename,Value)` updates the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object of each `Simulink.LookupTable` coefficient in the `Aero.FixedWing` object hierarchy. The updated name is a compilation of all component `Name` values in the `Aero.FixedWing` hierarchy, with this format:

- Listed in descending order
- Separated by underscores (_)
- Appended by the `stateOutput` and `stateVariable` values of each `Simulink.LookupTable` location

Input Arguments

`aircraft` — `Aero.FixedWing` coefficient object

scalar | `Aero.FixedWing` | `Aero.FixedWing.Surface` | `Aero.FixedWing.Control` | `Aero.FixedWing.Thrust` | `Aero.FixedWing.Coefficient`

`Aero.FixedWing` coefficient object, specified as a scalar, of type `Aero.FixedWing`, `Aero.FixedWing.Surface`, `Aero.FixedWing.Control`, `Aero.FixedWing.Thrust`, or `Aero.FixedWing.Coefficient`.

`Rename,Value` — Option to update `Name` property in `Simulink.lookuptable.StructTypeInfo` object

`on` (default) | `off`

Option to update the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object, specified as:

- `'on'` — To modify the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object.
- `'off'` — Do not modify the `Name` field in the `Simulink.lookuptable.StructTypeInfo` object.

Example: `'Rename','on'`

Data Types: `string` | `char`

Output Arguments

`aircraft` — Modified `Aero.FixedWing` object

scalar

Modified `Aero.FixedWing` object with the modified coefficients at the specified locations, returned as a scalar.

Examples

Update Aircraft Name and View Updated Coefficients

Update the aircraft name and view the updated coefficients.

```
aircraft = astSkyHogg;
aircraft.Properties.Name = 'NewName';
aircraft = update(aircraft);
aircraft.Coefficients.Values{1}.StructTypeInfo.Name

ans =

    'NewName_CD_Zero'
```

Update Aircraft Name But Do Not Propagate

Update the aircraft name, but do not propagate the new name to the coefficients.

```
aircraft = astSkyHogg;
aircraft.Properties.Name = 'NewName';
aircraft = update(aircraft, 'Rename', 'off');
aircraft.Coefficients.Values{1}.StructTypeInfo.Name

ans =

    'SkyHogg_CD_Zero'
```

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `getCoefficient` | `setCoefficient` | `Simulink.LookupTable` | `Simulink.lookuptable.StructTypeInfo`

Topics

“Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121
“Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106
“Customize Fixed-Wing Aircraft with Additional Aircraft States” on page 5-113

update

Class: `Aero.FixedWing.Coefficient`
Package: `Aero`

Update `Aero.FixedWing.Coefficient` object

Syntax

```
FixedWingCoefficient = update(FixedWingCoefficient,Rename,Value)
```

Description

`FixedWingCoefficient = update(FixedWingCoefficient,Rename,Value)` updates the `Aero.FixedWing.Coefficient` object.

Input Arguments

`FixedWingCoefficient` — `Aero.FixedWingCoefficient` object
scalar

`Aero.FixedWing.Coefficient` object, specified as a scalar.

`Rename,Value` — Option to update `Name` property in `Simulink.lookuptable.StructTypeInfo` object
`on` (default) | `off`

Option to update the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object, specified as:

- `'on'` — Modify the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object.

  This method sets the `Name` property in the `Simulink.lookuptable.StructTypeInfo` objects to `name_stateOutput_stateVariable`, where:

  - `name` is the combined string from the component name joined with all component names above it.
  - `stateOutput` and `stateVariable` are the `stateOutput` and `stateVariable` values from each specific `Simulink.LookupTable` location, respectively.
- `'off'` — Do not modify the `Name` field in the `Simulink.lookuptable.StructTypeInfo` object.

Example: `'Rename','on'`

Data Types: `string` | `char`

Output Arguments

`FixedWingCoefficient` — Modified `Aero.FixedWing.Coefficient` object
scalar

Modified `Aero.FixedWing.Coefficient` object with modified coefficients at the specified locations, specified as a scalar.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Simulink.lookuptable.StructTypeInfo`

update

Class: `Aero.FixedWing.Surface`
Package: `Aero`

Update `Aero.FixedWing.Surface` object

Syntax

```
FixedWingSurface = update(FixedWingSurface,Rename,Value)
```

Description

`FixedWingSurface = update(FixedWingSurface,Rename,Value)` updates the surface object `FixedWingSurface`.

Input Arguments

`FixedWingSurface` — `Aero.FixedWingSurface` object
scalar

`Aero.FixedWingSurface` surface object, specified as a scalar.

`Rename,Value` — Option to update `Name` property in `Simulink.lookuptable.StructTypeInfo` object
`on` (default) | `off`

Option to update the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object, specified as:

- `'on'` — Modify the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object.

  The method sets the `Name` property in the `Simulink.lookuptable.StructTypeInfo` objects to `name_stateOutput_stateVariable`, where:

  - `name` is the combined string from the component name joined with all component names above it.
  - `stateOutput` and `stateVariable` are the `stateOutput` and `stateVariable` values from each specific `Simulink.LookupTable` location, respectively.
- `'off'` — Do not modify the `Name` field in the `Simulink.lookuptable.StructTypeInfo` object.

Example: `'Rename','on'`

Data Types: `string` | `char`

Output Arguments

`FixedWingSurface` — Modified `Aero.FixedWing.Surface` object
scalar

Modified `Aero.FixedWing.Surface` object with the modified coefficients at the specified locations, returned as a scalar.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Simulink.lookuptable.StructTypeInfo`

update

Class: `Aero.FixedWing.Thrust`
Package: `Aero`

Update `Aero.FixedWing.Thrust` object

Syntax

```
FixedWingThrust = update(FixedWingThrust,Rename,Value)
```

Description

`FixedWingThrust = update(FixedWingThrust,Rename,Value)` updates the `Aero.FixedWingThrust` object.

Input Arguments

`FixedWingThrust` — `Aero.FixedWingThrust` object

`scalar`

`Aero.FixedWingThrust` thrust object, specified as a scalar.

`Rename,Value` — Option to update `Name` property in `Simulink.lookuptable.StructTypeInfo` object

`on` (default) | `off`

Option to update the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object, specified as:

- `'on'` — Modify the `Name` property in the `Simulink.lookuptable.StructTypeInfo` object.

  The method sets the `Name` property in the `Simulink.lookuptable.StructTypeInfo` objects to `name_stateOutput_stateVariable`, where:

  - `name` is the combined string from the component name joined with all component names above it.
  - `stateOutput` and `stateVariable` are the `stateOutput` and `stateVariable` values from each specific `Simulink.LookupTable` location, respectively.
- `'off'` — Do not modify the `Name` field in the `Simulink.lookuptable.StructTypeInfo` object.

Example: `'Rename','on'`

Data Types: `string` | `char`

Output Arguments

`FixedWingThrust` — Modified `Aero.FixedWing.Thrust` object

`Aero.FixedWing.Thrust`

Modified `Aero.FixedWingThrust` object with the modified coefficients at the specified locations.

Version History

Introduced in R2021a

See Also

`Aero.FixedWing` | `Simulink.lookuptable.StructTypeInfo`

update

Class: `Aero.Body`
Package: `Aero`

Update body position and orientation as function of time

Syntax

```
update(h,t)
```

Description

`update(h,t)` updates the body position and orientation of body `h` as a function of time `t`.

Input Arguments

`h` — Aerospace body object

`Aero.Body` object

Aerospace body object, specified as an `Aero.Body` object.

`t` — Time

scalar

Time, specified as a scalar, in seconds.

Data Types: `double`

Examples

Update Body with Time

Update the body `b` with a time of 5 seconds.

```
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);
```

Version History

Introduced in R2007a

See Also

Classes
`Aero.Animation` | `Aero.Body`

Functions
`load`

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`.

Version History

Introduced in R2007a

See Also

`play`

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);
```

Version History

Introduced in R2007a

See Also

`GenerateRunScript` | `initialize` | `play`

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.

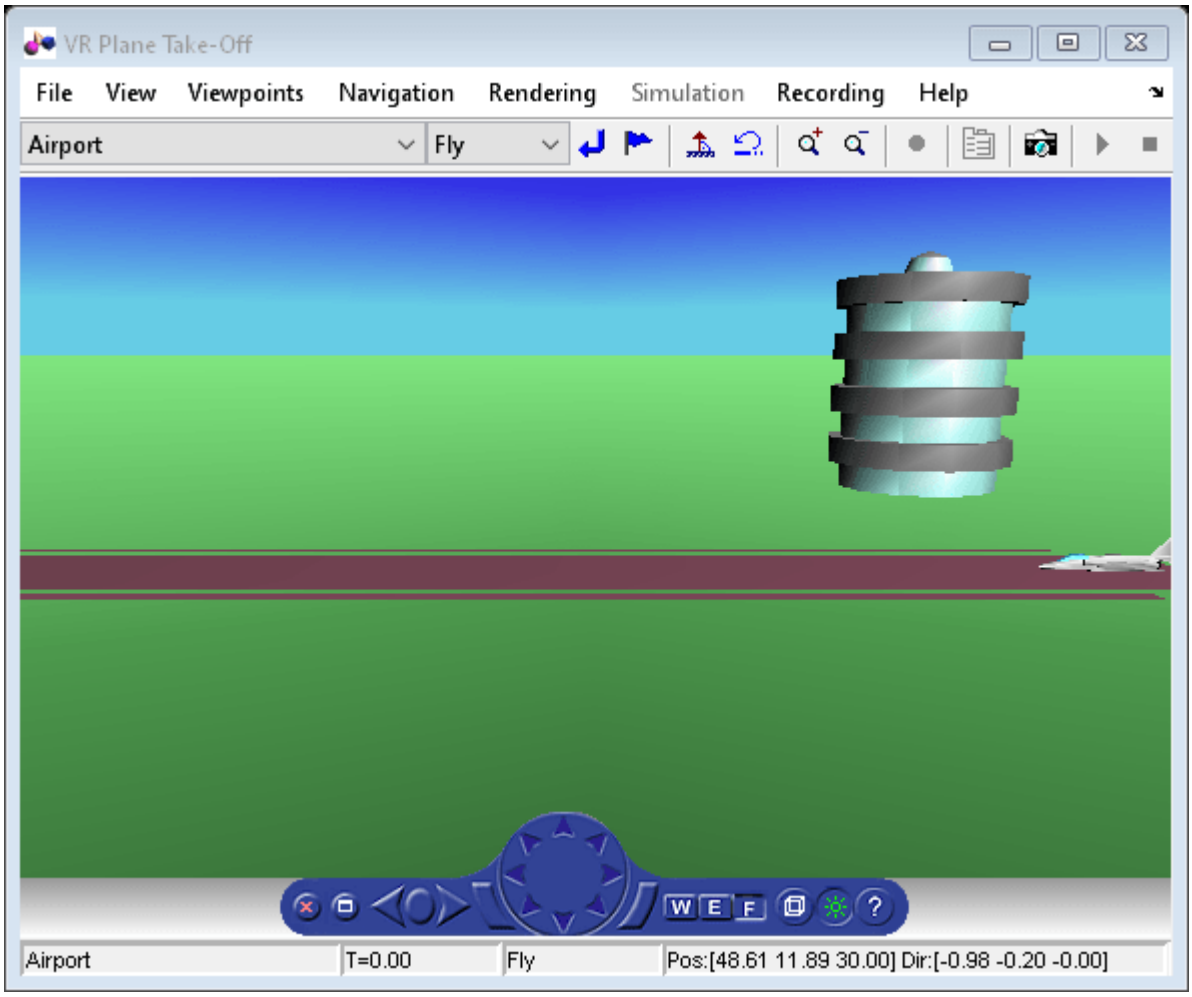
Note This function requires that you load the node and time series data first.

Examples

Change Node Position

Change the node position as a function of time.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
```




```
load takeoffData
h.Nodes{7}.TimeSeriesSource = takeoffData;
h.Nodes{7}.TimeSeriesSourceType = 'StructureWithTime';
h.Nodes{7}.update(5);
```

Version History

Introduced in R2007b

See Also

`updateNodes`

updateBodies

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

Update bodies of animation object

Syntax

```
updateBodies(h,time)
```

Description

`updateBodies(h,time)` sets the new position and attitude of movable bodies in the animation object `h`. This function updates the bodies contained in the animation object `h`.

Input Arguments

`h` — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

`time` — Animation time

scalar

Animation time, specified as a scalar, in seconds.

Data Types: `double`

Examples

Update `Aero.Animation` Body

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 = createBody(h,'pa24-250_orange.ac','Ac3d');
load simdata;
h.Bodies{1}.TimeSeriesSource = simdata;
t = 0;
updateBodies(h,t);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

updateCamera

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

Update camera in animation object

Syntax

```
updateCamera(h,time)
```

Description

`updateCamera(h,time)` updates the camera in the animation object `h`.

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. So the camera requires a few steps to position itself correctly in the chase plane position.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

time — Animation time

scalar

Animation time, specified as a scalar, in seconds.

Data Types: `double`

Examples

Update Camera in `Aero.Animation` Object

Configure a body with the `TimeSeriesSource` property set to `simdata.` Then, update the camera with time `t` equal to 0.

```
h = Aero.Animation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
idx1 = createBody(h,'pa24-250_orange.ac','Ac3d');
load simdata;
h.Bodies{1}.TimeSeriesSource = simdata;
t = 0;
updateCamera(h,t);
```

Version History

Introduced in R2007a

See Also

`Aero.Animation`

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.

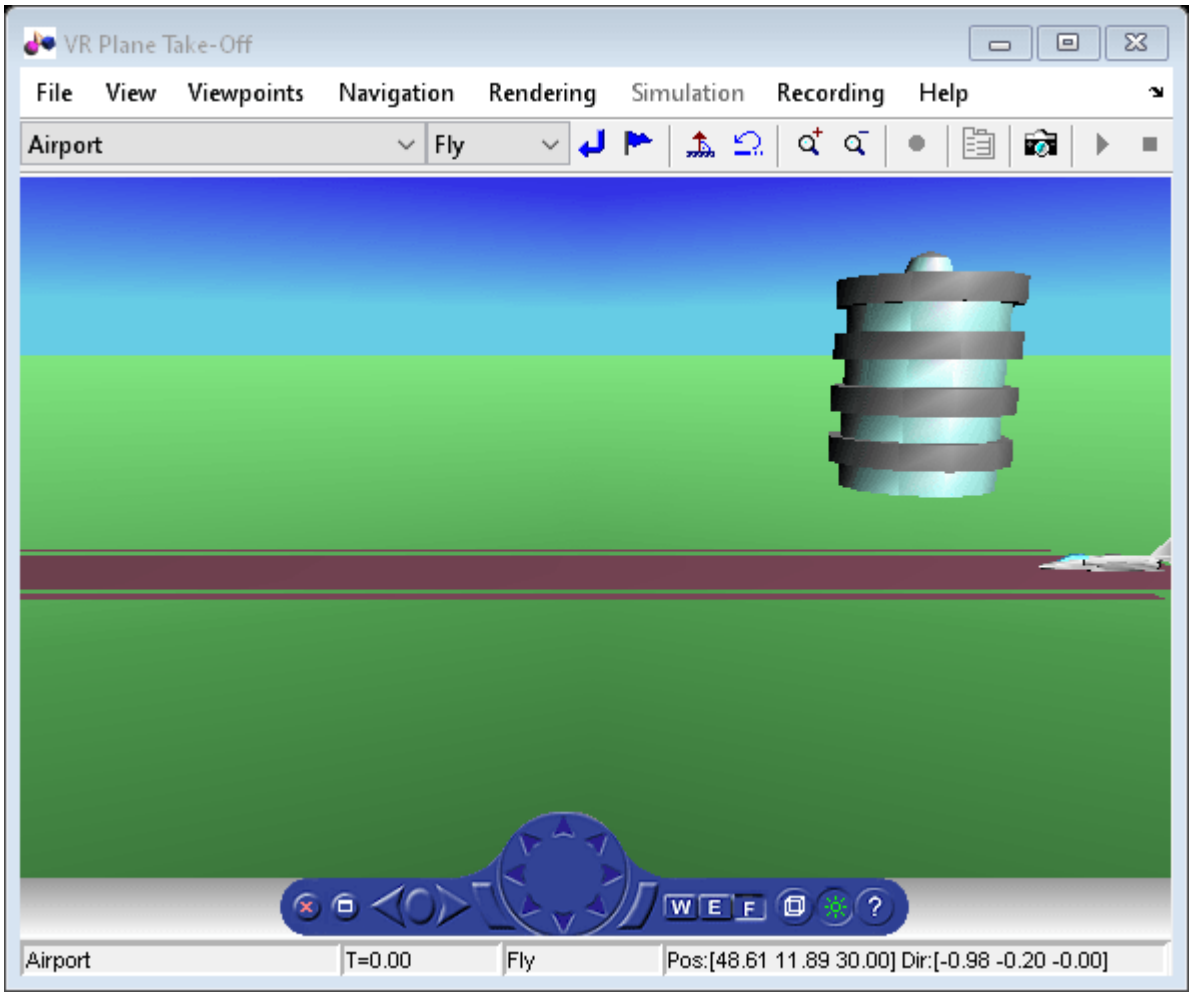
Note This function requires that you load the node and time series data first.

Examples

Update Node with Time

This example shows how to update node `h` with time in 5 seconds.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = 'asttkoff.wrl';
h.initialize();
```




```
load takeoffData
h.Nodes{7}.TimeSeriesSource = takeoffData;
h.Nodes{7}.TimeSeriesSourceType = 'StructureWithTime';
h.Nodes{7}.CoordTransformFcn = @vranimCustomTransform;
h.updateNodes(2);
```

Version History

Introduced in R2007b

See Also

addNode | update

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.

Version History

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.

Version History

Introduced in R2007b

See Also

`Aero.VirtualRealityAnimation`

wait

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

Wait until animation stops playing

Syntax

```
wait(h)
```

Description

`wait(h)` suspends execution until the animation of animation object `h` finishes. MATLAB is blocked until the animation stops.

Input Arguments

h — Aerospace animation object

`Aero.Animation` object

Aerospace animation object, specified as an `Aero.Animation` object.

Examples

Wait for Animation to Stop

```
h = Aero.Animation;
wait(h);
```

Version History

Introduced in R2022a

See Also

`Aero.Animation`

wait (Aero.FlightGearAnimation)

Wait until animation is done playing

Syntax

```
wait(h)
h.wait
```

Description

`wait(h)` and `h.wait` wait until the animation is done playing. MATLAB is blocked until the animation stops.

Examples

Wait until animation of FlightGear animation object, `h` stops running.

```
h = Aero.FlightGearAnimation;
h.wait;
```

Version History

Introduced in R2022a

See Also

`Aero.FlightGearAnimation`

wait (Aero.VirtualRealityAnimation)

Wait until animation is done playing

Syntax

```
wait(h)
h.wait
```

Description

`wait(h)` and `h.wait` wait until the animation is done playing. MATLAB is blocked until the animation stops.

Examples

Wait until animation of `VirtualRealityAnimation` animation object, `h` stops running.

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

Version History

Introduced in R2022a

See Also

`Aero.VirtualRealityAnimation`

walkerDelta

Create Walker-Delta constellation in satellite scenario

Syntax

```
sat = walkerDelta(scenario,radius,inclination,totalSatellites,geometryPlanes,
phasing)
sat = walkerDelta(__,Name=Value)
```

Description

`sat = walkerDelta(scenario,radius,inclination,totalSatellites,geometryPlanes, phasing)` creates an array of satellites, `sat`, inside the satellite scenario `scenario` using specified geometric properties such as radius, `radius`, inclination, `inclination`, total number of satellites, `totalSatellites`, number of geometry planes, `geometryPlanes`, and phasing between satellites, `phasing`. The ascending nodes of the orbital planes are evenly spaced over 360 degrees.

For more information on Walker-Delta constellations, see “Algorithms” on page 4-999.

`sat = walkerDelta(__,Name=Value)` creates an array of satellites using one or more optional `Name=Value` arguments. Use this option with any of the input argument combinations in the previous syntax.

Examples

Model Galileo Constellation as Walker-Delta Constellation

Model a Galileo constellation as a Walker-Delta constellation that contains 24 satellites in three planes inclined at 56 degrees (56:24/3/1) in a 29599.8 km orbit. Provide an initial argument of latitude offset of 15 degrees.

Create a default satellite scenario object.

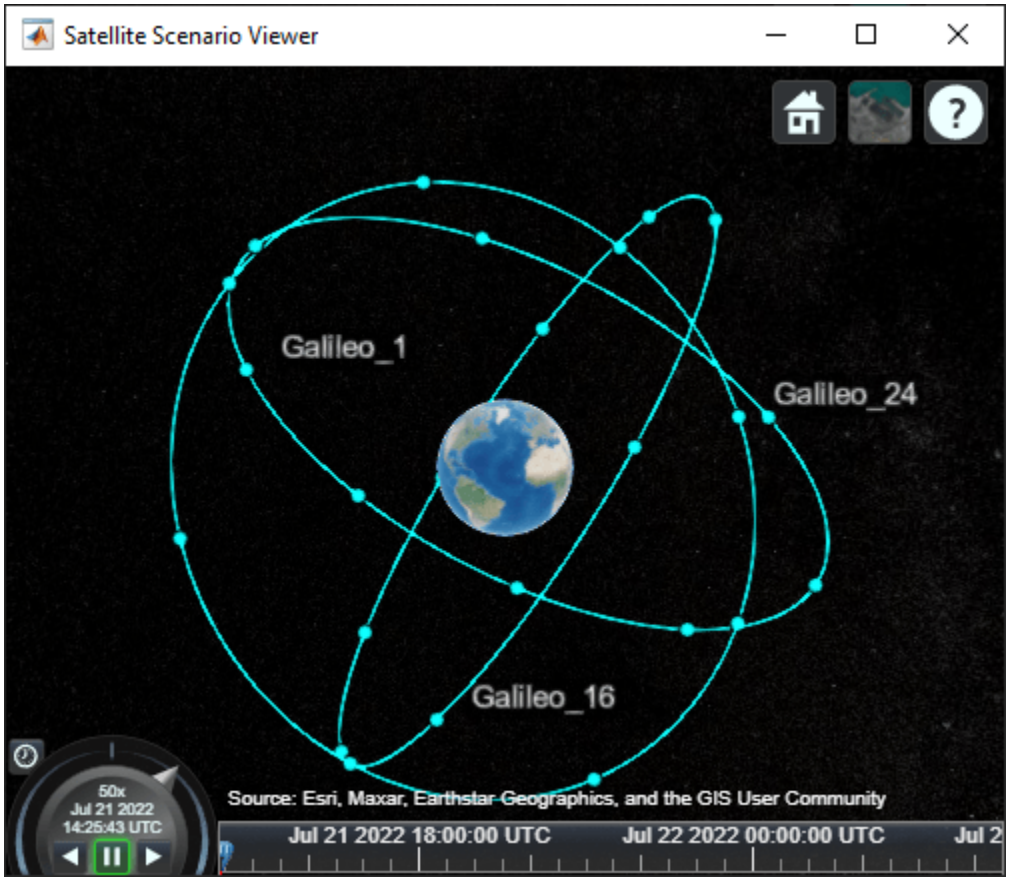
```
sc = satelliteScenario;
```

Create Walker-Delta constellation that contains 24 satellites in three planes inclined at 56 degrees (56:24/3/1) in a 29599.8 kilometer orbit.

```
sat = walkerDelta(sc, 29599.8e3, 56, 24, 3, 1, ...
       ArgumentOfLatitude=15, Name="Galileo");
```

Visualize the scenario using the Satellite Scenario Viewer. Hide the orbits and labels of the satellites by setting the `ShowDetails` name-value pair to `false`. Mouse over satellites to show their labels. Click on a satellite to reveal its label, orbit, and any other hidden graphics. Click on the satellite again to dismiss them.

```
satelliteScenarioViewer(sc, ShowDetails=false);
```



Input Arguments

`scenario` — Satellite scenario
`satelliteScenario` object

Satellite scenario, specified as a `satelliteScenario` object.

`radius` — Orbital radius
scalar

Orbital radius, specified as a scalar, in meters.

Data Types: `double`

`inclination` — Inclination
scalar

Inclination, specified as a scalar, in degrees.

Data Types: `double`

`totalSatellites` — Total number of satellites
scalar positive integer

Total number of satellites, specified as a scalar positive integer.

Data Types: `double`

`geometryPlanes` — Number of equally spaced geometry planes
scalar positive integer

Number of equally spaced geometry planes, specified as a scalar positive integer.

Data Types: `double`

`phasing` — Phasing between satellites
scalar integer greater than or equal to 0 and less than `geometryPlanes`

Phasing between satellites in adjacent planes, specified as a scalar integer greater than or equal to 0 and less than `geometryPlanes`. The change in true anomaly for equivalent satellites in neighboring planes is calculated as:

`(phasing*360/totalSatellites).`

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `sat = walkerDelta(sc,29599.8e3,56, 24,3,1,Name="Galileo")` creates an array of satellites named Galileo.

`RAAN` — Right ascension of ascending node
0 (default) | scalar value from 0 to 360

Right ascension of ascending node, specified as a scalar value from 0 to 360. RAAN is the angle in the equatorial plane from the *x*-axis to the location of the ascending node, point at which the satellite crosses the equator from south to north, in degrees. The function uses this value as the starting point for distribution of the satellites along the equator.

Data Types: `double`

`ArgumentOfLatitude` — Argument of latitude
0 (default) | scalar value from 0 to 360

Argument of latitude, specified as a scalar value from 0 to 360. `ArgumentOfLatitude` is the angle between the ascending node and the body. The function uses this value as the starting point for distribution of satellites along the first orbital track.

Data Types: `double`

`Name` — Constellation name
`WalkerDelta` (default) | scalar

Constellation name, specified as a scalar.

Individual satellite names within the constellation use the constellation name appended with increasing whole numbers starting with 1, for example, `Name_1`, `Name_2`, and so forth.

Data Types: `char` | `string`

`OrbitPropagator` — Name of orbit propagator
`"two-body-keplerian"` (default) | `"sgp4"` | `"sdp4"`

Name of orbit propagator used for propagating satellite position and velocity, specified as:

- `"two-body-keplerian"` — Two-Body-Keplerian orbit propagator based on the relative two-body model that assumes a spherical gravity field for the Earth and neglects third body effects and other environmental perturbations. It is considered the least accurate.
- `"sgp4"` — Simplified General Perturbations-4 orbit propagator.
- `"sdp4"` — Simplified Deep-Space Perturbations-4 orbit propagator.

Data Types: `string` | `char`

Output Arguments

`sat` — Satellites in scenario
array of `Satellite` objects

Satellite in scenario, returned as an array of `Satellite` objects.

Algorithms

Walker-Delta constellations are a common solution for maximizing geometric coverage over Earth while minimizing the number of satellites required to perform the mission. Walker-Delta constellation patterns use the notation:

`I:T/P/F`.

where:

- `I` — Orbital inclination
- `T` — Total number of satellites, which must be divisible by `F`
- `P` — Number of equally spaced geometric planes
- `F` — Phasing between satellites in adjacent planes

To define the radial height of the circular orbit (with respect to the Earth center), the function also requires a radius `r`.

In addition:

- The ascending nodes of the orbital planes of a Walker-Delta constellation are uniformly distributed at intervals of 360/`P` deg around the equator.
- The number of satellites per plane, `satellitesPerPlane`, is defined as

  `satellitesPerPlane=T/P`.

  The satellites in each orbital plane are distributed at intervals of 360/`satellitesPerPlane` deg. `F` represents the interplane phasing, the number of empty slots between the first satellites in each orbital plane.

Note Walker-Delta constellations differ from Walker-Star constellations. For Walker-Delta constellations, the function distributes ascending nodes of the orbital planes across 360 degrees. For Walker-Star constellations, the function distributes ascending nodes across 180 degrees.

Version History

Introduced in R2022a

See Also

`satelliteScenario` | `satellite` | `access` | `walkerStar`

Topics

“Analyze Access Between a Satellite Constellation and an Aircraft” on page 5-228

walkerStar

Create Walker-Star constellation in satellite scenario

Syntax

```
sat = walkerStar(scenario,radius,inclination,totalSatellites,geometryPlanes,
phasing)
sat = walkerStar(__,Name=Value)
```

Description

`sat = walkerStar(scenario,radius,inclination,totalSatellites,geometryPlanes, phasing)` creates an array of satellites, `sat`, inside the satellite scenario `scenario` using specified geometric properties such as radius, `radius`, inclination, `inclination`, total number of satellites, `totalSatellites`, number of geometry planes, `geometryPlanes`, and phasing between satellites, `phasing`. The ascending nodes of the orbital planes are evenly spaced over 180 degrees.

For more information on Walker-Star constellations, see “Algorithms” on page 4-1004.

`sat = walkerStar(__,Name=Value)` creates an array of satellites using one or more optional `Name=Value` arguments. Use this option with any of the input argument combinations in the previous syntax.

Examples

Model Iridium-Like Constellation as Walker-Star Constellation

Model a constellation similar to Iridium as a Walker-Star constellation containing 66 satellites in six planes inclined at 86.4 degrees (86.4:66/6/2) in a 781 km orbit.

Create a default satellite scenario object.

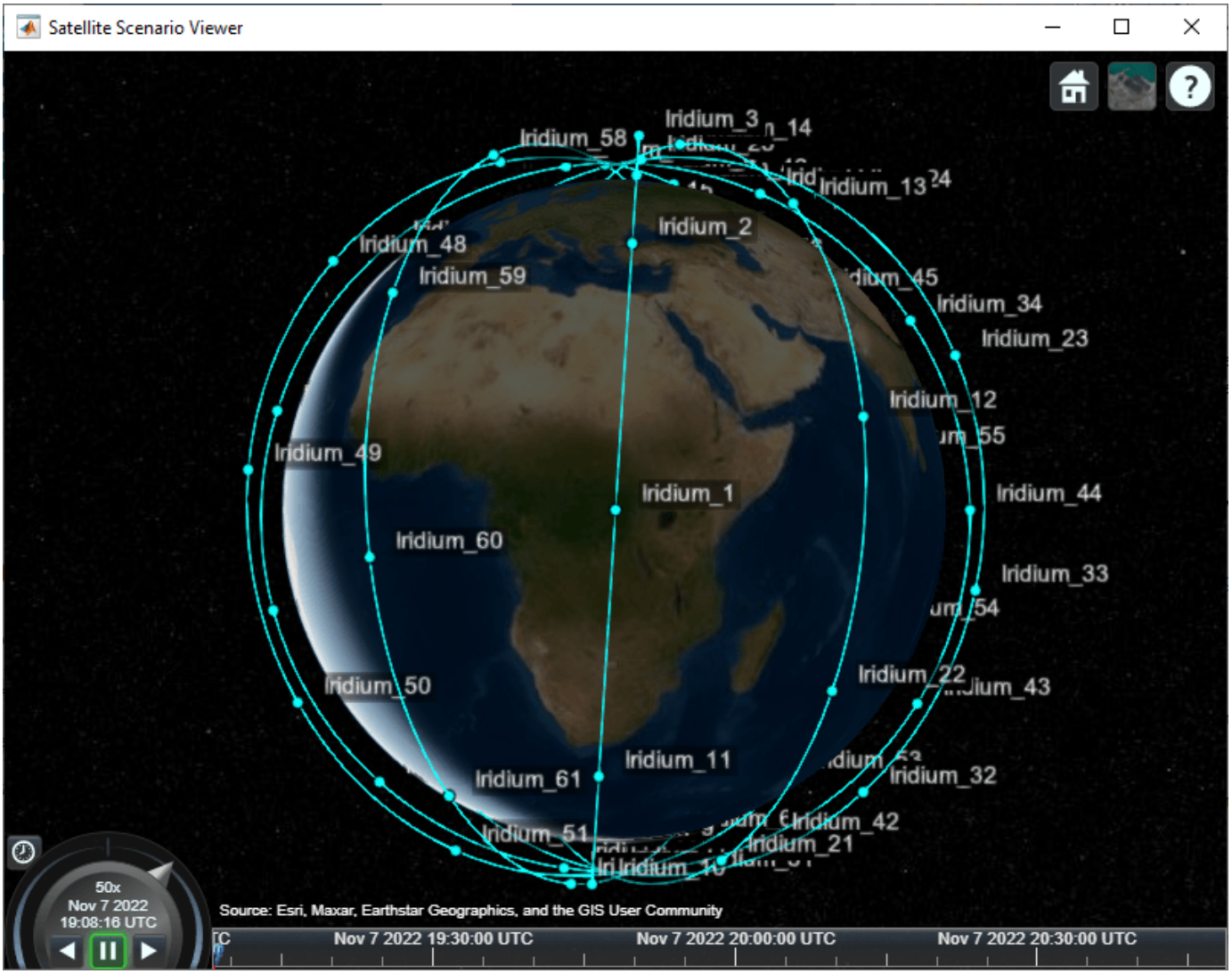
```
sc = satelliteScenario;
```

Create Walker-Star constellation that contains 66 satellites in six planes inclined at 86.4 degrees (86.4:66/6/2) in a 781 kilometer orbit.

```
sat = walkerStar(sc, 781e3+6378.14e3, 86.4, 66, 6, 2, Name="Iridium");
```

Visualize the scenario using the Satellite Scenario Viewer. Hide the orbits and labels of the satellites by setting the `ShowDetails` name-value pair to false. Mouse over satellites to show their labels. Click on a satellite to reveal its label, orbit, and any other hidden graphics. Click on the satellite again to dismiss them.

```
satelliteScenarioViewer(sc);
```



Input Arguments

`scenario` — Satellite scenario

`satelliteScenario` object

Satellite scenario, specified as a `satelliteScenario` object.

`radius` — Orbital radius

scalar

Orbital radius, specified as a scalar, in meters.

Data Types: `double`

`inclination` — Inclination

scalar

Inclination, specified as a scalar, in degrees.

Data Types: `double`

`totalSatellites` — Total number of satellites
scalar positive integer

Total number of satellites, specified as a scalar positive integer.

Data Types: `double`

`geometryPlanes` — Number of equally spaced geometry planes
scalar positive integer

Number of equally spaced geometry planes, specified as a scalar positive integer.

Data Types: `double`

`phasing` — Phasing between satellites
scalar integer greater than or equal to 0 and less than `geometryPlanes`

Phasing between satellites in adjacent planes, specified as a scalar integer greater than or equal to 0 and less than `geometryPlanes`. The change in true anomaly for equivalent satellites in neighboring planes is calculated as:

```
(phasing*180/totalSatellites).
```

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `sat = walkerStar(sc,29599.8e3,56, 24,3,1,Name="Galileo")` creates an array of satellites named Galileo.

`RAAN` — Right ascension of ascending node
0 (default) | scalar value from 0 to 180

Right ascension of ascending node, specified as a scalar value from 0 to 180. RAAN is the angle in the equatorial plane from the *x*-axis to the location of the ascending node, point at which the satellite crosses the equator from south to north, in degrees. The function uses this value as the starting point for distribution of the satellites along the equator.

Data Types: `double`

`ArgumentOfLatitude` — Argument of latitude
0 (default) | scalar value from 0 to 180

Argument of latitude, specified as a scalar value from 0 to 180. `ArgumentOfLatitude` is the angle between the ascending node and the body. The function uses this value as the starting point for distribution of satellites along the first orbital track.

Data Types: `double`

`Name` — Constellation name
`WalkerStar` (default) | scalar

Constellation name, specified as a scalar.

Individual satellite names within the constellation use the constellation name appended with increasing whole numbers starting with 1, for example, `Name_1`, `Name_2`, and so forth.

Data Types: `char` | `string`

`OrbitPropagator` — Name of orbit propagator
`"two-body-keplerian"` (default) | `"sgp4"` | `"sdp4"`

Name of orbit propagator used for propagating satellite position and velocity, specified as:

- `"two-body-keplerian"` — Two-Body-Keplerian orbit propagator based on the relative two-body model that assumes a spherical gravity field for the Earth and neglects third body effects and other environmental perturbations. It is considered the least accurate.
- `"sgp4"` — Simplified General Perturbations-4 orbit propagator.
- `"sdp4"` — Simplified Deep-Space Perturbations-4 orbit propagator.

Data Types: `string` | `char`

Output Arguments

`sat` — Satellites in scenario
array of `Satellite` objects

Satellite in scenario, returned as an array of `Satellite` objects.

Algorithms

Walker-Star constellations are a common solution for maximizing geometric coverage over Earth while minimizing the number of satellites required to perform the mission. Walker-Star constellation patterns use the notation:

`I:T/P/F.`

where:

- `I` — Orbital inclination
- `T` — Total number of satellites, which must be divisible by `F`
- `P` — Number of equally spaced geometric planes
- `F` — Phasing between satellites in adjacent planes

To define the radial height of the circular orbit (with respect to the Earth center), the function also requires a radius `r`.

In addition:

- The ascending nodes of the orbital planes of a Walker-Star constellation are uniformly distributed at intervals of 180/`P` deg around the equator.
- The number of satellites per plane, `satellitesPerPlane`, is defined as

  `satellitesPerPlane=T/P.`

The satellites in each orbital plane are distributed at intervals of 180/`satellitesPerPlane` deg. `F` represents the interplane phasing, the number of empty slots between the first satellites in each orbital plane.

Note Walker-Star constellations differ from Walker-Delta constellations. For Walker-Star constellations, the function distributes ascending nodes of the orbital planes across 180 degrees. For Walker-Delta constellations, the function distributes ascending nodes across 360 degrees.

Version History

Introduced in R2023a

See Also

`satelliteScenario` | `satellite` | `access` | `walkerDelta`

Topics

“Analyze Access Between a Satellite Constellation and an Aircraft” on page 5-228

wrldmagm

Use World Magnetic Model

Syntax

```
[XYZ,H,D,I,F] = wrldmagm(height,latitude,longitude,decimalYear)
[XYZ,H,D,I,F] = wrldmagm(height,latitude,longitude,decimalYear,model)
[XYZ,H,D,I,F] = wrldmagm(height,latitude,longitude,decimalYear,
'Custom',filename)
```

Description

`[XYZ,H,D,I,F] = wrldmagm(height,latitude,longitude,decimalYear)` calculates the Earth magnetic field at a specific location and time using the World Magnetic Model (WMM) WMM2020.

`[XYZ,H,D,I,F] = wrldmagm(height,latitude,longitude,decimalYear,model)` calculates the Earth magnetic field using World Magnetic Model `model`.

`[XYZ,H,D,I,F] = wrldmagm(height,latitude,longitude,decimalYear,
'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 WMM2020

Calculate the magnetic model 1000 meters over Natick, Massachusetts, on July 4, 2020, using the WMM2020 model.

```
[XYZ, H, D, I, F] = wrldmagm(1000, 42.283, -71.35, decyear(2020,7,4),'2020')
```

```
XYZ =
   1.0e+04 *

    1.9738
   -0.5014
    4.7556

H =
   2.0364e+04

D =
  -14.2536

I =
   66.8184
```

```
F =
   5.1733e+04
```

Calculate Magnetic Model for Downloaded `WMM2020.COF` file

Calculate the magnetic model 1000 meters over Natick, Massachusetts, on July 4, 2020, using a downloaded `WMM2020.COF` file.

```
[XYZ, H, D, I, F] = wrldmagm(1000, 42.283, -71.35, decyear(2020,7,4),'Custom','WMM2020.COF')
```

```
XYZ =
   1.0e+04 *

    1.9738
   -0.5014
    4.7556

H =
   2.0364e+04

D =
  -14.2536

I =
   66.8184

F =
   5.1733e+04
```

Input Arguments

`height` — Distance

scalar

Distance from the surface of the Earth, specified as a scalar, in meters.

Data Types: `double`

`latitude` — Geodetic latitude

scalar

Geodetic latitude, specified as a scalar, in degrees. North latitude is positive and south latitude is negative.

This function accepts latitude values greater than 90 and less than -90.

Data Types: `double`

`longitude` — Geodetic longitude

scalar

Geodetic longitude, specified as a scalar, in degrees. East longitude is positive and west longitude is negative. This function accepts longitude values greater than 180 and less than -180.

Data Types: `double`

`decimalYear` — Decimal year
scalar

Year, in 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
`'2020'` (default) | `'2015v2'` | `'2015'` | `'2015v1'` | `'2010'` | `'2005'` | `'2000'` | character vector | string

World Magnetic Model, specified as a character vector or string.

Model	Description
`'2020'`	WMM2020 (epoch 2020–2025).
`'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.

`D` — Declination
scalar

Declination, returned as a scalar, in degrees (+ve east).

`I` — Inclination
scalar

Inclination, returned as a scalar, in degrees (+ve down).

`F` — Total intensity
scalar

Total intensity, returned as a scalar, in nanotesla (nT).

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.
- This function has the limitations of the World Magnetic Model (WMM). WMM2020 is valid between -1km and 850km, as outlined in the World Magnetic Model 2020 Technical Report.
- 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.

Version History

Introduced in R2006b

See Also

`decyear` | `igrfmagm`

External Websites
https://www.ngdc.noaa.gov/geomag/WMM/DoDWMM.shtml

Aerospace Toolbox Examples

- “Import from USAF Digital DATCOM Files” on page 5-2
- “Create a Flight Animation from Trajectory Data” on page 5-17
- “Estimate G Forces for Flight Data” on page 5-21
- “Calculating Best Glide Quantities” on page 5-26
- “Overlaying Simulated and Actual Flight Data” on page 5-31
- “Comparing Zonal Harmonic Gravity Model to Other Gravity Models” on page 5-39
- “Calculating Compressor Power Required in a Supersonic Wind Tunnel” on page 5-46
- “Analyzing Flow with Friction Through an Insulated Constant Area Duct” on page 5-52
- “Determine Heat Transfer and Mass Flow Rate in a Ramjet Combustion Chamber” on page 5-57
- “Solving for the Exit Flow of a Supersonic Nozzle” on page 5-63
- “Visualize World Magnetic Model Contours for 2020 Epoch” on page 5-73
- “Visualizing Geoid Height for Earth Geopotential Model 1996” on page 5-82
- “Marine Navigation Using Planetary Ephemerides” on page 5-88
- “Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-96
- “Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99
- “Aerospace Flight Instruments in App Designer” on page 5-104
- “Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106
- “Customize Fixed-Wing Aircraft with Additional Aircraft States” on page 5-113
- “Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121
- “Modeling Satellite Constellations Using Ephemeris Data” on page 5-130
- “Satellite Constellation Access to Ground Station” on page 5-140
- “Comparison of Orbit Propagators” on page 5-151
- “Detect and Track LEO Satellite Constellation with Ground Radars” on page 5-159
- “Get Started with Fixed-Wing Aircraft” on page 5-170
- “Customize Fixed-Wing Aircraft with the Object Interface” on page 5-186
- “Multi-Hop Path Selection Through Large Satellite Constellation” on page 5-196
- “Modeling Custom Satellite Attitude and Gimbal Steering” on page 5-204
- “Aircraft Design Optimization with the Fixed-Wing Object” on page 5-215
- “Compute Indicated Airspeed for Pitot-Static Airspeed Indicator” on page 5-221
- “Differences Between Corrected Airspeeds” on page 5-224
- “Analyze Access Between a Satellite Constellation and an Aircraft” on page 5-228
- “Satellite Conjunction Finder” on page 5-237

Import 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 DATCOM    *
                                                  *    PROGRAM REV. JAN 96   DIRECT INQUIRIES TO:    *
                                                  *   WRIGHT LABORATORY  (WL/FIGC)  ATTN: W. BLAKE   *
                                                  *         WRIGHT PATTERSON AFB, OHIO  45433        *
                                                  *    PHONE (513) 255-6764,   FAX (513) 258-4054    *
                                                  ************************************************
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 VERSION OF D
                                                    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 =   0.26404
0                           MACH= 0.2000 LIFT-CURVE-SLOPE =   0.09811 /DEG.      XAC =   0.26457
1                               AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VERSION OF D
                                                  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 =   0.25854
0                           MACH= 0.2000 LIFT-CURVE-SLOPE =   0.09761 /DEG.      XAC =   0.25881
1                               AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VERSION OF D
                                                   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 =   0.25854
0                         MACH= 0.2000 LIFT-CURVE-SLOPE =   0.09761 /DEG.      XAC =   0.25881
1                               AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VERSION OF
                                         CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDESLIP
                                        WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGURATION
                                 SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 ----------------------  FLIGHT CONDITIONS  -----------------------           --------------  F
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.      REFER
 NUMBER                                                       NUMBER              AREA       LONG
             FT       FT/SEC     LB/FT**2       DEG R         1/FT               FT**2        FT
0 0.100    5000.00     109.70   1.7609E+03     500.843     6.1507E+05           225.800      5.75
0                                                                  -------------------DERIVATIVE (PE
0 ALPHA     CD       CL       CM       CN       CA       XCP        CLA          CMA          CYE
0
   -2.0    0.032    0.113   -0.0340   0.112    0.035   -0.304    8.926E-02   -2.105E-02   -3.458E
    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 VERSION OF
                                                         DYNAMIC DERIVATIVES
                                        WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGURATION
                                 SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 ----------------------  FLIGHT CONDITIONS  -----------------------           --------------  F
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.      REFER
 NUMBER                                                       NUMBER              AREA       LONG
             FT       FT/SEC     LB/FT**2       DEG R         1/FT               FT**2        FT
0 0.100    5000.00     109.70   1.7609E+03     500.843     6.1507E+05           225.800      5.75
                                                     DYNAMIC DERIVATIVES (PER DEGREE)
0           -------PITCHING-------    -----ACCELERATION------    --------------ROLLING-----------
0   ALPHA       CLQ          CMQ           CLAD         CMAD         CLP          CYP          CN
0
    -2.00    9.739E-02   -8.918E-02     1.874E-02   -4.247E-02   -7.824E-03   -1.516E-03   -1.498
```

```
     0.00                              1.913E-02   -4.336E-02   -8.226E-03   -1.649E-03   -4.034
     2.00                              1.991E-02   -4.512E-02   -8.599E-03   -1.792E-03   -6.631
     4.00                              2.003E-02   -4.540E-02   -8.890E-03   -1.942E-03   -9.290
     8.00                              1.952E-02   -4.424E-02   -9.387E-03   -2.262E-03   -1.479
1                            AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VERSION OF
                                      CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDESLIP
                                    WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGURATION
                                 SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 ----------------------  FLIGHT CONDITIONS  ------------------------           --------------  R
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.      REFER
 NUMBER                                                       NUMBER             AREA       LONG
             FT       FT/SEC     LB/FT**2       DEG R         1/FT              FT**2        FT
0 0.200    5000.00     219.39   1.7609E+03     500.843     1.2301E+06          225.800      5.75
0                                                                  -------------------DERIVATIVE (PE
0 ALPHA     CD       CL       CM       CN       CA       XCP        CLA           CMA          CYB
0
   -2.0    0.028    0.114   -0.0335   0.113    0.032   -0.297    9.000E-02   -2.124E-02   -3.465E
    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 VERSION OF
                                                DYNAMIC DERIVATIVES
                                    WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGURATION
                                 SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 ----------------------  FLIGHT CONDITIONS  ------------------------           --------------  R
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.      REFER
 NUMBER                                                       NUMBER             AREA       LONG
             FT       FT/SEC     LB/FT**2       DEG R         1/FT              FT**2        FT
0 0.200    5000.00     219.39   1.7609E+03     500.843     1.2301E+06          225.800      5.75
                                                       DYNAMIC DERIVATIVES (PER DEGREE)
0          -------PITCHING-------    -----ACCELERATION------    --------------ROLLING-----------
0   ALPHA       CLQ          CMQ          CLAD         CMAD         CLP          CYP          CN
0
    -2.00    9.840E-02   -8.993E-02     1.900E-02   -4.307E-02   -7.877E-03   -1.525E-03   -1.499
     0.00                              1.940E-02   -4.398E-02   -8.276E-03   -1.659E-03   -4.038
     2.00                              2.018E-02   -4.574E-02   -8.646E-03   -1.802E-03   -6.637
     4.00                              2.030E-02   -4.602E-02   -8.934E-03   -1.953E-03   -9.297
     8.00                              1.978E-02   -4.483E-02   -9.423E-03   -2.273E-03   -1.479
1                            AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VERSION OF
                                      CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDESLIP
                                    WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGURATION
                                 SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 ----------------------  FLIGHT CONDITIONS  ------------------------           --------------  R
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.      REFER
 NUMBER                                                       NUMBER             AREA       LONG
             FT       FT/SEC     LB/FT**2       DEG R         1/FT              FT**2        FT
0 0.100    8000.00     108.52   1.5721E+03     490.151     5.6457E+05          225.800      5.75
```

```
0                                                                    ------------------DERIVATIVE (PI
0 ALPHA     CD       CL       CM       CN       CA       XCP        CLA          CMA          CYB
0
   -2.0    0.032    0.113   -0.0340   0.112    0.036   -0.305    8.926E-02   -2.106E-02   -3.458E
    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 VERSION OF
                                                      DYNAMIC DERIVATIVES
                                          WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGURATION
                                    SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 ----------------------  FLIGHT CONDITIONS  ------------------------           -------------  R
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.      REFER
 NUMBER                                                       NUMBER             AREA       LONG
            FT       FT/SEC     LB/FT**2       DEG R         1/FT              FT**2        FT
0 0.100    8000.00     108.52   1.5721E+03     490.151     5.6457E+05          225.800      5.75
                                                      DYNAMIC DERIVATIVES (PER DEGREE)
0           -------PITCHING-------     -----ACCELERATION------     --------------ROLLING-----------
0   ALPHA       CLQ          CMQ           CLAD         CMAD         CLP          CYP          CN
0
    -2.00    9.739E-02   -8.918E-02     1.874E-02   -4.247E-02   -7.824E-03   -1.516E-03   -1.498
     0.00                               1.913E-02   -4.336E-02   -8.226E-03   -1.649E-03   -4.034
     2.00                               1.991E-02   -4.512E-02   -8.599E-03   -1.792E-03   -6.631
     4.00                               2.003E-02   -4.540E-02   -8.890E-03   -1.942E-03   -9.290
     8.00                               1.952E-02   -4.424E-02   -9.387E-03   -2.262E-03   -1.479
1                              AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VERSION OF
                                        CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDESLIP
                                          WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGURATION
                                    SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 ----------------------  FLIGHT CONDITIONS  ------------------------           -------------  R
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.      REFER
 NUMBER                                                       NUMBER             AREA       LONG
            FT       FT/SEC     LB/FT**2       DEG R         1/FT              FT**2        FT
0 0.200    8000.00     217.04   1.5721E+03     490.151     1.1291E+06          225.800      5.75
0                                                                    ------------------DERIVATIVE (PI
0 ALPHA     CD       CL       CM       CN       CA       XCP        CLA          CMA          CYB
0
   -2.0    0.028    0.114   -0.0335   0.113    0.032   -0.297    9.000E-02   -2.124E-02   -3.465E
    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 VERSION OF
                                                    DYNAMIC DERIVATIVES
                                          WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGURATION
                                    SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 ----------------------  FLIGHT CONDITIONS  ------------------------           --------------  R
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.      REFER
 NUMBER                                                       NUMBER             AREA       LONG
             FT       FT/SEC     LB/FT**2       DEG R         1/FT              FT**2        FT
0 0.200    8000.00     217.04   1.5721E+03     490.151     1.1291E+06          225.800      5.75
                                                    DYNAMIC DERIVATIVES (PER DEGREE)
0           -------PITCHING-------     -----ACCELERATION------     --------------ROLLING-----------
0   ALPHA       CLQ          CMQ           CLAD         CMAD         CLP          CYP          CN
0
    -2.00    9.840E-02   -8.993E-02     1.900E-02   -4.307E-02   -7.877E-03   -1.525E-03   -1.499
     0.00                               1.940E-02   -4.398E-02   -8.276E-03   -1.659E-03   -4.038
     2.00                               2.018E-02   -4.574E-02   -8.646E-03   -1.802E-03   -6.637
     4.00                               2.030E-02   -4.602E-02   -8.934E-03   -1.953E-03   -9.297
     8.00                               1.978E-02   -4.483E-02   -9.424E-03   -2.273E-03   -1.479
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}, and C_{mq}

have data only in the first alpha value. Here are the imported data values.

`data.cyb`

```
ans =
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 =
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 =
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 =
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 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 tab = aerotab
    data.(tab) = fillmissing(data.(tab), "previous", MissingLocations=data.(tab)==99999);
end
```

Here are the updated imported data values:

```
data.cyb
```

```
ans =
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 =
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 =
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
```

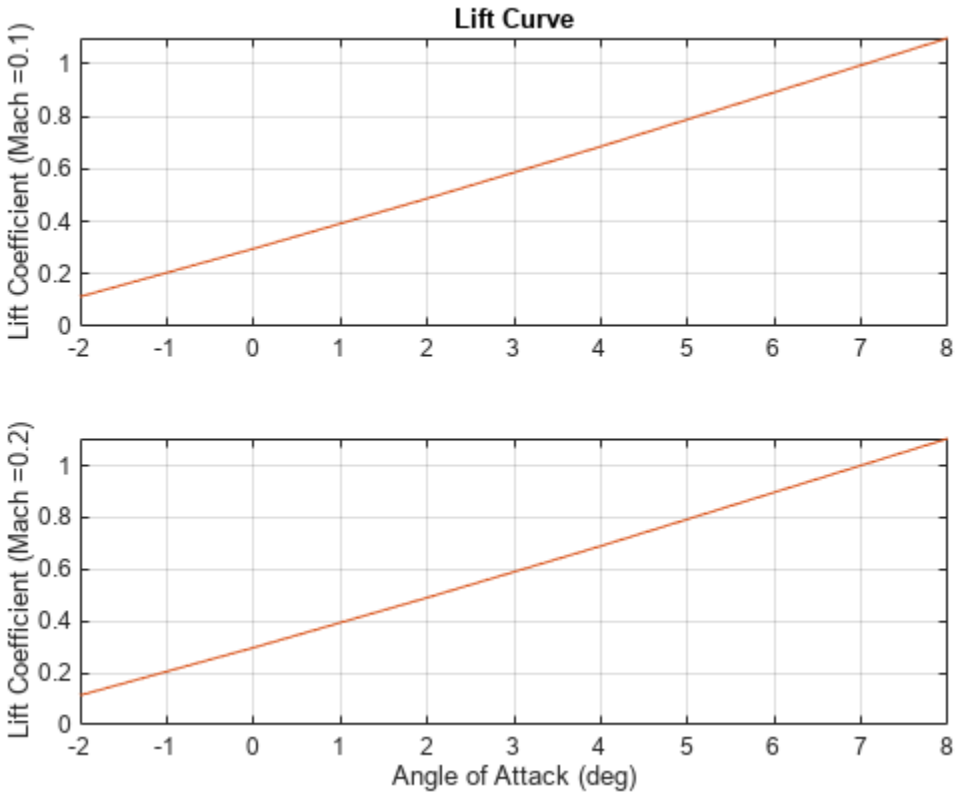
```
data.cmq
```

```
ans =
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
```

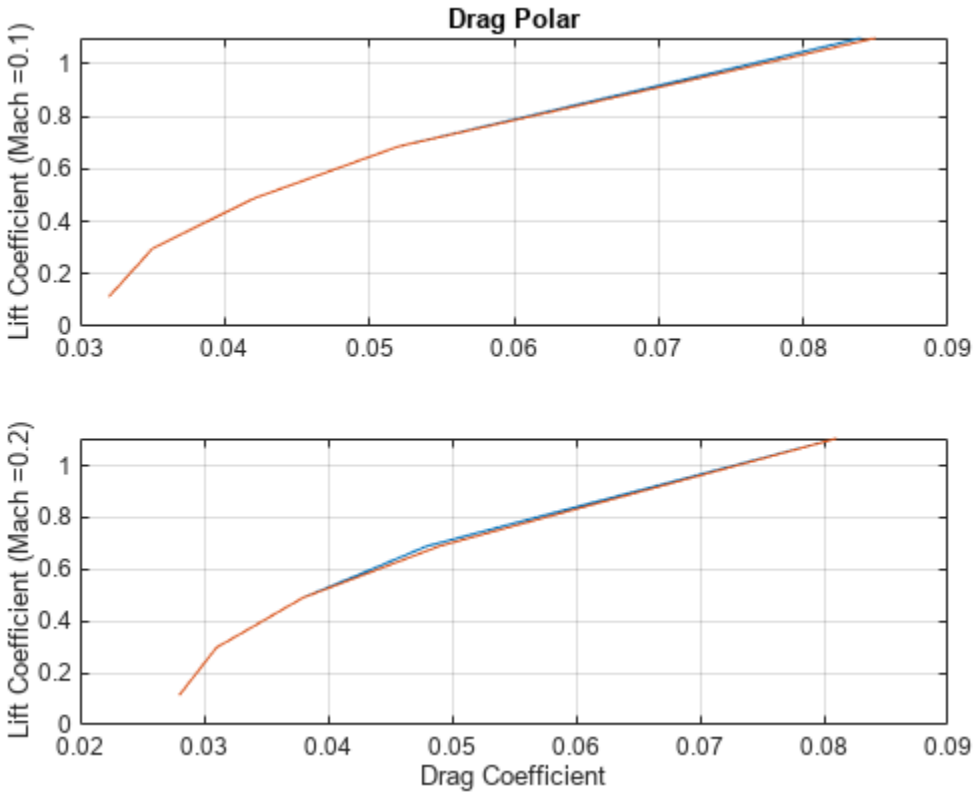
Plotting Aerodynamic Coefficients

Plot lift curve, drag polar, and pitching moments.

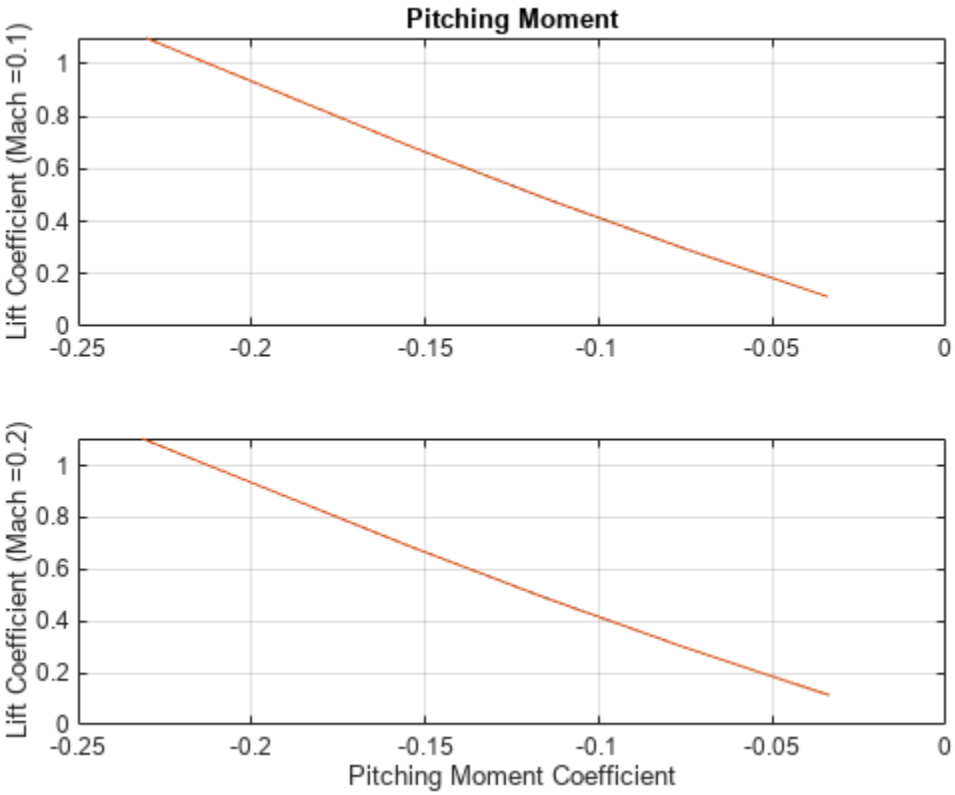
```
h1 = figure;
figtitle = {'Lift Curve' ''};
for k=1:2
    subplot(2,1,k)
    plot(data.alpha,permute(data.cl(:,k,:),[1 3 2]))
    grid
    ylabel(['Lift Coefficient (Mach =' num2str(data.mach(k)) ')'])
    title(figtitle{k});
end
xlabel('Angle of Attack (deg)')
```




```
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')
```



See Also

`datcomimport`

Related Examples

- “Digital DATCOM Data” on page 2-11

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 HL20 folder into the $FLIGHTGEAR/data/Aircraft/ folder. The HL20 folder is located in the working folder of this example, or under the following folder:

```
fullfile(matlabroot, "toolbox", "aero", "animation")
```

Load Recorded Flight Trajectory Data

The flight trajectory data for this example is stored in a comma separated value formatted file. Use **readmatrix** to read the data from the file.

```
tdata = readmatrix('asthl20log.csv');
```

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','r
```

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 aircraft geometry model, and the network information for FlightGear flight simulator.

```
h.FlightGearBaseDirectory = 'C:\Program Files\FlightGear';
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” on page 2-44.

```
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)
```

```
             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
               Architecture: 'Default'
                TimeScaling: 5
            FramesPerSecond: 12
                     TStart: NaN
                     TFinal: NaN
          TimeSeriesSource: [1x1 timeseries]
       TimeSeriesSourceType: 'Timeseries'
          TimeSeriesReadFcn: @TimeseriesRead
```

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.

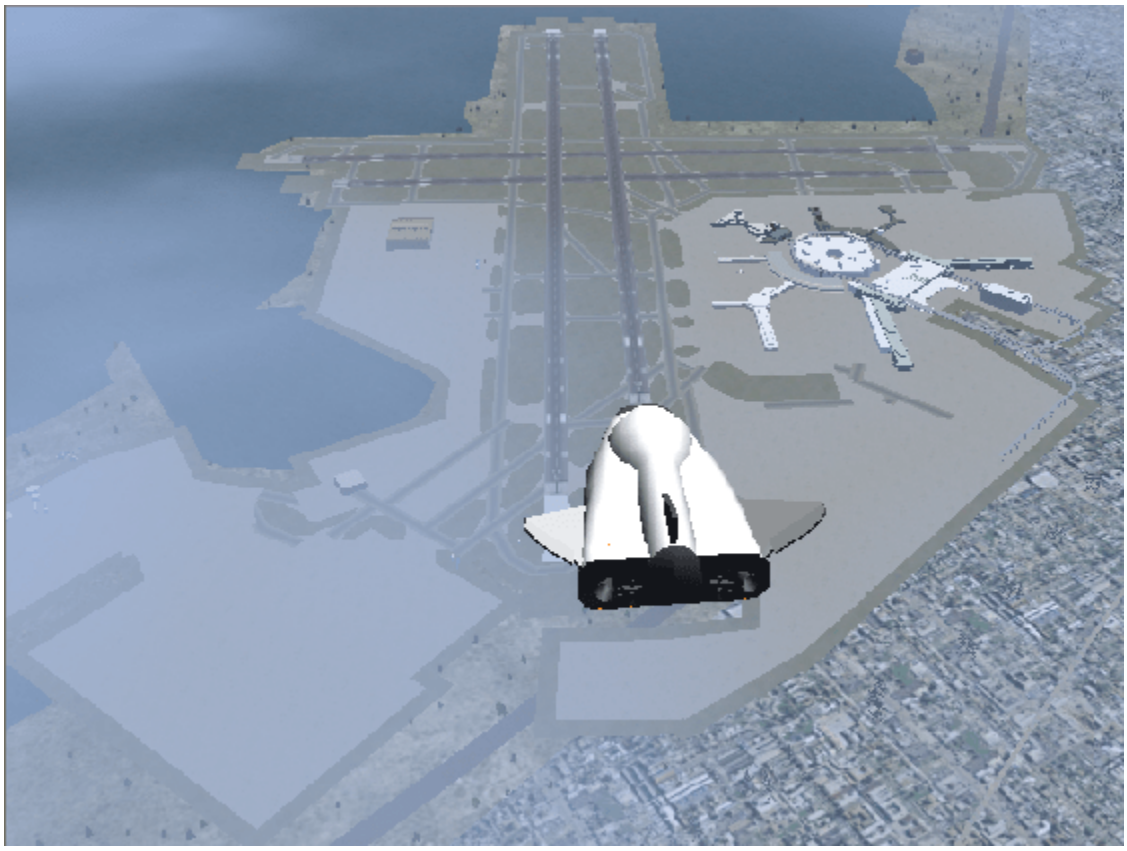
Play the Flight Animation of Trajectory Data

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('astfganim01.png','png'));
axis off;
set(gca,'Position',[ 0 0 1 1 ]);
set(gcf,'MenuBar','none');
```



See Also

`Aero.FlightGearAnimation`

Related Examples

- “Aero.FlightGearAnimation Objects” on page 2-42

Estimate 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 pilot-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}, and C_{mq}

have data only in the first alpha value. By default, missing data points are set to 99999 and filled using the "previous" method of fillmissing. 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.

```
data{1}.cyb = fillmissing(data{1}.cyb, "previous", "MissingLocations", data{1}.cyb == 99999);
data{1}.cnb = fillmissing(data{1}.cnb, "previous", "MissingLocations", data{1}.cnb == 99999);
data{1}.clq = fillmissing(data{1}.clq, "previous", "MissingLocations", data{1}.clq == 99999);
data{1}.cmq = fillmissing(data{1}.cmq, "previous", "MissingLocations", data{1}.cmq == 99999);
```

Interpolate Stability and Dynamic Derivatives at Flight Conditions

The stability and dynamic derivatives in the digital DATCOM structure are 3-D tables that are functions of Mach number, angle of attack in degrees, and altitude in feet. To perform 3-D linear interpolation (`griddedInterpolant`), the indices for the derivative tables are required to be monotonic full grid of points. Indices of this form are generated by the `ndgrid` function.

```
[mnum, alp, h] = ndgrid( 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 to 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 );
```

GriddedInterpolant can be used to linearly interpolate derivative tables to find static and dynamic derivatives at the flight conditions.

```
F = griddedInterpolant( mnum, alp, h, pagetranspose(data{1}.cd), 'linear');
cd = F(Mach, alphadeg, alt);
F = griddedInterpolant( mnum, alp, h, pagetranspose(data{1}.cyb), 'linear');
cyb = F(Mach, alphadeg, alt);
F = griddedInterpolant( mnum, alp, h, pagetranspose(data{1}.cl), 'linear');
cl = F(Mach, alphadeg, alt);
F = griddedInterpolant( mnum, alp, h, pagetranspose(data{1}.cyp), 'linear');
cyp = F(Mach, alphadeg, alt);
F = griddedInterpolant( mnum, alp, h, pagetranspose(data{1}.clad), 'linear');
clad = F(Mach, alphadeg, alt);
```

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 to degrees by the function `convang`.

```
betadeg  = convang( beta, 'rad', 'deg' );
```

To calculate the aerodynamic coefficients, the body angular rates (omega) need to be given in the stability axes, like the derivatives. The function `dcmbody2stability` generates the direction cosine matrix for body axes to stability axes (Tsb) when the sideslip angle (beta) is set to zero.

```
Tsb = dcmbody2stability( 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 than 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,length(omeg

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 a 3-D array, the pagetranspose function is used.

```
Tbs = pagetranspose(Tsb);
```

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(k)]);
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

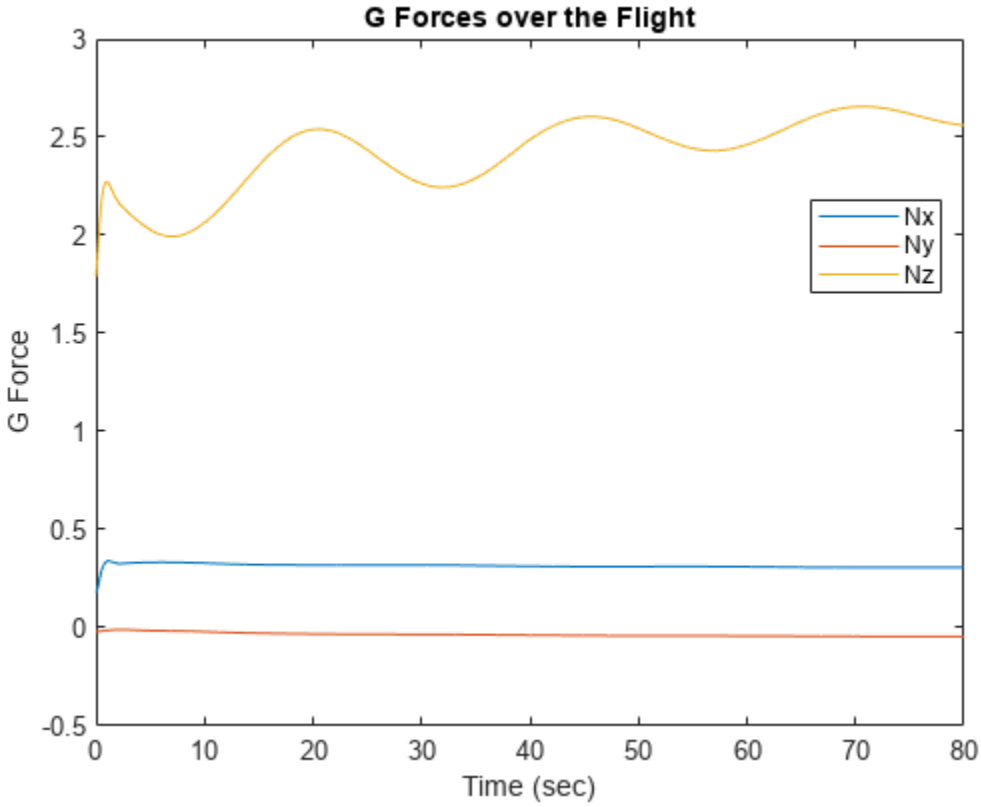
Using the calculated forces, estimate G forces during the flight.

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 .* [1,1,-1];
```

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')
```




```
close(h1);
```

See Also

atmoscoesa | convacc | convvel | correctairspeed | datcomimport | dpressure

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_D_bg = D_bg./(qbar*S);
```

$$C_{L_{bg}} = \frac{L_{bg}}{\bar{q}S}$$

```
C_L_bg = L_bg./(qbar*S);
```

Summary of Best Glide Values

Here are the best glide values:

$KCAS_{bg} = 71.9 KCAS$

$\gamma_{bg} = -5.38 \text{deg}$

$C_{D_{bg}} = 0.074$

$C_{L_{bg}} = 0.7859$

$D_{bg} = 224.9 lbf$

$L_{bg} = 2389.4 lbf$

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

$$C_{L_{bg}}$$

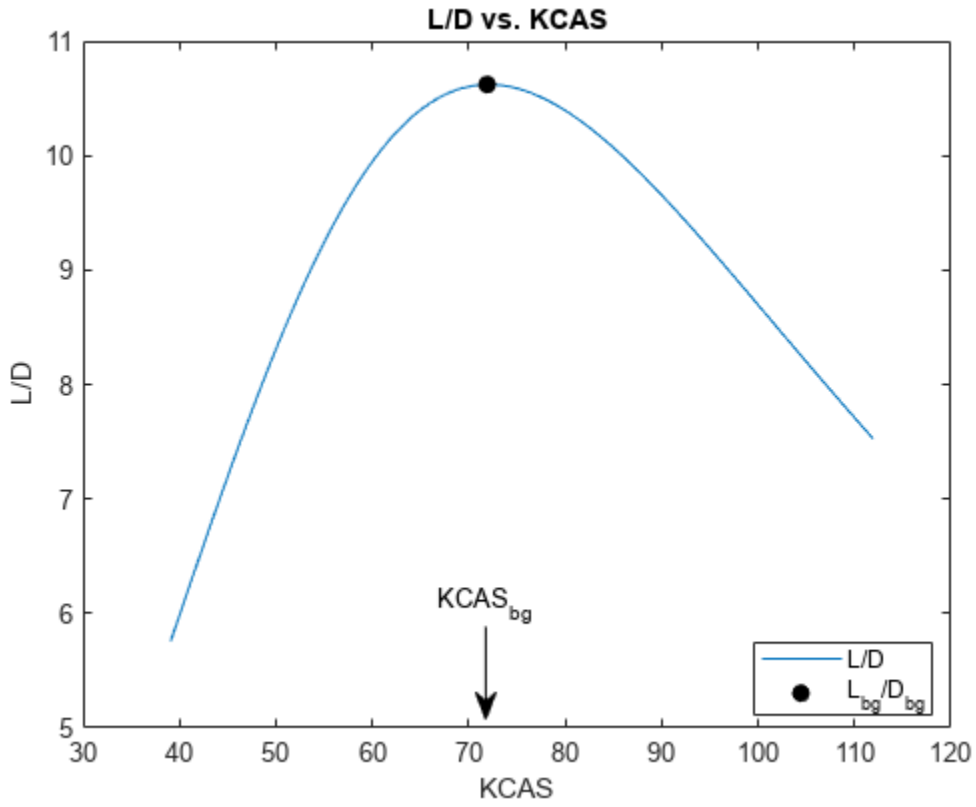
from above, the angle of attack is about 7 degrees. Adding the flight path angle (i.e. best glide angle) from above shows the fuselage pitch (attitude angle theta) to be about 2 degrees.

```
L = W;
```

Plot L/D versus KCAS

As expected, the maximum L/D occurs at approximately the best glide velocity calculated above.

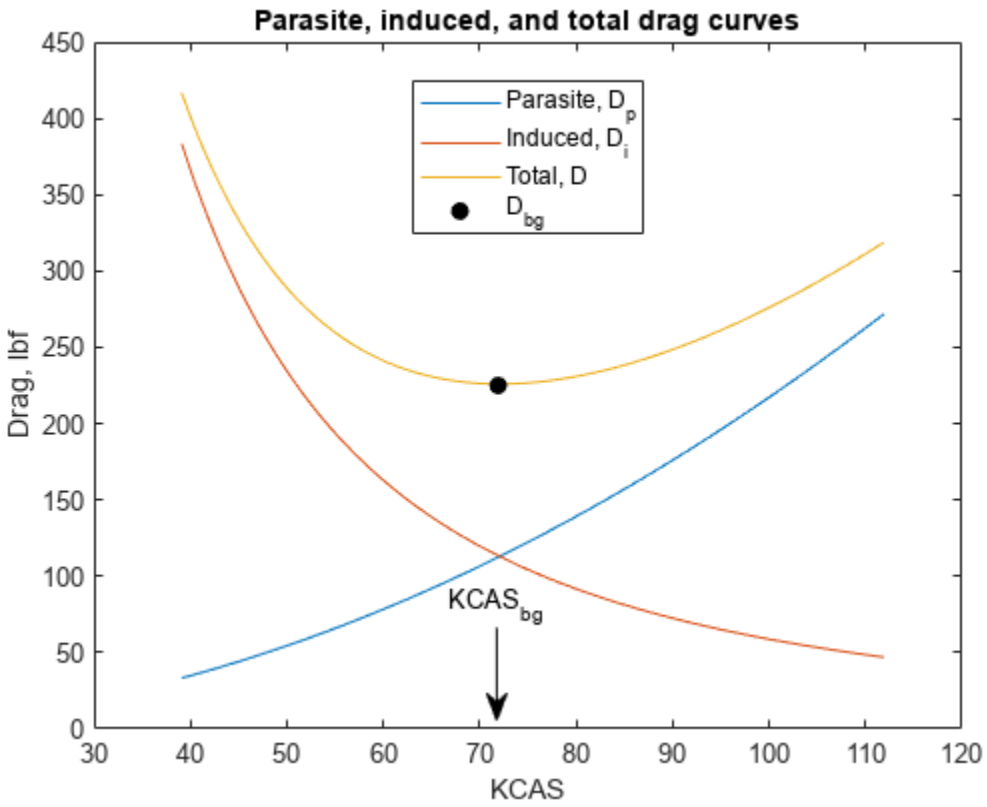
```
h1 = figure;
plot(KCAS,L./D);
title('L/D vs. KCAS');
xlabel('KCAS'); ylabel('L/D');
hold on
plot(KCAS_bg,L_bg/D_bg,'Marker','o','MarkerFaceColor','black',...
    'MarkerEdgeColor','black','Color','white');
hold off
legend('L/D','L_{bg}/D_{bg}','Location','Best');
annotation('textarrow',[0.49 0.49],[0.23 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.

See Also

`atmoscoesa` | `convvel` | `correctairspeed`

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

Create an instance of the `Aero.Animation` object.

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

Set the Animation Object Properties

Set the number of frames per second. This controls the rate at which frames are displayed in the figure window.

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

Set 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 subsecond precision.

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

Create and Load Bodies

Load the bodies using `createBody` for the animation object, `h`. This example uses these bodies to work with and display the simulated and actual flight trajectories. The first body is orange and represents simulated data. The second body is blue and represents 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.mat','simdata')
load('fltdata.mat','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/animation/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/animation/doFirstOrderChaseCameraDynamics.m`

Set `'PositionFcn'`.

```
h.Camera.PositionFcn = @doFirstOrderChaseCameraDynamics;
```

Display Body Geometries in Figure

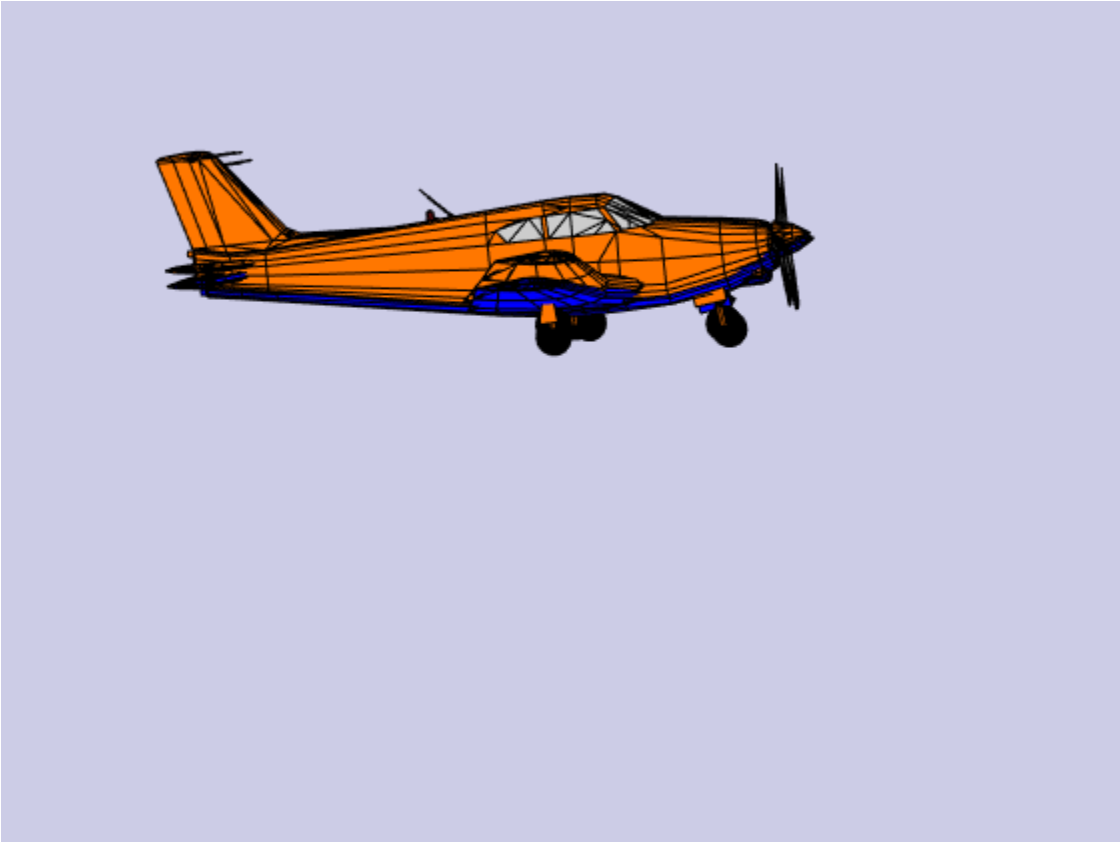
Use the `show` method to create the figure graphics object for the animation object.

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

Use the Animation Object to Play Back Flight Trajectories

Use the `play` method to animate bodies for the duration of the timeseries data. Using this method illustrates the slight differences between the simulated and flight data.

```
h.play();
h.updateCamera(5);
```



Wait

Wait for the animation to finish before editing the object properties.

```
h.wait();
```

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/animation/staticCameraPosition.m`

Set the new `'PositionFcn'`.

```
h.Camera.PositionFcn = @staticCameraPosition;
```

Run the animation with new `'PositionFcn'`.

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

Wait

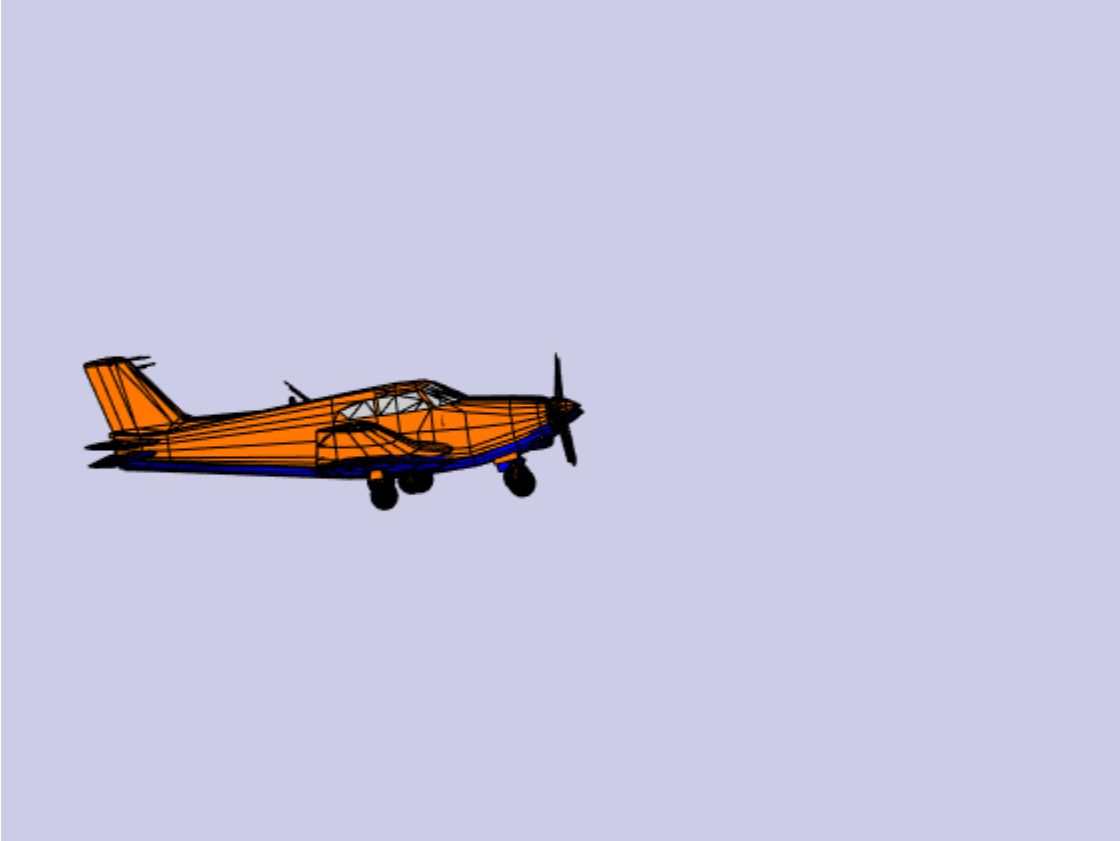
Wait for the animation to finish before editing the object properties.

```
h.wait();
```

Move Bodies

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

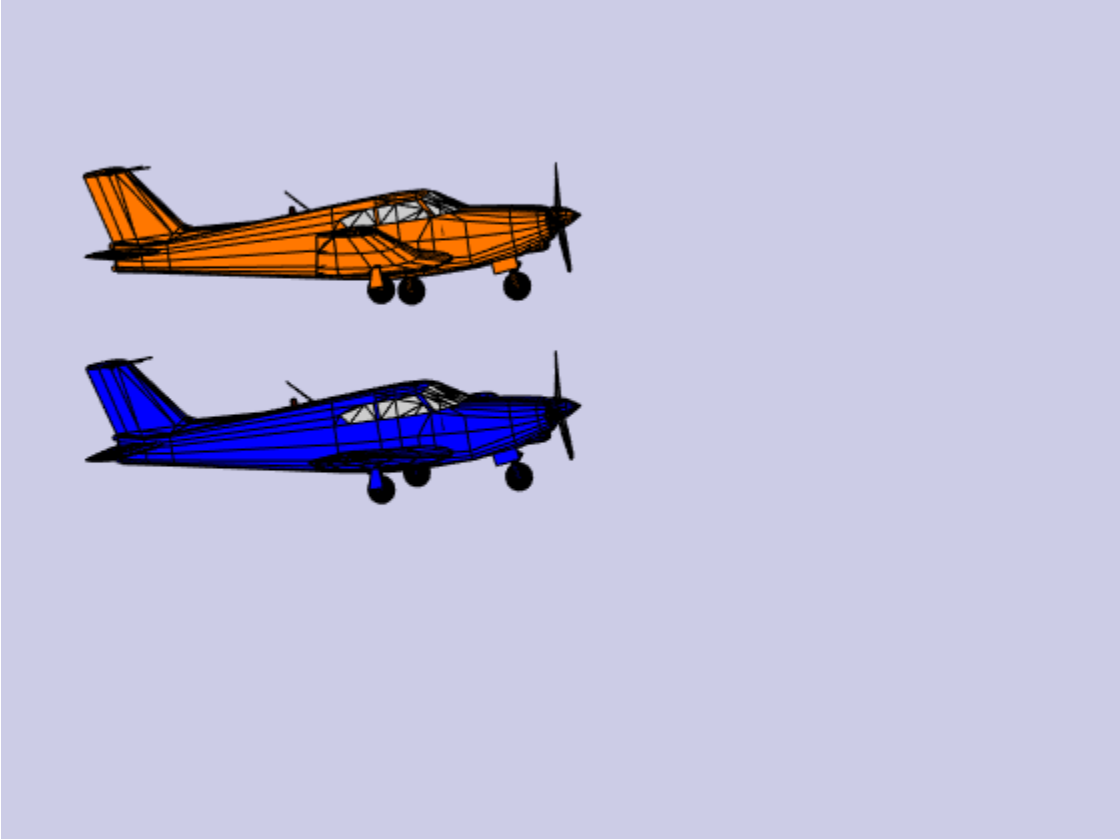
Reposition the bodies by first getting the current body position and then separating the bodies.

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

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;
```

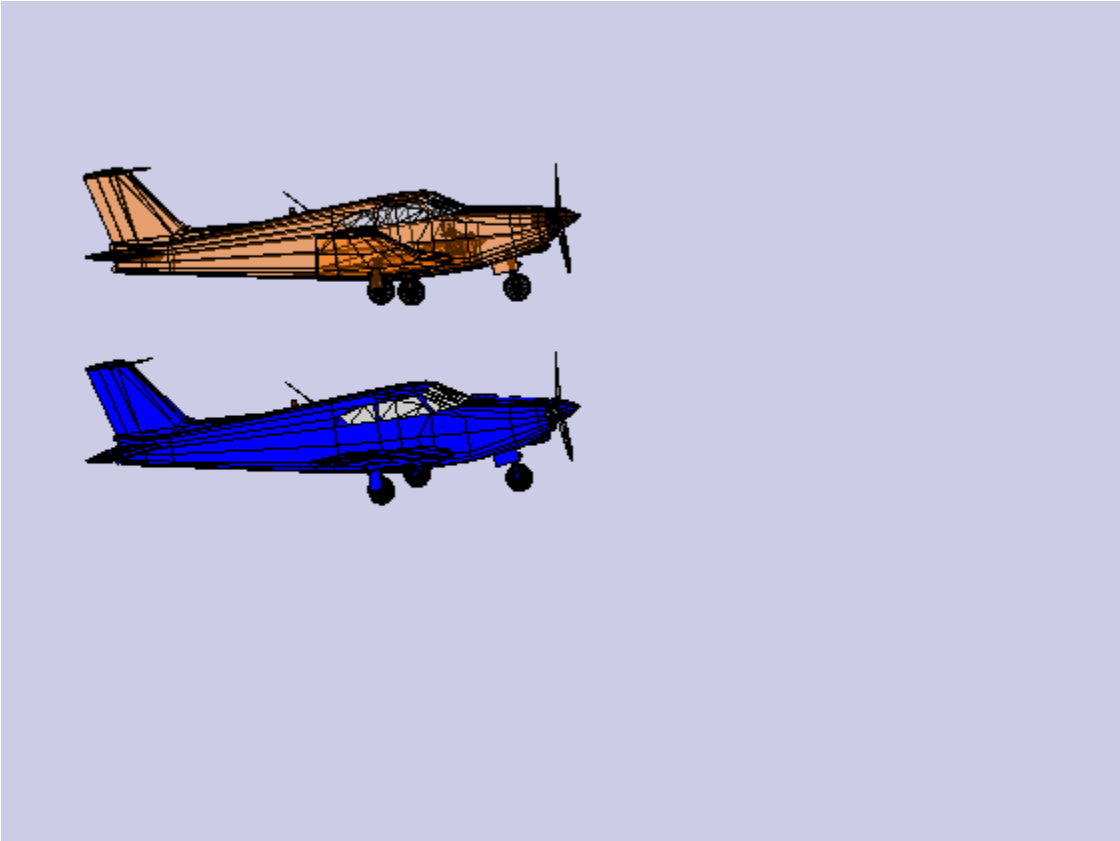
Set desired face and edge alpha values.

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

Get 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

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;
```

Set the desired patch color (red).

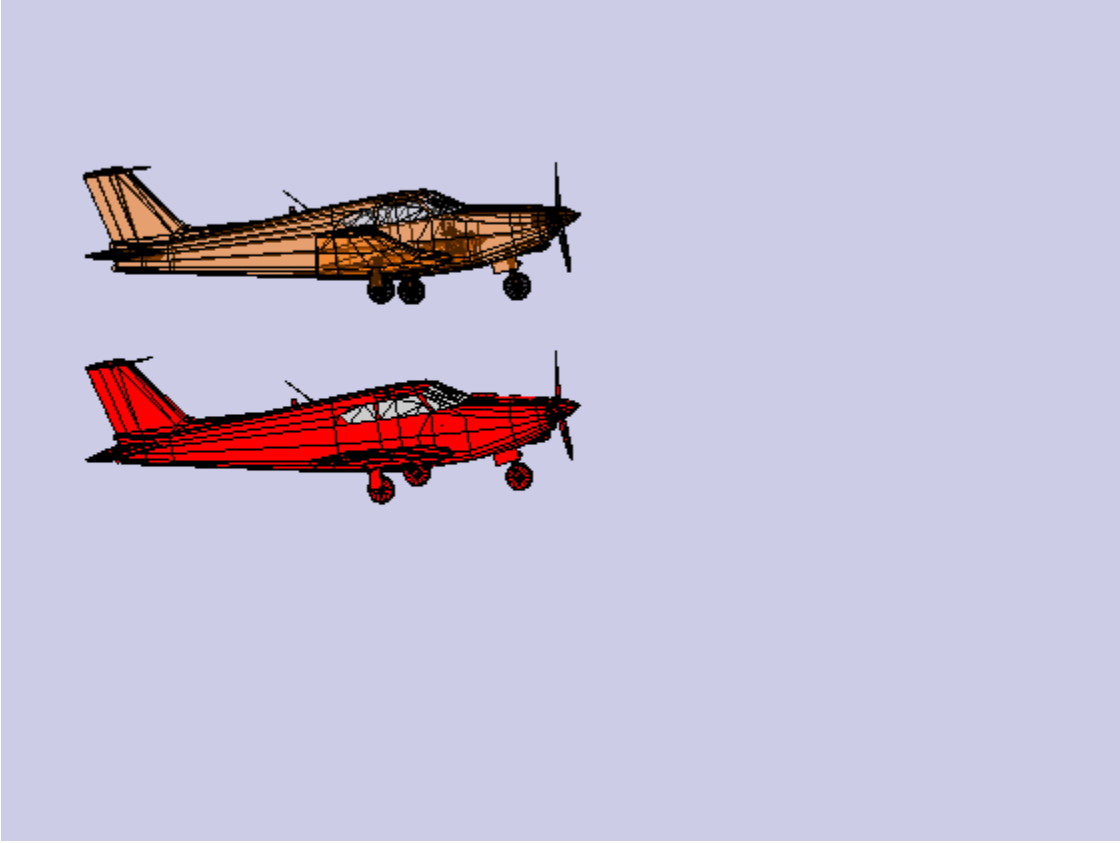
```
desiredColor = [1 0 0];
```

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, use an alternative approach to selective coloring.

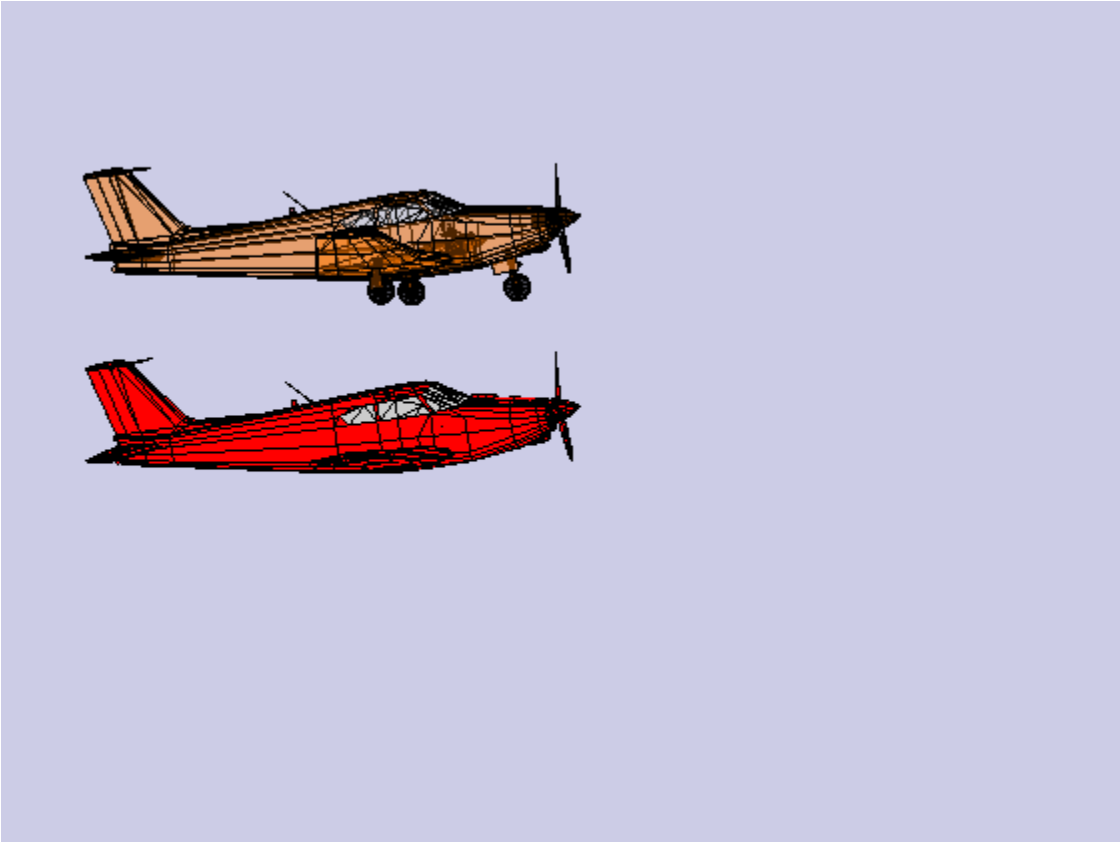
```
for k = 1:size(patchHandles3,1)
    tempFaceColor = get(patchHandles3(k),'FaceVertexCData');
    tempName = h.Bodies{2}.Geometry.FaceVertexColorData(k).name;
    if ~contains(tempName,'Windshield') &&...
       ~contains(tempName,'front-windows') &&...
       ~contains(tempName,'rear-windows')
 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



Turn Off Landing Gear on Second Body

Turn off the landing gear for the second body. To do this, the code 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

Close and delete.

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

```
%#ok<*REPMAT>
```

See Also

`Aero.Animation` | `updateBodies` | `updateCamera`

Related Examples

- “Aero.Animation Objects” on page 2-23
- “Simulated and Actual Flight Data Using Aero.Animation Objects” on page 2-24

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 the ECEF coordinate system is geocentric, you can use spherical equations to determine the position from the geocentric 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 zonal harmonic gravity components in the ECEF coordinate system in meters per seconds squared for an array of ECEF position inputs.

```
[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. WGS84 gravity is an array of geodetic latitudes in degrees and 0 degrees longitude computed 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 true fals
```

Determine Gravity for Spherical Earth

Compute the array of point-mass 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]);
gned_zonal = squeeze(pagemtimes(dcm_ef,gxyz_zonal))';

figure(1)
plot(lat_gd, gned_zonal(:,3), lat, -gd_sphere, '--', lat_gd, gd_wgs84, '-.')
legend('Zonal Harmonic', 'Point-mass', 'WGS84','Location','North')
xlabel('Geodetic Latitude (degrees)')
ylabel('Down gravity (m/s^2)')
grid on
```

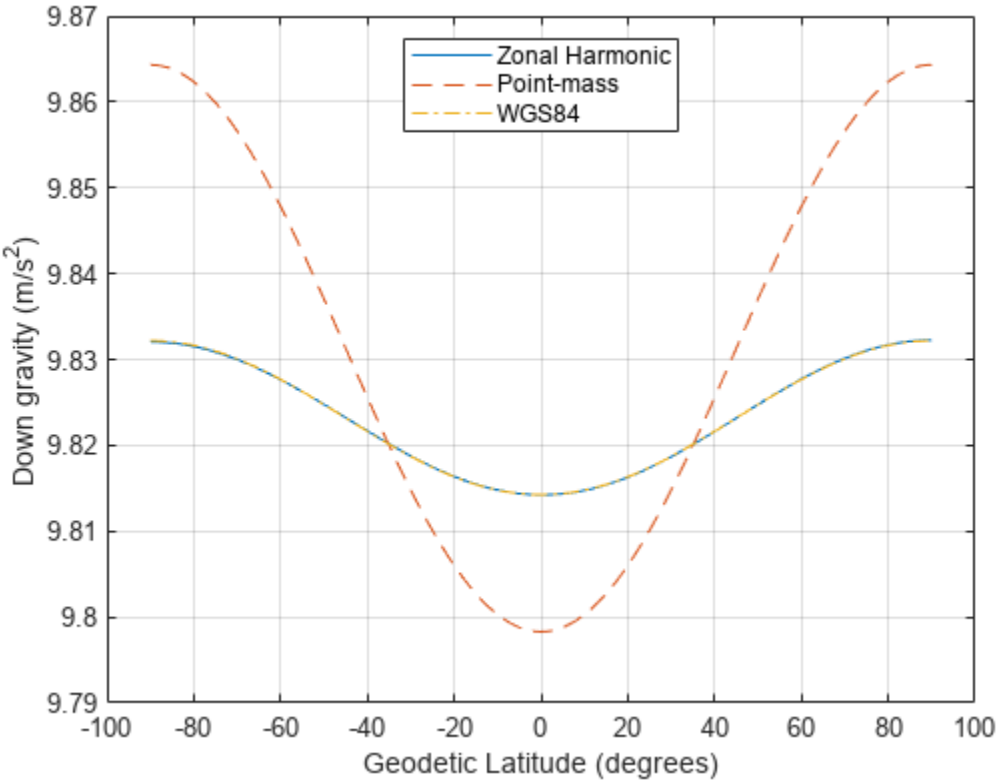


Figure 1: Gravity in the Down-axis in meters per second squared

```
figure(2)
plot( lat_gd, gned_zonal(:,1), [lat(1) lat(end)], [0 0], '--', lat_gd, gn_wgs84, '-.');
legend('Zonal Harmonic', 'Point-mass', 'WGS84','Location','Best')
xlabel('Geodetic Latitude (degrees)')
ylabel('North gravity (m/s^2)')
grid on
hold off
```

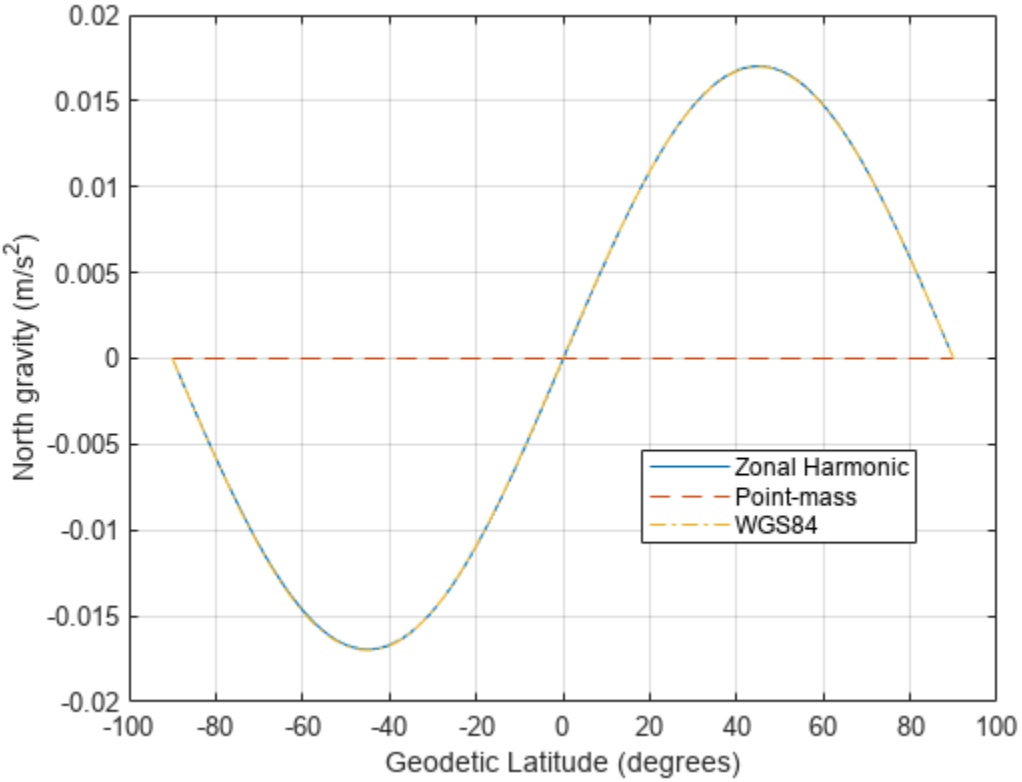


Figure 2: Gravity in the North-axis in meters per second squared

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', [false true fa
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 on
```

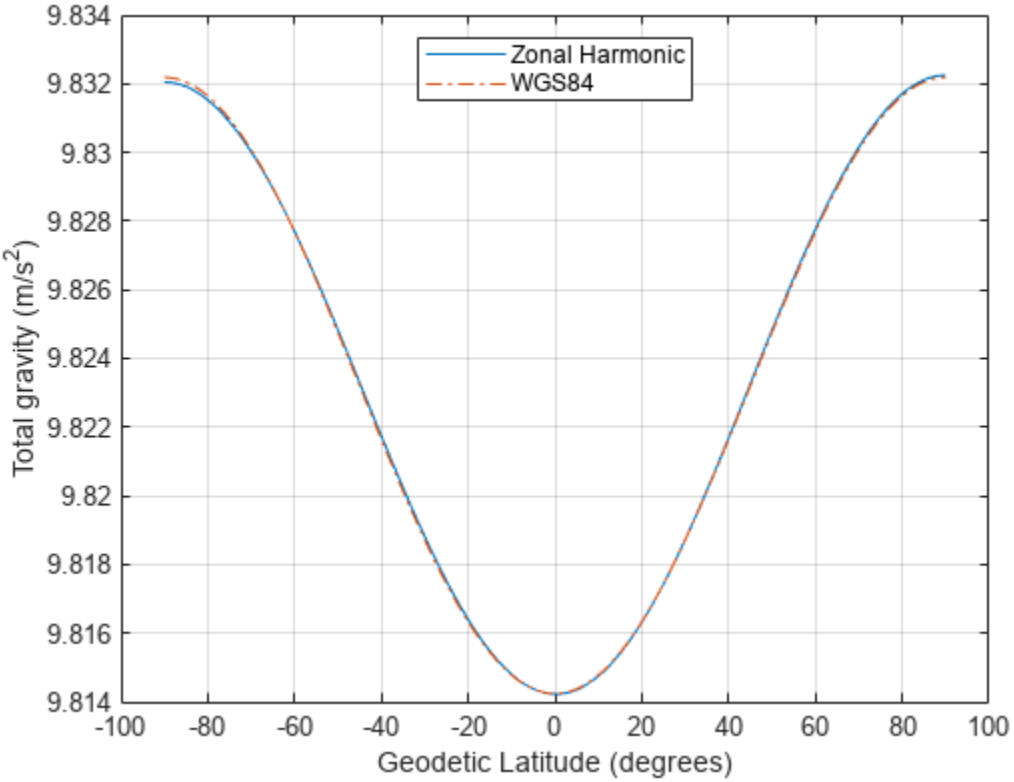


Figure 3: Total gravity in meters per second squared

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. WGS84 gravity is an array of geodetic latitudes in degrees and 0 degrees longitude computed 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, 'Exact', [false
```

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', [false fa
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]);
gned_cent_zonal = squeeze(pagemtimes(dcm_ef,gxyz_cent_zonal))';

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 on
```

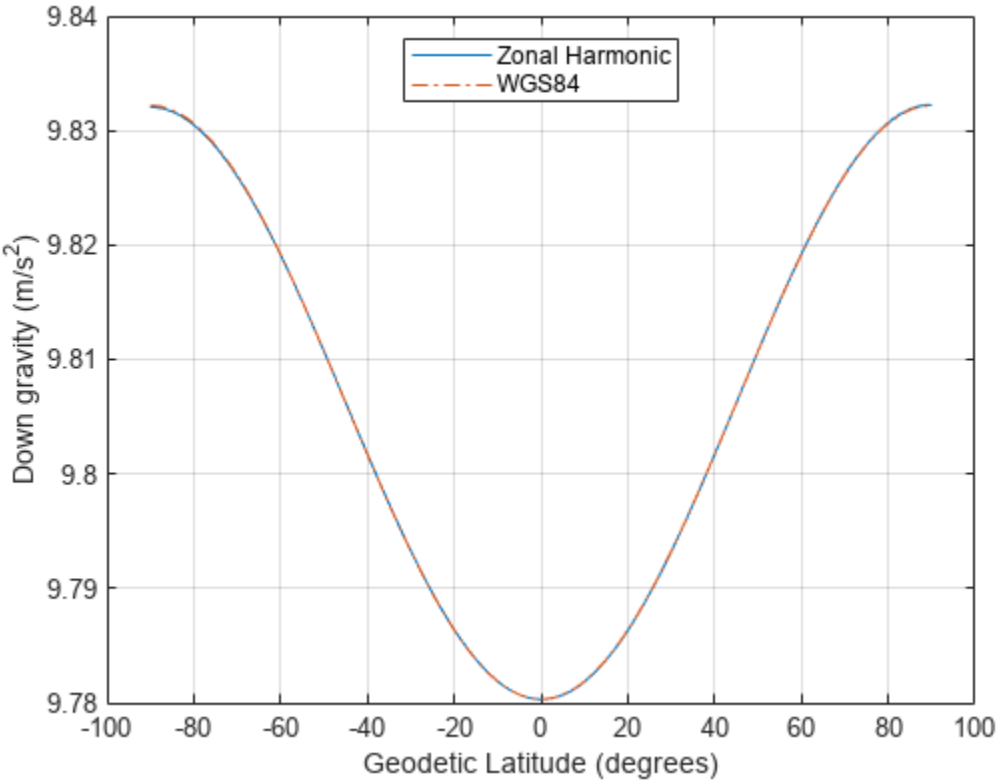


Figure 4: Gravity with centrifugal effects in the Down-axis in meters per second squared

```
figure(5)
plot( lat_gd, gned_cent_zonal(:,1), lat_gd, gn_cent_wgs84, '--', lat_gd, (gned_zonal(:,1)-gn_wgs8
```

```
axis([-100 100 -0.0002 0.0002])
legend('Zonal Harmonic', 'WGS84', 'Error Between Models w/o Centrifugal Effects', 'Location','Bes
xlabel('Geodetic Latitude (degrees)')
ylabel('North gravity (m/s^2)')
grid on
```

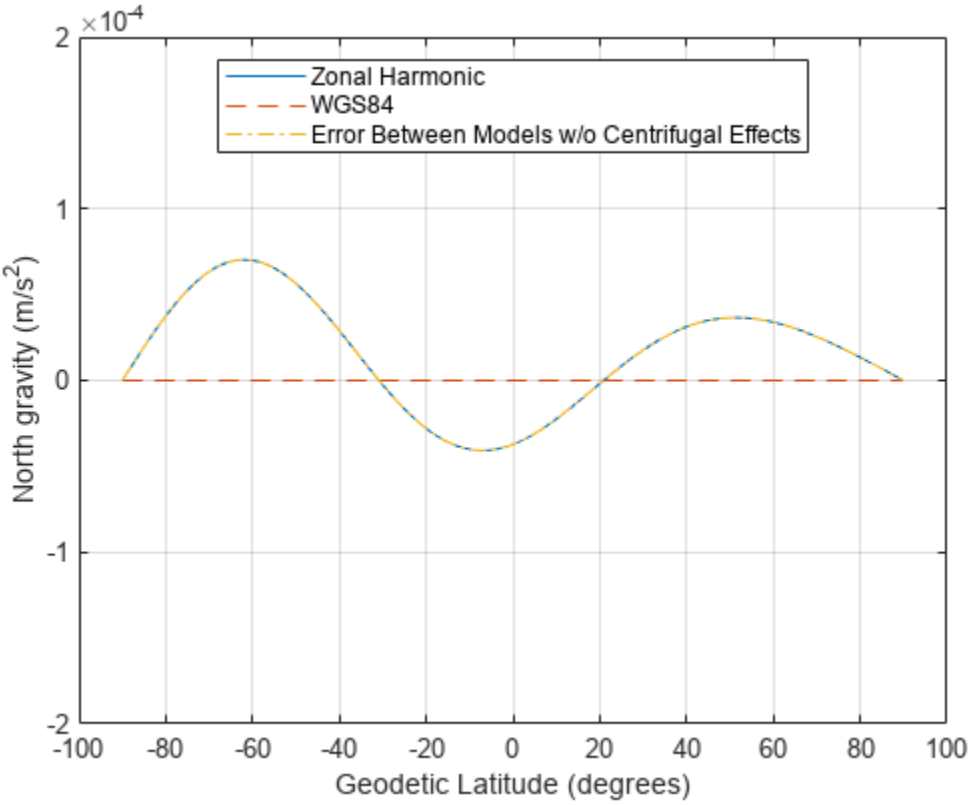


Figure 5: Gravity in the North-axis in meters per second squared

```
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 on
```

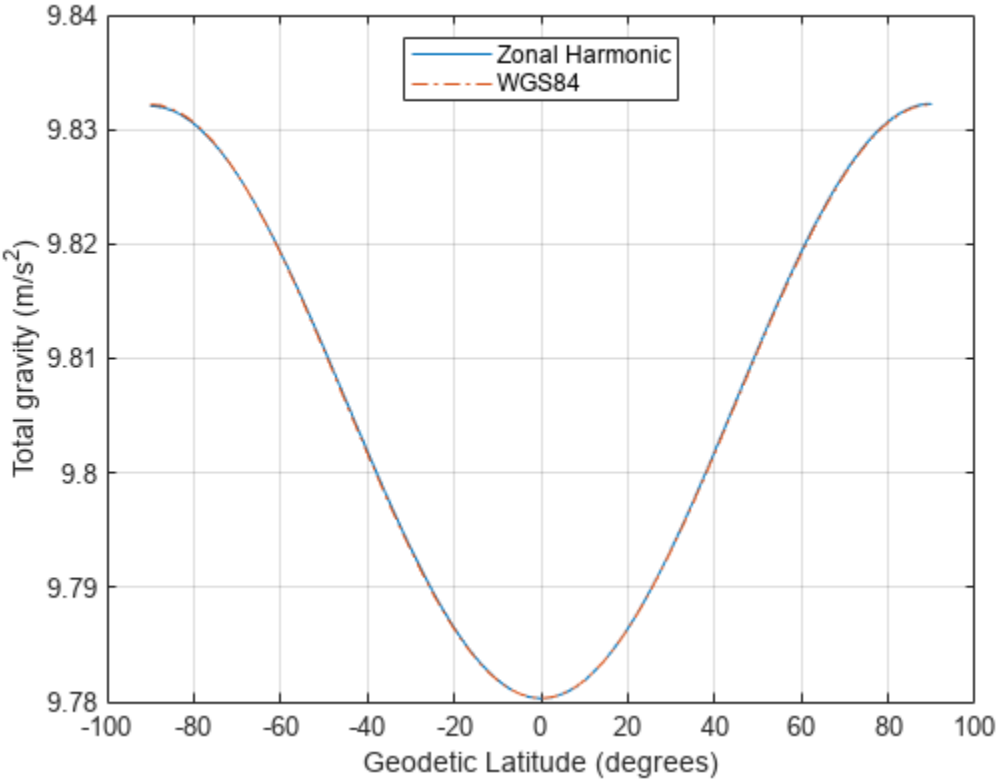


Figure 6: Total gravity with centrifugal effects in meters per second squared

See Also

`gravitycentrifugal` | `gravitywgs84` | `gravityzonal`

Calculating Compressor Power Required in a Supersonic Wind Tunnel

This example shows how to calculate the required compressor power in a supersonic wind tunnel.

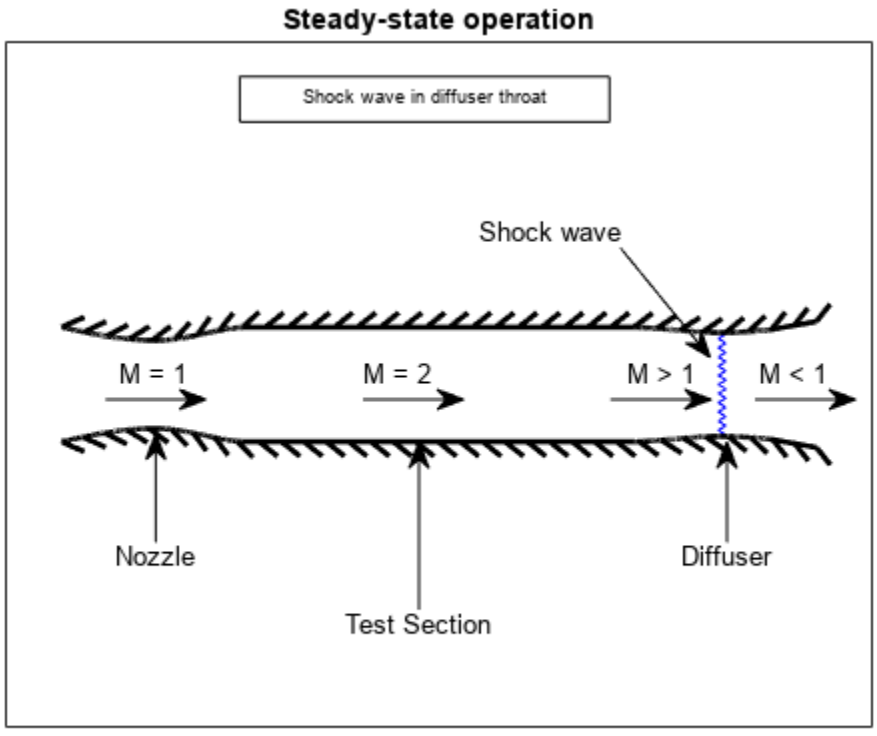
Problem Definition

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 = plotSupersonicWindTunnel('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] = atmosisa(he
```

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 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:

$$\dot{m} = \rho AV = \rho AMa$$

where all flow quantities are the values in the test section:

$\dot{m} = Mass\ flow\ rate$

$\rho = Density$

$A = Cross - sectional\ area\ of\ test\ section$

$V = Velocity$

$M = Mach\ number$

$a = Local\ speed\ of\ sound$

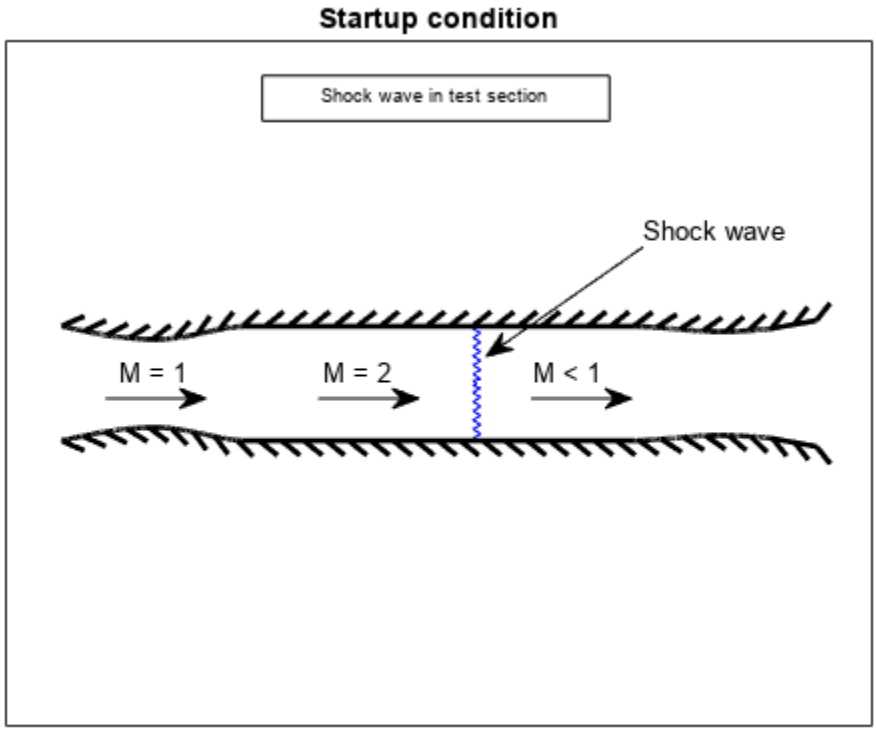
```
massFlowRate = testSectionDensity * testSectionArea * testMach * testSectionSpeedOfSound; % [kg /
```

Finally, calculate the power required by the compressor during steady-state operation.

```
powerSteadyState = specificWork * massFlowRate;  % [kW]
```

Calculating Work and Power Required During Startup

```
startupPicture = plotSupersonicWindTunnel('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.

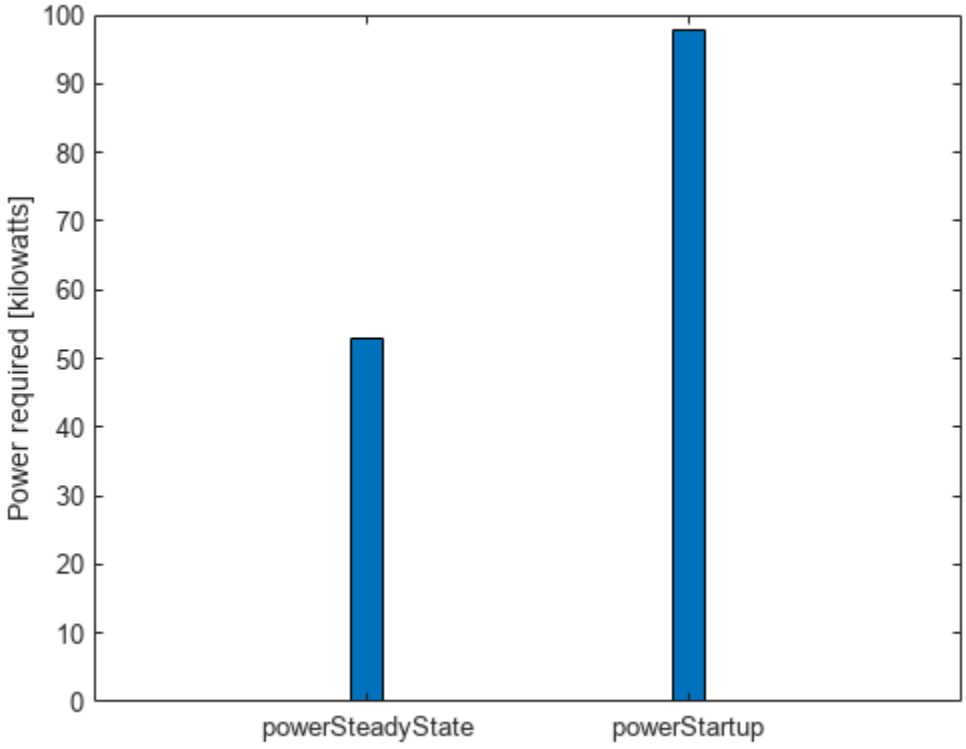
```
specificWorkStartup = cp * testSectionStagTemp * ((1 / stagPressRatioStartup)^((k - 1) / k) - 1)
```

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'})
```



Reference

[1] James, J. E. A., "Gas Dynamics, Second Edition", Allyn and Bacon, Inc, Boston, 1984.

See Also

`atmosisa` | `flowfanno` | `flowisentropic` | `flownormalshock` | `flowprandtlmeyer`

Related Examples

- “Analyzing Flow with Friction Through an Insulated Constant Area Duct” on page 5-52
- “Determine Heat Transfer and Mass Flow Rate in a Ramjet Combustion Chamber” on page 5-57
- “Solving for the Exit Flow of a Supersonic Nozzle” on page 5-63

Analyzing Flow with Friction Through an Insulated Constant Area Duct

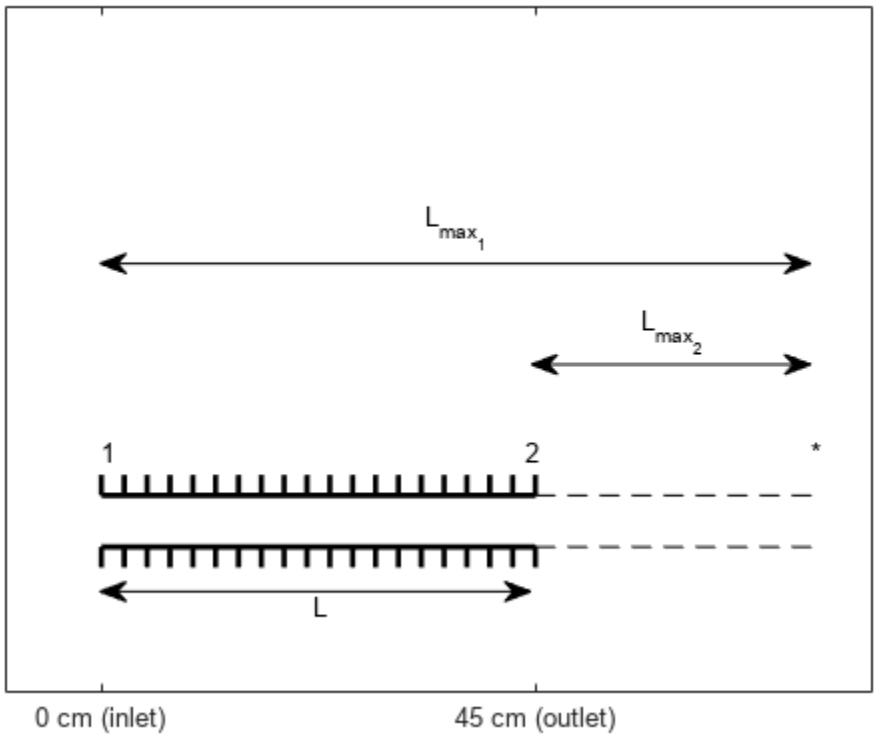
This example shows how to implement a steady, viscous flow through an insulated, constant-area duct using Aerospace Toolbox software. This flow is also called Fanno line flow.

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 = plotFannoFlowDuct;
```



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 the inlet pressure ratio. Use the `flowfanno` function:

```
[~, 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 want to calculate are

```
outletMach          % [dimensionless]
```

```
outletMach = 0.7093
```

```
outletTemperature   % [K]
```

```
outletTemperature = 292.2018
```

```
outletPressure      % [kPa]
```

```
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.

Reference

[1] James, J. E. A., "Gas Dynamics, Second Edition", Allyn and Bacon, Inc, Boston, 1984.

See Also

`flowfanno` | `flowisentropic` | `flownormalshock` | `flowprandtlmeyer`

Related Examples

- “Calculating Compressor Power Required in a Supersonic Wind Tunnel” on page 5-46
- “Determine Heat Transfer and Mass Flow Rate in a Ramjet Combustion Chamber” on page 5-57

• “Solving for the Exit Flow of a Supersonic Nozzle” on page 5-63

Determine Heat Transfer and Mass Flow Rate in a Ramjet Combustion Chamber

This example shows how to use 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, 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

$\dot{m} = Mass\ flow\ rate\ [kg/s]$

$V = Velocity\ [m/s]$

$p = pressure\ [kPa]$

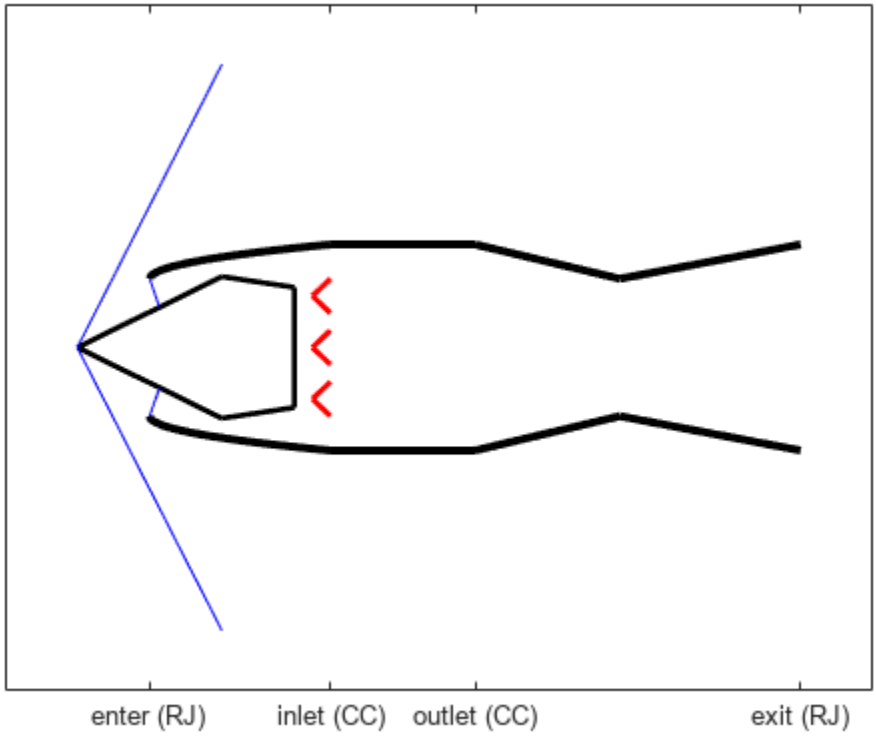
$A = Cross - sectional\ area\ [m^2]$

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 = plotRamjetSchematic;
```



Note, the thrust equation directly takes 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 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" indicates 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, making the thrust decrease. As a result, 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 you maximize fuel efficiency around the design conditions of an engine.

Reference

[1] James, J. E. A., "Gas Dynamics, Second Edition", Allyn and Bacon, Inc, Boston, 1984.

See Also

`flowfanno` | `flowisentropic` | `flownormalshock` | `flowprandtlmeyer`

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.

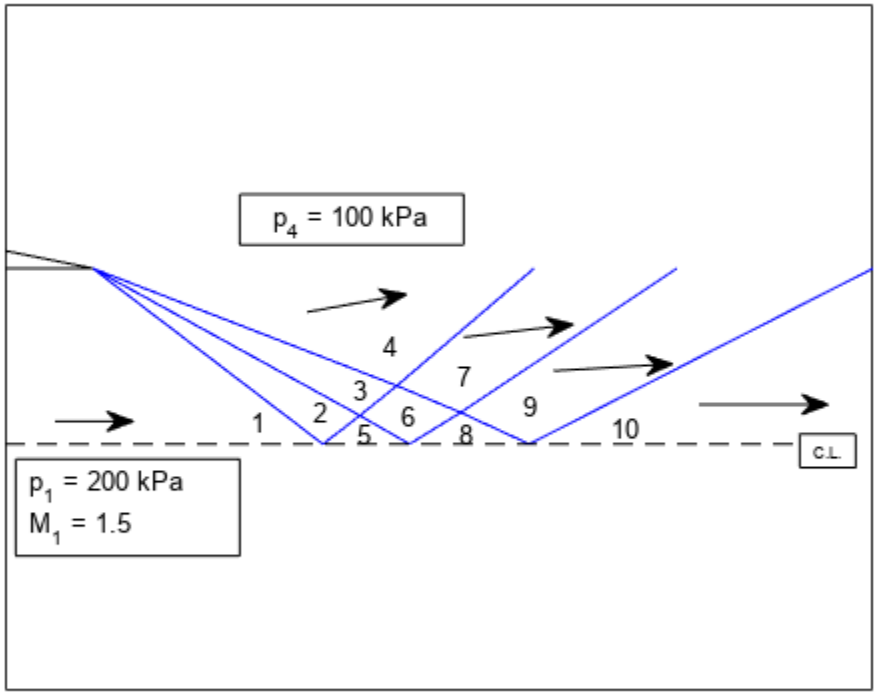
Solve for the flow field downstream of a supersonic nozzle using the method of characteristics. The Mach number at the exit plane is 1.5 and the pressure at the exit plane is 200 kilopascals. The back pressure is 100 kilopascals.

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 = plotExpansionSchematic('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.

The blue lines in the previous figure 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 location 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, 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 or 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 equals 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,

$$\Delta\theta_I = \Delta\nu_I = \theta_6 - \theta_5 = \nu_6 - \nu_5$$

Rearranging this as:

$$\nu_6 - \theta_6 = \nu_5 - \theta_5 \quad (1)$$

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:

$$\nu_6 + \theta_6 = \nu_3 + \theta_3 \quad (2)$$

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 as 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; the

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(size(flowAngles));
machAngles  = zeros(size(flowAngles));

for i = 1:numel(flowAngles)
    [machNumbers(i), ~, machAngles(i)] = flowprandtlmeyer(k, prandtlMeyerAngles(i), 'nu');
end

typeOne = flowAngles - machAngles;
typeTwo = flowAngles + machAngles;

m = (1:numel(machNumbers)).';
disp(table(m,round(flowAngles,2),round(prandtlMeyerAngles,2),round(machNumbers,3),round(machAngle
    'VariableNames',["Region","theta (Deg)","nu (Deg)","Mach","mu (Deg)","type I (Deg)","type II
```

Region	theta (Deg)	nu (Deg)	Mach	mu (Deg)	type I (Deg)	type II (Deg)
1	0	11.91	1.5	41.81	-41.81	41.81
2	4.45	16.35	1.65	37.29	-32.85	41.74
3	8.89	20.8	1.803	33.7	-24.8	42.59
4	13.34	25.24	1.959	30.69	-17.35	44.03
5	0	20.8	1.803	33.7	-33.7	33.7
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
8	0	29.69	2.122	28.11	-28.11	28.11
9	4.45	34.14	2.294	25.84	-21.4	30.29
10	0	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 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 the 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, separating 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 reference 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

$$m_{i,j} = \textit{Slope of the boundary between region i and region j}$$

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.

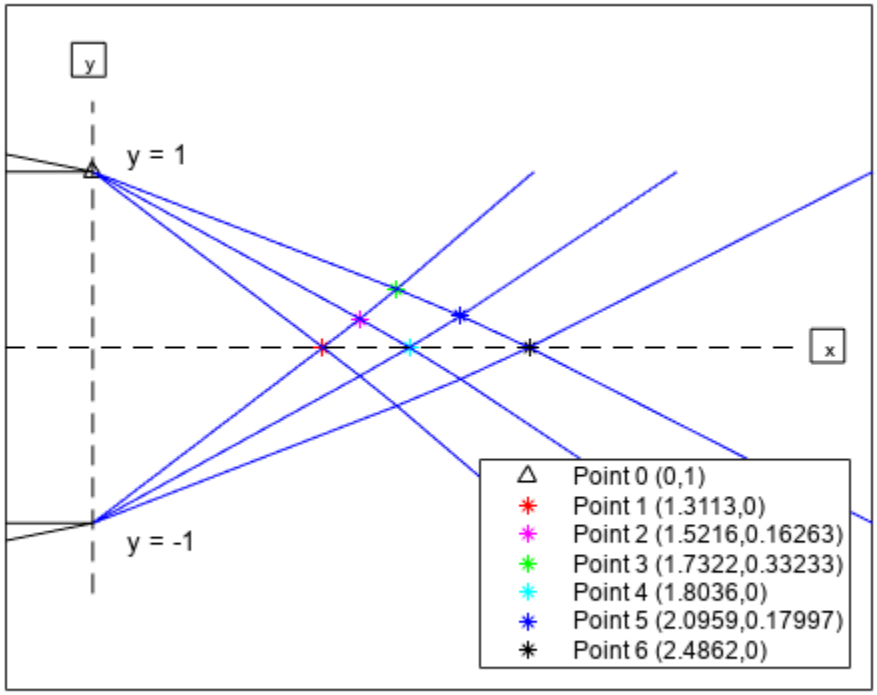
```
slope89 = tand( (typeOne(8) + typeOne(9)) / 2);
y6      = 0;
x6      = (slope89 * x5 - y5) / slope89;
```

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 = plotExpansionSchematic('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.

Reference

[1] James, J. E. A., "Gas Dynamics, Second Edition", Allyn and Bacon, Inc, Boston, 1984.

See Also

`flowfanno` | `flowisentropic` | `flownormalshock` | `flowprandtlmeyer`

Related Examples

- “Calculating Compressor Power Required in a Supersonic Wind Tunnel” on page 5-46
- “Analyzing Flow with Friction Through an Insulated Constant Area Duct” on page 5-52
- “Determine Heat Transfer and Mass Flow Rate in a Ramjet Combustion Chamber” on page 5-57

Visualize World Magnetic Model Contours for 2020 Epoch

This example shows how to visualize contour plots of the calculated values for the Earth's magnetic field using World Magnetic Model 2020 (WMM2020) overlaid on maps of the Earth.

The Mapping Toolbox™ is required to generate the maps of the Earth.

Generate Values for Earth's Magnetic Field

Calculate these values for the Earth magnetic field. To implement the World Magnetic Model 2020 (WMM2020), use the `wrldmagm` function.

- 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 these `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 = '2020';
decimal_year = 2020;
```

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 2020 and decimal year 2020.

Assume zero height.

```
% height = 0;
```

Geodetic Longitude value in degrees to use for latitude sweep.

```
% geod_lon = -180:1:180;
```

Geodetic Latitude values in degrees to sweep.

```
% geod_lat = -89.5:.5:89.5;
```

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
```

To obtain magnetic parameters for each latitude and longitude value, use the `wrldmagm` function.

```
%     [xyz, h, dec, dip, f] = wrldmagm(height, geod_lat(latIdx),geod_lon(lonIdx), decimal_year, r
```

Store the results.

```
%     WMMResults(latIdx,1:7,lonIdx) = [xyz' h dec dip f];
%
%     end
% end
```

Load the saved data in the MAT-file.

```
WMMFileName = 'astWMMResults_Epoch_2020_decyear_2020.mat';
load(WMMFileName);
```

Read in the continent land areas for plot overlay using the 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_year, model_e
```

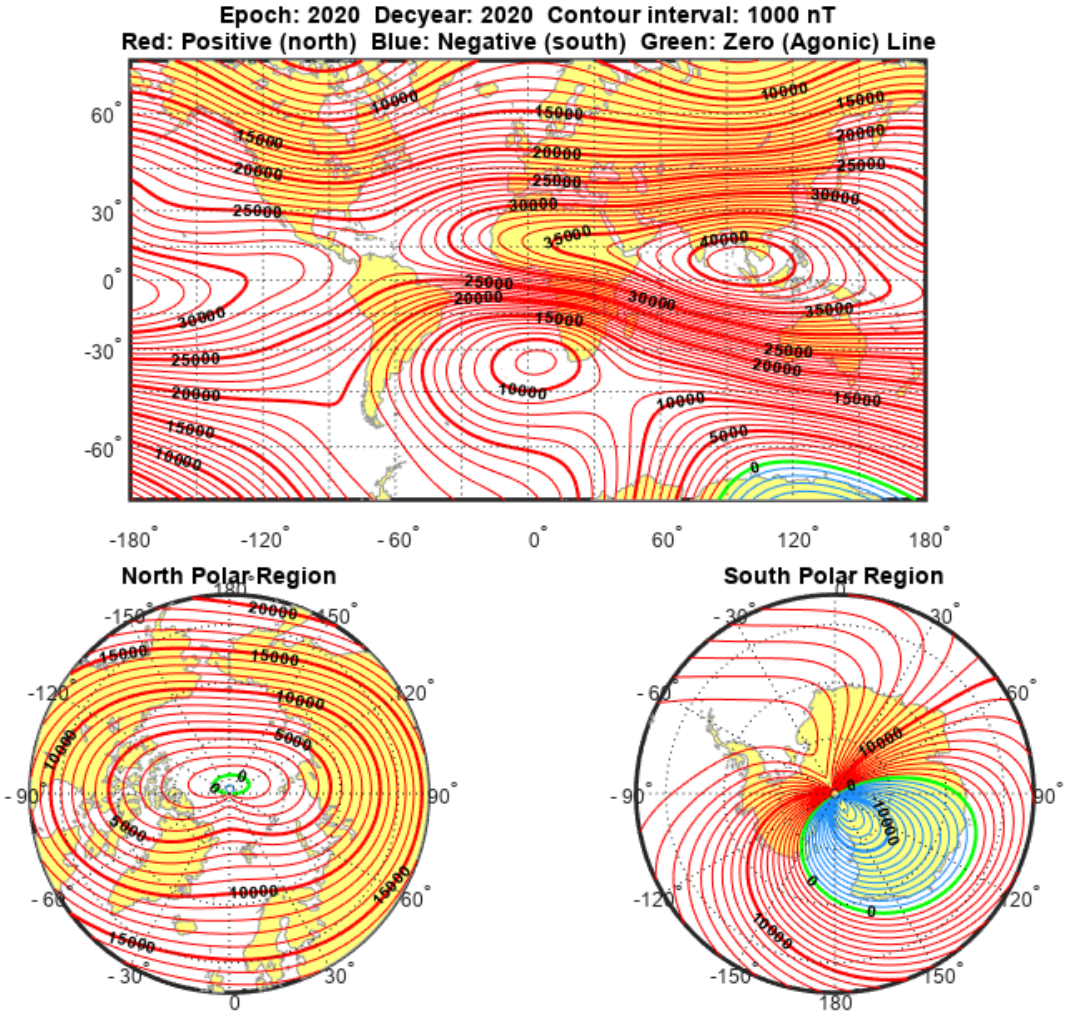


Figure 1: North Component of Magnetic Field Vector, X (nT)

```
hY = figure;
set(hY,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 2, landAreas, geod_lat, geod_lon, decimal_year, model_e
```

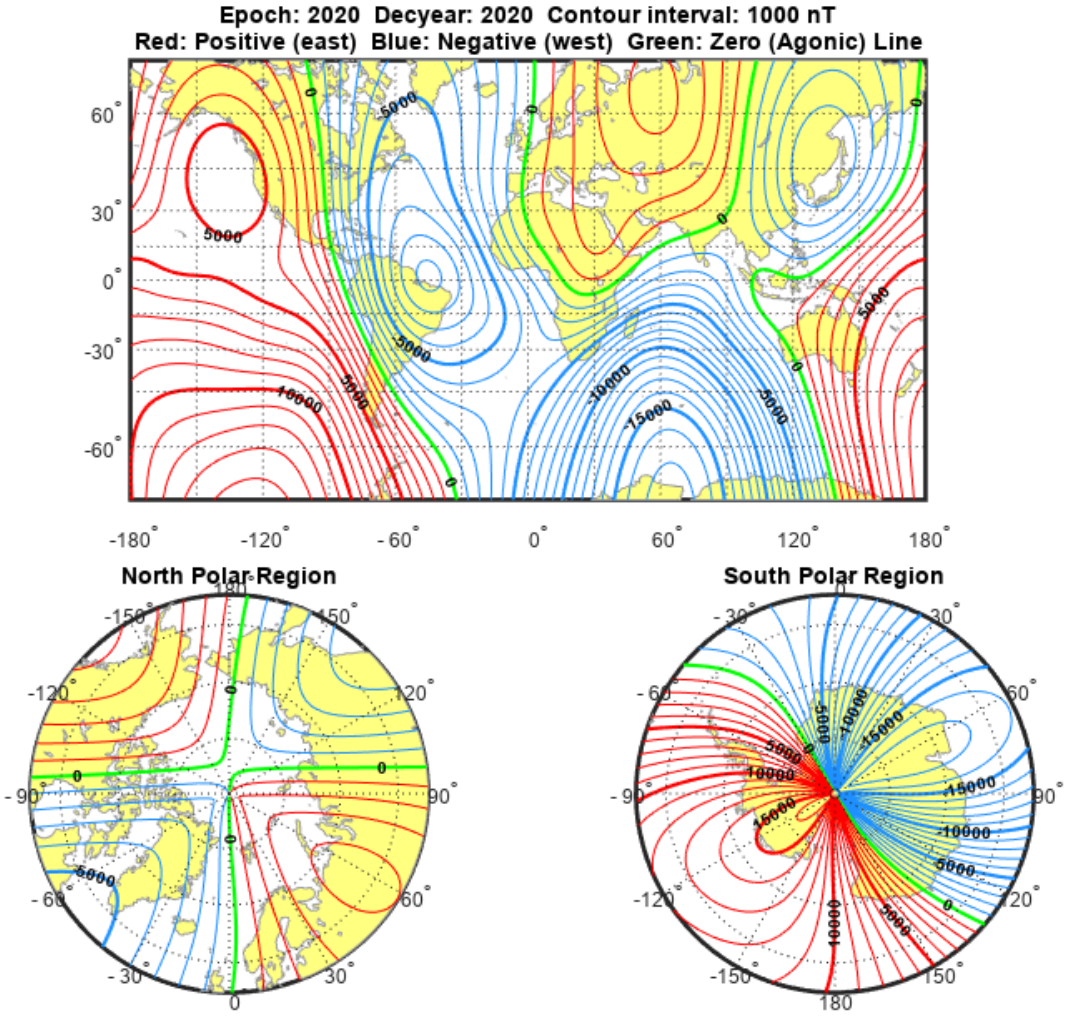


Figure 2: East Component of Magnetic Field Vector, Y (nT)

```
hZ = figure;
set(hZ,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 3, landAreas, geod_lat, geod_lon, decimal_year, model_e
```

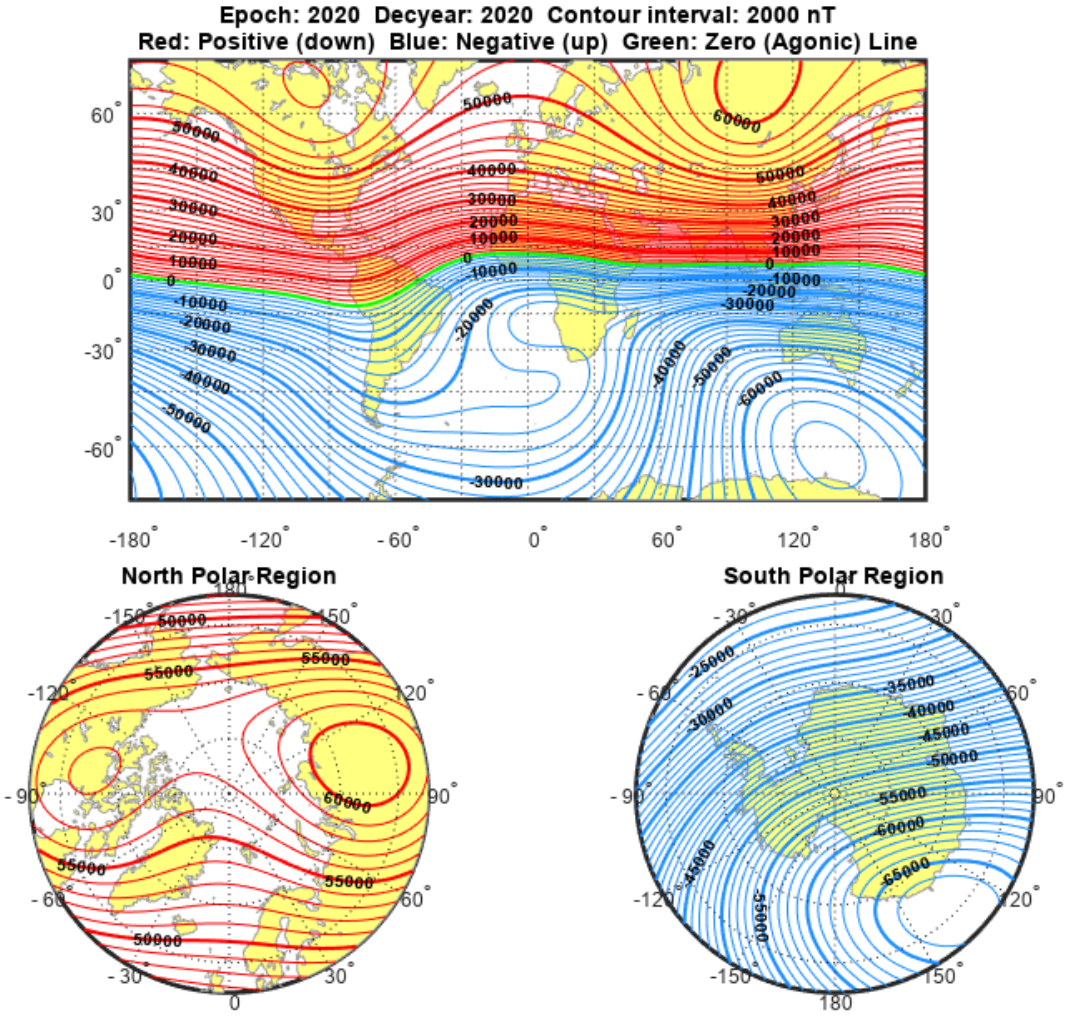


Figure 3: Down Component of Magnetic Field Vector, Z (nT)

```
hH = figure;
set(hH,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 4, landAreas, geod_lat, geod_lon, decimal_year, model_ep
```

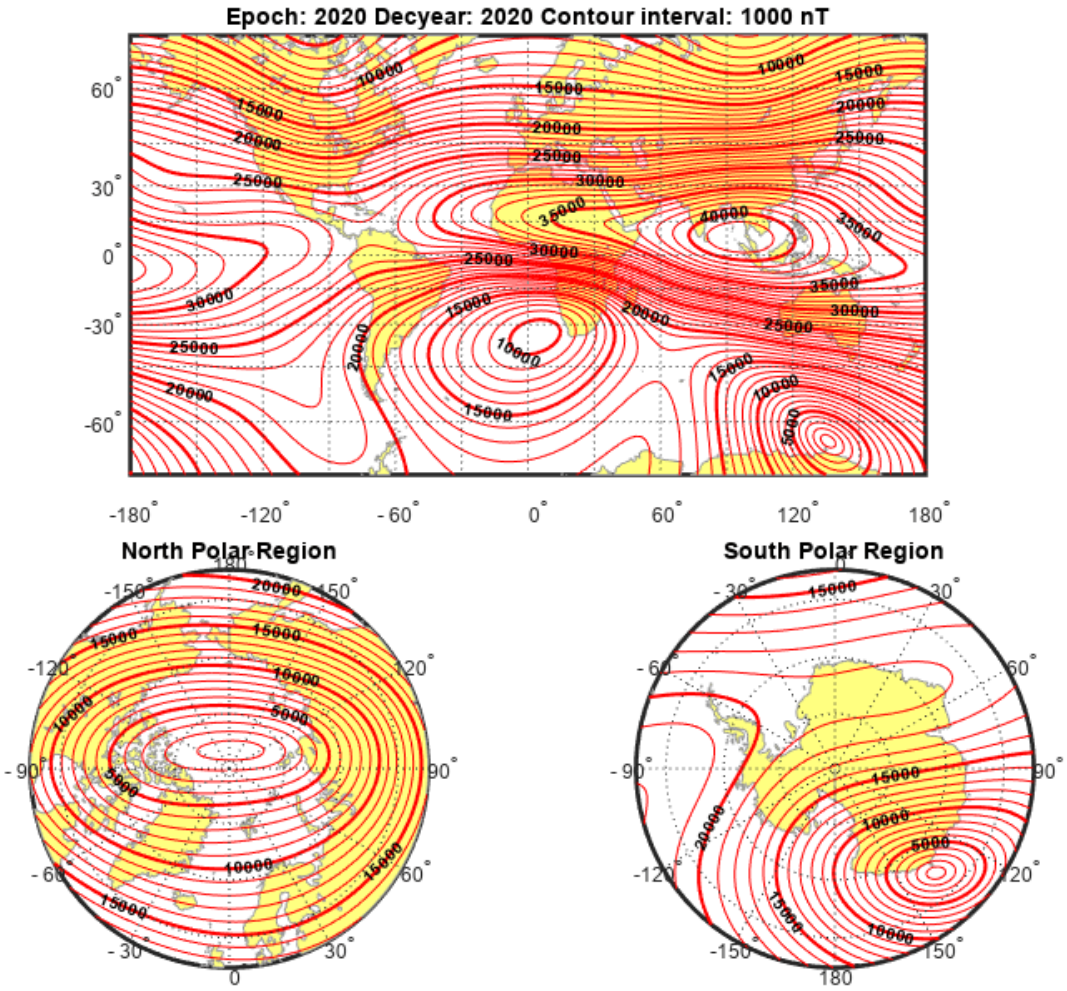


Figure 4: Horizontal Intensity, H (nT)

```
hDEC = figure;
set(hDEC,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 5, landAreas, geod_lat, geod_lon, decimal_year, model_e
```

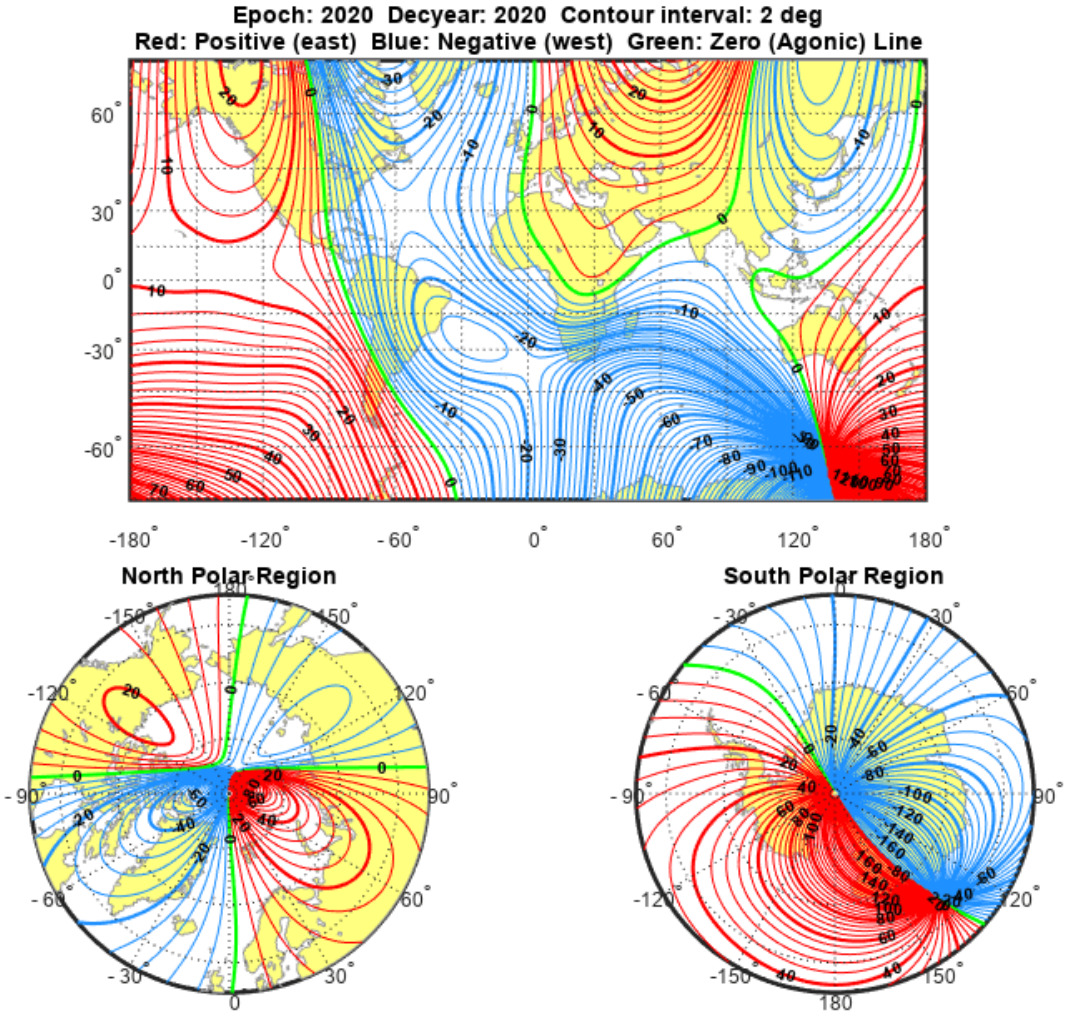


Figure 5: Declination, DEC (deg)

```
hDIP = figure;
set(hDIP,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 6, landAreas, geod_lat, geod_lon, decimal_year, model_ep
```

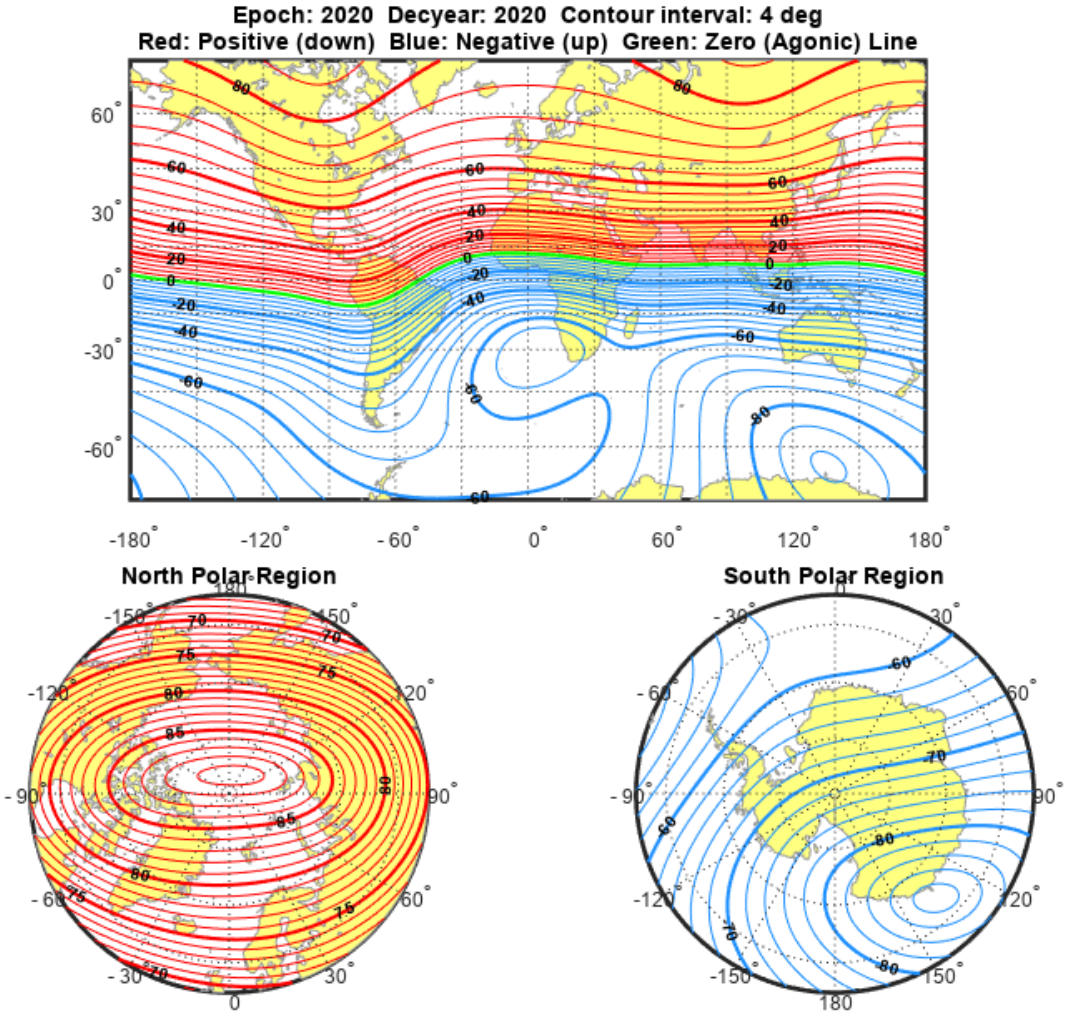


Figure 6: Inclination, DIP (deg)

```
hF = figure;
set(hF,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 7, landAreas, geod_lat, geod_lon, decimal_year, model_e
```

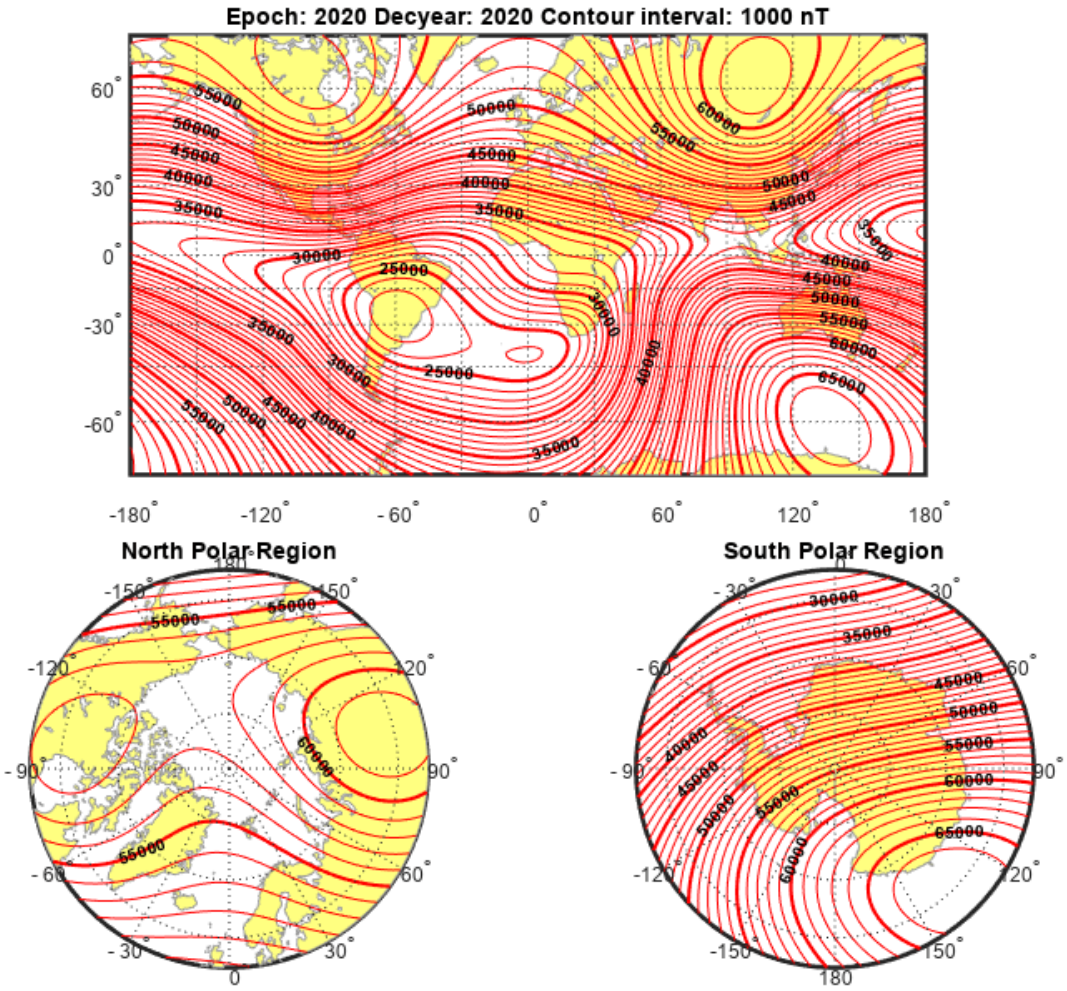


Figure 7: Total Intensity, F (nT)

```
close (hX, hY, hZ, hH, hDEC, hDIP, hF)
```

See Also

`wrldmagm`

Visualizing Geoid Height for Earth Geopotential Model 1996

This example shows how to calculate the Earth geoid height using the EGM96 Geopotential Model of Aerospace Toolbox software. It also shows how to visualize the results with contour maps overlaid on maps of the Earth. Mapping Toolbox™ and Simulink® 3D Animation™ are required to generate the visualizations.

Generating Values for Earth Geopotential Model 1996

To implement the EGM96 Geopotential Model, calculate values for the Earth geopotential using the `geoidheight` function.

Use the following code to generate 260281 data points for calculating values of the Earth 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

% Longitude value in degrees to use for latitude sweep.
lon = -180:gridDegInc:180;          %degrees

% Geodetic Latitude values to sweep.
geod_lat = -90:gridDegInc:90;       %degrees

% Loop through longitude values for each array of latitudes -90:90.
for lonIdx = size(lon,2):-1:1

    % Longitude must be the same dimension as the latitude array
    lon_temp = lon(lonIdx)*ones(1,numel(geod_lat));      % degrees
    geoidResults(:,lonIdx) = geoidheight(geod_lat,lon_temp,'None');

end
```

Loading Geoid Data File and Coastal Data

```
geoidFileName = 'GeoidResults_05deg_180.mat';
geoidData = load(geoidFileName);
coast = load('coastlines.mat');
```

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 = georefcells([-90, 90], [-180, 180], size(geoidData.geoidResults));
ast2DGeoidPlot(RRR,geoidData.geoidResults,coast,geoidData.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.

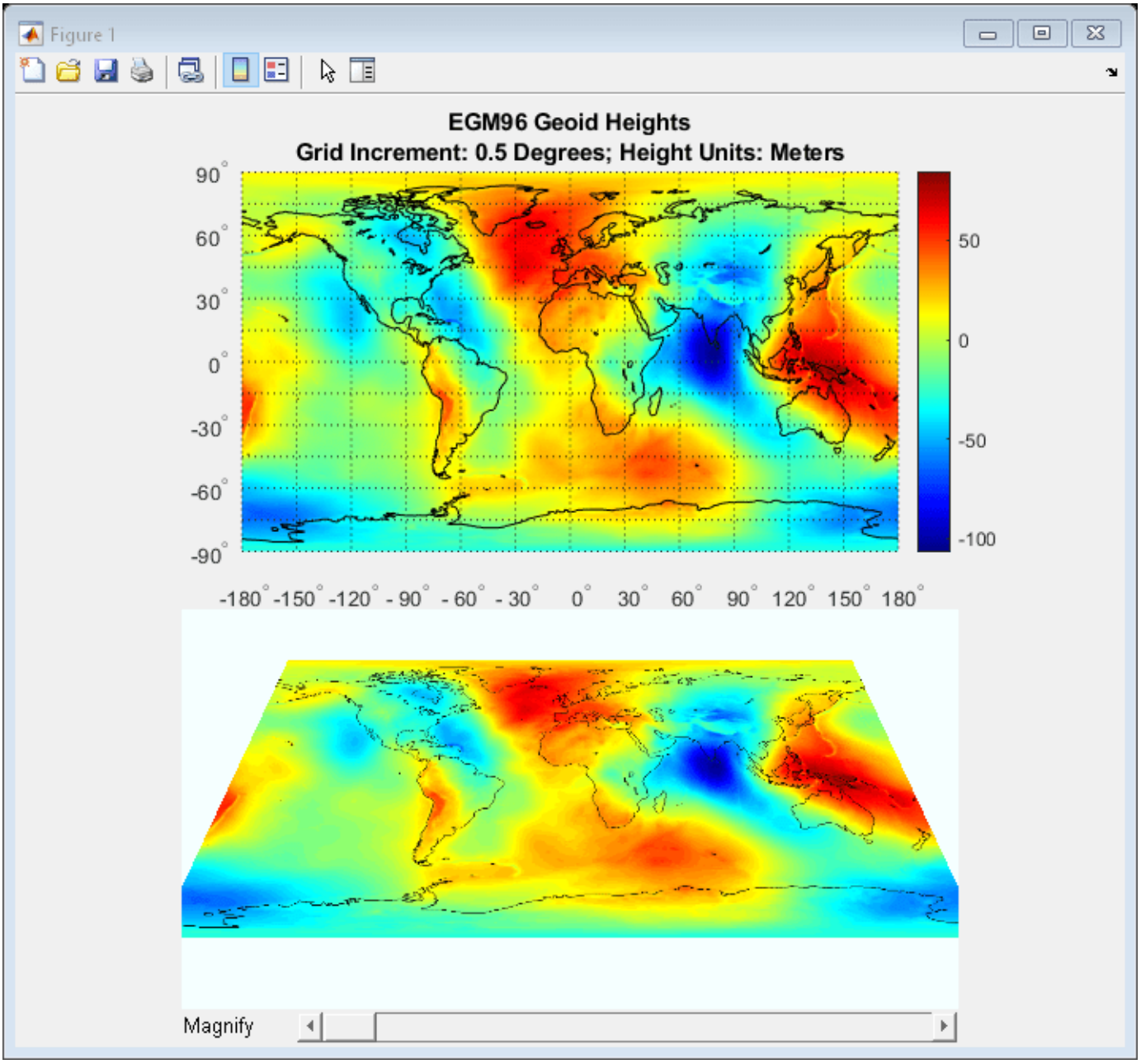
```
geoidGrid.height = 0*actualHeights;
```

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);
```



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(geoidData.geoidResults,RRR)
```

Plot land mass outline.

```
plotm(coast.coastlat,coast.coastlon,'Color','k')
colormap('jet');
```

Plot Title.

```
title({'EGM96 Geoid Heights';['Grid Increment: ' ,num2str(geoidData.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]);
```

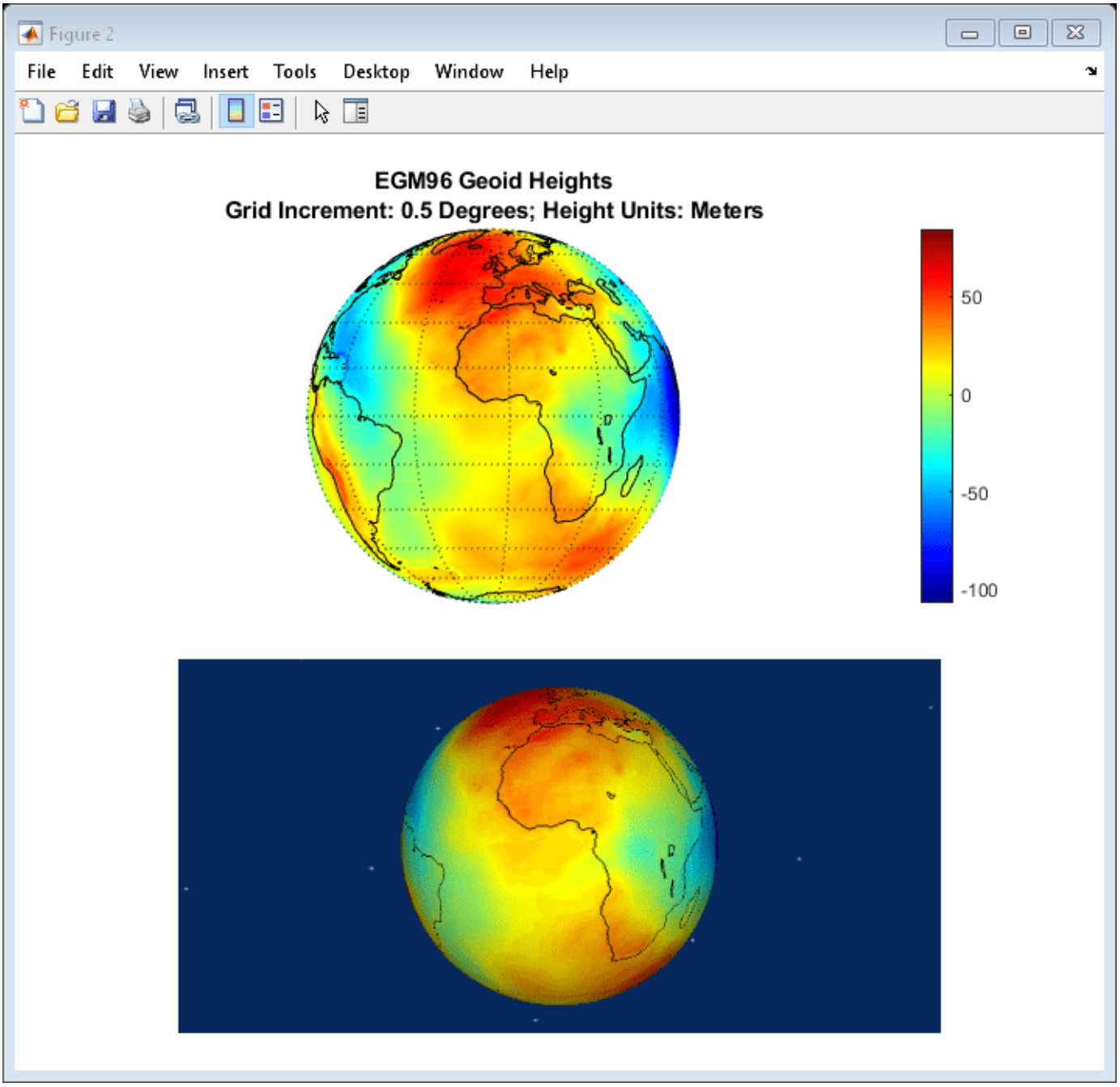


Figure 2
File Edit View Insert Tools Desktop Window Help
EGM96 Geoid Heights
Grid Increment: 0.5 Degrees; Height Units: Meters
50
0
-50
-100

```
vrdrawnow;
```

Clean Up

```
close(h2D,h3D)
close(www2D);close(www3D);
delete(www2D);delete(www3D);
```

See Also

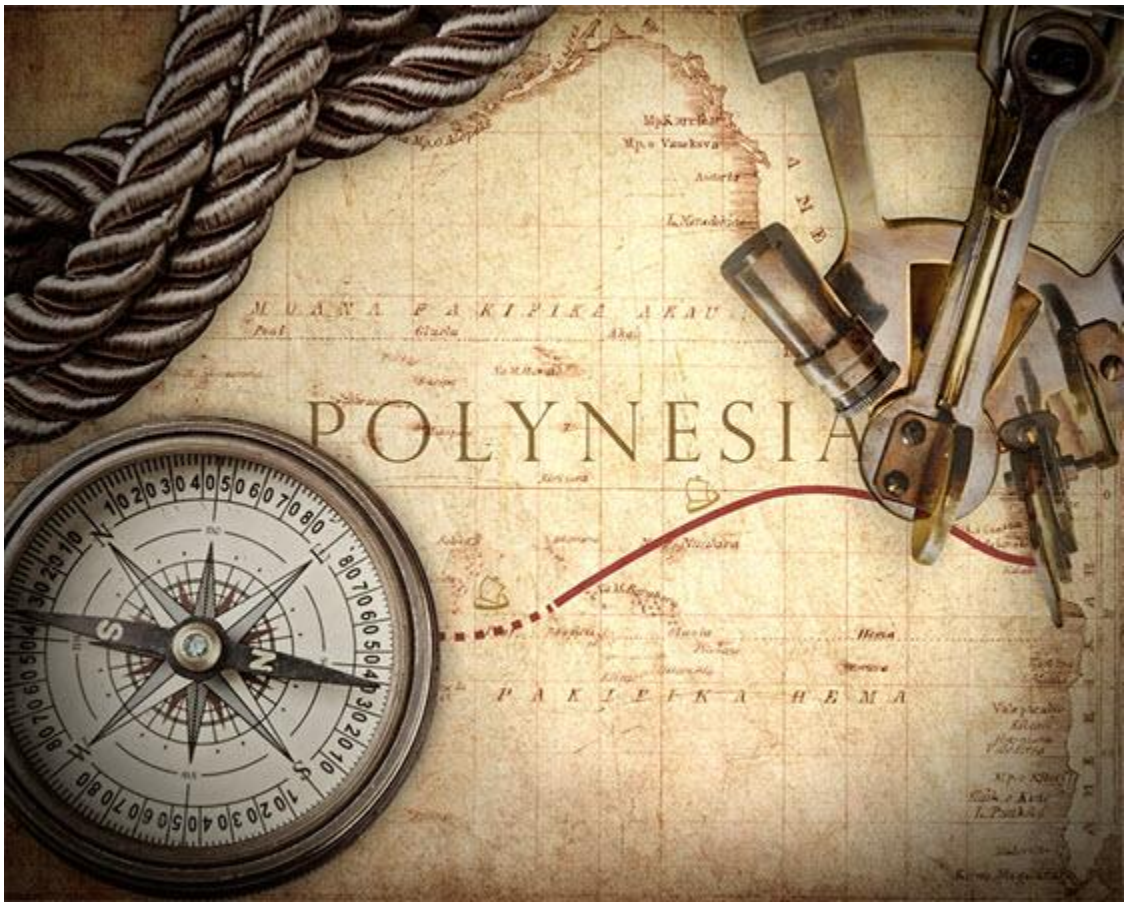
Classes
`vr.canvas` | `vrworld`

Functions
`geoidheight` | `vrnode`

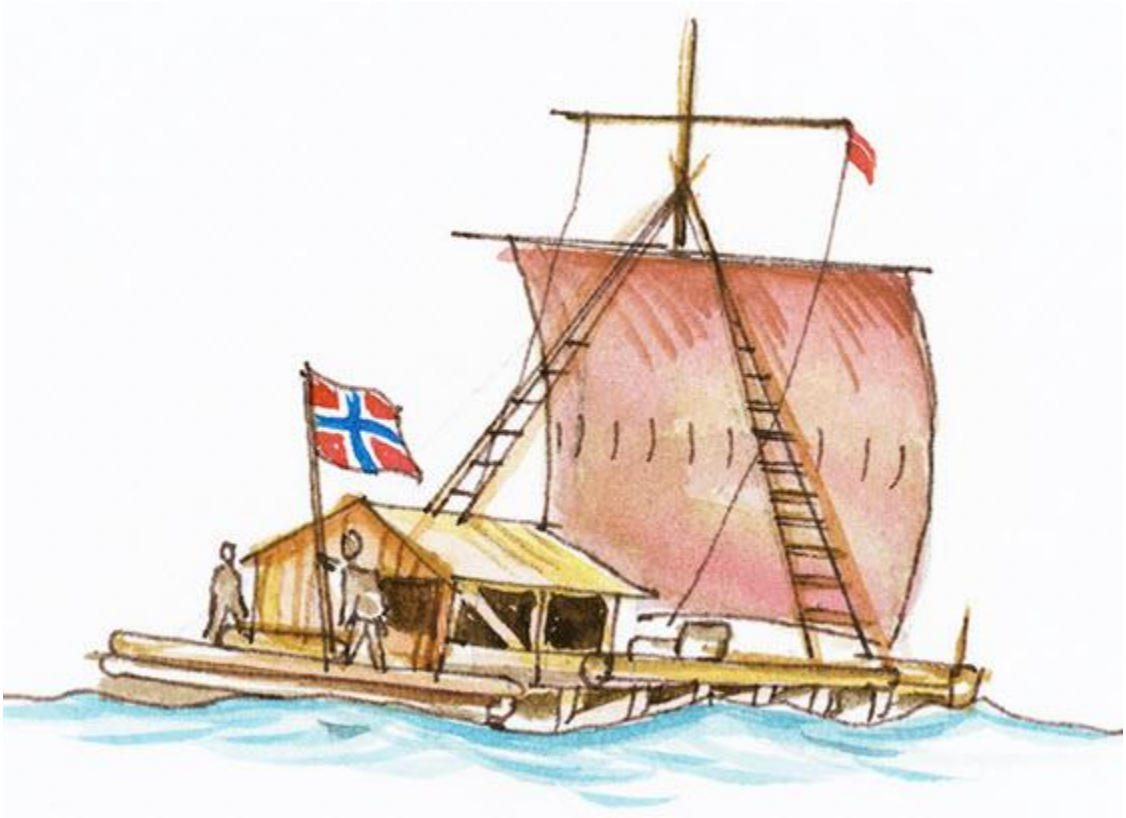
Marine Navigation Using Planetary Ephemerides

This example shows how to perform celestial navigation of a marine vessel using the planetary ephemerides and an Earth-Centered Inertial to Earth-Centered Earth-Fixed (ECI to ECEF) transformation.

This example uses the Mapping Toolbox™. You must also download data for the example using the *aeroDataPackage* function.



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. 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 the 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 these 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.

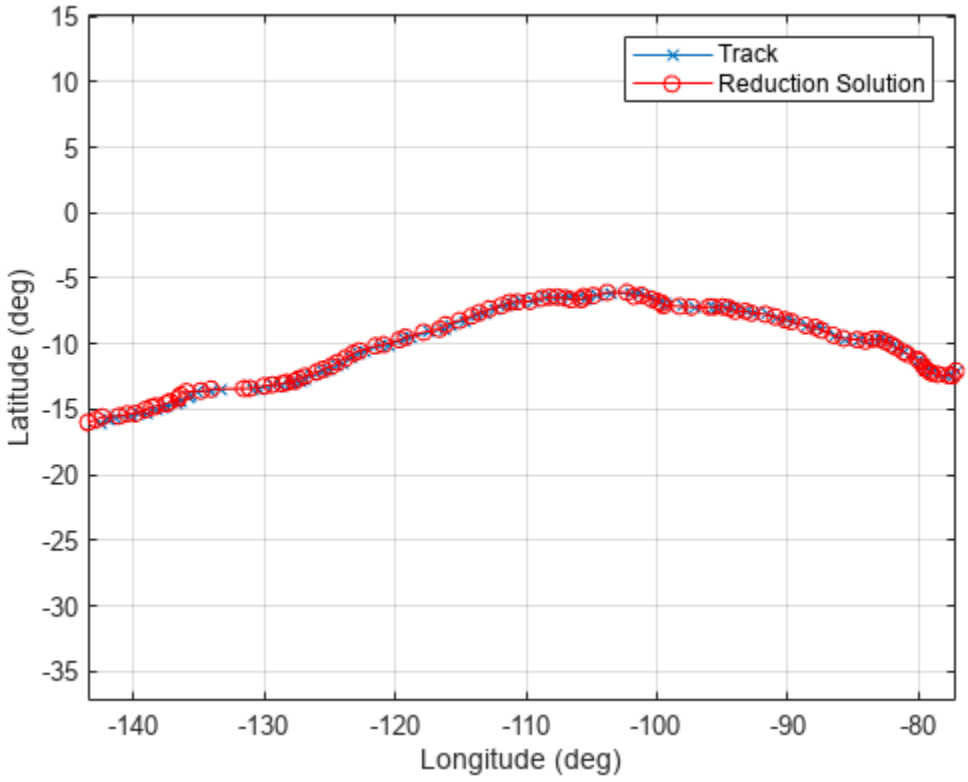
```
    longFix(m+1) = obs.longitude + deltaLongFix;
    latFix(m+1) = obs.latitude + deltaLatFix;
```

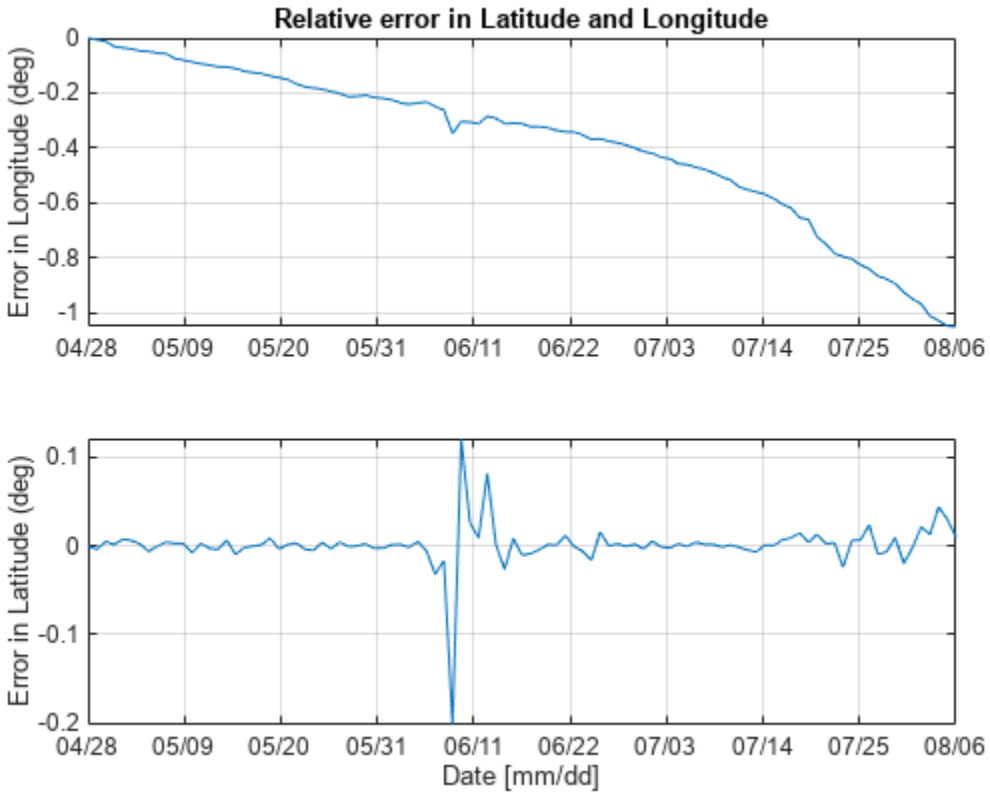
```
end
```

Navigation Solution Visualization

This figure shows the actual track and sight reduction solutions.

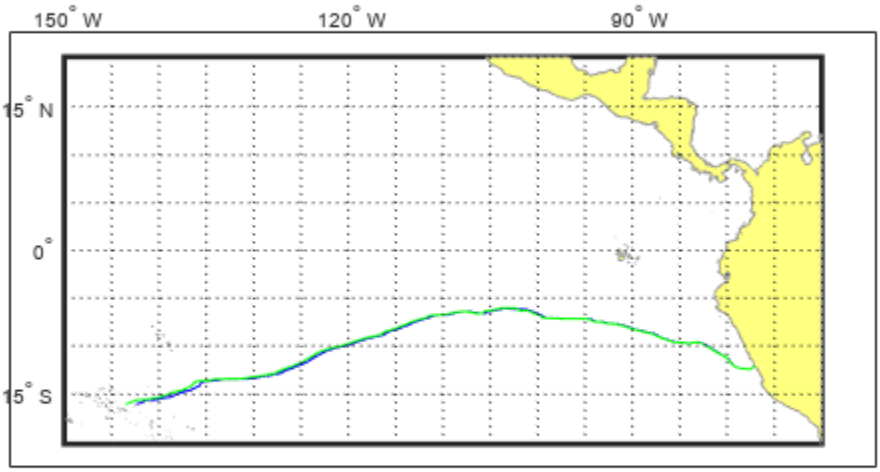
```
astHelperVisualization(long,lat,longFix,latFix,'Plot')
```





You can use the Mapping Toolbox to obtain a more detailed graph depicting the sight reduction solutions with the American continent and French Polynesia.

```
if builtin('license','test','MAP_Toolbox')
    astHelperVisualization(long,lat,longFix,latFix,'Map')
end
```



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.

See Also

`aeroDataPackage` | `planetEphemeris` | `juliandate` | `deltaUT1` | `dcmeci2ecef`

Related Examples

- “Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-96

Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation

This example shows how to estimate the analemma. 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* function.

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 from January 1, 2014 to December 31, 2014 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 *dvTT* array.

```
jdTDB = tdbjuliandate([dvTT.Year' dvTT.Month' dvTT.Day' dvTT.Hour' dvTT.Minute' dvTT.Second']);
```

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,:);
```

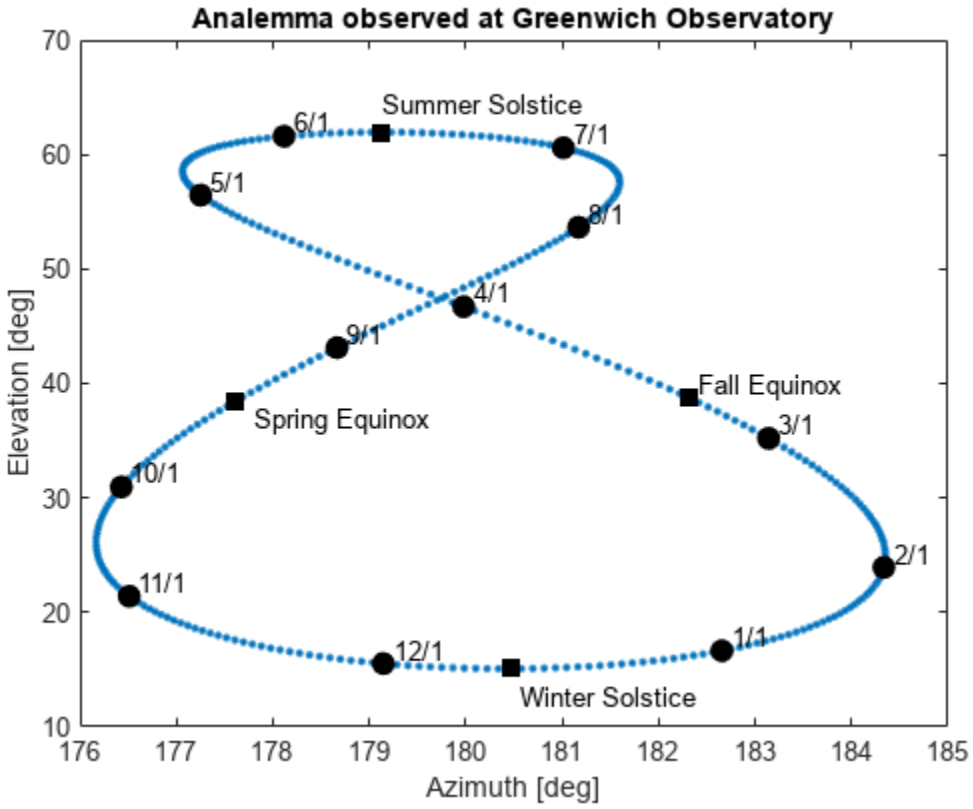
To get solstices and equinoxes (for 2014 are 3/20, 6/21, 9/22, 12/21):

```
solsticeEquinox = [ aer(dvUTC(:,2)==3 & dvUTC(:,3)==20, 1:2); ...
                    aer(dvUTC(:,2)==6 & dvUTC(:,3)==21, 1:2); ...
                    aer(dvUTC(:,2)==9 & dvUTC(:,3)==22, 1:2); ...
                    aer(dvUTC(:,2)==12 & dvUTC(:,3)==21, 1:2) ];
```

Plot Results

Plot the analemma. Along the analemma, plot points for the whole year, first days of the month, equinoxes, and solstices.

```
firstDays = (12:-1:1) + "/" + 1;

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','HorizontalAlignment
text(solsticeEquinox(1,1)+.2, solsticeEquinox(1,2)-1.5, 'Spring Equinox', 'Color', 'k','Horizonta
text(solsticeEquinox(2,1), solsticeEquinox(2,2)+2.5, 'Summer Solstice', 'Color', 'k','HorizontalA
text(solsticeEquinox(3,1)+.1, solsticeEquinox(3,2)+1.2, 'Fall Equinox', 'Color', 'k','HorizontalA
text(solsticeEquinox(4,1)+.1, solsticeEquinox(4,2)-2.5, 'Winter Solstice', 'Color', 'k','Horizont
```



See Also

`planetEphemeris` | `aeroDataPackage` | `deltaCIP` | `aer` | `eci2aer` | `polarMotion` | `tdbjuliandate`

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
yaw = simdata(:,7);
yaw(yaw<0) = yaw(yaw<0)+2*pi;
simdata(:,7) = yaw;
```

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.updateBodies(simdata(1,1));
h.updateCamera(simdata(1,1));
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)-502 572 502],
    'Color',[0.2667 0.2706 0.2784],'Resize','off');
```

To prevent the live script from adding a new image for each ui element added, set the visibility property to "off".

```
fig.Visible = "off";
```

Load panel image into an axis:

```
imgPanel = imread('FlightInstrumentPanel.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 = [25 250];
air.ScaleColorLimits = [0,60; 50,200; 200,225; 225,250];
```

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 = 8000;
```

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 flightInstrumentsAnimationCallback 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) flightInstrumentsAnimationCallback(fig,simdata,h,event);
```

To display the time selected in the slider, create a label component.

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

To display the figure, set the Visibility property to "on".

```
fig.Visible = "on";
```

See Also

Classes
`Aero.Animation`

Methods
`createBody` | `updateBodies` | `updateCamera`

Functions
`uifigure` | `uiaeroairspeed` | `uiaeroaltimeter` | `uiaeroclimb` | `uiaeroegt` | `uiaeroheading` | `uiaerohorizon` | `uiaerorpm` | `uiaeroturn`

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

Related Examples

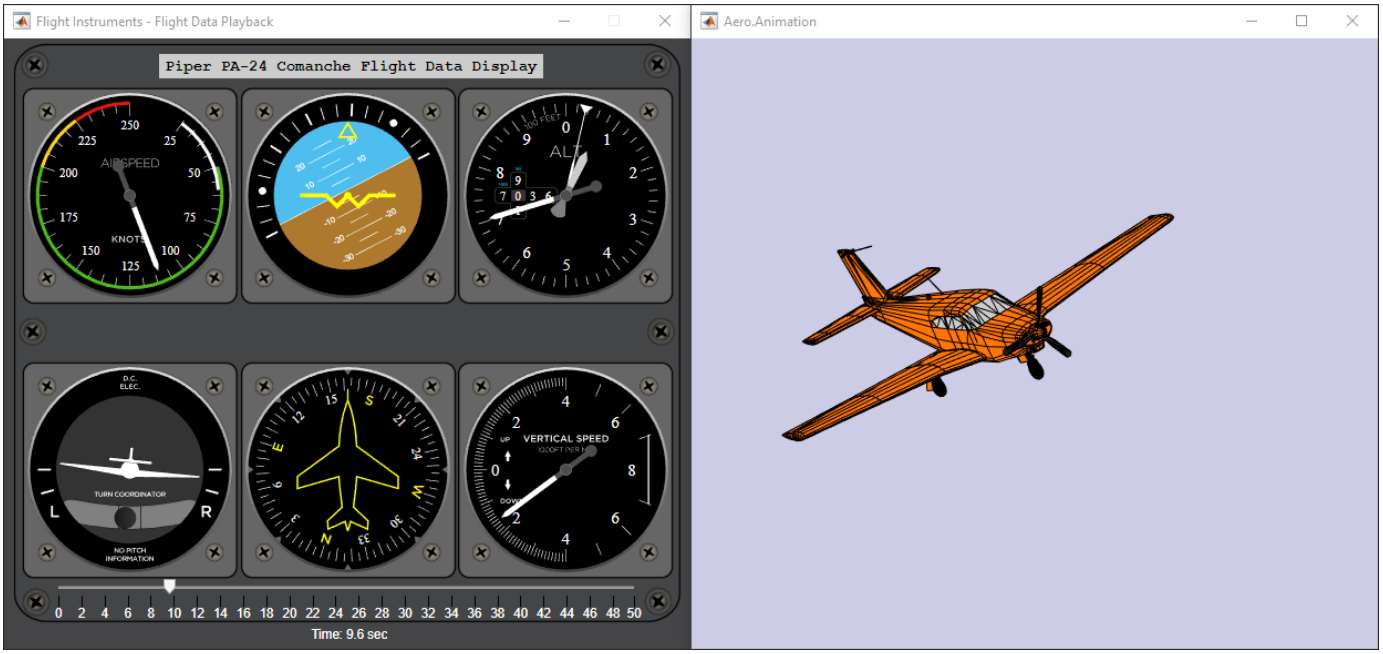
- “Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
- “Flight Instrument Components in App Designer” on page 2-54
- “Aerospace Flight Instruments in App Designer” on page 5-104

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.



See Also

Functions

`uifigure` | `uiaeroairspeed` | `uiaeroaltimeter` | `uiaeroclimb` | `uiaeroegt` | `uiaeroheading` | `uiaerohorizon` | `uiaerorpm` | `uiaeroturn`

Properties

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

Related Examples

- “Create and Configure Flight Instrument Component and an Animation Object” on page 2-51
- “Flight Instrument Components in App Designer” on page 2-54
- “Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-99

Analyze State-Space Model for Linear Control and Static Stability Analysis

This example shows how to convert a fixed-wing aircraft to a linear time invariant (LTI) state-space model for linear analysis.

This example describes:

- Importing and filling data from a DATCOM file.
- Constructing a fixed-wing aircraft from DATCOM data.
- Calculating static stability of the fixed-wing aircraft.
- Linearizing the fixed-wing aircraft around an initial state.
- Validating the static stability analysis with a dynamic response.
- Isolating the elevator-to-pitch transfer function and designing a feedback controller for the elevator.

Defining the Fixed-Wing Aircraft and State

This example uses a DATCOM file created for the Sky Hogg aircraft.

First, import the DATCOM output file using datcomimport.

```
allData = datcomimport('astSkyHoggDatcom.out', false, 0);
skyHoggData = allData{1}
```

```
skyHoggData = struct with fields:
          case: 'SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG'
          mach: [0.1000 0.2000 0.3000 0.3500]
           alt: [1000 3000 5000 7000 9000 11000 13000 15000]
         alpha: [-16 -12 -8 -4 -2 0 2 4 8 12]
         nmach: 4
          nalt: 8
        nalpha: 10
         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: 1
       highasy: 0
       highcon: 0
          tjet: 0
```

```
      hypeff: 0
          lb: 0
         pwr: 0
        grnd: 0
       wsspn: 18.7000
       hsspn: 5.7000
      ndelta: 5
       delta: [-20 -10 0 10 20]
      deltal: []
      deltar: []
         ngh: 0
      grndht: []
      config: [1x1 struct]
     version: 1976
          cd: [10x4x8 double]
          cl: [10x4x8 double]
          cm: [10x4x8 double]
          cn: [10x4x8 double]
          ca: [10x4x8 double]
         xcp: [10x4x8 double]
         cma: [10x4x8 double]
         cyb: [10x4x8 double]
         cnb: [10x4x8 double]
         clb: [10x4x8 double]
         cla: [10x4x8 double]
       qqinf: [10x4x8 double]
         eps: [10x4x8 double]
    depsdalp: [10x4x8 double]
         clq: [10x4x8 double]
         cmq: [10x4x8 double]
        clad: [10x4x8 double]
        cmad: [10x4x8 double]
         clp: [10x4x8 double]
         cyp: [10x4x8 double]
         cnp: [10x4x8 double]
         cnr: [10x4x8 double]
         clr: [10x4x8 double]
     dcl_sym: [5x4x8 double]
     dcm_sym: [5x4x8 double]
  dclmax_sym: [5x4x8 double]
  dcdmin_sym: [5x4x8 double]
    clad_sym: [5x4x8 double]
     cha_sym: [5x4x8 double]
     chd_sym: [5x4x8 double]
    dcdi_sym: [10x5x4x8 double]
```

Next, prepare the DATCOM lookup tables.

DATCOM lookup tables might have missing values due to the tables only filling one value for the whole column.

This missing data is represented in the lookup tables as the value 99999 and can be filled using the "previous" method of fillmissing.

In this example, $C_{y\beta}$, $C_{n\beta}$, C_{Lq}, and C_{mq} have missing data.

```
skyHoggData.cyb = fillmissing(skyHoggData.cyb, "previous", "MissingLocations", skyHoggData.cyb ==
skyHoggData.cnb = fillmissing(skyHoggData.cnb, "previous", "MissingLocations", skyHoggData.cnb ==
skyHoggData.clq = fillmissing(skyHoggData.clq, "previous", "MissingLocations", skyHoggData.clq ==
skyHoggData.cmq = fillmissing(skyHoggData.cmq, "previous", "MissingLocations", skyHoggData.cmq ==
```

With the missing data filled, the fixed-wing aircraft can be constructed.

First, the fixed-wing aircraft is prepared with the desired aircraft name.

Optionally, the aircraft name can be extracted from the "case" field on the DATCOM struct by passing an empty fixed-wing object.

```
skyHogg = Aero.FixedWing();
skyHogg.Properties.Name = "Sky_Hogg";
skyHogg.DegreesOfFreedom = "3DOF";
[skyHogg, cruiseState] = datcomToFixedWing(skyHogg, skyHoggData);
```

The datcomToFixedWing will convert all compatible data from the datcom struct into the fixed-wing object and its state. However, the returned state still needs processing to get the desired initial conditions of the aircraft.

In this example, the environment, mass, inertia, airspeed, and center of pressure need adjusting.

```
h = 2000;
cruiseState.AltitudeMSL = h;
cruiseState.Environment = aircraftEnvironment(skyHogg,"ISA",h);

cruiseState.U = 169.42;
cruiseState.Mass = 1299.214;
cruiseState.Inertia.Variables = [5787.969 0 117.64;0 6928.93 0;-117.64 0 11578.329];
cruiseState.CenterOfPressure = [0.183, 0, 0];
```

Calculating Static Stability

Performing a static stability analysis helps determine the dynamic stability of the system without calculating a dynamic system response.

```
stability = staticStability(skyHogg, cruiseState)
```

```
stability=6×8 table
            U           V           W          Alpha         Beta           P           Q
        ________    _________    ________    ________    _________    _________    ________

    FX  "Stable"    ""           ""          ""          ""           ""           ""
    FY  ""          "Neutral"    ""          ""          ""           ""           ""
    FZ  ""          ""           "Stable"    ""          ""           ""           ""
    L   ""          ""           ""          ""          "Neutral"    "Neutral"    ""
    M   "Stable"    ""           ""          "Stable"    ""           ""           "Stable"
    N   ""          ""           ""          ""          "Neutral"    ""           ""
```

With statically stable forces and moments, with perturbations to forward and vertical speeds, and perturbations to angle of attack, the dynamic stability of the system tends towards an oscillating steady-state when perturbing the forward and vertical speeds.

To verify this behavior, use the Control System Toolbox™.

Linearizing the Fixed-Wing Aircraft

To use the tools within the Control System Toolbox™, linearize the aircraft around a state.

This is done by using the linearize method with the same cruise state as before.

```
linSys = linearize(skyHogg, cruiseState)
```

```
linSys =

  A =
                 XN          XD           U           W           Q       Theta
   XN             0           0           1           0           0  -2.586e-07
   XD             0           0           0           1           0      -2.957
   U              0   1.319e-06   -0.001714   -0.000759           0      -0.5617
   W              0           0   -0.002705     -0.3319        2.95  -4.867e-08
   Q              0           0     0.03443       -1.19     -0.4252   8.882e-11
   Theta          0           0           0           0           1           0

  B =
              Delta
   XN             0
   XD             0
   U      -0.0004314
   W        -0.02084
   Q          -2.239
   Theta           0

  C =
             XN     XD      U      W      Q  Theta
   XN         1      0      0      0      0      0
   XD         0      1      0      0      0      0
   U          0      0      1      0      0      0
   W          0      0      0      1      0      0
   Q          0      0      0      0      1      0
   Theta      0      0      0      0      0      1

  D =
          Delta
   XN         0
   XD         0
   U          0
   W          0
   Q          0
   Theta      0

Continuous-time state-space model.
```

Validating Static Stability with Dynamic Response

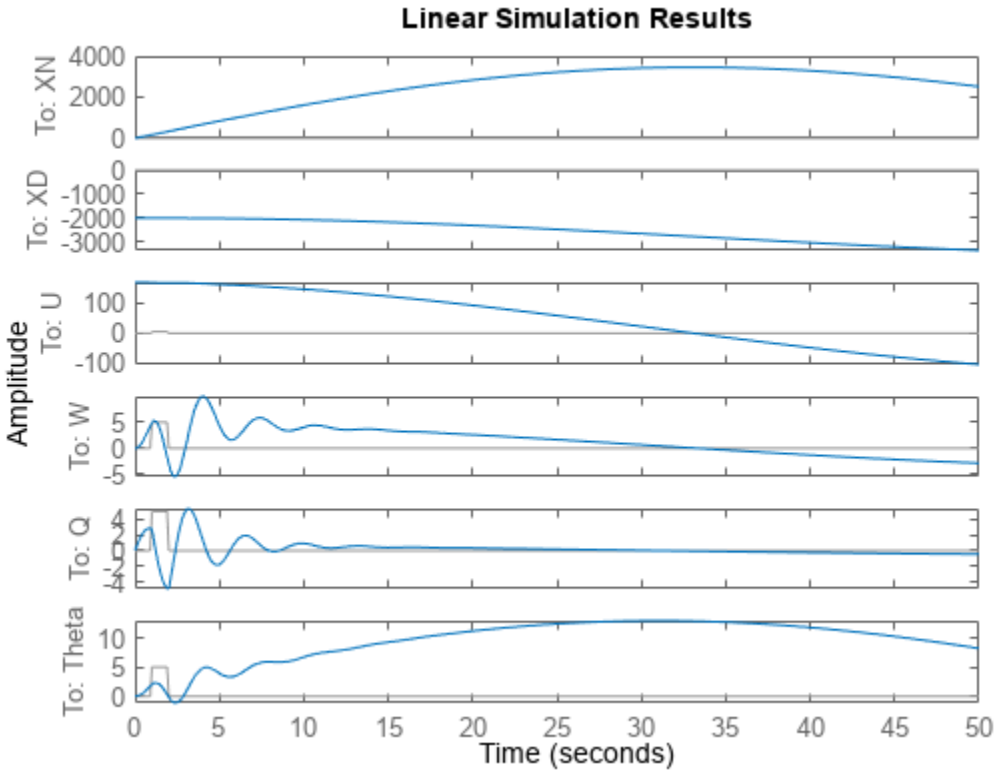
With the linear state-space model constructed, you can plot the dynamic behavior of the system.

To verify the static stability results with the dynamic behavior of the system, plot the states space model against the initial conditions.

To induce a perturbation in the system, a 5 degree step for 1 second is added to the elevator signal.

```
x0 = getState(cruiseState, linSys.OutputName);
t = linspace(0, 50, 500);
```

```
u = zeros(size(t));
u(t > 1 & t < 2) = 5;
lsim(linSys,u,t,x0)
```



As expected from the static stability analysis, the airspeed and pitch-rate is stable when responding to a small perturbation in the elevator.

Isolating the Elevator-Pitch Response

In addition to the static stability verification, isolating the control surfaces to their intended dynamic response can help design controllers specific to the individual surfaces.

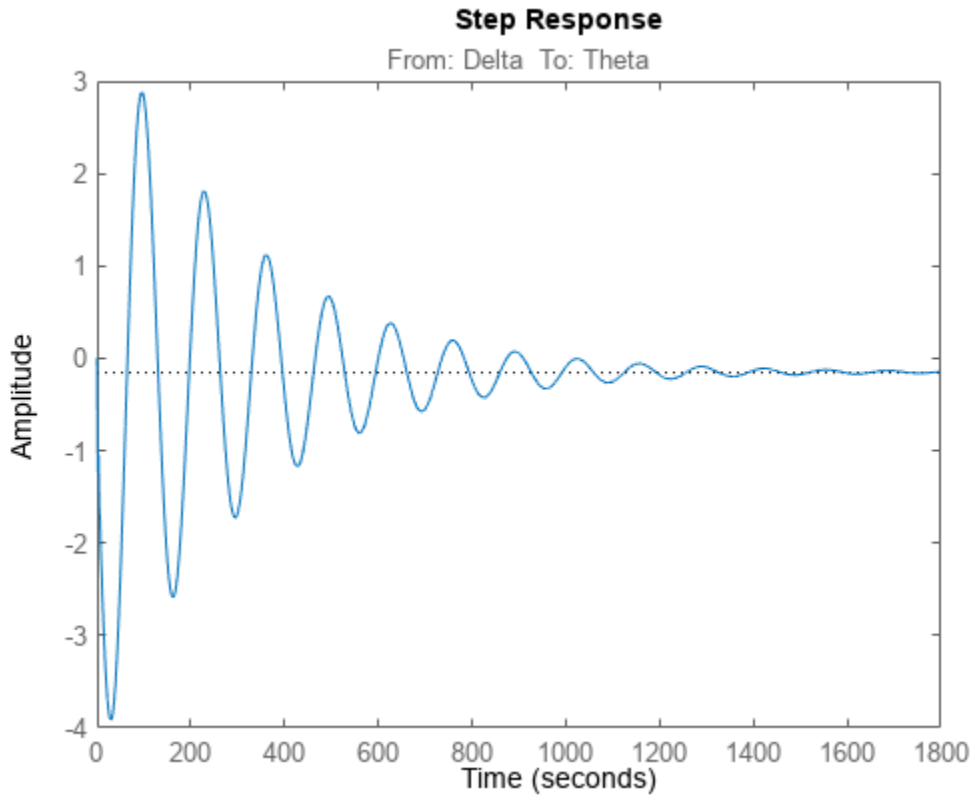
In this case, there is only a single control surface, the elevator.

The elevator controls the pitch response of the aircraft. To show a pitch response, isolate the elevator input to the pitch angle to elevator transfer function.

```
linSysElevatorTF = tf(linSys(6,1))
```

```
linSysElevatorTF =

  From input "Delta" to output "Theta":
           -2.239 s^3 - 0.7221 s^2 - 0.001232 s - 8.935e-09
  ---------------------------------------------------------------------
  s^5 + 0.7587 s^4 + 3.652 s^3 + 0.02567 s^2 + 0.008226 s + 5.711e-08

Continuous-time transfer function.
```

```
step(linSysElevatorTF)
```



As can be seen by the step plot, the pitch response to elevator input has an undesirable oscillatory nature and large steady-state error.

By adding a PID feedback controller to the elevator input, a much more desirable pitch response can be achieved.

```
C = pidtune(linSysElevatorTF, "PID")
```

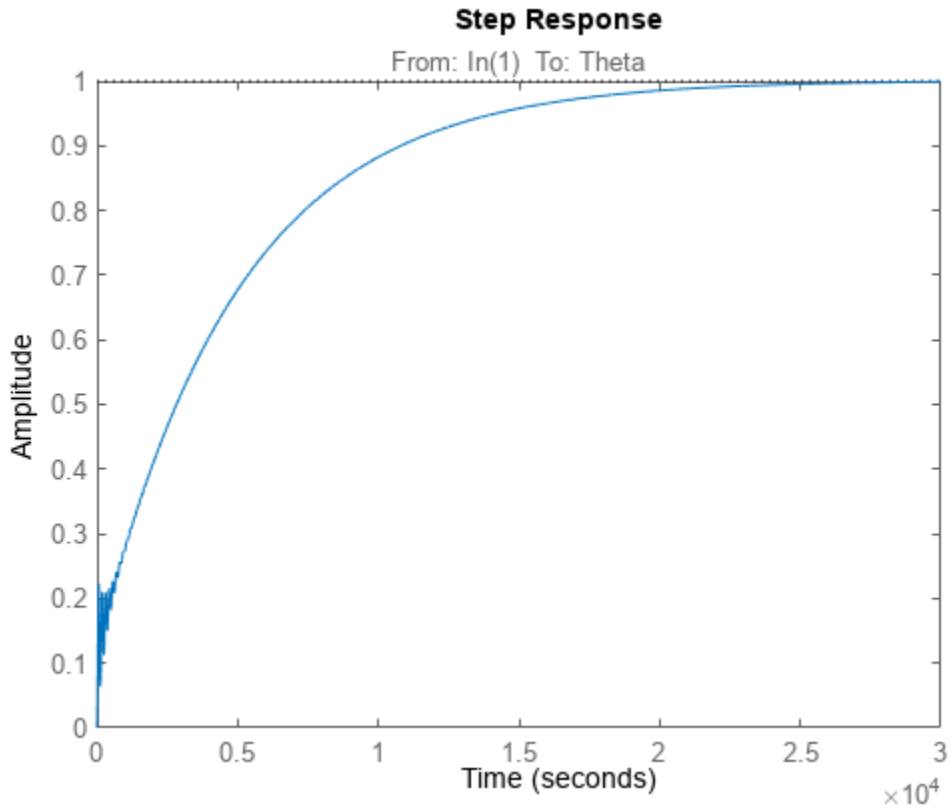
```
C =

       1
  Ki * ---
       s

  with Ki = -0.00153

Continuous-time I-only controller.
```

```
elevatorFeedback = feedback(linSysElevatorTF * C, 1)
```

```
elevatorFeedback =

  From input to output "Theta":
                0.003417 s^3 + 0.001102 s^2 + 1.881e-06 s + 1.364e-11
  ---------------------------------------------------------------------------
  s^6 + 0.7587 s^5 + 3.652 s^4 + 0.02909 s^3 + 0.009328 s^2 + 1.938e-06 s + 1.364e-11
```

```
Continuous-time transfer function.
```

```
step(elevatorFeedback)
```



See Also

Classes
`Aero.FixedWing`

Functions
`datcomimport` | `linearize`

Related Examples

- “Get Started with Fixed-Wing Aircraft” on page 5-170
- “Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121

Customize Fixed-Wing Aircraft with Additional Aircraft States

This example shows how to construct and define a custom state for a fixed-wing aircraft.

This example describes:

- Defining custom states and when they might be used.
- Creating a basic custom state.
- Creating an advanced custom state.
- Using a custom state in the analysis of a fixed-wing aircraft.

What Are Custom States?

By default, the fixed-wing state object has a fixed set of state values. These include angle of attack, airspeed, altitude, and others.

These states are used within the fixed-wing object to dimensionalize non-dimensional coefficients or provide data to lookup table breakpoints.

However, there are cases where this default set of states does not capture all of the desired states of an aircraft. This is when custom states are used.

By defining a custom state, it is possible to create new state values which can be used within any component of a fixed-wing aircraft.

Defining a Custom State

To create custom states with the `Aero.FixedWing.State` class:

1. Define a new class. This class must inherit from the `Aero.FixedWing.State`.
2. Define custom states by adding new dependent properties to the class.
3. Define the get.State method in the custom state class.

Below is a simple example state where the custom state class, `MyState`, is defined with a custom state value, `MyValue`.

The `get` methods of dependent properties can access any other property on the state. In this case, `MyValue` uses ground forward speed, `U`.

```
classdef MyState < Aero.FixedWing.State

    properties (Dependent)
        MyValue
    end

    methods
        function value = get.MyValue(obj)
            value = obj.U * 10;
        end
    end
end
```

A more advanced example of the custom state is the De Havilland Beaver aircraft model [1] which uses a number of custom states to dimensionalize its coefficients. This custom state can be seen as "astDehavillandBeaverState" below.

```
classdef astDehavillandBeaverState < Aero.FixedWing.State

    properties (Dependent)
        Alpha2
        Alpha3
        Beta2
        Beta3

        b2V
        cV
        qcV
        pb2V
        rb2V
        betab2V

        AileronAlpha
        FlapAlpha
        ElevatorBeta2
        RudderAlpha
    end

    methods
        function value = get.Alpha2(obj)
            value = obj.Alpha ^ 2;
        end
        function value = get.Alpha3(obj)
            value = obj.Alpha ^ 3;
        end
        function value = get.Beta2(obj)
            value = obj.Beta ^ 2;
        end
        function value = get.Beta3(obj)
            value = obj.Beta ^ 3;
        end

        function value = get.b2V(obj)
           value = 14.6300 / (2*obj.Airspeed);
        end
        function value = get.cV(obj)
           value = 1.5875 / (obj.Airspeed);
        end

        function value = get.qcV(obj)
           value = obj.Q * obj.cV;
        end
        function value = get.pb2V(obj)
           value = obj.P * obj.b2V;
        end
        function value = get.rb2V(obj)
           value = obj.R * obj.b2V;
        end
        function value = get.betab2V(obj)
           value = obj.Beta * obj.b2V;
        end
```

```
        function value = get.AileronAlpha(obj)
            value = obj.getState("Aileron") * obj.Alpha;
        end
        function value = get.FlapAlpha(obj)
            value = obj.getState("Flap") * obj.Alpha;
        end
        function value = get.ElevatorBeta2(obj)
            value = obj.getState("Elevator") * obj.Beta2;
        end
        function value = get.RudderAlpha(obj)
            value = obj.getState("Rudder") * obj.Alpha;
        end
    end
end
```

This custom state not only directly uses the pre-defined state properties from the fixed-wing state, but also uses the getState method to extract the control surface deflection angles. Any combination of states or methods can be used in the get methods for custom states.

Using a Custom State

With the custom state defined, use the custom state in the analysis methods.

```
[beaver, cruise] = astDehavillandBeaver()
```

```
beaver =
  FixedWing with properties:

        ReferenceArea: 23.2300
        ReferenceSpan: 14.6300
      ReferenceLength: 1.5875
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x3 Aero.FixedWing.Surface]
              Thrusts: [1x1 Aero.FixedWing.Thrust]
          AspectRatio: 9.2138
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"


cruise =
  astDehavillandBeaverState with properties:

                  Alpha2: 0.0170
                  Alpha3: 0.0022
                   Beta2: 0.0036
                   Beta3: 2.1974e-04
                     b2V: 0.1625
                      cV: 0.0353
                     qcV: 0
                    pb2V: 0
                    rb2V: 0
             AileronAlpha: 0.0012
                FlapAlpha: 0
            ElevatorBeta2: -1.5476e-04
```

```
             RudderAlpha: -0.0060
                   Alpha: 0.1303
                    Beta: 0.0603
                AlphaDot: 0
                 BetaDot: 0
                    Mass: 2.2882e+03
                 Inertia: [3x3 table]
          CenterOfGravity: [0 0 0]
       CenterOfPressure: [0 0 0]
            AltitudeMSL: 2202
           GroundHeight: 0
                     XN: 0
                     XE: 0
                     XD: -2202
                      U: 44.5400
                      V: 2.7140
                      W: 5.8360
                    Phi: 0
                  Theta: 0.1309
                    Psi: 0
                      P: 0
                      Q: 0
                      R: 0
                 Weight: 2.2448e+04
           AltitudeAGL: 2202
              Airspeed: 45.0026
           GroundSpeed: 45.0026
            MachNumber: 0.1357
          BodyVelocity: [44.5400 2.7140 5.8360]
        GroundVelocity: [44.5400 2.7140 5.8360]
                    Ur: 44.5400
                    Vr: 2.7140
                    Wr: 5.8360
         FlightPathAngle: 0.1303
             CourseAngle: 0.0609
    InertialToBodyMatrix: [3x3 double]
    BodyToInertialMatrix: [3x3 double]
        BodyToWindMatrix: [3x3 double]
        WindToBodyMatrix: [3x3 double]
   BodyToStabilityMatrix: [3x3 double]
   StabilityToBodyMatrix: [3x3 double]
          DynamicPressure: 998.6513
              Environment: [1x1 Aero.Aircraft.Environment]
            ControlStates: [1x5 Aero.Aircraft.ControlState]
         OutOfRangeAction: "Limit"
         DiagnosticAction: "Warning"
               Properties: [1x1 Aero.Aircraft.Properties]
               UnitSystem: "Metric"
           TemperatureSystem: "Kelvin"
               AngleSystem: "Radians"
```

```
cruise.Alpha2
```

```
ans = 0.0170
```

```
[F, M] = forcesAndMoments(beaver, cruise)
```

```
F = 3×1
10³ ×

    0.4037
   -1.3285
    4.7465


M = 3×1
10³ ×

   -1.5125
    2.3497
    0.3744
```

```
dydt = nonlinearDynamics(beaver, cruise)
```

```
dydt = 12×1

   44.9207
    2.7140
   -0.0276
    0.1764
   -0.5806
    2.0743
   -0.2619
    0.3391
    0.0297
         0
      ⋮
```

```
[stability, derivatives] = staticStability(beaver, cruise)
```

```
stability=6×8 table
             U           V           W          Alpha        Beta         P           Q          ___
          ________    ________    ________    ________    ________    ________    ________    ___

    FX    "Stable"    ""          ""          ""          ""          ""          ""          ""
    FY    ""          "Stable"    ""          ""          ""          ""          ""          ""
    FZ    ""          ""          "Stable"    ""          ""          ""          ""          ""
    L     ""          ""          ""          ""          "Stable"    "Stable"    ""          ""
    M     "Stable"    ""          ""          "Stable"    ""          ""          "Stable"    ""
    N     ""          ""          ""          ""          "Stable"    ""          ""          "St


derivatives=6×8 table
             U          V          W          Alpha        Beta         P          Q          R
          _______    _______    _______    ___________    _______    _______    _______    _____

    FX    -32.531     2.7704     601.22          26968     124.91          0    -552.22
    FY    -33.152    -398.64    -16.894        -558.01     -17973    -467.59          0     1382
    FZ    -410.16    -4.8834    -2867.8    -1.2531e+05    -220.18          0    -2445.2
    L     -37.919    -469.28    -10.703        -254.35     -21158     -27832          0     9350
    M      222.86     74.529     -930.3         -42740     3360.4          0     -20214    -1866
    N      12.771     62.732     1.6733       -0.35231     2828.4    -8744.1     1909.6    -6134
```

States can also be created using the fixedWingStateCustom function.

This function is identical to the fixedWingState function except for the addition of a string input which specified the state object to create.

```
state = fixedWingStateCustom("astDehavillandBeaverState",beaver)
```

```
state =
  astDehavillandBeaverState with properties:

                  Alpha2: 0
                  Alpha3: 0
                   Beta2: 0
                   Beta3: 0
                     b2V: 0.1463
                      cV: 0.0318
                     qcV: 0
                    pb2V: 0
                    rb2V: 0
             AileronAlpha: 0
                FlapAlpha: 0
            ElevatorBeta2: 0
              RudderAlpha: 0
                    Alpha: 0
                     Beta: 0
                 AlphaDot: 0
                  BetaDot: 0
                     Mass: 0
                  Inertia: [3x3 table]
          CenterOfGravity: [0 0 0]
         CenterOfPressure: [0 0 0]
              AltitudeMSL: 0
             GroundHeight: 0
                       XN: 0
                       XE: 0
                       XD: 0
                        U: 50
                        V: 0
                        W: 0
                      Phi: 0
                    Theta: 0
                      Psi: 0
                        P: 0
                        Q: 0
                        R: 0
                   Weight: 0
              AltitudeAGL: 0
                 Airspeed: 50
              GroundSpeed: 50
               MachNumber: 0.1469
             BodyVelocity: [50 0 0]
           GroundVelocity: [50 0 0]
                       Ur: 50
                       Vr: 0
                       Wr: 0
          FlightPathAngle: 0
              CourseAngle: 0
     InertialToBodyMatrix: [3x3 double]
     BodyToInertialMatrix: [3x3 double]
         BodyToWindMatrix: [3x3 double]
```

```
          WindToBodyMatrix: [3x3 double]
     BodyToStabilityMatrix: [3x3 double]
     StabilityToBodyMatrix: [3x3 double]
           DynamicPressure: 1.5312e+03
               Environment: [1x1 Aero.Aircraft.Environment]
            ControlStates: [1x5 Aero.Aircraft.ControlState]
          OutOfRangeAction: "Limit"
          DiagnosticAction: "Warning"
                Properties: [1x1 Aero.Aircraft.Properties]
                UnitSystem: "Metric"
         TemperatureSystem: "Kelvin"
                AngleSystem: "Radians"
```

```
state2 = fixedWingStateCustom("astDehavillandBeaverState",beaver,aircraftEnvironment(beaver,"COES
```

```
state2 =
  astDehavillandBeaverState with properties:

                    Alpha2: 0
                    Alpha3: 0
                     Beta2: 0
                     Beta3: 0
                       b2V: 0.1463
                        cV: 0.0318
                       qcV: 0
                      pb2V: 0
                      rb2V: 0
              AileronAlpha: 0
                 FlapAlpha: 0
             ElevatorBeta2: 0
               RudderAlpha: 0
                     Alpha: 0
                      Beta: 0
                 AlphaDot: 0
                  BetaDot: 0
                     Mass: 0
                  Inertia: [3x3 table]
           CenterOfGravity: [0 0 0]
          CenterOfPressure: [0 0 0]
               AltitudeMSL: 0
              GroundHeight: 0
                        XN: 0
                        XE: 0
                        XD: 0
                         U: 50
                         V: 0
                         W: 0
                       Phi: 0
                     Theta: 0
                       Psi: 0
                         P: 0
                         Q: 0
                         R: 0
                    Weight: 0
               AltitudeAGL: 0
                  Airspeed: 50
               GroundSpeed: 50
```

```
            MachNumber: 0.1486
          BodyVelocity: [50 0 0]
        GroundVelocity: [50 0 0]
                    Ur: 50
                    Vr: 0
                    Wr: 0
       FlightPathAngle: 0
           CourseAngle: 0
  InertialToBodyMatrix: [3x3 double]
  BodyToInertialMatrix: [3x3 double]
      BodyToWindMatrix: [3x3 double]
      WindToBodyMatrix: [3x3 double]
 BodyToStabilityMatrix: [3x3 double]
 StabilityToBodyMatrix: [3x3 double]
       DynamicPressure: 1.3896e+03
           Environment: [1x1 Aero.Aircraft.Environment]
         ControlStates: [1x5 Aero.Aircraft.ControlState]
      OutOfRangeAction: "Limit"
      DiagnosticAction: "Warning"
            Properties: [1x1 Aero.Aircraft.Properties]
            UnitSystem: "Metric"
     TemperatureSystem: "Kelvin"
           AngleSystem: "Radians"
```

References

1 Rauw, M.O.: "A Simulink Environment for Flight Dynamics and Control analysis - Application to the DHC-2 'Beaver' ". Part I: "Implementation of a model library in Simulink". Part II: "Nonlinear analysis of the 'Beaver' autopilot". MSc-thesis, Delft University of Technology, Faculty of Aerospace Engineering. Delft, The Netherlands, 1993.

See Also

Classes

`Aero.FixedWing` | `Aero.FixedWing.State`

Methods

`staticStability`

Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft

This example shows the process of creating and analyzing a fixed-wing aircraft in MATLAB® using Cessna C182 geometry and coefficient data.

The data used to create the aircraft is taken from Airplane Flight Dynamics and Controls by Jan Roskam [1 on page 5-129].

This example describes:

- Setting up fixed-wing aerodynamic and control surfaces by creating and nesting an elevator control surface and then creating the aileron, rudder, wing, and vertical stabilizer.
- Creating the propulsion models on the fixed-wing aircraft models similar to control surfaces.
- Defining the full aircraft.
- Defining the coefficients on the aircraft.
- Preparing the aircraft for numerical analysis.
- Performing numerical analysis.

Setting Up Fixed-Wing Aerodynamic and Control Surfaces

The `Aero.FixedWing.Surface` class can serve as both an aerodynamic and control surface.

This behavior is controlled by the `Controllable` property on the class.

Setting `Controllable` to `on` creates a control surface and a `ControlState` variable.

Nesting a control surface on an aerodynamic surface mimics the actual construction of the aircraft.

Below, the example creates an elevator control surface and nests it on the horizontal stabilizer.

```
elevator = fixedWingSurface("Elevator", "on","Symmetric",[-20,20])
```

```
elevator =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: "Elevator"
          Properties: [1x1 Aero.Aircraft.Properties]
```

```
elevator.Coefficients = fixedWingCoefficient("Elevator")
```

```
elevator =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
```

```
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: "Elevator"
          Properties: [1x1 Aero.Aircraft.Properties]
```

```
horizontalStabilizer = fixedWingSurface("HorizontalStabilizer", "Surfaces", elevator)
```

```
horizontalStabilizer =
  Surface with properties:

            Surfaces: [1x1 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: Inf
        MinimumValue: -Inf
        Controllable: off
            Symmetry: "Symmetric"
    ControlVariables: [0x0 string]
          Properties: [1x1 Aero.Aircraft.Properties]
```

Each property on the fixed-wing objects can also be set through Name,Value arguments on construction. This method of creation will be used in the rest of the example.

Next, construct the ailerons, rudder, wing, and vertical stabilizer.

```
aileron = fixedWingSurface("Aileron", "on", "Asymmetric", [-20,20], ...
    "Coefficients", fixedWingCoefficient("Aileron"));

rudder = fixedWingSurface("Rudder", "on", "Symmetric", [-20,20], ...
    "Coefficients", fixedWingCoefficient("Rudder"));

wing = fixedWingSurface("Wing","Surfaces", aileron);

verticalStabilizer = fixedWingSurface("VerticalStabilizer","Surfaces", rudder);
```

Defining Propulsion

Use the `Aero.FixedWing.Thrust` object to create the propulsion models on the fixed-wing aircraft models similar to control surfaces.

The `Aero.FixedWing.Thrust` object is always controllable. It cannot be nested like aerodynamic and control surfaces.

```
propeller = fixedWingThrust("Propeller","Coefficients", fixedWingCoefficient("Propeller"))
```

```
propeller =
  Thrust with properties:

        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 1
        MinimumValue: 0
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: "Propeller"
          Properties: [1x1 Aero.Aircraft.Properties]
```

Constructing the Aircraft

With the aerodynamic surfaces, control surface, and thrust components defined, define the full aircraft.

First, define a separate `Aero.Aircraft.Properties` class for the aircraft. Use this class to keep track of versions on components and which components a given aircraft is using.

All `Aero.FixedWing` and `Aero.Aircraft` classes contain this property.

```
C182Properties = Aero.Aircraft.Properties(...
    "Name"       , "Cessna C182", ...
    "Type"       , "General Aviation", ...
    "Version"    , "1.0", ...
    "Description", "Cessna 182 Example")
```

```
C182Properties =
  Properties with properties:

           Name: "Cessna C182"
    Description: "Cessna 182 Example"
           Type: "General Aviation"
        Version: "1.0"
```

```
C182 = Aero.FixedWing(...
    "Properties"       , C182Properties, ...
    "UnitSystem"       , "English (ft/s)", ...
    "AngleSystem"      , "Radians", ...
    "TemperatureSystem", "Fahrenheit", ...
    "ReferenceArea"    , 174, ...
    "ReferenceSpan"    , 36, ...
    "ReferenceLength"  , 4.9, ...
    "Surfaces"         , [wing, horizontalStabilizer, verticalStabilizer], ...
    "Thrusts"          , propeller)
```

```
C182 =
  FixedWing with properties:

        ReferenceArea: 174
        ReferenceSpan: 36
      ReferenceLength: 4.9000
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x3 Aero.FixedWing.Surface]
              Thrusts: [1x1 Aero.FixedWing.Thrust]
          AspectRatio: 7.4483
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "English (ft/s)"
    TemperatureSystem: "Fahrenheit"
          AngleSystem: "Radians"
```

Setting the Aircraft Coefficients

Next, define the coefficients on the aircraft.

These coefficients describe the dynamic behavior of the aircraft. This example defines scalar constant coefficients, which define the linear behavior of the aircraft.

To define non-linear dynamic behavior of a fixed-wing aircraft, define `Simulink.LookupTable` coefficients. Simulink.LookupTables are not used in this example. To see an example using Simulink.LookupTables, open the “Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106 example.

By default, all coefficients are 0.

```
BodyCoefficients = {
    'CD', 'Zero', 0.027;
    'CL', 'Zero', 0.307;
    'Cm', 'Zero', 0.04;
    'CD', 'Alpha', 0.121;
    'CL', 'Alpha', 4.41;
    'Cm', 'Alpha', -0.613;
    'CD', 'AlphaDot', 0
    'CL', 'AlphaDot',  1.7;
    'Cm', 'AlphaDot', -7.27;
    'CD', 'Q', 0;
    'CL', 'Q', 3.9;
    'Cm', 'Q', -12.4;
    'CY', 'Beta', -0.393;
    'Cl', 'Beta', -0.0923;
    'Cn', 'Beta', 0.0587;
    'CY', 'P', -0.075;
    'Cl', 'P', -0.484;
    'Cn', 'P', -0.0278;
    'CY', 'R', 0.214;
    'Cl', 'R', 0.0798;
    'Cn', 'R', -0.0937;
    };

C182 = setCoefficient(C182, BodyCoefficients(:, 1), BodyCoefficients(:, 2), BodyCoefficients(:, 3
```

Coefficients can be defined on any component on the aircraft. These components can include any `Aero.FixedWing.Surface` or `Aero.FixedWing.Thrust`.

The `setCoefficient` method provides a `Component` `Name,Value` argument, which takes the coefficients on the desired component name, obviating the need to know exactly where the component is on the aircraft.

Valid component names depend on the `Name` property on the component.

```
AileronCoefficients = {
    'CY', 'Aileron', 0;
    'Cl', 'Aileron', 0.229;
    'Cn', 'Aileron', -0.0216;
    };
RudderCoefficients = {
    'CY', 'Rudder', 0.187;
    'Cl', 'Rudder', 0.0147;
    'Cn', 'Rudder', -0.0645;
    };
ElevatorCoefficients = {
    'CD', 'Elevator', 0;
    'CL', 'Elevator', 0.43;
    'Cm', 'Elevator', -1.122;
    };
```

```
PropellerCoefficients = {
    'CD', 'Propeller', -21.1200;
    };

C182 = setCoefficient(C182, AileronCoefficients(:, 1), AileronCoefficients(:, 2), AileronCoeffici
C182 = setCoefficient(C182, ElevatorCoefficients(:, 1), ElevatorCoefficients(:, 2), ElevatorCoeff
C182 = setCoefficient(C182, RudderCoefficients(:, 1), RudderCoefficients(:, 2), RudderCoefficient
C182 = setCoefficient(C182, PropellerCoefficients(:, 1), PropellerCoefficients(:, 2), PropellerCo
```

Defining the Current State

The fixed-wing aircraft is fully constructed and ready for numerical analysis.

To perform numerical analysis on a fixed-wing aircraft, define an `Aero.FixedWing.State` object.

The `Aero.FixedWing.State` object defines the current state of the `Aero.FixedWing` object at an instance in time. The `Aero.FixedWing.State` is also where dynamic physical properties of the aircraft, including the mass and inertia, are defined.

In this example, we analyze the cruise state.

```
CruiseState = Aero.FixedWing.State(...
    "UnitSystem",C182.UnitSystem,...
    "AngleSystem",C182.AngleSystem, ...
    "TemperatureSystem",C182.TemperatureSystem, ...
    "Mass",82.2981, ...
    "U", 220.1, ...
    "AltitudeMSL",5000);

CruiseState.Inertia.Variables = [
    948, 0,    0   ;
    0  , 1346, 0   ;
    0  , 0   , 1967;
    ];

CruiseState.CenterOfGravity = [0.264, 0 , 0] .* C182.ReferenceLength;
CruiseState.CenterOfPressure = [0.25, 0, 0] .* C182.ReferenceLength;
CruiseState.Environment = aircraftEnvironment(C182,"ISA",CruiseState.AltitudeMSL);
```

Setting Up the Control States

In addition to the environment and dynamic physical properties, the `Aero.FixedWing.State` class also holds the current control surface deflections and thrust positions. These positions are held in the `ControlStates` property. Use this class to set up the control states.

By default, this property is empty. Initialize the property from the control surface and thrust information on the aircraft.

To set up these control states, use the `setupControlStates` method below.

```
CruiseState = setupControlStates(CruiseState, C182)
```

```
CruiseState =
  State with properties:

                    Alpha: 0
                     Beta: 0
                 AlphaDot: 0
```

```
               BetaDot: 0
                  Mass: 82.2981
               Inertia: [3x3 table]
        CenterOfGravity: [1.2936 0 0]
      CenterOfPressure: [1.2250 0 0]
           AltitudeMSL: 5000
          GroundHeight: 0
                    XN: 0
                    XE: 0
                    XD: -5000
                     U: 220.1000
                     V: 0
                     W: 0
                   Phi: 0
                 Theta: 0
                   Psi: 0
                     P: 0
                     Q: 0
                     R: 0
                Weight: 2.6488e+03
            AltitudeAGL: 5000
               Airspeed: 220.1000
            GroundSpeed: 220.1000
             MachNumber: 0.2006
           BodyVelocity: [220.1000 0 0]
         GroundVelocity: [220.1000 0 0]
                    Ur: 220.1000
                    Vr: 0
                    Wr: 0
        FlightPathAngle: 0
            CourseAngle: 0
   InertialToBodyMatrix: [3x3 double]
   BodyToInertialMatrix: [3x3 double]
       BodyToWindMatrix: [3x3 double]
       WindToBodyMatrix: [3x3 double]
  BodyToStabilityMatrix: [3x3 double]
  StabilityToBodyMatrix: [3x3 double]
        DynamicPressure: 49.6090
            Environment: [1x1 Aero.Aircraft.Environment]
          ControlStates: [1x6 Aero.Aircraft.ControlState]
       OutOfRangeAction: "Limit"
       DiagnosticAction: "Warning"
             Properties: [1x1 Aero.Aircraft.Properties]
             UnitSystem: "English (ft/s)"
      TemperatureSystem: "Fahrenheit"
             AngleSystem: "Radians"
```

Perform this only once per aircraft configuration. If no control surfaces or thrusts have been added or removed to the aircraft, skip this step.

Performing Numerical Analysis

At this point, the aircraft and state are now fully constructed and defined.

A number of numerical analysis methods come with the fixed-wing aircraft, including forces and moments, non-linear dynamics, and static stability.

Forces and Moments

To calculate the forces and moments on the aircraft at an instance in time, use the `forcesAndMoments` method.

These forces and moments are in the aircraft body frame. Coefficients defined in a different frame have the appropriate transformation matrices applied to translate them to the body frame.

```
[F, M] = forcesAndMoments(C182, CruiseState)
```

```
F = 3×1

 -233.0633
         0
   -1.2484


M = 3×1
10^3 ×

         0
    1.5101
         0
```

Nonlinear Dynamics

To calculate the aircraft dynamic behavior, use the `nonlinearDynamics` method.

The `nonlinearDynamics` method returns a vector of the rates of change of the selected degrees of freedom on the aircraft. The size of the vector depends on the degrees of freedom. To calculate the aircraft dynamic behavior over time, use the vector in conjunction with an ode solver, such as ode45.

To quickly iterate between the fidelities of different aircraft designs, or trim unnecessary states from the output vector, change the selected degrees of freedom. These rates of change are defined below:

```
load("astFixedWingDOFtable.mat").DOFtable
```

```
ans=12×4 table
                  PM4    PM6    3DOF    6DOF
                  ___    ___    ____    ____

    dXN /dt       "X"    "X"    "X"     "X"
    dXE /dt       ""     "X"    ""      "X"
    dXD /dt       "X"    "X"    "X"     "X"
    dU /dt        "X"    "X"    "X"     "X"
    dV /dt        ""     "X"    ""      "X"
    dW /dt        "X"    "X"    "X"     "X"
    dP /dt        ""     ""     ""      "X"
    dQ /dt        ""     ""     "X"     "X"
    dR /dt        ""     ""     ""      "X"
    dPhi /dt      ""     ""     ""      "X"
    dTheta /dt    ""     ""     "X"     "X"
    dPsi /dt      ""     ""     ""      "X"
```

```
dydt = nonlinearDynamics(C182, CruiseState)
```

```
dydt = 12×1

  220.1000
         0
         0
   -2.8319
         0
   -0.0152
         0
    1.1219
         0
         0
      ⋮
```

Static Stability

Static stability is the tendency of an aircraft to return to its original state after a small perturbation from an initial state. It is an important feature of aircraft for civilian use and reduces the need for complex controllers to maintain dynamic stability. Under some conditions, aircraft that require advanced maneuverability might opt for static instability.

The `Aero.FixedWing` object static stability method calculates from changes in forces and moments due to perturbations at the current state of an aircraft.

The method compares the perturbations against a predefined set of criteria as less-than, greater-than, or equal-to zero. You can also specify custom criteria. The method then evaluates the static stability as:

- If the criteria is satisfied, then the perturbation is statically stable.
- If the criteria is not satisfied, then the perturbation is statically unstable.
- If the perturbation is 0, the perturbation is statically neutral.

The `staticStability` method does not perform a requirements-based analysis. Only use this method in the preliminary design phase.

```
[stability, derivatives] = staticStability(C182, CruiseState)
```

```
stability=6×8 table
             U            V            W          Alpha        Beta         P           Q
          ________    __________    ________    ________    ________    ________    ________    _

    FX    "Stable"    ""            ""          ""          ""          ""          ""          
    FY    ""          "Unstable"    ""          ""          ""          ""          ""          
    FZ    ""          ""            "Stable"    ""          ""          ""          ""          
    L     ""          ""            ""          ""          "Stable"    "Stable"    ""          
    M     "Stable"    ""            ""          "Stable"    ""          ""          "Stable"    
    N     ""          ""            ""          ""          "Stable"    ""          ""          

derivatives=6×8 table
             U            V            W        Alpha       Beta          P           Q          R
          _______    ___________    _______    ______    _________    _______    _______    _____

    FX    -2.1178    -7.2475e-07     7.2946    1605.9    -0.035089          0          0
    FY          0         14.354          0         0       3159.3     52.945          0    -151
    FZ     -24.08     -5.457e-07    -174.01    -38300      -0.0265          0    -374.73
```

```
L           0           -138          0          0       -30374     -12300          0       20
M      13.722    -5.7526e-06    -129.74     -28555     -0.28018          0    -5863.8
N           0         81.892          0          0        18024    -710.13          0    -2370
```

References

1 Roskam, J., "Airplane Flight Dynamics and Automatic Flight Controls (Part 1)", DAR Corporation, 2003.

See Also

`fixedWingAircraft` | `fixedWingCoefficient` | `fixedWingState` | `fixedWingSurface` | `aircraftEnvironment` | `fixedWingThrust`

Related Examples

- “Get Started with Fixed-Wing Aircraft” on page 5-170
- “Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106

Modeling Satellite Constellations Using Ephemeris Data

This example demonstrates how to add time-stamped ephemeris data for a constellation of 24 satellites (similar to ESA Galileo GNSS constellation) to a satellite scenario for access analysis. The example uses data generated by the Aerospace Blockset **`Orbit Propagator`** block. For more information, see the Aerospace Blockset example *Constellation Modeling with the Orbit Propagator Block*.

The **`satelliteScenario`** object supports loading previously generated, time-stamped satellite ephemeris data into a scenario from a **`timeseries`** or **`timetable`** object. An ephemeris is a table containing position (and optionally velocity) state information of a satellite during a given period of time. Ephemeris data used to add satellites to the scenario object is interpolated via the `makima` interpolation method to align with the scenario time steps. This allows you to incorporate data generated by a Simulink model into either a new or existing satelliteScenario.

Define Mission Parameters and Constellation Initial Conditions

Specify a start date and duration for the mission. This example uses MATLAB structures to organize mission data. These structures make accessing data later in the example more intuitive. They also help declutter the global base workspace.

```
mission.StartDate = datetime(2020, 11, 30, 22, 23, 24);
mission.Duration  = hours(24);
```

The constellation in this example is a Walker-Delta constellation modeled similar to Galileo, the European GNSS (global navigation satellite system) constellation. The constellation consists of 24 satellites in medium Earth orbit (MEO). The satellites' Keplerian orbital elements at the mission start date epoch are:

```
mission.ConstellationDefinition = table( ...
    29599.8e3 * ones(24,1), ...                % Semi-major axis (m)
    0.0005    * ones(24,1), ...                % Eccentricity
    56        * ones(24,1), ...                % Inclination (deg)
    350       * ones(24,1), ...                % Right ascension of the ascending node (deg)
    sort(repmat([0 120 240], 1,8))', ...       % Argument of periapsis (deg)
    [0:45:315, 15:45:330, 30:45:345]', ...     % True anomaly (deg)
    'VariableNames', ["a (m)", "e", "i (deg)", ...
    "Ω (deg)", "ω (deg)", "ν (deg)"]);
mission.ConstellationDefinition
```

```
ans=24×6 table
     a (m)        e       i (deg)    Ω (deg)    ω (deg)    ν (deg)
    ________    ______    _______    _______    _______    _______

    2.96e+07    0.0005      56         350          0          0
    2.96e+07    0.0005      56         350          0         45
    2.96e+07    0.0005      56         350          0         90
    2.96e+07    0.0005      56         350          0        135
    2.96e+07    0.0005      56         350          0        180
    2.96e+07    0.0005      56         350          0        225
    2.96e+07    0.0005      56         350          0        270
    2.96e+07    0.0005      56         350          0        315
    2.96e+07    0.0005      56         350        120         15
    2.96e+07    0.0005      56         350        120         60
    2.96e+07    0.0005      56         350        120        105
```

```
    2.96e+07    0.0005      56         350        120        150
    2.96e+07    0.0005      56         350        120        195
    2.96e+07    0.0005      56         350        120        240
    2.96e+07    0.0005      56         350        120        285
    2.96e+07    0.0005      56         350        120        330
       ⋮
```

Load Ephemeris Timeseries Data

The timeseries objects contain position and velocity data for all 24 satellites in the constellation. The data is referenced in the International Terrestrial Reference frame (ITRF), which is an Earth-centered Earth-fixed (ECEF) coordinate system. The data was generated using the Aerospace Blockset **`Orbit Propagator`** block. For more information, see the Aerospace Blockset example *Constellation Modeling with the Orbit Propagator Block*.

```
mission.Ephemeris = load("SatelliteScenarioEphemerisData.mat", "TimeseriesPosITRF", "TimeseriesVe
mission.Ephemeris.TimeseriesPosITRF
```

```
  timeseries

  Common Properties:
            Name: ''
            Time: [57x1 double]
        TimeInfo: [1x1 tsdata.timemetadata]
            Data: [24x3x57 double]
        DataInfo: [1x1 tsdata.datametadata]

  More properties, Methods
```

```
mission.Ephemeris.TimeseriesVelITRF
```

```
  timeseries

  Common Properties:
            Name: ''
            Time: [57x1 double]
        TimeInfo: [1x1 tsdata.timemetadata]
            Data: [24x3x57 double]
        DataInfo: [1x1 tsdata.datametadata]

  More properties, Methods
```

Load the Satellite Ephemerides into a `satelliteScenario` Object

Create a satellite scenario object for the analysis.

```
scenario = satelliteScenario(mission.StartDate, mission.StartDate + hours(24), 60);
```

Use the **`satellite`** method to add all 24 satellites to the satellite scenario from the ECEF position and velocity timeseries objects. This example uses position and velocity information; however satellites can also be added from position data only and velocity states are then estimated. Available coordinate frames for Name-Value pair `CoordinateFrame` are `"ECEF"`, `"Inertial"`, and `"Geographic"`. If the timeseries object contains a value for ts`.TimeInfo.StartDate`, the method uses that value as the epoch for the timeseries object. If no `StartDate` is defined, the method uses the scenario start date by default.

```
sat = satellite(scenario, mission.Ephemeris.TimeseriesPosITRF, mission.Ephemeris.TimeseriesVelITR
    CoordinateFrame="ecef", Name="GALILEO " + (1:24))
```

```
sat =
  1x24 Satellite array with properties:

    Name
    ID
    ConicalSensors
    Gimbals
    Transmitters
    Receivers
    Accesses
    GroundTrack
    Orbit
    OrbitPropagator
    MarkerColor
    MarkerSize
    ShowLabel
    LabelFontColor
    LabelFontSize
```

```
disp(scenario)
```

```
  satelliteScenario with properties:

         StartTime: 30-Nov-2020 22:23:24
          StopTime: 01-Dec-2020 22:23:24
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×24 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Alternatively, satellites can also be added as ephemerides to the satellite scenario as a MATLAB **timetable**, **table**, or **tscollection**. For example, a **timetable** containing the first 3 satellites of the position **timeseries** object in the previous section, formatted for use with **satelliteScenario** objects is shown below.

- Satellites are represented by variables (column headers).
- Each row contains a position vector associated with the row's `Time` property.

```
timetable(...
datetime(getabstime(mission.Ephemeris.TimeseriesPosITRF), Locale="en_US"), ...
squeeze(mission.Ephemeris.TimeseriesPosITRF.Data(1,:,:))', ...
squeeze(mission.Ephemeris.TimeseriesPosITRF.Data(2,:,:))', ...
squeeze(mission.Ephemeris.TimeseriesPosITRF.Data(3,:,:))',...
VariableNames=["Satellite_1", "Satellite_2", "Satellite_3"])
```

```
ans=57×3 timetable
            Time                          Satellite_1                                  Satellite_2
    ____________________    ____________________________________    ____________________________

    30-Nov-2020 22:23:24    1.8249e+07    -2.2904e+07    -4.2009e+06    2.3678e+07     -1.075e+07
    30-Nov-2020 22:23:38    1.8252e+07    -2.2909e+07    -4.1563e+06    2.3662e+07    -1.0735e+07
    30-Nov-2020 22:24:53    1.8268e+07    -2.2937e+07     -3.933e+06    2.3584e+07    -1.0663e+07
```

```
    30-Nov-2020 22:31:05    1.8326e+07    -2.3055e+07    -2.8121e+06    2.3185e+07     -1.028e+0
    30-Nov-2020 22:48:39    1.8326e+07    -2.3223e+07     3.9182e+05    2.2005e+07    -8.9966e+0
    30-Nov-2020 23:08:30    1.8076e+07    -2.3078e+07     3.9992e+06    2.0643e+07    -7.2057e+0
    30-Nov-2020 23:28:27    1.7624e+07    -2.2538e+07     7.5358e+06    1.9321e+07    -5.0678e+0
    30-Nov-2020 23:50:59    1.6968e+07    -2.1428e+07     1.1328e+07    1.7977e+07    -2.3021e+0
    01-Dec-2020 00:14:27    1.6244e+07    -1.9712e+07     1.4937e+07    1.6838e+07     8.7771e+0
    01-Dec-2020 00:38:42    1.5585e+07    -1.7375e+07     1.8189e+07    1.6017e+07      4.355e+0
    01-Dec-2020 01:04:35    1.5124e+07    -1.4345e+07     2.1006e+07    1.5585e+07     8.1065e+0
    01-Dec-2020 01:31:17    1.5035e+07     -1.079e+07     2.3096e+07     1.562e+07     1.1816e+0
    01-Dec-2020 01:58:58    1.5443e+07    -6.8501e+06     2.4303e+07    1.6102e+07     1.5274e+0
    01-Dec-2020 02:27:08    1.6406e+07    -2.8152e+06     2.4478e+07    1.6925e+07     1.8197e+0
    01-Dec-2020 02:55:18    1.7869e+07      1.001e+06     2.3582e+07    1.7894e+07     2.0376e+0
    01-Dec-2020 03:23:29    1.9711e+07      4.381e+06     2.1653e+07    1.8787e+07     2.1739e+0
      ⋮
```

Set Graphical Properties on the Satellites

Set satellite in the same orbital plane to have the same orbit color.

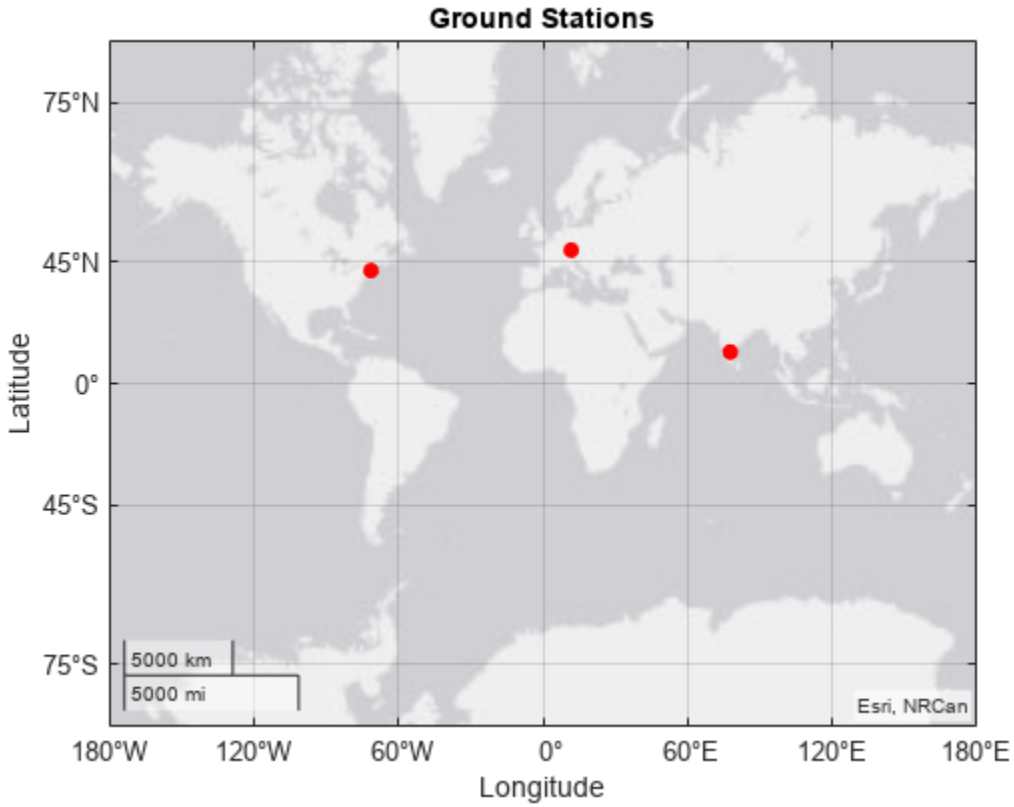
```
set(sat(1:8), MarkerColor="#FF6929");
set(sat(9:16), MarkerColor="#139FFF");
set(sat(17:24), MarkerColor="#64D413");
orbit = [sat(:).Orbit];
set(orbit(1:8), LineColor="#FF6929");
set(orbit(9:16), LineColor="#139FFF");
set(orbit(17:24), LineColor="#64D413");
```

Add Ground Stations to Scenario

To provide accurate positioning data, a location on Earth must have access to at least 4 satellites in the constellation at any given time. In this example, use three locations to compare total constellation access over the 1 day analysis window to different regions of Earth:

- Natick, Massachusetts, USA (42.30048°, -71.34908°)
- München, Germany (48.23206°, 11.68445°)
- Bangalore, India (12.94448°, 77.69256°)

```
gsUS = groundStation(scenario, 42.30048, -71.34908, ...
    MinElevationAngle=10, Name="Natick");
gsUS.MarkerColor = "red";
gsDE = groundStation(scenario, 48.23206, 11.68445, ...
    MinElevationAngle=10, Name="Munchen");
gsDE.MarkerColor = "red";
gsIN = groundStation(scenario, 12.94448, 77.69256, ...
    MinElevationAngle=10, Name="Bangalore");
gsIN.MarkerColor = "red";

figure
geoscatter([gsUS.Latitude gsDE.Latitude gsIN.Latitude], ...
    [gsUS.Longitude gsDE.Longitude gsIN.Longitude], "red", "filled")
geolimits([-75 75], [-180 180])
title("Ground Stations")
```



Compute Ground Station to Satellite Access (Line-of-Sight Visibility)

Calculate line-of-sight access between the ground stations and each individual satellite using the `access` method.

```
accessUS = access(gsUS, sat);
accessDE = access(gsDE, sat);
accessIN = access(gsIN, sat);
```

Set access colors to match orbital plane colors assigned earlier in the example.

```
set(accessUS, LineWidth="1");
set(accessUS(1:8), LineColor="#FF6929");
set(accessUS(9:16), LineColor="#139FFF");
set(accessUS(17:24), LineColor="#64D413");

set(accessDE, LineWidth="1");
set(accessDE(1:8), LineColor="#FF6929");
set(accessDE(9:16), LineColor="#139FFF");
set(accessDE(17:24), LineColor="#64D413");

set(accessIN, LineWidth="1");
set(accessIN(1:8), LineColor="#FF6929");
set(accessIN(9:16), LineColor="#139FFF");
set(accessIN(17:24), LineColor="#64D413");
```

View the full access table between each ground station and all satellites in the constellation as tables. Sort the access intervals by interval start time. Satellites added from ephemeris data do not display values for `StartOrbit` and `EndOrbit`.

```
intervalsUS = accessIntervals(accessUS);
intervalsUS = sortrows(intervalsUS, "StartTime", "ascend")
```

```
intervalsUS=40×8 table
     Source        Target       IntervalNumber         StartTime              EndTime
    ________    ____________    ______________    ____________________    ____________________

    "Natick"    "GALILEO 1"           1           30-Nov-2020 22:23:24    01-Dec-2020 04:04:24
    "Natick"    "GALILEO 2"           1           30-Nov-2020 22:23:24    01-Dec-2020 01:24:24
    "Natick"    "GALILEO 3"           1           30-Nov-2020 22:23:24    30-Nov-2020 22:57:24
    "Natick"    "GALILEO 12"          1           30-Nov-2020 22:23:24    01-Dec-2020 00:00:24
    "Natick"    "GALILEO 13"          1           30-Nov-2020 22:23:24    30-Nov-2020 23:05:24
    "Natick"    "GALILEO 18"          1           30-Nov-2020 22:23:24    01-Dec-2020 04:00:24
    "Natick"    "GALILEO 19"          1           30-Nov-2020 22:23:24    01-Dec-2020 01:42:24
    "Natick"    "GALILEO 20"          1           30-Nov-2020 22:23:24    30-Nov-2020 22:46:24
    "Natick"    "GALILEO 11"          1           30-Nov-2020 22:25:24    01-Dec-2020 00:18:24
    "Natick"    "GALILEO 17"          1           30-Nov-2020 22:50:24    01-Dec-2020 05:50:24
    "Natick"    "GALILEO 8"           1           30-Nov-2020 23:20:24    01-Dec-2020 07:09:24
    "Natick"    "GALILEO 7"           1           01-Dec-2020 01:26:24    01-Dec-2020 10:00:24
    "Natick"    "GALILEO 24"          1           01-Dec-2020 01:40:24    01-Dec-2020 07:12:24
    "Natick"    "GALILEO 14"          1           01-Dec-2020 03:56:24    01-Dec-2020 07:15:24
    "Natick"    "GALILEO 6"           1           01-Dec-2020 04:05:24    01-Dec-2020 12:14:24
    "Natick"    "GALILEO 23"          1           01-Dec-2020 04:10:24    01-Dec-2020 08:03:24
      ⋮
```

```
intervalsDE = accessIntervals(accessDE);
intervalsDE = sortrows(intervalsDE, "StartTime", "ascend")
```

```
intervalsDE=40×8 table
     Source         Target       IntervalNumber         StartTime              EndTime
    ________    ____________    ______________    ____________________    ____________________

    "Munchen"    "GALILEO 2"           1           30-Nov-2020 22:23:24    01-Dec-2020 04:34:24
    "Munchen"    "GALILEO 3"           1           30-Nov-2020 22:23:24    01-Dec-2020 01:58:24
    "Munchen"    "GALILEO 4"           1           30-Nov-2020 22:23:24    30-Nov-2020 23:05:24
    "Munchen"    "GALILEO 10"          1           30-Nov-2020 22:23:24    30-Nov-2020 23:58:24
    "Munchen"    "GALILEO 19"          1           30-Nov-2020 22:23:24    01-Dec-2020 01:36:24
    "Munchen"    "GALILEO 20"          1           30-Nov-2020 22:23:24    01-Dec-2020 00:15:24
    "Munchen"    "GALILEO 21"          1           30-Nov-2020 22:23:24    30-Nov-2020 22:28:24
    "Munchen"    "GALILEO 9"           1           30-Nov-2020 22:34:24    01-Dec-2020 02:22:24
    "Munchen"    "GALILEO 18"          1           30-Nov-2020 22:41:24    01-Dec-2020 02:31:24
    "Munchen"    "GALILEO 1"           1           30-Nov-2020 23:05:24    01-Dec-2020 06:42:24
    "Munchen"    "GALILEO 16"          1           30-Nov-2020 23:29:24    01-Dec-2020 04:47:24
    "Munchen"    "GALILEO 15"          1           01-Dec-2020 00:50:24    01-Dec-2020 07:27:24
    "Munchen"    "GALILEO 17"          1           01-Dec-2020 01:05:24    01-Dec-2020 03:00:24
    "Munchen"    "GALILEO 8"           1           01-Dec-2020 01:57:24    01-Dec-2020 08:25:24
    "Munchen"    "GALILEO 14"          1           01-Dec-2020 02:36:24    01-Dec-2020 10:19:24
    "Munchen"    "GALILEO 7"           1           01-Dec-2020 04:35:24    01-Dec-2020 09:43:24
      ⋮
```

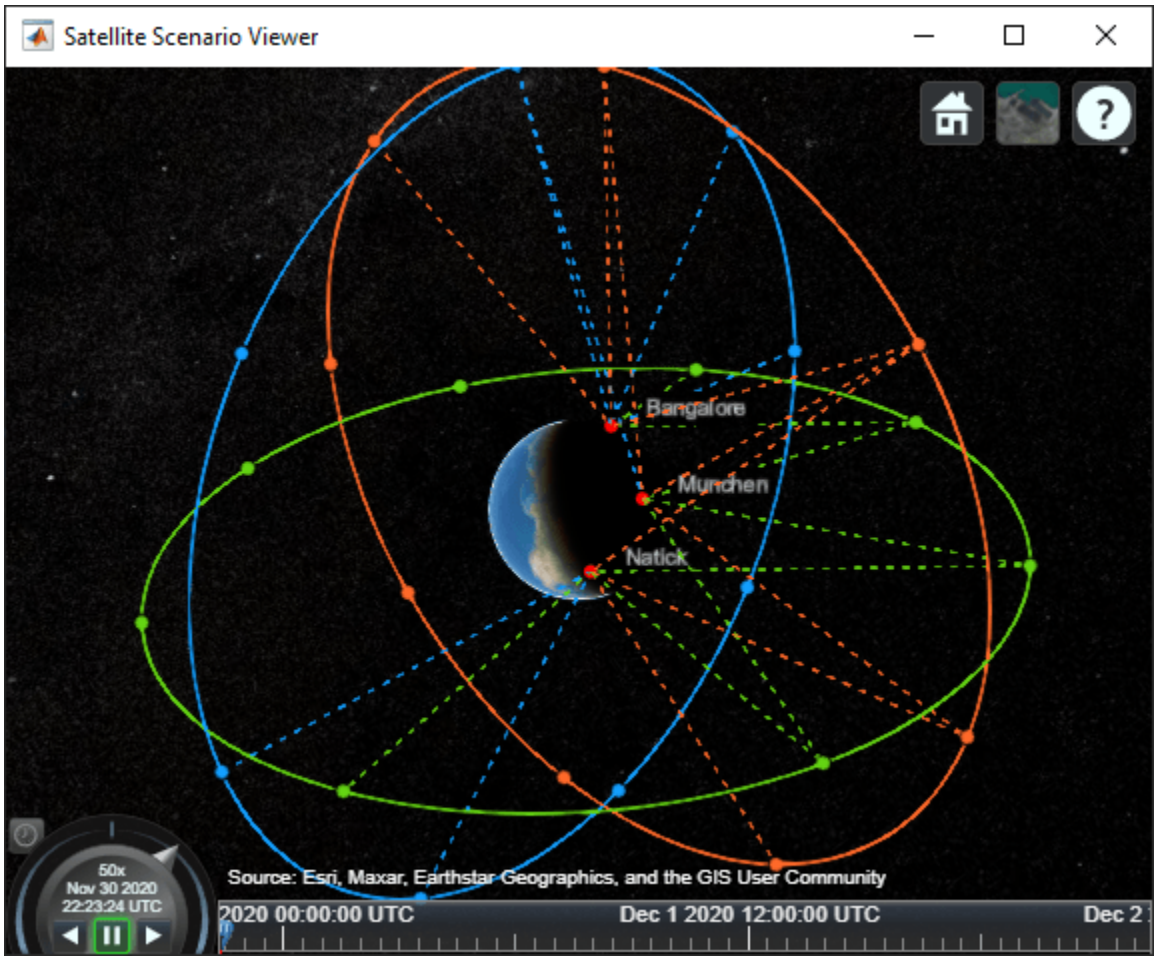
```
intervalsIN = accessIntervals(accessIN);
intervalsIN = sortrows(intervalsIN, "StartTime", "ascend")
```

```
intervalsIN=31×8 table
      Source          Target       IntervalNumber         StartTime              EndTime
    ___________    _____________    ______________    ____________________    ____________________

    "Bangalore"    "GALILEO 3"           1          30-Nov-2020 22:23:24    01-Dec-2020 05:12:24
    "Bangalore"    "GALILEO 4"           1          30-Nov-2020 22:23:24    01-Dec-2020 02:59:24
    "Bangalore"    "GALILEO 5"           1          30-Nov-2020 22:23:24    01-Dec-2020 00:22:24
    "Bangalore"    "GALILEO 9"           1          30-Nov-2020 22:23:24    01-Dec-2020 03:37:24
    "Bangalore"    "GALILEO 10"          1          30-Nov-2020 22:23:24    01-Dec-2020 00:09:24
    "Bangalore"    "GALILEO 16"          1          30-Nov-2020 22:23:24    01-Dec-2020 08:44:24
    "Bangalore"    "GALILEO 21"          1          30-Nov-2020 22:23:24    30-Nov-2020 23:25:24
    "Bangalore"    "GALILEO 22"          1          30-Nov-2020 22:23:24    30-Nov-2020 22:58:24
    "Bangalore"    "GALILEO 15"          1          01-Dec-2020 00:17:24    01-Dec-2020 11:16:24
    "Bangalore"    "GALILEO 2"           1          01-Dec-2020 00:25:24    01-Dec-2020 07:10:24
    "Bangalore"    "GALILEO 22"          2          01-Dec-2020 00:48:24    01-Dec-2020 05:50:24
    "Bangalore"    "GALILEO 21"          2          01-Dec-2020 01:32:24    01-Dec-2020 08:29:24
    "Bangalore"    "GALILEO 1"           1          01-Dec-2020 03:06:24    01-Dec-2020 07:17:24
    "Bangalore"    "GALILEO 20"          1          01-Dec-2020 03:36:24    01-Dec-2020 12:38:24
    "Bangalore"    "GALILEO 14"          1          01-Dec-2020 05:48:24    01-Dec-2020 13:29:24
    "Bangalore"    "GALILEO 19"          1          01-Dec-2020 05:53:24    01-Dec-2020 17:06:24
      ⋮
```

View the Satellite Scenario

Open a 3-D viewer window of the scenario. The viewer window contains all 24 satellites and the three ground stations defined earlier in this example. A line is drawn between each ground station and satellite during their corresponding access intervals. Hide the details of the satellites and ground stations by setting the `ShowDetails` name-value pair to `false`. Show satellite orbits and labels for the ground station locations.

```
viewer3D = satelliteScenarioViewer(scenario, ShowDetails=false);
show(sat.Orbit);
gsUS.ShowLabel = true;
gsUS.LabelFontSize = 11;
gsDE.ShowLabel = true;
gsDE.LabelFontSize = 11;
gsIN.ShowLabel = true;
gsIN.LabelFontSize = 11;
```

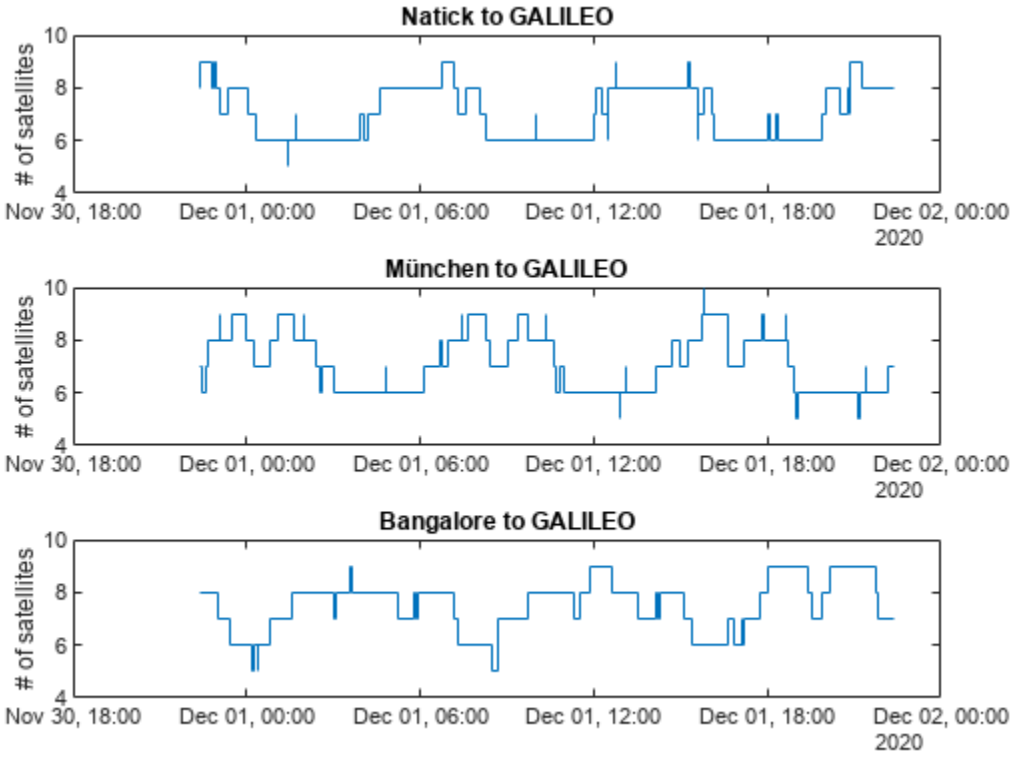


Compare Access Between Ground Stations

Calculate access status between each satellite and ground station using the accessStatus method. Each row of the output array corresponds with a satellite in the constellation. Each column corresponds with time steps in the scenario. A value of `True` indicates that the satellite can access the aircraft at that specific time sample. The second output of `accessStatus` contains the time steps of the scenario. Plot cumulative access for each ground station over the one day analysis window.

```
[statusUS, timeSteps] = accessStatus(accessUS);
statusDE = accessStatus(accessDE);
statusIN = accessStatus(accessIN);

% Sum cumulative access at each timestep
statusUS = sum(statusUS, 1);
statusDE = sum(statusDE, 1);
statusIN = sum(statusIN, 1);

subplot(3,1,1);
stairs(timeSteps, statusUS);
title("Natick to GALILEO")
ylabel("# of satellites")
subplot(3,1,2);
stairs(timeSteps, statusDE);
```

```
title("München to GALILEO")
ylabel("# of satellites")
subplot(3,1,3);
stairs(timeSteps, statusIN);
title("Bangalore to GALILEO")
ylabel("# of satellites")
```



Collect access interval metrics for each ground station in a table for comparison.

```
statusTable = [table(height(intervalsUS), height(intervalsDE), height(intervalsIN)); ...
    table(sum(intervalsUS.Duration)/3600, sum(intervalsDE.Duration)/3600, sum(intervalsIN.Duratio
    table(mean(intervalsUS.Duration/60), mean(intervalsDE.Duration/60), mean(intervalsIN.Duration
    table(mean(statusUS, 2), mean(statusDE, 2), mean(statusIN, 2)); ...
    table(min(statusUS), min(statusDE), min(statusIN)); ...
    table(max(statusUS), max(statusDE), max(statusIN))];
statusTable.Properties.VariableNames = ["Natick", "München", "Bangalore"];
statusTable.Properties.RowNames = ["Total # of intervals", "Total interval time (hrs)",...
    "Mean interval length (min)", "Mean # of satellites in view", ...
    "Min # of satellites in view", "Max # of satellites in view"];
statusTable
```

```
statusTable=6×3 table
                                Natick    München    Bangalore
                                ______    _______    _________

    Total # of intervals           40        40          31
    Total interval time (hrs)      167.88    169.95      180.42
    Mean interval length (min)     251.82    254.93      349.19
```

```
Mean # of satellites in view     7.018    7.1041      7.5337
Min # of satellites in view          5         5           5
Max # of satellites in view          9        10           9
```

Walker-Delta constellations like Galileo are evenly distributed across longitudes. Natick and München are located at similar latitudes, and therefore have very similar access characteristics with respect to the constellation. Bangalore is at a latitude closer to the equator. Despite having a lower number of individual access intervals, it has the highest average number of satellites in view, the highest overall interval time, and the longest average interval duration (by about 95 minutes). All locations always have at least 4 satellites in view, as is required for GNSS trilateration.

References

[1] Wertz, James R, David F. Everett, and Jeffery J. Puschell. *Space Mission Engineering: The New Smad*. Hawthorne, CA: Microcosm Press, 2011. Print.

[2] The European Space Agency: Galileo Facts and Figures. https://www.esa.int/Applications/Navigation/Galileo/Facts_and_figures

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer` | `Satellite` | `GroundStation` | `Access`

Functions

`satellite` | `hide` | `groundStation` | `accessIntervals`

More About

- “Satellite Scenario Key Concepts” on page 2-65
- “Satellite Scenario Overview” on page 2-74

Satellite Constellation Access to Ground Station

This example demonstrates how to set up access analysis between a ground station and conical sensors onboard a constellation of satellites. A ground station and a conical sensor belonging to a satellite are said to have access to one another if the ground station is inside the conical sensor's field of view and the conical sensor's elevation angle with respect to the ground station is greater than or equal to the latter's minimum elevation angle. The scenario involves a constellation of 40 low-Earth orbit satellites and a geographical site. Each satellite has a camera with a field of view of 90 degrees. The entire constellation of satellites is tasked with photographing the geographical site, which is located at 42.3001 degrees North and 71.3504 degrees West. The photographs are required to be taken between 12 May 2020 1:00 PM UTC and 12 May 2020 7:00 PM UTC when the site is adequately illuminated by the sun. In order to capture good quality pictures with minimal atmospheric distortion, the satellite's elevation angle with respect to the site should be at least 30 degrees (please note that 30 degrees was arbitrarily chosen for illustrative purposes). During the 6 hour interval, it is required to determine the times during which each satellite can photograph the site. It is also required to determine the percentage of time during this interval when at least one satellite's camera can see the site. This percentage quantity is termed the system-wide access percentage.

Create a Satellite Scenario

Create a satellite scenario using `satelliteScenario`. Use `datetime` to set the start time to 12-May-2020 1:00:00 PM UTC, and the stop time to 12-May-2020 7:00:00 PM UTC. Set the simulation sample time to 30 seconds.

```
startTime = datetime(2020,5,12,13,0,0);
stopTime = startTime + hours(6);
sampleTime = 30; % seconds
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc =
  satelliteScenario with properties:

         StartTime: 12-May-2020 13:00:00
          StopTime: 12-May-2020 19:00:00
        SampleTime: 30
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add Satellites to the Satellite Scenario

Use `satellite` to add satellites to the scenario from the TLE file `leoSatelliteConstellation.tle`. The TLE file defines the mean orbital parameters of 40 generic satellites in nearly circular low-Earth orbits at an altitude and inclination of approximately 500 km and 55 degrees respectively.

```
tleFile = "leoSatelliteConstellation.tle";
sat = satellite(sc,tleFile)
```

```
sat =
  1x40 Satellite array with properties:
```

```
    Name
    ID
    ConicalSensors
    Gimbals
    Transmitters
    Receivers
    Accesses
    GroundTrack
    Orbit
    OrbitPropagator
    MarkerColor
    MarkerSize
    ShowLabel
    LabelFontColor
    LabelFontSize
```

Add Cameras to the Satellites

Use `conicalSensor` to add a conical sensor to each satellite. These conical sensors represent the cameras. Specify their `MaxViewAngle` to be 90 degrees, which defines the field of view.

```
names = sat.Name + " Camera";
cam = conicalSensor(sat,"Name",names,"MaxViewAngle",90)
```

```
cam =
  1x40 ConicalSensor array with properties:

    Name
    ID
    MountingLocation
    MountingAngles
    MaxViewAngle
    Accesses
    FieldOfView
```

Define the Geographical Site to be Photographed in the Satellite Scenario

Use `groundStation` to add a ground station, which represents the geographical site to be photographed. Specify its `MinElevationAngle` to be 30 degrees. If latitude and longitude are not specified, they default to 42.3001 degrees North and 71.3504 degrees West.

```
name = "Geographical Site";
minElevationAngle = 30; % degrees
geoSite = groundStation(sc, ...
    "Name",name, ...
    "MinElevationAngle",minElevationAngle)
```

```
geoSite =
  GroundStation with properties:

                Name:  Geographical Site
                  ID:  81
            Latitude:  42.3 degrees
           Longitude:  -71.35 degrees
            Altitude:  0 meters
```

```
    MinElevationAngle:  30 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         MarkerColor:  [1 0.4118 0.1608]
          MarkerSize:  6
           ShowLabel:  true
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
```

Add Access Analysis Between the Cameras and the Geographical Site

Use `access` to add access analysis between each camera and the geographical site. The access analyses will be used to determine when each camera can photograph the site.

```
ac = access(cam,geoSite);

% Properties of access analysis objects
ac(1)
```

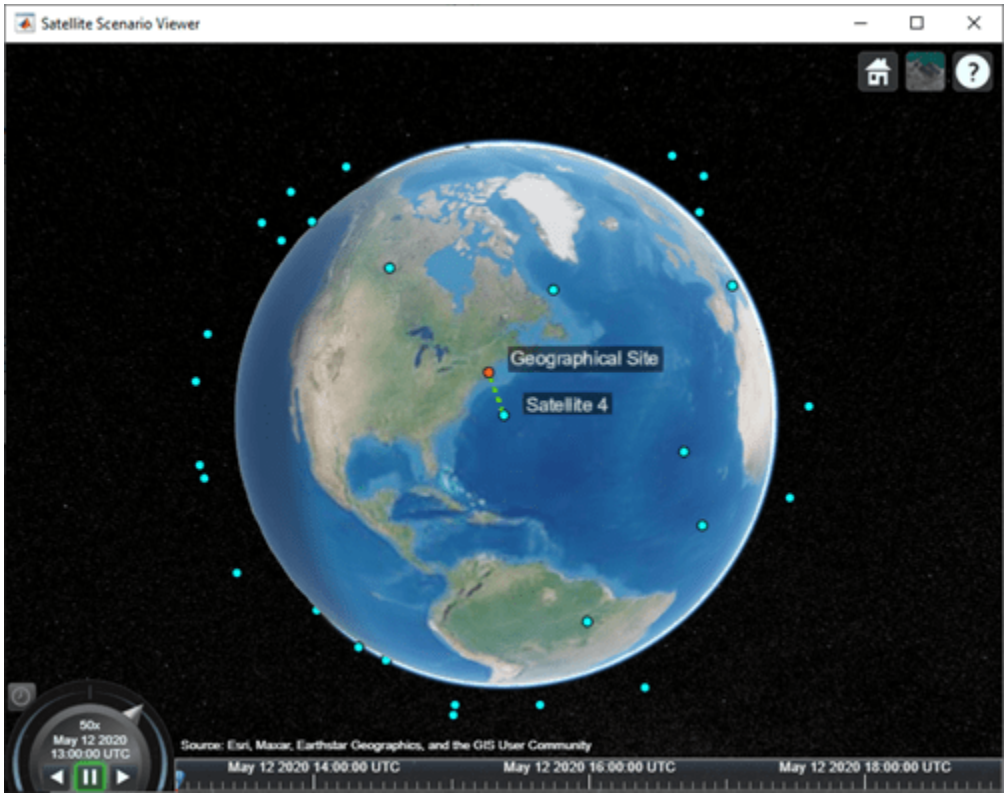
```
ans =
  Access with properties:

    Sequence:  [41 81]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

Visualize the Scenario

Use `satelliteScenarioViewer` to launch a satellite scenario viewer and visualize the scenario. Hide the orbits and labels of satellites and ground stations by setting the `ShowDetails` name-value pair to `false`. Show labels for the geographical site and Satellite 4, and center the satellite in view.

```
v = satelliteScenarioViewer(sc,"ShowDetails",false);
sat(4).ShowLabel = true;
geoSite.ShowLabel = true;
show(sat(4));
```

When the `ShowDetails` property is set to `false`, only satellites and ground stations will be shown. Labels, orbits, fields of view, and ground tracks will be hidden. Mouse over satellites and ground stations to show their labels. Click on a satellite or ground station to reveal its label, orbit, and any other hidden graphics. Click on the satellite or ground station again to dismiss them.



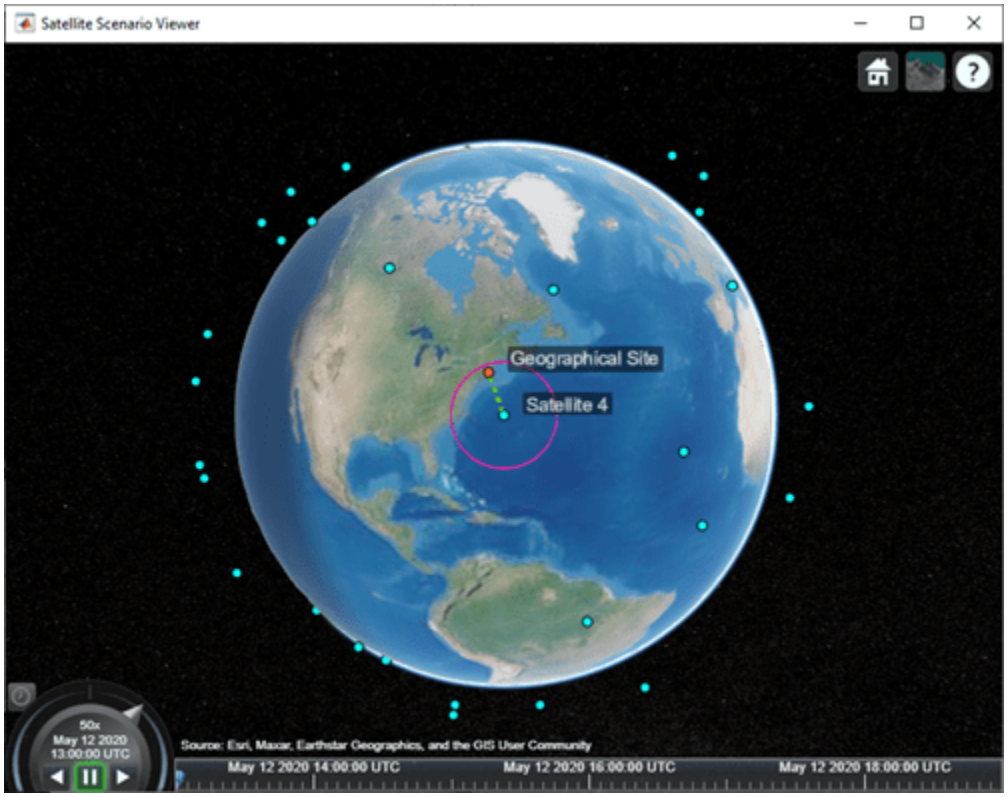
The viewer may be used as a visual confirmation that the scenario has been set up correctly. The violet line indicates that the camera on Satellite 4 and the geographical site have access to one another. This means that the geographical site is inside the camera's field of view and the camera's elevation angle with respect to the site is greater than or equal to 30 degrees. For the purposes of this scenario, this means that the camera can successfully photograph the site.

Visualize the Field Of View of the Camera

Use `fieldOfView` to visualize the field of view of each camera on Satellite 4.

```
fov = fieldOfView(cam([cam.Name] == "Satellite 4 Camera"))
```

```
fov =
  FieldOfView with properties:

        LineWidth: 1
        LineColor: [1 0.0745 0.6510]
    VisibilityMode: 'inherit'
```

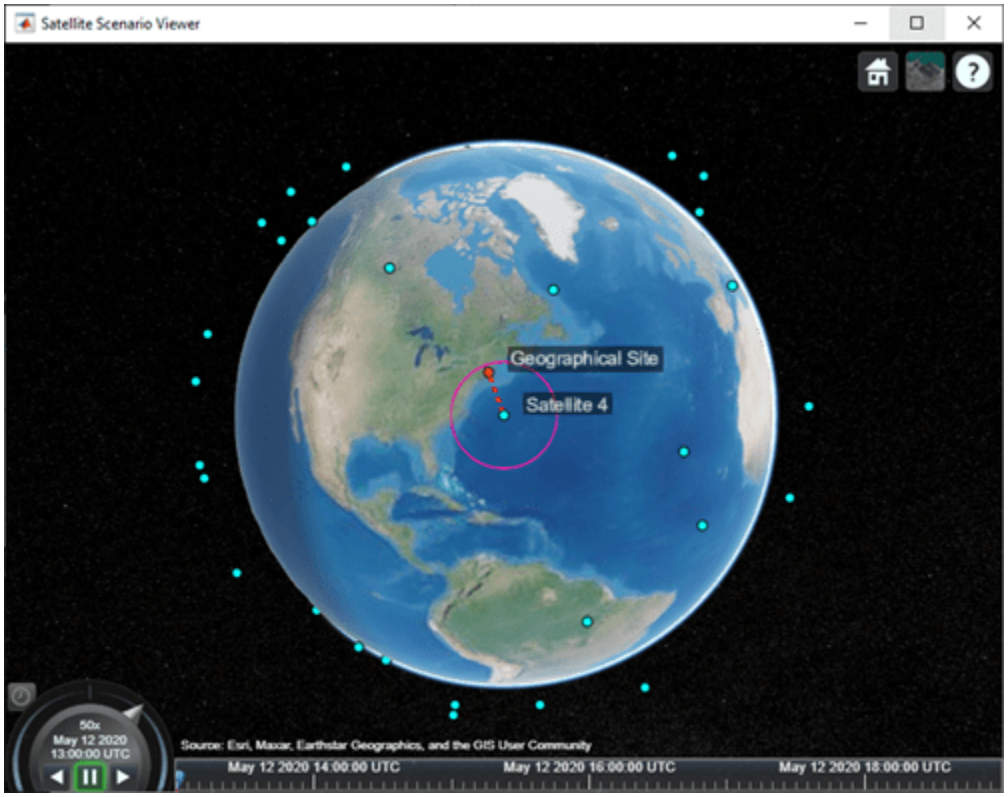


The presence of the geographical site inside the contour is a visual confirmation that it is inside the field of view of the camera onboard Satellite 4.

Customize the Visualizations

Change the color of access visualizations to red.

```
ac.LineColor = 'red';
```



Determine the Times when the Cameras can Photograph the Geographical Site

Use `accessIntervals` to determine the times when there is access between each camera and the geographical site. These are the times when the camera can photograph the site.

```
accessIntervals(ac)
```

```
ans=30×8 table
          Source                 Target           IntervalNumber        StartTime
    ____________________    ____________________    ______________    ____________________    __
    "Satellite 1 Camera"    "Geographical Site"          1           12-May-2020 13:36:00    12
    "Satellite 1 Camera"    "Geographical Site"          2           12-May-2020 15:23:00    12
    "Satellite 2 Camera"    "Geographical Site"          1           12-May-2020 14:30:30    12
    "Satellite 3 Camera"    "Geographical Site"          1           12-May-2020 13:28:30    12
    "Satellite 4 Camera"    "Geographical Site"          1           12-May-2020 13:00:00    12
    "Satellite 4 Camera"    "Geographical Site"          2           12-May-2020 14:46:00    12
    "Satellite 5 Camera"    "Geographical Site"          1           12-May-2020 16:28:30    12
    "Satellite 6 Camera"    "Geographical Site"          1           12-May-2020 17:05:30    12
    "Satellite 7 Camera"    "Geographical Site"          1           12-May-2020 16:20:00    12
    "Satellite 8 Camera"    "Geographical Site"          1           12-May-2020 15:18:00    12
    "Satellite 8 Camera"    "Geographical Site"          2           12-May-2020 17:03:30    12
    "Satellite 9 Camera"    "Geographical Site"          1           12-May-2020 17:55:30    12
    "Satellite 10 Camera"   "Geographical Site"          1           12-May-2020 18:44:30    12
    "Satellite 11 Camera"   "Geographical Site"          1           12-May-2020 18:39:30    12
    "Satellite 12 Camera"   "Geographical Site"          1           12-May-2020 17:58:00    12
    "Satellite 29 Camera"   "Geographical Site"          1           12-May-2020 13:09:30    12
      ⋮
```

The above table consists of the start and end times of each interval during which a given camera can photograph the site. The duration of each interval is reported in seconds. StartOrbit and EndOrbit are the orbit counts of the satellite that the camera is attached to when the access begins and ends. The count starts from the scenario start time.

Use `play` to visualize the simulation of the scenario from its start time to stop time. It can be seen that the green lines appear whenever the camera can photograph the geographical site.

```
play(sc);
```

Calculate System-Wide Access Percentage

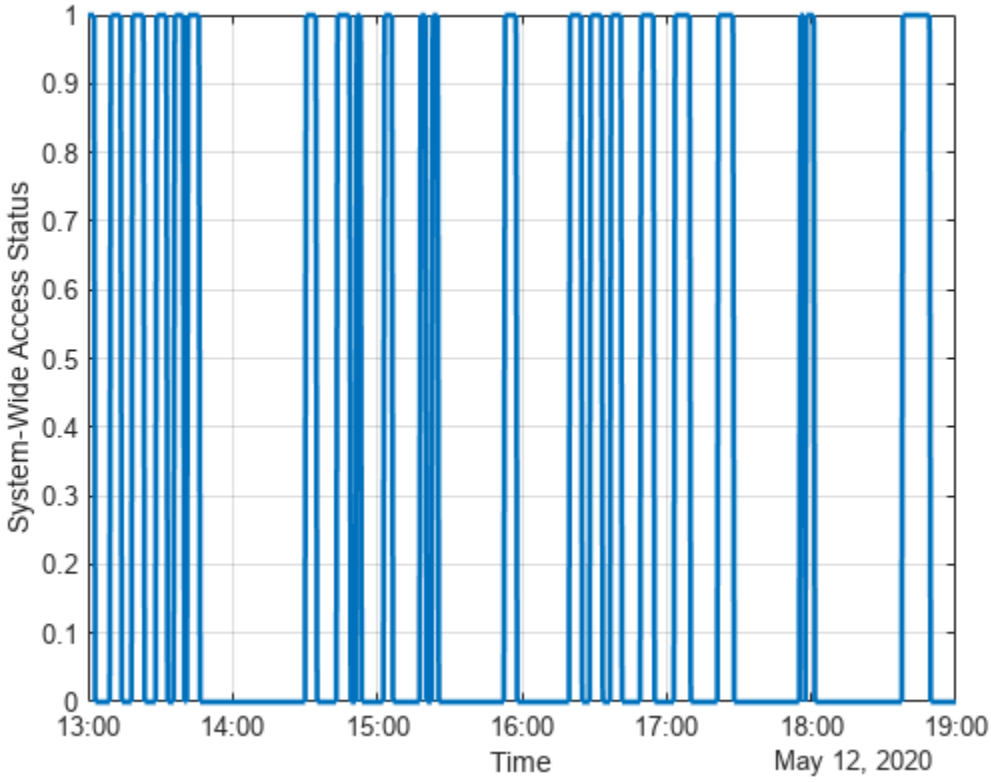
In addition to determining the times when each camera can photograph the geographical site, it is also required to determine the system-wide access percentage, which is the percentage of time from the scenario start time to stop time when at least one satellite can photograph the site. This is computed as follows:

- For each camera, calculate the access status history to the site using `accessStatus`. For a given camera, this is a row vector of logicals, where each element in the vector represents the access status corresponding to a given time sample. A value of `True` indicates that the camera can photograph the site at that specific time sample.
- Perform a logical `OR` on all these row vectors corresponding to access of each camera to the site. This will result in a single row vector of logicals, in which a given element is true if at least one camera can photograph the site at the corresponding time sample for a duration of one scenario sample time of 30 seconds.
- Count the number of elements in the vector whose value is `True`. Multiply this quantity by the sample time of 30 seconds to determine the total time in seconds when at least one camera can photograph the site.
- Divide this quantity by the scenario duration of 6 hours and multiply by 100 to get the system-wide access percentage.

```
for idx = 1:numel(ac)
    [s,time] = accessStatus(ac(idx));

    if idx == 1
        % Initialize system-wide access status vector in the first iteration
        systemWideAccessStatus = s;
    else
        % Update system-wide access status vector by performing a logical OR
        % with access status for the current camera-site access
        % analysis
        systemWideAccessStatus = or(systemWideAccessStatus,s);
    end
end
```

Use `plot` to plot the system-wide access status with respect to time.

```
plot(time,systemWideAccessStatus,"LineWidth",2);
grid on;
xlabel("Time");
ylabel("System-Wide Access Status");
```



Whenever system-wide access status is 1 (`True`), at least one camera can photograph the site.

Use `nnz` to determine the number of elements in `systemWideAccessStatus` whose value is `True`.

```
n = nnz(systemWideAccessStatus)
```

```
n = 203
```

Determine the total time when at least one camera can photograph the site. This is accomplished by multiplying the number of `True` elements by the scenario's sample time.

```
systemWideAccessDuration = n*sc.SampleTime % seconds
```

```
systemWideAccessDuration = 6090
```

Use `seconds` to calculate the total scenario duration.

```
scenarioDuration = seconds(sc.StopTime - sc.StartTime)
```

```
scenarioDuration = 21600
```

Calculate the system-wide access percentage.

```
systemWideAccessPercentage = (systemWideAccessDuration/scenarioDuration)*100
```

```
systemWideAccessPercentage = 28.1944
```

Improve the System-Wide Access Percentage by Making the Cameras Track the Geographical Site

The default attitude configuration of the satellites is such that their yaw axes point straight down towards nadir (the point on Earth directly below the satellite). Since the cameras are aligned with the yaw axis by default, they point straight down as well. As a result, the geographical site goes outside the field of view of the cameras before their elevation angle dips below 30 degrees. Therefore, the cumulative access percentage is limited by the cameras' field of view.

If instead the cameras always point at the geographical site, the latter is always inside the cameras' field of view as long as the Earth is not blocking the line of sight. Consequently, the system-wide access percentage will now be limited by the `MinElevationAngle` of the geographical site, as opposed to the cameras' field of view. In the former case, the access intervals began and ended when the site entered and left the camera's field of view. It entered the field of view some time after the camera's elevation angle went above 30 degrees, and left the field of view before its elevation angle dipped below 30 degrees. However, if the cameras constantly point at the site, the access intervals will begin when the elevation angle rises above 30 degrees and end when it dips below 30 degrees, thereby increasing the duration of the intervals. Therefore, the system-wide access percentage will increase as well.

Since the cameras are rigidly attached to the satellites, each satellite is required to be continuously reoriented along its orbit so that its yaw axis tracks the geographical site. As the cameras are aligned with the yaw axis, they too will point at the site. Use `pointAt` to make each satellite's yaw axis track the geographical site.

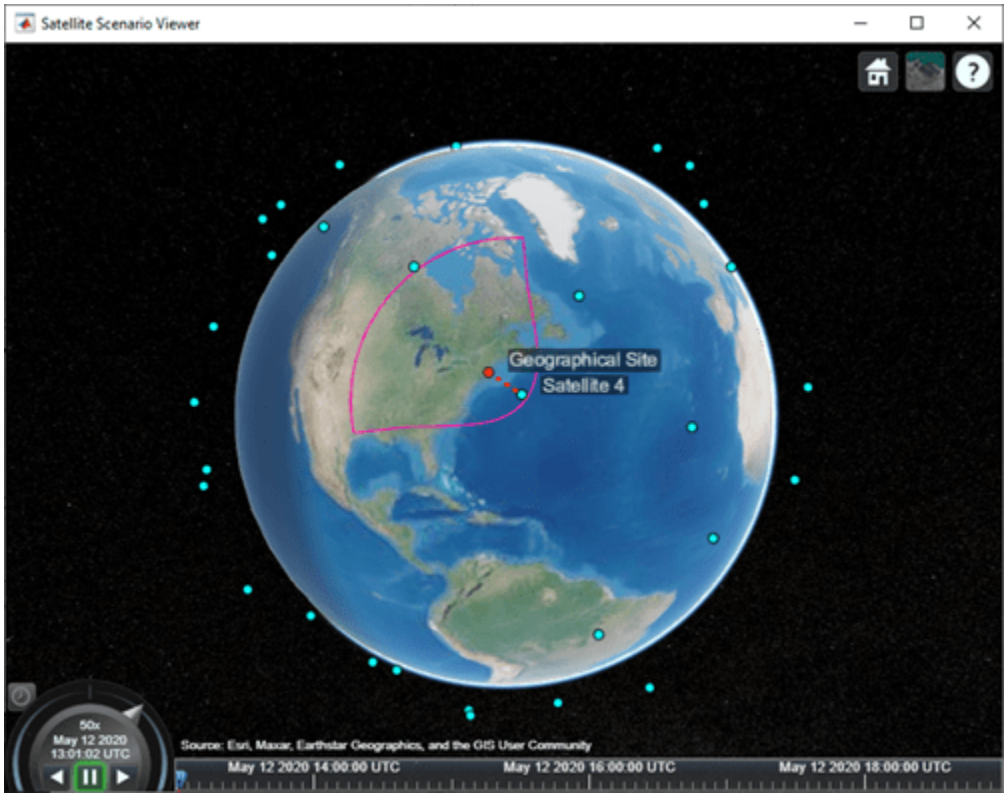
```
pointAt(sat,geoSite);
```

Re-calculate the system-wide access percentage.

```
% Calculate system-wide access status
for idx = 1:numel(ac)
    [s,time] = accessStatus(ac(idx));

    if idx == 1
        % Initialize system-wide access status vector in the first iteration
        systemWideAccessStatus = s;
    else
        % Update system-wide access status vector by performing a logical OR
        % with access status for the current camera-site combination
        systemWideAccessStatus = or(systemWideAccessStatus,s);
    end
end

% Calculate system-wide access percentage
n = nnz(systemWideAccessStatus);
systemWideAccessDuration = n*sc.SampleTime;
systemWideAccessPercentageWithTracking = (systemWideAccessDuration/scenarioDuration)*100

systemWideAccessPercentageWithTracking = 38.3333
```

The system-wide access percentage has improved by about 36%. This is the result of the cameras continuously pointing at the geographical site. This can be visualized by using `play` again.

```
play(sc)
```



The field of view contour is no longer circular because the camera is not pointing straight down anymore as it is tracking the geographical site.

Exploring the Example

This example demonstrated how to set up access analysis between a ground station and conical sensors onboard a constellation of satellites. The conical sensors represented cameras onboard the satellites, and the ground station represented the geographical site to be photographed. The times at which cameras onboard the satellites can photograph the geographical site were determined using access analysis. Additionally, system-wide access percentage was computed to determine the percentage of time during a 6 hour period when at least one satellite can photograph the site. It was seen that these results depended on the direction at which the cameras were pointing.

These results are also a function of:

- Orbit of the satellites
- `MinElevationAngle` of the geographical site
- Mounting position and location of the cameras with respect to the satellites
- Field of view (`MaxViewAngle`) of the cameras if they are not continuously pointing at the geographical site

Modify the above parameters to your requirements and observe their influence on the access intervals and system-wide access percentage. The orbit of the satellites can be changed by explicitly specifying their Keplerian orbital elements using `satellite`. Additionally, the cameras can be mounted on `gimbals`, which can be rotated independent of the satellite. This way, the satellites can

point straight down (the default behavior), while the gimbals can be configured so that the cameras independently track the geographical site.

See Also

`datetime` | `satelliteScenario` | `satelliteScenarioViewer` | `Satellite` | `GroundStation`

Comparison of Orbit Propagators

This example compares the orbits predicted by the Two-Body-Keplerian, Simplified General Perturbations-4 (SGP4) and Simplified Deep-Space Perturbations-4 (SDP4) orbit propagators. An orbit propagator is a solver that calculates the position and velocity of an object whose motion is predominantly influenced by gravity from celestial bodies. The Two-Body-Keplerian orbit propagator is based on the relative two-body model that assumes a spherical gravity field for the Earth and neglects third body effects and other environmental perturbations, and hence, is the least accurate. The SGP4 orbit propagator accounts for secular and periodic orbital perturbations caused by Earth's geometry and atmospheric drag, and is applicable to near-Earth satellites whose orbital period is less than 225 minutes. The SDP4 orbit propagator builds upon SGP4 by accounting for solar and lunar gravity, and is applicable to satellites whose orbital period is greater than or equal to 225 minutes. The default orbit propagator for `satelliteScenario` is SGP4 for satellites whose orbital period is less than 225 minutes, and SDP4 otherwise.

Create a Satellite Scenario

Create a satellite scenario by using the `satelliteScenario` function. Set the start time to 11-May-2020 12:35:38 PM UTC, and the stop time to 13-May-2020 12:35:38 PM UTC, by using the `datetime` function. Set the sample time to 60 seconds.

```
startTime = datetime(2020,5,11,12,35,38);
stopTime = startTime + days(2);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc =
  satelliteScenario with properties:

         StartTime: 11-May-2020 12:35:38
          StopTime: 13-May-2020 12:35:38
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Add Satellites to the Satellite Scenario

Add three satellites to the satellite scenario from the two-line element (TLE) file `eccentricOrbitSatellite.tle` by using the `satellite` function. TLE is a data format used for encoding the orbital elements of an Earth-orbiting object defined at a specific time. Assign a Two-Body-Keplerian orbit propagator to the first satellite, SGP4 to the second satellite, and SDP4 to the third satellite.

```
tleFile = "eccentricOrbitSatellite.tle";
satTwoBodyKeplerian = satellite(sc,tleFile, ...
    "Name","satTwoBodyKeplerian", ...
    "OrbitPropagator","two-body-keplerian")
```

```
satTwoBodyKeplerian =
  Satellite with properties:
```

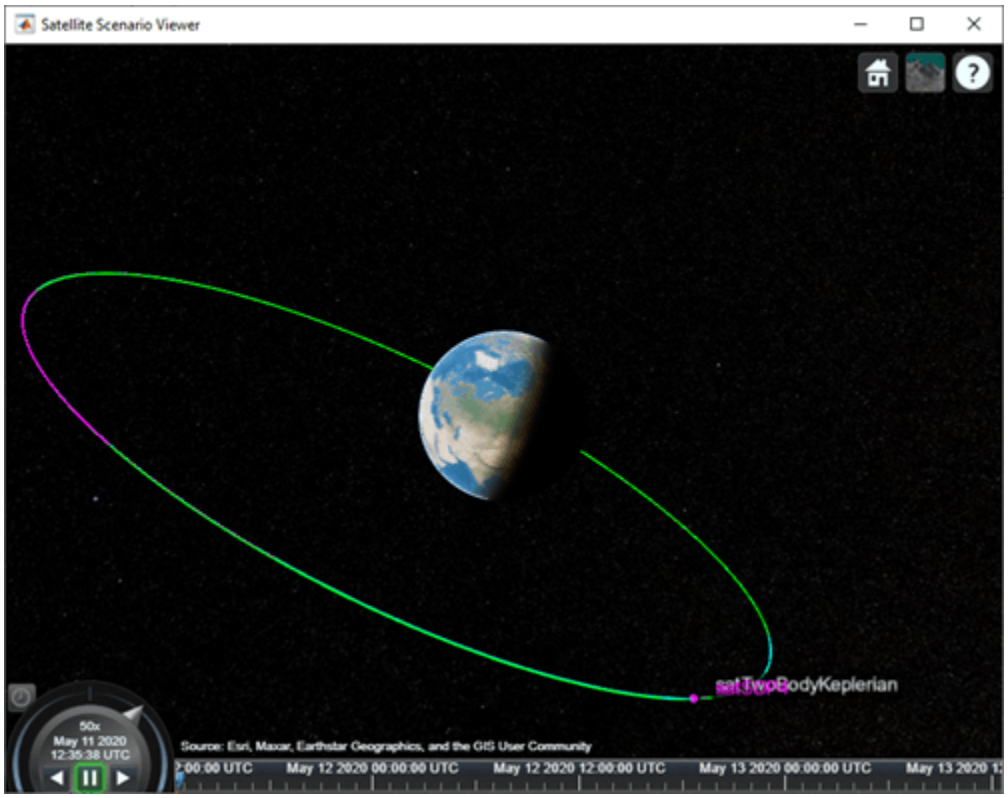
```
                Name:  satTwoBodyKeplerian
                  ID:  1
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
               Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
     OrbitPropagator:  two-body-keplerian
         MarkerColor:  [0.059 1 1]
          MarkerSize:  6
           ShowLabel:  true
      LabelFontColor:  [1 1 1]
       LabelFontSize:  15


satSGP4 = satellite(sc,tleFile, ...
    "Name","satSGP4", ...
    "OrbitPropagator","sgp4")

satSGP4 =
  Satellite with properties:

                Name:  satSGP4
                  ID:  2
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
               Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
     OrbitPropagator:  sgp4
         MarkerColor:  [0.059 1 1]
          MarkerSize:  6
           ShowLabel:  true
      LabelFontColor:  [1 1 1]
       LabelFontSize:  15


satSDP4 = satellite(sc,tleFile, ...
    "Name","satSDP4", ...
    "OrbitPropagator","sdp4")

satSDP4 =
  Satellite with properties:

                Name:  satSDP4
                  ID:  3
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
               Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
     OrbitPropagator:  sdp4
         MarkerColor:  [0.059 1 1]
```

```
      MarkerSize:  6
       ShowLabel:  true
  LabelFontColor:  [1 1 1]
   LabelFontSize:  15
```

Visualize the Satellites and their Orbits

Launch a satellite scenario viewer and visualize the satellite scenario by using the `satelliteScenarioViewer` function. Set the visualizations of `satTwoBodyKeplerian` to red, `satSGP4` to green, and `satSDP4` to magenta.

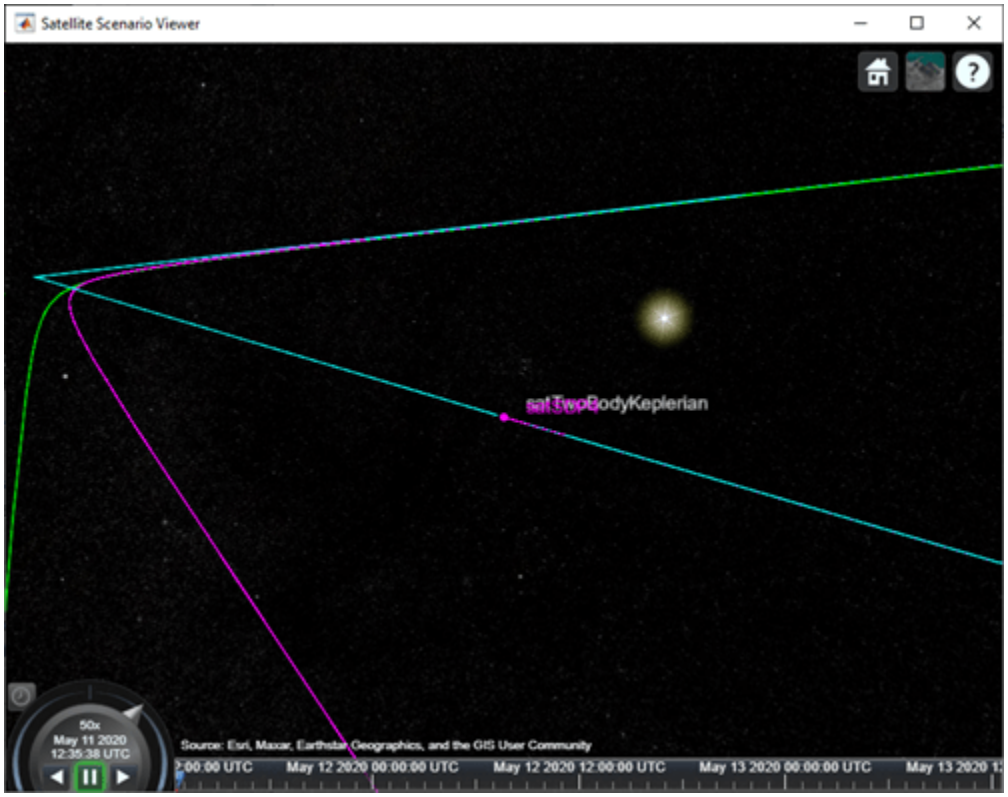
```
v = satelliteScenarioViewer(sc);
satSGP4.MarkerColor = [0 1 0];
satSGP4.Orbit.LineColor = [0 1 0];
satSGP4.LabelFontColor = [0 1 0];
satSDP4.MarkerColor = [1 0 1];
satSDP4.Orbit.LineColor = [1 0 1];
satSDP4.LabelFontColor = [1 0 1];
```



Focus the camera on `satTwoBodyKeplerian` by using the `camtarget` function.

```
camtarget(v,satTwoBodyKeplerian);
```

Left-click anywhere inside the satellite scenario viewer window and move the mouse while holding the click to pan the camera. Adjust the zoom level using the scroll wheel to bring all three satellites into view.



Visualize a Dynamic Animation of the Satellite Movement

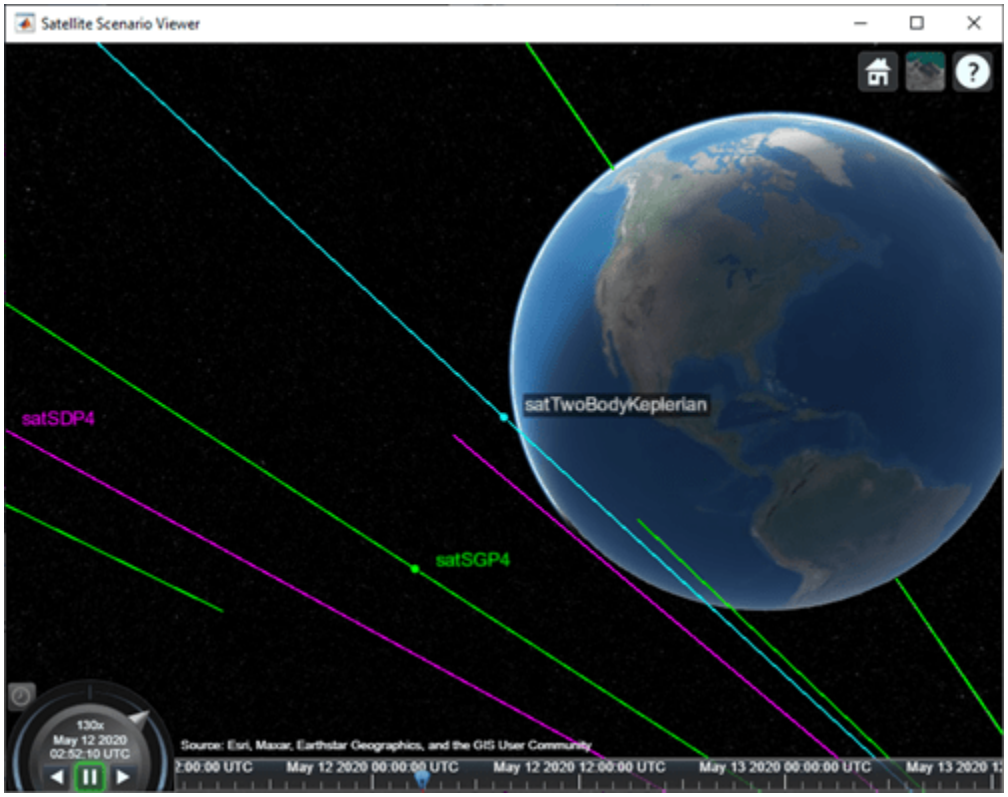
Visualize the movement of the satellites by using the `play` function on the satellite scenario. The `play` function simulates the satellite scenario from the specified `StartTime` to `StopTime` using a step size specified by `SampleTime`, and plays the results on the satellite scenario viewer.

```
play(sc)
```

Use the playback controls located at the bottom of the satellite scenario viewer window to control the playback speed and direction. Focus the camera again on `satTwoBodyKeplerian` by using the `camtarget` function, and bring all three satellites into view by adjusting the zoom level.

```
camtarget(v,satTwoBodyKeplerian);
```

The positions of the three satellites diverge over time.



Obtain the Position and Velocity History of the Satellites

Return the position and velocity history of the satellites in the Geocentric Celestial Reference Frame (GCRF) by using the `states` function.

```
[positionTwoBodyKeplerian,velocityTwoBodyKeplerian,time] = states(satTwoBodyKeplerian);
[positionSGP4,velocitySGP4] = states(satSGP4);
[positionSDP4,velocitySDP4] = states(satSDP4);
```

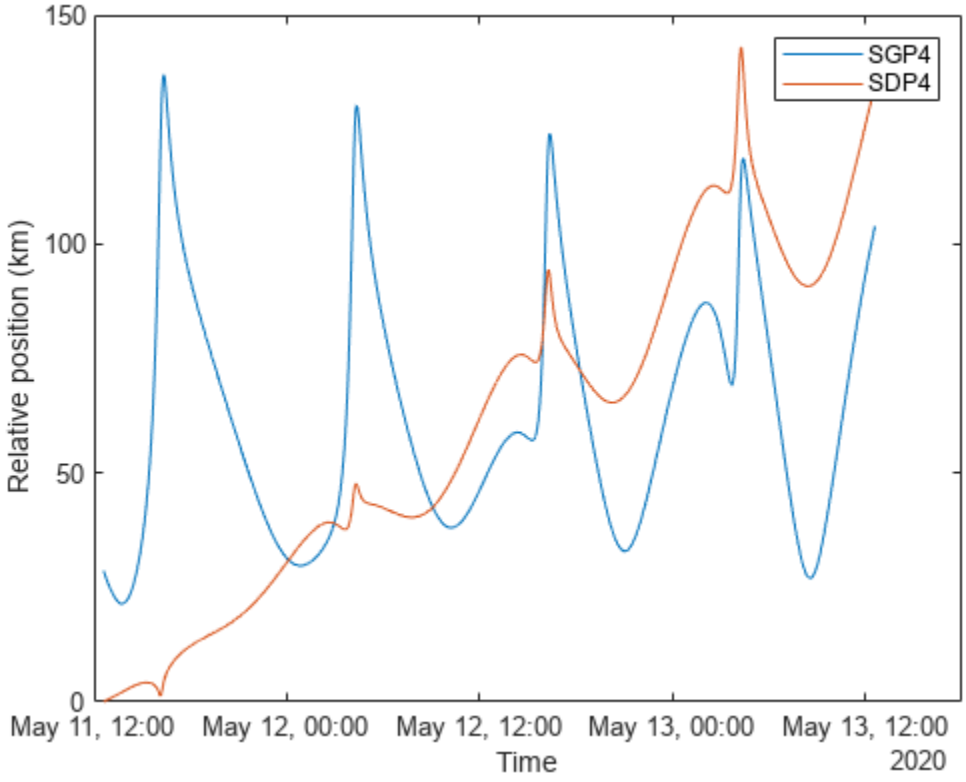
Plot Magnitude of Relative Position with Respect to Two-Body-Keplerian Prediction

Calculate the magnitude of the relative position of `satSGP4` and `satSDP4` with respect to `satTwoBodyKeplerian` by using the `vecnorm` function.

```
sgp4RelativePosition = vecnorm(positionSGP4 - positionTwoBodyKeplerian,2,1);
sdp4RelativePosition = vecnorm(positionSDP4 - positionTwoBodyKeplerian,2,1);
```

Plot the magnitude of the relative positions in kilometers of `satSGP4` and `satSDP4` with respect to that of `satTwoBodyKeplerian` by using the `plot` function.

```
sgp4RelativePositionKm = sgp4RelativePosition/1000;
sdp4RelativePositionKm = sdp4RelativePosition/1000;
plot(time,sgp4RelativePositionKm,time,sdp4RelativePositionKm)
xlabel("Time")
ylabel("Relative position (km)")
legend("SGP4","SDP4")
```



The initial relative position of `satSGP4` is non-zero and that of `satSDP4` is zero because the initial positions of `satTwoBodyKeplerian` and `satSDP4` are calculated from the TLE file using the SDP4 orbit propagator, while the initial position of `satSGP4` is calculated using the SGP4 orbit propagator. Over time, the position of `satSDP4` deviates from that of `satTwoBodyKeplerian` because the subsequent positions of the former are calculated using the SDP4 orbit propagator, while those of the latter are calculated using the Two-Body-Keplerian orbit propagator. The SDP4 orbit propagator provides higher precision because unlike the Two-Body-Keplerian orbit propagator, it accounts for oblateness of the Earth, atmospheric drag, and gravity from the sun and the moon.

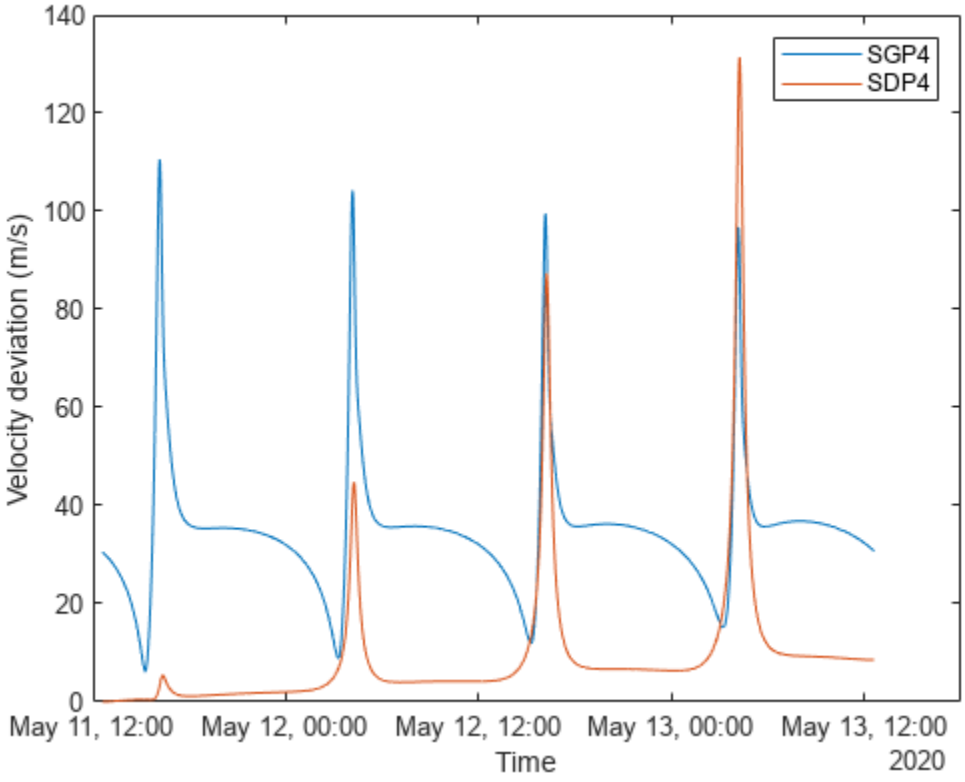
Plot Magnitude of Relative Velocity with Respect to Two-Body-Keplerian Prediction

Calculate the magnitude of the relative velocity of `satSGP4` and `satSDP4` with respect to `satTwoBodyKeplerian` by using the `vecnorm` function.

```
sgp4RelativeVelocity = vecnorm(velocitySGP4 - velocityTwoBodyKeplerian,2,1);
sdp4RelativeVelocity = vecnorm(velocitySDP4 - velocityTwoBodyKeplerian,2,1);
```

Plot the magnitude of the relative velocities in meters per second of `satSGP4` and `satSDP4` with respect to `satTwoBodyKeplerian` by using the `plot` function.

```
plot(time,sgp4RelativeVelocity,time,sdp4RelativeVelocity)
xlabel("Time")
ylabel("Velocity deviation (m/s)")
legend("SGP4","SDP4")
```



The initial relative velocity of `satSDP4` is zero because just like the initial position, the initial velocity of `satTwoBodyKeplerian` and `satSDP4` are also calculated from the TLE file using the SDP4 orbit propagator. Over time, the velocity of `satSDP4` deviates from that of `satTwoBodyKeplerian` because at all other times, the velocity of `satTwoBodyKeplerian` is calculated using the Two-Body-Keplerian orbit propagator, which has lower precision when compared to that of the SDP4 orbit propagator that is used for calculating the velocity of `satSDP4`. The spikes correspond to the periapsis (the closest point in the orbit from the center of mass of the Earth), where the magnitudes of the velocity errors are pronounced.

Conclusion

The deviations in the plots are the result of varying levels of accuracy of the three orbit propagators. The Two-Body-Keplerian orbit propagator is the least accurate as it assumes that the gravity field of the Earth is spherical, and also neglects all other sources of orbital perturbations. The SGP4 orbit propagator is more accurate as it accounts for the oblateness of the Earth and atmospheric drag. The SDP4 orbit propagator is the most accurate among the three because it also accounts for solar and lunar gravity, which is more pronounced in this example because the orbital period is greater than 225 minutes, thereby taking the satellite farther away from the Earth.

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer` | `Satellite`

Functions
`camtarget` | `play` | `satellite`

More About

- “Satellite Scenario Key Concepts” on page 2-65
- “Satellite Scenario Overview” on page 2-74

Detect and Track LEO Satellite Constellation with Ground Radars

This example shows how to import a Two-Line Element (TLE) file of a satellite constellation, simulate radar detections of the constellation, and track the constellation.

The task of populating and maintaining a catalog of space objects orbiting Earth is crucial in space surveillance. This task consists of several processes: detecting and identifying new objects and adding them to the catalog, updating known objects orbits in the catalog, tracking orbit changes throughout their lifetime, and predicting reentries in the atmosphere. In this example, we are study how to detect and track new satellites and add them to a catalog.

To guarantee safe operations in space and prevent collisions with other satellites or known debris, it important to correctly detect and track newly launched satellites. Space agencies typically share prelaunch information, which can be used to select a search strategy. A Low Earth Orbit (LEO) satellite search strategy consisting of fence-type radar systems is commonly used. A fence-type radar system searches a finite volume in space and detects satellites as they pass through its field of view. This strategy can detect and track newly launched constellation quickly. [1]

Importing a satellite constellation from a TLE file

Two-Line Element sets are a common data format to save orbital information of satellites. You can use the `satelliteScenario` object to import satellite orbits defined in a TLE file. By default, the imported satellite orbits are propagated using the SGP4 orbit propagation algorithm which provides good accuracy for LEO objects. In this example, these orbits provide with the ground truth to test the radar tracking system capability to detect newly launched satellites.

```
% Create a satellite scenario
satscene = satelliteScenario;
% Add satellites from TLE file.
tleFile = "leoSatelliteConstellation.tle";
constellation = satellite(satscene, tleFile);
```

Use the satellite scenario viewer to visualize the constellation.

```
play(satscene);
```

Simulating synthetic detections and track constellation

Modeling space surveillance radars

Define two stations with fan-shaped radar beams looking into space. The fans cut through the satellite orbits to maximize the number of detections. The radar stations located on North America form an East-West fence.

```
% First station coordinates in LLA
station1 = [48 -80 0];

% Second station coordinates in LLA
station2 = [50 -117 0];
```

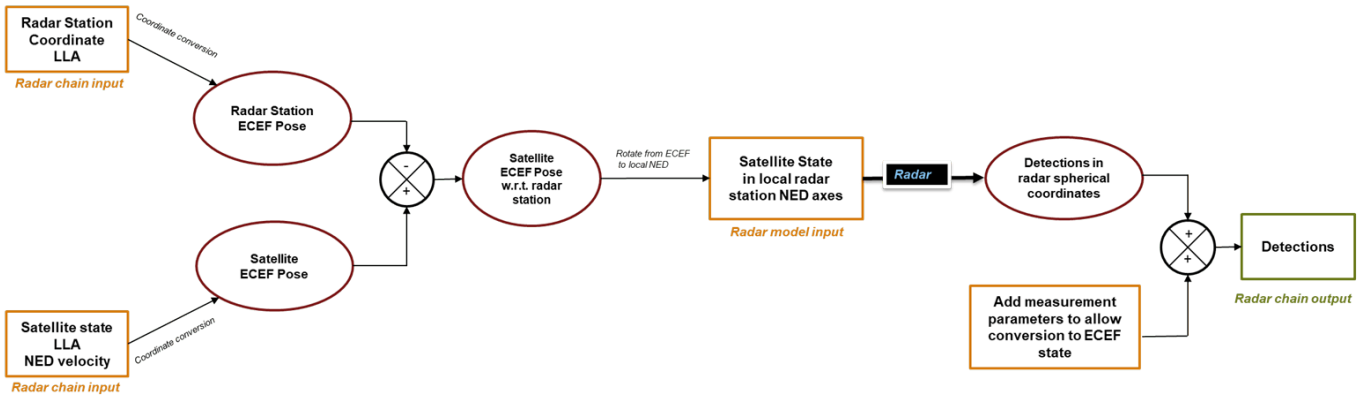
Each station is equipped with a radar, which is modeled by using a `fusionRadarSensor` object. In order to detect satellites in the LEO range, the radar has the following requirements:

- Detecting a 10 dBsm object up to 2000 km away
- Resolving objects horizontally and vertically with a precision of 100 m at 2000 km range
- Having a field of view of 120 degrees in azimuth and 30 degrees in elevation
- Looking up into space

```
% Create fan-shaped monostatic radars
fov = [120;40];
radar1 = fusionRadarSensor(1,...
    'UpdateRate',0.1,... 10 sec
    'ScanMode','No scanning',...
    'MountingAngles',[0 90 0],... look up
    'FieldOfView',fov,... degrees
    'ReferenceRange',2000e3,... m
    'RangeLimits',  [0 2000e3], ... m
    'ReferenceRCS', 10,... dBsm
    'HasFalseAlarms',false,...
    'HasNoise', true,...
    'HasElevation',true,...
    'AzimuthResolution',0.03,... degrees
    'ElevationResolution',0.03,... degrees
    'RangeResolution',2000, ... m % accuracy ~= 2000 * 0.05 (m)
    'DetectionCoordinates','Sensor Spherical',...
    'TargetReportFormat','Detections');

radar2 = clone(radar1);
radar2.SensorIndex = 2;
```

Radar Processing Chain

In this example, several coordinate conversions and axes transformation are performed to properly run the radar tracking chain. The diagram below illustrates how the inputs defined above are transformed and passed to the radar.



In the first step, you calculate each satellite pose in the local radar station NED axes. You achieve this by first obtaining the ground station ECEF pose and converting the satellite position and velocity to the ECEF coordinates. The radar input is obtained by taking the differences between the satellite pose and the ground station pose and rotating the differences into ground station local NED axes. See the **`assembleRadarInputs`** supporting function for the implementation details.

In the second step, you add the required information to the detection object so that the tracker can operate with an ECEF state. You use the `MeasurementParameters` property in each object detection to achieve that purpose, as shown in the **`addMeasurementParams`** supporting function.

Defining a tracker

The radar models you defined above output detections. To estimate the satellite orbits, you use a tracker. The Sensor Fusion and Tracking Toolbox™ provides a variety of multi-object trackers. In this example, you choose a Joint Probabilistic Data Association (JPDA) tracker because it offers a good balance of tracking performance and computational cost.

You need to define a tracking filter for the tracker. You can use a lower fidelity model than SGP4, such as a Keplerian integration of the equation of motion, to track the satellite. Often, the lack of fidelity in the motion model of targets is compensated by measurement updates and incorporating process noise in the filter. The supporting function **`initKeplerUKF`** defines the tracking filter.

```
% Define the tracker
tracker = trackerJPDA('FilterInitializationFcn',@initKeplerUKF,...
    'HasDetectableTrackIDsInput',true,...
    'ClutterDensity',1e-40,...
    'AssignmentThreshold',1e4,...
    'DeletionThreshold',[5 8],...
    'ConfirmationThreshold',[5 8]);
```

Running the simulation

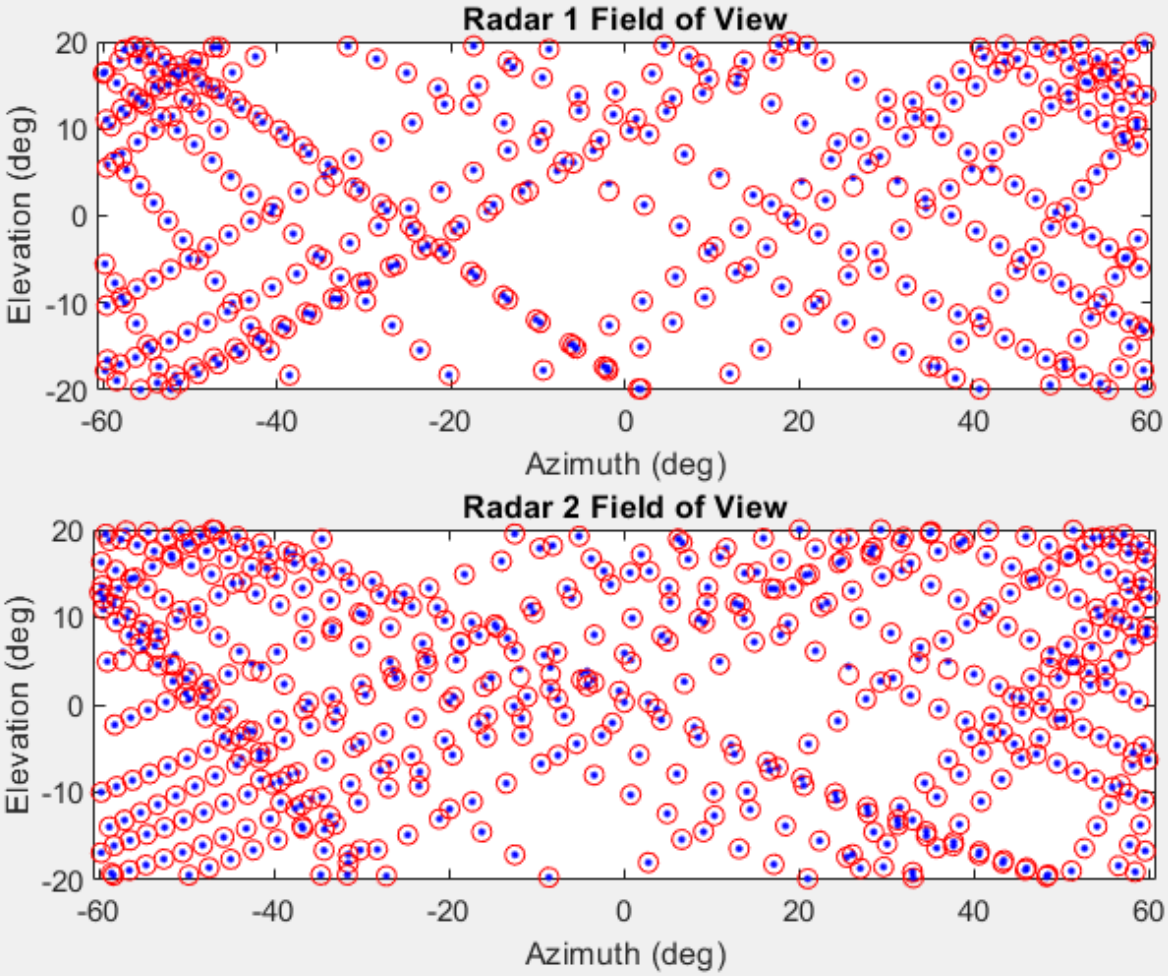
In the remainder of this example, you step through the scenario to simulate radar detections and track satellites. This section uses the `trackingGlobeViewer` for visualization. You use this class to display sensor and tracking data with uncertainty ellipses and show the true position of each satellite.

```
viewer = trackingGlobeViewer('ShowDroppedTracks',false, 'PlatformHistoryDepth',700);

% Define coverage configuration of each radar and visualize it on the globe
ned1 = dcmecef2ned(station1(1), station1(2));
ned2 = dcmecef2ned(station2(1), station2(2));
covcon(1) = coverageConfig(radar1,lla2ecef(station1),quaternion(ned1,'rotmat','frame'));
covcon(2) = coverageConfig(radar2,lla2ecef(station2),quaternion(ned2, 'rotmat','frame'));
plotCoverage(viewer, covcon, 'ECEF');
```

You first generate the entire history of the states of the constellation over 5 hours. Then, you simulate radar detections and generate tracks in a loop.

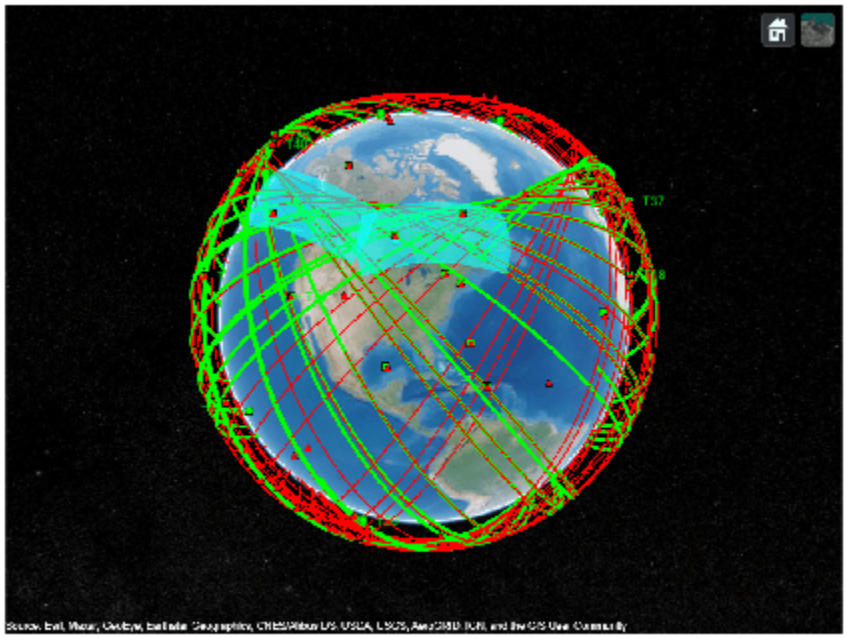
```
satscene.StopTime = satscene.StartTime + hours(5);
satscene.SampleTime = 10;
numSteps = ceil(seconds(satscene.StopTime - satscene.StartTime)/satscene.SampleTime);

% Get constellation positions and velocity over the course of the simulation
plats = repmat(...
    struct('PlatformID',0,'Position',[0 0 0], 'Velocity', [0 0 0]),...
    numSteps, 40);
for i=1:numel(constellation)
    [pos, vel] = states(constellation(i),'CoordinateFrame',"ECEF");
    for j=1:numSteps
        plats(j,i).Position = pos(:,j)';
        plats(j,i).Velocity = vel(:,j)';
        plats(j,i).PlatformID = i;
    end
```

```
end

% Initialize tracks and tracks log
confTracks = objectTrack.empty(0,1);
trackLog = cell(1,numSteps);

% Initialize radar plots
radarplt = helperRadarPlot(fov);

% Set random seed for reproducible results
s = rng;
rng(2020);
step = 0;
while step < numSteps
    time = step*satscene.SampleTime;
    step = step + 1;

    % Generate detections of the constellation following the radar
    % processing chain
    targets1 = assembleRadarInputs(station1, plats(step,:));
    [dets1,numdets1] = radar1(targets1, time);
    dets1 = addMeasurementParams(dets1,numdets1,station1);

    targets2 = assembleRadarInputs(station2, plats(step,:));
    [dets2, numdets2] = radar2(targets2, time);
    dets2 = addMeasurementParams(dets2, numdets2, station2);

    detections = [dets1; dets2];
    updateRadarPlots(radarplt,targets1, targets2 ,dets1, dets2)

    % Generate and update tracks
    detectableInput = isDetectable(tracker,time, covcon);
    if ~isempty(detections) || isLocked(tracker)
        [confTracks,~,~,info] = tracker(detections,time,detectableInput);
    end
    trackLog{step} = confTracks;

    % Update viewer
    plotPlatform(viewer, plats(step,:),'ECEF', 'Color',[1 0 0],'LineWidth',1);
    plotDetection(viewer, detections,'ECEF');
    plotTrack(viewer, confTracks, 'ECEF','Color',[0 1 0], 'LineWidth',3);

end
```



The plot above shows the orbits (blue dots) and the detections (red circles) from the point of view of each radar.

```
% Restore previous random seed state
rng(s);

figure;
snapshot(viewer);
```



After 5 hours of tracking, about half the constellation is tracked successfully. Maintaining tracks with a partial orbit coverage is challenging since satellites can often stay undetected for long periods of time in this configuration. In this example, there are only two radar stations. Additional tracking stations are expected to generate better tracking performance. The assignment metrics, which evaluate the tracking performance by comparing between the true objects and the tracks, are shown below.

```
% Show Assignment metrics
truthIdFcn = @(x)[x.PlatformID];

tam = trackAssignmentMetrics('DistanceFunctionFormat','custom',...
    'AssignmentDistanceFcn',@distanceFcn,...
    'DivergenceDistanceFcn',@distanceFcn,...
    'TruthIdentifierFcn',truthIdFcn,...
    'AssignmentThreshold',1000,...
    'DivergenceThreshold',2000);

for i=1:numSteps
    % Extract the tracker and ground truth at the i-th tracker update
    tracks = trackLog{i};
    truths = plats(i,:);
    % Extract summary of assignment metrics against tracks and truths
    [trackAM,truthAM] = tam(tracks, truths);
end


% Show cumulative metrics for each individual recorded truth object
results = truthMetricsTable(tam);
results(:,{'TruthID','AssociatedTrackID','BreakLength','EstablishmentLength'})

ans=40×4 table
    TruthID    AssociatedTrackID    BreakLength    EstablishmentLength
```

1	52	5	232
2	24	0	594
3	4	0	56
4	55	11	492
5	18	0	437
6	48	8	811
7	21	0	513
8	27	0	661
9	39	0	1221
10	50	0	1504
11	43	0	1339
12	37	0	1056
13	NaN	0	1800
14	NaN	0	1800
15	NaN	0	1800
16	NaN	0	1800
⋮			

The table above lists 40 satellites in the launched constellation and shows the tracked satellites with associated track IDs. A track ID of value NaN indicates that the satellite is not tracked by the end of the simulation. This either means that the orbit of the satellite did not pass through the field of view of one of the two radars or the track of the satellite has been dropped. The tracker can drop a track due to an insufficient number of initial detections, which leads to a large uncertainty on the estimate. Alternately, the tracker can drop the track if the satellite is not re-detected soon enough, such that the lack of updates leads to divergence and eventually deletion.

Summary

In this example, you have learned how to use the `satelliteScenario` object from the Aerospace Toolbox™ to import orbit information from TLE files. You propagated the satellite trajectories using SGP4 and visualized the scenario using the Satellite Scenario Viewer. You learned how to use the radar and tracker models from the Sensor Fusion and Tracking Toolbox™ to model a space surveillance radar tracking system. The constructed tracking system can predict the estimated orbit of each satellite using a low fidelity model.

Supporting functions

`initKeplerUKF` initializes an Unscented Kalman filter using the Keplerian motion model. Set `Alpha` = 1, `Beta` = 0, and `Kappa` = 0 to ensure robustness of the unscented filter over long prediction period.

```
function filter = initKeplerUKF(detection)

radarsphmeas = detection.Measurement;
[x, y, z] = sph2cart(deg2rad(radarsphmeas(1)),deg2rad(radarsphmeas(2)),radarsphmeas(3));
radarcartmeas = [x y z];
Recef2radar = detection.MeasurementParameters.Orientation;
ecefmeas = detection.MeasurementParameters.OriginPosition + radarcartmeas*Recef2radar;
% This is equivalent to:
% Ry90 = [0 0 -1 ; 0 1 0; 1 0 0]; % frame rotation of 90 deg around y axis
% nedmeas(:) = Ry90' * radarcartmeas(:);
% ecefmeas = lla2ecef(lla) +  nedmeas * dcmecef2ned(lla(1),lla(2));
initState = [ecefmeas(1); 0; ecefmeas(2); 0; ecefmeas(3); 0];
```

```
sigpos = 2;% m
sigvel = 0.5;% m/s^2

filter = trackingUKF(@keplerorbit,@cvmeas,initState,...
    'Alpha', 1, 'Beta', 0, 'Kappa', 0, ...
    'StateCovariance', diag(repmat([1000, 10000].^2,1,3)),...
    'ProcessNoise',diag(repmat([sigpos, sigvel].^2,1,3)));

end

function state = keplerorbit(state,dt)
% keplerorbit performs numerical integration to predict the state of
% Keplerian bodies. The state is [x;vx;y;vy;z;vz]

% Runge-Kutta 4th order integration method:
k1 = kepler(state);
k2 = kepler(state + dt*k1/2);
k3 = kepler(state + dt*k2/2);
k4 = kepler(state + dt*k3);

state = state + dt*(k1+2*k2+2*k3+k4)/6;

    function dstate=kepler(state)
        x =state(1,:);
        vx = state(2,:);
        y=state(3,:);
        vy = state(4,:);
        z=state(5,:);
        vz = state(6,:);

        mu = 398600.4405*1e9; % m^3 s^-2
        omega = 7.292115e-5; % rad/s

        r = norm([x y z]);
        g = mu/r^2;

        % Coordinates are in a non-intertial frame, account for Coriolis
        % and centripetal acceleration
        ax = -g*x/r + 2*omega*vy + omega^2*x;
        ay = -g*y/r - 2*omega*vx + omega^2*y;
        az = -g*z/r;
        dstate = [vx;ax;vy;ay;vz;az];
    end
end
```

`isDetectable` is used in the example to determine which tracks are detectable at a given time.

```
function detectInput = isDetectable(tracker,time,covcon)

if ~isLocked(tracker)
    detectInput = zeros(0,1,'uint32');
    return
end
tracks = tracker.predictTracksToTime('all',time);
if isempty(tracks)
    detectInput = zeros(0,1,'uint32');
else
    alltrackid = [tracks.TrackID];
```

```
    isDetectable = zeros(numel(tracks),numel(covcon),'logical');
    for i = 1:numel(tracks)
        track = tracks(i);
        pos_scene = track.State([1 3 5]);
        for j=1:numel(covcon)
            config = covcon(j);
            % rotate position to sensor frame:
            d_scene = pos_scene(:) - config.Position(:);
            scene2sens = rotmat(config.Orientation,'frame');
            d_sens = scene2sens*d_scene(:);
            [az,el] = cart2sph(d_sens(1),d_sens(2),d_sens(3));
            if abs(rad2deg(az)) <= config.FieldOfView(1)/2 && abs(rad2deg(el)) < config.FieldOfV
                isDetectable(i,j) = true;
            else
                isDetectable(i,j) = false;
            end
        end
    end

    detectInput = alltrackid(any(isDetectable,2))';
end
end
```

assembleRadarInput is used to construct the constellation poses in each radar body frame. This is the first step described in the diagram.

```
function targets = assembleRadarInputs(station, constellation)
% For each satellite in the constellation, derive its pose with respect to
% the radar frame.
% inputs:
%          - station        :  LLA vector of the radar ground station
%          - constellation  :  Array of structs containing the ECEF position
%                              and ECEF velocity of each satellite
% outputs:
%          - targets        :  Array of structs containing the pose of each
%                              satellite with respect to the radar, expressed in the local
%                              ground radar frame (NED)

% Template structure for the outputs which contains all the field required
% by the radar step method
targetTemplate =  struct( ...
                'PlatformID', 0, ...
                'ClassID', 0, ...
                'Position', zeros(1,3), ...
                'Velocity', zeros(1,3), ...
                'Acceleration', zeros(1,3), ...
                'Orientation', quaternion(1,0,0,0), ...
                'AngularVelocity', zeros(1,3),...
                'Dimensions', struct( ...
                            'Length', 0, ...
                            'Width', 0, ...
                            'Height', 0, ...
                            'OriginOffset', [0 0 0]), ...
                'Signatures', {{rcsSignature}});


% First fill in the current satellite ECEF pose
targetPoses = repmat(targetTemplate, 1, numel(constellation));
```

```
for i=1:numel(constellation)
    targetPoses(i).Position = constellation(i).Position;
    targetPoses(i).Velocity = constellation(i).Velocity;
    targetPoses(i).PlatformID = constellation(i).PlatformID;
    % Orientation and angular velocity are left null, assuming satellite to
    % be point targets with a uniform rcs
end

% Then derive the radar pose in ECEF based on the ground station location
Recef2station = dcmecef2ned(station(1), station(2));
radarPose.Orientation = quaternion(Recef2station,'rotmat','frame');
radarPose.Position = lla2ecef(station);
radarPose.Velocity = zeros(1,3);
radarPose.AngularVelocity = zeros(1,3);

% Finally, take the difference and rotate each vector to the ground station
% NED axes
targets = targetPoses;
for i=1: numel(targetPoses)
    thisTgt = targetPoses(i);
    pos = Recef2station*(thisTgt.Position(:) - radarPose.Position(:));
    vel = Recef2station*(thisTgt.Velocity(:) - radarPose.Velocity(:)) - cross(radarPose.AngularVe
    angVel = thisTgt.AngularVelocity(:) - radarPose.AngularVelocity(:);
    orient = radarPose.Orientation' * thisTgt.Orientation;

    % Store into target structure array
    targets(i).Position(:) = pos;
    targets(i).Velocity(:) = vel;
    targets(i).AngularVelocity(:) = angVel;
    targets(i).Orientation = orient;
end
end
```

addMeasurementParam implements step 2 in the radar chain process as described in the diagram.

```
function dets = addMeasurementParams(dets, numdets, station)
% Add radar station information to the measurement parameters
Recef2station = dcmecef2ned(station(1), station(2));
for i=1:numdets
    dets{i}.MeasurementParameters.OriginPosition = lla2ecef(station);
    dets{i}.MeasurementParameters.IsParentToChild = true; % parent = ecef, child = radar
    dets{i}.MeasurementParameters.Orientation = dets{i}.MeasurementParameters.Orientation' * Rece
end
end
```

distanceFcn is used with the assignment metrics to evaluate the tracking assignment.

```
function d = distanceFcn(track, truth)

estimate = track.State([1 3 5 2 4 6]);
true = [truth.Position(:) ; truth.Velocity(:)];
cov = track.StateCovariance([1 3 5 2 4 6], [1 3 5 2 4 6]);
d = (estimate - true)' / cov * (estimate - true);
end
```

Reference

[1] Sridharan, Ramaswamy, and Antonio F. Pensa, eds. *Perspectives in Space Surveillance*. MIT Press, 2017.

See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer` | `Satellite`

Functions
`play`

More About

- “Satellite Scenario Key Concepts” on page 2-65
- “Satellite Scenario Overview” on page 2-74

Get Started with Fixed-Wing Aircraft

This example shows how to create and use fixed-wing aircraft in MATLAB®.

For an example of setting realistic coefficients on an aircraft and calculating static stability, see "Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft”.

For an example of importing coefficients from Digital DATCOM analysis and linearizing to a state-space model, see “Analyze State-Space Model for Linear Control and Static Stability Analysis”.

For an example of creating custom states, see “Customize Fixed-Wing Aircraft with Additional Aircraft States”.

What Is a Fixed-Wing Aircraft?

Fixed-wing aircraft encompass all aircraft that generate lift from fixed airfoil surfaces extending off the main body. The standard configuration for fixed-wing aircraft is a large main wing near the center of gravity and horizontal and vertical stabilizers at the end of the body.

The large main wing of the fixed-wing aircraft generates lift, with the horizontal and vertical stabilizers providing reaction forces and moments for stability and control. However, unlike rotary-wing aircraft, the fixed-wing aircraft's wings are fixed in place. Therefore, to provide the airflow to generate the necessary lift to fly aircraft off the ground, the fixed-wing aircraft wings require forward movement. This forward movement is typically created from a thrust vector generated by a jet engine or propeller.

Fixed-Wing Aircraft Construction Workflow

The construction of a fixed-wing aircraft model requires these components:

- The configuration of the aircraft
- What aerodynamic surfaces exist on the aircraft?
- What control surfaces exist on the aircraft?
- What thrust vectors exist on the aircraft?
- The numerical model of the aircraft
- The current state of the aircraft

This example follows this workflow to illustrate how to construct a fixed-wing aircraft application for numerical analysis in MATLAB.

Fixed-Wing Aircraft Configuration

This example constructs a basic 3-control surface, standard-configuration aircraft.

For this example, only the control surfaces and body will be defined.

To start, define the control surfaces using the fixedWingSurface function.

```
surface = fixedWingSurface("mysurface")
```

```
surface =
  Surface with properties:
```

```
            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: Inf
        MinimumValue: -Inf
        Controllable: off
            Symmetry: "Symmetric"
    ControlVariables: [0x0 string]
          Properties: [1x1 Aero.Aircraft.Properties]
```

The surface has many properties that help define a fixed-wing aircraft surface, in particular the controllability, coefficients, maximum and minimum values. symmetry, and any nested surfaces a surface might have.

For this aircraft, the aileron is an asymmetric control surface with a maximum and minimum deflection of 20 and -20 degrees, respectively.

To construct this surface, name-value pairs can be specified to set each property to their desired level.

```
aileron = fixedWingSurface("aileron", "on", "Asymmetric", [-20, 20])
```

```
aileron =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Asymmetric"
    ControlVariables: ["aileron_1"    "aileron_2"]
          Properties: [1x1 Aero.Aircraft.Properties]
```

Using the same pattern as the aileron construct the elevators and rudders.

These two surfaces follow the same pattern as the aileron, but are defined as symmetric control surfaces.

```
elevator = fixedWingSurface("elevator", "on", "Symmetric", [-20, 20])
```

```
elevator =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: "elevator"
          Properties: [1x1 Aero.Aircraft.Properties]
```

```
rudder = fixedWingSurface("rudder", "on", "Symmetric", [-20, 20])
```

```
rudder =
  Surface with properties:
```

```
           Surfaces: [1x0 Aero.FixedWing.Surface]
       Coefficients: [1x1 Aero.FixedWing.Coefficient]
       MaximumValue: 20
       MinimumValue: -20
       Controllable: on
           Symmetry: "Symmetric"
   ControlVariables: "rudder"
         Properties: [1x1 Aero.Aircraft.Properties]
```

In addition to the control surfaces of the aircraft, also define the thrust vectors.

For this example, it is assumed that there is a single thrust vector along the body of the aircraft.

Define this thrust vector using the fixedWingThrust function.

```
propeller = fixedWingThrust("propeller","on","Symmetric",[0, 0.75])
```

```
propeller =
  Thrust with properties:

       Coefficients: [1x1 Aero.FixedWing.Coefficient]
       MaximumValue: 0.7500
       MinimumValue: 0
       Controllable: on
           Symmetry: "Symmetric"
   ControlVariables: "propeller"
         Properties: [1x1 Aero.Aircraft.Properties]
```

Thrust is nearly identical to surface with the exception that it is assumed to be controllable by default, whereas the surface is not controllable by default.

With these control surfaces and thrust vectors defined, create an aircraft using the fixedWingAircraft function.

This aircraft carries the full definition of the fixed-wing aircraft. This example uses it in all analysis methods.

The reference area, span, and length help dimensionalize non-dimensional coefficients used in the analysis methods.

For simplicity, this aircraft uses a reference area, span, and length of 3, 2, and 1, respectively.

```
aircraft = fixedWingAircraft("MyAircraft", 3,2,1)
```

```
aircraft =
  FixedWing with properties:

       ReferenceArea: 3
       ReferenceSpan: 2
     ReferenceLength: 1
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1x0 Aero.FixedWing.Surface]
             Thrusts: [1x0 Aero.FixedWing.Thrust]
         AspectRatio: 1.3333
```

```
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

Additionally, the control surfaces and thrust vectors can be applied to the aircraft

```
aircraft.Surfaces = [aileron, elevator, rudder]
```

```
aircraft =
  FixedWing with properties:

        ReferenceArea: 3
        ReferenceSpan: 2
      ReferenceLength: 1
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x3 Aero.FixedWing.Surface]
              Thrusts: [1x0 Aero.FixedWing.Thrust]
          AspectRatio: 1.3333
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

```
aircraft.Thrusts = propeller
```

```
aircraft =
  FixedWing with properties:

        ReferenceArea: 3
        ReferenceSpan: 2
      ReferenceLength: 1
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x3 Aero.FixedWing.Surface]
              Thrusts: [1x1 Aero.FixedWing.Thrust]
          AspectRatio: 1.3333
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

With the aircraft body construct and control surfaces set, the aircraft is now fully constructed.

However, the current construction of the aircraft does not have much numerical meaning as the numerical model has defaults of 0.

To remedy this, set numeric coefficients.

Fixed-Wing Aircraft Numerical Modeling

To perform numerical modeling using fixed-wing aircraft in MATLAB, define known coefficients that represent the nonlinear behavior of the aircraft at its various operating states. Aircraft coefficients are the fixed set of coefficients that define body forces and moments, excluding reaction forces due to control surfaces or thrust vectors.

Obtain these coefficients through a variety of methods, such as Digital DATCOM, Computational Fluid Dynamics (CFD) analysis, or using first-principles preliminary analysis calculations.

If using Digital DATCOM to calculate the numeric coefficients, directly convert the Digital DATCOM struct to a fixed-wing aircraft using datcomToFixedWing.

You can also manually import assign the coefficients to the aircraft.

Fixed-wing aircraft coefficients reside on the aircraft in several places. As can be seen above, the aircraft itself, and every surface and thrust on the aircraft, has a set of coefficients.

Control surface coefficients define the forces and moments due to control surface deflections.

Thrust coefficients define the forces and moments due to the various propulsion methods.

All these independent forces and moments summed together provide the full forces and moments definition of the aircraft, and in turn the nonlinear dynamics.

Define coefficients using the fixedWingCoefficient function.

```
coeff = fixedWingCoefficient
```

```
coeff =
  Coefficient with properties:

                    Table: [6x1 table]
                   Values: {6x1 cell}
           StateVariables: "Zero"
              StateOutput: [6x1 string]
           ReferenceFrame: "Wind"
    MultiplyStateVariables: on
           NonDimensional: on
               Properties: [1x1 Aero.Aircraft.Properties]
```

The fixedWingCoefficient function defines coefficient-specific properties, including the reference frame, specifying dimensional or non-dimensional coefficients, and specifying the state variable multiply behavior.

Setting coefficient values can be done through the setCoefficient function.

Retrieving coefficient values can be done through the getCoefficient function.

To view the coefficients of a component in a table, use the Table property on the returned coefficient.

```
CL_alpha = 0.2;
aircraft = setCoefficient(aircraft, "CL", "Alpha", CL_alpha)
```

```
aircraft =
  FixedWing with properties:

       ReferenceArea: 3
       ReferenceSpan: 2
     ReferenceLength: 1
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
    DegreesOfFreedom: "6DOF"
            Surfaces: [1x3 Aero.FixedWing.Surface]
```

```
              Thrusts: [1x1 Aero.FixedWing.Thrust]
          AspectRatio: 1.3333
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

```
getCoefficient(aircraft, "CL", "Alpha")
```

```
ans = 0.2000
```

```
aircraft.Coefficients.Table
```

```
ans=6×9 table
          Zero    U    Alpha    AlphaDot    Q    Beta    BetaDot    P    R
          ____    _    _____    ________    _    ____    _______    _    _

    CD     0      0       0         0        0     0          0       0    0
    CY     0      0       0         0        0     0          0       0    0
    CL     0      0     0.2         0        0     0          0       0    0
    Cl     0      0       0         0        0     0          0       0    0
    Cm     0      0       0         0        0     0          0       0    0
    Cn     0      0       0         0        0     0          0       0    0
```

You can also set coefficients on the nested surfaces and thrust vectors using their component name.

The component name is the same name that was set on the aircraft, surface, and thrust.

```
CL_0_elevator = 0.15;
aircraft = setCoefficient(aircraft, "CL", "Zero", CL_0_elevator, Component="elevator")
```

```
aircraft =
  FixedWing with properties:

        ReferenceArea: 3
        ReferenceSpan: 2
      ReferenceLength: 1
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x3 Aero.FixedWing.Surface]
              Thrusts: [1x1 Aero.FixedWing.Thrust]
          AspectRatio: 1.3333
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

```
getCoefficient(aircraft, "CL", "Zero",Component="elevator")
```

```
ans = 0.1500
```

```
aircraft.Surfaces(2).Coefficients.Table
```

```
ans=6×1 table
          Zero
          ____
```

```
    CD       0
    CY       0
    CL    0.15
    Cl       0
    Cm       0
    Cn       0
```

Set the specific coefficients using the setCoefficient function. Retrieve them using the getCoefficient function. These coefficients depend on:

- The input fixed-wing object
- The reference frame on the coefficient
- The state variables defined on the coefficient

The second input to setCoefficient and getCoefficient is the state output. To determine valid state outputs, refer to see the reference frame on the coefficient.

For example, if the reference frame is "Body", the valid state outputs are:

- "CX" - Coefficient of body X force
- "CY" - Coefficient of body Y force
- "CZ" - Coefficient of body Z force
- "Cl" - Rolling moment coefficient
- "Cm" - Pitching moment coefficient
- "Cn" - Yawing moment coefficient

```
coeff.ReferenceFrame = "Body";
coeff.Table
```

```
ans=6×1 table
          Zero
          ____

    CX     0
    CY     0
    CZ     0
    Cl     0
    Cm     0
    Cn     0
```

If the reference frame is "Wind", the valid state outputs are:

- "CD" - Coefficient of drag force
- "CY" - Coefficient of body Y force
- "CL" - Coefficient of lift force
- "Cl" - Rolling moment coefficient
- "Cm" - Pitching moment coefficient
- "Cn" - Yawing moment coefficient

```
coeff.ReferenceFrame = "Wind";
coeff.Table
```

```
ans=6×1 table
          Zero
          ____

    CD     0
    CY     0
    CL     0
    Cl     0
    Cm     0
    Cn     0
```

The third input argument to setCoefficient and getCoefficient is the state variables that determine the states the coefficients are defined with.

By default, the coefficient assumes no state relationship. These coefficients are defined with the "Zero" state variable, which means the coefficient has no states to multiply against.

In the case of the fixed-wing aircraft, there are a set of additional default states that are common to many aircraft definitions, namely, U, Alpha, AlphaDot, Beta, BetaDot, P, Q, and R.

Use any combination of these state outputs and state variables with setCoefficient and getCoefficient.

```
coeff.StateVariables = ["Alpha", "Beta"];
coeff = setCoefficient(coeff, "CL", "Beta", 5);
coeff.Table
```

```
ans=6×2 table
          Alpha    Beta
          _____    ____

    CD      0       0
    CY      0       0
    CL      0       5
    Cl      0       0
    Cm      0       0
    Cn      0       0
```

With the coefficients set on the aircraft, define the aircraft current state.

Fixed-Wing Aircraft States

The current state of an aircraft defines the properties that are independent of the fixed configuration.

These properties include the mass, inertia, airspeed, altitude, deflection angles, and others.

By separating the current state from the configuration, the aircraft coefficient data can remain fixed while individual states change over time.

Define fixed-wing aircraft states using the fixedWingState function.

```
state = fixedWingState(aircraft)
```

```
state =
  State with properties:

                    Alpha: 0
```

```
                  Beta: 0
              AlphaDot: 0
               BetaDot: 0
                  Mass: 0
               Inertia: [3x3 table]
        CenterOfGravity: [0 0 0]
      CenterOfPressure: [0 0 0]
           AltitudeMSL: 0
          GroundHeight: 0
                    XN: 0
                    XE: 0
                    XD: 0
                     U: 50
                     V: 0
                     W: 0
                   Phi: 0
                 Theta: 0
                   Psi: 0
                     P: 0
                     Q: 0
                     R: 0
                Weight: 0
            AltitudeAGL: 0
               Airspeed: 50
            GroundSpeed: 50
             MachNumber: 0.1469
           BodyVelocity: [50 0 0]
         GroundVelocity: [50 0 0]
                    Ur: 50
                    Vr: 0
                    Wr: 0
        FlightPathAngle: 0
            CourseAngle: 0
   InertialToBodyMatrix: [3x3 double]
   BodyToInertialMatrix: [3x3 double]
       BodyToWindMatrix: [3x3 double]
       WindToBodyMatrix: [3x3 double]
  BodyToStabilityMatrix: [3x3 double]
  StabilityToBodyMatrix: [3x3 double]
        DynamicPressure: 1.5312e+03
            Environment: [1x1 Aero.Aircraft.Environment]
          ControlStates: [1x6 Aero.Aircraft.ControlState]
       OutOfRangeAction: "Limit"
       DiagnosticAction: "Warning"
             Properties: [1x1 Aero.Aircraft.Properties]
             UnitSystem: "Metric"
      TemperatureSystem: "Kelvin"
            AngleSystem: "Radians"
```

The names of the properties on this state are the same as the state variable string names in the coefficients.

In the aircraft coefficient table above, each coefficient in the "Alpha" column is multiplied by the "Alpha" property in the state if "MultiplyStateVariables" is on.

This action applies for every state variable in every coefficient on the aircraft.

Some states are dependent states and depend on other properties within the state itself.

If a state depends on properties within the environment, you must define the current flying environment as well.

Define the current flying environment using the aircraftEnvironment function or by assigning to the environment directly on the state.

```
environment = aircraftEnvironment(aircraft, "ISA", 0)
```

```
environment =
  Environment with properties:

    WindVelocity: [0 0 0]
         Density: 1.2250
     Temperature: 288.1500
        Pressure: 101325
    SpeedOfSound: 340.2941
         Gravity: 9.8100
      Properties: [1x1 Aero.Aircraft.Properties]
```

```
state.Environment = environment
```

```
state =
  State with properties:

                    Alpha: 0
                     Beta: 0
                 AlphaDot: 0
                  BetaDot: 0
                     Mass: 0
                  Inertia: [3x3 table]
          CenterOfGravity: [0 0 0]
         CenterOfPressure: [0 0 0]
              AltitudeMSL: 0
             GroundHeight: 0
                       XN: 0
                       XE: 0
                       XD: 0
                        U: 50
                        V: 0
                        W: 0
                      Phi: 0
                    Theta: 0
                      Psi: 0
                        P: 0
                        Q: 0
                        R: 0
                   Weight: 0
               AltitudeAGL: 0
                  Airspeed: 50
               GroundSpeed: 50
                MachNumber: 0.1469
              BodyVelocity: [50 0 0]
            GroundVelocity: [50 0 0]
                        Ur: 50
                        Vr: 0
```

```
                    Wr: 0
          FlightPathAngle: 0
              CourseAngle: 0
     InertialToBodyMatrix: [3x3 double]
     BodyToInertialMatrix: [3x3 double]
         BodyToWindMatrix: [3x3 double]
         WindToBodyMatrix: [3x3 double]
    BodyToStabilityMatrix: [3x3 double]
    StabilityToBodyMatrix: [3x3 double]
          DynamicPressure: 1.5312e+03
              Environment: [1x1 Aero.Aircraft.Environment]
            ControlStates: [1x6 Aero.Aircraft.ControlState]
         OutOfRangeAction: "Limit"
         DiagnosticAction: "Warning"
               Properties: [1x1 Aero.Aircraft.Properties]
               UnitSystem: "Metric"
        TemperatureSystem: "Kelvin"
               AngleSystem: "Radians"
```

The environment is assumed to be the same unit system as the state. It is important to keep these unit systems aligned and to align the unit systems between each state and aircraft.

Creating an array of many states can be helpful for designing the sweep of parameters over which to perform calculations on the aircraft.

In this example, 11 states are created by varying mass, but holding airspeed constant.

```
mass = num2cell(1000:50:1500)
```

```
mass=1×11 cell array
    {[1000]}    {[1050]}    {[1100]}    {[1150]}    {[1200]}    {[1250]}    {[1300]}    {[1350]}
```

```
state = fixedWingState(aircraft, U=100)
```

```
state =
  State with properties:

                   Alpha: 0
                    Beta: 0
                AlphaDot: 0
                 BetaDot: 0
                    Mass: 0
                 Inertia: [3x3 table]
          CenterOfGravity: [0 0 0]
         CenterOfPressure: [0 0 0]
              AltitudeMSL: 0
             GroundHeight: 0
                       XN: 0
                       XE: 0
                       XD: 0
                        U: 100
                        V: 0
                        W: 0
                      Phi: 0
                    Theta: 0
                      Psi: 0
```

```
                        P: 0
                        Q: 0
                        R: 0
                   Weight: 0
               AltitudeAGL: 0
                  Airspeed: 100
               GroundSpeed: 100
                MachNumber: 0.2939
              BodyVelocity: [100 0 0]
            GroundVelocity: [100 0 0]
                        Ur: 100
                        Vr: 0
                        Wr: 0
          FlightPathAngle: 0
              CourseAngle: 0
     InertialToBodyMatrix: [3x3 double]
     BodyToInertialMatrix: [3x3 double]
         BodyToWindMatrix: [3x3 double]
         WindToBodyMatrix: [3x3 double]
    BodyToStabilityMatrix: [3x3 double]
    StabilityToBodyMatrix: [3x3 double]
          DynamicPressure: 6125
              Environment: [1x1 Aero.Aircraft.Environment]
            ControlStates: [1x6 Aero.Aircraft.ControlState]
         OutOfRangeAction: "Limit"
         DiagnosticAction: "Warning"
               Properties: [1x1 Aero.Aircraft.Properties]
               UnitSystem: "Metric"
        TemperatureSystem: "Kelvin"
              AngleSystem: "Radians"
```

```
states = repmat(state, size(mass))
```

```
states=1×11 object
  1x11 State array with properties:

    Alpha
    Beta
    AlphaDot
    BetaDot
    Mass
    Inertia
    CenterOfGravity
    CenterOfPressure
    AltitudeMSL
    GroundHeight
    XN
    XE
    XD
    U
    V
    W
    Phi
    Theta
    Psi
    P
    Q
```

```
    R
    Weight
    AltitudeAGL
    Airspeed
    GroundSpeed
    MachNumber
    BodyVelocity
    GroundVelocity
    Ur
    Vr
    Wr
    FlightPathAngle
    CourseAngle
    InertialToBodyMatrix
    BodyToInertialMatrix
    BodyToWindMatrix
    WindToBodyMatrix
    BodyToStabilityMatrix
    StabilityToBodyMatrix
    DynamicPressure
    Environment
    ControlStates
    OutOfRangeAction
    DiagnosticAction
    Properties
    UnitSystem
    TemperatureSystem
    AngleSystem


[states.Mass] = mass{:}

states=1×11 object
  1x11 State array with properties:

    Alpha
    Beta
    AlphaDot
    BetaDot
    Mass
    Inertia
    CenterOfGravity
    CenterOfPressure
    AltitudeMSL
    GroundHeight
    XN
    XE
    XD
    U
    V
    W
    Phi
    Theta
    Psi
    P
    Q
    R
    Weight
```

```
    AltitudeAGL
    Airspeed
    GroundSpeed
    MachNumber
    BodyVelocity
    GroundVelocity
    Ur
    Vr
    Wr
    FlightPathAngle
    CourseAngle
    InertialToBodyMatrix
    BodyToInertialMatrix
    BodyToWindMatrix
    WindToBodyMatrix
    BodyToStabilityMatrix
    StabilityToBodyMatrix
    DynamicPressure
    Environment
    ControlStates
    OutOfRangeAction
    DiagnosticAction
    Properties
    UnitSystem
    TemperatureSystem
    AngleSystem
```

The second environment input can also help create a state array that iterates over many altitudes using a standard atmosphere model.

```
statesH = fixedWingState(aircraft, aircraftEnvironment(aircraft, "ISA", [0, 1000, 2000]))
```

```
statesH=1×3 object
  1x3 State array with properties:

    Alpha
    Beta
    AlphaDot
    BetaDot
    Mass
    Inertia
    CenterOfGravity
    CenterOfPressure
    AltitudeMSL
    GroundHeight
    XN
    XE
    XD
    U
    V
    W
    Phi
    Theta
    Psi
    P
    Q
    R
```

```
    Weight
    AltitudeAGL
    Airspeed
    GroundSpeed
    MachNumber
    BodyVelocity
    GroundVelocity
    Ur
    Vr
    Wr
    FlightPathAngle
    CourseAngle
    InertialToBodyMatrix
    BodyToInertialMatrix
    BodyToWindMatrix
    WindToBodyMatrix
    BodyToStabilityMatrix
    StabilityToBodyMatrix
    DynamicPressure
    Environment
    ControlStates
    OutOfRangeAction
    DiagnosticAction
    Properties
    UnitSystem
    TemperatureSystem
    AngleSystem
```

Fixed-Wing Analysis Methods

With the construction of the aircraft and its states, you can now perform fixed-wing specific calculations.

These calculations can include:

1. Forces and moments
2. Nonlinear dynamics
3. Static stability
4. Linearization to a state-space model

```
F = zeros(numel(states), 3);
M = zeros(numel(states), 3);
dydt = zeros(numel(states), 12);
for i = 1:numel(states)
    [F(i,:), M(i,:)] = forcesAndMoments(aircraft, states(i));
    dydt(i,:) = nonlinearDynamics(aircraft, states(i));
end
```

The aircraft can also be used as a generic container to hold the aircraft definition as it is passed around to other parts of the program, such as in a Simulink lookup table or custom MATLAB analysis functions.

In summary, the fixed-wing aircraft functions provide a common definition to creating and manipulating fixed-wing aircraft within MATLAB.

See Also

`fixedWingAircraft` | `fixedWingCoefficient` | `fixedWingState` | `fixedWingSurface` | `aircraftEnvironment` | `fixedWingThrust`

Related Examples

- “Analyze State-Space Model for Linear Control and Static Stability Analysis” on page 5-106
- “Customize Fixed-Wing Aircraft with Additional Aircraft States” on page 5-113
- “Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft” on page 5-121

Customize Fixed-Wing Aircraft with the Object Interface

This example shows how to customize fixed-wing aircraft in MATLAB® using objects.

For an example of how to get started using fixed-wing aircraft in MATLAB, see “Get Started with Fixed-Wing Aircraft”.

For an example of setting realistic coefficients on an aircraft and calculating static stability, see “Determine Nonlinear Dynamics and Static Stability of Fixed-Wing Aircraft”.

For an example of importing coefficients from Digital DATCOM analysis and linearizing to a state-space model, see “Perform Controls and Static Stability Analysis with Linearized Fixed-Wing Aircraft”.

Fixed-Wing Object Interface

Each fixed-wing aircraft function returns an object of its defining type.

Functions provide convenience to constructing each object. However, their predefined input structure might be inconvenient to some workflows. For example, consider using objects if you want more control over the specific inputs of each component, or if you want to avoid repmat calls to create arrays of each component. Objects let you directly construct each component.

The table mapping each function to object can be seen below.

```
Component = ["Properties"; "Environment"; "Aircraft"; "States"; "Coefficients"; "Surfaces"; "Thru
Formal = ["Aero.Aircraft.Properties"; "Aero.Aircraft.Environment"; "Aero.FixedWing"; "Aero.FixedW
Informal = ["aircraftProperties"; "aircraftEnvironment"; "fixedWingAircraft"; "fixedWingState"; '
objMap = table(Formal, Informal, 'RowNames', Component, 'VariableNames',["Formal Interface", "In
```

```
objMap=7×2 table
                          Formal Interface            Informal Interface
                    ____________________________    ______________________

    Properties      "Aero.Aircraft.Properties"      "aircraftProperties"
    Environment     "Aero.Aircraft.Environment"     "aircraftEnvironment"
    Aircraft        "Aero.FixedWing"                "fixedWingAircraft"
    States          "Aero.FixedWing.State"          "fixedWingState"
    Coefficients    "Aero.FixedWing.Coefficient"    "fixedWingCoefficient"
    Surfaces        "Aero.FixedWing.Surface"        "fixedWingSurface"
    Thrust          "Aero.FixedWing.Thrust"         "fixedWingThrust"
```

The constructor for each fixed-wing aircraft component is structured the same way:

- The first argument is either a vector or repeating set of integers that specifies the size of the returned object array, like the "zeros" function, with the default being size 1.
- Every argument after the first argument is a name-value pair specifying the object property and value for setting non-default values.
- Each non-default value is set for every object in the returned object array.

For example, creating a 3-element fixed-wing state vector where each state has a mass of 50 kg would have the following syntax:

```
states = Aero.FixedWing.State(1,3, Mass=50)
```

```
states=1×3 object
  1x3 State array with properties:

    Alpha
    Beta
    AlphaDot
    BetaDot
    Mass
    Inertia
    CenterOfGravity
    CenterOfPressure
    AltitudeMSL
    GroundHeight
    XN
    XE
    XD
    U
    V
    W
    Phi
    Theta
    Psi
    P
    Q
    R
    Weight
    AltitudeAGL
    Airspeed
    GroundSpeed
    MachNumber
    BodyVelocity
    GroundVelocity
    Ur
    Vr
    Wr
    FlightPathAngle
    CourseAngle
    InertialToBodyMatrix
    BodyToInertialMatrix
    BodyToWindMatrix
    WindToBodyMatrix
    BodyToStabilityMatrix
    StabilityToBodyMatrix
    DynamicPressure
    Environment
    ControlStates
    OutOfRangeAction
    DiagnosticAction
    Properties
    UnitSystem
    TemperatureSystem
    AngleSystem
```

```
states.Mass
```

```
ans = 50
```

```
ans = 50
```

```
ans = 50
```

As can be seen from the returned Mass values, each state has a mass of 50 in the vector.

Following this format, this example constructs the same aircraft from the "Get Started with Fixed-Wing Aircraft" example, replacing the function interface with the associated object interface.

Fixed-Wing Aircraft Configuration

Create the 3 control surfaces using Aero.FixedWing.Surface.

By default, the surface objects are defined to be symmetric and not controllable, so these two properties must be set for the aileron along with the maximum and minimum values.

```
aileron = Aero.FixedWing.Surface(...
              Controllable="on", ...
              Symmetry="Asymmetric", ...
              MinimumValue=-20, ...
              MaximumValue=20)
```

```
aileron =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Asymmetric"
    ControlVariables: ["_1"    "_2"]
          Properties: [1x1 Aero.Aircraft.Properties]
```

Additionally, the set the name on the properties of the aileron surface object. This is helpful for setting coefficients later.

```
aileron.Properties.Name = "aileron"
```

```
aileron =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Asymmetric"
    ControlVariables: ["aileron_1"    "aileron_2"]
          Properties: [1x1 Aero.Aircraft.Properties]
```

For the elevator and rudder, the symmetry is already the desired value, so the "Symmetry" name-value argument can be excluded.

```
elevator = Aero.FixedWing.Surface(...
              Controllable="on", ...
              MinimumValue=-20, ...
              MaximumValue=20)
```

```
elevator =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: ""
          Properties: [1x1 Aero.Aircraft.Properties]
```

```
rudder = Aero.FixedWing.Surface(...
               Controllable="on", ...
               MinimumValue=-20, ...
               MaximumValue=20)
```

```
rudder =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: ""
          Properties: [1x1 Aero.Aircraft.Properties]
```

```
elevator.Properties.Name = "Elevator"
```

```
elevator =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: "Elevator"
          Properties: [1x1 Aero.Aircraft.Properties]
```

```
rudder.Properties.Name = "Rudder"
```

```
rudder =
  Surface with properties:

            Surfaces: [1x0 Aero.FixedWing.Surface]
        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 20
        MinimumValue: -20
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: "Rudder"
          Properties: [1x1 Aero.Aircraft.Properties]
```

With the control surfaces defined, define the thrust object using the Aero.FixedWing.Thrust object.

By default, the minimum and maximum values for the thrust object are 0 and 1, which represent the throttle lever position.

In this aircraft, they are limited to 0 and 0.75.

```
propeller = Aero.FixedWing.Thrust(MaximumValue=0.75)
```

```
propeller =
  Thrust with properties:

        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 0.7500
        MinimumValue: 0
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: ""
          Properties: [1x1 Aero.Aircraft.Properties]
```

The name of the thrust vector can also be set at this time.

```
propeller.Properties.Name = "propeller"
```

```
propeller =
  Thrust with properties:

        Coefficients: [1x1 Aero.FixedWing.Coefficient]
        MaximumValue: 0.7500
        MinimumValue: 0
        Controllable: on
            Symmetry: "Symmetric"
    ControlVariables: "propeller"
          Properties: [1x1 Aero.Aircraft.Properties]
```

With these control surfaces and thrust vectors defined, they can now be set on the body of the aircraft.

The aircraft body is defined through the Aero.FixedWing object.

For simplicity, this aircraft will have a reference area, span, and length of 3, 2, and 1, respectively.

```
aircraft = Aero.FixedWing(...
           ReferenceArea=3, ...
           ReferenceSpan=2, ...
           ReferenceLength=1)
```

```
aircraft =
  FixedWing with properties:

        ReferenceArea: 3
        ReferenceSpan: 2
      ReferenceLength: 1
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x0 Aero.FixedWing.Surface]
```

```
             Thrusts: [1x0 Aero.FixedWing.Thrust]
          AspectRatio: 1.3333
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

```
aircraft.Surfaces = [aileron, elevator, rudder]
```

```
aircraft =
  FixedWing with properties:

        ReferenceArea: 3
        ReferenceSpan: 2
      ReferenceLength: 1
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x3 Aero.FixedWing.Surface]
              Thrusts: [1x0 Aero.FixedWing.Thrust]
          AspectRatio: 1.3333
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

```
aircraft.Thrusts = propeller
```

```
aircraft =
  FixedWing with properties:

        ReferenceArea: 3
        ReferenceSpan: 2
      ReferenceLength: 1
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x3 Aero.FixedWing.Surface]
              Thrusts: [1x1 Aero.FixedWing.Thrust]
          AspectRatio: 1.3333
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

Set the aircraft coefficients using the setCoefficient and getCoefficient methods, creating a coefficient using the Aero.FixedWing.Coefficient object, or indexing directly to the coefficient values.

```
coeff = Aero.FixedWing.Coefficient
```

```
coeff =
  Coefficient with properties:

                    Table: [6x1 table]
                   Values: {6x1 cell}
           StateVariables: "Zero"
              StateOutput: [6x1 string]
           ReferenceFrame: "Wind"
```

```
    MultiplyStateVariables: on
            NonDimensional: on
                Properties: [1x1 Aero.Aircraft.Properties]
```

```
aircraft.Coefficients.Values{3,3} = 0.2
```

```
aircraft =
  FixedWing with properties:

        ReferenceArea: 3
        ReferenceSpan: 2
      ReferenceLength: 1
         Coefficients: [1x1 Aero.FixedWing.Coefficient]
     DegreesOfFreedom: "6DOF"
             Surfaces: [1x3 Aero.FixedWing.Surface]
              Thrusts: [1x1 Aero.FixedWing.Thrust]
          AspectRatio: 1.3333
           Properties: [1x1 Aero.Aircraft.Properties]
           UnitSystem: "Metric"
    TemperatureSystem: "Kelvin"
          AngleSystem: "Radians"
```

```
aircraft.Coefficients.Table
```

```
ans=6×9 table
          Zero    U    Alpha    AlphaDot    Q    Beta    BetaDot    P    R
          ____    _    _____    ________    _    ____    _______    _    _

    CD     0      0       0         0        0     0         0       0    0
    CY     0      0       0         0        0     0         0       0    0
    CL     0      0     0.2         0        0     0         0       0    0
    Cl     0      0       0         0        0     0         0       0    0
    Cm     0      0       0         0        0     0         0       0    0
    Cn     0      0       0         0        0     0         0       0    0
```

Fixed-Wing Aircraft States

Define fixed-wing aircraft states using the Aero.FixedWing.State object.

To directly create an array of states that all have the same properties, use the state constructor instead of using the repmat function.

In this example, 11 states are created by varying mass, but with constant airspeed.

```
mass = num2cell(1000:50:1500)
```

```
mass=1×11 cell array
    {[1000]}    {[1050]}    {[1100]}    {[1150]}    {[1200]}    {[1250]}    {[1300]}    {[1350]}
```

```
states = Aero.FixedWing.State(size(mass), U=100);
[states.Mass] = mass{:}
```

```
states=1×11 object
  1x11 State array with properties:
```

```
Alpha
Beta
AlphaDot
BetaDot
Mass
Inertia
CenterOfGravity
CenterOfPressure
AltitudeMSL
GroundHeight
XN
XE
XD
U
V
W
Phi
Theta
Psi
P
Q
R
Weight
AltitudeAGL
Airspeed
GroundSpeed
MachNumber
BodyVelocity
GroundVelocity
Ur
Vr
Wr
FlightPathAngle
CourseAngle
InertialToBodyMatrix
BodyToInertialMatrix
BodyToWindMatrix
WindToBodyMatrix
BodyToStabilityMatrix
StabilityToBodyMatrix
DynamicPressure
Environment
ControlStates
OutOfRangeAction
DiagnosticAction
Properties
UnitSystem
TemperatureSystem
AngleSystem
```

However, unlike the fixedWingState function, the control states are not automatically set up on the states from the aircraft.

To set up the control states, use the setupControlStates function.

```
states = setupControlStates(states, aircraft)
```

```
states=1×11 object
  1x11 State array with properties:

    Alpha
    Beta
    AlphaDot
    BetaDot
    Mass
    Inertia
    CenterOfGravity
    CenterOfPressure
    AltitudeMSL
    GroundHeight
    XN
    XE
    XD
    U
    V
    W
    Phi
    Theta
    Psi
    P
    Q
    R
    Weight
    AltitudeAGL
    Airspeed
    GroundSpeed
    MachNumber
    BodyVelocity
    GroundVelocity
    Ur
    Vr
    Wr
    FlightPathAngle
    CourseAngle
    InertialToBodyMatrix
    BodyToInertialMatrix
    BodyToWindMatrix
    WindToBodyMatrix
    BodyToStabilityMatrix
    StabilityToBodyMatrix
    DynamicPressure
    Environment
    ControlStates
    OutOfRangeAction
    DiagnosticAction
    Properties
    UnitSystem
    TemperatureSystem
    AngleSystem
```

See Also

Aero.FixedWing | Aero.FixedWing.Coefficient | Aero.FixedWing.State | Aero.FixedWing.Surface | Aero.FixedWing.Thrust

Related Examples

- “Customize Fixed-Wing Aircraft with Additional Aircraft States” on page 5-113

Multi-Hop Path Selection Through Large Satellite Constellation

This example shows how to determine the path through a large constellation consisting of 1,000 low-Earth orbit (LEO) satellites to gain access between two ground stations. Following this, it demonstrates how to calculate the intervals during the next 3 hour period when this path can be used.

Create Satellite Scenario

Assume that the path through the large constellation to establish access between two ground stations must be determined as of 10 December 2021, 6:27:57 PM UTC. We must then determine the times over the next 3 hours when this path can be used. Accordingly, create a satellite scenario with the appropriate `StartTime` and `StopTime`. Set `SampleTime` to 60 seconds. Since the path must be determined only for the first time step, set `AutoSimulate` of the scenario to false to prevent it from automatically advancing through the time steps until `StopTime`. When `AutoSimulate` is false, `SimulationStatus` becomes available.

```
startTime = datetime(2021,12,10,18,27,57); % 10 December 2021, 6:27:57 PM UTC
stopTime = startTime + hours(3);           % 10 December 2021, 9:27:57 PM UTC
sampleTime = 60;                           % Seconds
sc = satelliteScenario(startTime,stopTime,sampleTime,"AutoSimulate",false)
```

```
sc =
  satelliteScenario with properties:

           StartTime: 10-Dec-2021 18:27:57
            StopTime: 10-Dec-2021 21:27:57
          SampleTime: 60
      SimulationTime: 10-Dec-2021 18:27:57
    SimulationStatus: NotStarted
        AutoSimulate: 0
          Satellites: [1×0 matlabshared.satellitescenario.Satellite]
      GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
             Viewers: [0×0 matlabshared.satellitescenario.Viewer]
            AutoShow: 1
```

Add Large Constellation of Satellites

Add the large satellite constellation from the Two-Line-Element (TLE) file `largeConstellation.tle`. The constellation consists of 1,000 LEO satellites.

```
sat = satellite(sc,"largeConstellation.tle");
numSatellites = numel(sat)
```

```
numSatellites = 1000
```

Add Ground Stations

Add the ground stations. A multi-hop path must be established through the satellite constellation for access between the ground stations.

```
gsSource = groundStation(sc,42.3001,-71.3504, ... % Latitude and longitude in degrees
    "Name","Source Ground Station");
gsTarget = groundStation(sc,17.4351,78.3824, ...  % Latitude and longitude in degrees
    "Name","Target Ground Station");
```

Determine Elevation Angles of Satellites with Respect to Ground Stations

Determine the elevation angle of each satellite with respect to source and target ground stations corresponding to `StartTime`. Accordingly, use `advance` to simulate the scenario for the first time step, namely, `StartTime`. Following this, use `aer` to retrieve the elevation angle of each satellite with respect to the ground stations. Assume that for the initial routing, the elevation angle of the first satellite in the path with respect to "Source Ground Station" and the last satellite in the path with respect to "Target Ground Station" must be at least 30 degrees. Accordingly, determine the elevation angles that are greater than or equal to this value.

```
% Calculate the scenario state corresponding to StartTime.
advance(sc);

% Retrieve the elevation angle of each satellite with respect to the ground
% stations.
[~,elSourceToSat] = aer(gsSource,sat);
[~,elTargetToSat] = aer(gsTarget,sat);

% Determine the elevation angles that are greater than or equal to 30
% degrees.
elSourceToSatGreaterThanOrEqual30 = (elSourceToSat >= 30)';
elTargetToSatGreaterThanOrEqual30 = (elTargetToSat >= 30)';
```

Determine Best Satellite for Initial Access to Constellation

The best satellite to be used for the initial access to the large constellation is assumed to be the one that satisfies the following simultaneously:

- Has the closest range to "Target Ground Station".
- Has an elevation angle of at least 30 degrees with respect to "Source Ground Station".

```
% Find the indices of the elements of elSourceToSatGreaterThanOrEqual30
% whose value is true.
trueID = find(elSourceToSatGreaterThanOrEqual30 == true);

% These indices are essentially the indices of satellites in sat whose
% elevation angle with respect to "Source Ground Station" is at least 30
% degrees. Determine the range of these satellites to "Target Ground
% Station".
[~,~,r] = aer(sat(trueID), gsTarget);

% Determine the index of the element in r bearing the minimum value.
[~,minRangeID] = min(r);

% Determine the element in trueID at the index minRangeID.
id = trueID(minRangeID);

% This is the index of the best satellite for initial access to the
% constellation. This will be the first hop in the path. Initialize a
% variable 'node' that stores the first two nodes of the routing - namely,
% "Source Ground Station" and the best satellite for initial constellation
% access.
nodes = {gsSource sat(id)};
```

Determine Remaining Nodes in Path to Target Ground Station

The remaining nodes in the path are determined using a similar logic as what was used for determining the first satellite. If the elevation angle of the satellite in the current node is already at

least 30 degrees with respect to "Target Ground Station", the next node is "Target Ground Station", thereby completing the path. Otherwise, the next node in the constellation is chosen using a logic that is similar to what was used for determining the first satellite in the path. The next node is the one that simultaneously satisfies the following:

- Has the closest range to "Target Ground Station".
- Elevation angle with respect to the satellite in the current node is greater than or equal to -15 degrees.

The elevation value of -15 degrees is chosen because the horizon with respect to each satellite in the constellation is about -21.9813 degrees. This value can be derived by assuming a spherical Earth geometry and the fact that these satellites are in near-circular orbits at an altitude of roughly 500 kilometers. Note that the spherical Earth assumption is used only for computing the elevation angle of the horizon below. The satellite scenario simulation itself assumes a WGS84 ellipsoid model for the Earth.

```
earthRadius = 6378137;                                                    % meters
altitude = 500000;                                                        % meters
horizonElevationAngle = asind(earthRadius/(earthRadius + altitude)) - 90 % degrees
```

```
horizonElevationAngle = -21.9813
```

Any satellite whose elevation angle with respect to another satellite is greater than -21.9813 degrees is guaranteed to be visible to the latter. However, choosing -15 degrees provides an adequate margin.

The subsequent nodes are continually added to the path until reaching a satellite whose elevation angle with respect to "Target Ground Station" is at least 30 degrees. After this, the final node is "Target Ground Station" itself, and the routing is complete.

```
% Minimum elevation angle of satellite nodes with respect to the prior
% node.
minSatElevation = -15; % degrees

% Flag to specify if the complete multi-hop path has been found.
pathFound = false;

% Determine nodes of the path in a loop. Exit the loop once the complete
% multi-hop path has been found.
while ~pathFound
    % Index of the satellite in sat corresponding to current node is
    % updated to the value calculated as index for the next node in the
    % prior loop iteration. Essentially, the satellite in the next node in
    % prior iteration becomes the satellite in the current node in this
    % iteration.
    idCurrent = id;

    % This is the index of the element in elTargetToSatGreaterThanOrEqual30
    % tells if the elevation angle of this satellite is at least 30 degrees
    % with respect to "Target Ground Station". If this element is true, the
    % routing is complete, and the next node is the target ground station.
    if elTargetToSatGreaterThanOrEqual30(idCurrent)
        nodes = {nodes{:} gsTarget}; %#ok<CCAT>
        pathFound = true;
        continue
    end

    % If the element is false, the path is not complete yet. The next node
```

```
    % in the path must be determined from the constellation. Determine
    % which satellites have elevation angle that is greater than or equal
    % to -15 degrees with respect to the current node. To do this, first
    % determine the elevation angle of each satellite with respect to the
    % current node.
    [~,els] = aer(sat(idCurrent),sat);

    % Overwrite the elevation angle of the satellite with respect to itself
    % to be -90 degrees to ensure it does not get re-selected as the next
    % node.
    els(idCurrent) = -90;

    % Determine the elevation angles that are greater than or equal to -15
    % degrees.
    s = els >= minSatElevation;

    % Find the indices of the elements in s whose value is true.
    trueID = find(s == true);

    % These indices are essentially the indices of satellites in sat whose
    % elevation angle with respect to the current node is greater than or
    % equal to -15 degrees. Determine the range of these satellites to
    % "Target Ground Station".
    [~,~,r] = aer(sat(trueID), gsTarget);

    % Determine the index of the element in r bearing the minimum value.
    [~,minRangeID] = min(r);

    % Determine the element in trueID at the index minRangeID.
    id = trueID(minRangeID);

    % This is the index of the best satellite among those in sat to be used
    % for the next node in the path. Append this satellite to the 'nodes'
    % variable.
    nodes = {nodes{:} sat(id)}; %#ok<CCAT>
end
```

Determine Intervals When Calculated Path Can Be Used

We must now determine the intervals over the next 3 hours during which calculated path can be used. To accomplish this, manually stepping through each time step of the scenario is not necessary. Instead, we can make the scenario auto-simulate from `StartTime` to `StopTime`. Therefore, set `AutoSimulate` of the scenario to `true`.

```
sc.AutoSimulate = true;
```

Add an access analysis with the calculated nodes in the path. Set `LineColor` of the access visualization to red.

```
ac = access(nodes{:});
ac.LineColor = "red";
```

Determine the access intervals using the `accessIntervals` function. Since `AutoSimulate` has been set to `true`, the scenario will automatically simulate from `StartTime` to `StopTime` at the specified `SampleTime` before calculating the access intervals.These intervals are the times when the calculated multi-hop path can be used.

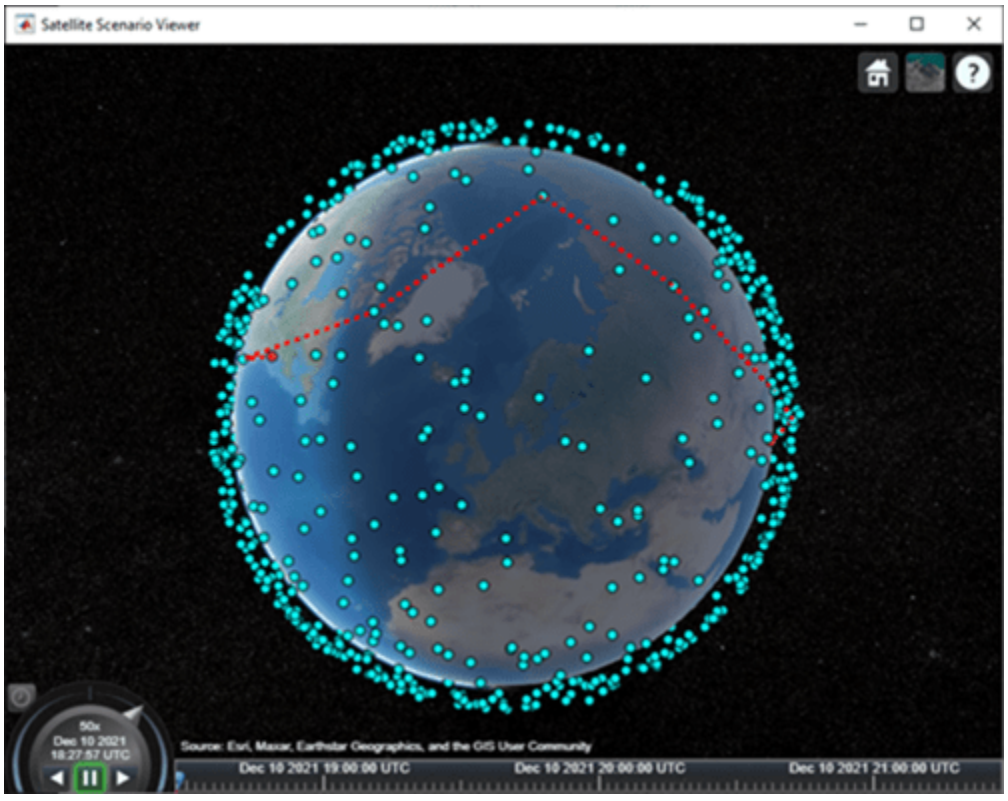
```
intvls = accessIntervals(ac)
```

```
intvls=2×8 table
            Source                    Target             IntervalNumber         StartTime
    ______________________    ______________________    ______________    ____________________

    "Source Ground Station"    "Target Ground Station"          1           10-Dec-2021 18:27:57
    "Source Ground Station"    "Target Ground Station"          2           10-Dec-2021 20:01:57
```

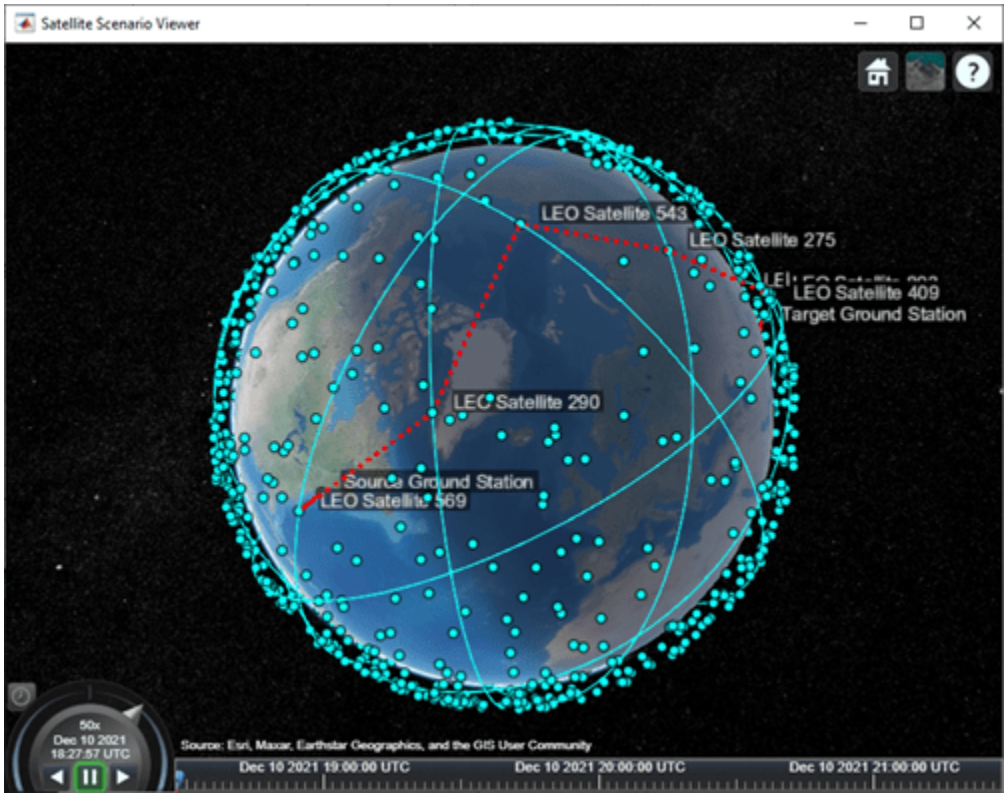
Visualize Path

Launch the satellite scenario viewer with `ShowDetails` set to `false`. When the `ShowDetails` property is set to `false`, only satellites and ground stations will be shown. Labels and orbits will be hidden. Mouse over satellites and ground stations to show their labels.The green lines represent the multi-hop path. Also, set `MarkerSize` of the satellites to 6 to further declutter the visualization. Set the camera position to 60 degrees latitude and 5 degrees longitude to bring the multi-hop path into view..

```
v = satelliteScenarioViewer(sc,"ShowDetails",false);
sat.MarkerSize = 6; % Pixels
campos(v,60,5);     % Latitude and longitude in degrees
```

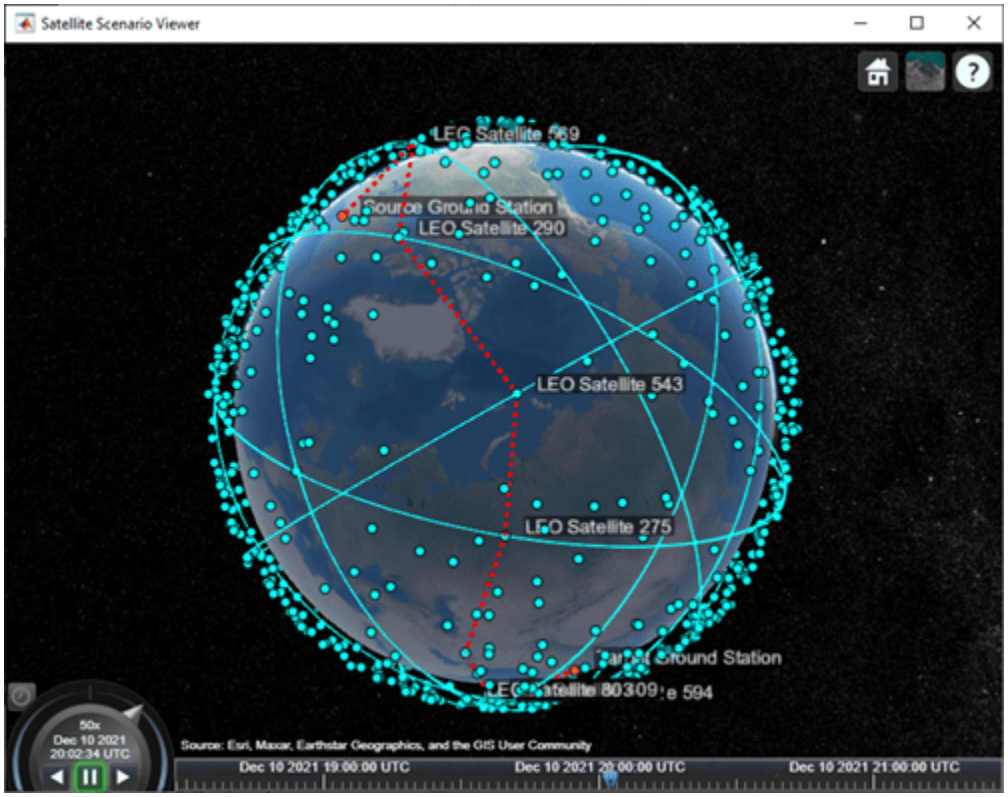


Click on the ground stations to display their labels without requiring to mouse over. Click on the satellites that are part of the multi-hop path to display their orbits and labels without requiring to mouse over.



Play the scenario.

```
play(sc);
```



Next Steps

This example demonstrated how to determine the multi-hop path through a large satellite constellation to gain access between two ground stations. Each subsequent node in the path was selected such that its distance to "Target Ground Station" was the minimum among those satellites whose elevation angle with respect to the previous node was at least -15 degrees. Note that this does not necessarily result in the shortest overall distance along the multi-hop path. One way to find the shortest path is to find all possible paths using a graph search algorithm and then choose the shortest path among them. To find the distance between the different satellites and ground stations, use the third output of the `aer` function.

Additionally, this example computed the path only once for the scenario `StartTime`, which was then used for the duration of the scenario. As a result, the ground stations had access only for 5 minutes out of the 3 hours spanning the scenario duration. To increase the access duration between the ground stations, you can modify the above code to re-compute the path at each scenario time step using `advance` after setting `AutoSimulate` to `false`. Calling `advance` advances `SimulationTime` by one `SampleTime`. Once you compute all the paths, set `AutoSimulate` back to true, create access objects corresponding to each unique path, and re-compute the access intervals by calling `accessIntervals` on all the access objects.

See Also

Objects

`satelliteScenario` | `satelliteScenarioViewer` | `Satellite` | `Access`

Functions

`access` | `accessIntervals` | `aer` | `play` | `satellite` | `pointAt`

More About

- “Satellite Scenario Key Concepts” on page 2-65
- “Satellite Scenario Overview” on page 2-74

Modeling Custom Satellite Attitude and Gimbal Steering

This example shows how to point a satellite or gimbal in a satellite scenario using logged orientation data from a `timetable` or `timeseries`. It uses data generated by the Aerospace Blockset™ **Spacecraft Dynamics** block. For more information about the data and how to generate it, see the Aerospace Blockset example *Analyzing Spacecraft Attitude Profiles with Satellite Scenario*.

The `satelliteScenario` object lets you load previously generated, time-stamped ephemeris and attitude data into a scenario as `timeseries` or `timetable` objects. Data is interpolated in the `scenario` object to align with the scenario time steps, allowing you to incorporate data generated in a Simulink model into either a new or existing `satelliteScenario` object. For this example, the satellite orbit and attitude states are precomputed by the **Spacecraft Dynamics** block. Load this data to the workspace and use it to model a satellite to a `satelliteScenario` object for access analysis.

Define Mission Parameters and Satellite Initial Conditions

Specify a start date and duration for the mission. This example uses MATLAB® structures to organize mission data. These structures make accessing data later in the example more intuitive. They also help declutter the global base workspace.

```
mission.StartDate = datetime(2021,1,1,12,0,0);
mission.Duration = hours(1.5);
```

Specify initial orbital elements for the satellite.

```
mission.Satellite.SemiMajorAxis  = 7.2e6; % meters
mission.Satellite.Eccentricity   = .05;
mission.Satellite.Inclination    = 70;    % deg
mission.Satellite.ArgOfPeriapsis = 0;     % deg
mission.Satellite.RAAN           = 215;   % deg
mission.Satellite.TrueAnomaly    = 200;   % deg
```

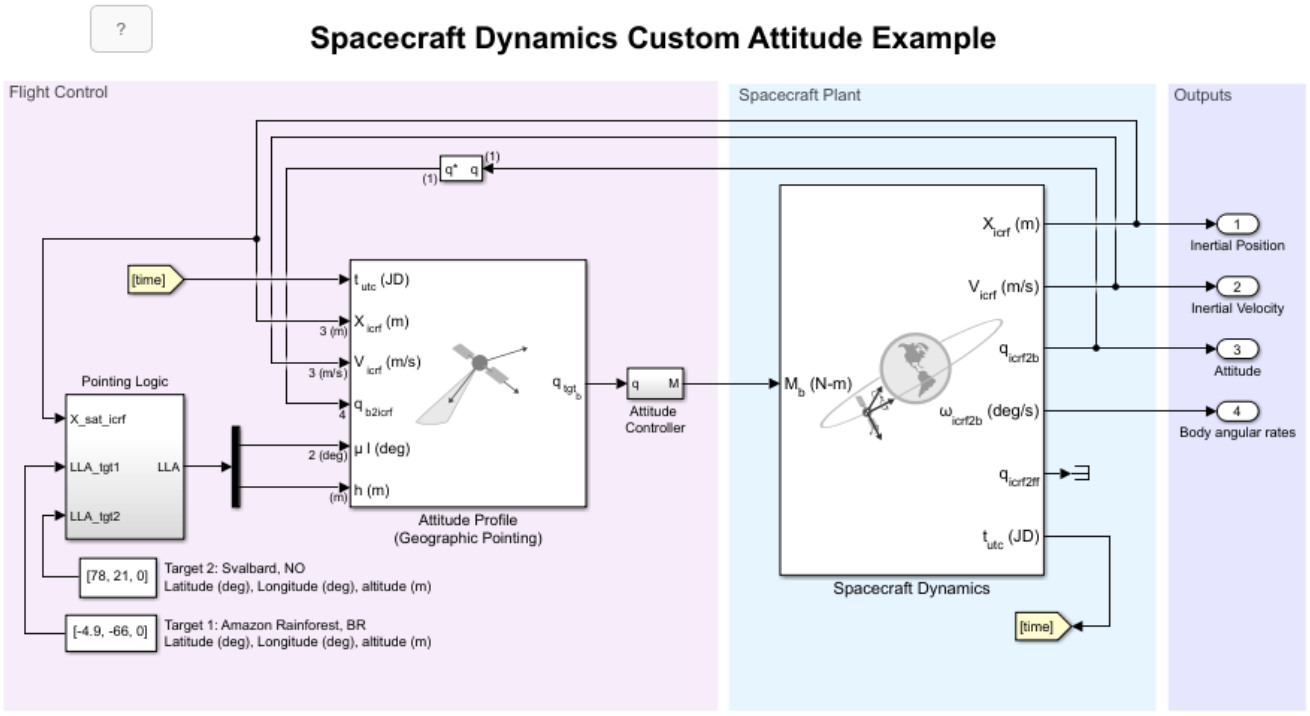
Specify an initial attitude state for the satellite.

```
mission.Satellite.q0 = [1, 0, 0, 0];
mission.Satellite.pqr = [0, 0, 0]; % deg/s
```

Load Ephemeris and Attitude Profile

The Simulink® model used to generate data for this example is configured to perform an Earth Observation mission during which a satellite performs a flyover of a region of the Amazon Rainforest to capture images of, and track deforestation trends in, the area.

The satellite points at the nadir when not actively imaging or downlinking to the ground station in Svalbard, NO. The Aerospace Blockset **Attitude Profile** block calculates commanded attitude. The satellite uses spherical harmonic gravity model EGM2008 for orbit propagation. Gravity gradient torque contributions are also included in attitude dynamics. For more information about this model, see the Aerospace Blockset example *Analyzing Spacecraft Attitude Profiles with Satellite Scenario*.



The `timetable` objects contain position and attitude data for the satellite throughout the mission. The data is referenced in the inertial (ICRF/GCRF) reference frame. Attitude values are expressed as quaternions, although Euler angles are also supported.

```
mission.Data = load("SatelliteScenarioCustomAttitudeData.mat", "PositionTimeTableGCRF", "Attitude
display(mission.Data.PositionTimeTableGCRF)
```

```
  1148×1 timetable

      Time                          Data
   __________    ____________________________________

   0 sec         5.2953e+06      4.784e+06     -2.422e+06
   5 sec         5.2783e+06     4.7859e+06    -2.4532e+06
   10 sec        5.2611e+06     4.7877e+06    -2.4842e+06
   13.263 sec    5.2499e+06     4.7888e+06    -2.5045e+06
   15 sec        5.2439e+06     4.7894e+06    -2.5152e+06

       :                             :

   5380 sec      6.3454e+06     3.4735e+06     2.1279e+06
   5385 sec      6.3479e+06     3.4893e+06     2.0962e+06
   5390 sec      6.3503e+06     3.5051e+06     2.0645e+06
   5395 sec      6.3526e+06     3.5207e+06     2.0327e+06
   5400 sec      6.3547e+06     3.5363e+06     2.0009e+06

   Display all 1148 rows.
```

```
display(mission.Data.AttitudeTimeTableGCRF2Body)
```

```
  1148×1 timetable

      Time                              Data
   __________    ____________________________________________

   0 sec            0.1509      0.48681      0.30311     -0.80522
   5 sec           0.15061      0.48761       0.3033     -0.80472
   10 sec          0.15003      0.48914      0.30368     -0.80375
   13.263 sec      0.14977      0.48986      0.30387     -0.80329
   15 sec          0.14967      0.49013      0.30395     -0.80311

       :                                :

   5380 sec      -0.043839     -0.72806     -0.33468      0.59666
   5385 sec       -0.04461     -0.72663      -0.3346      0.59838
   5390 sec       -0.04538     -0.72521     -0.33451      0.60009
   5395 sec      -0.046149     -0.72378     -0.33443      0.60181
   5400 sec      -0.046919     -0.72235     -0.33434      0.60351

   Display all 1148 rows.
```

Create the Satellite Scenario

Create a satellite scenario object to use for analysis. Specify a timestep of 1 minute.

```
scenario = satelliteScenario(mission.StartDate, ...
    mission.StartDate + mission.Duration, 60);
```

Add the two targets as ground stations in Brazil and Svalbard.

```
gsNO = groundStation(scenario, 78, 21, Name="Svalbard, NO")
```

```
gsNO =
  GroundStation with properties:

                 Name:  Svalbard, NO
                   ID:  1
             Latitude:  78 degrees
            Longitude:  21 degrees
             Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
          MarkerColor:  [1 0.4118 0.1608]
           MarkerSize:  6
            ShowLabel:  true
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
```

```
gsAmazon = groundStation(scenario, -4.9, -66, Name="Amazon Rainforest")
```

```
gsAmazon =
  GroundStation with properties:

                 Name:  Amazon Rainforest
```

```
                ID:  2
          Latitude:  -4.9 degrees
         Longitude:  -66 degrees
          Altitude:  0 meters
   MinElevationAngle:  0 degrees
      ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
             Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
        Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
           Receivers:  [1x0 satcom.satellitescenario.Receiver]
            Accesses:  [1x0 matlabshared.satellitescenario.Access]
         MarkerColor:  [1 0.4118 0.1608]
          MarkerSize:  6
           ShowLabel:  true
      LabelFontColor:  [1 1 1]
       LabelFontSize:  15
```

Add the Satellite From the Loaded Trajectory

Add the observation satellite to the scenario.

```
sat = satellite(scenario, mission.Data.PositionTimeTableGCRF, ...
    "CoordinateFrame", "inertial", "Name", "ObservationSat");
```

Add a conical sensor to the satellite, with a 35 deg half angle to represent the onboard camera. Enable field of view visualization in the scenario viewer. To assist in visualization, the sensor is mounted 10m from the satellite in the $+z$ direction.

```
snsr = conicalSensor(sat, MaxViewAngle=70, MountingLocation=[0 0 10]);
fieldOfView(snsr);
```

Add access between the conical sensor and the two ground stations.

```
acNO = access(snsr, gsNO)
```

```
acNO =
  Access with properties:

    Sequence:  [4 1]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

```
acAmazon = access(snsr, gsAmazon)
```

```
acAmazon =
  Access with properties:

    Sequence:  [4 2]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

Point the Satellite With the Loaded Attitude Profile

Use the `pointAt` method to associate the logged attitude `timetable` with the satellite. Parameter `ExtrapolationMethod` controls the pointing behavior outside of the `timetable` range.

```
pointAt(sat, mission.Data.AttitudeTimeTableGCRF2Body, ...
    "CoordinateFrame", "inertial", "Format", "quaternion", "ExtrapolationMethod", "nadir");
```

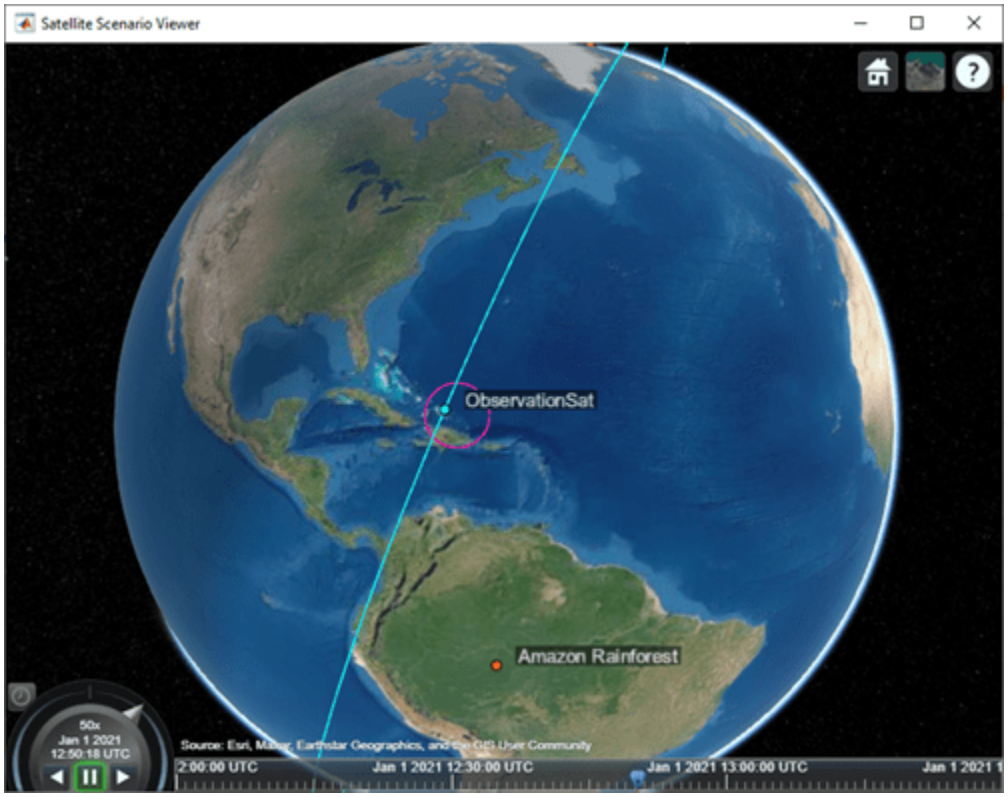
Visualize the Scenario

Open the **Satellite Scenario Viewer** to view and interact with the scenario.

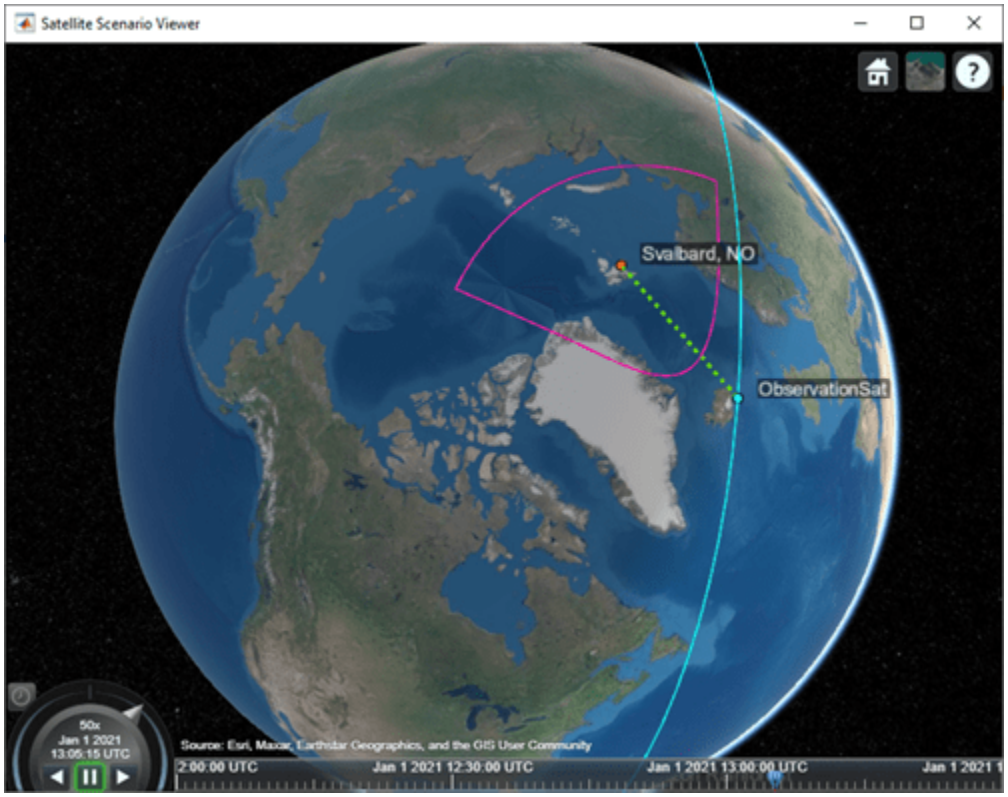
```
viewer1 = satelliteScenarioViewer(scenario);
```

The satellite points at the nadir to begin the scenario. As it nears Target 1 in the Amazon Rainforest, it slews to point and track this target.

After the imaging segment is complete, the satellite returns to pointing at the nadir.



As the satellite comes into range of the arctic ground station, it slews to point at this target.



Custom Gimbal Steering

This example shows how to import custom attitude data for a simple Earth Observation satellite mission, where the onboard camera is fixed to the satellite body. Another common approach is to fix the sensor on a gimbal and orient the sensor by manuevering the gimbal, rather than the spacecraft body itself. Modify the above scenario to mount the sensor on a gimbal and steer the gimbal to perform uniform sweeps of the area directly below the satellite.

Reset the satellite to always point at the nadir, overwriting the previously provided custom attitude profile.

```
delete(viewer1);
pointAt(sat, "nadir");
```

Delete the existing sensor object to remove it from the satellite and attach a new sensor with the same properties to a gimbal.

```
delete(snsr);
gim = gimbal(sat);
snsr = conicalSensor(gim, MaxViewAngle=70, MountingLocation=[0 0 10]);
fieldOfView(snsr);
```

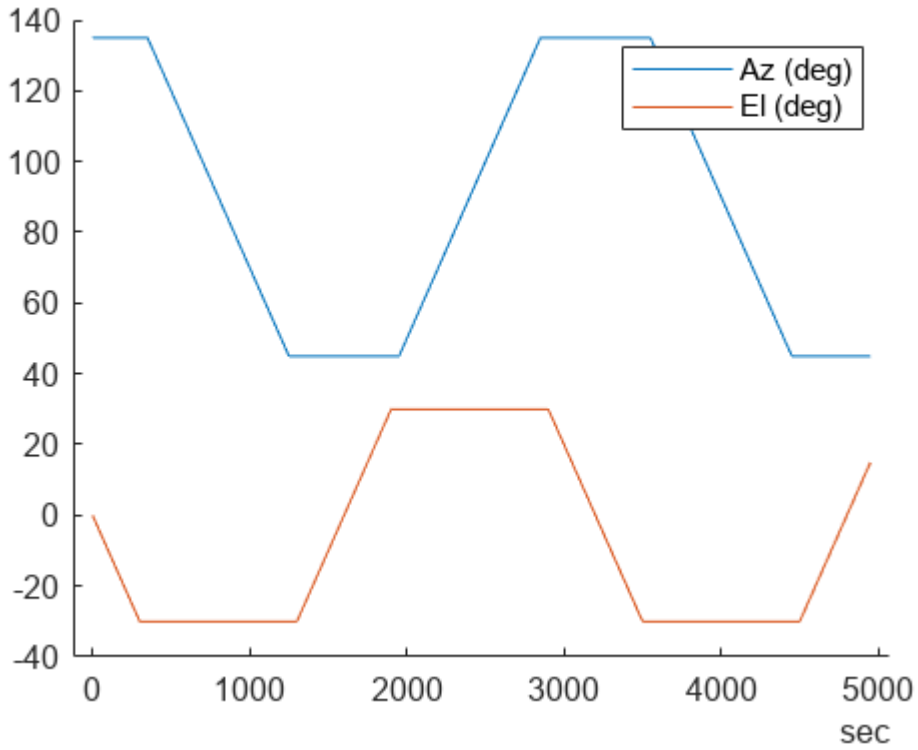
Define azimuth and elevation angles for gimbal steering to model a sweeping pattern over time below the satellite.

```
gimbalSweep.Time = seconds(1:50:5000)';

gimbalSweep.Az = [...
    45*ones(1,7),...
```

```
    45:-5:-45,...
    -45*ones(1,13),...
    -45:5:45,...
    45*ones(1,13),...
    45:-5:-45,...
    -45*ones(1,13)];
gimbalSweep.Az(end-2:end) = [];
gimbalSweep.Az = gimbalSweep.Az + 90;

gimbalSweep.El = [...
    0:-5:-30,...
    -30*ones(1,19),...
    -30:5:30,...
    30*ones(1,19),...
    30:-5:-30,...
    -30*ones(1,19),...
    -30:5:30];
gimbalSweep.El(end-2:end) = [];
```

Plot the commanded azimuth and elevation values over time.

```
figure(1)
hold on;
plot(gimbalSweep.Time', gimbalSweep.Az);
plot(gimbalSweep.Time', gimbalSweep.El);
hold off;
legend(["Az (deg)", "El (deg)"]);
```



Store the azimuth and elevation angles in a `timetable`.

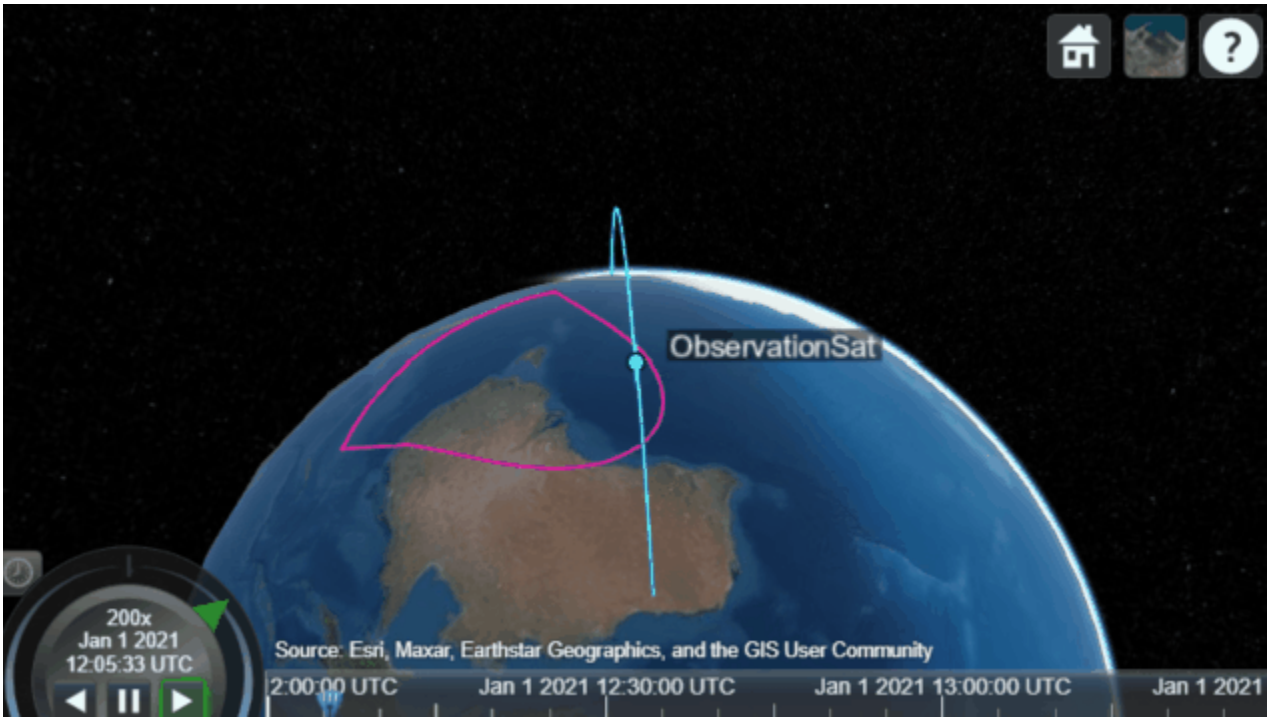
```
gimbalSweep.TT = timetable(gimbalSweep.Time, [gimbalSweep.Az', gimbalSweep.El']);
```

Steer the gimbal with the `timetable`. The gimbal returns to its default orientation for timesteps that are outside of the provided data.

```
pointAt(gim, gimbalSweep.TT);
```

View the updated scenario in the **Satellite Scenario Viewer**.

```
viewer2 = satelliteScenarioViewer(scenario);
```



See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer` | `Satellite` | `GroundStation` | `ConicalSensor` | `Gimbal` | `Access` | `FieldOfView`

Functions
`pointAt`

Related Examples

- “Modeling Satellite Constellations Using Ephemeris Data” on page 5-130
- “Analyzing Spacecraft Attitude Profiles with Satellite Scenario” (Aerospace Blockset)
- “Satellite Constellation Access to Ground Station” on page 5-140

More About

- “Satellite Scenario Key Concepts” on page 2-65

- “Satellite Scenario Overview” on page 2-74

Aircraft Design Optimization with the Fixed-Wing Object

This example shows how to optimize aircraft size and geometry using the Aero.FixedWing object to enforce static stability constraints.

Aircraft Design Objective

The aircraft is designed for the Regular Class of the Society of Automotive Engineers (SAE) Aero Design competition. In this competition, the teams aim to maximize their flight score, which is calculated as follows.

$$FS = 3\frac{2N_S + 2.2W_{BP}}{b_w + l_{cargo}}$$

where:

N_s: Number of spherical payloads

W_{BP}: Weight of boxed payloads [kg]

b_W: Wingspan [m]

l_{cargo}: Length of cargo bay [m]

All these variables are essentially continuous, except for N_s, which is discrete. This variable complicates the problem formulation. Thus, only the $N_s = 1$ case is considered.

Optimization Constraints

The competition imposes constraints that are important to the optimization:

- The gross takeoff weight cannot exceed 24.9 kg.
- The wingspan cannot exceed 3 m.
- The takeoff distance cannot exceed 30.5 m.

Beyond this, impose additional requirements to favor a conventional design:

- The aircraft must be statically stable.
- No interference is allowed between the wing, tail, payload, avionics, and landing gear.
- The wing and horizontal tail aspect ratios must be high enough for Prandtl lifting-line theory to be valid.
- The wing and tail aspect ratios must be low enough to guarantee structural strength.
- The wing aerodynamic center must be in the front half of the fuselage.
- The static margin must be constrained so the aircraft pitches easily, without being too sensitive to disturbances.

Additionally, all optimization variables are physical quantities and must be positive.

Assumptions

To further simplify the formulation, assume the wing has a simple geometry, with:

- No dihedral
- No sweep
- No wing twist
- No wing tip devices
- 0° incidence angle
- Wing Z-location at fuselage centerline

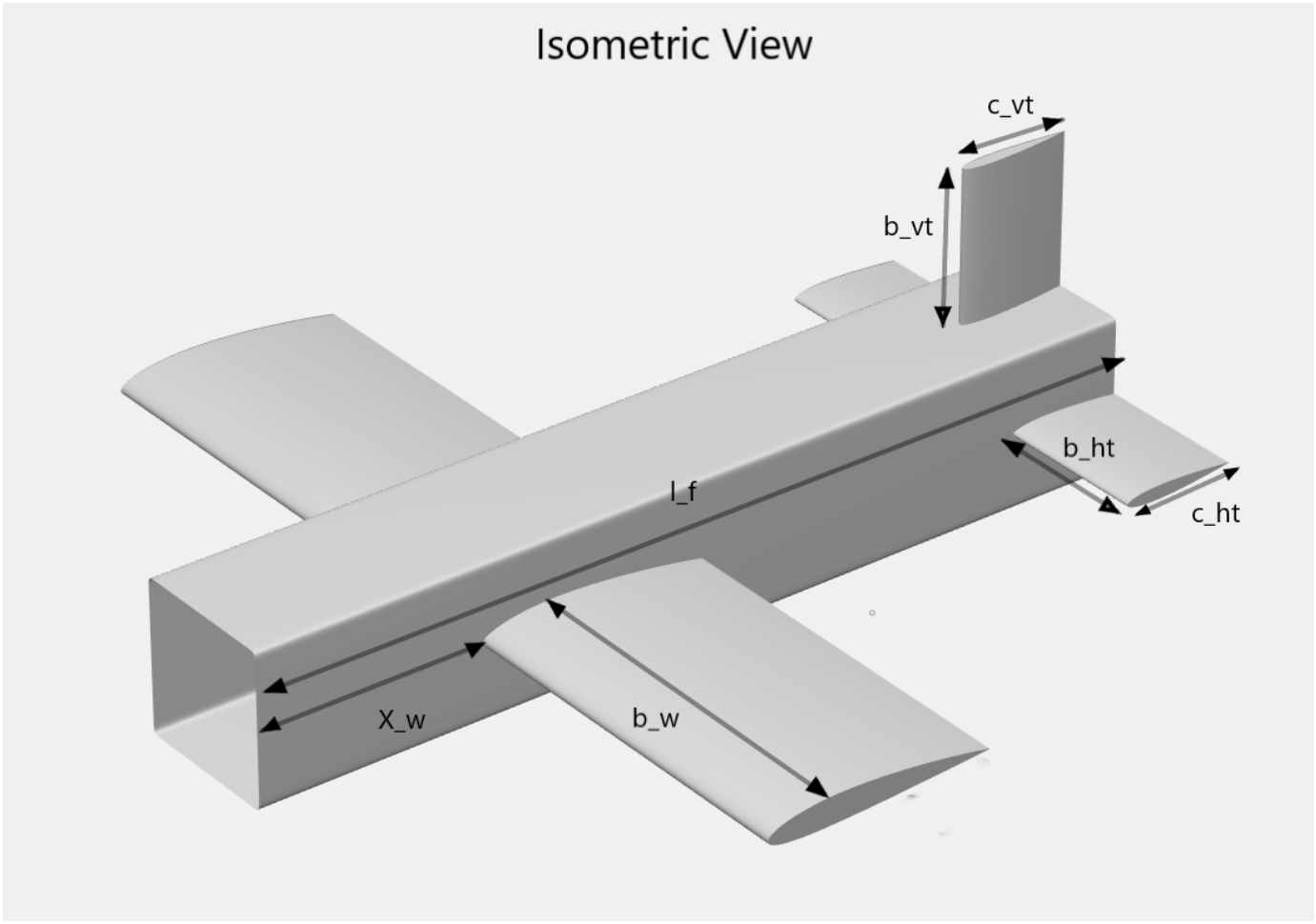
The design also omits the effect of any control surfaces and the propulsion system.

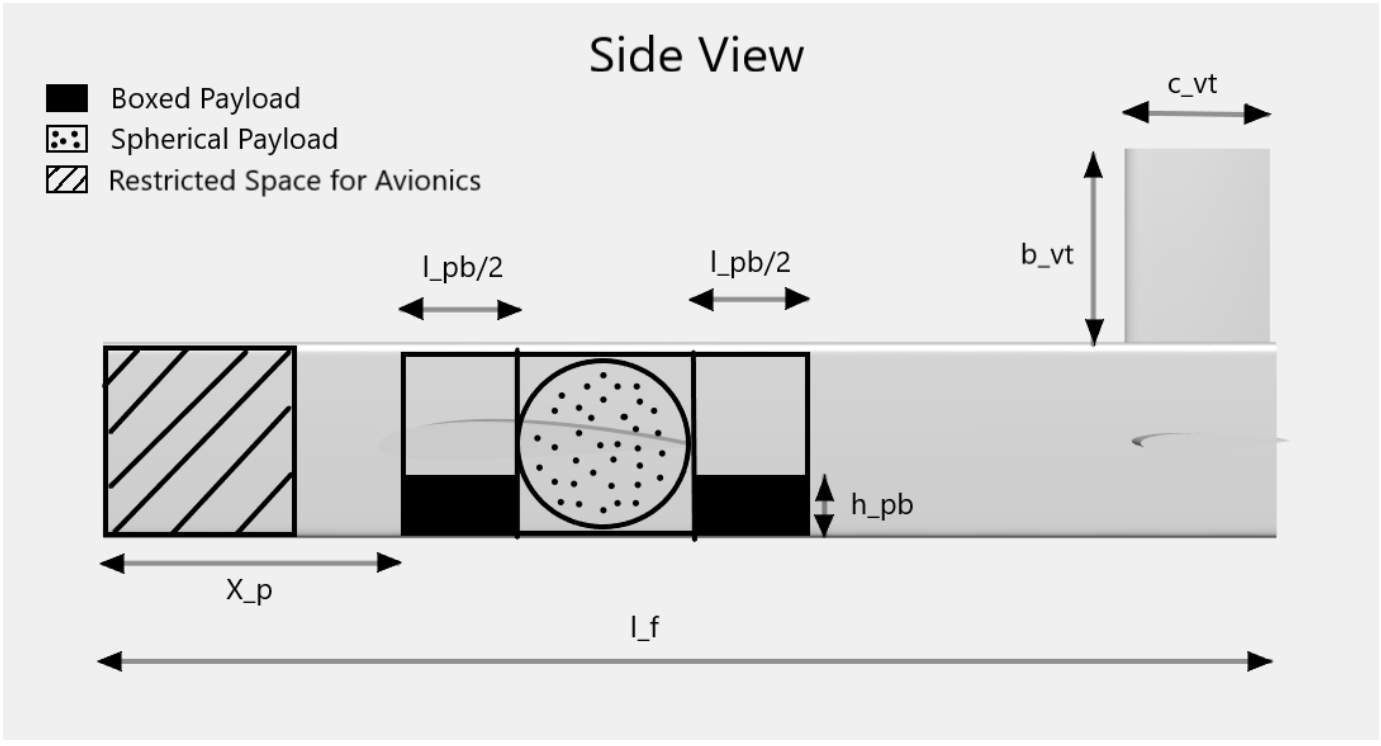
Optimization Problem Formulation

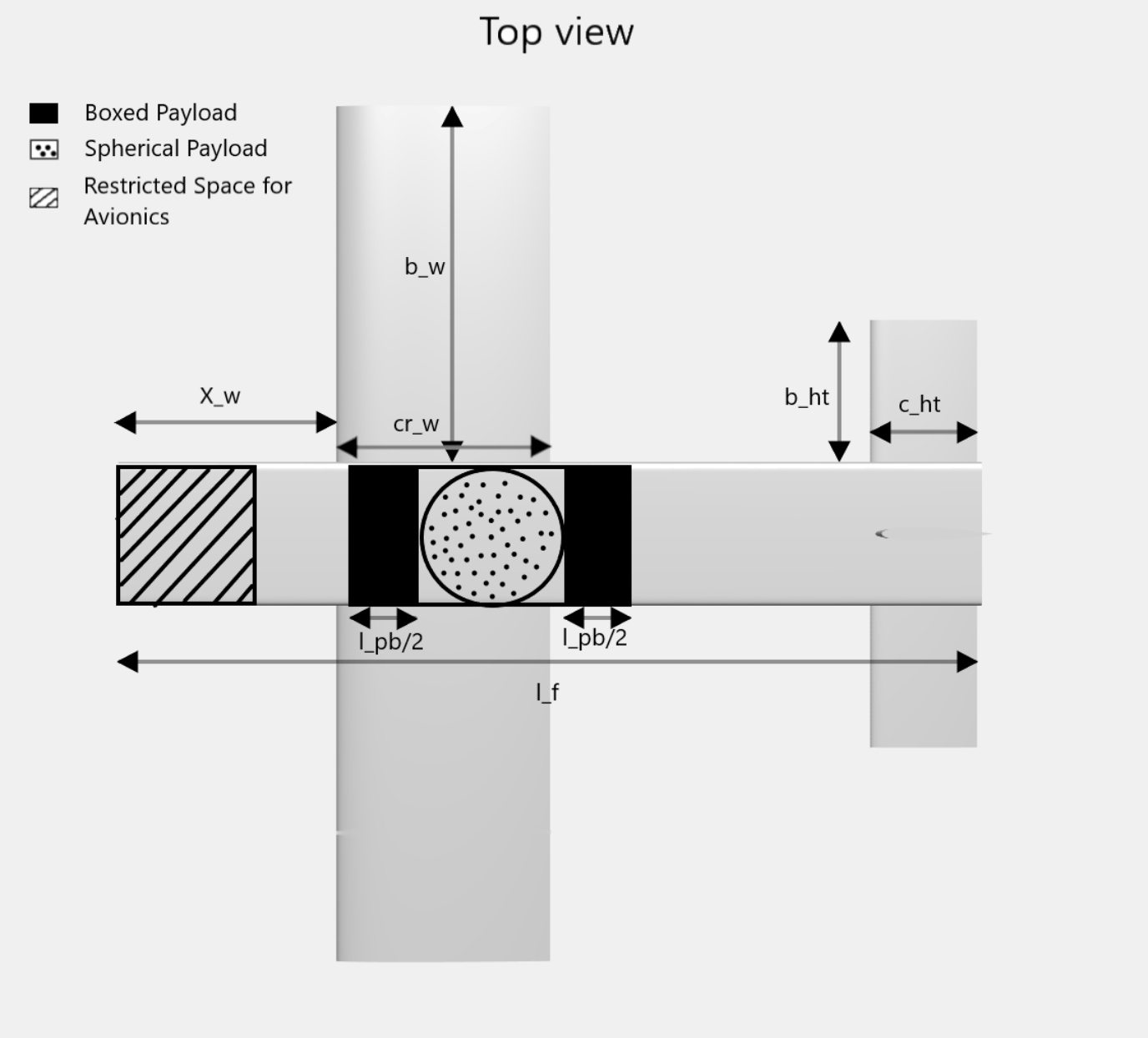
To optimize the aircraft geometry, use the problem-based approach. Start by defining problem constants and the optimization variables using the helper function `initializeAircraft`, which is included with this example. Organize these variables into six structures: `aircraft`, `wing`, `hTail`, `vTail`, `fuselage`, and `payload`. See a list of the optimization variables and their physical representation below.

```
[aircraft, wing, fuselage, hTail, vTail, payload, initialValues] = initializeAircraft;
```

Symbolic Variable	Physical Quantity
b_w	Wing Half Span
cr_w	Wing Root Chord
lambda_w	Wing Taper Ratio
X_w	Wing X Location
b_ht	Horizontal Tail Half Span
c_ht	Horizontal Tail Chord
b_vt	Vertical Tail Half Span
c_vt	Vertical Tail Chord
l_f	Length of fuselage
l_pb	Length of Boxed Payload
h_pb	Height of Boxed Payload
X_p	Cargo Bay X Location


Isometric View
c_vt
b_vt
b_ht
c_ht
l_f
X_w
b_w


Side View
Boxed Payload
Spherical Payload
Restricted Space for Avionics
c_vt
b_vt
l_pb/2
l_pb/2
h_pb
X_p
l_f



Set up the design optimization problem

Define an empty optimization problem and set maximization as its objective.

```
designprob = optimproblem('ObjectiveSense','maximize');
```

Estimate lift and drag coefficients for the wing, tail, and fuselage.

```
[aircraft, wing, fuselage, hTail, vTail, designprob] = ...
    addAerodynamics(aircraft, wing, fuselage, hTail, vTail, designprob);
```

Use the component build-up method to formulate the aircraft mass and introduce the gross takeoff weight constraint.

```
[aircraft, wing, fuselage, hTail, vTail, payload, designprob] = ...
    addWeightAndSizing(aircraft, wing, fuselage, hTail, vTail, payload, designprob);
```

To compute the static stability derivatives, create an instance of the Aero.FixedWing and Aero.FixedWing.State objects.

```
[myAircraft, CruiseState] = createFixedWing(aircraft.Mass, wing, hTail, initialValues);
```

Add the static stability constraints to the optimization problem. The following ten derivatives play a major role in an aircraft response to external disturbances and so are used as constraints: C_{D_u}, C_{m_u}, C_{Y_v}, C_{L_w}, C_{m_α}, C_{l_β}, C_{n_β}, C_{l_p}, C_{m_q}, C_{n_r}.

To compute these derivatives, the `addStability` function calculates all aerodynamic coefficients and the mass moment of inertia matrix for the aircraft. With this information, use the `staticStability` method of the Aero.FixedWing object to obtain the ten stability derivatives.

```
[aircraft, wing, vTail, designprob] = ...
    addStability(aircraft, wing, fuselage, hTail, vTail, payload, designprob, myAircraft, CruiseS
```

Estimate takeoff performance and include maximum takeoff distance, or ground roll, constraint.

```
[aircraft,designprob] = addPerformance(aircraft, wing.PlanformArea, wing.MeanChord, designprob);
```

Write the objective function to match the SAE Aero Design Regular Class flight score. Units are converted to metric.

```
designprob.Objective = 3*(2+2.2*payload.Boxed.Mass)/((2*wing.HalfSpan+payload.Length));
```

Set options for optimization solver

Let the solver run to completion by setting the maximum number of function evaluations and iterations to infinity.

```
options = optimoptions(designprob);
options.MaxFunctionEvaluations = Inf;
options.MaxIterations = Inf;
```

If steps become too small, stop the optimization early.

```
options.StepTolerance = 1e-4;
```

Have the solver automatically plot the results of each iteration while it runs.

```
options.PlotFcn = {'optimplotconstrviolation', 'optimplotfvalconstr'};
```

Use a parallel pool to speed up the computation.

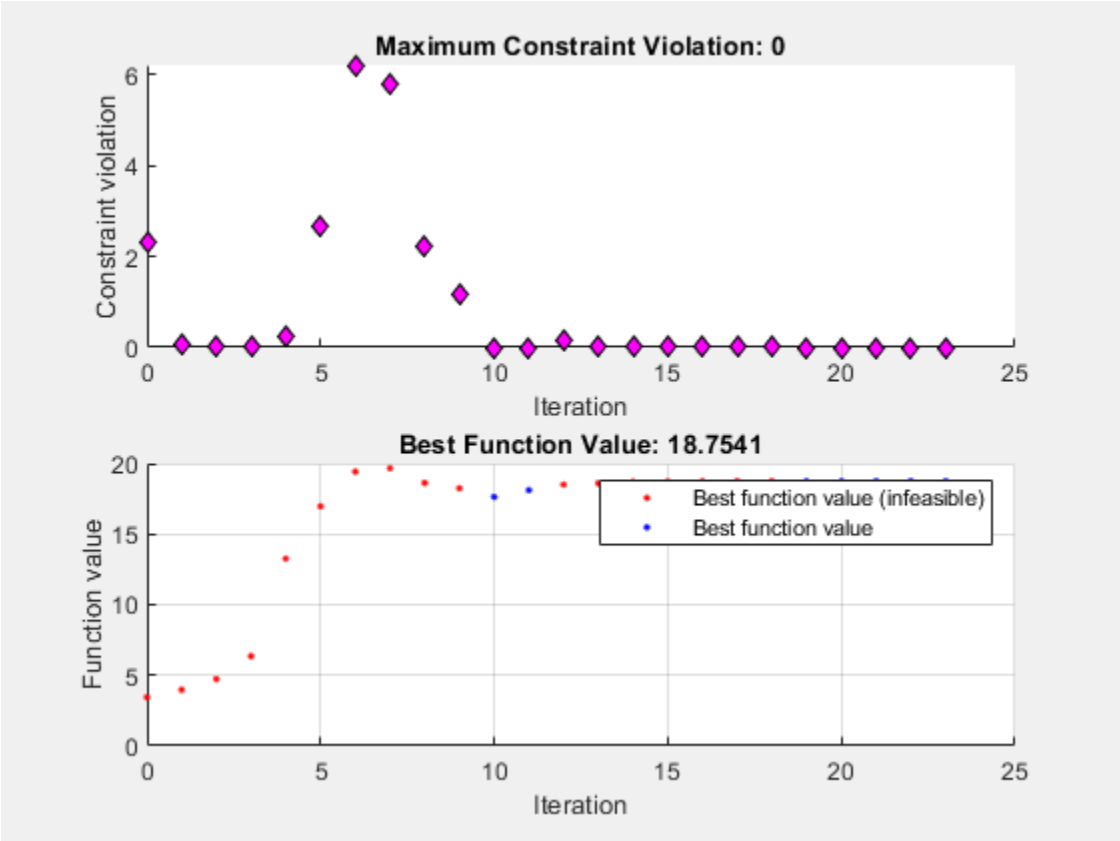
```
options.UseParallel = true;
options.Display = 'off';
```

Solve the optimization problem

Solving the problem returns both the maximum flight score for this design and the corresponding values of the 12 optimization variables.

```
[finalValues, maxScore] = solve(designprob, initialValues, 'Options', options);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 20 workers.
```




```
maxScore
```

```
maxScore = 18.7541
```

The solution quickly gets close to the optimal value, and most steps are spent on the final improvements. Due to the nonlinearity of this problem, the solution might not be globally optimal. Different initial conditions might result in a predicted flight score better than 18.754. The optimized design has a final geometry with a large tapered wing and a moderately sized tail placed at the end of the fuselage.

References

[1] Roskam, Jan. *Airplane Design, Part VI: Preliminary Calculation of Aerodynamic, Thurst and Power Characteristics*. Lawrence, KA: DAR Corporation, 2008.

[2] Roskam, Jan. *Airplane Flight Dynamics and Automatic Flight Controls, Part I*. Lawrence, KA: DAR Corporation, 2003.

[3] Raymer, Daniel P. *Aircraft Design: A Conceptual Approach*. Washington, DC: AIAA, 1992.

See Also

`Aero.FixedWing` | `Aero.FixedWing.State`

Compute Indicated Airspeed for Pitot-Static Airspeed Indicator

This example shows how to compute the indicated airspeed from true airspeed for a pitot-static airspeed indicator using the Aerospace Toolbox `correctairspeed` function.

Provide a True Airspeed

True airspeed is the airspeed that we would read ideally (and the airspeed value easily calculated using scripts or functions).

```
trueAirspeedInKTS = 72;
```

Calculate the Calibrated Airspeed

To calculate the calibrated airspeed, you adjust the true airspeed for errors introduced through the pitot-static airspeed indicators used to determine airspeed. These measurement errors are density error, compressibility error, and calibration error. The `correctairspeed` function can apply the density error and compressibility error computing calibrated airspeed from true airspeed.

Density Error

An airspeed indicator reads lower than true airspeed at higher altitudes. This is due to lower air density at altitude. 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, the result is in equivalent airspeed (EAS). Equivalent airspeed is true airspeed modified with the changes in atmospheric density that affect the airspeed indicator.

Compressibility Error

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.

```
altitudeInFt = 500;
altitude = convlength(altitudeInFt,'ft','m');

[~, speedOfSound, pressure0, ~] = atmoscoesa(altitude);

calibratedAirspeedInKTS = correctairspeed(trueAirspeedInKTS, speedOfSound, pressure0, 'TAS', 'CAS
```

Adjust to Indicated Airspeed for Pitot-Static Airspeed Indicator

Calibration error of the pitot-static airspeed indicator is the last adjustment to the airspeed value. The adjustment results in an indicated airspeed displayed on the airspeed indicator in the cockpit.

Calibration Error

The airspeed indicator has static vent(s) to maintain a pressure equal to atmospheric pressure inside the instrument. Position and placement of the static vent, angle of attack, and velocity of the aircraft determines the pressure inside the airspeed indicator and the amount of calibration error of the

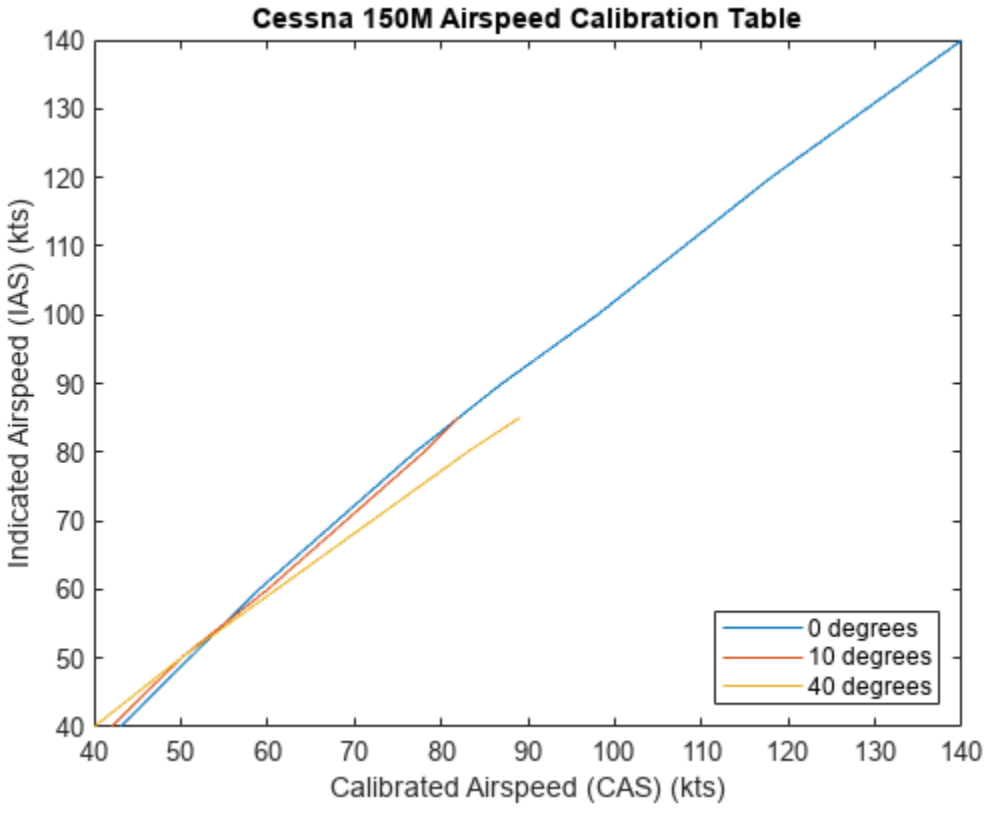
airspeed indicator. The calibration error is specific to a given aircraft design. A calibration table is usually given in the pilot operating handbook (POH) or in other aircraft specifications.

Example Calibration Tables

As an example, here is the Cessna 150M airspeed calibration table from "Pilot's Operating Handbook, Cessna 1976 150 Commuter, Cessna Model 150M", Cessna Aircraft Company, Wichita, Kansas, USA, 1976.

Calibration table for 0, 10, and 0 degrees of flap:

```
flaps0IAS = 40:10:140;
flaps0CAS = [43 51 59 68 77 87 98 108 118 129 140];

flaps10IAS = [40:10:80 85];
flaps10CAS = [42 50 60 69 78 82];

flaps40IAS = [40:10:80 85];
flaps40CAS = [40 50 61 72 83 89];

plot(flaps0CAS,flaps0IAS,flaps10CAS,flaps10IAS,flaps40CAS,flaps40IAS)
xlabel('Calibrated Airspeed (CAS) (kts)');
ylabel('Indicated Airspeed (IAS) (kts)');
title('Cessna 150M Airspeed Calibration Table');
legend ('0 degrees','10 degrees','40 degrees','Location','southeast');
```



Using this calibration table, the indicated airspeed (IAS) is determined from the calibrated airspeed (CAS).

```
flapSetting = 40 degrees;

if (flapSetting == 40)
    tableCAS = flaps40CAS;
    tableIAS = flaps40IAS;
elseif (flapSetting == 0)
    tableCAS = flaps0CAS;
    tableIAS = flaps0IAS;
else
    tableCAS = flaps10CAS;
    tableIAS = flaps10IAS;
end

indicatedAirspeedInKTS = interp1(tableCAS,tableIAS,calibratedAirspeedInKTS);
```

Indicated airspeed is displayed on the airspeed indicator in the cockpit instrumentation.

```
fig = uifigure('Name','Airspeed Indicator',...
    'Position',[50,650,220,220],...
    'Color',[0.2667 0.2706 0.2784],'Resize','off','Visible','off');
air = uiaeroairspeed('Parent',fig,'Position',[10 10 200 200], ...
    'Limits',[25 160],'ScaleColorLimits',[42,85; 47,107; 107,141; 141,145], ...
    'Airspeed', indicatedAirspeedInKTS);
fig.Visible = "on";
```

See Also

`correctairspeed` | `atmoscoesa` | `uiaeroairspeed`

Differences Between Corrected Airspeeds

This example shows the differences between corrected airspeeds and true airspeed (TAS). The corrected airspeeds are indicated airspeed (IAS), calibrated airspeed (CAS), and equivalent airspeed (EAS). You can use the Aerospace Toolbox `correctairspeed` function to calculate TAS, CAS, and EAS from one of the other two.

TAS: True airspeed

True airspeed is the airspeed that we would read ideally (and the airspeed value easily calculated using scripts or functions).

```
TASInKTS = 72;
```

To determine the speed of sound and pressure at altitude for a standard atmospheric day required to make the airspeed corrections, select an altitude.

```
altitudeInFt = 2500;
altitude = convlength(altitudeInFt,'ft','m');
[~, speedOfSound, pressure0, ~] = atmoscoesa(altitude);
```

EAS: Correcting for Density Error

An airspeed indicator reads lower than true airspeed (TAS) at higher altitudes. This is due to lower air density at altitude. 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, the result is in equivalent airspeed (EAS). Equivalent airspeed is true airspeed modified with the changes in atmospheric density that affect the airspeed indicator.

```
EASInKTS = correctairspeed(TASInKTS, speedOfSound, pressure0, 'TAS', 'EAS');
```

CAS: Correcting for Compressibility Error

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.

```
CASInKTS = correctairspeed( TASInKTS, speedOfSound, pressure0, 'TAS', 'CAS');
```

IAS: Correcting for Instrument and Position Error

Indicated airspeed (IAS) is displayed on the airspeed indicator in the cockpit. To calculate IAS, the calibration error of the pitot-static airspeed indicator is applied to the calibrated airspeed.

Calibration Error

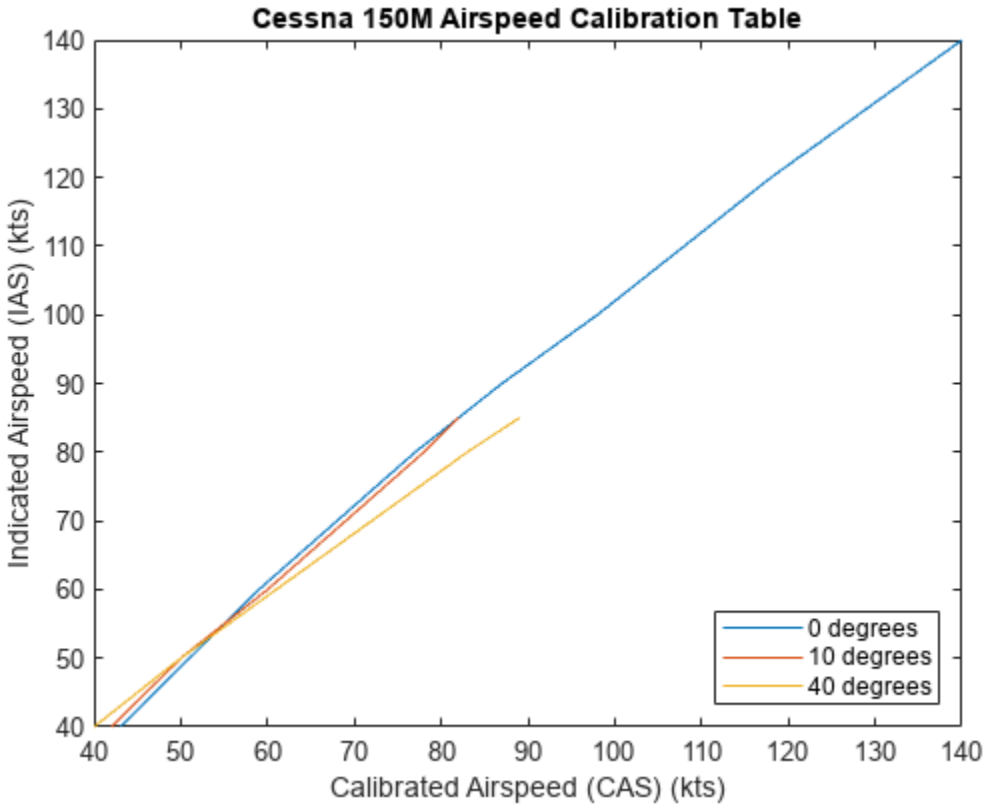
The airspeed indicator has static vent(s) to maintain a pressure equal to atmospheric pressure inside the instrument. Position and placement of the static vent, angle of attack, and velocity of the aircraft determines the pressure inside the airspeed indicator and the amount of calibration error of the airspeed indicator. The calibration error is specific to a given aircraft design. A calibration table is usually given in the pilot operating handbook (POH) or in other aircraft specifications.

Example Calibration Tables

As an example, here is the Cessna 150M airspeed calibration table from "Pilot's Operating Handbook, Cessna 1976 150 Commuter, Cessna Model 150M", Cessna Aircraft Company, Wichita, Kansas, USA, 1976.

Calibration table for 0, 10 and 0 degrees of flap:

```
flaps0IAS = 40:10:140;
flaps0CAS = [43 51 59 68 77 87 98 108 118 129 140];

flaps10IAS = [40:10:80 85];
flaps10CAS = [42 50 60 69 78 82];

flaps40IAS = [40:10:80 85];
flaps40CAS = [40 50 61 72 83 89];

plot(flaps0CAS,flaps0IAS,flaps10CAS,flaps10IAS,flaps40CAS,flaps40IAS)
xlabel('Calibrated Airspeed (CAS) (kts)');
ylabel('Indicated Airspeed (IAS) (kts)');
title('Cessna 150M Airspeed Calibration Table');
legend ('0 degrees','10 degrees','40 degrees','Location','southeast');
```



Using this calibration table, the indicated airspeed (IAS) is determined from calibrated airspeed.

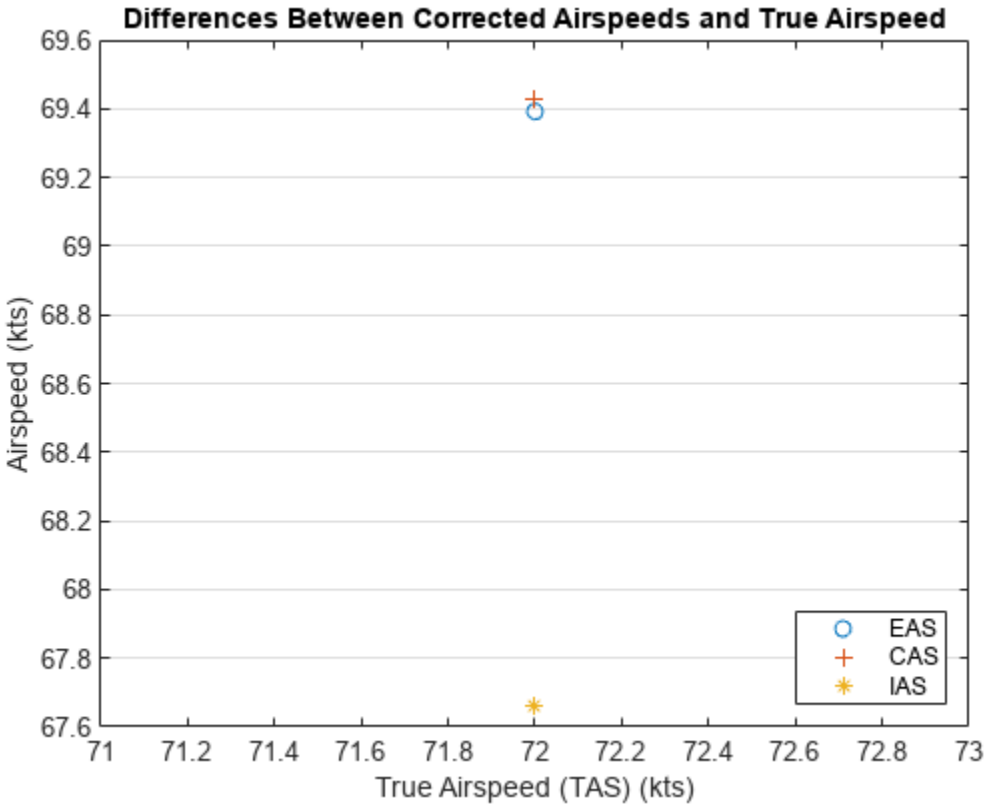
```
flapSetting = 40 degrees;
```

```
if (flapSetting == 40)
    tableCAS = flaps40CAS;
    tableIAS = flaps40IAS;
elseif (flapSetting == 0)
    tableCAS = flaps0CAS;
    tableIAS = flaps0IAS;
else
    tableCAS = flaps10CAS;
    tableIAS = flaps10IAS;
end

IASInKTS = interp1(tableCAS,tableIAS,CASInKTS);
```

Plot Airspeed Comparisons

Using the plot command, you can visualize the impact of correcting for the various sources of error in measuring airspeed compared to true airspeed.

```
plot(TASInKTS,EASInKTS,'o',TASInKTS,CASInKTS,'+',TASInKTS,IASInKTS,'*');
xlabel('True Airspeed (TAS) (kts)');
ylabel('Airspeed (kts)');
title('Differences Between Corrected Airspeeds and True Airspeed');
legend ('EAS','CAS','IAS','Location','southeast');
set(gca,"XGrid","off","YGrid","on");
```



See Also

`correctairspeed` | `atmoscoesa` | `uiaeroairspeed`

Analyze Access Between a Satellite Constellation and an Aircraft

This example shows how model an aircraft trajectory and a Walker-Delta satellite constellation using Aerospace Toolbox, and analyze and visualize the computed access between the aircraft and satellite constellation.

A `satelliteScenario` object represents a mission scenario consisting of satellites, ground stations, and the interactions between them. Use this object to model satellites and satellite constellations, model ground station networks, perform access analyses between satellites and ground stations, and visualize trajectories and analysis results.

In this example, model a Walker-Delta satellite constellation using the `walkerDelta` method of the `satelliteScenario` object. The `satelliteScenario` object also supports loading previously generated, time-stamped trajectory and attitude data into a scenario as `timeseries` or `timetable` objects. The aircraft trajectory is loaded from a MAT-file, and the data is added to the `satelliteScenario` as a `timetable` to analyze the interaction between the aircraft and the satellite constellation.

Define Mission Start Date and Duration

Specify a start date and duration for the mission.

```
mission.StartDate = datetime(2021,1,1,12,0,0);
mission.Duration = hours(2.5);
```

Load the Aircraft Trajectory

The provided aircraft trajectory begins at a latitude of 39.5 degrees, longitude of -75 degrees, and an altitude of 10000m, ending at a latitude of 45.9 degrees, longitude of -61.2 degrees, and an altitude of 10000m. The flight time is 2.5 hours. The trajectory is sampled every 10 seconds.

```
aircraft = load("aircraftLLA.mat", "trajectory");
```

Plot the aircraft trajectory using **`geoplot`**.

```
geoplot(aircraft.trajectory.LLA(:,1), aircraft.trajectory.LLA(:,2), "b-");
geolimits([30 50],[-110 -50]);
```

Add the Aircraft to a New Satellite Scenario

Create a `satelliteScenario` object for the mission.

```
mission.scenario = satelliteScenario(mission.StartDate, ...
    mission.StartDate + mission.Duration, 60);
```

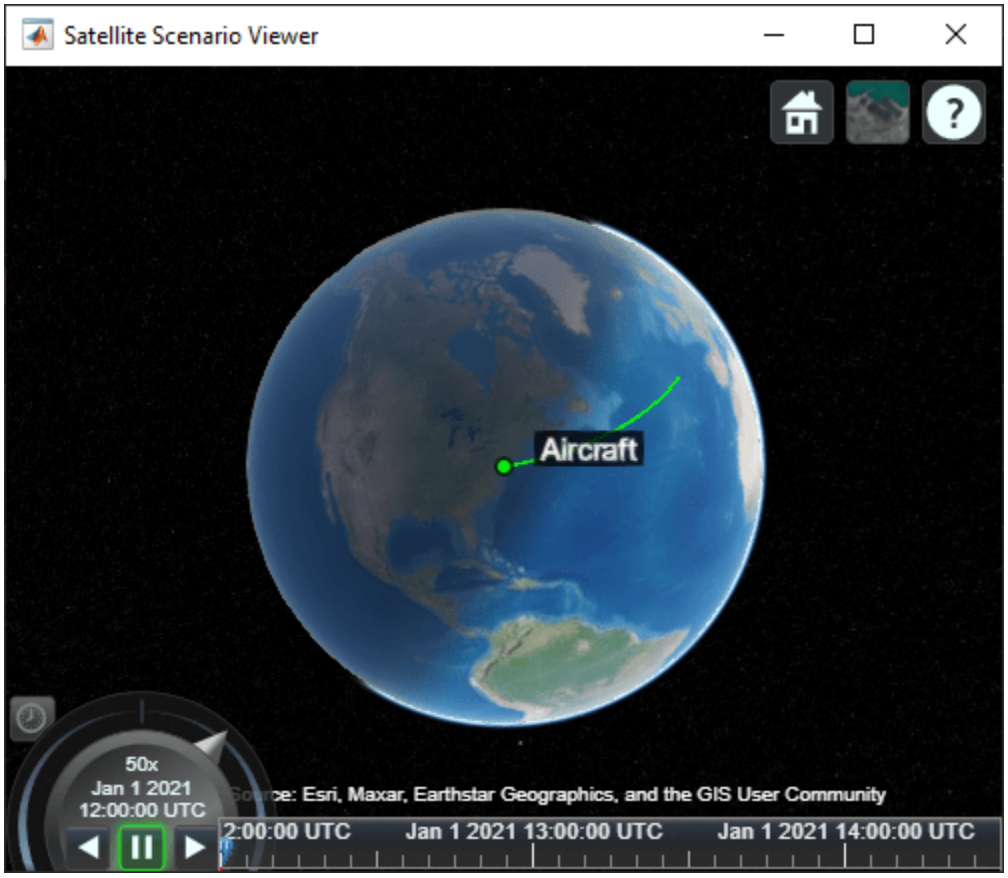
Add the aircraft to the scenario. Aircraft trajectories can be represented in a satellite scenario using the `Satellite` object with a custom trajectory loaded from a `timetable` or `timeseries` object. Use the latitude, longitude, and altitude data previously loaded to represent the aircraft trajectory in the scenario.

```
aircraft.obj = satellite(mission.scenario,aircraft.trajectory, CoordinateFrame="geographic", Name
aircraft.obj.MarkerColor = "green";
aircraft.obj.Orbit.LineColor = "green";
```

Visualize the generated aircraft trajectory using the Satellite Scenario Viewer.

```
mission.viewer = satelliteScenarioViewer(mission.scenario);
```

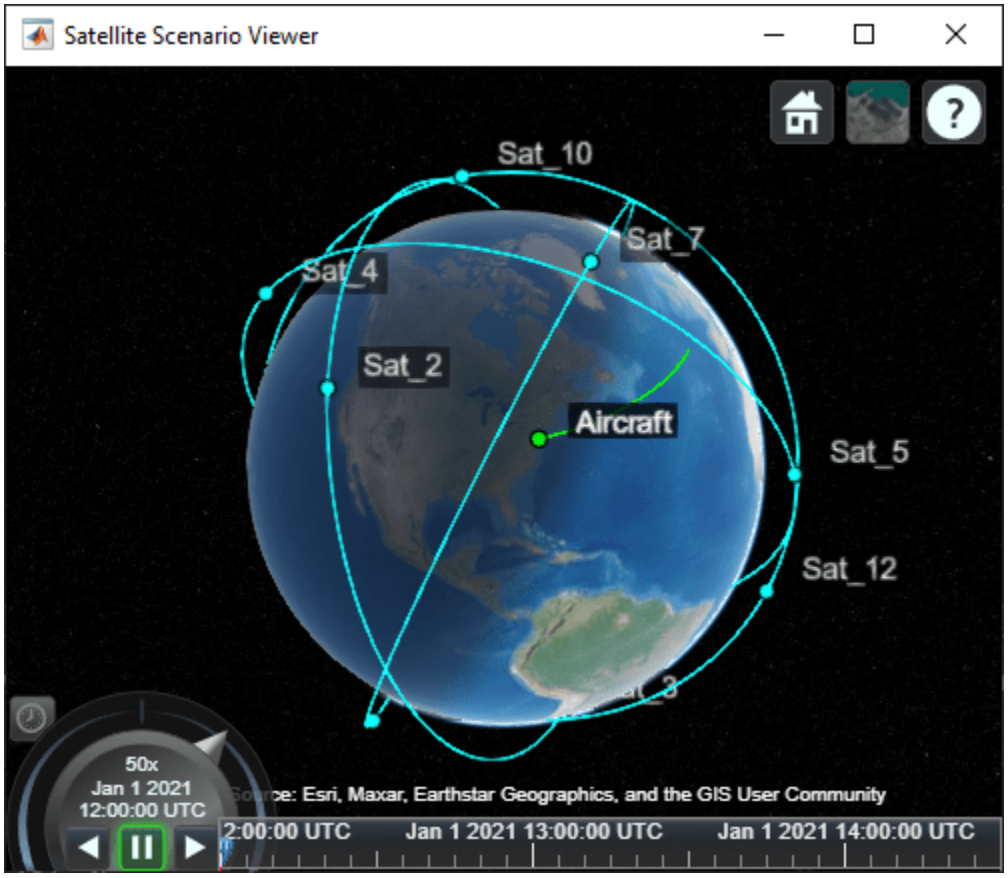
Use the Home button in the Satellite Scenario Viewer to return the viewer to an Earth-centric view.



Add a Walker-Delta Constellation to the Satellite Scenario

Add a Walker-Delta constellation to the scenario. The change in true anomaly for equivalent satellites in neighboring planes is calculated as Phasing*360/totalSatellites. Argument of Latitude defines the starting point for distribution of satellites along the first orbital track.

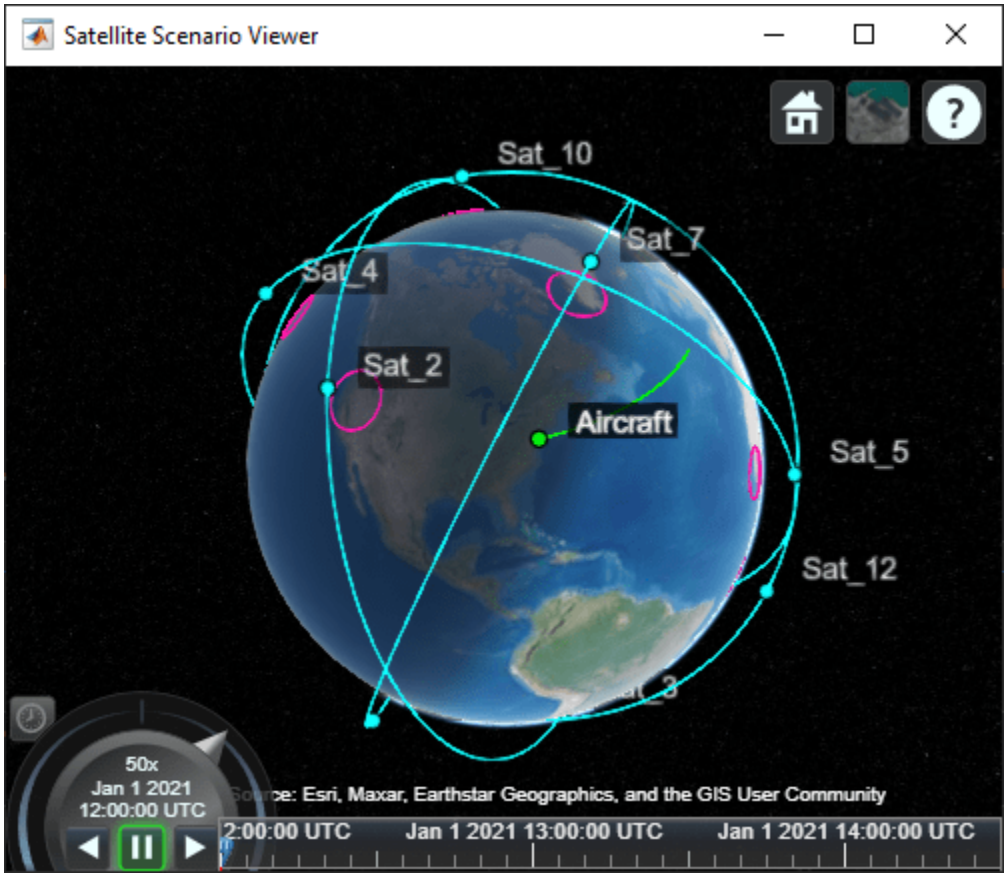
```
constellation.Radius = 7200000; % meters
constellation.Inclination = 70; % deg
constellation.TotalSatellites = 12;
constellation.GeometryPlanes = 4;
constellation.Phasing = 1;
constellation.ArgLat = 15; % deg

constellation.obj = walkerDelta(mission.scenario, ...
    constellation.Radius, ...
    constellation.Inclination, ...
    constellation.TotalSatellites, ...
    constellation.GeometryPlanes, ...
    constellation.Phasing, ...
    ArgumentOfLatitude=constellation.ArgLat, ...
    Name="Sat");
```



Add Sensors to the Constellation

Add a conical sensor to each satellite with a configurable half angle. Enable field of view visualization in the Satellite Scenario Viewer. To assist in visualization, the sensor is mounted 10m from the satellite, in the $+z$ direction.

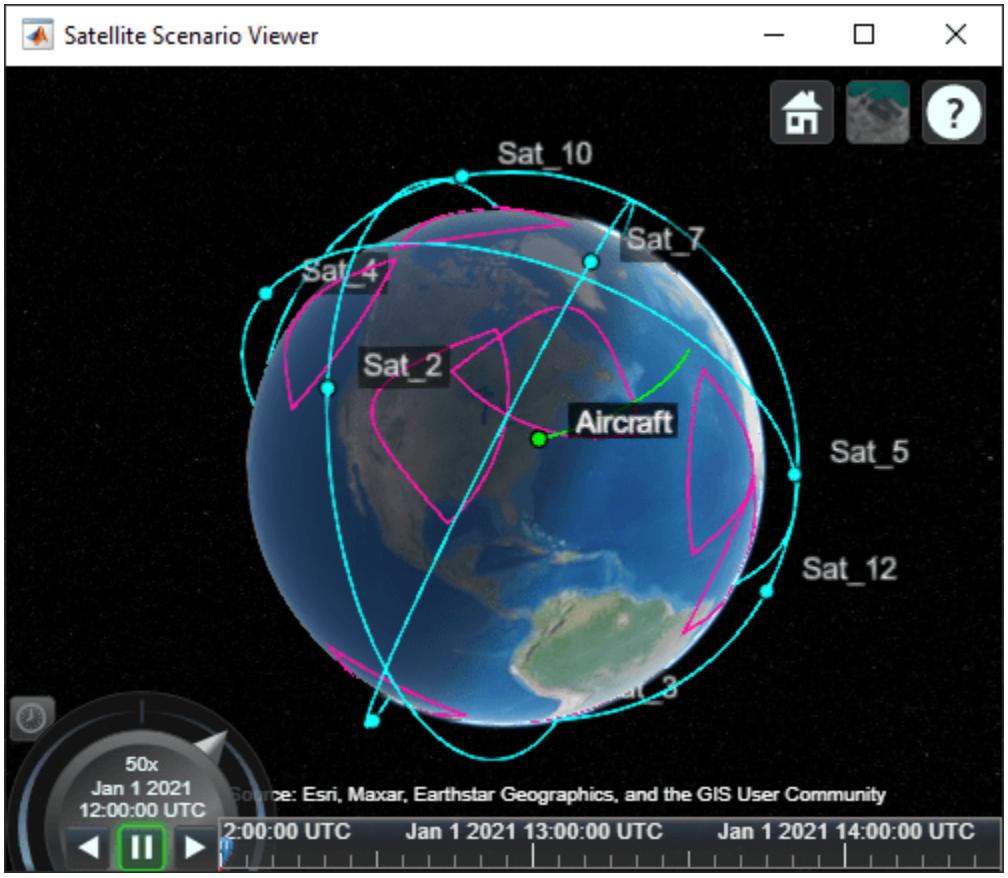
```
sensor.HalfAngle = 35; % deg
sensor.Names = constellation.obj.Name + " satellite";
sensor.obj = conicalSensor(constellation.obj, MaxViewAngle=sensor.HalfAngle*2, MountingLocation=
sensor.FOV.obj = fieldOfView(sensor.obj);
```



Configure the Constellation to Point at the Aircraft

Configure all the satellites in the constellation to point at the aircraft.

```
pointAt(constellation.obj,aircraft.obj);
```

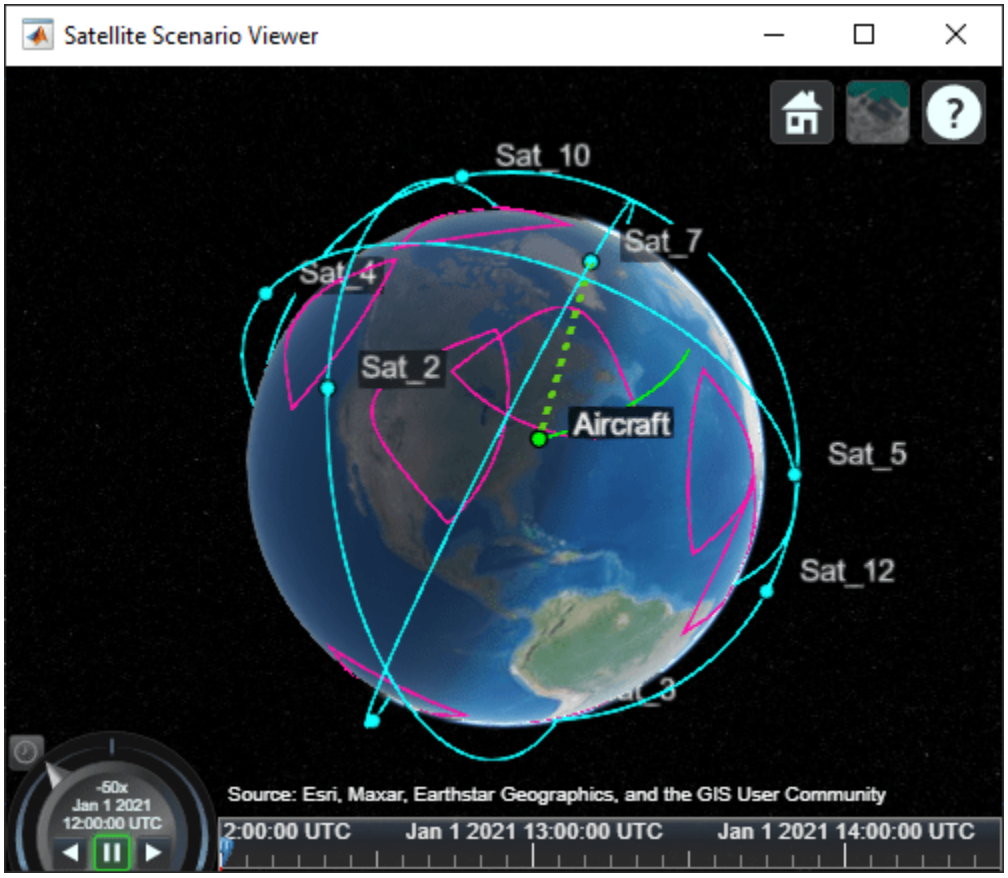


Determine the Satellite-to-Aircraft Access Intervals

Analyze the access intervals between the satellites in the constellation and the aircraft.

```
accessAnalysis.obj = access(sensor.obj, aircraft.obj);
accessAnalysis.Intervals = accessIntervals(accessAnalysis.obj);
accessAnalysis.Intervals = sortrows(accessAnalysis.Intervals,"StartTime");
disp(accessAnalysis.Intervals)
```

Source	Target	IntervalNumber	StartTime	EndTime
"Sat_4 satellite"	"Aircraft"	1	01-Jan-2021 12:14:00	01-Jan-2021 12
"Sat_9 satellite"	"Aircraft"	1	01-Jan-2021 12:19:00	01-Jan-2021 12
"Sat_6 satellite"	"Aircraft"	1	01-Jan-2021 12:48:00	01-Jan-2021 13
"Sat_8 satellite"	"Aircraft"	1	01-Jan-2021 12:55:00	01-Jan-2021 13
"Sat_5 satellite"	"Aircraft"	1	01-Jan-2021 13:23:00	01-Jan-2021 13
"Sat_7 satellite"	"Aircraft"	1	01-Jan-2021 13:32:00	01-Jan-2021 13
"Sat_4 satellite"	"Aircraft"	2	01-Jan-2021 13:57:00	01-Jan-2021 14
"Sat_12 satellite"	"Aircraft"	1	01-Jan-2021 13:57:00	01-Jan-2021 13
"Sat_9 satellite"	"Aircraft"	2	01-Jan-2021 14:09:00	01-Jan-2021 14
"Sat_11 satellite"	"Aircraft"	1	01-Jan-2021 14:26:00	01-Jan-2021 14



Calculate System-Wide Access Percentage

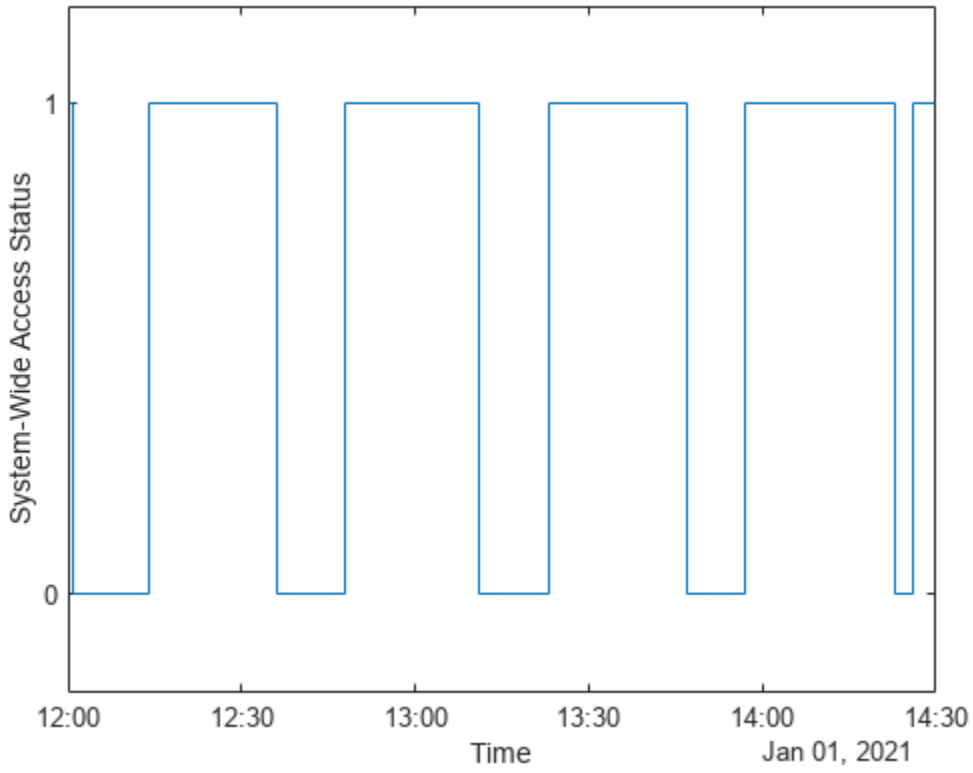
Determine the system-wide access percentage, which is the percentage of time from the scenario start time to stop time when at least one satellite can access the aircraft.

Calculate the access status between the aircraft and each individual satellite over time using `accessStatus`. Each row of the output array corresponds with a satellite in the constellation. Each column corresponds with time steps in the scenario. A value of `True` indicates that the satellite can access the aircraft at that specific time sample. The second output of `accessStatus` contains the timesteps of the scenario.

```
[accessAnalysis.Status, mission.TimeSteps] = accessStatus(accessAnalysis.obj);
```

Use `any` to perform a logical `OR` on all rows corresponding to access of each satellite to the aircraft. This results in a single row vector of logicals, in which a given element is true if at least one satellite can access the aircraft at the corresponding time step for a duration of one scenario sample time (60 seconds). Use `stairs` to plot system-wide access status with respect to time.

```
accessAnalysis.SystemWideAccessStatus = any(accessAnalysis.Status, 1);
stairs(mission.TimeSteps, accessAnalysis.SystemWideAccessStatus);
ylim([-.2, 1.2]);
xlabel("Time");
ylabel("System-Wide Access Status");
yticks([0,1]);
```



Count the number of elements in the vector whose value is **`True`**. Multiply this quantity by the sample time of 60 seconds to determine the total time in seconds when at least one satellite can access the aircraft.

```
accessAnalysis.SystemWideAccessDuration = sum(accessAnalysis.SystemWideAccessStatus) * mission.sc
```

Calculate the system-wide access percentage.

```
accessAnalysis.SystemWideAccessPercentage = (accessAnalysis.SystemWideAccessDuration/seconds(miss
disp("System Wide Access = " + accessAnalysis.SystemWideAccessPercentage + " %")
```

```
System Wide Access = 67.3333 %
```

Animate the Satellite Scenario

Configure the aircraft to display its ground track.

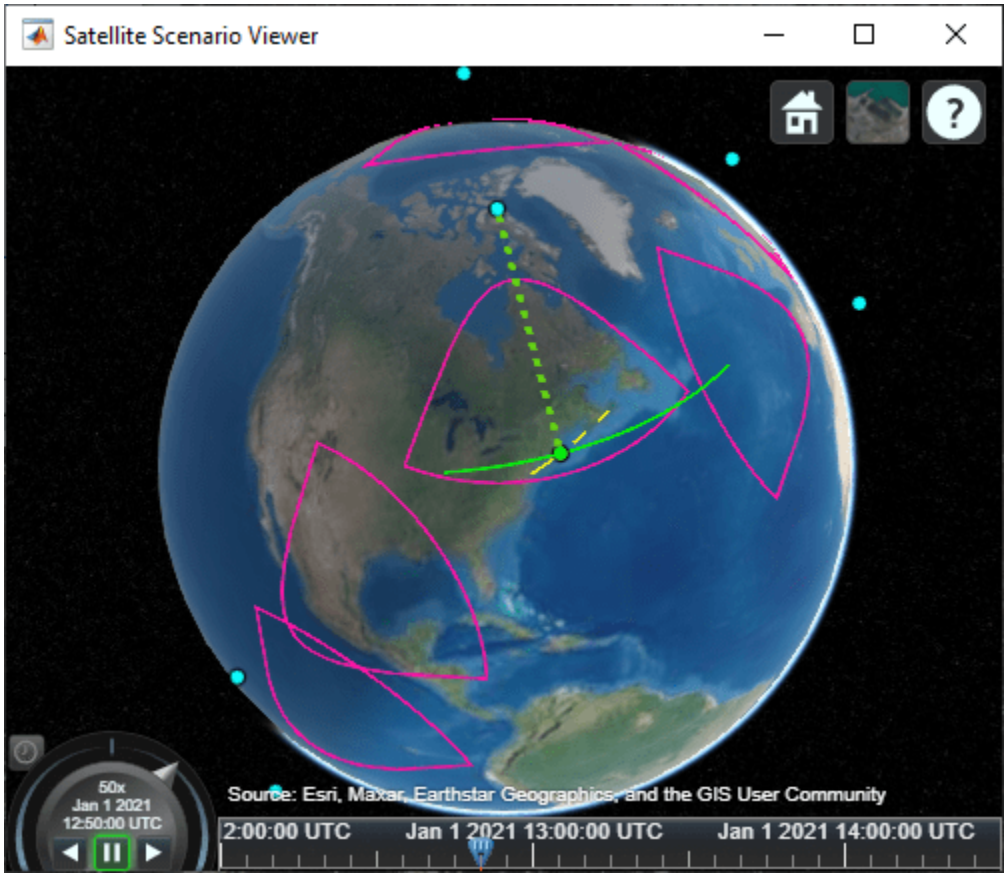
```
show(aircraft.obj.GroundTrack);
```

Configure the Satellite Scenario Viewer to hide orbits and labels.

```
hide(constellation.obj.Orbit);
constellation.obj.ShowLabel = false;
aircraft.obj.ShowLabel = false;
```

Play the scenario.

```
mission.viewer.PlaybackSpeedMultiplier = 200;
play(mission.scenario);
```



See Also

Objects
`satelliteScenario` | `satelliteScenarioViewer` | `Satellite` | `ConicalSensor` | `FieldOfView` | `Access`

Functions
`walkerDelta` | `pointAt` | `play` | `satellite` | `access` | `accessIntervals`

More About

- “Satellite Scenario Key Concepts” on page 2-65
- “Satellite Scenario Overview” on page 2-74

Satellite Conjunction Finder

This example shows how to find potential events of close proximity or conjuctions of two satellites.

To find potential upcoming conjuctions, the example uses the SOCRATES tool, available from the CelesTrak™ web site. SOCRATES generates lists of upcoming conjuctions using two-line elements (TLEs) that are stored in a text file for easy loading into the satellite scenario object. The conjuction lists generated by SOCRATES are used with permission from CelesTrak.

As a general workflow:

1. To find windows when the two satellites pass within a specified control distance, such as 200 km, the example uses the simulation capability within the `satelliteScenario` object . This search requires a sample time along the orbit of about 20 seconds.
2. Use `fzero` with a gradient function to find the time of closest approach for each window.
3. Visualize the output and confirm the same conjunction as SOCRATES has been located.

Disclaimer - Do not use publicly available TLEs for operational conjunction assessment prediction. This example is provided for informational purposes only. Satellite operators should contact the 18th Space Defense Squadron for data and analysis to support operational satellites.

Choose a Conjunction Scenario

The main page of SOCRATES provides two conjuction lists, the top 10 conjunctions by either maximum probability or maximum range. Clicking on a list shows a table of the conjunctions with the satellite names, date and time of closest approach (TCA), maximum probability of collision, and relative velocity. Each entry in the table has a button to display the TLEs in a separate window. You can copy these TLEs into a text file. An example of a copied conjunction follows.

```
type Conjunction_Starlink1079_CZ2DDeb.txt
```

```
STARLINK-1079
1 44937U 20001Z   22272.52136386 -.00000745  00000+0 -31120-4 0  9997
2 44937  53.0548 210.2777 0001443  83.0631 277.0522 15.06393387150649
CZ-2D DEB
1 43406U 12073D   22272.74034628  .00008296  00000+0  83006-3 0  9993
2 43406  97.9471 196.5364 0090895 225.9886 133.3815 14.88613293478469
```

Conjunction data from SOCRATES:

- `44937 STARLINK-1079 [+], Days since epoch: 3.817`
- `43406 CZ-2D DEB [-], Days since Epoch: 3.598`
- `Max Probability 1.363E-02`
- `Dilution Threshold (km) 0.004`
- `Min Range (km) 0.017`
- `Relative Velocity (km/sec) 5.861`
- `Start (UTC) 2022 Oct 03 08:06:48.977`
- `TCA (UTC) 2022 Oct 03 08:06:49.830`
- `Stop (UTC) 2022 Oct 03 08:06:50.683`

Parameters for the conjunction list are:

- Data current as of 2022 Sep 30 08:05 UTC
- Computation Interval: Start = 2022 Sep 30 00:00:00.000, Stop = 2022 Oct 07 00:00:00.000
- Computation Threshold: 5.0 km
- Considering: 9,400 Primaries, 23,748 Secondaries (115,200 Conjunctions)

This example uses `satelliteScenario` to validate the time and distance of closest approach as reported by `SOCRATES`. For this particular conjunction example, notice that the satellites have two close passes in seven days.

This example provides two StarLink satellite TLE text files and one ORSTED satellite and Iridrium33 satellite TLE text file retrieved from `SOCRATES`. The two StarLink satellite TLE text files produce over 200 coarse conjunction windows, as the satellites pass close to each other every half orbit. This analysis takes some time to run with a closest approach of 10 m found. The ORSTED and Iridium 33 TLE text file produces nearly 20 coarse windows, with a closest distance of 26 m.

From the list, choose a TLE text file to run.

```
tleFile = Starlink1079 & CZ2D
```

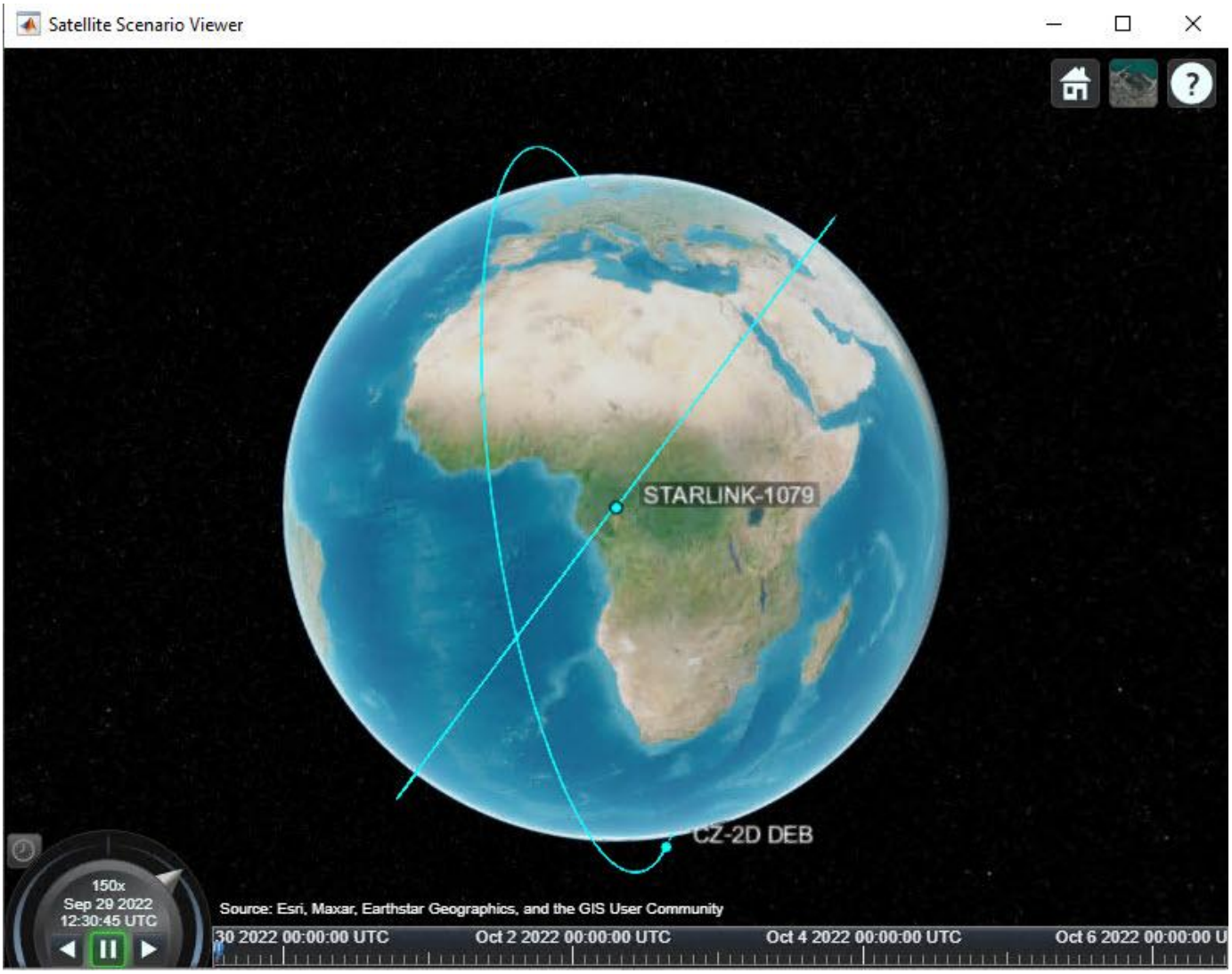
```
tleFile =
"Conjunction_Starlink1079_CZ2DDeb.txt"
```

Load the Satellite Elements into a Scenario Object

Load the TLEs directly into the `satelliteScenario` using a plain text file. TLEs are a text representation of the satellite orbital elements for use with the SGP4 or SDP4 propagators.

```
sc  = satelliteScenario;
satellites = satellite(sc,tleFile);
v = satelliteScenarioViewer(sc);
v.PlaybackSpeedMultiplier = 150;
```

Here is a screenshot of the viewer for the first example. The controls are on the bottom left. We can see that the orbits cross at a similar altitude.



Find Windows When the Satellites Pass Within a Specified Distance

Measure the time it takes to find all the windows with seven days, the standard time period to check for conjunctions according to the NASA handbook [1]. A sample time of 20 seconds is sufficient for LEO satellites.

```
startTime_init = sc.StartTime;
sc.StopTime  = startTime_init + days(7);
sc.SampleTime = 20;
```

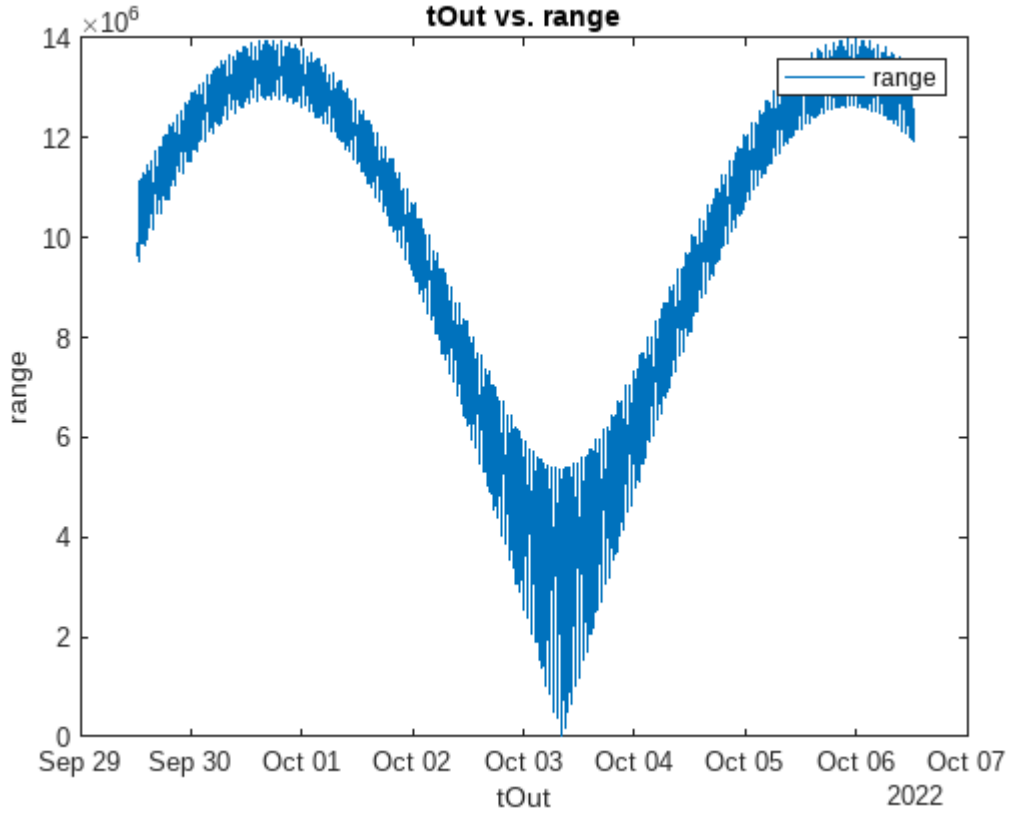
Get the satellite position data and associated times using the `aer` function. This method of the Satellite class computes azimuth angle, elevation angle, and the range of a satellite and another object. In this case, the other object is the other satellite.

```
[~,~,range,tOut] = aer(satellites(1),satellites(2));
```

Then, plot the range verses tOut.

```
% Create plot of tOut and range
h2 = plot(tOut,range,"DisplayName","range");
```

```
% Add xlabel, ylabel, title, and legend
xlabel("tOut")
ylabel("range")
title("tOut vs. range")
legend
```



Compute the distance between the two satellite positions. Determine the indices of the range when the satellite are closer than the target of 200 km. Then, use a `for` loop to find the times when this relative distance is less than a target of 200 km. The `diff` function finds the gaps in between the found points. If the value is 1, the points are adjacent to the window and the stop time can be extended. To create a window of time, add the ephemeric point before and after the found points.

```
dMinTarget = 200e3;
kCloseIdx  = find(range<dMinTarget);
dW         = [0 diff(kCloseIdx)];
kWindow    = zeros(2,length(kCloseIdx));
j          = 1;
for m = 1:length(dW)
  if dW(m)~=1
    % single point window
    kWindow(1,j) = max(1,kCloseIdx(m)-1);
    kWindow(2,j) = kCloseIdx(m)+1;
    j = j+1;
  elseif j>1
    % update window end as long as diff==1
    kWindow(2,j-1) = kCloseIdx(m)+1;
```

```
  end
end
```

Since any found multiple-point windows reduces the number of windows below the number of close points, limit the output to the windows found. The resulting number of found windows is printed to the command line.

```
kWindow = kWindow(:,1:j-1);
fprintf("%d windows found for dMin = %g km\n",size(kWindow,2),dMinTarget*1e-3);
```

```
2 windows found for dMin = 200 km
```

To get the datetime of the windows from output time array, use the window indices. Display windows to check the results. The windows are about a minute long.

```
windows = tOut(kWindow);
if size(windows,2)>=3
  disp(windows(:,1:3)')
else
  disp(windows')
end
```

```
   03-Oct-2022 08:06:05   03-Oct-2022 08:07:25
   03-Oct-2022 08:54:25   03-Oct-2022 08:55:05
```

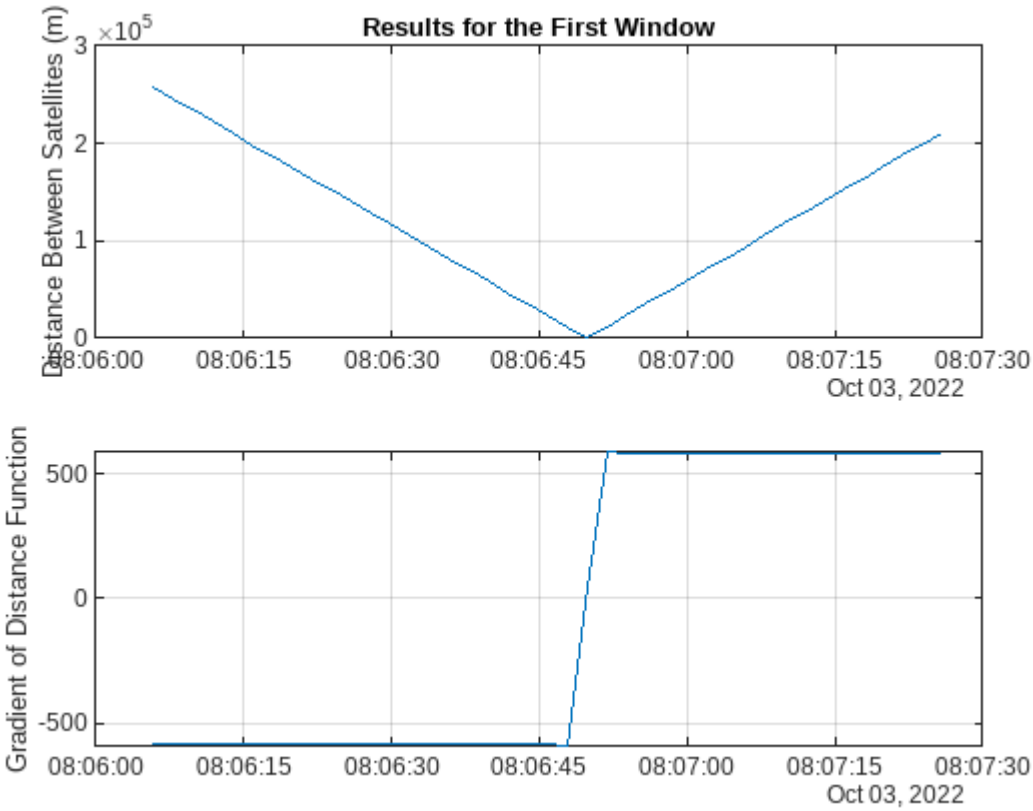
Test the Gradient Function Over One Window

To find the point of closest approach, look for the cost to change sign over the window. Using a `for` loop, call the gradient function for an array of times within the first window using a timestep of 2 seconds. The `gradientScenario` function is in a separate file. At each time step, compute a duration in days from the scenario start time. Store the gradient and distance between the satellites.

```
window = windows(:,1);
tVec   = window(1):seconds(2):window(2);
dRDot  = zeros(size(tVec));
dR     = zeros(size(tVec));
for k = 1:length(tVec)
  delT = days(tVec(k) - sc.StartTime);
  [dRDot(k),dR(k)] = gradientScenario(delT,satellites(1),satellites(2),sc.StartTime);
end
```

Use `figure` and `subplot` to create a plot of both the minimum distance and the gradient function output. Confirm the sign change at the point of closest approach.

```
figure('name','Gradient Test over One Window')
subplot(211)
plot(tVec,dR)
grid on
ylabel('Distance Between Satellites (m)')
title('Results for the First Window')
subplot(212)
plot(tVec,dRDot)
grid on
ylabel('Gradient of Distance Function')
```



Search the Windows for the Closest Approach

Pass in the start and stop time of the window to give `fzero` a range. The `days` function converts the datetime array to a double. Use an anonymous function to pass the `satelliteScenario` object to the gradient function. To measure the elapsed time of the `for` loop, use `tic` and `toc`. For some satellite pairs, the elapsed time might exceed a minute.

```
tMin = zeros(1,size(windows,2));
dRs  = zeros(1,size(windows,2));
h = waitbar(0,'Please wait, iterating over windows...');
tic
for k = 1:size(windows,2)
  tMin(k) = fzero(@(t) gradientScenario(t,satellites(1),satellites(2),sc.StartTime),days(windows
  [~,dRs(k)] = gradientScenario(tMin(k),satellites(1),satellites(2),sc.StartTime);
  waitbar(k/size(windows,2),h);
end
close(h)
toc
```

```
Elapsed time is 0.812581 seconds.
```

Find the minimum distance of all the windows and report the time of closest approach found. The distance is displayed in meters.

```
[rMin,iMin] = min(dRs);
tCA = sc.StartTime + tMin(iMin);
disp(tCA)
```

```
    03-Oct-2022 08:06:49
```

```
disp(dRs(iMin))
```

```
    17.1737
```

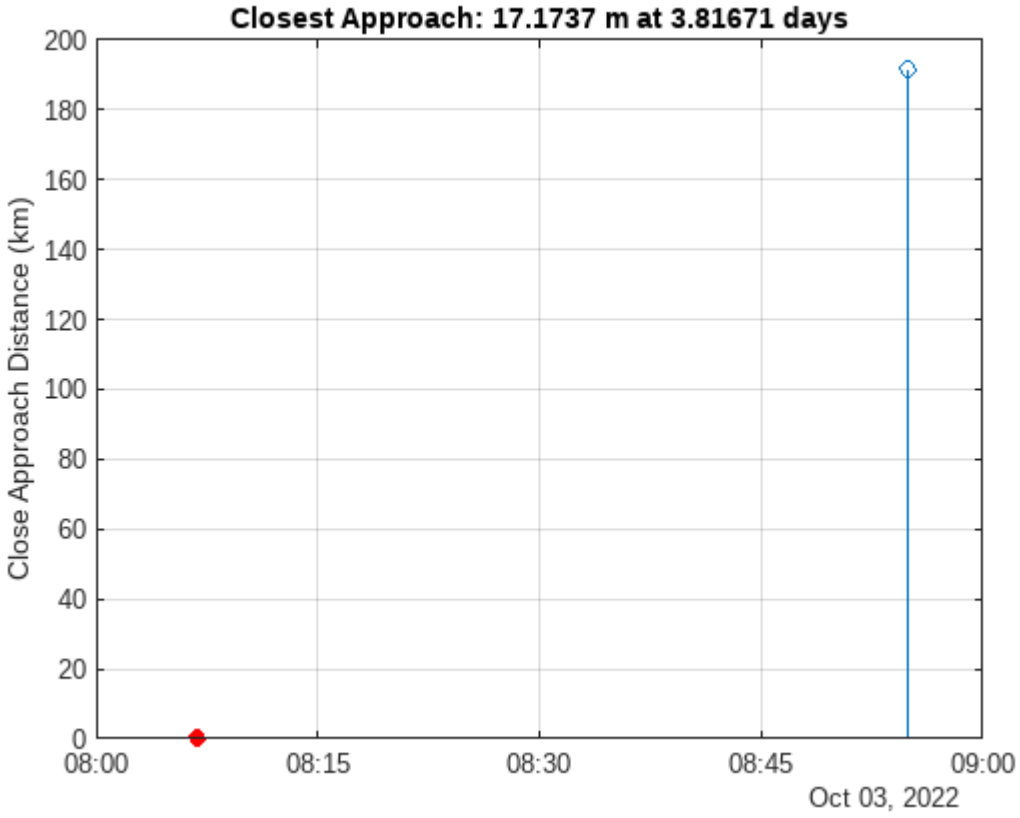
Verify the relative velocity between the satellites from SOCRATES using the `states` function and the time of closest approach. The velocity is in m/s.

```
[~,vel] = states(satellites,tCA); % satellite velocity in m/s
dV = vecnorm(vel(:,:,1) - vel(:,:,2));
disp(dV)
```

```
    5.8613e+03
```

To plot the resulting close passes, use the `stem` function. Mark the closest approach in red. Notice that the result is the same as that reported by SOCRATES.

```
figure('name','ConjunctionDetection')
stem(sc.StartTime + tMin,dRs*1e-3)
ylabel('Close Approach Distance (km)')
title(sprintf('Closest Approach: %g m at %g days',rMin,tMin(iMin)))
hold on; grid on;
stem(sc.StartTime + tMin(iMin),rMin*1e-3,'filled','r') % mark the closest approach
```



In summary, it is a two-step process to find the closest approach of one satellite to another during a seven day time window.

1. Find windows of close approach using a coarse filter with distance metric.
2. Find the closest pass within each window using `fzero`.

This two-step method can replicate the conjunctions reported by the SOCRATES website:

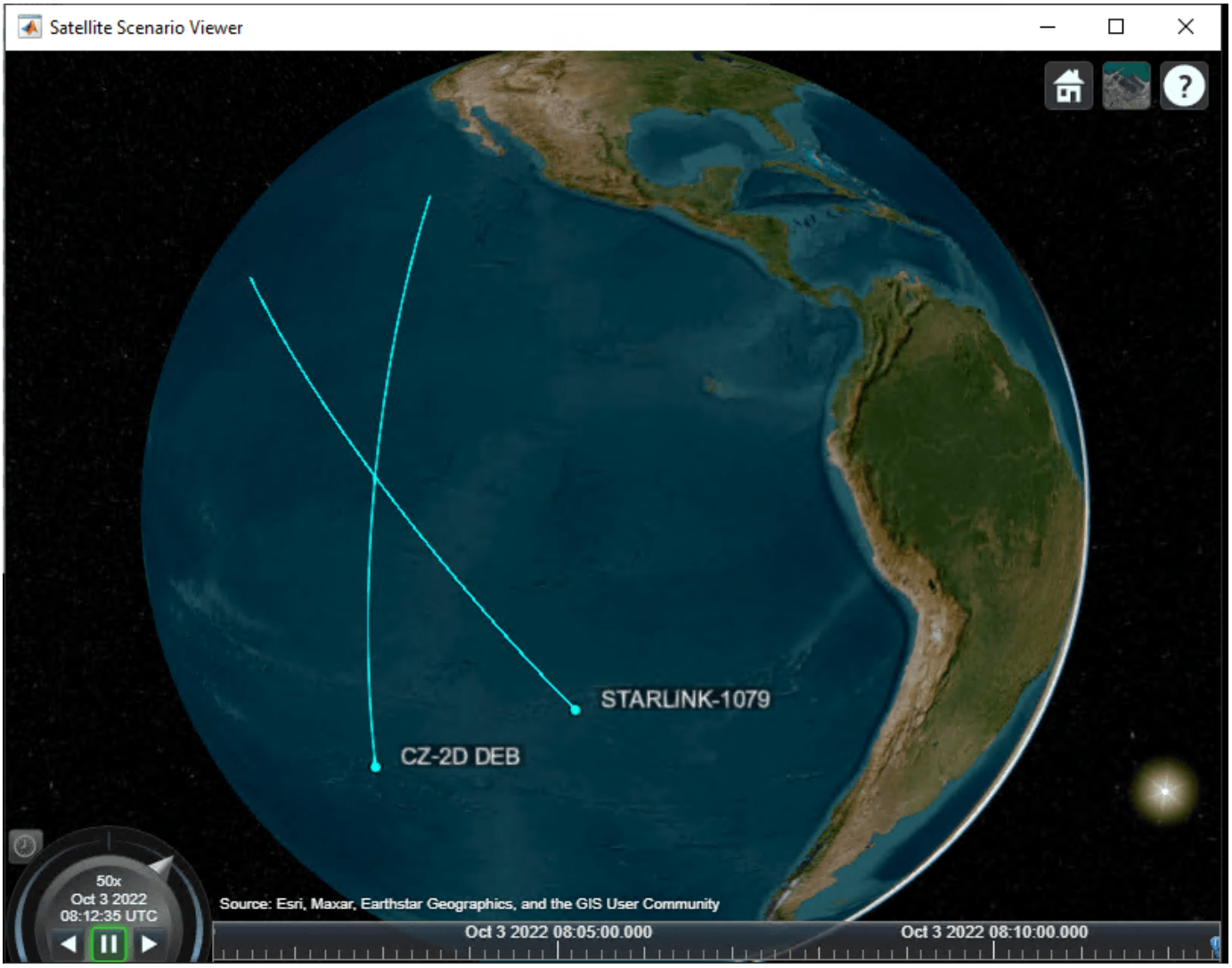
- Time of closest approach (TCA)
- Distance between the satellites
- Relative velocity

Update the Scenario Around the Found Conjunction

Now that we have the time of closest approach, update the simulation start and stop times to be a few minutes more or less. Use the `play` function to run the simulation.

Here is a screenshot of the conjunction from our first example.

```
sc.StartTime = tCA - days(0.004);
sc.StopTime = tCA + days(0.004);
v.PlaybackSpeedMultiplier = 50;
campos(v,-10.5641,-98.9574,9.1906e6);
play(sc);
```



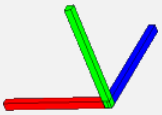
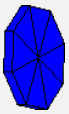
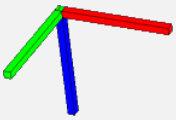
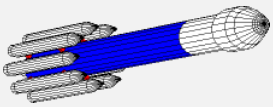
References

[1] NASA Spacecraft Conjunction Assessment and Collision Avoidance Best Practices Handbook, December 2020, NASA/SP-20205011318, **https://go.nasa.gov/34f9ijM**

AC3D Files and Thumbnails

AC3D Files and Thumbnails Overview

Aerospace Toolbox demos use the following AC3D files, located in the *matlabroot*`\toolbox\aero\animation\AC3D` 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`
	`redwedge.ac`
	`testrocket.ac`