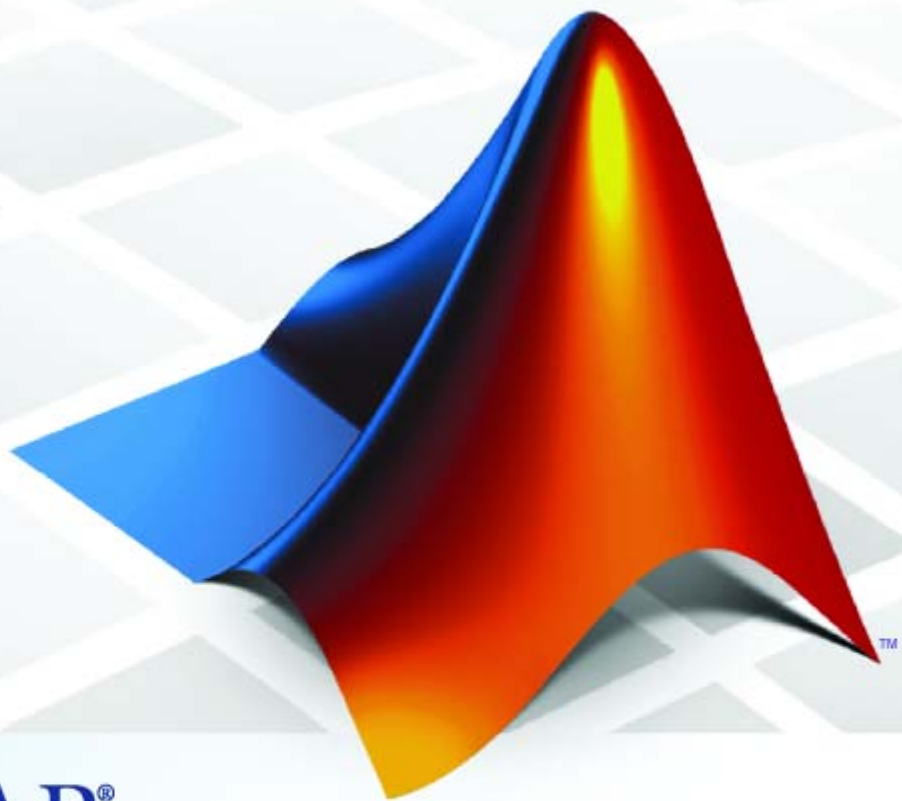


# Aerospace Toolbox 2

## User's Guide



MATLAB®

## How to Contact The MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

*Aerospace Toolbox User's Guide*

© COPYRIGHT 2006–2009 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](http://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

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

**Revision History**

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)



## Getting Started

### 1

<b>Product Overview</b> .....	1-2
<b>Related Products</b> .....	1-4
<b>Getting Online Help</b> .....	1-5
Exploring the Toolbox .....	1-5
Using the MATLAB Help System for Documentation and Demos .....	1-5

## Using Aerospace Toolbox

### 2

<b>Defining Coordinate Systems</b> .....	2-2
Fundamental Coordinate System Concepts .....	2-2
Coordinate Systems for Modeling .....	2-4
Coordinate Systems for Navigation .....	2-7
Coordinate Systems for Display .....	2-10
References .....	2-11
<b>Defining Aerospace Units</b> .....	2-12
<b>Importing Digital DATCOM Data</b> .....	2-14
Overview .....	2-14
Example of a USAF Digital DATCOM File .....	2-14
Importing Data from DATCOM Files .....	2-15
Examining Imported DATCOM Data .....	2-15
Filling in Missing DATCOM Data .....	2-17
Plotting Aerodynamic Coefficients .....	2-22
<b>3-D Flight Data Playback</b> .....	2-26

Aerospace Toolbox Animation Objects .....	2-26
Using Aero.Animation Objects .....	2-26
Using Aero.VirtualRealityAnimation Objects .....	2-35
Using Aero.FlightGearAnimation Object .....	2-48

## Function Reference

### 3

Animation Objects .....	3-3
Body Objects .....	3-4
Camera Objects .....	3-5
FlightGear Objects .....	3-5
Geometry Objects .....	3-6
Node Objects .....	3-7
Viewpoint Objects .....	3-8
Virtual Reality Objects .....	3-9
Axes Transformations .....	3-10
Environment .....	3-11
File Reading .....	3-11
Flight Parameters .....	3-12
Quaternion Math .....	3-12
Time .....	3-13

Unit Conversion ..... 3-13

**Alphabetical List**

**4**

**Class Reference**

**5**

**AC3D Files and Thumbnails**

**A**

Overview ..... A-2

**Index**





# Getting Started

---

- “Product Overview” on page 1-2
- “Related Products” on page 1-4
- “Getting Online Help” on page 1-5

## Product Overview

The Aerospace Toolbox product extends the MATLAB® technical computing environment by providing reference standards, environment models, and aerodynamic coefficient importing for performing advanced aerospace analysis to develop and evaluate your designs. The toolbox provides the following to enable you to visualize flight data in a three-dimensional environment and reconstruct behavioral anomalies in flight-test results:

- Aero.Animation, Aero.Body, Aero.Camera, and Aero.Geometry objects and associated methods
- An interface to the FlightGear flight simulator
- An interface to the Simulink® 3D Animation™ software

To ensure design consistency, the Aerospace Toolbox software provides utilities for unit conversions, coordinate transformations, and quaternion math, as well as standards-based environmental models for the atmosphere, gravity, and magnetic fields. You can import aerodynamic coefficients directly from the U.S. Air Force Digital Data Compendium (DATCOM) to carry out preliminary control design and vehicle performance analysis.

The toolbox provides you with the following main features:

- Provides standards-based environmental models for atmosphere, gravity, and magnetic fields.
- Converts units and transforms coordinate systems and spatial representations.
- Implements predefined utilities for aerospace parameter calculations, time calculations, and quaternion math.
- Imports aerodynamic coefficients directly from DATCOM.
- Interfaces to the FlightGear flight simulator, enabling visualization of vehicle dynamics in a three-dimensional environment.

The Aerospace Toolbox functions can be used in applications such as aircraft technology, telemetry data reduction, flight control analysis, navigation analysis, visualization for flight simulation, and environmental modeling, and can help you perform the following tasks:

- Analyze, initialize, and visualize a broad range of large aerospace system architectures, including aircraft, missiles, spacecraft (probes, satellites, manned and unmanned), and propulsion systems (engines and rockets), while reducing development time.
- Support and define new requirements for aerospace systems.
- Perform complex calculations and analyze data to optimize and implement your designs.
- Test the performance of flight tests.

The Aerospace Toolbox software maintains and updates the algorithms, tables, and standard environmental models, eliminating the need to provide internal maintenance and verification of the models and reducing the cost of internal software maintenance.

## Related Products

The Aerospace Toolbox software requires the MATLAB software.

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

Other related products are listed in the Aerospace Toolbox product page at the MathWorks Web site. They include toolboxes and blocksets that extend the capabilities of the MATLAB and Simulink products. These products will enhance your use of the toolbox in various applications.

For more information about any MathWorks™ software products, see either

- The online documentation for that product if it is installed
- The MathWorks Web site at [www.mathworks.com](http://www.mathworks.com)

## Getting Online Help

In this section...
“Exploring the Toolbox” on page 1-5
“Using the MATLAB Help System for Documentation and Demos” on page 1-5

### Exploring the Toolbox

A list of the toolbox functions is available to you by typing

```
help aero
```

You can view the code for any function by typing

```
type function_name
```

### Using the MATLAB Help System for Documentation and Demos

The MATLAB Help browser allows you to access the documentation and demo models for all the MATLAB and Simulink based products that you have installed. The online Help includes an online index and search system.

Consult the Help for Using MATLAB section of the MATLAB Desktop Tools and Development Environment documentation for more information about the MATLAB Help system.



# Using Aerospace Toolbox

---

- “Defining Coordinate Systems” on page 2-2
- “Defining Aerospace Units” on page 2-12
- “Importing Digital DATCOM Data” on page 2-14
- “3-D Flight Data Playback” on page 2-26

## Defining Coordinate Systems

In this section...
“Fundamental Coordinate System Concepts” on page 2-2
“Coordinate Systems for Modeling” on page 2-4
“Coordinate Systems for Navigation” on page 2-7
“Coordinate Systems for Display” on page 2-10
“References” on page 2-11

### Fundamental Coordinate System Concepts

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

#### Definitions

The Aerospace Toolbox software uses *right-handed* (RH) Cartesian coordinate systems. The *right-hand 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’s 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’s surface. The geographic poles are the intersection of the Earth’s surface and the geopolar axis. In general, the Earth’s 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's 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's 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's rotation about its axis. This is a *rotating, geocentric* system. The toolbox ignores the Earth's motion around the Sun, the Sun's motion in the Galaxy, and the Galaxy's motion through cosmos. In most applications, only the Earth's rotation matters.

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

## Motion with Respect to Other Planets

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

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

### Coordinate Systems for Modeling

Modeling aircraft and spacecraft is 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.

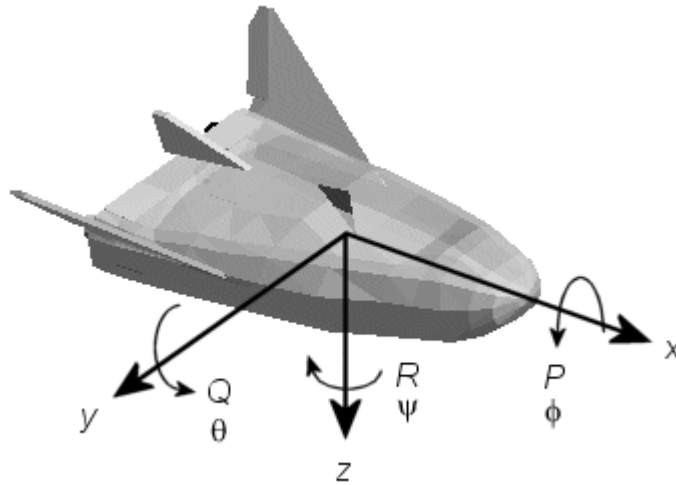
The orientation of the body coordinate axes is fixed in the shape of body.

- 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  $\Phi$ ,  $\Theta$ ,  $\Psi$ . They are

- $P$  or  $\Phi$ : Roll about the  $x$ -axis
- $Q$  or  $\Theta$ : Pitch about the  $y$ -axis
- $R$  or  $\Psi$ : Yaw about the  $z$ -axis



### 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  $V$ .

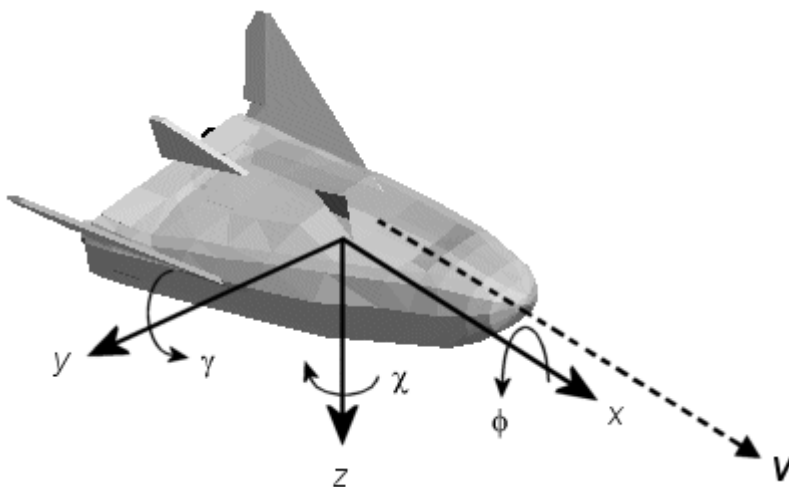
The orientation of the wind coordinate axes is fixed by the velocity  $V$ .

- 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  $\Phi$ ,  $\gamma$ ,  $\chi$ . They are

- $\Phi$ : Bank angle about the  $x$ -axis
- $\gamma$ : Flight path about the  $y$ -axis
- $\chi$ : Heading angle about the  $z$ -axis



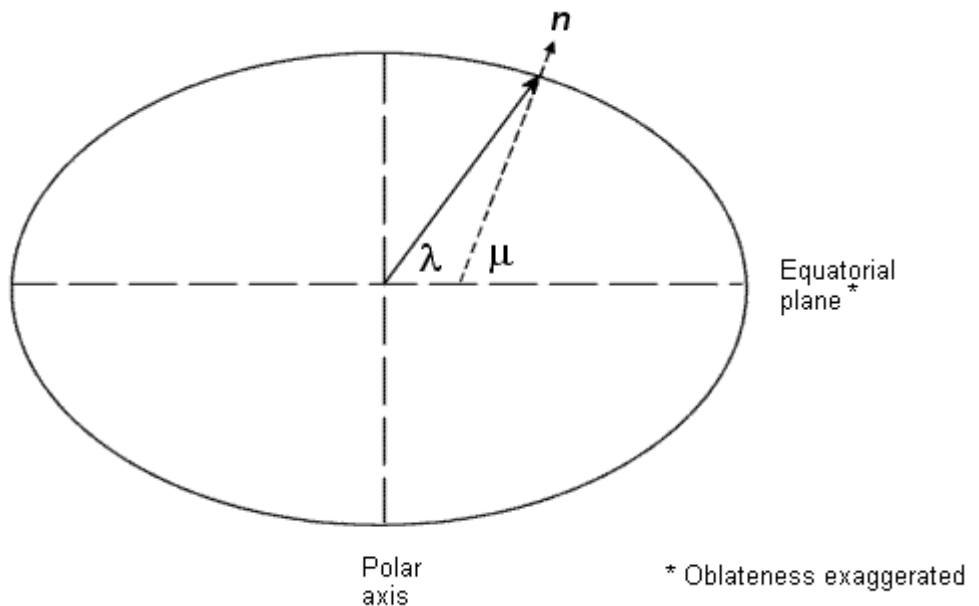
## Coordinate Systems for Navigation

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*  $\lambda$  on the Earth's surface is defined by the angle subtended by the radius vector from the Earth's center to the surface point with the equatorial plane.

The *geodetic latitude*  $\mu$  on the Earth's surface is defined by the angle subtended by the surface normal vector  $n$  and the equatorial plane.

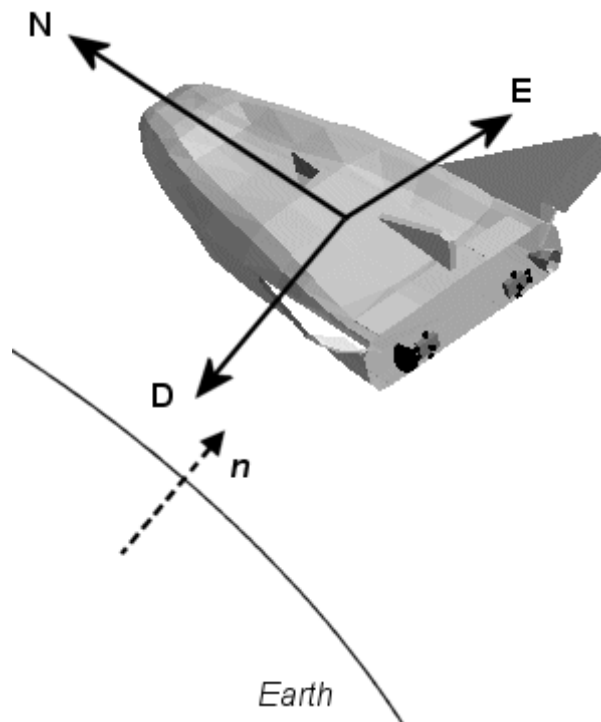


### NED Coordinates

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's 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's surface, antiparallel to the surface's outward normal  $n$ .

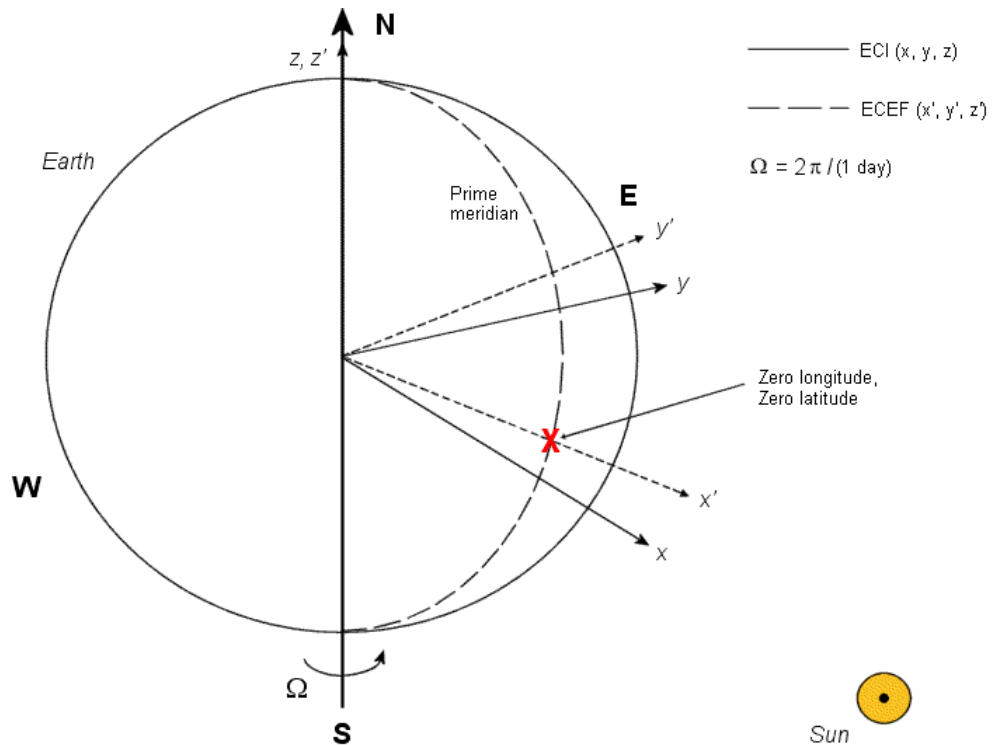
Flying at a constant altitude means flying at a constant  $z$  above the Earth's surface.



## ECI Coordinates

The Earth-centered inertial (ECI) system is a mixed inertial system. It is oriented with respect to the Sun. 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 in the Earth's equatorial plane exactly at the Sun. (This rule ignores the Sun's oblique angle to the equator, which varies with season. The actual Sun always remains in the  $x$ - $z$  plane.)
- The  $y$ -axis points into the eastward quadrant, perpendicular to the  $x$ - $z$  plane so as to satisfy the RH rule.



## Earth-Centered Coordinates

### **ECEF Coordinates**

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

- The  $z$ -axis points northward along the Earth's rotation axis.
- The  $x$ -axis points outward along the intersection of the Earth's equatorial plane and prime meridian.
- The  $y$ -axis points into the eastward quadrant, perpendicular to the  $x$ - $z$  plane so as to satisfy the RH rule.

### **Coordinate Systems for Display**

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

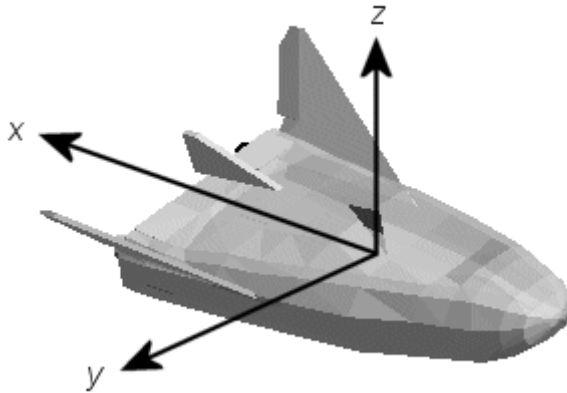
FlightGear is an open-source, third-party flight simulator with an interface supported by the Aerospace Toolbox product.

- “Working with the Flight Simulator Interface” on page 2-53 discusses the toolbox interface to FlightGear.
- See the FlightGear documentation at [www.flightgear.org](http://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.





## References

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

*Mapping Toolbox User's Guide*, The MathWorks, Inc., Natick, Massachusetts.  
[www.mathworks.com/access/helpdesk/help/toolbox/map/](http://www.mathworks.com/access/helpdesk/help/toolbox/map/).

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

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

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

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

## Defining 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 <sup>2</sup> (m/s <sup>2</sup> ), kilometers/second <sup>2</sup> (km/s <sup>2</sup> ), (kilometers/hour)/second (km/h-s), g-unit (g)	inches/second <sup>2</sup> (in/s <sup>2</sup> ), feet/second <sup>2</sup> (ft/s <sup>2</sup> ), (miles/hour)/second (mph/s), g-unit (g)
Angle	radian (rad), degree (deg), revolution	radian (rad), degree (deg), revolution
Angular acceleration	radians/second <sup>2</sup> (rad/s <sup>2</sup> ), degrees/second <sup>2</sup> (deg/s <sup>2</sup> ), revolutions/minute (rpm), revolutions/second (rps)	radians/second <sup>2</sup> (rad/s <sup>2</sup> ), degrees/second <sup>2</sup> (deg/s <sup>2</sup> ), revolutions/minute (rpm), revolutions/second (rps)
Angular velocity	radians/second (rad/s), degrees/second (deg/s), revolutions/minute (rpm)	radians/second (rad/s), degrees/second (deg/s), revolutions/minute (rpm)
Density	kilogram/meter <sup>3</sup> (kg/m <sup>3</sup> )	pound mass/foot <sup>3</sup> (lbm/ft <sup>3</sup> ), slug/foot <sup>3</sup> (slug/ft <sup>3</sup> ), pound mass/inch <sup>3</sup> (lbm/in <sup>3</sup> )
Force	newton (N)	pound (lb)
Inertia	kilogram-meter <sup>2</sup> (kg-m <sup>2</sup> )	slug-foot <sup>2</sup> (slug-ft <sup>2</sup> )
Length	meter (m)	inch (in), foot (ft), mile (mi), nautical mile (nm)

<b>Quantity</b>	<b>MKS (SI)</b>	<b>English</b>
Mass	kilogram (kg)	slug (slug), pound mass (lbm)
Pressure	pascal (Pa)	pound/inch <sup>2</sup> (psi), pound/foot <sup>2</sup> (psf), atmosphere (atm)
Temperature	kelvin (K), degrees Celsius (°C)	degrees Fahrenheit (°F), degrees Rankine (°R)
Torque	newton-meter (N-m)	pound-feet (lb-ft)
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

## Importing Digital DATCOM Data

### In this section...

“Overview” on page 2-14

“Example of a USAF Digital DATCOM File” on page 2-14

“Importing Data from DATCOM Files” on page 2-15

“Examining Imported DATCOM Data” on page 2-15

“Filling in Missing DATCOM Data” on page 2-17

“Plotting Aerodynamic Coefficients” on page 2-22

### Overview

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

The example used in the following topics is available as an Aerospace Toolbox demo. You can run the demo either by entering `astimportddatcom` in the MATLAB Command Window or by finding the demo entry (Importing from USAF Digital DATCOM Files) in the Demos browser and clicking **Run in the Command Window** on its demo page.

### Example of a USAF Digital DATCOM File

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

```
$FLTCON NMACH=2.0,MACH(1)=0.1,0.2$
$FLTCON NALT=2.0,ALT(1)=5000.0,8000.0$
$FLTCON NALPHA=5.,ALSCHD(1)=-2.0,0.0,2.0,
ALSCHD(4)=4.0,8.0,LOOP=2.0$
$OPTINS SREF=225.8,CBARR=5.75,BLREF=41.15$
```

```

$SYNTHS XCG=7.08,ZCG=0.0,XW=6.1,ZW=-1.4,ALIW=1.1,XH=20.2,
      ZH=0.4,ALIH=0.0,XV=21.3,ZV=0.0,VERTUP=.TRUE.$
$BODY NX=10.0,
      X(1)=-4.9,0.0,3.0,6.1,9.1,13.3,20.2,23.5,25.9,
      R(1)=0.0,1.0,1.75,2.6,2.6,2.6,2.0,1.0,0.0$
$WGPLNF CHRDT=4.0,SSPNE=18.7,SSPN=20.6,CHRDR=7.2,SAVSI=0.0,CHSTAT=0.25,
      TWISTA=-1.1,SSPNDD=0.0,DHDADI=3.0,DHDADO=3.0,TYPE=1.0$
NACA-W-6-64A412
$HTPLNF CHRDT=2.3,SSPNE=5.7,SSPN=6.625,CHRDR=0.25,SAVSI=11.0,
      CHSTAT=1.0,TWISTA=0.0,TYPE=1.0$
NACA-H-4-0012
$VTPLNF CHRDT=2.7,SSPNE=5.0,SSPN=5.2,CHRDR=5.3,SAVSI=31.3,
      CHSTAT=0.25,TWISTA=0.0,TYPE=1.0$
NACA-V-4-0012
CASEID SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG
DAMP
NEXT CASE

```

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

## Importing Data from DATCOM Files

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

```
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 =

    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]
c1a: [5x2x2 double]
cma: [5x2x2 double]
cyb: [5x2x2 double]
cnb: [5x2x2 double]
c1b: [5x2x2 double]
qqinf: [5x2x2 double]
eps: [5x2x2 double]
depsdalp: [5x2x2 double]
c1q: [5x2x2 double]
cmq: [5x2x2 double]
c1ad: [5x2x2 double]
cmad: [5x2x2 double]
c1p: [5x2x2 double]
cyp: [5x2x2 double]
cnp: [5x2x2 double]
cnr: [5x2x2 double]
c1r: [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(:, :, 1) =

1.0e+004 *

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

```

```
ans(:,:,2) =  
  
1.0e+004 *  
  
-0.0000 -0.0000  
9.9999 9.9999  
9.9999 9.9999  
9.9999 9.9999  
9.9999 9.9999
```

```
data.cnb  
ans(:,:,1) =  
  
1.0e+004 *  
  
0.0000 0.0000  
9.9999 9.9999  
9.9999 9.9999  
9.9999 9.9999  
9.9999 9.9999
```

```
ans(:,:,2) =  
  
1.0e+004 *  
  
0.0000 0.0000  
9.9999 9.9999  
9.9999 9.9999  
9.9999 9.9999  
9.9999 9.9999
```

```
data.clq  
ans(:,:,1) =  
  
1.0e+004 *  
  
0.0000 0.0000
```



```

9.9999  9.9999
9.9999  9.9999
9.9999  9.9999
9.9999  9.9999

```

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

```

1.0e+004 *

0.0000  0.0000
9.9999  9.9999
9.9999  9.9999
9.9999  9.9999
9.9999  9.9999

```

```
data.cmq
```

```
ans(:, :, 1) =
```

```

1.0e+004 *

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

```

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

```

1.0e+004 *

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

```

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

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

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

Here are the updated imported data values.

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

    -0.0035    -0.0035
    -0.0035    -0.0035
    -0.0035    -0.0035
    -0.0035    -0.0035
    -0.0035    -0.0035
```

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

    -0.0035    -0.0035
    -0.0035    -0.0035
    -0.0035    -0.0035
    -0.0035    -0.0035
    -0.0035    -0.0035
```

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

    1.0e-003 *

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

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

```
1.0e-003 *
```

```
0.9190 0.8829  
0.9190 0.8829  
0.9190 0.8829  
0.9190 0.8829  
0.9190 0.8829
```

```
data.clq
```

```
ans(:,:,1) =
```

```
0.0974 0.0984  
0.0974 0.0984  
0.0974 0.0984  
0.0974 0.0984  
0.0974 0.0984
```

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

```
0.0974 0.0984  
0.0974 0.0984  
0.0974 0.0984  
0.0974 0.0984  
0.0974 0.0984
```

```
data.cmq
```

```
ans(:,:,1) =
```

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

```
ans(:,:,2) =  
  
    -0.0892    -0.0899  
    -0.0892    -0.0899  
    -0.0892    -0.0899  
    -0.0892    -0.0899  
    -0.0892    -0.0899
```

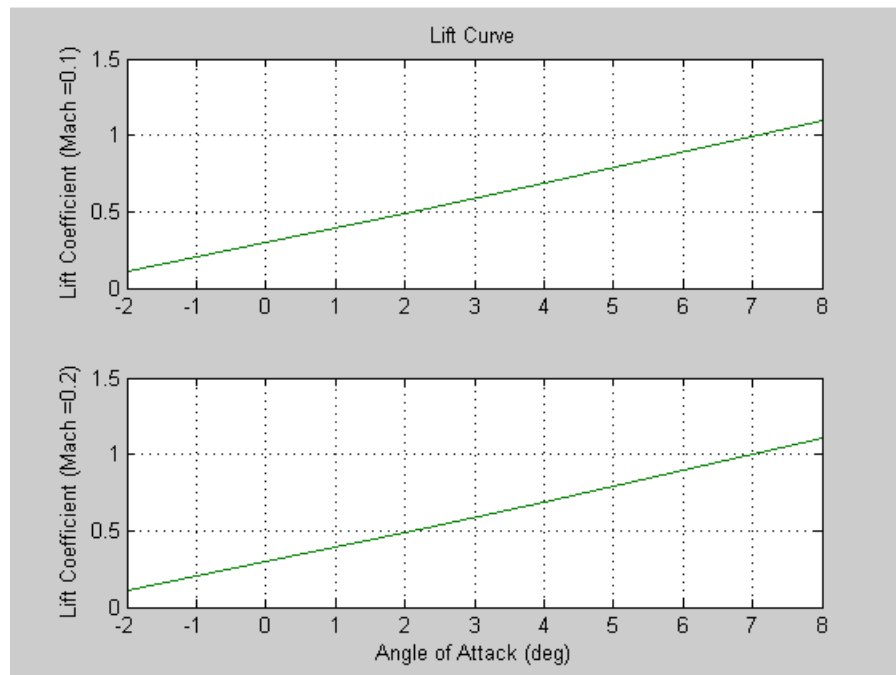
### Plotting Aerodynamic Coefficients

You can now plot the aerodynamic coefficients:

- “Plotting Lift Curve Moments” on page 2-22
- “Plotting Drag Polar Moments” on page 2-23
- “Plotting Pitching Moments” on page 2-24

### Plotting Lift Curve Moments

```
h1 = figure;  
figtitle = {'Lift Curve' ''};  
for k=1:2  
    subplot(2,1,k)  
    plot(data.alpha,permute(data.c1(:,k,:),[1 3 2]))  
    grid  
    ylabel(['Lift Coefficient (Mach = ' num2str(data.mach(k)) ' '])  
    title(figtitle{k});  
end  
xlabel('Angle of Attack (deg)')
```

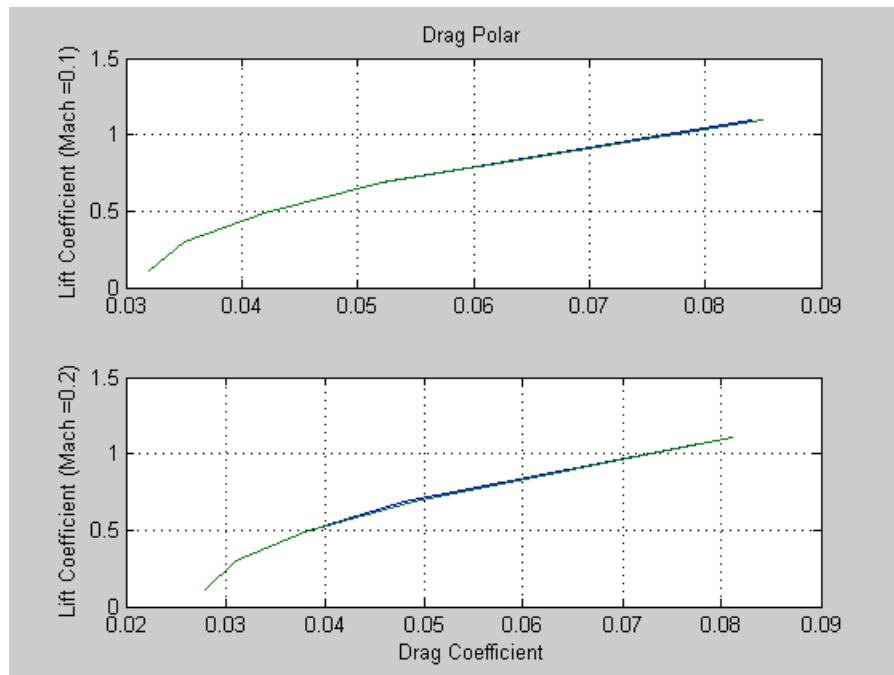


## Plotting Drag Polar Moments

```

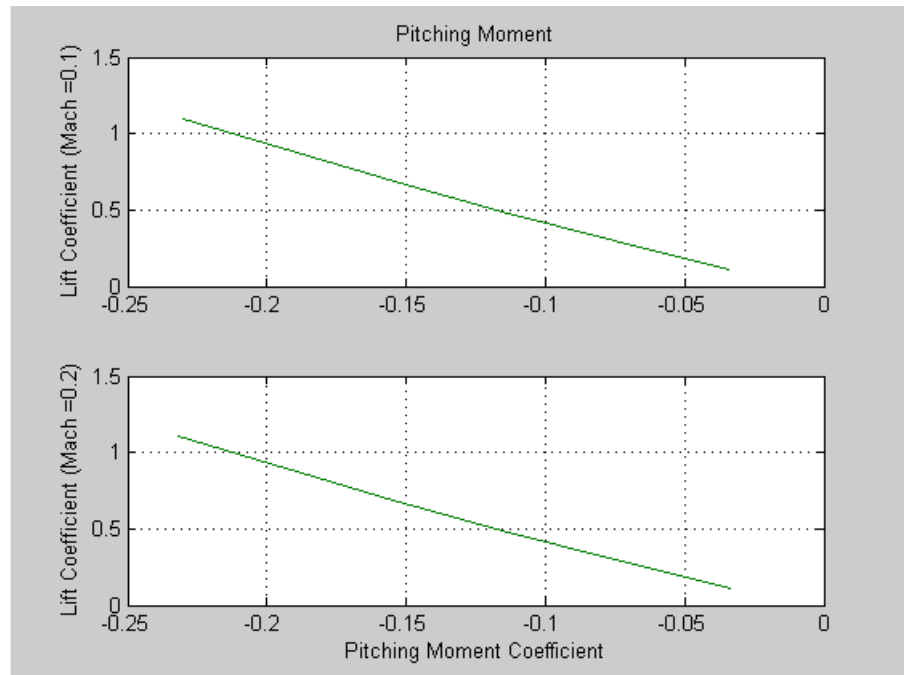
h2 = figure;
figtitle = {'Drag Polar' ''};
for k=1:2
    subplot(2,1,k)
    plot(permute(data.cd(:,k,:),[1 3 2]),permute(data.cl(:,k,:),[1 3 2]))
    grid
    ylabel(['Lift Coefficient (Mach = ' num2str(data.mach(k)) ' ')'])
    title(figtitle{k})
end
xlabel('Drag Coefficient')

```



### 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.c1(:,k,:),[1 3 2]))  
    grid  
    ylabel(['Lift Coefficient (Mach = ' num2str(data.mach(k)) ')'])  
    title(figtitle{k})  
end  
xlabel('Pitching Moment Coefficient')
```



## 3-D Flight Data Playback

### In this section...

“Aerospace Toolbox Animation Objects” on page 2-26

“Using Aero.Animation Objects” on page 2-26

“Using Aero.VirtualRealityAnimation Objects” on page 2-35

“Using Aero.FlightGearAnimation Object” on page 2-48

### 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` — You can use this animation object to visualize flight data without any other tool or toolbox. The following objects support this object.
  - `Aero.Body`
  - `Aero.Camera`
  - `Aero.Geometry`
- `Aero.VirtualRealityAnimation` — You can use this animation object to visualize flight data with the Simulink 3D Animation product. The following objects support this object.
  - `Aero.Node`
  - `Aero.Viewpoint`
- `Aero.FlightGearAnimation`

You can use this animation object to visualize flight data with the FlightGear simulator.

### Using Aero.Animation Objects

The toolbox interface to animation objects uses the Handle Graphics® product. The demo, `Overlaying Simulated and Actual Flight Data (astmlanim)`, visually



compares simulated and actual flight trajectory data. It does this by creating animation objects, creating bodies for those objects, and loading the flight trajectory data. This section describes what happens when the demo runs.

- 1 Create and configure an animation object.
  - a Configure the animation object.
  - b Create and load bodies for that object.
- 2 Load recorded data for flight trajectories.
- 3 Display body geometries in a figure window.
- 4 Play back flight trajectories using the animation object.
- 5 Manipulate the camera.
- 6 Manipulate bodies, as follows:
  - a Move and reposition bodies.
  - b Create a transparency in the first body.
  - c Change the color of the second body.
  - d Turn off the landing gear of the second body.

## Running the Demo

- 1 Start the MATLAB software.
- 2 Run the demo either by entering `astm1anim` in the MATLAB Command Window or by finding the demo entry (Overlaying Simulated and Actual Flight Data) in the Demos browser and clicking **Run in the Command Window** on its demo page.

While running, the demo performs several steps by issuing a series of commands, as explained below.

## Creating and Configuring an Animation Object

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

- 1 Create an animation object.

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

- 2 Configure the animation object to set the number of frames per second (FramesPerSecond) property. This 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. The settings in this demo result in a time step of approximately 0.5 s.

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

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

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

### Loading Recorded Data for Flight Trajectories

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

- `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 demo needs to

set the body object **TimeSeriesSourceType** parameter to **Custom**, 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 demo provides the custom read function in `matlabroot\toolbox\aero\astdemos\CustomReadBodyTSDData.m`.

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

## Displaying Body Geometries in a Figure Window

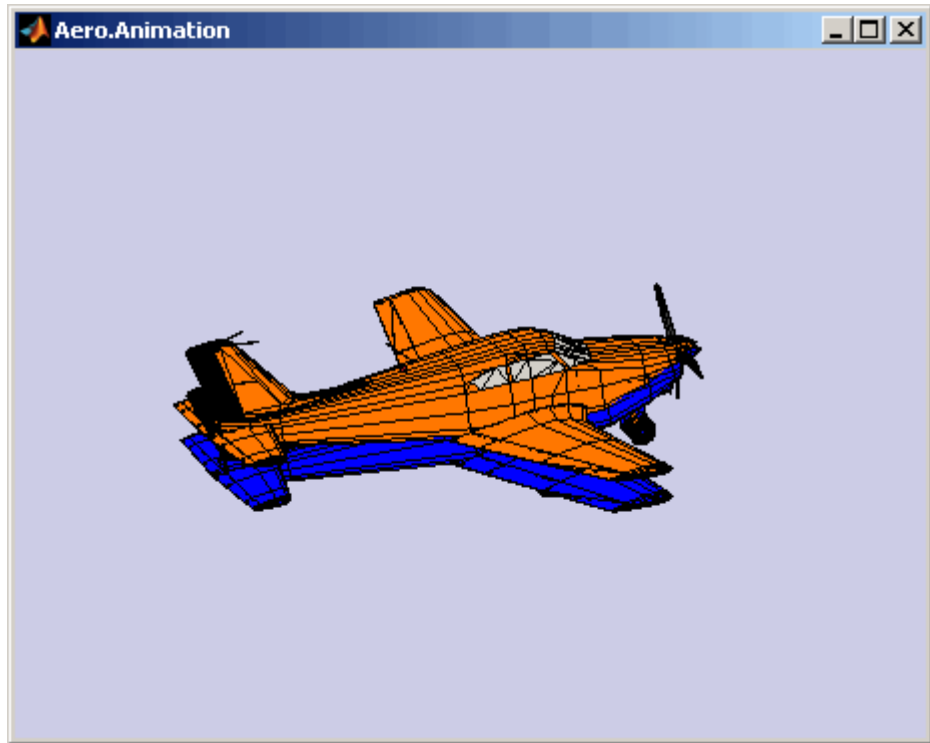
This command creates a figure object for the animation object.

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

## Playing Back Flight Trajectories Using the Animation Object

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

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



### Manipulating the Camera

This command series describes 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 the section “Playing Back Flight Trajectories Using the Animation Object” on page 2-29, the camera object uses a default value for the `PositionFcn` property. In this command series, the demo references a custom `PositionFcn` function, which uses a static position based on the position of the bodies; no dynamics are involved. The custom `PositionFcn` function is located in the `matlabroot\toolbox\ aero\astdemos` directory.

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

```
h.Camera.PositionFcn = @staticCameraPosition;
```

- 2 Run the animation again.

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

## Manipulating Bodies

This section illustrates some of the actions you can perform on bodies.

**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 Aero.Animation.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 Aero.Animation.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 “Creating 3-D Models with Patches” in the MATLAB documentation).

---

**Note** The use of transparencies might decrease animation speed on platforms that use software OpenGL<sup>®</sup> rendering (see `opengl` in the MATLAB documentation).

---

- 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 desired face and edge alpha values for the transparency.

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

- 3** Get the current face and edge alpha data and change all values to the desired alpha values. In the figure, note 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 will manipulate 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. Note the following:

- 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
```

## Using Aero.VirtualRealityAnimation Objects

The Aerospace Toolbox interface to virtual reality animation objects uses the Simulink 3D Animation software. See `Aero.VirtualRealityAnimation`, `Aero.Node`, and `Aero.Viewpoint` for details.

- 1** Create and configure an animation object.
  - a** Configure the animation object.
  - b** Initialize that object.
- 2** Enable the tracking of changes to virtual worlds.
- 3** Load the animation world.
- 4** Load time series data for simulation.
- 5** Set coordination information for the object.
- 6** Add a chase helicopter to the object.
- 7** Load time series data for chase helicopter simulation.
- 8** Set coordination information for the new object.
- 9** Add a new viewpoint for the helicopter.

- 10 Play the animation.
- 11 Create a new viewpoint.
- 12 Add a route.
- 13 Add another helicopter.
- 14 Remove bodies.
- 15 Revert to the original world.

### Running the Demo

- 1 Start the MATLAB software.
- 2 Run the demo either by entering `astvranim` in the MATLAB Command Window or by finding the demo entry (Visualize Aircraft Takeoff via the Simulink 3D Animation product) in the Demos browser and clicking **Run in the Command Window** on its demo page.

While running, the demo performs several steps by issuing a series of commands, as explained below.

### Creating and Configuring a Virtual Reality Animation Object

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

- 1 Create an animation object.

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

- 2 Configure the animation object to set the number of frames per second (`FramesPerSecond`) property. This 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. The settings in this demo result in a time step of approximately 0.5 s.

- 4 Specify the `.wrl` file for the `vrworld` object.

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

The virtual reality animation object reads in the `.wrl` file.

## Enabling Aero.VirtualRealityAnimation Methods to Track Changes to Virtual Worlds

`Aero.VirtualRealityAnimation` methods that change the current virtual reality world use a temporary `.wrl` file to manage those changes. To enable these methods to work in a write-protected directory such as `astdemos`, type the following.

- 1 Copy the virtual world file, `asttkoff.wrl`, to a temporary directory.

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

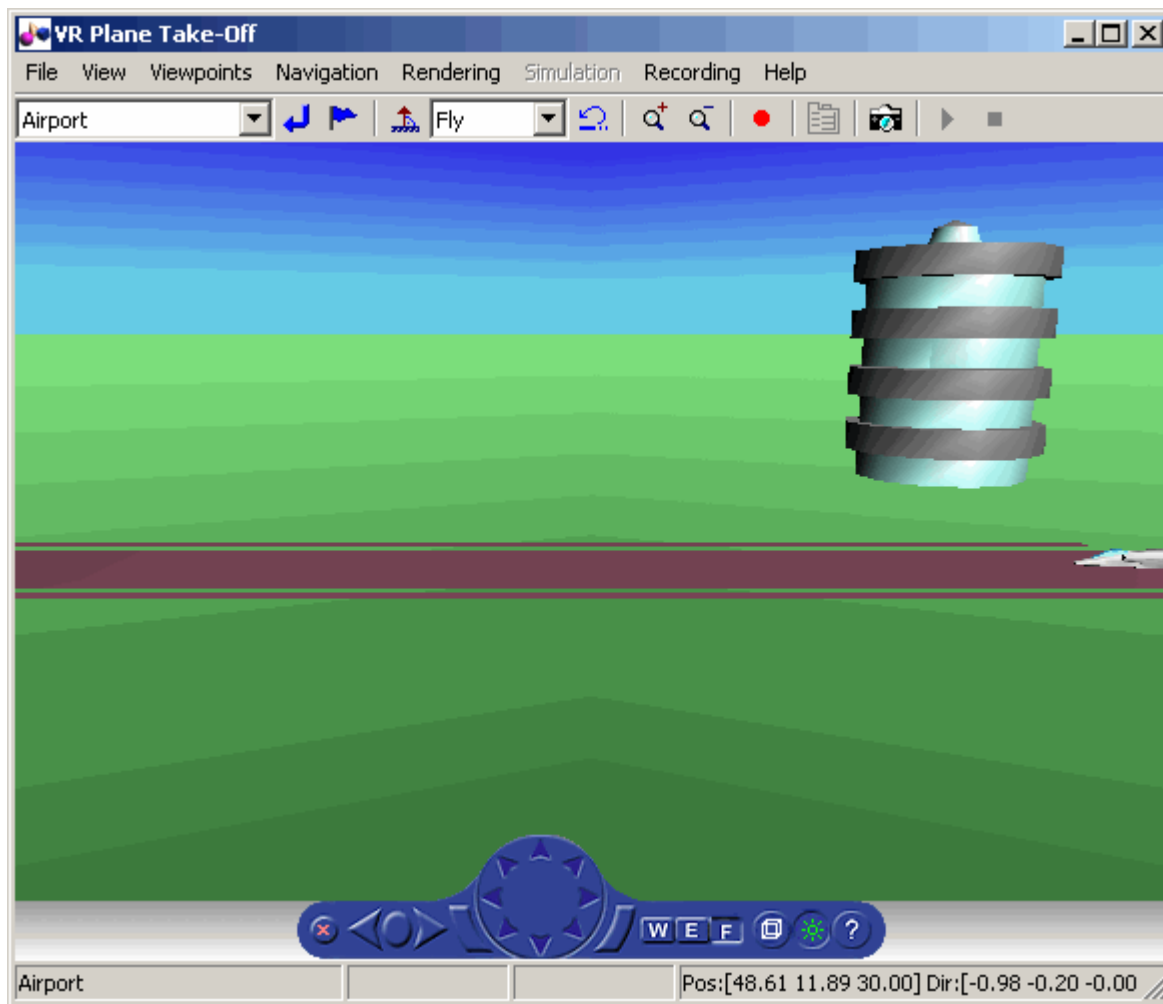
- 2 Set the `asttkoff.wrl` world filename to the copied `.wrl` file.

```
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
```

## Loading the Animation World

Load the animation world described in the `VRWorldFilename` field of the animation object. When parsing the world, this method creates node objects for existing nodes with `DEF` names. The `initialize` method also opens the Simulink 3D Animation Viewer.

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



### Displaying Figures

While working with this demo, you can capture a view of a scene with the `takeVRCapture` tool. This tool is specific to the `astvranim` demo. To display the initial scene, type

```
takeVRCapture(h.VRFigure);
```

A MATLAB figure window displays with the initial scene.

### Loading Time Series Data for Simulation

To prepare for simulation, set the simulation time series data. `takeoffData.mat` contains logged simulated data that you can use to set the time series data. `takeoffData` is set up as the Simulink structure 'StructureWithTime', which is a default data format.

**1** Load the `takeoffData`.

```
load takeoffData
```

**2** Set the time series data for the node.

```
h.Nodes{7}.TimeseriesSource = takeoffData;
h.Nodes{7}.TimeseriesSourceType = 'StructureWithTime';
```

### Aligning the Position and Rotation Data with Surrounding Virtual World Objects

The virtual reality animation object expects positions and rotations in aerospace body coordinates. If the input data coordinate system is different, as is the case in this demo, you must create a coordinate transformation function to correctly line up the position and rotation data with the surrounding objects in the virtual world. This code should set the coordinate transformation function for the virtual reality animation. The custom transfer function for this demo is `matlabroot/toolbox/aero/astdemos/vranimCustomTransform.m`. In this demo, if the input translation coordinates are  $[x1, y1, z1]$ , the custom transform function must adjust them as:

$$[X, Y, Z] = -[y1, x1, z1]$$

To run this custom transformation function, type:

```
h.Nodes{7}.CoordTransformFcn = @vranimCustomTransform;
```

### Viewing the Nodes in a Virtual Reality Animation Object

While working with this demo, you can view all the nodes currently in the virtual reality animation object with the `nodeInfo` method.

```
h.nodeInfo;
```

This method displays the nodes currently in your demo:

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

### Adding a Chase Helicopter

As part of the demo, add a chase helicopter node to your demo. Use the `addNode` method to add another node to the virtual reality animation object.

---

**Note** By default, each time you add or remove a node, or when you call the `saveas` method, a message shows the current `.wrl` file location. To disable this message, set the `ShowSaveWarning` property in the virtual reality animation object. You can disable this message before adding the chase helicopter.

---

- 1 Disable the message.

```
h.ShowSaveWarning = false;
```

- 2 Add the chase helicopter node.

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

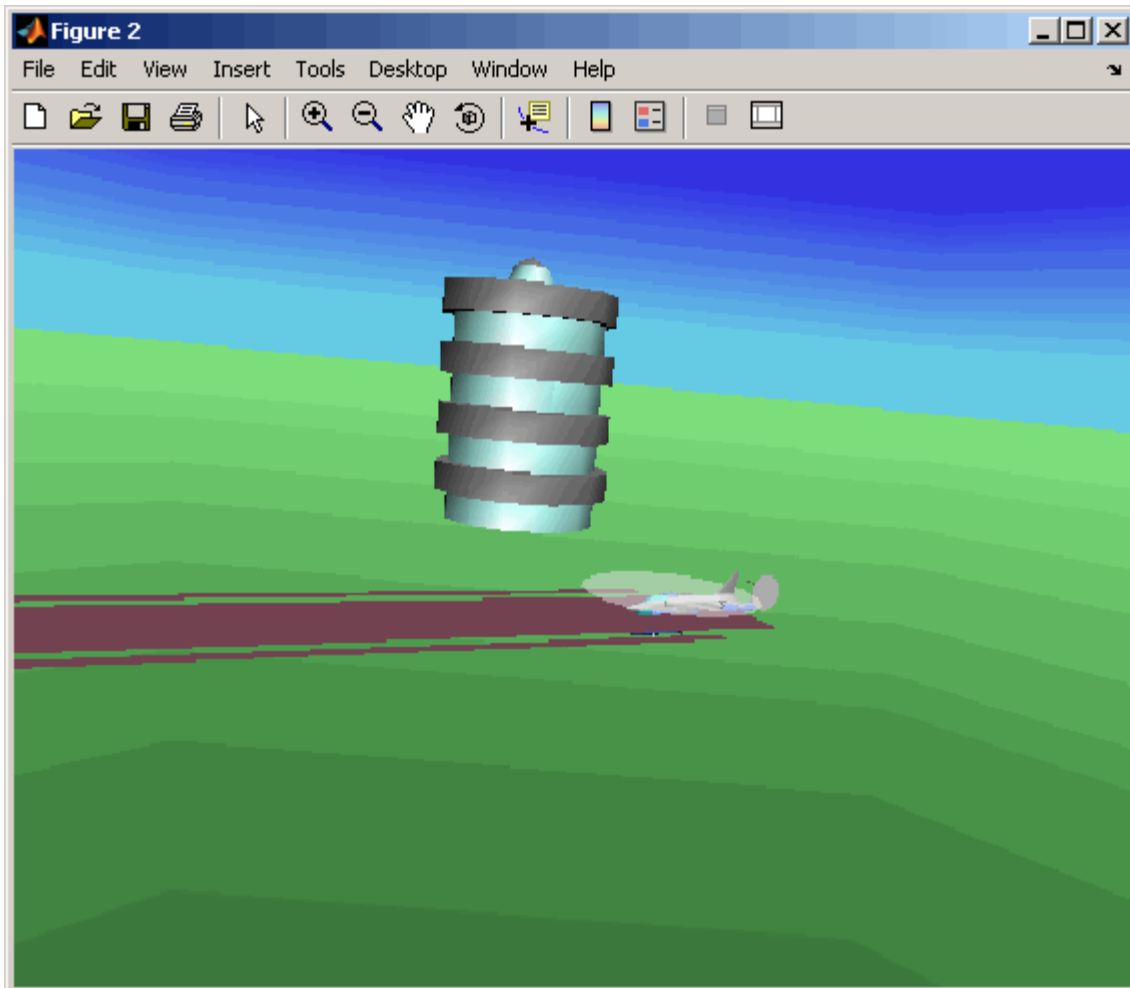
The helicopter appears in the Simulink 3D Animation Viewer.

- 3 Move the camera angle of the virtual reality figure to view the aircraft and newly added helicopter.

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

- 4 View the scene with the chase helicopter.

```
takeVRCapture(h.VRFigure);
```



### Loading Time Series Data for Simulation

To prepare to simulate the chase helicopter, set the simulation time series data. `chaseData.mat` contains logged simulated data that you can use to set the time series data. `chaseData` is set up as the Simulink structure 'StructureWithTime', which is a default data format.

1 Load the chaseData.

```
load chaseData
```

2 Set the time series data for the node.

```
h.Nodes{2}.TimeseriesSource = chaseData;
```

### **Aligning the Chase Helicopter Position and Rotation Data with Surrounding Virtual World Objects**

Use the custom transfer function to align the chase helicopter.

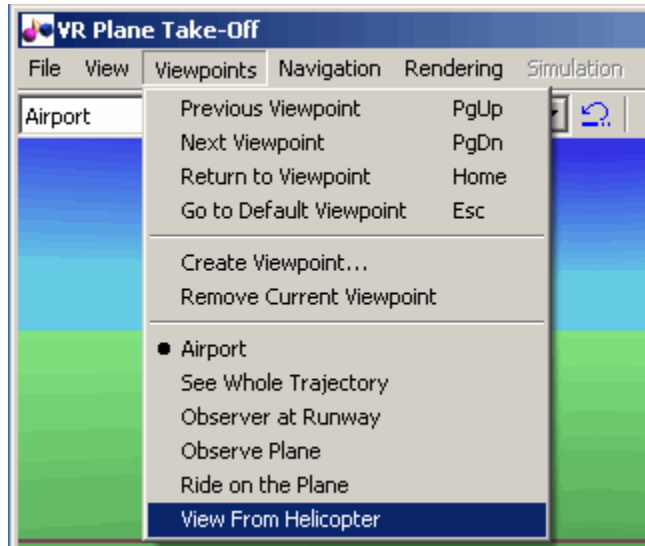
```
h.Nodes{2}.CoordTransformFcn = @vranimCustomTransform;
```

### **Adding a New Viewpoint**

To add a viewpoint for the chase helicopter, use the `addViewpoint` method. New viewpoints appear in the **Viewpoints** menu of the Simulink 3D Animation Viewer. Type the following to add the viewpoint **View From Helicopter** to the **Viewpoints** menu.

```
h.addViewpoint(h.Nodes{2}.VRNode, 'children', 'chaseView', 'View From Helicopter');
```





## Playing Back the Simulation

The play command animates the virtual reality world for the given position and angle for the duration of the time series data. Set the orientation of the viewpoint first.

- 1 Set the orientation of the viewpoint via the vrnoded object associated with the node object for the viewpoint.

```
setfield(h.Nodes{1}.VRNode,'orientation',[0 1 0 convang(160,'deg','rad')]);
set(h.VRFigure,'Viewpoint','View From Helicopter');
```

- 2 Play the animation.

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

## Adding a Route to the Camera1 Node

The vrworld has a Ride on the Plane viewpoint. To enable this viewpoint to function as intended, connect the plane position to the Camera1 node with the addRoute method. This method adds a VRML ROUTE statement.

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

### **Adding Another Helicopter and Viewing All Bodies Simultaneously**

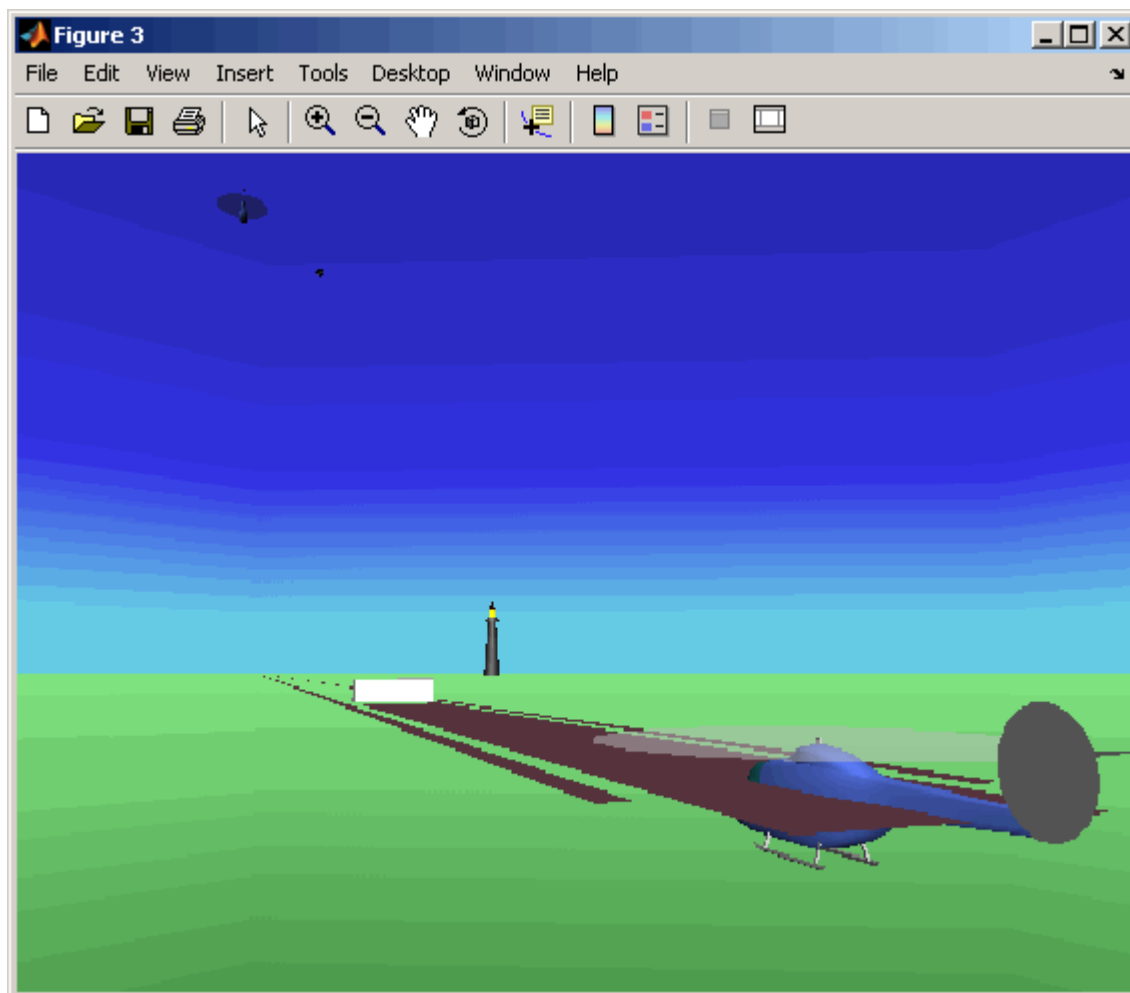
You can add another helicopter to the scene and also change the viewpoint to one that views all three bodies in the scene at once.

- 1** Add a new node, Lynx1.

```
h.addNode('Lynx1',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wr1']);
```

- 2** Change the viewpoint to one that views all three bodies.

```
set(h.VRFigure,'Viewpoint','See Whole Trajectory');
```



## Removing Bodies

Use the `removeNode` method to remove the second helicopter. To obtain the name of the node to remove, use the `nodeInfo` method.

- 1 View all the nodes.

```
h.nodeInfo
```

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

- 2** Remove the Lynx1 node.

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

- 3** Change the viewpoint to one that views the whole trajectory.

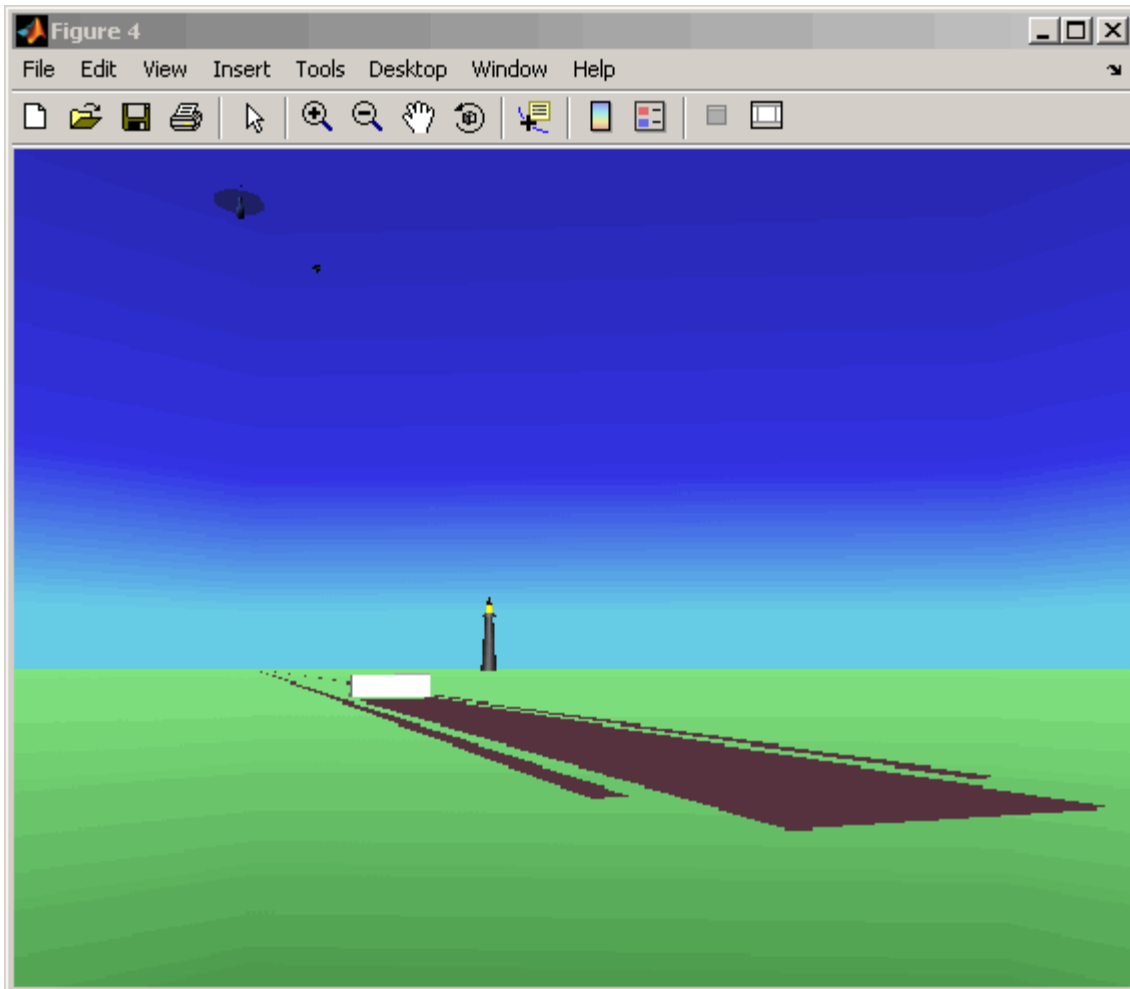
```
set(h.VRFigure,'Viewpoint','See Whole Trajectory');
```

- 4** Check that you have removed the node.

```
h.nodeInfo
```

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

The following figure is a view of the entire trajectory with the third body removed.



### Reverting to the Original World

The original file name is stored in the 'VRWorldOldFilename' property of the virtual reality animation object. To display the original world, set 'VRWorldFilename' to the original name and reinitialize it.

1 Revert to the original world, 'VRWorldFilename'.

```
h.VRWorldFilename = h.VRWorldOldFilename{1};
```

2 Reinitialize the restored world.

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

### Closing and Deleting Worlds

To close and delete a world, use the `delete` method.

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

### Using Aero.FlightGearAnimation Object

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 explains how to obtain and install the third-party FlightGear flight simulator. It then explains how to play back 3-D flight data by using a FlightGear demo, provided with your Aerospace Toolbox software, as an example.

- “About the FlightGear Interface” on page 2-48
- “Configuring Your Computer for FlightGear” on page 2-49
- “Installing and Starting FlightGear” on page 2-52
- “Working with the Flight Simulator Interface” on page 2-53
- “Running the Demo” on page 2-56

### 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 using 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. See “Working with the Flight Simulator Interface” on page 2-53 for interface details.

FlightGear is a separate software entity neither created, owned, nor maintained by The MathWorks.

- To report bugs in or request enhancements to the Aerospace Toolbox FlightGear interface, contact MathWorks Technical Support at [http://www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html).
- To report bugs or request enhancements to FlightGear itself, visit [www.flightgear.org](http://www.flightgear.org) and use the contact page.

**Obtaining FlightGear.** You can obtain FlightGear from [www.flightgear.org](http://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 Web site, [www.flightgear.org](http://www.flightgear.org).

The MathWorks tests of FlightGear's performance and stability indicate significant sensitivity to computer video cards, driver versions, and driver settings. You need OpenGL support with hardware acceleration activated. The OpenGL settings are particularly important. Without proper setup, performance can drop from about a 30 frames-per-second (fps) update rate to less than 1 fps.

**Graphics Recommendations for Microsoft Windows.** The MathWorks recommends the following for Windows® users:

- Choose a graphics card with good OpenGL performance.
- Always use the latest tested and stable driver release for your video card. Test the driver thoroughly on a few computers before deploying to others.

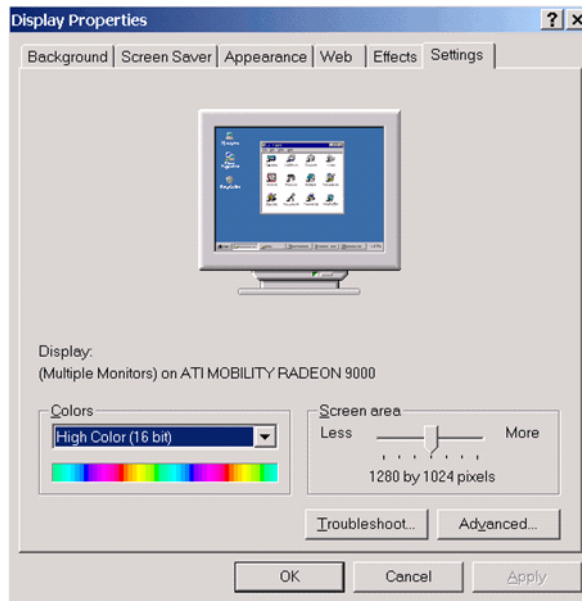
For Microsoft® Windows XP systems running on x86 (32-bit) or AMD-64/EM64T chip architectures, the graphics card operates in the unprotected kernel space known as Ring Zero. This means that glitches in

the driver can cause the Windows operating system to lock or crash. Before buying a large number of computers for 3-D applications, test, with your vendor, one or two computers to find a combination of hardware, operating system, drivers, and settings that are stable for your applications.

**Setting Up OpenGL Graphics on Windows.** For complete information on Silicon Graphics OpenGL settings, refer to the documentation at the OpenGL Web site, [www.opengl.org](http://www.opengl.org).

Follow these steps to optimize your video card settings. Your driver's panes might look different.

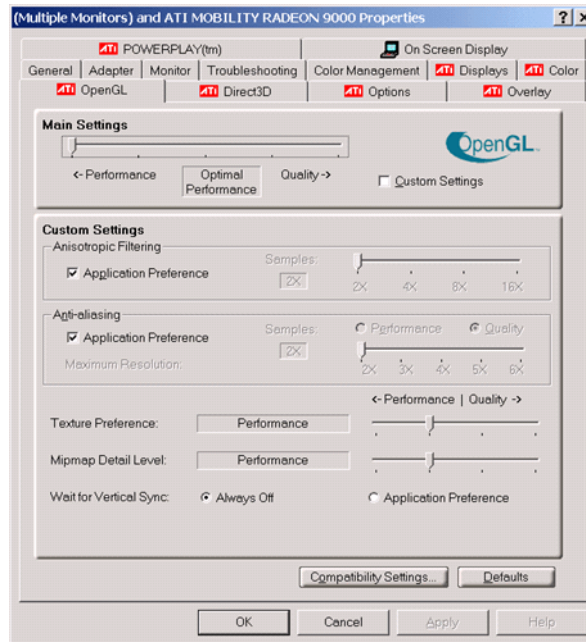
- 1 Ensure that you have activated the OpenGL hardware acceleration on your video card. On Windows, access this configuration through **Start > Settings > Control Panel > Display**, which opens the following dialog box. Select the **Settings** tab.



- 2 Click the **Advanced** button in the lower right of the dialog box, which opens the graphics card's custom configuration dialog box, and go to the



**OpenGL** tab. For an ATI Mobility Radeon 9000 video card, the **OpenGL** pane looks like this:



- 3** For best performance, move the **Main Settings** slider near the top of the dialog box to the **Performance** end of the slider.
- 4** If stability is a problem, try other screen resolutions, other color depths in the **Displays** pane, and other OpenGL acceleration modes.

Many cards perform much better at 16 bits-per-pixel color depth (also known as 65536 color mode, 16-bit color). For example, on an ATI Mobility Radeon 9000 running a given model, 30 fps are achieved in 16-bit color mode, while 2 fps are achieved in 32-bit color mode.

**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 Web site, [www.flightgear.org](http://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 Web site.

**Using MATLAB Graphics Controls to Configure Your OpenGL Settings.** You can also control your OpenGL rendering from the MATLAB command line with the MATLAB Graphics `opengl` command. Consult the `opengl` command reference for more information.

### **Installing and Starting FlightGear**

The extensive FlightGear documentation guides you through the installation in detail. Consult the documentation section of the FlightGear Web site for complete installation instructions: [www.flightgear.org](http://www.flightgear.org).

Keep the following points in mind:

- Generous central processor speed, system and video RAM, and virtual memory are essential for good flight simulator performance.  
The MathWorks recommends a minimum of 512 megabytes of system RAM and 128 megabytes of video RAM for reasonable performance.
- Be sure to have sufficient disk space for the FlightGear download and installation.
- The MathWorks recommends configuring your computer's graphics card before you install FlightGear. See the preceding section, "Configuring Your Computer for FlightGear" on page 2-49.
- Shutting down all running applications (including the MATLAB software) before installing FlightGear is recommended.
- The MathWorks tests indicate that 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 (<http://www.mathworks.com/products/aerotb/requirements.html>). The following table lists the properties you should be aware of before you start to use FlightGear.

<b>FlightGear Property</b>	<b>Directory Description</b>	<b>Platforms</b>	<b>Typical Location</b>
FlightGearBase-Directory	FlightGear installation directory.	Windows	C:\Program Files\FlightGear (default)
		Sun™ Solaris™ or Linux	Directory into which you installed FlightGear
		Mac®	/Applications (directory to which you dragged the FlightGear icon)
GeometryModelName	Model geometry directory	Windows	C:\Program Files\FlightGear\data\Aircraft\HL20 (default)
		Solaris or Linux	\$FlightGearBaseDirectory/data/Aircraft/HL20
		Mac	\$FlightGearBaseDirectory/FlightGear.app/Contents/Resources/data/Aircraft/HL20

## Working with the Flight Simulator Interface

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

You need to have FlightGear installed and configured before attempting to simulate this model. See “About the FlightGear Interface” on page 2-48.

To run the demo:

- 1 Import the aircraft geometry into FlightGear.
- 2 Run the demo. The demo performs the following steps:
  - a Loads recorded trajectory data
  - b Creates a time series object from trajectory data
  - c Creates a FlightGearAnimation object
- 3 Modify the animation object properties, if needed.
- 4 Create a run script for launching FlightGear flight simulator.
- 5 Start FlightGear flight simulator.
- 6 Play back the flight trajectory.

The following sections describe how to perform these steps in detail.

**Importing the Aircraft Geometry into FlightGear.** Before running the demo, 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 “Installing and Starting FlightGear” on page 2-52.

If your platform is Windows:

- 1 Go to your installed FlightGear directory. Open the data directory, then the Aircraft directory: `FlightGear\data\Aircraft\`.
- 2 You may already have an HL20 subdirectory there, if you have previously run the Aerospace Blockset NASA HL-20 with FlightGear Interface demo. In this case, you don’t have to do anything, because the geometry model is the same.

Otherwise, copy the HL20 folder from the `matlabroot\toolbox\aero\aedemos\` directory to the `FlightGear\data\Aircraft\` directory. This folder contains the

preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file `matlabroot\toolbox\aero\aedemos\HL20\models\HL20.xml` defines the geometry.

If your platform is Solaris or Linux:

- 1** Go to your installed FlightGear directory. Open the data directory, then the Aircraft directory: `$FlightGearBaseDirectory/data/Aircraft/`.
- 2** You may already have an HL20 subdirectory there, if you have previously run the Aerospace Blockset NASA HL-20 with FlightGear Interface demo. In this case, you do not have to do anything, because the geometry model is the same.

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

If your platform is Mac:

- 1** Open a terminal.
- 2** List the contents of the Aircraft directory. For example, type
 

```
ls $FlightGearBaseDirectory/data/Aircraft/
```
- 3** You may already have an HL20 subdirectory there, if you have previously run the Aerospace Blockset NASA HL-20 with FlightGear Interface demo. In this case, you do not have to do anything, because the geometry model is the same. Continue to “Running the Demo” on page 2-27.

Otherwise, copy the HL20 folder from the

```
matlabroot/toolbox/aero/aedemos/
```

directory to the

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

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

### Running the Demo

- 1 Start the MATLAB software.
- 2 Run the demo either by entering `astfganim` in the MATLAB Command Window or by finding the demo entry (Displaying Flight Trajectory Data) in the Demos browser and clicking **Run in the Command Window** on its demo page.

While running, the demo performs several steps by issuing a series of commands, as explained below.

**Loading Recorded Flight Trajectory Data.** The flight trajectory data for this example is stored in a comma separated value formatted file. Using `csvread`, the data is read from the file starting at row 1 and column 0, which skips the header information.

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

**Creating a Time Series Object from Trajectory Data.** The time series object, `ts`, is created from the latitude, longitude, altitude, and Euler angle data along with 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 Opening a `FlightGearAnimation` object.

```
h = fganimation;
```

- 2 Setting `FlightGearAnimation` object properties for the time series.

```
h.TimeseriesSourceType = 'Timeseries';
h.TimeseriesSource = ts;
```

- 3** Setting FlightGearAnimation object properties relating to FlightGear. These properties include the path to the installation directory, the version number, the aircraft geometry model, and network information for the FlightGear flight simulator.

```
h.FlightGearBaseDirectory = 'C:\Program Files\FlightGear10';
h.FlightGearVersion = '1.0';
h.GeometryModelName = 'HL20';
h.DestinationIpAddress = '127.0.0.1';
h.DestinationPort = '5502';
```

- 4** Setting 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** Setting the seconds of animation data per second of wall-clock time.

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

- 6** Checking the FlightGearAnimation object properties and their values.

```
get(h)
```

At this point, the demo stops running and returns the FlightGearAnimation object, h:

```
TimeseriesSource: [196x1 timeseries]
TimeseriesSourceType: 'Timeseries'
TimeseriesReadFcn: @TimeseriesRead
TimeScaling: 5
FramesPerSecond: 12
FlightGearVersion: '1.0'
OutputFileName: 'runfg.bat'
```

```
FlightGearBaseDirectory: 'C:\Program Files\FlightGear10'  
GeometryModelName: 'HL20'  
DestinationIpAddress: '127.0.0.1'  
DestinationPort: '5502'  
AirportId: 'KSFO'  
RunwayId: '10L'  
InitialAltitude: 7224  
InitialHeading: 113  
OffsetDistance: 4.7200  
OffsetAzimuth: 0
```

You can now set the object properties for data playback (see “Modifying the FlightGearAnimation Object Properties” on page 2-58).

**Modifying the FlightGearAnimation Object Properties.** Modify the FlightGearAnimation object properties as needed. If your FlightGear installation directory is other than that in the demo (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 2-58).

**Generating the Run Script.** To start FlightGear with the desired initial conditions (location, date, time, weather, operating modes), it is best to 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.

This file does not need to be generated each time the data is viewed, only when the desired initial conditions or FlightGear information changes.

You are now ready to start FlightGear (see “Starting the FlightGear Flight Simulator” on page 2-59).

**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 view, helicopter view, chase view, and so on.

---

You are now ready to play back data (see “Playing Back the Flight Trajectory” on page 2-59).

**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.



# Function Reference

---

Animation Objects (p. 3-3)	Manipulate Aero.Animation objects
Body Objects (p. 3-4)	Manipulate Aero.Body objects
Camera Objects (p. 3-5)	Manipulate Aero.Camera objects
FlightGear Objects (p. 3-5)	Manipulate Aero.FlightGearAnimation objects
Geometry Objects (p. 3-6)	Manipulate Aero.Geometry objects
Node Objects (p. 3-7)	Manipulate Aero.Node objects
Viewpoint Objects (p. 3-8)	Manipulate Aero.Viewpoint objects
Virtual Reality Objects (p. 3-9)	Manipulate Aero.VirtualRealityAnimation objects
Axes Transformations (p. 3-10)	Transform axes of coordinate systems to different types, such as Euler angles to quaternions and vice versa
Environment (p. 3-11)	Simulate various aspects of aircraft environment, such as atmosphere conditions, gravity, magnetic fields, and wind
File Reading (p. 3-11)	Read standard aerodynamic file formats into the MATLAB interface
Flight Parameters (p. 3-12)	Various flight parameters, including ideal airspeed correction, Mach number, and dynamic pressure

Quaternion Math (p. 3-12)

Common mathematical and matrix operations, including quaternion multiplication, division, normalization, and rotating vector by quaternion

Time (p. 3-13)

Time calculations, including Julian dates, decimal year, and leap year

Unit Conversion (p. 3-13)

Convert common measurement units from one system to another, such as converting acceleration from feet per second to meters per second and vice versa

## Animation Objects

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

## Body Objects

Body (Aero.Body)	Construct body object for use with animation object
findstartstoptimes (Aero.Body)	Return start and stop times of time series data
generatePatches (Aero.Body)	Generate patches for body with loaded face, vertex, and color data
load (Aero.Body)	Get geometry data from source
move (Aero.Body)	Change animation body position and orientation
update (Aero.Body)	Change body position and orientation as function of time

## Camera Objects

Camera (Aero.Camera)	Construct camera object for use with animation object
update (Aero.Camera)	Update camera position based on time and position of other Aero.Body objects

## FlightGear Objects

ClearTimer (Aero.FlightGearAnimation)	Clear and delete timer for animation of FlightGear flight simulator
delete (Aero.FlightGearAnimation)	Destroy FlightGear animation object
fganimation (Aero.FlightGearAnimation)	Construct FlightGear animation object
GenerateRunScript (Aero.FlightGearAnimation)	Generate run script for FlightGear flight simulator
initialize (Aero.FlightGearAnimation)	Set up FlightGear animation object
play (Aero.FlightGearAnimation)	Animate FlightGear flight simulator using given position/angle time series
SetTimer (Aero.FlightGearAnimation)	Set name of timer for animation of FlightGear flight simulator
update (Aero.FlightGearAnimation)	Update position data to FlightGear animation object

## **Geometry Objects**

Geometry (Aero.Geometry)

Construct 3-D geometry for use with animation object

read (Aero.Geometry)

Read geometry data using current reader



## Node Objects

findstartstoptimes (Aero.Node)	Return start and stop times for time series data
move (Aero.Node)	Change node translation and rotation
Node (Aero.Node)	Create node object for use with virtual reality animation
update (Aero.Node)	Change node position and orientation versus time data

## **Viewpoint Objects**

Viewpoint (Aero.Viewpoint)

Create viewpoint object for use in virtual reality animation

## Virtual Reality Objects

addNode (Aero.VirtualRealityAnimation)	Add existing node to current virtual reality world
addRoute (Aero.VirtualRealityAnimation)	Add VRML ROUTE statement to virtual reality animation
addViewpoint (Aero.VirtualRealityAnimation)	Add viewpoint for virtual reality animation
delete (Aero.VirtualRealityAnimation)	Destroy virtual reality animation object
initialize (Aero.VirtualRealityAnimation)	Create and populate virtual reality animation object
nodeInfo (Aero.VirtualRealityAnimation)	Create list of nodes associated with virtual reality animation object
play (Aero.VirtualRealityAnimation)	Animate virtual reality world for given position and angle in time series data
removeNode (Aero.VirtualRealityAnimation)	Remove node from virtual reality animation object
removeViewpoint (Aero.VirtualRealityAnimation)	Remove viewpoint node from virtual reality animation
saveas (Aero.VirtualRealityAnimation)	Save virtual reality world associated with virtual reality animation object
updateNodes (Aero.VirtualRealityAnimation)	Change virtual reality animation node position and orientation as function of time
VirtualRealityAnimation (Aero.VirtualRealityAnimation)	Construct virtual reality animation object

## Axes Transformations

<code>angle2dcm</code>	Create direction cosine matrix from rotation angles
<code>angle2quat</code>	Convert rotation angles to quaternion
<code>dcm2alphabeta</code>	Convert direction cosine matrix to angle of attack and sideslip angle
<code>dcm2angle</code>	Create rotation angles from direction cosine matrix
<code>dcm2latlon</code>	Convert direction cosine matrix to geodetic latitude and longitude
<code>dcm2quat</code>	Convert direction cosine matrix to quaternion
<code>dcmbody2wind</code>	Convert angle of attack and sideslip angle to direction cosine matrix
<code>dcmecef2ned</code>	Convert geodetic latitude and longitude to direction cosine matrix
<code>ecef2lla</code>	Convert Earth-centered Earth-fixed (ECEF) coordinates to geodetic coordinates
<code>geoc2geod</code>	Convert geocentric latitude to geodetic latitude
<code>geod2geoc</code>	Convert geodetic latitude to geocentric latitude
<code>lla2ecef</code>	Convert geodetic coordinates to Earth-centered Earth-fixed (ECEF) coordinates
<code>quat2angle</code>	Convert quaternion to rotation angles
<code>quat2dcm</code>	Convert quaternion to direction cosine matrix

## Environment

atmoscira	Use COSPAR International Reference Atmosphere 1986 model
atmoscoesa	Use 1976 COESA model
atmosisa	Use International Standard Atmosphere model
atmoslapse	Use Lapse Rate Atmosphere model
atmosnonstd	Use climatic data from MIL-STD-210 or MIL-HDBK-310
atmosnrlmsise00	Implement mathematical representation of 2001 United States Naval Research Laboratory Mass Spectrometer and Incoherent Scatter Radar Exosphere
atmosphalt	Calculate pressure altitude based on ambient pressure
gravitywgs84	Implement 1984 World Geodetic System (WGS84) representation of Earth's gravity
wrldmagm	Use World Magnetic Model

## File Reading

datcomimport	Bring DATCOM file into MATLAB environment
--------------	---

## Flight Parameters

airspeed	Compute airspeed from velocity
alphabet	Compute incidence and sideslip angles
correctairspeed	Calculate equivalent airspeed (EAS), calibrated airspeed (CAS), or true airspeed (TAS) from one of other two airspeeds
dpressure	Compute dynamic pressure using velocity and density
geocradius	Estimate radius of ellipsoid planet at geocentric latitude
machnumber	Compute Mach number using velocity and speed of sound
rrdelta	Compute relative pressure ratio
rrsigma	Compute relative density ratio
rrtheta	Compute relative temperature ratio

## Quaternion Math

quatconj	Calculate conjugate of quaternion
quatdivide	Divide quaternion by another quaternion
quatinv	Calculate inverse of quaternion
quatmod	Calculate modulus of quaternion
quatmultiply	Calculate product of two quaternions
quatnorm	Calculate norm of quaternion
quatnormalize	Normalize quaternion
quatrotate	Rotate vector by quaternion

## Time

decyear	Calculate decimal year
juliandate	Calculate Julian date
leapyear	Determine leap year
mjuliandate	Calculate modified Julian date

## Unit Conversion

convacc	Convert from acceleration units to desired acceleration units
convang	Convert from angle units to desired angle units
convangacc	Convert from angular acceleration units to desired angular acceleration units
convangvel	Convert from angular velocity units to desired angular velocity units
convdensity	Convert from density units to desired density units
convforce	Convert from force units to desired force units
convlength	Convert from length units to desired length units
convmass	Convert from mass units to desired mass units
convpres	Convert from pressure units to desired pressure units

convtemp

Convert from temperature units to  
desired temperature units

convvel

Convert from velocity units to  
desired velocity units



# Alphabetical List

---

# addNode (Aero.VirtualRealityAnimation)

---

**Purpose** Add existing node to current virtual reality world

**Syntax** `addNode(h, node_name, wr1_file)`  
`h.addNode(node_name, wr1_file)`

**Description** `addNode(h, node_name, wr1_file)` and `h.addNode(node_name, wr1_file)` add an existing node, `node_name`, to the current virtual reality world. The `wr1_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 `wr1_file`. `node_name` must be unique from other node names in the same `.wr1` file. `wr1_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 `.wr1` 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 `.wr1` 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 `.wr1` file, place each node in a separate `.wr1` file.

**Example** Add node to world defined in `chaseHelicopter.wr1`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot, '/toolbox/aero/astdemos/asttkoff.wr1'];
copyfile(h.VRWorldFilename,[tempdir, 'asttkoff.wr1'],'f');
h.VRWorldFilename = [tempdir, 'asttkoff.wr1'];
h.initialize();
h.addNode('Lynx',[matlabroot, '/toolbox/aero/astdemos/chaseHelicopter.wr1']);
```

**See Also** `Aero.Node`, `move`, `removeNode`, `updateNodes`,  
`Aero.VirtualRealityAnimation`

# addRoute (Aero.VirtualRealityAnimation)

---

## Purpose

Add VRML ROUTE statement to virtual reality animation

## Syntax

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

## Description

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

## Examples

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

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

## See Also

`addViewpoint`

# addViewpoint (Aero.VirtualRealityAnimation)

---

**Purpose** 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 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 string you want to describe the viewpoint and

# addViewpoint (Aero.VirtualRealityAnimation)

---

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 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  $\Theta$ ).

---

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

---

## Examples

Add a viewpoint named chaseView.

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

## See Also

addRoute, removeViewpoint

# airspeed

---

**Purpose** Compute airspeed from velocity

**Syntax** `as = airspeed(v)`

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

**Examples** Determine the airspeed for velocity in feet per second:

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

```
as =
```

```
91.4538
```

Determine the airspeed for velocity in knots:

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

```
as =
```

```
54.1849
```

```
5.4083
```

**See Also** `alpha`, `correctairspeed`, `dpressure`, `machnumber`

**Purpose** Construct animation object

**Syntax** `h = Aero.Animation`

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

---

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

---

**Examples** `h=Aero.Animation`

# Aero.Animation.addBody

---

**Purpose** Add loaded body to animation object and generate its patches

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

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

**Inputs**

<code>h</code>	Animation object.
<code>b</code>	Loaded body.

**Outputs**

<code>idx</code>	Index of the body to be added.
------------------	--------------------------------

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

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



## Purpose

Create body and its associated patches in animation

## Syntax

```
idx = createBody(h,bodyDataSrc)
idx = h.createBody(bodyDataSrc)
idx = createBody(h,bodyDataSrc,geometrysource)
idx = h.createBody(bodyDataSrc,geometrysource)
```

## Description

`idx = createBody(h,bodyDataSrc)` and `idx = h.createBody(bodyDataSrc)` create a new body using the `bodyDataSrc`, makes its patches, and adds it to the animation object `h`. This command assumes a default geometry source type set to `Auto`.

`idx = createBody(h,bodyDataSrc,geometrysource)` and `idx = h.createBody(bodyDataSrc,geometrysource)` create a new body using the `bodyDataSrc` file, makes its patches, and adds it to the animation object `h`. `geometrysource` is the geometry source type for the body.

## Inputs

`bodyDataSrc`

Source of data for body.

`geometrysource`

Geometry source type for body:

- **Auto** — Recognizes `.mat` extensions as MAT-files, `.ac` extensions as Ac3d files, and structures containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables. Default.
- **Variable** — Recognizes structures containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables.
- **MatFile** — Recognizes `.mat` extensions as MAT-files.
- **Ac3d** — Recognizes `.ac` extensions as Ac3d files.
- **Custom** — Recognizes custom extensions.

# Aero.Animation.createBody

---

## Outputs

`idx` Index of the body to be created.

## Examples

Create a body for the animation object, `h`. Use the Ac3d format data source `pa24-250_orange.ac`, for the body.

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

**Purpose** Destroy animation object

**Syntax** `delete(h)`  
`h.delete`

**Description** `delete(h)` and `h.delete` destroy the animation object `h`. This function also destroys the animation object figure, and any objects that the animation object contained (for example, bodies, camera, and geometry).

**Inputs** `h` Animation object.

**Examples** Delete the animation object, `h`.

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

# Aero.Animation.hide

---

<b>Purpose</b>	Hide animation figure
<b>Syntax</b>	<code>hide(h)</code> <code>h.hide</code>
<b>Description</b>	<code>hide(h)</code> and <code>h.hide</code> hide (close) the figure for the animation object <code>h</code> . Use <code>Aero.Animation.show</code> to redisplay the animation object figure.
<b>Inputs</b>	<code>h</code> Animation object.
<b>Examples</b>	Hide the animation object figure that the <code>Aero.Animation.show</code> method displays. <pre>h=Aero.Animation; h.show; h.hide;</pre>

<b>Purpose</b>	Initialize animation graphics if needed		
<b>Syntax</b>	<pre>initIfNeeded(h) h.initIfNeeded</pre>		
<b>Description</b>	<code>initIfNeeded(h)</code> and <code>h.initIfNeeded</code> initialize animation object graphics if necessary.		
<b>Inputs</b>	<table><tr><td><code>h</code></td><td>Animation object.</td></tr></table>	<code>h</code>	Animation object.
<code>h</code>	Animation object.		
<b>Examples</b>	Initialize the animation object graphics of <code>h</code> as needed. <pre>h=Aero.Animation; h.initIfNeeded;</pre>		

# Aero.Animation.initialize

---

**Purpose** Create animation object figure and axes and build patches for bodies

**Syntax** `initialize(h)`  
`h.initialize`

**Description** `initialize(h)` and `h.initialize` create a figure and axes for the animation object `h`, and builds patches for the bodies associated with the animation object. If there is an existing figure, this function

- 1** Clears out the old figure and its patches.
- 2** Creates a new figure and axes with default values.
- 3** Repopulates the axes with new patches using the surface to patch data from each body.

**Inputs** `h` Animation object.

**Examples** Initialize the animation object, `h`.

```
h = Aero.Animation;  
h.initialize();
```

**Purpose** Move body in animation object

**Syntax** `moveBody(h,idx,translation,rotation)`  
`h.moveBody(idx,translation,rotation)`

**Description** `moveBody(h,idx,translation,rotation)` and `h.moveBody(idx,translation,rotation)` set a new position and attitude for the body specified with the index `idx` in the animation object `h`. `translation` is a 1-by-3 matrix in the aerospace body coordinate system. `rotation` is a 1-by-3 matrix, in radians, that specifies the rotations about the right-hand  $x$ - $y$ - $z$  sequence of coordinate axes. The order of application of the rotation is  $z$ - $y$ - $x$  ( $R$ - $Q$ - $P$ ).

<b>Inputs</b>	<code>h</code>	Animation object.
	<code>translation</code>	1-by-3 matrix in the aerospace body coordinate system.
	<code>rotation</code>	1-by-3 matrix, in radians, that specifies the rotations about the right-hand $x$ - $y$ - $z$ sequence of coordinate axes.
	<code>idx</code>	Body specified with this index.

**Examples** Move the body with the index 1 to position offset from the original by + [0 0 -3] and rotation, `rot1`.

```
h = Aero.Animation;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
pos1 = h.Bodies{1}.Position;
rot1 = h.Bodies{1}.Rotation;
h.moveBody(1,pos1 + [0 0 -3],rot1);
```

# Aero.Animation.play

---

**Purpose** Animate Aero.Animation object given position/angle time series

**Syntax** play(h)  
play.h

**Description** play(h) and play.h animate the loaded geometry in h for the current TimeseriesDataSource at the specified rate given by the 'TimeScaling' property (in seconds of animation data per second of wall-clock time) and animated at a certain number of frames per second using the 'FramesPerSecond' property.

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

'Timeseries' MATLAB time series data with six values per time:

x y z phi theta psi

The values are resampled.

'Simulink.Timeseries' Simulink.Timeseries (Simulink signal logging):

- First data item  
x y z
- Second data item  
phi theta psi



'StructureWithTime'

Simulink struct with time (for example, Simulink root outport logging 'Structure with time'):

- `signals(1).values:` x y z
- `signals(2).values:` phi theta psi

Signals are linearly interpolated vs. time using `interp1`.

'Array6DoF'

A double-precision array in `n` rows and 7 columns for 6-DoF data: time x y z phi theta psi. If a double-precision array of 8 or more columns is in 'TimeseriesSource', the first 7 columns are used as 6-DoF data.

'Array3DoF'

A double-precision array in `n` rows and 4 columns for 3-DoF data: time x z theta. If a double-precision array of 5 or more columns is in 'TimeseriesSource', the first 4 columns are used as 3-DoF data.

'Custom'

Position and angle data is retrieved from 'TimeseriesSource' by the currently registered 'TimeseriesReadFcn'.

The following are limitations for the `TStart` and `TFinal` values:

- `TStart` and `TFinal` must be numeric.
- `TStart` and `TFinal` cannot be `Inf` or `NaN`.
- `TFinal` must be greater than or equal to `TStart`.

# Aero.Animation.play

---

- TFinal cannot be greater than the maximum Timeseries time.
- TStart cannot be less than the minimum Timeseries time.

The time advancement algorithm used by play is based on animation frames counted by ticks:

```
ticks = ticks + 1;  
time = tstart + ticks*FramesPerSecond*TimeScaling;
```

where

TimeScaling	Specify the seconds of animation data per second of wall-clock time.
FramesPerSecond	Specify the number of frames per second used to animate the 'TimeseriesSource'.

For default 'TimeseriesReadFcn' methods, the last frame played is the last time value.

Time is in seconds, position values are in the same units as the geometry data loaded into the animation object, and all angles are in radians.

---

**Note** If there is a 15% difference between the expected time advance and the actual time advance, this method will generate the following warning:

```
TimerPeriod has been set to <value>. You may wish to modify the animation  
TimeScaling and FramesPerSecond properties to compensate for the  
millisecond limit of the TimerPeriod. See documentation for details.
```

---

## Inputs

h Animation object.

## Examples

Animate the body, idx1, for the duration of the time series data.

```
h = Aero.Animation;  
h.FramesPerSecond = 10;  
h.TimeScaling = 5;  
idx1 = h.createBody('pa24-250_orange.ac', 'Ac3d');  
load simdata;  
h.Bodies{1}.TimeSeriesSource = simdata;  
h.show();  
h.play();
```

# Aero.Animation.removeBody

---

**Purpose** Remove one body from animation

**Syntax** `h = removeBody(h,idx)`  
`h = h.removeBody(idx)`

**Description** `h = removeBody(h,idx)` and `h = h.removeBody(idx)` remove the body specified by the index `idx` from the animation object `h`.

**Inputs**

<code>h</code>	Animation object.
<code>idx</code>	Body specified with this index.

**Examples** Remove the body identified by the index, 1.

```
h = Aero.Animation;  
idx1 = h.createBody('pa24-250_orange.ac', 'Ac3d');  
h = removeBody(h,1)
```

<b>Purpose</b>	Show animation object figure
<b>Syntax</b>	<code>show(h)</code> <code>h.show</code>
<b>Description</b>	<code>show(h)</code> and <code>h.show</code> create the figure graphics object for the animation object <code>h</code> . Use the <code>Aero.Animation.hide</code> function to close the figure.
<b>Inputs</b>	<code>h</code> Animation object.
<b>Examples</b>	Show the animation object, <code>h</code> . <pre>h = Aero.Animation; idx1 = h.createBody('pa24-250_orange.ac', 'Ac3d'); h.show;</pre>

# Aero.Animation.updateBodies

---

**Purpose** Update bodies of animation object

**Syntax** `h = updateBodies(time)`  
`h.updateBodies(time)`

**Description** `h = updateBodies(time)` and `h.updateBodies(time)` set the new position and attitude of movable bodies in the animation object `h`. This function updates the bodies contained in the animation object `h`. `time` is a scalar in seconds.

**Examples** Configure a body with `TimeSeriesSource` set to `simdata`, then update the body with time  $t$  equal to 0.

```
h = Aero.Animation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
idx1 = h.createBody('pa24-250_orange.ac', 'Ac3d');
load simdata;
h.Bodies{1}.TimeSeriesSource = simdata;
t = 0;
h.updateBodies(t);
```

**Purpose** Update camera in animation object

**Syntax** `updateCamera(h,time)`  
`h.updateCamera(time)`

**Description** `updateCamera(h,time)` and `h.updateCamera(time)` update the camera in the animation object `h`. `time` is a scalar in seconds.

---

**Note** The `PositionFcn` property of a camera object controls the camera position relative to the bodies in the animation. The default camera `PositionFcn` follows the path of a first order chase vehicle. Therefore, it takes a few steps for the camera to position itself correctly in the chase plane position.

---

**Inputs**

<code>h</code>	Animation object.
<code>time</code>	Scalar in seconds.

**Examples** Configure a body with `TimeSeriesSource` set to `simdata`, then update the camera with time  $t$  equal to 0.

```
h = Aero.Animation;  
h.FramesPerSecond = 10;  
h.TimeScaling = 5;  
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');  
load simdata;  
h.Bodies{1}.TimeSeriesSource = simdata;  
t = 0;  
h.updateCamera(t);
```

# Aero.Animation.Bodies property

---

**Purpose** Specify name of animation object

**Values** MATLAB array  
**Default:** [ ]

**Description** This property specifies the bodies that the animation object contains.



# Aero.Animation.Camera property

---

<b>Purpose</b>	Specify camera that animation object contains
<b>Values</b>	handle Default: [ ]
<b>Description</b>	This property specifies the camera that the animation object contains.

# Aero.Animation.Figure property

---

<b>Purpose</b>	Specify name of figure object
<b>Values</b>	MATLAB array <b>Default:</b> [ ]
<b>Description</b>	This property specifies the name of the figure object.

# Aero.Animation.FigureCustomizationFcn property

---

<b>Purpose</b>	Specify figure customization function
<b>Values</b>	MATLAB array Default: [ ]
<b>Description</b>	This property specifies the figure customization function.

# Aero.Animation.FramesPerSecond property

---

<b>Purpose</b>	Animation rate
<b>Values</b>	MATLAB array <b>Default:</b> 12
<b>Description</b>	This property specifies rate in frames per second.

# Aero.Animation.Name property

---

**Purpose** Specify name of animation object

**Values** String  
Default: ' '

**Description** This property specifies the name of the animation object.

# Aero.Animation.TCurrent property

---

<b>Purpose</b>	Current time
<b>Values</b>	double Default: 0
<b>Description</b>	This property specifies the current time.

# Aero.Animation.TFinal property

---

<b>Purpose</b>	End time
<b>Values</b>	double Default: NaN
<b>Description</b>	This property specifies the end time.

# Aero.Animation.TStart property

---

**Purpose** Start time

**Values** double  
Default: NaN

**Description** This property specifies the start time.



# Aero.Animation.TimeScaling property

---

<b>Purpose</b>	Scaling time
<b>Values</b>	double Default: 1
<b>Description</b>	This property specifies the time, in seconds.

# alphabet

---

**Purpose** Compute incidence and sideslip angles

**Syntax** `[a b] = alphabet(v)`

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

**Examples** Determine the incidence and sideslip angles for velocity in feet per second:

```
[alpha beta] = alphabet([84.3905 33.7562 10.1269])
```

```
alpha =
```

```
0.1194
```

```
beta =
```

```
0.3780
```

Determine the incidence and sideslip angles for velocity in knots:

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

```
alpha =
```

```
0.1194
```

```
0.3805
```

```
beta =
```

```
0.3780
```

```
0.0926
```

**See Also**      airspeed, machnumber

# angle2dcm

---

**Purpose** Create direction cosine matrix from rotation angles

**Syntax**  
`n = angle2dcm(r1, r2, r3)`  
`n = angle2dcm(r1, r2, r3, s)`

**Description** `n = angle2dcm(r1, r2, r3)` calculates the direction cosine matrix, `n`, for a given set of rotation angles, `r1`, `r2`, `r3`. `r1` is an `m` array of first rotation angles. `r2` is an `m` array of second rotation angles. `r3` is an `m` array of third rotation angles. `n` returns a 3-by-3-by-`m` matrix containing `m` direction cosine matrices. Rotation angles are input in radians.

`n = angle2dcm(r1, r2, r3, s)` calculates the direction cosine matrix, `n`, for a given set of rotation angles, `r1`, `r2`, `r3`, and a specified rotation sequence, `s`.

The default rotation sequence is 'ZYX', where `r1` is z-axis rotation, `r2` is y-axis rotation, and `r3` is x-axis rotation.

Supported rotation sequence strings are 'ZYX', 'YZY', 'ZXY', 'XZX', 'YXZ', 'YXY', 'YZX', 'YZY', 'XYZ', 'XYX', 'XZY', and 'XZX'.

**Examples** Determine the direction cosine matrix from rotation angles:

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

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

Determine the direction cosine matrix from multiple rotation angles:

```
yaw = [0.7854 0.5];  
pitch = [0.1 0.3];  
roll = [0 0.1];
```

```
dcm = angle2dcm( pitch, roll, yaw, 'YXZ' )
```

```
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
```

## See Also

`angle2dcm`, `dcm2angle`, `dcm2quat`, `quat2dcm`, `quat2angle`

# angle2quat

---

**Purpose** Convert rotation angles to quaternion

**Syntax**  
`q = angle2quat(r1,r2,r3)`  
`q = angle2quat(r1,r2,r3,s)`

**Description** `q = angle2quat(r1,r2,r3)` calculates the quaternion, `q`, for the three rotation angles, `r1`, `r2`, `r3`. `q` returns an `m`-by-4 matrix containing `m` quaternions. `q` has its scalar number as the first column. Rotation angles are input in radians.

`r1`            `m` array of first rotation angles.

`r2`            `m` array of second rotation angles.

`r3`            `m` array of third rotation angles.

`q = angle2quat(r1,r2,r3,s)` calculates the quaternion, `q`, for a given set of rotation angles, `r1`, `r2`, `r3`, and a specified rotation sequence, `s`.

The default rotation sequence is 'ZYX', where `r1` is z-axis rotation, `r2` is y-axis rotation, and `r3` is x-axis rotation.

Supported rotation sequence strings, `s`, are 'ZYX', 'YZZ', 'ZXY', 'ZXZ', 'YXZ', 'YXY', 'YZX', 'YZY', 'XYZ', 'XYX', 'XZY', and 'XZX'.

## Examples

Determine the quaternion from rotation angles:

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

Determine the quaternion from multiple rotation angles:

```
yaw = [0.7854 0.5];  
pitch = [0.1 0.3];  
roll = [0 0.1];
```

```
q = angle2quat(pitch, roll, yaw, 'YZZ')
q =
    0.9227    0.0191    0.0462    0.3822
    0.9587    0.0848    0.1324    0.2371
```

## See Also

[angle2dcm](#), [dcm2angle](#), [dcm2quat](#), [quat2angle](#), [quat2dcm](#)

**Purpose** Use 1976 COESA model

**Syntax** `[T, a, P, rho] = atmoscoesa(h, action)`

**Description** `[T, a, P, rho] = atmoscoesa(h, action)` implements the mathematical representation of the 1976 Committee on Extension to the Standard Atmosphere (COESA) United States standard lower atmospheric values for absolute temperature, pressure, density, and speed of sound for the input geopotential altitude.

Inputs for `atmoscoesa` are:

<code>h</code>	An array of <code>m</code> geopotential heights, in meters
<code>action</code>	A string to determine action for out-of-range input. Specify if out-of-range input invokes a 'Warning', 'Error', or no action ('None'). The default is 'Warning'.

Outputs calculated for the COESA model are:

<code>T</code>	An array of <code>m</code> temperatures, in kelvin
<code>a</code>	An array of <code>m</code> speeds of sound, in meters per second
<code>P</code>	An array of <code>m</code> air pressures, in pascal
<code>rho</code>	An array of <code>m</code> air densities, in kilograms per meter cubed

**Examples** Calculate the COESA model at 1000 meters with warnings for out-of-range inputs:

```
[T, a, P, rho] = atmoscoesa(1000)
```



T =  
281.6500

a =  
336.4341

P =  
8.9875e+004

rho =  
1.1116

Calculate the COESA model at 1000, 11,000, and 20,000 meters with errors for out-of-range inputs:

[T, a, P, rho] = atmoscoesa([1000 11000 20000], 'Error')

T =  
281.6500 216.6500 216.6500

a =  
336.4341 295.0696 295.0696

P =

1.0e+004 \*

8.9875    2.2632    0.5475

rho =

1.1116    0.3639    0.0880

## **Assumptions and Limitations**

Below the geopotential altitude of 0 m (0 feet) and above the geopotential altitude of 84,852 m (approximately 278,386 feet), temperature values are extrapolated linearly and pressure values are extrapolated logarithmically. Density and speed of sound are calculated using a perfect gas relationship.

## **References**

*U.S. Standard Atmosphere*, 1976, U.S. Government Printing Office, Washington, D.C.

## **See Also**

atmosisa, atmospalt, atmosnonstd, atmospalt

**Purpose** Use COSPAR International Reference Atmosphere 1986 model

**Syntax** [T alt zwind] = atmoscira(lat, ctype, coord, mtype, month, action)

**Description** [T alt zwind] = atmoscira(lat, ctype, coord, mtype, month, action) implements the mathematical representation of the Committee on Space Research (COSPAR) International Reference Atmosphere (CIRA) from 1986 model. The CIRA 1986 model provides a mean climatology of temperature, zonal wind, and geopotential height or pressure with nearly pole-to-pole coverage (80 degrees S to 80 degrees N) for 0 to 120 kilometers, encompassing the troposphere, middle atmosphere, and lower thermosphere. You can use this mathematical representation as a function of pressure or geopotential height.

Inputs for atmoscira are:

lat	An array of m geopotential heights, in meters.
ctype	A string to determine representation of coordinate type. Specify: <ul style="list-style-type: none"> <li>• 'Pressure' <p>Uses pressure in pascal.</p> </li> <li>• 'GPHeight' <p>Uses geopotential height in meters.</p> </li> </ul>
coord	Depending on the value of ctype, this argument specifies one of the following arrays: <ul style="list-style-type: none"> <li>• m pressures in pascal</li> <li>• m geopotential height in meters</li> </ul>

<code>mtype</code>	<p>A string that selects one of the following mean value types:</p> <ul style="list-style-type: none"><li>• 'Monthly' Monthly values. This is the default.</li><li>• 'Annual' Annual values. Valid when <code>ctype</code> has a value of 'Pressure'.</li></ul>
<code>month</code>	<p>A scalar value that selects the month in which the mean values are taken. This argument applies only when <code>mtype</code> has a value of 'Monthly'. Valid are 1 (January) through 12 (December). The default value is 1.</p>
<code>action</code>	<p>A string to determine action for out-of-range input. Specify if out-of-range input invokes a 'Warning', 'Error', or no action ('None'). The default is 'Warning'.</p>

Outputs calculated for the CIRA 1986 model are:

<code>T</code>	<p>An array of temperatures.</p> <ul style="list-style-type: none"><li>• If <code>m</code> is 'Monthly', an array of <code>m</code> temperatures, in kelvin.</li><li>• If <code>mtype</code> is 'Annual', an array of <code>m</code>-by-7 values. See "T Array if <code>mtype</code> is Annual" on page 4-45 for a description of the values in this array.</li></ul>
<code>alt</code>	<p>If <code>mtype</code> is 'Monthly', an array of <code>m</code> geopotential heights or <code>m</code> air pressures.</p>

- If `ctype` is 'Pressure'

This is an array `m` geopotential heights.

- If `ctype` is 'GPHeight'

This is an array `m` air pressure.

If `mtype` is 'Annual', an array of `m`-by-7 values for geopotential heights. This array is defined only for the northern hemisphere (`lat` is greater than 0). See “alt Array if `mtype` is Annual” on page 4-46 for a description of the values in this array.

`zwind`

An array of zonal winds:

- If `mtype` is 'Monthly'

This is an array in meters per second

- If `mtype` is 'Annual', an array of `m`-by-7 values. See “`zwind` if `mtype` is Annual” on page 4-46 for a description of the values in this array.

### **T Array if `mtype` is Annual**

- If `mtype` is 'Annual'

This is an array of `m`-by-7 values. These values are:

- 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

### **alt Array if mtype is Annual**

If mtype is 'Annual', alt is array of m-by-7 with the following values:

- 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

### **zwind if mtype is Annual**

If mtype is 'Annual', zwind is an array of m-by-7 with the following values:

- Annual mean zonal winds in meters per second
- Annual zonal winds cycle amplitude in meters per second
- Annual zonal winds cycle phase in month of maximum
- Semiannual zonal winds cycle amplitude in meters per second
- Semiannual zonal winds cycle phase in month of maximum
- Terannual zonal winds cycle amplitude in meters per second
- Terannual zonal winds cycle phase in month of maximum

## **Examples**

Calculate the mean monthly values for temperature (T), geopotential height (alt), and zonal wind (zwind) using the CIRA 1986 model at 45 degrees latitude and 101,300 pascal for January with out-of-range actions generating warnings:

```
[T alt zwind] = atmoscira( 45, 'Pressure', 101300 )
T =
    280.6000
alt =
    -18
zwind =
    3.3000
```

Calculate the mean monthly values for temperature (T), pressure (pres), and zonal wind (zwind) using the CIRA 1986 model at 45 degrees latitude and 20,000 meters for October with out-of-range actions generating warnings:

```
[T pres zwind] = atmoscira( 45, 'GPHeight', 20000, 'Monthly', 10)
T =
    215.8500
pres =
    5.5227e+003
zwind =
    9.5000
```

Calculate the mean monthly values for temperature (T), pressure (pres), and zonal wind (zwind) using the CIRA 1986 model at 45 and -30 degrees latitude and 20,000 meters for October with out-of-range actions generating errors:

```
[T pres zwind] = atmoscira( [45 -30], 'GPHeight', 20000, 10, 'error')
T =
    215.8500  213.9000
pres =
    1.0e+003 *
    5.5227  5.6550
zwind =
    9.5000  4.3000
```

Calculate the mean monthly values for temperature (T), geopotential height (alt), and zonal wind (zwind) using the CIRA 1986 model at 45

degrees latitude and 2000 pascal and at -30 degrees latitude and 101,300 pascal for September with out-of-range actions generating warnings:

```
[T alt zwind] = atmoscira( [45 -30], 'Pressure', [2000 101300], 9)
T =
    223.5395    290.9000
alt =
    1.0e+004 *
    2.6692    0.0058
zwind =
    0.6300   -1.1000
```

Calculate the annual values for temperature (T), geopotential height (alt), and zonal wind (zwind) using the CIRA 1986 model at 45 degrees latitude and 2000 pascal with out-of-range actions generating warnings:

```
[T alt zwind] = atmoscira( 45, 'Pressure', 2000, 'Annual' )
T =
    221.9596    5.0998    6.5300    1.9499    1.3000    1.0499    1.3000
alt =
    1.0e+004 *
    2.6465    0.0417    0.0007    0.0087    0.0001    0.0015    0.0002
zwind =
    4.6099    14.7496    0.6000    1.6499    4.6000    0.5300    1.4000
```

## Assumptions and Limitations

This function uses a corrected version of the CIRA data files provided by J. Barnett in July 1990 in ASCII format.

This function has the limitations of the CIRA 1986 model. The values for the CIRA 1986 model are limited to the regions of 80 degrees S to 80 degrees N on the Earth and geopotential heights of 0 to 120 kilometers. In each monthly mean data set, values at 80 degrees S for 101,300 pascal or 0 meters were omitted because these levels are within the Antarctic land mass. Zonal mean pressure in constant altitude coordinates: pressure data was not available below 20 kilometers. Therefore, this is the bottom level of the CIRA climatology.



**References**

Fleming, E. L., Chandra, S., Shoerberl, M. R., Barnett, J. J., *Monthly Mean Global Climatology of Temperature, Wind, Geopotential Height and Pressure for 0-120 km*, NASA TM100697, February 1988

<http://modelweb.gsfc.nasa.gov/atmos/cospar1.html>

**See Also**

atmoscoesa, atmosisa, atmoslapse, atmosnonstd, atmospalt

# atmosisa

---

**Purpose** Use International Standard Atmosphere model

**Syntax** `[T, a, P, rho] = atmosisa(h)`

**Description** `[T, a, P, rho] = atmosisa(h)` implements the mathematical representation of the International Standard Atmosphere values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude.

Input required by `atmosisa` is:

`h` An array of `m` geopotential heights, in meters

Outputs calculated for the International Standard Atmosphere are:

`T` An array of `m` temperatures, in kelvin

`a` An array of `m` speeds of sound, in meters per second

`P` An array of `m` air pressures, in pascal

`rho` An array of `m` air densities, in kilograms per meter cubed

**Examples** Calculate the International Standard Atmosphere at 1000 meters:

```
[T, a, P, rho] = atmosisa(1000)
```

```
T =
```

```
281.6500
```

```
a =
```

```
336.4341
```

P =

8.9875e+004

rho =

1.1116

Calculate the International Standard Atmosphere at 1000, 11,000, and 20,000 meters:

[T, a, P, rho] = atmosisa([1000 11000 20000])

T =

281.6500 216.6500 216.6500

a =

336.4341 295.0696 295.0696

P =

1.0e+004 \*

8.9875 2.2632 0.5475

rho =

1.1116 0.3639 0.0880

**Assumptions  
and  
Limitations**

Below the geopotential altitude of 0 km and above the geopotential altitude of the tropopause, temperature and pressure values are held. Density and speed of sound are calculated using a perfect gas relationship.

**References**

*U.S. Standard Atmosphere*, 1976, U.S. Government Printing Office, Washington, D.C.

**See Also**

atmoscoesa, atmoslapse, atmosnonstd, atmospalt

**Purpose** Use Lapse Rate Atmosphere model

**Syntax** `[T, a, P, rho] = atmoslapse(h, g, gamma, r, l, hts, htp, rho0, p0, t0)`

**Description** `[T, a, P, rho] = atmoslapse(h, g, gamma, r, l, hts, htp, rho0, p0, t0)` implements the mathematical representation of the lapse rate atmospheric equations for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude. This atmospheric model is customizable by specifying the atmospheric properties in the function input.

Inputs required by `atmoslapse` are:

<code>h</code>	An array of <code>m</code> geopotential heights, in meters
<code>g</code>	A scalar of acceleration due to gravity, in meters per second squared
<code>gamma</code>	A scalar of specific heat ratio
<code>r</code>	A scalar of characteristic gas constant, in joule per kilogram-kelvin
<code>l</code>	A scalar of lapse rate, in kelvin per meter
<code>hts</code>	A scalar of height of troposphere, in meters
<code>htp</code>	A scalar of height of tropopause, in meters
<code>rho0</code>	A scalar of air density at mean sea level, in kilograms per meter cubed
<code>p0</code>	A scalar of static pressure at mean sea level, in pascal
<code>t0</code>	A scalar of absolute temperature at mean sea level, in kelvin

Outputs calculated for the lapse rate atmosphere are:

# atmoslapse

---

T	An array of m temperatures, in kelvin
a	An array of m speeds of sound, in meters per second
P	An array of m air pressures, in pascal
rho	An array of m air densities, in kilograms per meter cubed

## Examples

Calculate the atmosphere at 1000 meters with the International Standard Atmosphere input values:

```
[T, a, P, rho] = atmoslapse(1000, 9.80665, 1.4, 287.0531, 0.0065, ...  
    11000, 20000, 1.225, 101325, 288.15 )
```

T =

281.6500

a =

336.4341

P =

8.9875e+004

rho =

1.1116

**Assumptions  
and  
Limitations**

Below the geopotential altitude of 0 km and above the geopotential altitude of the tropopause, temperature and pressure values are held. Density and speed of sound are calculated using a perfect gas relationship.

**References**

*U.S. Standard Atmosphere*, 1976, U.S. Government Printing Office, Washington, D.C.

**See Also**

atmoscoesa, atmosisa, atmosnonstd, atmospalt

# atmosnrlmsise00

---

## Purpose

Implement mathematical representation of 2001 United States Naval Research Laboratory Mass Spectrometer and Incoherent Scatter Radar Exosphere

## Syntax

```
[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec)
[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, lst)
[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, f107a,
    f107, aph)
[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, flags)
[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, otype)
[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec,
    action)
```

## Description

[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec) and implement the mathematical representation of the 2001 United States Naval Research Laboratory Mass Spectrometer and Incoherent Scatter Radar Exosphere (NRLMSISE-00). NRLMSISE-00 calculates the neutral atmosphere empirical model from the surface to lower exosphere (0 to 1,000,000 meters) with the option of including contributions from anomalous oxygen which can affect satellite drag above 500,000 meters.

[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, lst) lets you specify an array of *m* local apparent solar time (hours).

[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, f107a, f107, aph) lets you specify an array of *m* 81 day average of F10.7 flux (centered on doy), an array of *m* daily F10.7 flux for previous day, and an array of *m*-by-7 of magnetic index information.

[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, flags) lets you specify an array of 23 to enable or disable particular variations for the outputs.

[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, otype) lets you specify a string for total mass density output.

[T rho] = atmosnrlmsise00(h, lat, lon, year, doy, sec, action) lets you specify out-of-range input action.

Inputs for atmosnrlmsise00 are:



---

h	An array of $m$ altitudes, in meters.
lat	An array of $m$ geodetic latitudes, in meters.
long	An array of $m$ longitudes, in degrees.
year	Depending on the value of <code>ctype</code> , this argument specifies one of the following arrays: <ul style="list-style-type: none"><li>• <math>m</math> pressures in pascal</li><li>• <math>m</math> geopotential height in meters</li></ul>
doy	An array $m$ day of year.
sec	An array of $m$ seconds in day in universal time (UT)
lst	An array of $m$ local apparent solar time (hours). To obtain a physically realistic value, <code>lst</code> is set to $(\text{sec}/3600 + \text{lon}/15)$ by default. See “Limitations” on page 4-61 for more information.
F107a	An array of $m$ 81 day average of F10.7 flux (centered on day of year ( <code>doy</code> )). If F107A is input, F107 and <code>aph</code> must also be input. The effects of F107A are neither large nor well established below 80,000 meters; therefore, the default value is set to 150. See “Limitations” on page 4-61 for more information.
F107	An array of $m$ daily F10.7 flux for previous day. If F107 is input, F107A and <code>aph</code> must also be input. The effects of F107 are neither large nor well established below 80,000 meters; therefore, the default value is set to 150. See “Limitations” on page 4-61 for more information.

aph	An array of m-by-7 of magnetic index information. If aph is input, F107a and F107 must also be input. 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 prior to current time, and average of eight 3 hour AP indices from 36 to 57 hours prior to current time. The effects of daily magnetic index are neither large nor well established below 80,000 meters. As a result, the default value is set to 4. See "Limitations" on page 4-61 for more information.
flags	An array of 23 to enable or disable particular variations for the outputs. See following table.
otype	A string for total mass density output.  `Oxygen` Total mass density outputs include anomalous oxygen number density.  `NoOxygen` Total mass density outputs do not include anomalous oxygen number density.
action	A string to determine action for out-of-range input. Specify if out-of-range input invokes a 'Warning', 'Error', or no action ('None'). The default is 'Warning'.

These flags, associated with the eleventh input, enable or disable particular variations for the outputs.

Field	Description
Flags(1)	F10.7 effect on mean

Field	Description
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 block uses the entire matrix of magnetic index information (APH) instead of APH(:, 1)
Flags(10)	All UT, longitudinal effects
Flags(11)	Longitudinal
Flags(12)	UT and mixed UT, longitudinal
Flags(13)	Mixed AP, UT, longitudinal
Flags(14)	Terdiurnal
Flags(15)	Departures from diffusive equilibrium
Flags(16)	All exospheric temperature variations
Flags(17)	All variations from 120,000 meter temperature (TLB)
Flags(18)	All lower thermosphere (TN1) temperature variations
Flags(19)	All 120,000 meter gradient (S) variations
Flags(20)	All upper stratosphere (TN2) temperature variations
Flags(21)	All variations from 120,000 meter values (ZLB)
Flags(22)	All lower mesosphere temperature (TN3) variations
Flags(23)	Turbopause scale height variations

Outputs calculated for the neutral atmosphere empirical model are:

Field	Description
T	An array of N-by-2 values of temperature, in kelvin. The first column is exospheric temperature in kelvin, the second column is temperature at altitude, in kelvin.
rho	<p>An array of N-by-9 values of densities (kg/m<sup>3</sup> or 1/m<sup>3</sup>) in selected density units. The column order is:</p> <ul style="list-style-type: none"> <li>Density of He, in 1/m<sup>3</sup></li> <li>Density of O, in 1/m<sup>3</sup></li> <li>Density of N<sub>2</sub>, in 1/m<sup>3</sup></li> <li>Density of O<sub>2</sub>, in 1/m<sup>3</sup></li> <li>Density of Ar, in 1/m<sup>3</sup></li> <li>Total mass density, in 1/m<sup>3</sup></li> <li>Density of H, in 1/m<sup>3</sup></li> <li>Density of N, in 1/m<sup>3</sup></li> <li>Anomalous oxygen number density, in 1/m<sup>3</sup></li> </ul> <p>rho(6), total mass density, is defined as the sum of the mass densities of He, O, N<sub>2</sub>, O<sub>2</sub>, Ar, H, and N. Optionally, rho(6) can include the mass density of anomalous oxygen making rho(6), the effective total mass density for drag.</p>

## Examples

Calculate the temperatures, densities not including anomalous oxygen using the NRLMSISE-00 model at 10,000 meters, 45 degrees latitude, -50 degrees longitude, on January 4, 2007 at 0 UT using 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)
```

Calculate the temperatures, densities not including anomalous oxygen using the NRLMSISE-00 model at 10,000 meters, 45 degrees latitude, -50 degrees longitude, and at 25,000 meters, 47 degrees latitude, -55 degrees longitude on January 4, 2007 at 0 UT using 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])
```

Calculate the temperatures, densities including anomalous oxygen using the NRLMSISE-00 model at 10,000 meters, 45 degrees latitude, -50 degrees longitude, on January 4, 2007 at 0 UT using 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')
```

Calculate the temperatures, densities including anomalous oxygen using the NRLMSISE-00 model at 100,000 meters, 45 degrees latitude, -50 degrees longitude, on January 4, 2007 at 0 UT using defined values for flux, and magnetic index data, and default local solar time with out of range actions generating 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')
```

## Limitations

If `flags` array length, `m`, is 23 and all available inputs are not specified, this function assumes that `flags` is set.

This function has the limitations of the NRLMSISE-00 model. For more information, see the NRLMSISE-00 model documentation.

# atmosnrlmsise00

---

The NRLMSISE-00 model uses `sec`, `lst`, and `lon` 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:

$$\text{lst} = \text{sec}/3600 + \text{lon}/15$$

Departures from this equation for `lst` can be included if available but are of minor importance.

The F107 and F107A values that are used to 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. The following site provides both classes of values:  
[ftp://ftp.ngdc.noaa.gov/STP/SOLAR\\_DATA/SOLAR\\_RADIO/FLUX/](ftp://ftp.ngdc.noaa.gov/STP/SOLAR_DATA/SOLAR_RADIO/FLUX/)

## References

<http://ccmc.gsfc.nasa.gov/modelweb/atmos/nrlmsise00.html>

## See Also

`atmoscira`

---

<b>Purpose</b>	Use climatic data from MIL-STD-210 or MIL-HDBK-310						
<b>Syntax</b>	<code>[T, a, P, rho] = atmosnonstd(h, atype, extreme, freq, extalt, action, spec)</code>						
<b>Description</b>	<p><code>[T, a, P, rho] = atmosnonstd(h, atype, extreme, freq, extalt, action, spec)</code> 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) for absolute temperature, pressure, density, and speed of sound for the input geopotential altitude.</p> <p>Inputs for <code>atmosnonstd</code> are:</p> <table><tr><td><code>h</code></td><td>An array of <code>m</code> geopotential heights, in meters</td></tr><tr><td><code>atype</code></td><td>A string selecting the representation of 'Profile' or 'Envelope' for the atmospheric data. 'Profile' is the realistic atmospheric profiles associated with extremes at specified altitudes. 'Profile' is recommended for simulation of vehicles vertically traversing the atmosphere, or when the total influence of the atmosphere is needed. 'Envelope' uses extreme atmospheric values at each altitude. 'Envelope' is recommended for vehicles traversing the atmosphere horizontally, without much change in altitude.</td></tr><tr><td><code>extreme</code></td><td>A string selecting the atmospheric parameter which is the extreme value. Atmospheric parameters that can be specified are 'High temperature', 'Low temperature', 'High density', 'Low density', 'High pressure' and 'Low pressure'. 'High pressure' and 'Low pressure' are available only when <code>atype</code> is 'Envelope'.</td></tr></table>	<code>h</code>	An array of <code>m</code> geopotential heights, in meters	<code>atype</code>	A string selecting the representation of 'Profile' or 'Envelope' for the atmospheric data. 'Profile' is the realistic atmospheric profiles associated with extremes at specified altitudes. 'Profile' is recommended for simulation of vehicles vertically traversing the atmosphere, or when the total influence of the atmosphere is needed. 'Envelope' uses extreme atmospheric values at each altitude. 'Envelope' is recommended for vehicles traversing the atmosphere horizontally, without much change in altitude.	<code>extreme</code>	A string selecting the atmospheric parameter which is the extreme value. Atmospheric parameters that can be specified are 'High temperature', 'Low temperature', 'High density', 'Low density', 'High pressure' and 'Low pressure'. 'High pressure' and 'Low pressure' are available only when <code>atype</code> is 'Envelope'.
<code>h</code>	An array of <code>m</code> geopotential heights, in meters						
<code>atype</code>	A string selecting the representation of 'Profile' or 'Envelope' for the atmospheric data. 'Profile' is the realistic atmospheric profiles associated with extremes at specified altitudes. 'Profile' is recommended for simulation of vehicles vertically traversing the atmosphere, or when the total influence of the atmosphere is needed. 'Envelope' uses extreme atmospheric values at each altitude. 'Envelope' is recommended for vehicles traversing the atmosphere horizontally, without much change in altitude.						
<code>extreme</code>	A string selecting the atmospheric parameter which is the extreme value. Atmospheric parameters that can be specified are 'High temperature', 'Low temperature', 'High density', 'Low density', 'High pressure' and 'Low pressure'. 'High pressure' and 'Low pressure' are available only when <code>atype</code> is 'Envelope'.						

freq	A string selecting percent of time the extreme values would occur. Valid values for freq include 'Extreme values', '1%', '5%', '10%', and '20%'. 'Extreme values', '5%', and '20%' are available only when atype is 'Envelope'. When using atype of 'Envelope' and freq of '5%', '10%', and '20%', only the extreme parameter selected (temperature, density, or pressure) has a valid output. All other parameter outputs are zero.
extalt	A scalar value, in kilometers, selecting geometric altitude at which the extreme values occur. extalt applies only when atype is 'Profile'. Valid values for extalt include 5 (16404 ft), 10 (32808 ft), 20 (65617 ft), 30 (98425 ft), and 40 (131234 ft).
action	A string to determine action for out-of-range input. Specify if out-of-range input invokes a 'Warning', 'Error', or no action ('None'). The default is 'Warning'.
spec	A string specifying the atmosphere model, MIL-STD-210C or MIL-HDBK-310: '210c' or '310'. The default is '310'.

Outputs calculated for the lapse rate atmosphere are:

T	An array of $m$ temperatures, in kelvin
a	An array of $m$ speeds of sound, in meters per second
P	An array of $m$ air pressures, in pascal
rho	An array of $m$ air densities, in kilograms per meter cubed



## Examples

Calculate the nonstandard atmosphere profile with high density occurring 1% of the time at 5 kilometers from MIL-HDBK-310 at 1000 meters with warnings for out-of-range inputs:

```
[T, a, P, rho] = atmosnonstd( 1000, 'Profile', 'High density', '1%', 5 )
```

T =

248.1455

a =

315.7900

P =

8.9893e+004

rho =

1.2620

Calculate the nonstandard atmosphere envelope with high pressure occurring 20% of the time from MIL-STD-210C at 1000, 11,000, and 20,000 meters with errors for out-of-range inputs:

```
[T, a, P, rho] = atmosnonstd([1000 11000 20000], 'Envelope', ...
                              'High pressure', '20%', 'Error', '210c' )
```

T =

0 0 0

a =

0 0 0

P =

1.0e+004 \*  
9.1598 2.5309 0.6129

rho =

0 0 0

## Assumptions and Limitations

All values are held below the geometric altitude of 0 m (0 feet) and above the geometric altitude of 80,000 meters (approximately 262,000 feet). The envelope atmospheric model has a few exceptions where values are held below the geometric altitude of 1 kilometer (approximately 3281 feet) and above the geometric altitude of 30,000 meters (approximately 98,425 feet). These exceptions are due to lack of data in MIL-STD-210 or MIL-HDBK-310 for these conditions.

In general, temperature values are interpolated linearly and density values are interpolated logarithmically. Pressure and speed of sound are calculated using a perfect gas relationship. The envelope atmospheric model has a few exceptions where the extreme value is the only value provided as an output. Pressure in these cases is interpolated logarithmically. These envelope atmospheric model exceptions apply to all cases of high and low pressure, high and low temperature, and high and low density, excluding the extreme values and 1% frequency of occurrence. These exceptions are due to lack of data in MIL-STD-210 or MIL-HDBK-310 for these conditions.

A limitation is that climatic data for the region south of 60 degrees S latitude is excluded from consideration in MIL-STD-210 or MIL-HDBK-310.

This function uses the metric version of data from the MIL-STD-210 or MIL-HDBK-310 specifications. A limitation of this is some inconsistent data between the metric and English data. Locations where these inconsistencies occur are within the envelope data for low density, low temperature, high temperature, low pressure, and high pressure. The most noticeable differences occur in the following values:

- For low density envelope data with 5% frequency, the density values in metric units are inconsistent at 4 km and 18 km and 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 and 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.

## References

*Global Climatic Data for Developing Military Products (MIL-STD-210C)*, 9 January 1987, Department of Defense, Washington, D.C.

*Global Climatic Data for Developing Military Products (MIL-HDBK-310)*, 23 June 1997, Department of Defense, Washington, D.C.

## See Also

atmoscoesa, atmosisa, atmoslapse, atmospalt

# atmospalt

---

**Purpose** Calculate pressure altitude based on ambient pressure

**Syntax** `h = atmospalt(p, action)`

**Description** `h = atmospalt(p, 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 also known as the mean sea level (MSL) altitude.

Inputs for `atmospalt` are:

<code>P</code>	An array of $m$ ambient pressures, in pascal
<code>action</code>	A string to determine action for out-of-range input. Specify if out-of-range input invokes a 'Warning', 'Error', or no action ('None'). The default is 'Warning'.

Output is:

<code>h</code>	An array of $m$ pressure altitudes or MSL altitudes, in meters
----------------	--

**Examples** Calculate the pressure altitude at a static pressure of 101,325 Pa with warnings for out-of-range inputs:

```
h = atmospalt(101325)
```

```
h =
```

```
0
```

Calculate the pressure altitude at static pressures of 101,325 and 26,436 Pa with errors for out-of-range inputs:

```
h = atmospalt([101325 26436], 'Error' )
```

```
h =
```

```
1.0e+004 *
```

```
0      1.0000
```

**Assumptions  
and  
Limitations**

Below the pressure of 0.3961 Pa (approximately 0.00006 psi) and above the pressure of 101,325 Pa (approximately 14.7 psi), altitude values are extrapolated logarithmically. Air is assumed to be dry and an ideal gas.

**References**

*U.S. Standard Atmosphere*, 1976, U.S. Government Printing Office, Washington, D.C.

**See Also**

atmoscoesa

# Body (Aero.Body)

---

**Purpose** Construct body object for use with animation object

**Syntax** `h = Aero.Body`

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

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

The animation object has the following properties:

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

See `Aero.Body` for further details.

**See Also** `Aero.Body`

**Purpose** Construct camera object for use with animation object

**Syntax** `h = Aero.Camera`

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

The animation object has the following properties:

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

See `Aero.Camera` for further details.

**See Also** `Aero.Camera`

# ClearTimer (Aero.FlightGearAnimation)

---

<b>Purpose</b>	Clear and delete timer for animation of FlightGear flight simulator
<b>Syntax</b>	<code>ClearTimer(h)</code> <code>h.ClearTimer</code>
<b>Description</b>	<code>ClearTimer(h)</code> and <code>h.ClearTimer</code> clear and delete the MATLAB timer for the animation of the FlightGear flight simulator.
<b>Examples</b>	Clear and delete the MATLAB timer for animation of the FlightGear animation object, h:  <pre>h = Aero.FlightGearAnimation h.SetTimer h.ClearTimer h.SetTimer('FGTimer')</pre>
<b>See Also</b>	<code>SetTimer</code>



**Purpose** Convert from acceleration units to desired acceleration units

**Syntax** `a = convacc(v, ui, uo)`

**Description** `a = convacc(v, ui, uo)` computes the conversion factor from specified input acceleration units, `ui`, to specified output acceleration units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'ft/s^2'	Feet per second squared
'm/s^2'	Meters per second squared
'km/s^2'	Kilometers per second squared
'in/s^2'	Inches per second squared
'km/h-s'	Kilometers per hour per second
'mph/s'	Miles per hour per second
'G''s'	g-units

**Examples** Convert three accelerations from feet per second squared to meters per second squared:

```
a = convacc([3 10 20], 'ft/s^2', 'm/s^2')
```

```
a =
```

```
0.9144    3.0480    6.0960
```

**See Also** `convang`, `convangacc`, `convangvel`, `convdensity`, `convforce`, `convlength`, `convmass`, `convpres`, `convtemp`, `convvel`

# convang

---

**Purpose** Convert from angle units to desired angle units

**Syntax** `a = convang(v, ui, uo)`

**Description** `a = convang(v, ui, uo)` computes the conversion factor from specified input angle units, `ui`, to specified output angle units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'deg'	Degrees
'rad'	Radians
'rev'	Revolutions

**Examples** Convert three angles from degrees to radians:

```
a = convang([3 10 20], 'deg', 'rad')
```

```
a =
```

```
0.0524    0.1745    0.3491
```

**See Also** `convacc`, `convangacc`, `convangvel`, `convdensity`, `convforce`, `convlength`, `convmass`, `convpres`, `convtemp`, `convvel`

**Purpose** Convert from angular acceleration units to desired angular acceleration units

**Syntax** `a = convangacc(v, ui, uo)`

**Description** `a = convangacc(v, ui, uo)` computes the conversion factor from specified input angular acceleration units, `ui`, to specified output angular acceleration units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

<code>'deg/s^2'</code>	Degrees per second squared
<code>'rad/s^2'</code>	Radians per second squared
<code>'rpm/s'</code>	Revolutions per minute per second

**Examples** Convert three angular accelerations from degrees per second squared to radians per second squared:

```
a = convangacc([0.3 0.1 0.5], 'deg/s^2', 'rad/s^2')
```

```
a =
```

```
0.0052    0.0017    0.0087
```

**See Also** `convacc`, `convang`, `convangvel`, `convdensity`, `convforce`, `convlength`, `convmass`, `convpres`, `convtemp`, `convvel`

# convangvel

---

**Purpose** Convert from angular velocity units to desired angular velocity units

**Syntax** `a = convangvel(v, ui, uo)`

**Description** `a = convangvel(v, ui, uo)` computes the conversion factor from specified input angular velocity units, `ui`, to specified output angular velocity units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'deg/s'	Degrees per second
'rad/s'	Radians per second
'rpm'	Revolutions per minute

**Examples** Convert three angular velocities from degrees per second to radians per second:

```
a = convangvel([0.3 0.1 0.5], 'deg/s', 'rad/s')
```

```
a =
```

```
0.0052    0.0017    0.0087
```

**See Also** `convacc`, `convang`, `convangacc`, `convdensity`, `convforce`, `convlength`, `convmass`, `convpres`, `convtemp`, `convvel`

**Purpose** Convert from density units to desired density units

**Syntax** `a = convdensity(v, ui, uo)`

**Description** `a = convdensity(v, ui, uo)` computes the conversion factor from specified input density units, `ui`, to specified output density units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m`-by-`n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'lbm/ft^3'	Pound mass per feet cubed
'kg/m^3'	Kilograms per meters cubed
'slug/ft^3'	Slugs per feet cubed
'lbm/in^3'	Pound mass per inch cubed

**Examples** Convert three densities from pound mass per feet cubed to kilograms per meters cubed:

```
a = convdensity([0.3 0.1 0.5], 'lbm/ft^3', 'kg/m^3')
```

```
a =
```

```
4.8055    1.6018    8.0092
```

**See Also** `convacc`, `convang`, `convangacc`, `convangvel`, `convforce`, `convlength`, `convmass`, `convpres`, `convtemp`, `convvel`

# convforce

---

**Purpose** Convert from force units to desired force units

**Syntax** `a = convforce(v, ui, uo)`

**Description** `a = convforce(v, ui, uo)` computes the conversion factor from specified input force units, `ui`, to specified output force units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

<code>'lbf'</code>	Pound force
<code>'N'</code>	Newton

**Examples** Convert three forces from pound force to newtons:

```
a = convforce([120 1 5], 'lbf', 'N')
```

```
a =
```

```
533.7866    4.4482    22.2411
```

**See Also** `convacc`, `convang`, `convangacc`, `convangvel`, `convdensity`, `convlength`, `convmass`, `convpres`, `convtemp`, `convvel`

**Purpose** Convert from length units to desired length units

**Syntax** `a = convlength(v, ui, uo)`

**Description** `a = convlength(v, ui, uo)` computes the conversion factor from specified input length units, `ui`, to specified output length units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'ft'	Feet
'm'	Meters
'km'	Kilometers
'in'	Inches
'mi'	Miles
'naut mi'	Nautical miles

**Examples** Convert three lengths from feet to meters:

```
a = convlength([3 10 20], 'ft', 'm')  
  
a =  
  
    0.9144    3.0480    6.0960
```

**See Also** `convacc`, `convang`, `convangacc`, `convangvel`, `convdensity`, `convforce`, `convmass`, `convpres`, `convtemp`, `convvel`

# convmass

---

**Purpose** Convert from mass units to desired mass units

**Syntax** `a = convmass(v, ui, uo)`

**Description** `a = convmass(v, ui, uo)` computes the conversion factor from specified input mass units, `ui`, to specified output mass units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'lbm'	Pound mass
'kg'	Kilograms
'slugs'	Slugs

**Examples** Convert three masses from pound mass to kilograms:

```
a = convmass([3 1 5], 'lbm', 'kg')
```

```
a =
```

```
1.3608    0.4536    2.2680
```

**See Also** `convacc`, `convang`, `convangacc`, `convangvel`, `convdensity`, `convforce`, `convlength`, `convpres`, `convtemp`, `convvel`



**Purpose** Convert from pressure units to desired pressure units

**Syntax** `a = convpres(v, ui, uo)`

**Description** `a = convpres(v, ui, uo)` computes the conversion factor from specified input pressure units, `ui`, to specified output pressure units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'psi'	Pound force per square inch
'Pa'	Pascal
'psf'	Ppound force per square foot
'atm'	Atmosphere

**Examples** Convert two pressures from pound force per square inch to atmospheres:

```
a = convpres([14.696 35], 'psi', 'atm')
```

```
a =
```

```
1.0000 2.3816
```

**See Also** `convacc`, `convang`, `convangacc`, `convangvel`, `convdensity`, `convforce`, `convlength`, `convmass`, `convtemp`, `convvel`

# convtemp

---

**Purpose** Convert from temperature units to desired temperature units

**Syntax** `a = convtemp(v, ui, uo)`

**Description** `a = convtemp(v, ui, uo)` computes the conversion factor from specified input temperature units, `ui`, to specified output temperature units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m-by-n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'K'	Kelvin
'F'	Degrees Fahrenheit
'C'	Degrees Celsius
'R'	Degrees Rankine

**Examples** Convert three temperatures from degrees Celsius to degrees Fahrenheit:

```
a = convtemp([0 100 15], 'C', 'F')
```

```
a =
```

```
32.0000 212.0000 59.0000
```

**See Also** `convacc`, `convang`, `convangacc`, `convangvel`, `convdensity`, `convforce`, `convlength`, `convmass`, `convpres`, `convvel`

**Purpose** Convert from velocity units to desired velocity units

**Syntax** `a = convvel(v, ui, uo)`

**Description** `a = convvel(v, ui, uo)` computes the conversion factor from specified input velocity units, `ui`, to specified output velocity units, `uo`, and applies the conversion factor to the input, `v`, to produce the output, `a`, in the desired units. `v` and `a` are floating-point arrays of size `m`-by-`n`. All of the values in `v` must have the same unit conversions from `ui` to `uo`. `ui` and `uo` are strings.

Supported unit strings are:

'ft/s'	Feet per second
'm/s'	Meters per second
'km/s'	Kilometers per second
'in/s'	Inches per second
'km/h'	Kilometers per hour
'mph'	Miles per hour
'kts'	Knots
'ft/min'	Feet per minute

**Examples** Convert three velocities from feet per minute to meters per second:

```
a = convvel([30 100 250], 'ft/min', 'm/s')
```

```
a =
```

```
0.1524    0.5080    1.2700
```

**See Also** `convacc`, `convang`, `convangacc`, `convangvel`, `convdensity`, `convforce`, `convlength`, `convmass`, `convpres`, `convtemp`

# correctairspeed

---

**Purpose** Calculate equivalent airspeed (EAS), calibrated airspeed (CAS), or true airspeed (TAS) from one of other two airspeeds

**Syntax** `as = correctairspeed(v, a, p0, ai, ao)`

**Description** `as = correctairspeed(v, a, p0, ai, ao)` computes the conversion factor from specified input airspeed, `ai`, to specified output airspeed, `ao`, using speed of sound, `a`, and static pressure `p0`. The conversion factor is applied to the input airspeed, `v`, to produce the output, `as`, in the desired airspeed. `v`, `as`, `a`, and `p0` are floating-point arrays of size `m`. All of the values in `v` must have the same airspeed conversions from `ai` to `ao`. `ai` and `ao` are strings.

Input required by `correctairspeed` is:

<code>v</code>	Airspeed in meters per second
<code>a</code>	Speed of sound in meters per second
<code>p0</code>	Static air pressure in pascal
<code>ai</code>	Input airspeed string
<code>ao</code>	Output airspeed string

Supported airspeed strings are:

<code>'TAS'</code>	True airspeed
<code>'CAS'</code>	Calibrated airspeed
<code>'EAS'</code>	Equivalent airspeed

Output, `as`, is calculated as airspeed in meters per second.

## Examples

Convert three airspeeds from true airspeed to equivalent airspeed at 1000 meters:

```
as = correctairspeed([25.7222; 10.2889; 3.0867], 336.4, 89874.6, 'TAS', 'EAS')
```

```
as =
```

```
24.5057
9.8023
2.9407
```

Convert airspeeds from true airspeed to equivalent airspeed at 1000 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, 'TAS', 'EAS')

as =
```

```
24.5057
10.2887
3.0866
```

## Assumptions and Limitations

Based on assumption of compressible, isentropic (subsonic flow), dry air with constant specific heat ratio ( $\gamma$ ).

## References

Lowry, J.T., *Performance of Light Aircraft*, AIAA Education Series, Washington, D.C., 1999

*Aeronautical Vestpocket Handbook*, United Technologies Pratt & Whitney, August, 1986

## See Also

airspeed

# datcomimport

---

**Purpose** 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)` takes a file name, `file`, as a string (or a cell array of file names as strings), and imports aerodynamic data from `file` into a cell array of structures, `aero`. Before reading the DATCOM file, the function initializes values to 99999 to show when there is not a full set of data for the DATCOM case.

`aero = datcomimport(file, usenan)` is an alternate method allowing the replacement of data points with NaN or zero where no DATCOM methods exist or where the method is not applicable. The default value for `usenan` is `true`.

`aero = datcomimport(file, usenan, verbose)` is an alternate method to display the status of the DATCOM file being read. The default value for `verbose` is 2, which displays a wait bar. Other options are 0, which displays no information, and 1, which displays text to the MATLAB Command Window.

`aero = datcomimport(file, usenan, verbose, filetype)` is an alternate method that allows you to specify which type of DATCOM file to read. The default value for `filetype` is 6, which reads the `for006.dat` file output by DATCOM. The other option is 21, which reads the `for021.dat` file output by DATCOM 2007.

---

**Note** If `filetype` is 21, the last entry in `aero` is a table the function reads breakpoints collected from all of the cases.

---

Fields for the 1976, 1999, and 2007 versions of the type 6 output files are described in the following:

- “Fields for 1976 Version” on page 4-87
- “Fields for 1999 Version” on page 4-106
- “Fields for 2007 Version” on page 4-111

Fields for 2007 version of the type 21 output file are described in

- “Fields for 2007 Version of Type 21 Output File” on page 4-116

### **Fields for 1976 Version**

The fields of aero depend on the data within the DATCOM file:

#### **Common Fields for the 1976 Version**

<b>Field</b>	<b>Description</b>	<b>Default</b>
case	String 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

## Common Fields for the 1976 Version (Continued)

Field	Description	Default
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	String denoting the specified system of units for the case.	'ft'
deriv	String 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



**Common Fields for the 1976 Version (Continued)**

<b>Field</b>	<b>Description</b>	<b>Default</b>
stype	Scalar denoting the type of asymmetric flap for the case.	[ ]
trim	Logical denoting the reading of trim data for the case. When trim runs are read, this value is set to true.	false
damp	Logical denoting the reading of dynamic derivative data for the case. When dynamic derivative runs are read, this value is set to true.	false
build	Scalar denoting the reading of build data for the case. When build runs are read, this value is set to 10.	1
part	Logical denoting the reading of partial data for the case. When partial runs are written for each Mach number, this value is set to true.	false
highsym	Logical denoting the reading of symmetric flap high-lift data for the case. When symmetric flap runs are read, this value is set to true.	false
highasy	Logical denoting the reading of asymmetric flap high-lift data for the case. When asymmetric flap runs are read, this value is set to true.	false
highcon	Logical denoting the reading of control/trim tab high-lift data for the case. When control/trim tab runs are read, this value is set to true.	false

## Common Fields for the 1976 Version (Continued)

Field	Description	Default
tjet	Logical denoting the reading of transverse-jet control data for the case. When transverse-jet control runs are read, this value is set to true.	false
hypeff	Logical denoting the reading of hypersonic flap effectiveness data for the case. When hypersonic flap effectiveness runs are read, this value is set to true.	false
lb	Logical denoting the reading of low aspect ratio wing or lifting body data for the case. When low aspect ratio wing or lifting body runs are read, this value is set to true.	false
pwr	Logical denoting the reading of power effects data for the case. When power effects runs are read, this value is set to true.	false
grnd	Logical denoting the reading of ground effects data for the case. When ground effects runs are read, this value is set to true.	false
wsspn	Scalar denoting the semi-span theoretical panel for wing. This value is used to determine if the configuration contains a canard.	1

**Common Fields for the 1976 Version (Continued)**

<b>Field</b>	<b>Description</b>	<b>Default</b>
hsspn	Scalar denoting the semi-span theoretical panel for horizontal tail. This value is used to determine if the configuration contains a canard.	1
ndelta	Number of control surface deflections: delta, deltal, or deltar.	0
delta	Array of control-surface streamwise deflection angles.	[]
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.  <pre> config.downwash = false; config.body = false; config.wing = false; config.htail = false; config.vtail = false; config.vfin = false; </pre>
version	Version of DATCOM file.	1976

## Static Longitude and Lateral Stability Fields Available for the 1976 Version

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 with respect to alpha.	alpha, mach, alt, build, grndht, delta
cma	Derivatives of pitching-moment coefficients with respect to alpha.	alpha, mach, alt, build, grndht, delta
cyb	Derivatives of side-force coefficients with respect to sideslip angle.	alpha, mach, alt, build, grndht, delta
cnb	Derivatives of yawing-moment coefficients with respect to sideslip angle.	alpha, mach, alt, build, grndht, delta

**Static Longitude and Lateral Stability Fields Available for the 1976 Version  
(Continued)**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
clb	Derivatives of rolling-moment coefficients with respect 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 with respect to angle of attack.	alpha, mach, alt, build, grndht, delta

**Dynamic Derivative Fields for the 1976 Version**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
clq	Rolling-moment 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

## Dynamic Derivative Fields for the 1976 Version (Continued)

Field	Matrix of...	Function of...
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

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, will 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 will tend to rotate the flap trailing edge down.	delta, mach, alt
dcdi_sym	Incremental induced drag coefficients due to flap deflection.	alpha, delta, mach, alt

**High-Lift and Control Fields Available for Asymmetric Flaps for the 1976 Version**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
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**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
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

## High-Lift and Control Fields Available for Control/Trim Tabs for the 1976 Version (Continued)

Field	Matrix of...	Function of...
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

Field	Matrix of...	Function of...
cl_ustrim	Untrimmed lift coefficients, which are defined positive for an up-acting load.	alpha, mach, alt
cd_ustrim	Untrimmed drag coefficients, which are defined positive for an aft-acting load.	alpha, mach, alt
cm_ustrim	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



**High-Lift and Control Fields Available for Trim for the 1976 Version (Continued)**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
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

## High-Lift and Control Fields Available for Trim for the 1976 Version (Continued)

Field	Matrix of...	Function of...
hm_tailtrim	Trimmed stabilizer hinge-moment coefficients, which are defined positive for a stabilizer rotation with leading edge up and trailing edge down.	alpha, mach, alt
cl_trimi	Lift coefficients at trim incidence. These coefficients are defined positive for an up-acting load.	alpha, mach, alt
cd_trimi	Drag coefficients at trim incidence. These coefficients are defined positive for an aft-acting load.	alpha, mach, alt

## Transverse Jet Control Fields for the 1976 Version

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 strings 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

**Transverse Jet Control Fields for the 1976 Version (Continued)**

<b>Field</b>	<b>Description</b>	<b>Stored with Indices of...</b>
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**

<b>Field</b>	<b>Matrix of...</b>	<b>Stored with Indices of...</b>
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**

<b>Field</b>	<b>Matrix of...</b>	<b>Stored with Indices of...</b>
wetarea_b	Body wetted area.	mach, alt, number of runs

## Auxiliary and Partial Fields Available for the 1976 Version (Continued)

Field	Matrix of...	Stored with Indices of...
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

**Auxiliary and Partial Fields Available for the 1976 Version (Continued)**

<b>Field</b>	<b>Matrix of...</b>	<b>Stored with Indices of...</b>
gamma2pialpvr	$\gamma \cdot (2 \cdot \pi \cdot \alpha \cdot v \cdot r) \cdot t$	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
cmotheta w	$(cmo / \theta) \cdot w$	mach, alt
cmotheta h	$(cmo / \theta) \cdot h$	mach, alt
espeff	$(\epsilon_{soln}) \cdot eff$	alpha, mach, and alt
despdalpeff	$d(\epsilon_{soln}) / d(\alpha) \cdot eff$	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_****	$(clb / cl) \cdot mfb$ , where **** is either wb (wing-body) or bht (body-horizontal tail).	mach, alt.
cnam14_****	$(cna)_m = 1.4$ , where **** is either wb (wing-body) or bht (body-horizontal tail).	mach, alt

## Auxiliary and Partial Fields Available for the 1976 Version (Continued)

Field	Matrix of...	Stored with Indices of...
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)

**Auxiliary and Partial Fields Available for the 1976 Version (Continued)**

<b>Field</b>	<b>Matrix of...</b>	<b>Stored with Indices of...</b>
mac_ _ **	Mean aerodynamic chords, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
qcmac_ _ **	Quarter chord x (mac), where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
ymac_ _ **	y (mac), where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
cd0_ _ **	Zero lift drags, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)

## Auxiliary and Partial Fields Available for the 1976 Version (Continued)

Field	Matrix of...	Stored with Indices of...
friccoeff_*_**	Friction coefficients, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
cla_b_***	cla-b(***), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
cla_***_b	cla-***(b), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
k_b_***	k-b(***), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
k_***_b	k-***(b), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
xacc_b_***	xac/c-b(***), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
cdlcl2_***	cdl/cl <sup>2</sup> , where *** is either w (wing) or ht (stabilizer).	mach, alt
clbcl_***	clb/cl, where *** is either w (wing) or ht (stabilizer).	mach, alt



**Auxiliary and Partial Fields Available for the 1976 Version (Continued)**

<b>Field</b>	<b>Matrix of...</b>	<b>Stored with Indices of...</b>
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
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

### Common Fields for the 1999 Version

Field	Description	Default
case	String 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 version 1999 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.	[ ]

**Common Fields for the 1999 Version (Continued)**

<b>Field</b>	<b>Description</b>	<b>Default</b>
dim	String denoting the specified system of units for the case.	'ft'
deriv	String 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
hypreff	Logical denoting the reading of hypersonic data for the case. When hypersonic data is read, this value is set to true.	false

# datcomimport

## Common Fields for the 1999 Version (Continued)

Field	Description	Default
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.	As follows:  <pre> 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 = []; </pre>
version	Version of DATCOM file.	1999

## Static Longitude and Lateral Stability Fields Available for the 1999 Version

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

**Static Longitude and Lateral Stability Fields Available for the 1999 Version  
(Continued)**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
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 with respect to alpha.	alpha, mach, alt, build
cma	Derivatives of pitching-moment coefficients with respect to alpha.	alpha, mach, alt, build
cyb	Derivatives of side-force coefficients with respect to sideslip angle.	alpha, mach, alt, build
cnb	Derivatives of yawing-moment coefficients with respect to sideslip angle.	alpha, mach, alt, build
clb	Derivatives of rolling-moment coefficients with respect 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

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 Version

### Common Fields for the 2007 Version

Field	Description	Default
case	String 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 version 2007 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.	[ ]

## Common Fields for the 2007 Version (Continued)

Field	Description	Default
blref	Scalar denoting the lateral reference length.	[ ]
dim	String denoting the specified system of units for the case.	'ft'
deriv	String 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



**Common Fields for the 2007 Version (Continued)**

<b>Field</b>	<b>Description</b>	<b>Default</b>
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.	As follows:  <pre> 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 = []; </pre>
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 Version

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, mach, alt, 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 with respect to alpha.	alpha, mach, alt, build
cma	Derivatives of pitching-moment coefficients with respect to alpha.	alpha, mach, alt, build
cyb	Derivatives of side-force coefficients with respect to sideslip angle.	alpha, mach, alt, build
cnb	Derivatives of yawing-moment coefficients with respect to sideslip angle.	alpha, mach, alt, build
clb	Derivatives of rolling-moment coefficients with respect to sideslip angle.	alpha, mach, alt, build
clod	Ratios of lift coefficient to drag coefficient.	alpha, mach, alt, build

**Static Longitude and Lateral Stability Fields Available for the 2007 Version  
(Continued)**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
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 Version**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
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 Version of Type 21 Output File

### Common Fields for the 2007 Version of Type 21 Output File

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.	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 version 2007 of DATCOM is loop is 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.	As follows:  <pre>config.fin1.delta = zeros(1,8); config.fin2.delta = zeros(1,8); config.fin3.delta = zeros(1,8); config.fin4.delta = zeros(1,8);</pre>
version	Version of DATCOM file.	2007

**Static Longitude and Lateral Stability Fields Available for the 2007 Version of Type 21**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
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 Version of 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

**Dynamic Derivative Fields for the 2007 Version of Type 21 (Continued)**

<b>Field</b>	<b>Matrix of...</b>	<b>Function of...</b>
clnq	Yawing-moment due to pitch rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cllq	Rolling-moment due to pitch rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cnp	Yawing-moment derivatives due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cap	Axial-force due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cyp	Lateral-force derivatives due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
clnp	Yawing-moment due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta

## Dynamic Derivative Fields for the 2007 Version of Type 21 (Continued)

Field	Matrix of...	Function of...
c1lp	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
c1lr	Rolling-moment due to yaw rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta



## Examples

Read the 1976 version Digital DATCOM output file `astdatcom.out`:

```
aero = datcomimport('astdatcom.out')
```

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:

```
usenan = false;  
aero = datcomimport('astdatcom.out', usenan, 1 )
```

## Assumptions and Limitations

The operational limitations of the 1976 version DATCOM apply to the data contained in AERO. For more information on DATCOM limitations, see [1], section 2.4.5.

USAF Digital DATCOM data for wing section, horizontal tail section, vertical tail section and ventral fin section are not read.

## References

1. AFFDL-TR-79-3032: *The USAF Stability and Control DATCOM*, Volume 1, Users Manual
2. AFRL-VA-WP-TR-1998-3009: *MISSILE DATCOM*, Users Manual – 1997 FORTRAN 90 Revision

# dcm2alphabeta

---

**Purpose** Convert direction cosine matrix to angle of attack and sideslip angle

**Syntax** [a b] = dcm2alphabeta(n)

**Description** [a b] = dcm2alphabeta(n) calculates the angle of attack and sideslip angle, a and b, for a given direction cosine matrix, n. n is a 3-by-3-by-m matrix containing m orthogonal direction cosine matrices. a is an m array of angles of attack. b is an m array of sideslip angles. n performs the coordinate transformation of a vector in body-axes into a vector in wind-axes. Angles of attack and sideslip angles are output in radians.

**Examples** Determine the angle of attack and sideslip angle from direction cosine matrix:

```
dcm = [ 0.8926    0.1736    0.4162; ...
       -0.1574    0.9848   -0.0734; ...
       -0.4226     0         0.9063];
[alpha beta] = dcm2alphabeta(dcm)
```

```
alpha =

    0.4363
```

```
beta =

    0.1745
```

Determine the angle of attack and sideslip angle from multiple direction cosine matrices:

```
dcm = [ 0.8926    0.1736    0.4162; ...
       -0.1574    0.9848   -0.0734; ...
       -0.4226     0         0.9063];
dcm(:,:,2) = [ 0.9811    0.0872    0.1730; ...
              -0.0859    0.9962   -0.0151; ...
              -0.1736     0         0.9848];
```

```
[alpha beta] = dcm2alphabeta(dcm)
```

```
alpha =
```

```
    0.4363  
    0.1745
```

```
beta =
```

```
    0.1745  
    0.0873
```

## See Also

[angle2dcm](#), [dcm2angle](#), [dcmbody2wind](#)

# dcm2angle

---

**Purpose** Create rotation angles from direction cosine matrix

**Syntax**

```
[r1 r2 r3] = dcm2angle(n)
[r1 r2 r3] = dcm2angle(n, s)
[r1 r2 r3] = dcm2angle(n, s, lim)
```

**Description** [r1 r2 r3] = dcm2angle(n) calculates the set of rotation angles, r1, r2, r3, for a given direction cosine matrix, n. n is a 3-by-3-by-m matrix containing m direction cosine matrices. r1 returns an m array of first rotation angles. r2 returns an m array of second rotation angles. r3 returns an m array of third rotation angles. Rotation angles are output in radians.

[r1 r2 r3] = dcm2angle(n, s) calculates the set of rotation angles, r1, r2, r3, for a given direction cosine matrix, n, and a specified rotation sequence, s.

The default rotation sequence is 'ZYX', where r1 is z-axis rotation, r2 is y-axis rotation, and r3 is x-axis rotation.

Supported rotation sequence strings are 'ZYX', 'YZY', 'ZXY', 'ZXZ', 'YXZ', 'YXY', 'YZX', 'YZY', 'XYZ', 'XYX', 'XZY', and 'XZX'.

[r1 r2 r3] = dcm2angle(n, s, lim) calculates the set of rotation angles, r1, r2, r3, for a given direction cosine matrix, n, a specified rotation sequence, s, and a specified angle constraint, lim. lim is a string specifying either 'Default' or 'ZeroR3'. See “Assumptions and Limitations” on page 4-125 for full definitions of angle constraints.

**Examples** Determine the rotation angles from direction cosine matrix:

```
dcm = [1 0 0; 0 1 0; 0 0 1];
[yaw, pitch, roll] = dcm2angle( dcm )
yaw =

    0

pitch =
```

```

0
roll =
0

```

Determine the rotation angles from multiple direction cosine matrices:

```

dcm      = [ 1 0 0; 0 1 0; 0 0 1];
dcm(:,2) = [ 0.85253103550038  0.47703040785184 -0.21361840626067; ...
            -0.43212157513194  0.87319830445628  0.22537893734811; ...
            0.29404383655186 -0.09983341664683  0.95056378592206];
[pitch, roll, yaw] = dcm2angle( dcm, 'YXZ' )

pitch =

    0
0.3000

roll =

    0
0.1000

yaw =

    0
0.5000

```

## Assumptions and Limitations

The 'Default' limitations for the 'ZYX', 'ZXY', 'YXZ', 'YZX', 'XYZ', and 'XZY' implementations generate an r2 angle that lies between  $\pm 90$  degrees, and r1 and r3 angles that lie between  $\pm 180$  degrees.

The 'Default' limitations for the 'ZYZ', 'ZXZ', 'YXY', 'YZY', 'YXY', and 'XZX' implementations generate an r2 angle that lies between 0 and 180 degrees, and r1 and r3 angles that lie between  $\pm 180$  degrees.

## dcm2angle

---

The 'ZeroR3' limitations for the 'ZYX', 'ZXY', 'YXZ', 'YZX', 'XYZ', and 'XZY' implementations generate an r2 angle that lies between  $\pm 90$  degrees, and r1 and r3 angles that lie between  $\pm 180$  degrees. However, when r2 is  $\pm 90$  degrees, r3 is set to 0 degrees.

The 'ZeroR3' limitations for the 'ZYZ', 'ZXZ', 'YXY', 'YZY', 'YXX', and 'XZX' implementations generate an r2 angle that lies between 0 and 180 degrees, and r1 and r3 angles that lie between  $\pm 180$  degrees. However, when r2 is 0 or  $\pm 180$  degrees, r3 is set to 0 degrees.

### See Also

`angle2dcm`, `dcm2quat`, `quat2dcm`, `quat2angle`

**Purpose** Convert direction cosine matrix to geodetic latitude and longitude

**Syntax** `[lat lon] = dcm2latlon(n)`

**Description** `[lat lon] = dcm2latlon(n)` calculates the geodetic latitude and longitude, `lat` and `lon`, for a given direction cosine matrix, `n`. `n` is a 3-by-3-by-`m` matrix containing `m` orthogonal direction cosine matrices. `lat` is an `m` array of geodetic latitudes. `lon` is an `m` array of longitudes. `n` performs the coordinate transformation of a vector in Earth-centered Earth-fixed (ECEF) axes into a vector in north-east-down (NED) axes. Geodetic latitudes and longitudes are output in degrees.

**Examples** Determine the geodetic latitude and longitude from direction cosine matrix:

```
dcm = [ 0.3747    0.5997    0.7071; ...
        0.8480   -0.5299         0; ...
        0.3747    0.5997   -0.7071];
[lat lon] = dcm2latlon(dcm)
```

```
lat =

    44.9995
```

```
lon =

   -122.0005
```

Determine the geodetic latitude and longitude from multiple direction cosine matrices:

```
dcm = [ 0.3747    0.5997    0.7071; ...
        0.8480   -0.5299         0; ...
        0.3747    0.5997   -0.7071];
dcm(:,:,2) = [-0.0531    0.6064    0.7934; ...
              0.9962    0.0872         0; ...
```

# dcm2latlon

---

```
          -0.0691    0.7903    -0.6088];  
[lat lon] = dcm2latlon(dcm)
```

```
lat =
```

```
    44.9995  
    37.5028
```

```
lon =
```

```
 -122.0005  
  -84.9975
```

## See Also

[angle2dcm](#), [dcm2angle](#), [dcmecef2ned](#)



**Purpose** Convert direction cosine matrix to quaternion

**Syntax** `q = dcm2quat(n)`

**Description** `q = dcm2quat(n)` calculates the quaternion, `q`, for a given direction cosine matrix, `n`. Input `n` is a 3-by-3-by-`m` matrix of orthogonal direction cosine matrices. The direction cosine matrix performs the coordinate transformation of a vector in inertial axes to a vector in body axes. `q` returns an `m`-by-4 matrix containing `m` quaternions. `q` has its scalar number as the first column.

**Examples** Determine the quaternion from direction cosine matrix:

```
dcm = [0 1 0; 1 0 0; 0 0 1];
q = dcm2quat(dcm)
```

```
q =
```

```
    0.7071         0         0         0
```

Determine the quaternions from multiple direction cosine matrices:

```
dcm          = [ 0 1 0; 1 0 0; 0 0 1];
dcm(:, :, 2) = [ 0.4330    0.2500   -0.8660; ...
                0.1768    0.9186    0.3536; ...
                0.8839   -0.3062    0.3536];
q = dcm2quat(dcm)
```

```
q =
```

```
    0.7071         0         0         0
    0.8224    0.2006    0.5320    0.0223
```

**See Also** `angle2dcm`, `dcm2angle`, `angle2quat`, `quat2dcm`, `quat2angle`

# dcmbody2wind

---

**Purpose** Convert angle of attack and sideslip angle to direction cosine matrix

**Syntax** `n = dcmbody2wind(a, b)`

**Description** `n = dcmbody2wind(a, b)` calculates the direction cosine matrix, `n`, for given angle of attack and sideslip angle, `a`, `b`. `a` is an `m` array of angles of attack. `b` is an `m` array of sideslip angles. `n` returns a 3-by-3-by-`m` matrix containing `m` direction cosine matrices. `n` performs the coordinate transformation of a vector in body-axes into a vector in wind-axes. Angles of attack and sideslip angles are input in radians.

**Examples** Determine the direction cosine matrix from angle of attack and sideslip angle:

```
alpha = 0.4363;
beta = 0.1745;
dcm = dcmbody2wind(alpha, beta)
```

```
dcm =
```

```
    0.8926    0.1736    0.4162
   -0.1574    0.9848   -0.0734
   -0.4226         0    0.9063
```

Determine the direction cosine matrix from multiple angles of attack and sideslip angles:

```
alpha = [0.4363 0.1745];
beta = [0.1745 0.0873];
dcm = dcmbody2wind(alpha, beta)
```

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

```
    0.8926    0.1736    0.4162
   -0.1574    0.9848   -0.0734
   -0.4226         0    0.9063
```

```
dcm(:, :, 2) =
```

```
    0.9811    0.0872    0.1730  
   -0.0859    0.9962   -0.0151  
   -0.1736         0    0.9848
```

## See Also

`angle2dcm`, `dcm2alphabet`, `dcm2angle`

# dcmecef2ned

---

**Purpose** Convert geodetic latitude and longitude to direction cosine matrix

**Syntax** `n = dcmecef2ned(lat, lon)`

**Description** `n = dcmecef2ned(lat, lon)` calculates the direction cosine matrix, `n`, for a given set of geodetic latitude and longitude, `lat`, `lon`. `lat` is an `m` array of geodetic latitudes. `lon` is an `m` array of longitudes. `n` returns a 3-by-3-by-`m` matrix containing `m` direction cosine matrices. `n` performs the coordinate transformation of a vector in Earth-centered Earth-fixed (ECEF) axes into a vector in north-east-down (NED) axes. Geodetic latitudes and longitudes are input in degrees.

**Examples** Determine the direction cosine matrix from geodetic latitude and longitude:

```
lat = 45;  
lon = -122;  
dcm = dcmecef2ned(lat, lon)
```

```
dcm =  
  
    0.3747    0.5997    0.7071  
    0.8480   -0.5299         0  
    0.3747    0.5997   -0.7071
```

Determine the direction cosine matrix from multiple geodetic latitudes and longitudes:

```
lat = [45 37.5];  
lon = [-122 -85];  
dcm = dcmecef2ned(lat, lon)
```

```
dcm(:, :, 1) =  
  
    0.3747    0.5997    0.7071  
    0.8480   -0.5299         0  
    0.3747    0.5997   -0.7071
```

```
dcm(:, :, 2) =  
    -0.0531    0.6064    0.7934  
    0.9962    0.0872         0  
   -0.0691    0.7903   -0.6088
```

**See Also** [angle2dcm](#), [dcm2angle](#), [dcm2latlon](#)

# decyear

---

**Purpose** Calculate decimal year

**Syntax**

```
dy = decyear(v)
dy = decyear(s,f)
dy = decyear(y,mo,d)
dy = decyear([y,mo,d])
dy = decyear(y,mo,d,h,mi,s)
dy = decyear([y,mo,d,h,mi,s])
```

**Description** `dy = decyear(v)` converts one or more date vectors, `v`, into decimal year, `dy`. Input `v` can be an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. `decyear` returns a column vector of `m` decimal years.

A date vector contains six elements, specifying year, month, day, hour, minute, and second. A partial date vector has three elements, specifying year, month, and day. Each element of `v` must be a positive double-precision number.

`dy = decyear(s,f)` converts one or more date strings, `s`, to decimal year, `dy`, using format string `f`. `s` can be a character array where each row corresponds to one date string, or a one-dimensional cell array of strings. `decyear` returns a column vector of `m` decimal years, where `m` is the number of strings in `s`.

All of the date strings in `s` must have the same format `f`, which must be composed of date format symbols listed in the `datestr` function reference page. Formats containing the letter `Q` are not accepted by `decyear`.

Certain formats may not contain enough information to compute a date number. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. Date strings with two-character years are interpreted to be within the 100 years centered around the current year.

`dy = decyear(y,mo,d)` and `dy = decyear([y,mo,d])` return the decimal year for corresponding elements of the `y,mo,d` (year,month,day)

arrays. `y`, `mo`, and `d` must be arrays of the same size (or any of them can be a scalar).

`dy = decyear(y,mo,d,h,mi,s)` and `dy = decyear([y,mo,d,h,mi,s])` return the decimal year for corresponding elements of the `y,mo,d,h,mi,s` (year,month,day,hour,minute,second) arrays. The six arguments must be arrays of the same size (or any of them can be a scalar).

## Examples

Calculate decimal year for May 24, 2005:

```
dy = decyear('24-May-2005','dd-mmm-yyyy')
```

```
dy =
```

```
2.0054e+003
```

Calculate decimal year for December 19, 2006:

```
dy = decyear(2006,12,19)
```

```
dy =
```

```
2.0070e+003
```

Calculate decimal year for October 10, 2004, at 12:21:00 p.m.:

```
dy = decyear(2004,10,10,12,21,0)
```

```
dy =
```

```
2.0048e+003
```

## Assumptions and Limitations

The calculation of decimal year does not take into account leap seconds.

## See Also

`juliandate`, `leapyear`, `mjuliandate`

# delete (Aero.FlightGearAnimation)

---

**Purpose** 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 animation object contains.

**Examples** Delete the FlightGear animation object, h.

```
h=Aero.FlightGearAnimation;  
h.delete;
```

**See Also** initialize



# delete (Aero.VirtualRealityAnimation)

---

**Purpose** Destroy virtual reality animation object

**Syntax** delete(h)  
h.delete

**Description** delete(h) and h.delete destroy the virtual reality animation object h. This function also destroys the temporary file, if it exists, cleans up the vrfigure object, the animation object timer, and closes the vrworld object.

**Examples** Delete the virtual reality animation object, h.

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

**See Also** initialize

# dpressure

---

**Purpose** Compute dynamic pressure using velocity and density

**Syntax** `q = dpressure(v, r)`

**Description** `q = dpressure(v, r)` computes  $m$  dynamic pressures,  $q$ , from an  $m$ -by-3 array of velocities,  $v$ , and an array of  $m$  densities,  $r$ .  $v$  and  $r$  must have the same length units.

**Examples** Determine dynamic pressure for velocity in feet per second and density in slugs per feet cubed:

```
q = dpressure([84.3905 33.7562 10.1269], 0.0024)
```

```
q =  
10.0365
```

Determine dynamic pressure for velocity in meters per second and density in kilograms per meters cubed:

```
q = dpressure([25.7222 10.2889 3.0867], [1.225 0.3639])
```

```
q =  
475.9252  
141.3789
```

Determine dynamic pressure for velocity in meters per second and density in kilograms per meters cubed:

```
q = dpressure([50 20 6; 5 0.5 2], [1.225 0.3639])
```

```
q =
```

1.0e+003 \*

1.7983

0.0053

**See Also**

airspeed, machnumber

# ecef2lla

---

**Purpose** Convert Earth-centered Earth-fixed (ECEF) coordinates to geodetic coordinates

**Syntax**

```
lla = ecef2lla(p)
lla = ecef2lla(p, model)
lla = ecef2lla(p, f, Re)
```

**Description** `lla = ecef2lla(p)` converts the  $m$ -by-3 array of ECEF coordinates,  $p$ , to an  $m$ -by-3 array of geodetic coordinates (latitude, longitude and altitude),  $lla$ .  $lla$  is in [degrees degrees meters].  $p$  is in meters. The default ellipsoid planet is WGS84.

`lla = ecef2lla(p, model)` is an alternate method for converting the coordinates for a specific ellipsoid planet. Currently only 'WGS84' is supported for `model`.

`lla = ecef2lla(p, f, Re)` is another alternate method for converting the coordinates for a custom ellipsoid planet defined by flattening,  $f$ , and the equatorial radius,  $Re$ , in meters.

## Examples

Determine latitude, longitude, and altitude at a coordinate:

```
lla = ecef2lla([4510731 4510731 0])
```

```
lla =
```

```
0 45.0000 999.9564
```

Determine latitude, longitude, and altitude at multiple coordinates, specifying WGS84 ellipsoid model:

```
lla = ecef2lla([4510731 4510731 0; 0 4507609 4498719], 'WGS84')
```

```
lla =
```

```
0 45.0000 999.9564
```

```
45.1358 90.0000 999.8659
```

Determine latitude, longitude, and altitude at multiple coordinates, specifying custom ellipsoid model:

```
f = 1/196.877360;  
Re = 3397000;  
lla = ecef2lla([4510731 4510731 0; 0 4507609 4498719], f, Re)
```

```
lla =
```

```
1.0e+006 *
```

```
0 0.0000 2.9821  
0.0000 0.0001 2.9801
```

## See Also

geoc2geod, geod2geoc, lla2ecef

# **fanimation (Aero.FlightGearAnimation)**

---

**Purpose** Construct FlightGear animation object

**Syntax** `h = fanimation`  
`h = Aero.FlightGearAnimation`

**Description** `h = fanimation` 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 = fanimation
```

**See Also** `Aero.FlightGearAnimation`

**Purpose**

Return start and stop times of time series data

**Syntax**

```
[tstart,tstop] = findstartstoptimes(h,tsdata)
[tstart,stop] = h.findstartstoptimes(tsdata)
```

**Description**

[tstart,tstop] = findstartstoptimes(h,tsdata) and [tstart,stop] = h.findstartstoptimes(tsdata) return the start and stop times of time series data *tsdata* for the animation body object *h*.

**Examples**

Find the start and stop times of the time series data, *tsdata*.

```
b=Aero.Body;
b.load('pa24-250_orange.ac','Ac3d');
tsdata = [ ...
    0, 1,1,1, 0,0,0; ...
    10 2,2,2, 1,1,1; ];
b.TimeSeriesSource = tsdata;
[tstart,tstop] = findstartstoptimes(b,tsdata);
```

**See Also**

load

# findstartstoptimes (Aero.Node)

---

**Purpose** Return start and stop times for time series data

**Syntax** `[tstart,tstop] = findstartstoptimes(h,tsdata)`  
`[tstart,stop] = h.findstartstoptimes(tsdata)`

**Description** `[tstart,tstop] = findstartstoptimes(h,tsdata)` and `[tstart,stop] = h.findstartstoptimes(tsdata)` return the start and stop times of time series data `tsdata` for the virtual reality animation object `h`.

**Examples** Find the start and stop times of the time series data, `takeoffData`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot, '/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir, 'asttkoff.wrl'], 'f');
h.VRWorldFilename = [tempdir, 'asttkoff.wrl'];
h.initialize();
load takeoffData;
h.Nodes{7}.TimeseriesSource = takeoffData;
h.Nodes{7}.TimeseriesSourceType = 'StructureWith Time';
[tstart,stop]=h.Nodes{7}.findstartstoptimes;
```



# generatePatches (Aero.Body)

---

**Purpose** Generate patches for body with loaded face, vertex, and color data

**Syntax** generatePatches(h, ax)  
h.generatePatches(ax)

**Description** generatePatches(h, ax) and h.generatePatches(ax) generate patches for the animation body object h using the loaded face, vertex, and color data in ax.

**Examples** Generate patches for b using the axes, ax.

```
b=Aero.Body;  
b.load('pa24-250_orange.ac','Ac3d');  
f = figure;  
ax = axes;  
b.generatePatches(ax);
```

**See Also** load

# GenerateRunScript (Aero.FlightGearAnimation)

---

**Purpose** 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 directory. The default value is 'D:\Applications\FlightGear'.
GeometryModelName	Specify the name of the folder containing the desired model geometry in the <i>FlightGear\data\Aircraft</i> directory. 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'.

# GenerateRunScript (Aero.FlightGearAnimation)

---

AirportId	Specify the airport ID. The list of supported airports is available in the FlightGear interface, under <b>Location</b> . 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.

## Examples

Create a run script, `runfg.bat`, to start FlightGear flight simulator using the default object settings:

```
h = fanimation
GenerateRunScript(h)
```

Create a run script, `myscript.bat`, to start FlightGear flight simulator using the default object settings:

```
h = fanimation
h.OutputFileName = 'myscript.bat'
GenerateRunScript(h)
```

## See Also

`initialize`, `play`, `update`

# geoc2geod

---

**Purpose** Convert geocentric latitude to geodetic latitude

**Syntax**  
`gd = geoc2geod(gc, r)`  
`gd = geoc2geod(gc, r, model)`  
`gd = geoc2geod(gc, r, f, Re)`

**Description** `gd = geoc2geod(gc, r)` converts an array of  $m$  geocentric latitudes, `gc`, and an array of radii from the center of the planet, `r`, into an array of  $m$  geodetic latitudes, `gd`. Both `gc` and `gd` are in degrees. `r` is in meters.

`gd = geoc2geod(gc, r, model)` is an alternate method for converting from geocentric to geodetic latitude for a specific ellipsoid planet. Currently only 'WGS84' is supported for `model`.

`gd = geoc2geod(gc, r, f, Re)` is another alternate method for converting from geocentric to geodetic latitude for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters.

Geometric relationships are used to calculate the geodetic latitude in this noniterative method.

**Examples** Determine geodetic latitude given a geocentric latitude and radius:

```
gd = geoc2geod(45, 6379136)
```

```
gd =
```

```
45.1921
```

Determine geodetic latitude at multiple geocentric latitudes, given a radius and specifying WGS84 ellipsoid model:

```
gd = geoc2geod([0 45 90], 6379136, 'WGS84')
```

```
gd =
```

```
0 45.1921 90.0000
```

Determine geodetic latitude at multiple geocentric latitudes, given a radius and specifying custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
gd = geoc2geod([0 45 90], 6379136, f, Re)
```

```
gd =
```

```
0    45.1550    90.0000
```

## Assumptions and Limitations

This implementation generates a geodetic latitude that lies between  $\pm 90$  degrees.

## References

- Jackson, E.B., *Manual for a Workstation-based Generic Flight Simulation Program (LaRCsim) Version 1.4*, NASA TM 110164, April, 1995
- Hedgley, D. R., Jr., *An Exact Transformation from Geocentric to Geodetic Coordinates for Nonzero Altitudes*, NASA TR R-458, March, 1976
- Clynch, J. R., *Radius of the Earth — Radii Used in Geodesy*, Naval Postgraduate School, 2002, <http://www.oc.nps.navy.mil/oc2902w/geodesy/radiigeo.pdf>
- Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, John Wiley & Sons, New York, NY, 1992
- Edwards, C. H., and D. E. Penny, *Calculus and Analytical Geometry*, 2nd Edition, Prentice-Hall, Englewood Cliffs, NJ, 1986

## See Also

geod2geoc, ecef211a, 11a2ecef

# geocradius

---

**Purpose** Estimate radius of ellipsoid planet at geocentric latitude

**Syntax**

```
r = geocradius(lambda)
r = geocradius(lambda, model)
r = geocradius(lambda, f, Re)
```

**Description** `r = geocradius(lambda)` estimates the radius, `r`, of an ellipsoid planet at a particular geocentric latitude, `lambda`. `lambda` is in degrees. `r` is in meters. The default ellipsoid planet is WGS84.

`r = geocradius(lambda, model)` is an alternate method for estimating the radius for a specific ellipsoid planet. Currently only 'WGS84' is supported for `model`.

`r = geocradius(lambda, f, Re)` is another alternate method for estimating the radius for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters.

**Examples** Determine radius at 45 degrees latitude:

```
r = geocradius(45)
```

```
r =
```

```
6.3674e+006
```

Determine radius at multiple latitudes:

```
r = geocradius([0 45 90])
```

```
r =
```

```
1.0e+006 *
```

```
6.3781    6.3674    6.3568
```

Determine radius at multiple latitudes, specifying WGS84 ellipsoid model:

```
r = geocradius([0 45 90], 'WGS84')
```

```
r =
```

```
1.0e+006 *
```

```
6.3781    6.3674    6.3568
```

Determine radius at multiple latitudes, specifying custom ellipsoid model:

```
f = 1/196.877360;
```

```
Re = 3397000;
```

```
r = geocradius([0 45 90], f, Re)
```

```
r =
```

```
1.0e+006 *
```

```
3.3970    3.3883    3.3797
```

## References

Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, John Wiley & Sons, New York, NY, 1992

Zipfel, P. H., and D. E. Penny, *Modeling and Simulation of Aerospace Vehicle Dynamics*, AIAA Education Series, Reston, VA, 2000

## See Also

geoc2geod, geod2geoc

# geod2geoc

---

**Purpose** Convert geodetic latitude to geocentric latitude

**Syntax**  
`gc = geod2geoc(gd, h)`  
`gc = geod2geoc(gd, h, model)`  
`gc = geod2geoc(gd, h, f, Re)`

**Description** `gc = geod2geoc(gd, h)` converts an array of  $m$  geodetic latitudes, `gd`, and an array of mean sea level altitudes, `h`, into an array of  $m$  geocentric latitudes, `gc`. Both `gc` and `gd` are in degrees. `h` is in meters.

`gc = geod2geoc(gd, h, model)` is an alternate method for converting from geodetic to geocentric latitude for a specific ellipsoid planet. Currently only 'WGS84' is supported for `model`.

`gc = geod2geoc(gd, h, f, Re)` is another alternate method for converting from geodetic to geocentric latitude for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters.

**Examples** Determine geocentric latitude given a geodetic latitude and altitude:

```
gc = geod2geoc(45, 1000)
```

```
gc =  
44.8076
```

Determine geocentric latitude at multiple geodetic latitudes and altitudes, specifying WGS84 ellipsoid model:

```
gc = geod2geoc([0 45 90], [1000 0 2000], 'WGS84')
```

```
gc =  
0  
44.8076  
90.0000
```



Determine geocentric latitude at multiple geodetic latitudes, given an altitude and specifying custom ellipsoid model:

```
f = 1/196.877360;  
Re = 3397000;  
gc = geod2geoc([0 45 90], 2000, f, Re)
```

```
gc =  
  
      0  
    44.7084  
    90.0000
```

### **Assumptions and Limitations**

This implementation generates a geocentric latitude that lies between  $\pm 90$  degrees.

### **References**

Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, John Wiley & Sons, New York, NY, 1992

### **See Also**

geoc2geod, ecef2lla, lla2ecef

# geoidegm96

---

**Purpose** Calculate geoid height as determined from EGM96 Geopotential Model

**Syntax**  
N = geoidegm96(lat, long)  
N = geoidegm96(lat, long, action)

**Description** N = geoidegm96(lat, long) calculates the geoid height as determined from the EGM96 Geopotential Model. It calculates geoid heights to 0.01 meters. This function interpolates geoid heights from a 15-minute grid of point values in the tide-free system, using the EGM96 Geopotential Model to the degree and order 360. The geoid undulations are relative to the WGS84 ellipsoid.

N = geoidegm96(lat, long, action) calculates the geoid height as determined from the EGM96 Geopotential Model. This function performs action if latitude or longitude are out of range.

Inputs required by geoidegm96:

lat	An array of <i>m</i> geocentric latitudes, in degrees, where north latitude is positive and south latitude is negative. lat must be of type single or double. If lat is not within the range -90 to 90, inclusive, this function wraps the value to be within the range.
long	An array of <i>m</i> geocentric longitudes, in degrees, where east longitude is positive and west longitude is negative. long must be of type single or double. If long is not within the range 0 to 360 inclusive, this function wraps the value to be within the range.
action	A string to determine action for out-of-range input. Specify if out-of-range input invokes a

'Warning', 'Error', or no action ('None'). The default is 'Warning'.

## Examples

Calculate the geoid height at 42.4 degrees N latitude and 71.0 degrees E longitude.

```
N = geoidgm96( 42.4, 71.0)
```

Calculate the geoid height at two different locations, with out-of-range actions generating warnings.

```
N = geoidgm96( [39.3,33.4], [-77.2, 36.5])
```

Calculate the geoid height with latitude wrapping, with out-of-range actions displaying no warnings.

```
N = geoidgm96(100,150,'None')
```

## Limitations

This function has the limitations of the 1996 Earth Geopotential Model. For more information, see <http://www.ngdc.noaa.gov/seg/gravity/document/html/egm96.shtml>.

The WGS84 EGM96 geoid undulations have an error range of +/-0.5 to +/-1.0 meters worldwide.

## References

NIMA TR8350.2: "Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems."

NASA/TP-1998-206861: "The Development of the Joint NASA GSFC and NIMA Geopotential Model EGM96"

National Geospatial-Intelligence Agency Website:  
<http://earth-info.nga.mil/GandG/wgs84/gravitymod/egm96/egm96.html>

## See Also

gravitywgs84

# Geometry (Aero.Geometry)

---

**Purpose** Construct 3-D geometry for use with animation object

**Syntax** `h = Aero.Geometry`

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

See `Aero.Geometry` for further details.

**See Also** `Aero.Geometry`

## Purpose

Implement 1984 World Geodetic System (WGS84) representation of Earth's gravity

## Syntax

```
g = gravitywgs84(h, lat)
g = gravitywgs84(h, lat, lon, method, [noatm, nocent, prec,
    jd], action)
gt = gravitywgs84(h, lat, lon, 'Exact', [noatm, nocent, prec,
    jd], action)
[g gn] = gravitywgs84(h, lat, lon, 'Exact', [noatm, nocent,
    prec, jd], action)
```

## Description

`g = gravitywgs84(h, lat)` implements the mathematical representation of the geocentric equipotential ellipsoid of WGS84. Using `h`, an array of `m` altitudes in meters, and `lat`, an array of `m` geodetic latitudes in degrees, calculates `g`, an array of `m` gravity values in the direction normal to the Earth's surface at a specific location. The default calculation method is Taylor Series. Gravity precision is controlled via the `method` parameter.

`g = gravitywgs84(h, lat, lon, method, [noatm, nocent, prec, jd], action)` lets you specify both latitude and longitude, as well as other optional inputs, when calculating gravity values in the direction normal to the Earth's surface. In this format, `method` can be either 'CloseApprox' or 'Exact'.

`gt = gravitywgs84(h, lat, lon, 'Exact', [noatm, nocent, prec, jd], action)` calculates an array of total gravity values in the direction normal to the Earth's surface.

`[g gn] = gravitywgs84(h, lat, lon, 'Exact', [noatm, nocent, prec, jd], action)` calculates gravity values in the direction both normal and tangential to the Earth's surface.

Inputs for `gravitywgs84` are:

<code>h</code>	An array of <code>m</code> altitudes, in meters
<code>lat</code>	An array of <code>m</code> geodetic latitudes, in degrees, where north latitude is positive, and south latitude is negative
<code>lon</code>	An array of <code>m</code> geodetic longitudes, in degrees, where east longitude is positive, and west longitude is negative. This input is available only with <code>method</code> specified as <code>'CloseApprox'</code> or <code>'Exact'</code> .
<code>method</code>	A string specifying the method to calculate gravity: <code>'TaylorSeries'</code> , <code>'CloseApprox'</code> , or <code>'Exact'</code> . The default is <code>'TaylorSeries'</code> .
<code>noatm</code>	A logical value specifying the exclusion of Earth's atmosphere. Set to <code>true</code> for the Earth's gravitational field to exclude the mass of the atmosphere. Set to <code>false</code> for the value for the Earth's gravitational field to include the mass of the atmosphere. This option is available only with <code>method</code> specified as <code>'CloseApprox'</code> or <code>'Exact'</code> . The default is <code>false</code> .
<code>nocent</code>	A logical value specifying the removal of centrifugal effects. Set to <code>true</code> to calculate gravity based on pure attraction resulting from the normal gravitational potential. Set to <code>false</code> to calculate gravity including the centrifugal force resulting from the Earth's angular velocity. This option is available only with <code>method</code> specified as <code>'CloseApprox'</code> or <code>'Exact'</code> . The default is <code>false</code> .

<code>prec</code>	A logical value specifying the presence of a precessing reference frame. Set to <code>true</code> for the angular velocity of the Earth to be calculated using the International Astronomical Union (IAU) value of the Earth's angular velocity and the precession rate in right ascension. To obtain the precession rate in right ascension, Julian Centuries from Epoch J2000.0 is calculated using the Julian date, <code>jd</code> . If set to <code>false</code> , the angular velocity of the Earth used is the value of the standard Earth rotating at a constant angular velocity. This option is available only with method specified as <code>'CloseApprox'</code> or <code>'Exact'</code> . The default is <code>false</code> .
<code>jd</code>	A scalar value specifying Julian date used to calculate Julian Centuries from Epoch J2000.0. This input is available only with method specified as <code>'CloseApprox'</code> or <code>'Exact'</code> .
<code>action</code>	A string to determine action for out-of-range input. Specify if out-of-range input invokes a <code>'Warning'</code> , <code>'Error'</code> , or no action ( <code>'None'</code> ). The default is <code>'Warning'</code> .

Outputs calculated for the Earth's gravity include:

- g** An array of  $m$  gravity values in the direction normal to the Earth's surface at a specific `lat lon` location. A positive value indicates a downward direction.
- gt** An array of  $m$  total gravity values in the direction normal to the Earth's surface at a specific `lat lon` location. A positive value indicates a downward direction. This option is available only with `method` specified as 'Exact'.
- gn** An array of  $m$  gravity values in the direction tangential to the Earth's surface at a specific `lat lon` location. A positive value indicates a northward direction. This option is available only with `method` specified as 'Exact'.

## Examples

Calculate the normal gravity at 5000 meters and 55 degrees latitude using the Taylor Series approximation method with errors for out-of-range inputs:

```
g = gravitywgs84( 5000, 55, 'TaylorSeries', 'Error' )
```

```
g =
```

```
9.7997
```

Calculate the normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude using the Close Approximation method with atmosphere, centrifugal effects, and no precessing, with warnings for out-of-range inputs:

```
g = gravitywgs84( 15000, 45, 120, 'CloseApprox' )
```

```
g =
```



9.7601

Calculate the normal and tangential gravity at 1000 meters, 0 degrees latitude, and 20 degrees longitude using the Exact method with atmosphere, centrifugal effects, and no precessing, with warnings for out-of-range inputs:

```
[g, gt] = gravitywgs84( 1000, 0, 20, 'Exact' )
```

g =

9.7772

gt =

0

Calculate the normal and tangential gravity at 1000 meters, 0 degrees latitude, and 20 degrees longitude and 11,000 meters, 30 degrees latitude, and 50 degrees longitude using the Exact method with atmosphere, centrifugal effects, and no precessing, with no actions for out-of-range inputs:

```
h = [1000; 11000];
```

```
lat = [0; 30];
```

```
lon = [20; 50];
```

```
[g, gt] = gravitywgs84( h, lat, lon, 'Exact', 'None' )
```

g =

9.7772

9.7594

```
gt =  
  
1.0e-004 *  
  
0  
-0.7751
```

Calculate the normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude and 5000 meters, 55 degrees latitude, and 100 degrees longitude using the Close Approximation method with atmosphere, no centrifugal effects, and no precessing, with warnings for out-of-range inputs:

```
h = [15000 5000];  
lat = [45 55];  
lon = [120 100];  
g = gravitywgs84( h, lat, lon, 'CloseApprox', [false true false 0] )
```

```
g =  
  
9.7771    9.8109
```

Calculate the normal and tangential gravity at 1000 meters, 0 degrees latitude, and 20 degrees longitude using the Exact method with atmosphere, centrifugal effects, and precessing at Julian date 2451545, with warnings for out-of-range inputs:

```
[g, gt] = gravitywgs84( 1000, 0, 20, 'Exact', ...  
                        [ false false true 2451545 ], 'Warning' )
```

```
g =  
  
9.7772
```

```
gt =
```

```
0
```

Calculate the normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude using the Close Approximation method with no atmosphere, with centrifugal effects, and with precessing at Julian date 2451545, with errors for out-of-range inputs:

```
g = gravitywgs84( 15000, 45, 120, 'CloseApprox', ...  
    [ true false true 2451545 ], 'Error' )
```

```
g =
```

```
9.7601
```

Calculate the total normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude using the Exact method with no atmosphere, with centrifugal effects, and with precessing at Julian date 2451545, with errors for out-of-range inputs:

```
g = gravitywgs84( 15000, 45, 120, 'Exact', ...  
    [ true false true 2451545 ], 'Error' )
```

```
g =
```

```
9.7601
```

## Assumptions and Limitations

The WGS84 gravity calculations are based on the assumption of a geocentric equipotential ellipsoid of revolution. Since the gravity potential is assumed to be the same everywhere on the ellipsoid, there must be a specific theoretical gravity potential that can be uniquely determined from the four independent constants defining the ellipsoid.

Use of the WGS84 Taylor Series model should be limited to low geodetic heights. It is sufficient near the surface when submicrogal precision is not necessary. At medium and high geodetic heights, it is less accurate.

Use of the WGS84 Close Approximation model should be limited to a geodetic height of 20,000.0 meters (approximately 65,620.0 feet). Below this height, it gives results with submicrogal precision.

To predict and determine a satellite orbit with high accuracy, use the EGM96 through degree and order 70.

## References

NIMA TR8350.2: "Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems."

# initialize (Aero.FlightGearAnimation)

---

**Purpose** Set up FlightGear animation object

**Syntax** `initialize(h)`  
`h.initialize`

**Description** `initialize(h)` and `h.initialize` set up the FlightGear version, IP address, and socket for the FlightGear animation object `h`.

**Examples** Initialize the animation object, `h`.

```
h = Aero.FlightGearAnimation;  
h.initialize();
```

**See Also** `delete`, `play`, `GenerateRunScript`, `update`

# initialize (Aero.VirtualRealityAnimation)

---

**Purpose** Create and populate virtual reality animation object

**Syntax** initialize(h)  
h.initialize

**Description** initialize(h) and h.initialize create a virtual reality animation world and populate the virtual reality animation object h. If a previously initialized virtual reality animation object exists, and that object has user-specified data, this function saves the previous object to be reset after the initialization.

**Examples** Initialize the virtual reality animation object, h.

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

**See Also** delete, play

**Purpose**

Calculate Julian date

**Syntax**

```
jd = juliandate(v)
jd = juliandate(s,f)
jd = juliandate(y,mo,d)
jd = juliandate([y,mo,d])
jd = juliandate(y,mo,d,h,mi,s)
jd = juliandate([y,mo,d,h,mi,s])
```

**Description**

`jd = juliandate(v)` converts one or more date vectors, `v`, into Julian date, `jd`. Input `v` can be an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. `juliandate` returns a column vector of `m` Julian dates, which are the number of days and fractions since noon Universal Time on January 1, 4713 BCE.

A date vector contains six elements, specifying year, month, day, hour, minute, and second. A partial date vector has three elements, specifying year, month, and day. Each element of `v` must be a positive double-precision number.

`jd = juliandate(s,f)` converts one or more date strings, `s`, into Julian date, `jd`, using format string `f`. `s` can be a character array where each row corresponds to one date string, or a one-dimensional cell array of strings. `juliandate` returns a column vector of `m` Julian dates, where `m` is the number of strings in `s`.

All of the date strings in `s` must have the same format `f`, which must be composed of date format symbols listed in the `datestr` function reference page. Formats containing the letter `Q` are not accepted by `juliandate`.

Certain formats may not contain enough information to compute a date number. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. Date strings with two-character years are interpreted to be within the 100 years centered around the current year.

`jd = juliandate(y,mo,d)` and `jd = juliandate([y,mo,d])` return the decimal year for corresponding elements of the `y,mo,d`

# juliandate

---

(year,month,day) arrays. `y`, `mo`, and `d` must be arrays of the same size (or any of them can be a scalar).

`jd = juliandate(y,mo,d,h,mi,s)` and `jd = juliandate([y,mo,d,h,mi,s])` return the Julian dates for corresponding elements of the `y,mo,d,h,mi,s` (year,month,day,hour,minute,second) arrays. The six arguments must be arrays of the same size (or any of them can be a scalar).

## Examples

Calculate Julian date for May 24, 2005:

```
jd = juliandate('24-May-2005','dd-mmm-yyyy')  
  
jd =  
  
2.4535e+006
```

Calculate Julian date for December 19, 2006:

```
jd = juliandate(2006,12,19)  
  
jd =  
  
2.4541e+006
```

Calculate Julian date for October 10, 2004, at 12:21:00 p.m.:

```
jd = juliandate(2004,10,10,12,21,0)  
  
jd =  
  
2.4533e+006
```

## Assumptions and Limitations

This function is valid for all common era (CE) dates in the Gregorian calendar.

The calculation of Julian date does not take into account leap seconds.

## See Also

`decyear`, `leapyear`, `mjuliandate`



**Purpose** Determine leap year

**Syntax** `ly = leapyear(year)`

**Description** `ly = leapyear(year)` determines whether one or more years are leap years or not. The output, `ly`, is a logical array. `year` should be numeric.

**Examples** Determine whether 2005 is a leap year:

```
ly = leapyear(2005)
```

```
ly =
```

```
0
```

Determine whether 2000, 2005, and 2020 are leap years:

```
ly = leapyear([2000 2005 2020])
```

```
ly =
```

```
1    0    1
```

**Assumptions and Limitations** The determination of leap years is done by Gregorian calendar rules.

**See Also** `decyear`, `juliandate`, `mjuliandate`

# lla2ecef

---

**Purpose** Convert geodetic coordinates to Earth-centered Earth-fixed (ECEF) coordinates

**Syntax**

```
p = lla2ecef(lla)
p = lla2ecef(lla, model)
p = lla2ecef(lla, f, Re)
```

**Description** `p = lla2ecef(lla)` converts an  $m$ -by-3 array of geodetic coordinates (latitude, longitude and altitude), `lla`, to an  $m$ -by-3 array of ECEF coordinates, `p`. `lla` is in [degrees degrees meters]. `p` is in meters. The default ellipsoid planet is WGS84.

`p = lla2ecef(lla, model)` is an alternate method for converting the coordinates for a specific ellipsoid planet. Currently only 'WGS84' is supported for `model`.

`p = lla2ecef(lla, f, Re)` is another alternate method for converting the coordinates for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters.

**Examples** Determine ECEF coordinates at a latitude, longitude, and altitude:

```
p = lla2ecef([0 45 1000])
```

```
p =
```

```
1.0e+006 *
```

```
4.5107    4.5107    0
```

Determine ECEF coordinates at multiple latitudes, longitudes, and altitudes, specifying WGS84 ellipsoid model:

```
p = lla2ecef([0 45 1000; 45 90 2000], 'WGS84')
```

```
p =
```

```

1.0e+006 *
    4.5107    4.5107    0
    0.0000    4.5190    4.4888

```

Determine ECEF coordinates at multiple latitudes, longitudes, and altitudes, specifying custom ellipsoid model:

```

f = 1/196.877360;
Re = 3397000;
p = lla2ecef([0 45 1000; 45 90 2000], f, Re)

```

p =

```

1.0e+006 *
    2.4027    2.4027    0
    0.0000    2.4096    2.3852

```

## See Also

ecef2lla, geoc2geod, geod2geoc

# load (Aero.Body)

---

**Purpose** Get geometry data from source

**Syntax**

```
load(h, bodyDataSrc)
h.load(bodyDataSrc)
load(h, bodyDataSrc, geometrystate)
h.load(bodyDataSrc, geometrystate)
```

**Description** `load(h, bodyDataSrc)` and `h.load(bodyDataSrc)` load the graphics data from the body graphics file. This command assumes a default geometry source type set to `Auto`.

`load(h, bodyDataSrc, geometrystate)` and `h.load(bodyDataSrc, geometrystate)` load the graphics data from the body graphics file, `bodyDataSrc`, into the face, vertex, and color data of the animation body object `h`. Then, when axes `ax` is available, you can use this data to generate patches with `generatePatches`. `geometrystate` is the geometry source type for the body.

By default `geometrystate` is set to `Auto`, which recognizes `.mat` extensions as MAT-files, `.ac` extensions as Ac3d files, and structures containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables. If you want to use alternate file extensions or file types, enter one of the following:

- `Auto`
- `Variable`
- `MatFile`
- `Ac3d`
- `Custom`

**Examples** Load the graphic data from the graphic data file, `pa24-250_orange.ac`, into `b`.

```
b=Aero.Body;
b.load('pa24-250_orange.ac', 'Ac3d');
```

**See Also**      generatePatches, move, update

# machnumber

---

**Purpose** Compute Mach number using velocity and speed of sound

**Syntax** `mach = machnumber(v, a)`

**Description** `mach = machnumber(v, a)` computes `m` Mach numbers, `mach`, from an `m`-by-3 array of velocities, `v`, and an array of `m` speeds of sound, `a`. `v` and `a` must have the same length units.

**Examples** Determine the Mach number for velocity and speed of sound in feet per second:

```
mach = machnumber([84.3905 33.7562 10.1269], 1116.4505)
```

```
mach =
```

```
0.0819
```

Determine the Mach number for velocity and speed of sound in meters per second:

```
mach = machnumber([25.7222 10.2889 3.0867], [340.2941 295.0696])
```

```
mach =
```

```
0.0819 0.0945
```

Determine the Mach number for velocity and speed of sound in knots:

```
mach = machnumber([50 20 6; 5 0.5 2], [661.4789 573.5694])
```

```
mach =
```

```
0.0819
```

```
0.0094
```

**See Also**      airspeed, alphabeta, dpressure

# mjuliandate

---

**Purpose** Calculate modified Julian date

**Syntax**

```
mjd = mjuliandate(v)
mjd = mjuliandate(s,f)
mjd = mjuliandate(y,mo,d)
mjd = mjuliandate([y,mo,d])
mjd = mjuliandate(y,mo,d,h,mi,s)
mjd = mjuliandate([y,mo,d,h,mi,s])
```

**Description** `mjd = mjuliandate(v)` converts one or more date vectors, `v`, into modified Julian date, `mjd`. Input `v` can be an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. `mjuliandate` returns a column vector of `m` modified Julian dates. Modified Julian dates begin at midnight rather than noon and have the first two digits of the corresponding Julian date removed.

A date vector contains six elements, specifying year, month, day, hour, minute, and second. A partial date vector has three elements, specifying year, month, and day. Each element of `v` must be a positive double-precision number.

`mjd = mjuliandate(s,f)` converts one or more date strings, `s`, into modified Julian date, `mjd`, using format string `f`. `s` can be a character array where each row corresponds to one date string, or a one-dimensional cell array of strings. `mjuliandate` returns a column vector of `m` modified Julian dates, where `m` is the number of strings in `s`.

All of the date strings in `s` must have the same format `f`, which must be composed of date format symbols listed in the `datestr` function reference page. Formats containing the letter `Q` are not accepted by `mjuliandate`.

Certain formats may not contain enough information to compute a date number. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. Date strings with two-character years are interpreted to be within the 100 years centered around the current year.



`mjd = mjuliandate(y,mo,d)` and `mjd = mjuliandate([y,mo,d])` return the decimal year for corresponding elements of the `y,mo,d` (year,month,day) arrays. `y`, `mo`, and `d` must be arrays of the same size (or any of them can be a scalar).

`mjd = mjuliandate(y,mo,d,h,mi,s)` and `mjd = mjuliandate([y,mo,d,h,mi,s])` return the modified Julian dates for corresponding elements of the `y,mo,d,h,mi,s` (year,month,day,hour,minute,second) arrays. The six arguments must be arrays of the same size (or any of them can be a scalar).

## Examples

Calculate the modified Julian date for May 24, 2005:

```
mjd = mjuliandate('24-May-2005','dd-mmm-yyyy')  
  
mjd =  
  
53514
```

Calculate the modified Julian date for December 19, 2006:

```
mjd = mjuliandate(2006,12,19)  
  
mjd =  
  
54088
```

Calculate the modified Julian date for October 10, 2004, at 12:21:00 p.m.:

```
mjd = mjuliandate(2004,10,10,12,21,0)  
  
mjd =  
  
5.3289e+004
```

# mjuliandate

---

## **Assumptions and Limitations**

This function is valid for all common era (CE) dates in the Gregorian calendar.

The calculation of modified Julian date does not take into account leap seconds.

## **See Also**

decyear, juliandate, leapyear

**Purpose** Change animation body position and orientation

**Syntax** `move(h, translation, rotation)`  
`h.move(translation,rotation)`

**Description** `move(h, translation, rotation)` and `h.move(translation,rotation)` set a new position and orientation for the body object `h`. `translation` is a 1-by-3 matrix in the aerospace body  $x$ - $y$ - $z$  coordinate system. `rotation` is a 1-by-3 matrix, in radians, that specifies the rotations about the right-hand  $x$ - $y$ - $z$  sequence of coordinate axes. The order of application of the rotation is  $z$ - $y$ - $x$  ( $r$ - $q$ - $p$ ).

**Examples** Change animation body position to *newpos* and *newrot*.

```
h = Aero.Body;  
h.load('ac3d_xyzisrgb.ac','Ac3d');  
newpos = h.Position + 1.00;  
newrot = h.Rotation + 0.01;  
h.move(newpos,newrot);
```

**See Also** `load`

# move (Aero.Node)

---

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

**Examples** Move the Lynx body. This example uses the Simulink 3D Animation `vrnode/getfield` function to retrieve the translation and rotation. These coordinates are those used in the Simulink 3D Animation software.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot, '/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir, 'asttkoff.wrl'], 'f');
h.VRWorldFilename = [tempdir, 'asttkoff.wrl'];
h.initialize();
newtrans = getfield(h.Nodes{4}.VRNode, 'translation') + 1.0;
newrot = getfield(h.Nodes{4}.VRNode, 'rotation') + [.2 0.01 0.01 0.01];
h.Nodes{4}.move(newtrans, newrot);
```

**Limitations** This function cannot get the node position in aerospace body coordinates; it needs to use the `CoordTransformFcn` to do so.

This function cannot set a viewpoint position or orientation (see `addViewpoint`).

### See Also

`addNode`

# Node (Aero.Node)

---

**Purpose** Create node object for use with virtual reality animation

**Syntax** `h = Aero.Node`

**Description** `h = Aero.Node` creates a node object for use with virtual reality animation.

See `Aero.Node` for further details.

**See Also** `Aero.Node`

# nodeInfo (Aero.VirtualRealityAnimation)

---

**Purpose** Create list of nodes associated with virtual reality animation object

**Syntax**

```
nodeInfo(h)
h.nodeInfo
n = nodeInfo(h)
n = h.nodeInfo
```

**Description** `nodeInfo(h)` and `h.nodeInfo` create a list of nodes associated with the virtual reality animation object, `h`.

`n = nodeInfo(h)` and `n = h.nodeInfo` create a cell array (`n`) that contains the node information. The function stores the information in a cell array as follows:

```
N{1,n} = Node Index
N{2,n} = Node Name
N{3,n} = Node Type
```

where `n` is the number of nodes. You might want to use this function to find an existing node by name and then perform a certain action on it using the node index.

## Examples

Create list of nodes associated with virtual reality animation object, `h`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot, '/toolbox/aero/astdemos/asttkoff.wrl'];
h.initialize();
h.nodeInfo;
```

**See Also** `addNode`

# play (Aero.FlightGearAnimation)

---

**Purpose** 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.



# play (Aero.FlightGearAnimation)

---

'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.

# play (Aero.FlightGearAnimation)

---

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;...
```

# play (Aero.FlightGearAnimation)

---

```
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.SetTimer('MyFGTimer')
h.play('MyFGTimer')
```

## See Also

GenerateRunScript, initialize, update

# play (Aero.VirtualRealityAnimation)

---

**Purpose** 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

# play (Aero.VirtualRealityAnimation)

---

'StructureWithTime'	<p>Simulink struct with time (for example, Simulink root outport logging 'Structure with time'):</p> <ul style="list-style-type: none"><li>• <code>signals(1).values:</code> x y z</li><li>• <code>signals(2).values:</code> phi theta psi</li></ul> <p>Signals are linearly interpolated vs. time using <code>interp1</code>.</p>
'Array6DoF'	<p>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.</p>
'Array3DoF'	<p>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.</p>
'Custom'	<p>Position and angle data is retrieved from 'TimeseriesSource' by the currently registered 'TimeseriesReadFcn'.</p>

The time advancement algorithm used by play is based on animation frames counted by ticks:

```
ticks = ticks + 1;  
time = tstart + ticks*FramesPerSecond*TimeScaling;
```

where

# play (Aero.VirtualRealityAnimation)

---

TimeScaling	Specify the seconds of animation data per second of wall-clock time.
FramesPerSecond	Specify the number of frames per second used to animate the 'TimeseriesSource'.

For default 'TimeseriesReadFcn' methods, the last frame played is the last time value.

Time is in seconds, position values are in the same units as the geometry data loaded into the animation object, and all angles are in radians.

## Examples

Animate virtual reality world, `asttkoff`.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = [matlabroot, '/toolbox/aero/astdemos/asttkoff.wrl'];
h.initialize();
load takeoffData
h.Nodes{7}.TimeseriesSource = takeoffData;
h.Nodes{7}.TimeseriesSourceType = 'StructureWithTime';
h.Nodes{7}.CoordTransformFcn = @vranimCustomTransform;
h.play();
```

## See Also

`initialize`

**Purpose** Convert quaternion to rotation angles

**Syntax**  
`[r1 r2 r3] = quat2angle(q)`  
`[r1 r2 r3] = quat2angle(q, s)`

**Description** `[r1 r2 r3] = quat2angle(q)` calculates the set of rotation angles, `r1`, `r2`, `r3`, for a given quaternion, `q`. `q` is an `m`-by-4 matrix containing `m` quaternions. Each element of `q` must be a real number. `q` has its scalar number as the first column.

Rotation angles are output in radians.

`r1`  
 Returns an `m` array of first rotation angles.

`r2`  
 Returns an `m` array of second rotation angles.

`r3`  
 Returns an `m` array of third rotation angles.

`[r1 r2 r3] = quat2angle(q, s)` calculates the set of rotation angles, `r1`, `r2`, `r3`, for a given quaternion, `q`, and a specified rotation sequence, `s`.

The default rotation sequence is 'ZYX', where `r1` is z-axis rotation, `r2` is y-axis rotation, and `r3` is x-axis rotation.

Supported rotation sequence strings are 'ZYX', 'YZY', 'ZXY', 'XZX', 'YXZ', 'YXY', 'YZX', 'YZY', 'XYZ', 'XYX', 'XZY', and 'XZX'.

**Examples** Determine the rotation angles from `q = [1 0 1 0]`.

```
[yaw, pitch, roll] = quat2angle([1 0 1 0])
yaw =
    0
pitch =
    1.5708
roll =
    0
```

# quat2angle

---

Determine the rotation angles from multiple quaternions.

```
q = [1 0 1 0; 1 0.5 0.3 0.1];  
[pitch, roll, yaw] = quat2angle(q, 'YXZ')  
pitch =  
    1.5708  
    0.8073  
roll =  
     0  
    0.7702  
yaw =  
     0  
    0.5422
```

## Assumptions and Limitations

The limitations for the 'ZYX', 'ZXY', 'YXZ', 'YZX', 'XYZ', and 'XZY' implementations generate an r2 angle that lies between  $\pm 90$  degrees, and r1 and r3 angles that lie between  $\pm 180$  degrees.

The limitations for the 'ZYZ', 'ZXZ', 'YXY', 'YZY', 'YXY', and 'XZX' implementations generate an r2 angle that lies between 0 and 180 degrees, and r1 and r3 angles that lie between  $\pm 180$  degrees.

## See Also

angle2dcm, angle2quat, dcm2angle, dcm2quat, quat2dcm



**Purpose** Convert quaternion to direction cosine matrix

**Syntax** `n = quat2dcm(q)`

**Description** `n = quat2dcm(q)` calculates the direction cosine matrix, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns a 3-by-3-by-`m` matrix of direction cosine matrices. The direction cosine matrix performs the coordinate transformation of a vector in inertial axes to a vector in body axes. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

**Examples** Determine the direction cosine matrix from `q = [1 0 1 0]`:

```
dcm = quat2dcm([1 0 1 0])
```

```
dcm =
```

```

    0         0   -1.0000
    0    1.0000         0
  1.0000         0         0
```

Determine the direction cosine matrices from multiple quaternions:

```
q = [1 0 1 0; 1 0.5 0.3 0.1];
dcm = quat2dcm(q)
```

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

```

    0         0   -1.0000
    0    1.0000         0
  1.0000         0         0
```

```
dcm(:,:,2) =
```

# quat2dcm

---

0.8519	0.3704	-0.3704
0.0741	0.6148	0.7852
0.5185	-0.6963	0.4963

## See Also

[angle2dcm](#), [dcm2angle](#), [dcm2quat](#), [angle2quat](#), [quat2angle](#),  
[quatrotate](#)

**Purpose** Calculate conjugate of quaternion

**Syntax** `n = quatconj(q)`

**Description** `n = quatconj(q)` calculates the conjugate, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns an `m`-by-4 matrix of conjugates. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

**Examples** Determine the conjugate of `q = [1 0 1 0]`:

```
conj = quatconj([1 0 1 0])
```

```
conj =
```

```
1    0   -1    0
```

**See Also** `quatdivide`, `quatinv`, `quatmod`, `quatmultiply`, `quatnorm`, `quatnormalize`, `quatrotate`

# quatdivide

---

**Purpose** Divide quaternion by another quaternion

**Syntax** `n = quatdivide(q,r)`

**Description** `n = quatdivide(q,r)` calculates the result of quaternion division, `n`, for two given quaternions, `q` and `r`. Inputs `q` and `r` can each be either an `m`-by-4 matrix containing `m` quaternions, or a single 1-by-4 quaternion. `n` returns an `m`-by-4 matrix of quaternion quotients. Each element of `q` and `r` must be a real number. Additionally, `q` and `r` have their scalar number as the first column.

**Examples** Determine the division of two 1-by-4 quaternions:

```
q = [1 0 1 0];  
r = [1 0.5 0.5 0.75];  
d = quatdivide(q, r)
```

```
d =
```

```
0.7273    0.1212    0.2424   -0.6061
```

Determine the division of a 2-by-4 quaternion by a 1-by-4 quaternion:

```
q = [1 0 1 0; 2 1 0.1 0.1];  
r = [1 0.5 0.5 0.75];  
d = quatdivide(q, r)
```

```
d =
```

```
0.7273    0.1212    0.2424   -0.6061  
1.2727    0.0121   -0.7758   -0.4606
```

**See Also** `quatconj`, `quatinv`, `quatmod`, `quatmultiply`, `quatnorm`, `quatnormalize`, `quatrotate`

**Purpose** Calculate inverse of quaternion

**Syntax** `n = quatinv(q)`

**Description** `n = quatinv(q)` calculates the inverse, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns an `m`-by-4 matrix of inverses. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

**Examples** Determine the inverse of `q = [1 0 1 0]`:

```
qinv = quatinv([1 0 1 0])
```

```
qinv =
```

```
    0.5000         0   -0.5000         0
```

**See Also** `quatconj`, `quatdivide`, `quatmod`, `quatmultiply`, `quatnorm`, `quatnormalize`, `quatrotate`

# quatmod

---

**Purpose** Calculate modulus of quaternion

**Syntax** `n = quatmod(q)`

**Description** `n = quatmod(q)` calculates the modulus, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns a column vector of `m` moduli. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

**Examples** Determine the modulus of `q = [1 0 0 0]`:

```
mod = quatmod([1 0 0 0])
```

```
mod =
```

```
1
```

**See Also** `quatconj`, `quatdivide`, `quatinv`, `quatmultiply`, `quatnorm`, `quatnormalize`, `quatrotate`

**Purpose** Calculate product of two quaternions

**Syntax** `n = quatmultiply(q,r)`

**Description** `n = quatmultiply(q,r)` calculates the quaternion product, `n`, for two given quaternions, `q` and `r`. Inputs `q` and `r` can each be either an `m`-by-4 matrix containing `m` quaternions, or a single 1-by-4 quaternion. `n` returns an `m`-by-4 matrix of quaternion products. Each element of `q` and `r` must be a real number. Additionally, `q` and `r` have their scalar number as the first column.

---

**Note** Quaternion multiplication is not commutative.

---

**Examples** 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 =  
  
    0.5000    1.2500    1.5000    0.2500
```

Determine the product of a 1-by-4 quaternion with itself:

```
q = [1 0 1 0];  
mult = quatmultiply(q)
```

```
mult =  
  
    0    0    2    0
```

# quatmultiply

---

Determine the product of 1-by-4 and 2-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 =
```

```
    0.5000    1.2500    1.5000    0.2500  
    1.9000    1.1000    2.1000   -0.9000
```

## See Also

quatconj, quatdivide, quatinv, quatmod, quatnorm, quatnormalize,  
quatrotate



**Purpose** Calculate norm of quaternion

**Syntax** `n = quatnorm(q)`

**Description** `n = quatnorm(q)` calculates the norm, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns a column vector of `m` norms. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

**Examples** Determine the norm of `q = [1 0 0 0]`:

```
norm = quatnorm([1 0 0 0])
```

```
norm =
```

```
1
```

**See Also** `quatconj`, `quatdivide`, `quatinv`, `quatmod`, `quatmultiply`, `quatnormalize`, `quatrotate`

# quatnormalize

---

**Purpose** Normalize quaternion

**Syntax** `n = quatnormalize(q)`

**Description** `n = quatnormalize(q)` calculates the normalized quaternion, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns an `m`-by-4 matrix of normalized quaternions. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

**Examples** Normalize `q = [1 0 1 0]`:

```
normal = quatnormalize([1 0 1 0])
```

```
normal =
```

```
    0.7071         0    0.7071         0
```

**See Also** `quatconj`, `quatdivide`, `quatinv`, `quatmod`, `quatmultiply`, `quatnorm`, `quatrotate`

**Purpose** Rotate vector by quaternion

**Syntax** `n = quatrotate(q,r)`

**Description** `n = quatrotate(q,r)` calculates the rotated vector, `n`, for a quaternion, `q`, and a vector, `r`. `q` is either an `m`-by-4 matrix containing `m` quaternions, or a single 1-by-4 quaternion. `r` is either an `m`-by-3 matrix, or a single 1-by-3 vector. `n` returns an `m`-by-3 matrix of rotated vectors. Each element of `q` and `r` must be a real number. Additionally, `q` has its scalar number as the first column.

**Examples** 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 =

    -1.0000    1.0000    1.0000
```

Rotate a 1-by-3 vector by a 2-by-4 quaternion:

```
q = [1 0 1 0; 1 0.5 0.3 0.1];
r = [1 1 1];
n = quatrotate(q, r)
```

```
n =

    -1.0000    1.0000    1.0000
     0.8519    1.4741    0.3185
```

Rotate a 2-by-3 vector by a 1-by-4 quaternion:

```
q = [1 0 1 0];
r = [1 1 1; 2 3 4];
```

# quatrotate

---

```
n = quatrotate(q, r)
```

```
n =
```

```
-1.0000    1.0000    1.0000  
-4.0000    3.0000    2.0000
```

Rotate a 2-by-3 vector by a 2-by-4 quaternion:

```
q = [1 0 1 0; 1 0.5 0.3 0.1];
```

```
r = [1 1 1; 2 3 4];
```

```
n = quatrotate(q, r)
```

```
n =
```

```
-1.0000    1.0000    1.0000  
 1.3333    5.1333    0.9333
```

## See Also

quatconj, quatinv, quatmod, quatmultiply, quatnorm, quatnormalize

**Purpose**

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 `fieldsname`, `faces`, `vertices`, and `cdata` that define the geometry in the Handle Graphics patches.
- 'MatFile'  
Selects M-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');
```

# removeNode (Aero.VirtualRealityAnimation)

---

**Purpose** Remove node from virtual reality animation object

**Syntax** `removeNode(h,node)`  
`h.removeNode(node)`

**Description** `removeNode(h,node)` and `h.removeNode(node)` remove the node specified by `node` from the virtual reality animation object `h`. `node` can be either the node name or the node index. This function can remove only one node at a time.

---

**Note** You can use only this function to remove a node added by `addNode`. If you need to remove a node from a previously defined `.wrl` file, use a VRML editor.

---

**Examples** Remove the node, Lynx1.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
h.addNode('Lynx1',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wrl']);
h.removeNode('Lynx1');
```

**See Also** `addNode`

# removeViewpoint (Aero.VirtualRealityAnimation)

---

**Purpose** Remove viewpoint node from virtual reality animation

**Syntax** `removeViewpoint(h,viewpoint)`  
`h.removeViewpoint(viewpoint)`

**Description** `removeViewpoint(h,viewpoint)` and `h.removeViewpoint(viewpoint)` remove the viewpoint specified by `viewpoint` from the virtual reality animation object `h`. `viewpoint` can be either the viewpoint name or the viewpoint index. This function can remove only one viewpoint at a time.

---

**Note** You can use this function to remove a viewpoint added by `addViewpoint`. If you need to remove a viewpoint from a previously defined `.wrl` file, use a VRML editor.

---

**Examples** Remove the node, Lynx1.

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

**See Also** `addViewpoint`

# rrdelta

---

**Purpose** Compute relative pressure ratio

**Syntax** `d = rrdelta(p0, mach, g)`

**Description** `d = rrdelta(p0, mach, g)` computes `m` pressure relative ratios, `d`, from `m` static pressures, `p0`, `m` Mach numbers, `mach`, and `m` specific heat ratios, `g`. `p0` must be in pascals.

**Examples** Determine the relative pressure ratio for three pressures:

```
delta = rrdelta([101325 22632.0672 4328.1393], 0.5, 1.4)
```

```
delta =
```

```
1.1862    0.2650    0.0507
```

Determine the relative pressure ratio for three pressures and three different heat ratios:

```
delta = rrdelta([101325 22632.0672 4328.1393], 0.5, [1.4 1.35 1.4])
```

```
delta =
```

```
1.1862    0.2635    0.0507
```

Determine the relative pressure ratio for three pressures at three different conditions:

```
delta = rrdelta([101325 22632.0672 4328.1393], [0.5 1 2], [1.4 1.35 1.4])
```

```
delta =
```

```
1.1862    0.4161    0.3342
```



**Assumptions  
and  
Limitations**

For cases in which total pressure ratio is desired (Mach number is nonzero), the total pressures are calculated assuming perfect gas (with constant molecular weight, constant pressure specific heat, and constant specific heat ratio) and dry air.

**References**

*Aeronautical Vestpocket Handbook*, United Technologies Pratt & Whitney, August, 1986

**See Also**

rrsigma, rrtheta

# rrsigma

---

**Purpose** Compute relative density ratio

**Syntax** `s = rrsigma(rho, mach, g)`

**Description** `s = rrsigma(rho, mach, g)` computes `m` density relative ratios, `s`, from `m` static densities, `rho`, `m` Mach numbers, `mach`, and `m` specific heat ratios, `g`. `rho` must be in kilograms per meter cubed.

**Examples** Determine the relative density ratio for three densities:

```
sigma = rrsigma([1.225 0.3639 0.0953], 0.5, 1.4)
```

```
sigma =
```

```
1.1297    0.3356    0.0879
```

Determine the relative density ratio for three densities and three different heat ratios:

```
sigma = rrsigma([1.225 0.3639 0.0953], 0.5, [1.4 1.35 1.4])
```

```
sigma =
```

```
1.1297    0.3357    0.0879
```

Determine the relative density ratio for three densities at three different conditions:

```
sigma = rrsigma([1.225 0.3639 0.0953], [0.5 1 2], [1.4 1.35 1.4])
```

```
sigma =
```

```
1.1297    0.4709    0.3382
```

**Assumptions  
and  
Limitations**

For cases in which total density ratio is desired (Mach number is nonzero), the total density is calculated assuming perfect gas (with constant molecular weight, constant pressure specific heat, and constant specific heat ratio) and dry air.

**References**

*Aeronautical Vestpocket Handbook*, United Technologies Pratt & Whitney, August, 1986

**See Also**

rrdelta, rrtheta

# rrtheta

---

**Purpose** Compute relative temperature ratio

**Syntax** `th = rrtheta(t0, mach, g)`

**Description** `th = rrtheta(t0, mach, g)` computes `m` temperature relative ratios, `th`, from `m` static temperatures, `t0`, `m` Mach numbers, `mach`, and `m` specific heat ratios, `g`. `t0` must be in kelvin.

**Examples** Determine the relative temperature ratio for three temperatures:

```
th = rrtheta([273.15 310.9278 373.15], 0.5, 1.4)
```

```
th =
```

```
0.9953    1.1330    1.3597
```

Determine the relative temperature ratio for three temperatures and three different heat ratios:

```
th = rrtheta([273.15 310.9278 373.15], 0.5, [1.4 1.35 1.4])
```

```
th =
```

```
0.9953    1.1263    1.3597
```

Determine the relative temperature ratio for three temperatures at three different conditions:

```
th = rrtheta([273.15 310.9278 373.15], [0.5 1 2], [1.4 1.35 1.4])
```

```
th =
```

```
0.9953    1.2679    2.3310
```

**Assumptions  
and  
Limitations**

For cases in which total temperature ratio is desired (Mach number is nonzero), the total temperature is calculated assuming perfect gas (with constant molecular weight, constant pressure specific heat, and constant specific heat ratio) and dry air.

**References**

*Aeronautical Vestpocket Handbook*, United Technologies Pratt & Whitney, August, 1986

**See Also**

rrdelta, rrsigma

# saveas (Aero.VirtualRealityAnimation)

---

**Purpose** Save virtual reality world associated with virtual reality animation object

**Syntax** `saveas(h, filename)`  
`h.saveas(filename)`

**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.

**Examples** Save the world associated with `h`.

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

# SetTimer (Aero.FlightGearAnimation)

---

**Purpose** Set name of timer for animation of FlightGear flight simulator

**Syntax**

```
SetTimer(h)
h.SetTimer
SetTimer(h, MyFGTimer)
h.SetTimer('MyFGTimer')
```

**Description** SetTimer(h) and h.SetTimer set the name of the MATLAB timer for the animation of the FlightGear flight simulator. SetTimer(h, MyFGTimer) and h.SetTimer('MyFGTimer') set the name of the MATLAB timer for the animation of the FlightGear flight simulator and assign a custom name to the timer.

You can use this function to customize your FlightGear animation object. This customization allows you to simultaneously run multiple FlightGear objects if you want to use

- Multiple FlightGear sessions
- Different ports to connect to those sessions

**Examples** Set the MATLAB timer for animation of the FlightGear animation object, h:

```
h = Aero.FlightGearAnimation
h.SetTimer
```

Set the MATLAB timer used for animation of the FlightGear animation object, h, and assign a custom name, *MyFGTimer*, to the timer:

```
h = Aero.FlightGearAnimation
h.SetTimer('MyFGTimer')
```

**See Also** ClearTimer

# update (Aero.Body)

---

**Purpose** Change body position and orientation as function of time

**Syntax** update(h,t)  
h.update(t)

**Description** update(h,t) and h.update(t) change body position and orientation of body h as a function of time t. t is a scalar in seconds.

---

**Note** This function requires that you load the body geometry and time series data first.

---

**Examples** Update the body b with time in seconds of 5.

```
b=Aero.Body;
b.load('pa24-250_orange.ac','Ac3d');
tsdata = [ ...
          0, 1,1,1, 0,0,0; ...
          10 2,2,2, 1,1,1; ];
b.TimeSeriesSource = tsdata;
b.update(5);
```

**See Also** load



<b>Purpose</b>	Update camera position based on time and position of other Aero.Body objects
<b>Syntax</b>	<code>update(h,newtime,bodies)</code> <code>h.update(newtime,bodies)</code>
<b>Description</b>	<code>update(h,newtime,bodies)</code> and <code>h.update(newtime,bodies)</code> update the camera object, <code>h</code> , position and aim point data based on the new time, <code>newtime</code> , and position of other Aero.Body objects, <code>bodies</code> . This function updates the camera object <code>PrevTime</code> property to <code>newtime</code> .
<b>See Also</b>	<code>Aero.Animation.play</code>

# update (Aero.FlightGearAnimation)

---

**Purpose** Update position data to FlightGear animation object

**Syntax** `update(h,time)`  
`h.update(time)`

**Description** `update(h,time)` and `h.update(time)` update the position data to the FlightGear animation object via UDP. It sets the new position and attitude of body `h`. `time` is a scalar in seconds.

---

**Note** This function requires that you load the time series data and run FlightGear first.

---

**Examples** Configure a body with `TimeSeriesSource` set to `simdata`, then update the body with time `time` equal to 0.

```
h = Aero.FlightGearAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
load simdata;
h.TimeSeriesSource = simdata;
t = 0;
h.update(t);
```

**See Also** `GenerateRunScript`, `initialize`, `play`

**Purpose** Change node position and orientation versus time data

**Syntax** update(h,t)  
h.update(t)

**Description** update(h,t) and h.update(t) change node position and orientation of node h as a function of time t. t is a scalar in seconds.

---

**Note** This function requires that you load the node and time series data first.

---

**Examples** Move the Lynx body.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = [matlabroot, '/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir, 'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir, 'asttkoff.wrl'];
h.initialize();
load takeoffData
h.Nodes{7}.TimeseriesSource = takeoffData;
h.Nodes{7}.TimeseriesSourceType = 'StructureWithTime';
h.Nodes{7}.update(5);
```

**See Also** updateNodes

# updateNodes (Aero.VirtualRealityAnimation)

---

**Purpose** Change virtual reality animation node position and orientation as function of time

**Syntax** updateNodes(h,t)  
h.updateNotes(t)

**Description** updateNodes(h,t) and h.updateNotes(t) change node position and orientation of body h as a function of time t. t is a scalar in seconds.

---

**Note** This function requires that you load the node and time series data first.

---

**Examples** Update the node h with time in 5 seconds.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = [matlabroot, '/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir, 'asttkoff.wrl'], 'f');
h.VRWorldFilename = [tempdir, 'asttkoff.wrl'];
h.initialize();
load takeoffData
h.Nodes{7}.TimeseriesSource = takeoffData;
h.Nodes{7}.TimeseriesSourceType = 'StructureWithTime';
h.Nodes{7}.CoordTransformFcn = @vranimCustomTransform;
h.updateNodes(5);
```

**See Also** addNode, update

# Viewpoint (Aero.Viewpoint)

---

**Purpose** 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.

# VirtualRealityAnimation (Aero.VirtualRealityAnimation)

---

**Purpose** Construct virtual reality animation object

**Syntax** `h = Aero.VirtualRealityAnimation`

**Description** `h = Aero.VirtualRealityAnimation` constructs a virtual reality animation object. The animation object is returned to `h`.  
See `Aero.VirtualRealityAnimation` for further details.

**See Also** `Aero.VirtualRealityAnimation`

**Purpose** Use World Magnetic Model

**Syntax**

```
[xyz, h, dec, dip, f] = wrldmagm(height, lat, lon, dyear)
[xyz, h, dec, dip, f] = wrldmagm(height, lat, lon, dyear,
    '2005')
[xyz, h, dec, dip, f] = wrldmagm(height, lat, lon, dyear,
    '2000')
```

**Description** [xyz, h, dec, dip, f] = wrldmagm(height, lat, lon, dyear) calculates the Earth's magnetic field at a specific location and time using the World Magnetic Model (WMM). The default WMM is WMM-2005, which is valid from January 1, 2005, until December 31, 2009.

Inputs required by wrldmagm are:

height	A scalar value, in meters
lat	A scalar geodetic latitude, in degrees, where north latitude is positive, and south latitude is negative
lon	A scalar geodetic longitude, in degrees, where east longitude is positive, and west longitude is negative
dyear	A scalar decimal year. Decimal year is the desired year in a decimal format to include any fraction of the year that has already passed.

Outputs calculated for the Earth's magnetic field include:

xyz	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)

[xyz, h, dec, dip, f] = wrldmagm(height, lat, lon, dyear, '2005') is an alternate method for calling WMM-2005, or 2005 epoch.

[xyz, h, dec, dip, f] = wrldmagm(height, lat, lon, dyear, '2000') is the method for calling WMM-2000, or 2000 epoch.

## Examples

Calculate the magnetic model 1000 meters over Natick, Massachusetts on July 4, 2005, using WMM-2005:

```
[XYZ, H, DEC, DIP, F] = wrldmagm(1000, 42.283, -71.35, 2005.5068 )
```

XYZ =

```
1.0e+004 *
```

```
1.8976  
-0.5167  
4.9555
```

H =

```
1.9667e+004
```

DEC =

```
-15.2324
```

DIP =



68.3530

F =

5.3315e+004

## **Assumptions and 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's 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.

## **References**

<http://www.ngdc.noaa.gov/seg/WMM/DoDWMM.shtml>

“NOAA Technical Report: The US/UK World Magnetic Model for 2005–2010”

## **See Also**

decyear



# Class Reference

---

# Aero.Animation class

---

<b>Purpose</b>	Visualize aerospace animation																								
<b>Description</b>	Use the Aero.Animation class to visualize flight data without any other tool or toolbox. You only need the Aerospace Toolbox to visualize this data.																								
<b>Construction</b>	Aero.Animation Construct animation object																								
<b>Methods</b>	<table><tr><td>addBody</td><td>Add loaded body to animation object and generate its patches</td></tr><tr><td>createBody</td><td>Create body and its associated patches in animation</td></tr><tr><td>delete</td><td>Destroy animation object</td></tr><tr><td>hide</td><td>Hide animation figure</td></tr><tr><td>initialize</td><td>Create animation object figure and axes and build patches for bodies</td></tr><tr><td>initIfNeeded</td><td>Initialize animation graphics if needed</td></tr><tr><td>moveBody</td><td>Move body in animation object</td></tr><tr><td>play</td><td>Animate Aero.Animation object given position/angle time series</td></tr><tr><td>removeBody</td><td>Remove one body from animation</td></tr><tr><td>show</td><td>Show animation object figure</td></tr><tr><td>updateBodies</td><td>Update bodies of animation object</td></tr><tr><td>updateCamera</td><td>Update camera in animation object</td></tr></table>	addBody	Add loaded body to animation object and generate its patches	createBody	Create body and its associated patches in animation	delete	Destroy animation object	hide	Hide animation figure	initialize	Create animation object figure and axes and build patches for bodies	initIfNeeded	Initialize animation graphics if needed	moveBody	Move body in animation object	play	Animate Aero.Animation object given position/angle time series	removeBody	Remove one body from animation	show	Show animation object figure	updateBodies	Update bodies of animation object	updateCamera	Update camera in animation object
addBody	Add loaded body to animation object and generate its patches																								
createBody	Create body and its associated patches in animation																								
delete	Destroy animation object																								
hide	Hide animation figure																								
initialize	Create animation object figure and axes and build patches for bodies																								
initIfNeeded	Initialize animation graphics if needed																								
moveBody	Move body in animation object																								
play	Animate Aero.Animation object given position/angle time series																								
removeBody	Remove one body from animation																								
show	Show animation object figure																								
updateBodies	Update bodies of animation object																								
updateCamera	Update camera in animation object																								

## Properties

Bodies	Specify name of animation object
Camera	Specify camera that animation object contains
Figure	Specify name of figure object
FigureCustomizationFcn	Specify figure customization function
FramesPerSecond	Animation rate
Name	Specify name of animation object
TCurrent	Current time
TFinal	End time
TimeScaling	Scaling time
TStart	Start time

## See Also

`Aero.FlightGearAnimation`  
`Aero.VirtualRealityAnimation`  
“Using Aero.Animation Objects” on page 2-26

# Aero.Body

---

**Purpose** Create body object for use with animation object

**Syntax** `h = Aero.Body`

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

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

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

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

## Constructor Summary

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

## Method Summary

Method	Description
<code>findstartstoptimes</code>	Return start and stop times of time series data.

Method	Description
generatePatches	Generate patches for body with loaded face, vertex, and color data.
load	Get geometry data from source.
move	Change Aero.Body position and orientation.
update	Changes body position and orientation versus time data.

## Property Summary

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

Property	Description	Values
TimeseriesSourceType	Specify the type of time series data stored in 'TimeseriesSource'. Five values are available. They are listed in the following table. The default value is 'Array6DoF'.	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:

'Timeseries'	<p>MATLAB time series data with six values per time:</p> <p>lat lon alt phi theta psi</p> <p>The values are resampled.</p>
'StructureWithTime'	<p>Simulink struct with time (for example, Simulink root output logging 'Structure with time'):</p> <ul style="list-style-type: none"> <li>signals(1).values: lat lon alt</li> <li>signals(2).values: phi theta psi</li> </ul> <p>Signals are linearly interpolated vs. time using interp1.</p>



'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'.

## See Also

Aero.Geometry

# Aero.Camera

---

**Purpose** 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.

## 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 <code>Aero.Body</code> objects.

## Property Summary

Property	Description	Values
<code>CoordTransformFcn</code>	Specify a function that controls the coordinate transformation.	MATLAB array
<code>PositionFcn</code>	Specify a function that controls the position of a camera relative to an animation body.	MATLAB array
<code>Position</code>	Specify position of camera.	MATLAB array [-150, -50, 0]

Property	Description	Values
Offset	Specify offset of camera.	MATLAB array [-150, -50, 0]
AimPoint	Specify aim point of camera.	MATLAB array [0, 0, 0]
UpVector	Specify up vector of camera.	MATLAB array [0, 0, -1]
ViewAngle	Specify view angle of camera.	MATLAB array {3}
ViewExtent	Specify view extent of camera.	MATLAB array {[ -50, 50]}
xlim	Specify x-axis limit of camera.	MATLAB array {[ -50, 50]}
ylim	Specify y-axis limit of camera.	MATLAB array {[ -50, 50]}
zlim	Specify z-axis limit of camera.	MATLAB array {[ -50, 50]}
PrevTime	Specify previous time of camera.	MATLAB array {0}
UserData	Specify custom data.	MATLAB array {[ ]}

## See Also

Aero.Geometry

# Aero.FlightGearAnimation

---

**Purpose** 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`.

## Constructor

Method	Description
<code>fanimation</code>	Construct FlightGear animation object.

## Method Summary

Method	Description
<code>delete</code>	Destroy FlightGear animation object.
<code>initialize</code>	Set up FlightGear animation object.
<code>play</code>	Animate FlightGear flight simulator using given position/angle time series.
<code>update</code>	Update position data to FlightGear animation object.

## Property Summary

Properties	Description
<code>TimeseriesSource</code>	Specify variable that contains the time series data.
<code>TimeseriesSource-Type</code>	Specify the type of time series data stored in 'TimeseriesSource'. Five values are available. They are listed in the following table. The default value is 'Array6DoF'.
<code>TimeseriesReadFcn</code>	Specify a function to read the time series data if 'TimeseriesSourceType' is 'Custom'.
<code>TimeScaling</code>	Specify the seconds of animation data per second of wall-clock time. The default ratio is 1.

Properties	Description
FramesPerSecond	Specify the number of frames per second used to animate the 'TimeseriesSource'. The default value is 12 frames per second.
FlightGearVersion	Select your FlightGear software version: '0.9.3', '0.9.8', '0.9.9', '0.9.10', or '1.0'. The default version is '1.0'.
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'.
FlightGearBase-Directory	Specify the name of your FlightGear installation directory. The default value is 'D:\Applications\FlightGear'.
GeometryModelName	Specify the name of the folder containing the desired model geometry in the <i>FlightGear\data\Aircraft</i> directory. The default value is 'HL20'.
DestinationIp-Address	Specify your destination IP address. The default value is '127.0.0.1'.
DestinationPort	Specify your network flight dynamics model (fdm) port. This destination port should be an unused port that you can use when you launch FlightGear. The default value is '5502'.
AirportId	Specify the airport ID. The list of supported airports is available in the FlightGear interface, under <b>Location</b> . 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.

# Aero.FlightGearAnimation

---

Properties	Description
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.

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:  
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'.

## Examples

Construct a FlightGear animation object, h:

```
h = fganimation
```

## See Also

fganimation, generaterunscript, play

# Aero.Geometry

---

**Purpose** 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.	string
Source	Specify geometry data source.	string {'Auto', 'Variable', 'MatFile', 'Ac3dFile', 'Custom'}
Reader	Specify geometry reader.	MATLAB array
FaceVertexColor-Data	Specify the color of the geometry face vertex.	MATLAB array

**See Also** `read`



**Purpose** 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 `.wr1` 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 `.wr1` file has a node object created.

## Constructor Summary

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

## Method Summary

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

## Property Summary

Property	Description	Values
Name	Specify name of the node object.	string

Property	Description	Values
VRNode	Return the handle to the vrnode object associated with the node object (see the <i>Simulink 3D Animation User's Guide</i> ).	MATLAB array
CoordtransformFcn	Specify a function that controls the coordinate transformation.	MATLAB array
TimeseriesSource	Specify time series source.	MATLAB array
Timeseries-Source Type	Specify the type of time series data stored in 'TimeseriesSource'. Five values are available. They are listed in the following table. The default value is 'Array6DoF'.	string
Timeseries-ReadFcn	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:

'Timeseries'	<p>MATLAB time series data with six values per time:</p> <p>lat lon alt phi theta psi</p> <p>The values are resampled.</p>
'StructureWithTime'	<p>Simulink struct with time (for example, Simulink root output logging 'Structure with time'):</p> <ul style="list-style-type: none"><li>• signals(1).values: lat lon alt</li><li>• signals(2).values: phi theta psi</li></ul> <p>Signals are linearly interpolated vs. time using interp1.</p>
'Array6DoF'	<p>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.</p>
'Array3DoF'	<p>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.</p>
'Custom'	<p>Position and angle data is retrieved from 'TimeseriesSource' by the currently registered 'TimeseriesReadFcn'.</p>

# Aero.Viewpoint

---

**Purpose** 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.	string
Node	Specify node object that contains the viewpoint node.	MATLAB array

**Purpose** 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.

## Constructor Summary

Constructor	Description
<code>VirtualRealityAnimation</code>	Construct virtual reality animation object.

## Method Summary

Method	Description
<code>addNode</code>	Add existing node to current virtual reality world.
<code>addRoute</code>	Add VRML ROUTE statement to virtual reality animation.
<code>addViewpoint</code>	Add viewpoint for virtual reality animation.
<code>delete</code>	Destroy virtual reality animation object.
<code>initialize</code>	Create and populate virtual reality animation object.
<code>nodeInfo</code>	Create list of nodes associated with virtual reality animation object.
<code>play</code>	Animate virtual reality world for given position and angle in time series data.
<code>removeNode</code>	Remove node from virtual reality animation object.
<code>removeViewpoint</code>	Remove viewpoint node from virtual reality animation.

# Aero.VirtualRealityAnimation

---

Method	Description
saveas	Save virtual reality world associated with virtual reality animation object.
updateNodes	Set new translation and rotation of moveable items in virtual reality animation.

## Notes on Aero.VirtualRealityAnimation Methods

Aero.VirtualRealityAnimation methods that change the current virtual reality world use a temporary .wr1 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 .wr1 file.
- When you delete the virtual reality animation object, this action deletes the temporary file.
- Use the saveas method to save the temporary .wr1 file.
- These methods do not affect user-created .wr1 files.

## Property Summary

Property	Description	Values
Name	Specify name of the animation object.	string

<b>Property</b>	<b>Description</b>	<b>Values</b>
VRWorld	Returns the vrworld object associated with the animation object.	MATLAB array
VRWorldFilename	Specify the .wr1 file for the vrworld.	string
VRWorld-OldFilename	Specify the old .wr1 files for the vrworld.	MATLAB array
VRWorld-TempFilename	Specify the temporary .wr1 file for the animation object.	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 time, in seconds.	double
TFinal	Specify end time, in seconds.	double
TCurrent	Specify current time, in seconds.	double

# Aero.VirtualRealityAnimation

---

<b>Property</b>	<b>Description</b>	<b>Values</b>
FramesPerSecond	Specify rate, in frames per second.	double
ShowSaveWarning	Specify save warning display setting.	double

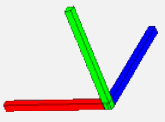
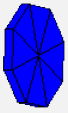

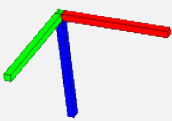
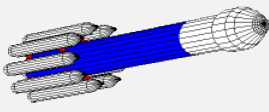







# AC3D Files and Thumbnails

---

## Overview

Aerospace Toolbox demos use the following AC3D files, located in the *matlabroot\toolbox\aero\astdemos* directory.

Thumbnail	AC3D File
	ac3d_xyzisrgb.ac
	blueoctagon.ac
	bluwedge.ac
	body_xyzisrgb.ac
	delta2.ac
	greenarrow.ac
	pa24_250_blue.ac
	pa24_250_orange.ac

Thumbnail	AC3D File
	redwedge.ac
	testrocket.ac



## A

- AC3D files A-2
- addBody (Aero.Animation) function 4-8
- addNode (Aero.VirtualRealityAnimation) function 4-2
- addRoute (Aero.VirtualRealityAnimation) function 4-3
- addViewpoint (Aero.VirtualRealityAnimation) function 4-4
- Aero.Animation
  - demo 2-27
  - flight simulator overview 2-26
  - introducing 2-26
- Aero.Animation object 4-7
- Aero.Body object 5-4
- Aero.Camera object 5-8
- Aero.FlightGearAnimation
  - demo 2-56
  - introducing 2-26
- Aero.FlightGearAnimation object 5-10
- Aero.Geometry object 5-14
- Aero.Node object 5-15
- Aero.Viewpoint function 5-18
- Aero.VirtualRealityAnimation
  - demo 2-36
  - flight simulator overview 2-35
  - introducing 2-26
  - virtual world 2-37
- Aero.VirtualRealityAnimation object 5-19
- Aerospace Toolbox
  - 3-D flight data playback 2-26
  - about 1-2
  - AC3D files A-2
  - animation objects 2-26
  - coordinate systems 2-2
  - flight data file access 2-14
  - online help 1-5
  - related products 1-4
- aerospace units

- definition 2-12
- airspeed function 4-6
- alphabeta function 4-34
- angle2dcm function 4-36
- angle2quat function 4-38
- animation objects
  - introducing 2-26
- atmoscira function 4-43
- atmoscoesa function 4-40
- atmosisa function 4-50
- atmoslapse function 4-53
- atmosnonstd function 4-63
- atmosnrmlmsise00 function 4-56
- atmospalt function 4-68

## B

- Bodies
  - properties 4-24
- Body (Aero.Body) function 4-70
- body coordinates 2-4

## C

- Camera
  - properties 4-25
- Camera (Aero.Camera) function 4-71
- ClearTimer(Aero.FlightGearAnimation) function 4-72
- convacc function 4-73
- convang function 4-74
- convangacc function 4-75
- convangvel function 4-76
- convdensity function 4-77
- convforce function 4-78
- convlength function 4-79
- convmass function 4-80
- convpres function 4-81
- convtemp function 4-82
- convvel function 4-83

- coordinate systems 2-2
  - approximations 2-3
  - body coordinates 2-4
  - definition 2-2
  - display 2-10
  - Earth-centered coordinates 2-9
  - ECEF coordinates 2-10
  - ECI coordinates 2-9
  - geocentric and geodetic latitudes 2-7
  - modeling 2-4
  - motion with respect to other planets 2-3
  - navigation 2-7
  - NED coordinates 2-8
  - references 2-11
  - rotational degrees of freedom 2-4 2-6
  - translational degrees of freedom 2-4 to 2-5
  - wind coordinates 2-5
- correctairspeed function 4-84
- createBody (Aero.Animation) function 4-9

## D

- datcomimport function 4-86
- dcm2alphabeta function 4-122
- dcm2angle function 4-124
- dcm2latlon function 4-127
- dcm2quat function 4-129
- dcmbody2wind function 4-130
- dcmecef2ned function 4-132
- decyear function 4-134
- delete (Aero.Animation) function 4-11
- delete (Aero.FlightGearAnimation)
  - function 4-136
- delete (Aero.VirtualRealityAnimation)
  - function 4-137
- demos
  - AC3D files A-2
  - astfganim 2-48
  - astimportddatcom 2-14
  - astmlanim 2-26

- astvranim 2-35
  - type astdatcom.in 2-14
- digital DATCOM
  - examining 2-15
  - importing 2-14
  - overview 2-14
  - plotting aerodynamic coefficients 2-22
- digital DATCOM file
  - example 2-14
  - importing data 2-15
- dpressure function 4-138

## E

- Earth-centered coordinates 2-9
- ECEF coordinates 2-10
- ecef2lla function 4-140
- ECI coordinates 2-9

## F

- fganimation (Aero.FlightGearAnimation)
  - function 4-142
- Figure
  - properties 4-26
- FigureCustomizationFcn
  - properties 4-27
- findstartstoptimes (Aero.Body)
  - function 4-143
- findstartstoptimes (Aero.Node)
  - function 4-144
- FlightGear
  - flight simulator overview 2-48
  - installing 2-52
  - obtaining 2-49
- FramesPerSecond
  - properties 4-28
- functions
  - addBody (Aero.Animation) 4-8

addNode  
     (Aero.VirtualRealityAnimation) 4-2  
 AddRoute  
     (Aero.VirtualRealityAnimation) 4-3  
 addViewpoint(Aero.VirtualRealityAnimation) 4-4  
 Aero.Viewpoint 5-18  
 airspeed 4-6  
 alphabeta 4-34  
 angle2dcm 4-36  
 angle2quat 4-38  
 atmoscira 4-43  
 atmoscoesa 4-40  
 atmosisa 4-50  
 atmoslapse 4-53  
 atmosnonstd 4-63  
 atmosnrlmsise00 4-56  
 atmospalt 4-68  
 Body (Aero.Body) 4-70  
 Camera (Aero.Camera) 4-71  
 ClearTimer(Aero.FlightGearAnimation) 4-72  
 convacc 4-73  
 convang 4-74  
 convangacc 4-75  
 convangvel 4-76  
 convdensity 4-77  
 convforce 4-78  
 convlength 4-79  
 convmass 4-80  
 convpres 4-81  
 convtemp 4-82  
 convvel 4-83  
 correctairspeed 4-84  
 createBody (Aero.Animation) 4-9  
 datcomimport 4-86  
 dcm2alphabeta 4-122  
 dcm2angle 4-124  
 dcm2latlon 4-127  
 dcm2quat 4-129  
 dcmbody2wind 4-130  
 dcmecef2ned 4-132  
 decyear 4-134  
 delete (Aero.Animation) 4-11  
 delete  
     (Aero.FlightGearAnimation) 4-136  
 delete  
     (Aero.VirtualRealityAnimation) 4-137  
 dpressure 4-138  
 ecef2lla 4-140  
 fganimation  
     (Aero.FlightGearAnimation) 4-142  
 findstartstoptimes (Aero.Body) 4-143  
 findstartstoptimes (Aero.Node) 4-144  
 generatePatches (Aero.Body) 4-145  
 GenerateRunScript  
     (Aero.FlightGearAnimation) 4-146  
 geoc2geod 4-148  
 geocradius 4-150  
 geod2geoc 4-152  
 geoiddegm96 4-154  
 gravitywgs84 4-157  
 hide (Aero.Animation) 4-12  
 initialize (Aero.Animation) 4-14  
 initialize  
     (Aero.FlightGearAnimation) 4-165  
 initialize  
     (Aero.VirtualRealityAnimation) 4-166  
 initIfNeeded (Aero.Animation) 4-13  
 juliandate 4-167  
 leapyear 4-169  
 lla2ecef 4-170  
 load (Aero.Body) 4-172  
 machnumber 4-174  
 mjuliandate 4-176  
 move (Aero.Body) 4-179  
 move (Aero.Node) 4-180  
 moveBody (Aero.Animation) 4-15  
 Node (Aero.Node) 4-182  
 nodeInfo  
     (Aero.VirtualRealityAnimation) 4-183  
 play (Aero.Animation) 4-16

play (Aero.FlightGearAnimation) 4-184  
play  
    (Aero.VirtualRealityAnimation) 4-188  
quat2angle 4-191  
quat2dcm 4-193  
quatconj 4-195  
quatdivide 4-196  
quatinv 4-197  
quatmod 4-198  
quatmultiply 4-199  
quatnorm 4-201  
quatnormalize 4-202  
quatrotate 4-203  
read (Aero.Geometry) 4-205  
removeBody (Aero.Animation) 4-20  
removeNode  
    (Aero.VirtualRealityAnimation) 4-206  
removeViewpoint  
    (Aero.VirtualRealityAnimation) 4-207  
rrdelta 4-208  
rrsigma 4-210  
rrtheta 4-212  
saveas  
    (Aero.VirtualRealityAnimation) 4-214  
SetTimer(Aero.FlightGearAnimation) 4-215  
show (Aero.Animation) 4-21  
update (Aero.Body) 4-216  
update (Aero.Camera) 4-217  
update  
    (Aero.FlightGearAnimation) 4-218  
update (Aero.Node) 4-219  
updateBodies (Aero.Animation) 4-22  
updateCamera (Aero.Animation) 4-23  
updateNodes  
    (Aero.VirtualRealityAnimation) 4-220  
Viewpoint (Aero.Viewpoint) 4-221  
VirtualRealityAnimation  
    (Aero.VirtualRealityAnimation) 4-222  
wrldmagn 4-223

**G**

generatePatches (Aero.Body) function 4-145  
GenerateRunScript  
    (Aero.FlightGearAnimation)  
        function 4-146  
geoc2geod function 4-148  
geocentric and geodetic latitudes 2-7  
geocradius function 4-150  
geod2geoc function 4-152  
geoidegm96 function 4-154  
Geometry (Aero.Geometry) object 4-156  
gravitywgs84 function 4-157

**H**

hide (Aero.Animation) function 4-12

**I**

importing  
    digital DATCOM data 2-14  
initialize (Aero.Animation) function 4-14  
initialize (Aero.FlightGearAnimation)  
    function 4-165  
initialize (Aero.VirtualRealityAnimation)  
    function 4-166  
initIfNeeded (Aero.Animation) function 4-13

**J**

juliandate function 4-167

**L**

leapyear function 4-169  
lla2ecef function 4-170  
load (Aero.Body) function 4-172

**M**

machnumber function 4-174



mjuliandate function 4-176  
 modeling 2-4  
 move (Aero.Body) function 4-179  
 move (Aero.Node) function 4-180  
 moveBody (Aero.Animation) function 4-15

## N

Name  
     properties 4-29  
 navigation 2-7  
 NED coordinates 2-8  
 Node (Aero.Node) function 4-182  
 nodeInfo (Aero.VirtualRealityAnimation)  
     function 4-183

## O

objects  
     Aero.Animation 4-7  
     Aero.Body 5-4  
     Aero.Camera 5-8  
     Aero.FlightGearAnimation 5-10  
     Aero.Geometry 5-14  
     Aero.Node 5-15  
     Aero.VirtualRealityAnimation 5-19  
     Geometry (Aero.Geometry) 4-156  
 online help 1-5

## P

play (Aero.Animation) function 4-16  
 play (Aero.FlightGearAnimation)  
     function 4-184  
 play (Aero.VirtualRealityAnimation)  
     function 4-188  
 properties  
     Bodies 4-24  
     Camera 4-25  
     Figure 4-26  
     FigureCustomizationFcn 4-27

FramesPerSecond 4-28  
 Name 4-29  
 TCurrent 4-30  
 TFinal 4-31  
 TimeScaling 4-33  
 TStart 4-32

## Q

quat2angle function 4-191  
 quat2dcm function 4-193  
 quatconj function 4-195  
 quatdivide function 4-196  
 quatinv function 4-197  
 quatmod function 4-198  
 quatmultiply function 4-199  
 quatnorm function 4-201  
 quatnormalize function 4-202  
 quatrotate function 4-203

## R

read (Aero.Geometry) function 4-205  
 removeBody (Aero.Animation) function 4-20  
 removeNode (Aero.VirtualRealityAnimation)  
     function 4-206  
 removeViewpoint  
     (Aero.VirtualRealityAnimation)  
     function 4-207  
 rotational degrees of freedom 2-4 2-6  
 rrdelta function 4-208  
 rrsigma function 4-210  
 rrtheta function 4-212

## S

saveas (Aero.VirtualRealityAnimation)  
     function 4-214  
 SetTimer(Aero.FlightGearAnimation)  
     function 4-215  
 show (Aero.Animation) function 4-21

**T**

TCurrent  
    properties 4-30  
TFinal  
    properties 4-31  
TimeScaling  
    properties 4-33  
translational degrees of freedom 2-4 to 2-5  
TStart  
    properties 4-32

**U**

update (Aero.Body) function 4-216  
update (Aero.Camera) function 4-217  
update (Aero.FlightGearAnimation)  
    function 4-218  
update (Aero.Node) function 4-219

updateBodies (Aero.Animation) function 4-22  
updateCamera (Aero.Animation) function 4-23  
updateNodes  
    (Aero.VirtualRealityAnimation)  
    function 4-220

**V**

Viewpoint (Aero.Viewpoint) function 4-221  
virtual world 2-37  
VirtualRealityAnimation  
    (Aero.VirtualRealityAnimation)  
    function 4-222

**W**

wind coordinates 2-5  
wrldmagm function 4-223