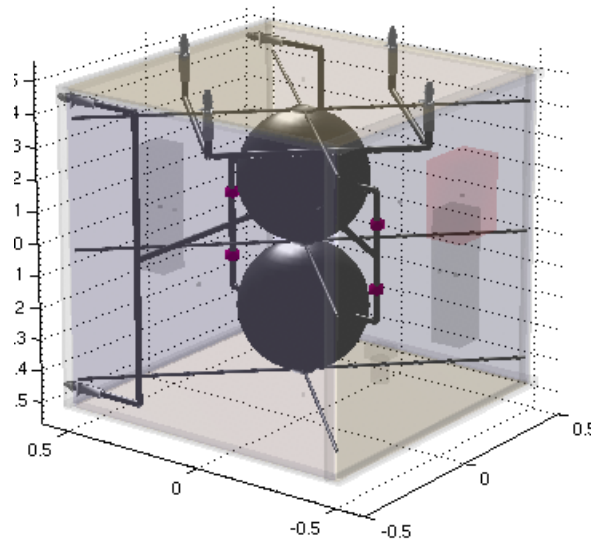




# Spacecraft Control Toolbox

## User's Guide

### Release 2017.1



---

This software described in this document is furnished under a license agreement. The software may be used, copied or translated into other languages only under the terms of the license agreement.

Spacecraft Control Toolbox

June 20, 2017

©Copyright 1996-2017 by Princeton Satellite Systems, Inc. All rights reserved.

Any provision of Princeton Satellite System Software to the U.S. Government is with Restricted Rights as follows: Use, duplication, or disclosure by the Government is subject to restrictions set forth in subparagraphs (a) through (d) of the Commercial Computer Restricted Rights clause at FAR 52.227-19 when applicable, or in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, and in similar clause in the NASA FAR Supplement. Any provision of Princeton Satellite Systems documentation to the U.S. Government is with Limited Rights. The contractor/manufacturer is Princeton Satellite Systems, Inc., 6 Market St. Suite 926, Plainsboro, New Jersey 08536.

Wavefront is a trademark of Alias Systems Corporation. MATLAB is a trademark of the MathWorks.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

**Princeton Satellite Systems, Inc.**  
6 Market St. Suite 926  
Plainsboro, New Jersey 08536

Technical Support/Sales/Info: <http://www.psatellite.com>

---

# CONTENTS

---

<b>Spacecraft Control Toolbox</b>	<b>i</b>
<b>Contents</b>	<b>iii</b>
<b>List of Examples</b>	<b>ix</b>
<b>List of Figures</b>	<b>xi</b>
<b>I Toolbox Basics</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Organization . . . . .	3
1.2 Requirements . . . . .	4
1.3 Installation . . . . .	4
1.4 Getting Started . . . . .	4
<b>2 Getting Help</b>	<b>7</b>
2.1 MATLAB's Built-in Help System . . . . .	7
2.2 Command Line Help . . . . .	11
2.3 FileHelp . . . . .	14
2.4 Searching in File Help . . . . .	15
2.5 Finder . . . . .	16
2.6 DemoPSS . . . . .	17
2.7 Graphical User Interface Help . . . . .	17
2.8 Finder . . . . .	18
2.9 Technical Support . . . . .	19
<b>3 Basic Functions</b>	<b>21</b>
3.1 Introduction . . . . .	21
3.2 Header Format . . . . .	21
3.3 Function Features . . . . .	23
3.4 Example Functions . . . . .	26
3.5 GUI Tools in the Toolbox . . . . .	33
<b>4 SCT Tutorial</b>	<b>35</b>
<b>5 Coordinates</b>	<b>39</b>
5.1 Transformation Matrices . . . . .	39
5.2 Quaternions . . . . .	40
5.3 Coordinate Frames . . . . .	41
<b>6 CAD</b>	<b>43</b>

6.1	Introduction	43
6.2	Building a Spacecraft Model Using <code>BuildCADModel</code>	43
6.3	Visualizing your CAD Model	64
<b>7</b>	<b>Attitude Simulations</b>	<b>69</b>
7.1	Dynamics Models	69
7.2	Wire Model	69
7.3	Tree Model	74
<b>8</b>	<b>Ephemeris</b>	<b>77</b>
8.1	Overview	77
8.2	Almanac functions	77
8.3	JPL Ephemeris	78
<b>9</b>	<b>Plotting Tool</b>	<b>81</b>
9.1	Introduction	81
9.2	Getting Started	82
9.3	GUI Layout and Features	83
9.4	Creating Figures	85
9.5	Grouping Variables	86
9.6	Excluding Variables	87
9.7	Deriving New Data	87
9.8	Animation	88
9.9	Templates	88
9.10	Plotting Output from Custom Simulation Software	89
9.11	Summary	93
<b>10</b>	<b>Pointing Budgets</b>	<b>95</b>
10.1	Pointing Budget	95
10.2	Pointing Budget GUI	96
<b>11</b>	<b>Designing Controllers</b>	<b>99</b>
11.1	Using the block diagram	99
11.2	Linear Quadratic Control	100
11.3	Single-Input-Single-Output	100
11.4	Eigenstructure Assignment	101
<b>12</b>	<b>CubeSat</b>	<b>107</b>
12.1	CubeSat Modeling	108
12.2	Simulation	110
12.3	Mission Planning	113
12.4	Visualization	114
12.5	Subsystems Modeling	116
<b>II</b>	<b>Attitude Control Design Examples</b>	<b>117</b>
<b>13</b>	<b>Attitude Control</b>	<b>119</b>
13.1	ACS Overview	119
13.2	Design Examples	119
<b>14</b>	<b>ACS Example</b>	<b>123</b>



14.1	The MAP Mission . . . . .	123
14.2	Control Modes . . . . .	124
14.3	Sensing and Actuation . . . . .	124
14.4	Control System Design . . . . .	124
14.5	Simulation Results . . . . .	124
14.6	The TMAPSim Script . . . . .	125
14.7	Summary . . . . .	132
<b>III Orbit Propagation and Maneuvers</b>		<b>133</b>
<b>15</b>	<b>Orbit Module</b>	<b>135</b>
15.1	Overview . . . . .	135
<b>16</b>	<b>Orbit Propagation</b>	<b>137</b>
16.1	Low Thrust Orbit Raising . . . . .	137
16.2	Reentry Due to Drag . . . . .	138
16.3	Topex Ephemerides . . . . .	139
16.4	Integration Tolerance . . . . .	143
<b>17</b>	<b>Orbit Maneuvers</b>	<b>145</b>
17.1	Orbit Maneuver Functions . . . . .	145
17.2	Launch and Early Orbit Operations . . . . .	146
17.3	Drag Correction Example . . . . .	147
17.4	Glideslopes . . . . .	150
<b>18</b>	<b>Orbit Propagator GUI</b>	<b>153</b>
18.1	The Orbit Propagator Tool . . . . .	153
18.2	Using the Graphical User Interface . . . . .	154
18.3	Plugin Functions . . . . .	157
18.4	Building a Plugin Function . . . . .	159
18.5	Reference Frames . . . . .	162
18.6	Sun . . . . .	163
18.7	Interfacing to STK . . . . .	163
18.8	Ground Coverage . . . . .	163
18.9	Orbit Propagation API . . . . .	164
<b>19</b>	<b>Fuel Budgets</b>	<b>167</b>
19.1	Fuel Budget . . . . .	167
<b>20</b>	<b>Orbit References</b>	<b>173</b>
20.1	Orbits . . . . .	173
20.2	Web Sites . . . . .	173
<b>IV Subsystems Analysis</b>		<b>175</b>
<b>21</b>	<b>Imaging</b>	<b>177</b>
21.1	ImageMatching . . . . .	177
21.2	ImageProcessing . . . . .	177
<b>22</b>	<b>Link Module</b>	<b>179</b>

22.1 Link Utilities . . . . .	179
22.2 Optical . . . . .	180
22.3 RF . . . . .	181
22.4 Downlink Analysis Example . . . . .	183
22.5 LADAR . . . . .	184
22.6 RADAR . . . . .	186
<b>23 Power</b>	<b>189</b>
<b>24 Propulsion Module</b>	<b>191</b>
24.1 Module Overview . . . . .	191
24.2 Solid Rocket Motor . . . . .	193
24.3 Blowdown Curve . . . . .	193
24.4 Electric propulsion . . . . .	194
24.5 Launch vehicle analysis . . . . .	195
<b>25 Thermal Module</b>	<b>197</b>
25.1 Module Overview . . . . .	197
25.2 Thermal Properties . . . . .	197
25.3 Thermal Analysis . . . . .	199
<b>V Attitude and Orbit Estimation</b>	<b>203</b>
<b>26 Introduction</b>	<b>205</b>
26.1 Overview . . . . .	205
<b>27 General Estimation</b>	<b>207</b>
27.1 Introduction . . . . .	207
27.2 Fixed Gain Estimators . . . . .	207
27.3 Variable Gain Estimators . . . . .	208
27.4 Extended Kalman Filter . . . . .	208
27.5 Continuous Discrete Extended Kalman Filter . . . . .	210
27.6 Unscented Kalman Filter . . . . .	210
<b>28 Stellar</b>	<b>213</b>
28.1 Introduction . . . . .	213
28.2 Star Catalogs . . . . .	213
28.3 Generating Star Data . . . . .	216
28.4 Centroiding . . . . .	217
28.5 Star Identification . . . . .	218
28.6 Kalman Filtering . . . . .	221
<b>29 Unscented Kalman Filter</b>	<b>223</b>
29.1 Introduction . . . . .	223
29.2 Pressure Kalman Filter . . . . .	223
<b>Appendices</b>	<b>226</b>
<b>A GOES Interface</b>	<b>229</b>
A.1 Introduction . . . . .	229

---

A.2	Loading GOES Data . . . . .	229
A.3	Getting GOES Longitude . . . . .	230
A.4	GOES functions . . . . .	230
A.5	For More Information . . . . .	230
A.6	References . . . . .	233
<b>B</b>	<b>Using Databases</b>	<b>235</b>
B.1	The Constant Database . . . . .	235
B.2	Merging Constant Databases . . . . .	236
<b>C</b>	<b>GUI Plug Ins</b>	<b>239</b>
C.1	GUI PlugIn Demo . . . . .	239
C.2	Writing Your Own GUI Function . . . . .	240
	<b>Index</b>	<b>247</b>



---

# CODE EXAMPLES

---

- 3.1 Earth radiation . . . . . 26
- 3.2 Torquer and thruster comparison results . . . . . 27
- 3.3 Antenna patch . . . . . 28
- 3.4 Hall thruster . . . . . 29
- 3.5 Solar flux from the sun . . . . . 30
- 6.1 Simple patch with one triangle . . . . . 44
- 7.1 Rigid Body Attitude Simulation . . . . . 70



---

# LIST OF FIGURES

---

2.1	MATLAB Help - Supplemental Software	8
2.2	Toolbox Documentation Main Page, R2016b	9
2.3	Toolbox Documentation, R2011b	9
2.4	Toolbox Demos	10
2.5	The file help GUI	14
2.6	The demo GUI	17
2.7	On-line Help	18
3.1	Atmospheric density from <code>AtmDens2</code>	23
3.2	Dual Spin Stability	27
3.3	Elliptical orbit from <code>RVFromKepler</code>	27
3.4	A sine wave using <code>Plot2D</code>	31
3.5	Pivot mechanism	31
3.6	Finder window	33
4.1	ECI and LVLH Coordinate Frames	35
4.2	Control torques	37
4.3	Control torque, ideal and with a 0.25 sec delay	38
5.1	Frames A and B	39
5.2	Selenographic to ECI frame	41
5.3	Areocentric frame	42
6.1	Component geometry and frames	45
6.2	Body geometry and frames	46
6.3	Cylinder demo of geometry and frames. Hinge vectors are in black, $r_A$ is cyan and $r_B$ is magenta. The axes are coded X (blue), Y (green), and Z (red).	46
6.4	Simple Sat	48
6.5	CreateComponent color display	56
6.6	BuildCADModel Component View	58
6.7	BuildCADModel Body view	59
6.8	Vehicle tab with Show Vehicle pushed	59
6.9	3D View obtained by pushing Show Vehicle, left, and another sail model shown using Show Spacecraft in Orbit, right	60
6.10	Subsystem Transparency and Exploded views	60
6.11	2D View from clicking Show 2D Plan	61
6.12	Tie fighter as loaded from a dxf file	62
6.13	DrawSCPlanPlugIn showing the XYZSat model	64
6.14	DrawSCPlanPlugIn showing the XYZSat model after a change to $g$	66
6.15	DrawSCPlanPlugIn showing a satellite with vectors. The green vector is nadir, the yellow vector the sun, and the red vector the orbital velocity.	66

7.1	Wire model	69
7.2	Wire simulation: Results	73
7.3	Topological Tree	74
7.4	Tree Simulation: One of several plots.	76
8.1	Planet trajectories as generated by the built-in demo of SolarSysJPL	80
9.1	Plotting Tool GUI with Newly Loaded Data File	82
9.2	Diagram of the Plotting Tool GUI	83
9.3	Plotting Tool Generated Figure	86
9.4	Animation GUI Snap-Shot	89
10.1	Open the GUI	96
10.2	Entering the first error	97
10.3	Entering a second error	97
10.4	Save window	97
11.1	Block diagram	99
11.2	Step response	101
11.3	LQ GUI	102
11.4	Step response from the GUI	103
11.5	SISO inputs	103
11.6	MapIO	104
11.7	SISO step response	104
11.8	Eigenstructure design GUI	106
11.9	Step response with eigenstructure assignment	106
12.1	Model of 2U CubeSat	109
12.2	Orbit simulation timestep results, simple on the left with with disturbances on the right.	111
12.3	Orbit evolution for an initial separation of 10 meters	111
12.4	CubeSatSimulation example results	112
12.5	Observation time windows	113
12.6	RapidSwath built-in demo results	114
12.7	Orbit visualization with PlotOrbit and GroundTrack	114
12.8	CubeSat model with deployable solar panels viewed with DrawCubeSat	115
12.9	Spacecraft visualization with sight lines using DrawSpacecraftInOrbit	115
12.10	Playback demo using PlaybackOrbitSim	116
13.1	Solar pressure torque map for double-gimballed solar array	120
14.1	The Microwave Anisotropy Probe	123
14.2	Motion of the unit vector	131
14.3	Precession of the unit vector	131
16.1	Comparison of an analytical low thrust model with the propagated solution.	138
16.2	Reentry Due to Drag	139
16.3	Topex Ephemeris Cycle 10	140
16.4	Difference Between the Topex Ephemeris and a Keplerian Orbit	141
16.5	Difference Between the Topex Ephemeris and a Propagated Spherical Earth Model	141
16.6	Topex Orbits in 3D	142
16.7	Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal Harmonics	142
16.8	Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal and 4 Tesseral Harmonics	143



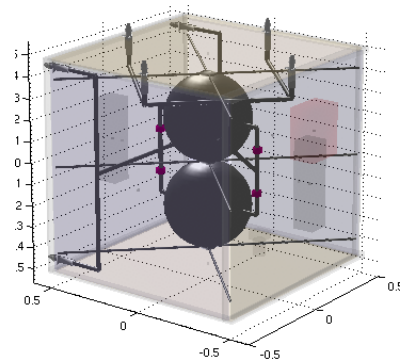
16.9 Effect of Integration Tolerances on Orbit Propagation . . . . .	144
17.1 Launch and early orbit operations analysis . . . . .	147
17.2 Orbit propagation GUI and aero model inputs . . . . .	149
17.3 Comparison of analytical drag with propagated . . . . .	149
17.4 Orbital elements . . . . .	150
17.5 Glideslope demo results . . . . .	151
18.1 FEngine GUI . . . . .	158
18.2 FAero GUI . . . . .	158
18.3 FSolar GUI . . . . .	158
18.4 FPlantNoise GUI . . . . .	159
18.5 FAtmDensity GUI . . . . .	159
21.1 CannyEdgeDetector demo . . . . .	177
21.2 GaussianImageFilter demo . . . . .	178
22.1 Shannon demo . . . . .	181
22.2 Atmospheric Gas Loss . . . . .	182
22.3 Sky temperature . . . . .	182
22.4 Results of LinkAnalysisTool . . . . .	183
22.5 Results of LADARTransmitPower . . . . .	185
22.6 Results of RadarEquation . . . . .	186
22.7 Results of RadarCrosssection . . . . .	187
23.1 Band gap . . . . .	189
23.2 Solar cell I-V curves . . . . .	190
23.3 Solar cell efficiency . . . . .	190
24.1 Solid motor firing . . . . .	193
24.2 Blowdown curve . . . . .	194
24.3 Optimal electric vehicle . . . . .	194
24.4 Two-State-to-Orbit Simulation . . . . .	195
25.1 Black body wavelength as a function of temperatures . . . . .	198
25.2 AbsDa demo . . . . .	198
25.3 Plancks Law . . . . .	199
25.4 Thermal network . . . . .	200
25.5 Thermal imaging demo . . . . .	202
25.6 Thermal imaging demo temperature plot . . . . .	202
27.1 Fixed gain Kalman filter . . . . .	208
27.2 UD filter . . . . .	209
27.3 Continuous discrete Kalman filter . . . . .	210
27.4 Clock error covariance for an orbit Kalman Filter . . . . .	211
28.1 Stellar attitude determination process. . . . .	214
28.2 LoadCatalog default . . . . .	215
28.3 FOV determination . . . . .	216
28.4 Centroiding demo . . . . .	218
28.5 StarCentroidCOM demo . . . . .	219
28.6 K-Vector Example in MATLAB . . . . .	219

---

28.7 Use of the EKF for stellar attitude determination. . . . .	221
29.1 Pressure Kalman Filter . . . . .	225
A-1 Loading GOES data . . . . .	231
A-2 GOES Magnetic Field Data . . . . .	231
A-3 GOES Electrons . . . . .	232
A-4 GOES Weather . . . . .	232
B-1 Constant database . . . . .	235
B-2 Searching for mass . . . . .	236
B-3 MergeConstantDB function . . . . .	237
C-1 PlugInDemo on starting . . . . .	239
C-2 PlugInDemo with the earth as center . . . . .	240
C-3 Selecting planets in the elements plugin . . . . .	241
C-4 PlugInDemo at the end of the simulation . . . . .	241
C-5 PlugInDemo 3D Spacecraft Display . . . . .	242

# PART I

## TOOLBOX BASICS





---

## INTRODUCTION

---

This chapter shows you how to install the Spacecraft Control Toolbox and how it is organized.

### 1.1 Organization

---

The Spacecraft Control Toolbox is composed of MATLAB m-files and mat-files, organized into a set of modules by subject. It is essentially a library of functions for analyzing spacecraft and missions. There is a substantial set of software which the Spacecraft Control Toolbox shares with the Aircraft Control Toolbox, and this software is in a module called *Common*. The core spacecraft files are in *SC* and *SCPro* along with special plotting tools in *Plotting*. The new *CubeSat* module demonstrates mission planning and simulation for nanosatellites. These five modules are referred to together as the *Core* toolbox. Functions for estimation, orbit analysis, subsystem analysis, and detailed attitude control system examples are grouped in separate modules. All of the modules are described in the following table, including the add-on modules which are purchased separately.

**Table 1.1:** Spacecraft Control Toolbox Modules

Module		Function
AerospaceUtils		CAD tools, coordinate transformations, atmosphere models
Common		Control design , math, quaternions, general estimation and Kalman filters, time conversions, graphics, utilities
CubeSat		CubeSat and nanosatellite modeling
Plotting		PlottingTool and AnimationGUI for visualization of complex simulations
SC		Attitude dynamics, pointing budgets, basic orbit dynamics, environment, sample CAD models, ephemeris, sensor and actuator modeling.
SCPro		Additional high-fidelity models for environment, sensors, actuators.
Imaging		Image processing functions
Link		Basic RF and optical link analysis.
Missions		In-depth attitude control system design and mission examples, including the hypothetical geosynchronous satellite ComStar
Orbit		Orbit mechanics, maneuver planning, fuel budgets, and high-fidelity simulation.
Power		Basic power modeling
Propulsion		Electric and chemical propulsion. Launch vehicle analysis.
SpacecraftEstimation		Attitude and orbit estimation, stellar attitude determination.
Thermal		Basic thermal modeling
Formation Flying	Add-On	Formation flying control
Fusion Propulsion	Add-On	Fusion propulsion analysis

Launch Vehicle	Add-On	Launch vehicle design
LunarCube	Add-On	Lunar extension for the CubeSat toolbox
SAAD	Add-On	Spin axis attitude determination
Solar Sail	Add-On	Solar sail control and mission analysis

The Spacecraft Control Toolbox core, *CubeSat*, *SpacecraftEstimation*, *Imaging*, *Orbit*, *Link*, *Propulsion*, and *Thermal* modules are described in this user’s guide. Each of these modules has its own part in the guide and is included in the Professional Edition of the toolbox. The Academic Edition consists of only the *AerospaceUtils*, *Common*, *CubeSat*, *Plotting*, and *SC* directories grouped together at the top. The CubeSat Toolbox consists of the *CubeSat* module and those files from the core toolbox needed to run them.

The *Formation Flying*, *Solar Sail*, and *SAAD* add-on modules have their own user’s guides. These modules can be purchased separately.

## 1.2 Requirements

---

MATLAB 2014a at a minimum is required to run all of the functions. Most of the functions will run on previous versions but we are no longer supporting them.

## 1.3 Installation

---

The preferred method of delivering the toolbox is a download from the Princeton Satellite Systems website. Put the folder extracted from the archive anywhere on your computer. There is no “installer” application to do the copying for you. We will refer to the folder containing your modules as PSSToolboxes. If you later purchase an add-on module, you would simply add it to this folder.

All you need to do now is to set the MATLAB path to include the folders in PSSToolboxes. We recommend using the supplied function `PSSSetPaths.m` instead of MATLAB’s path utility. From the MATLAB prompt, `cd` to your PSSToolboxes folder and then run `PSSSetPaths`. For example:

```
>> cd /Users/me/PSSToolboxes
>> PSSSetPaths
```

This will set all of the paths for the duration of the session, with the option of saving the new path for future sessions.

## 1.4 Getting Started

---

The first two functions that you should try are `DemoPSS` and `FileHelp`. Each toolbox or module has a `Demos` folder and a function `DemoPSS`. Do not move or remove this function from any of your modules! `DemoPSS.m` looks for other `DemoPSS` functions to determine where the demos are in the folders so it can display them in the `DemoPSS` GUI.

The `FileHelp` function provides a graphical interface to the MATLAB function headers. You can peruse the functions by folder to get a quick sense of your new product’s capabilities and search the function names and headers for keywords. `FileHelp` and `DemoPSS` provide the best way to get an overview of the Spacecraft Control Toolbox. Both are described in more detail in the next chapter.

Another useful utility function you should try is `Finder`. This GUI allows you to search your entire path, or selected folders, for instances of any keyword. While `FileHelp` allows you to search headers for keywords, `Finder` allows you to search for instances of a function or variable in the code itself. This GUI is described in Section 3.5.1. For example, you can use it to find all the demos that show the use of the function `SunV1`, which computes the sun vector from the date using an almanac.

\*\* New in SCT ver 9.0 - Demos and Functions can now be browsed by MATLAB's built-in help system. These are described in Section 2.1. This allows for the same searching and browsing capabilities as `DemoPSS` and `FileHelp`.





# GETTING HELP

---

This chapter shows you how to use the help systems built into PSS Toolboxes. There are several sources of help. Our toolboxes are now integrated into MATLAB's built-in help browser. Then, there is the MATLAB command line help which prints help comments for individual files and lists the contents of folders. Then, there are special help utilities built into the PSS toolboxes: one is the file help function, the second is the demo functions and the third is the graphical user interface help system. Additionally, you can submit technical support questions directly to our engineers via email.

## 2.1 MATLAB's Built-in Help System

---

### 2.1.1 Basic Information and Function Help

Our toolbox information can now be found in the MATLAB help system. To access this capability, simply open the MATLAB help system. As long as the toolbox is in the MATLAB path, it will appear in the contents pane. In more recent versions of MATLAB, you need to navigate to Supplemental Software from the main window, as shown in Figure 2.1 on page 6. The index page of the SCT documentation is shown in Figure 2.2 on page 7.

The help window from R2011b and earlier is depicted in Figure 2.3 on page 7.

This contains a lot information on the toolbox. It also allows you to search for functions as you would if you were searching for functions in the MATLAB root.

### 2.1.2 Published Demos

Another feature that has been added to the MATLAB help structure is the access to all of the toolbox demos. Every single demo is now listed, according to module and the folder. These can be found under the *Other Demos* or *Examples* portion of the Contents Pane. Each demo has its own webpage that goes through it step by step showing exactly what the script is doing and which functions it is calling. From each individual demo webpage you can also run the script to view the output, or open it in the editor. Note that you might want to save any changes to the demo under a new file name so that you can always have the original. Below is an example of demo page displayed in MATLAB help that shows where to find the toolbox demos as well as the the hierarchal structure used for browsing the demos.

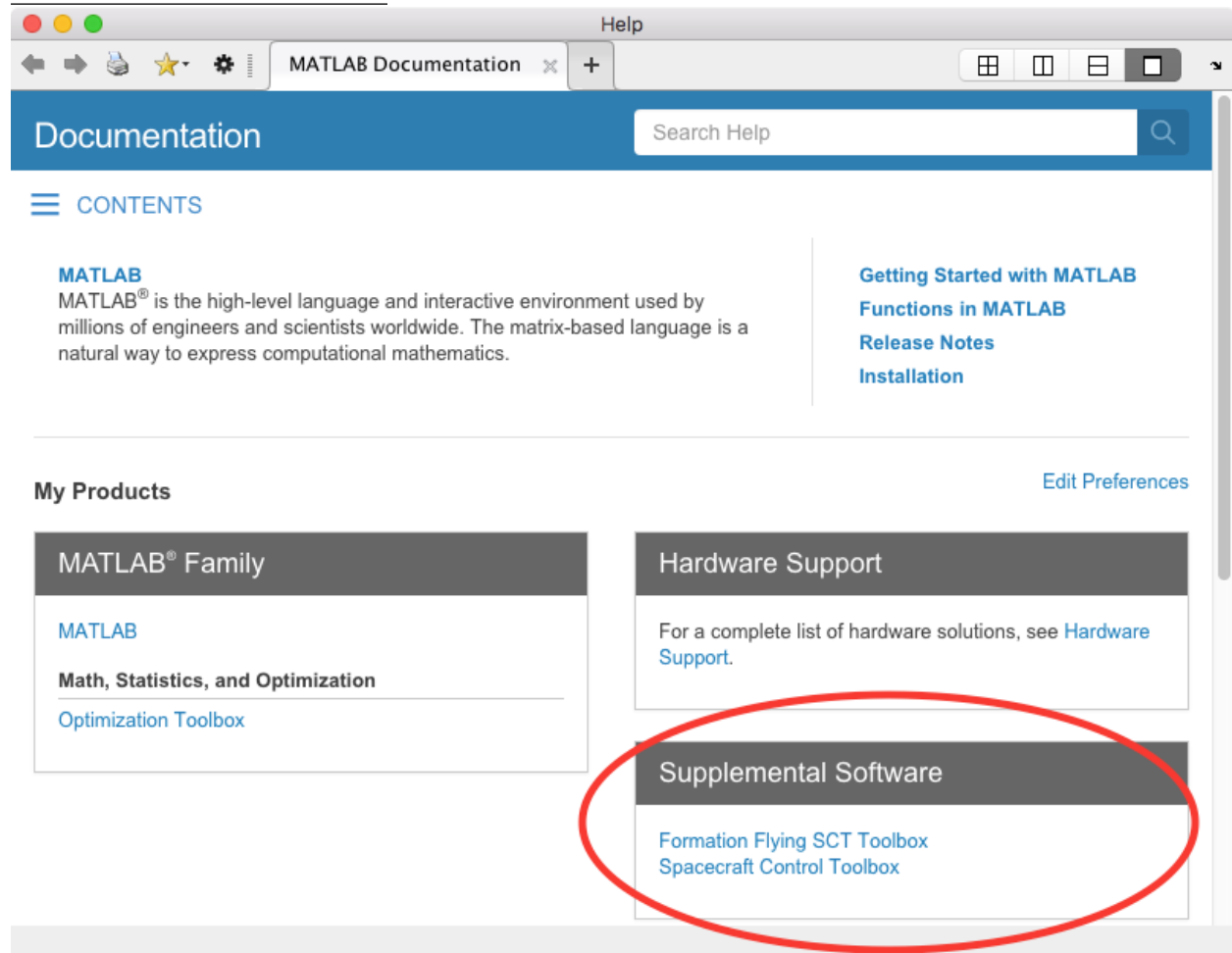
**Figure 2.1:** MATLAB Help - Supplemental Software

Figure 2.2: Toolbox Documentation Main Page, R2016b

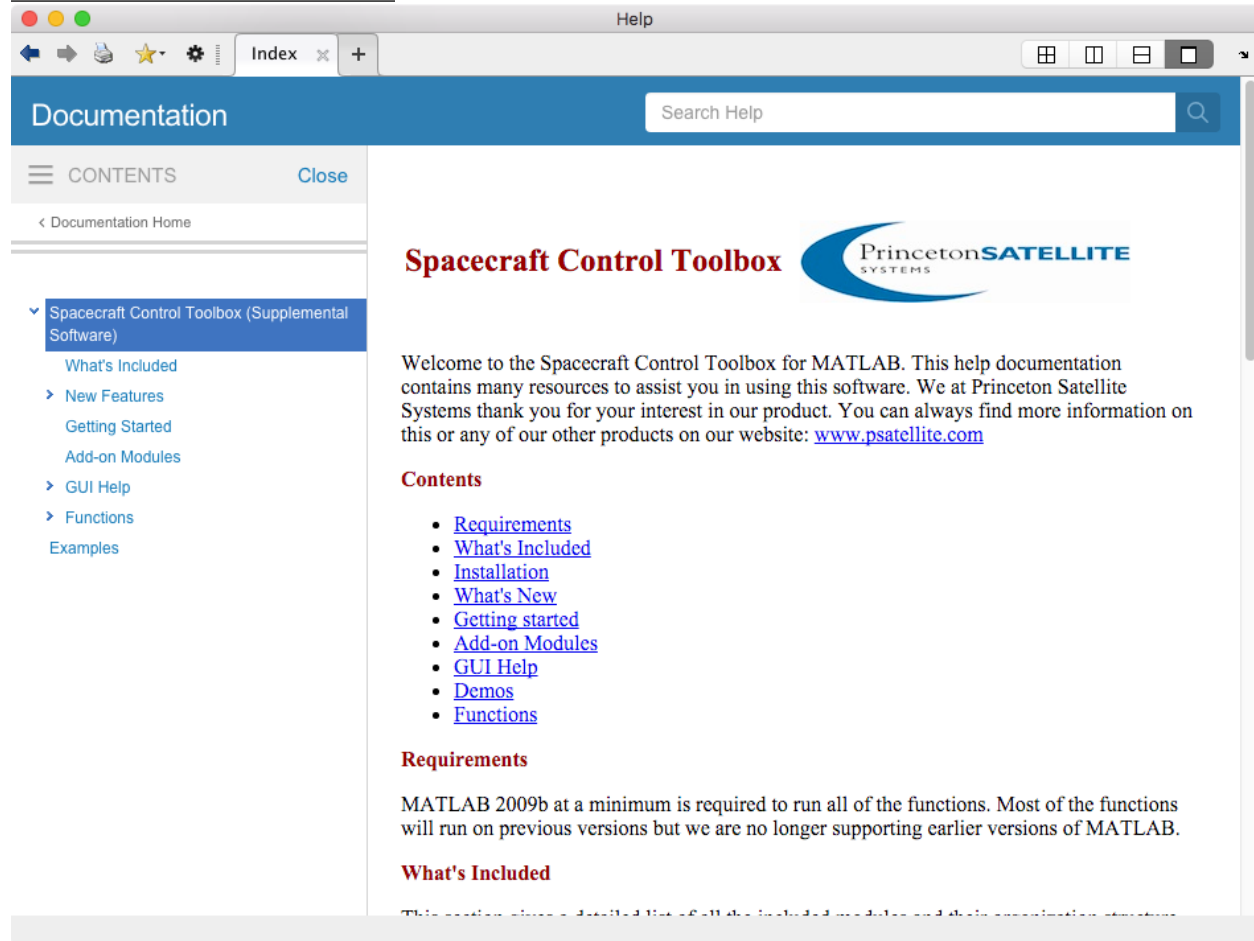


Figure 2.3: Toolbox Documentation, R2011b

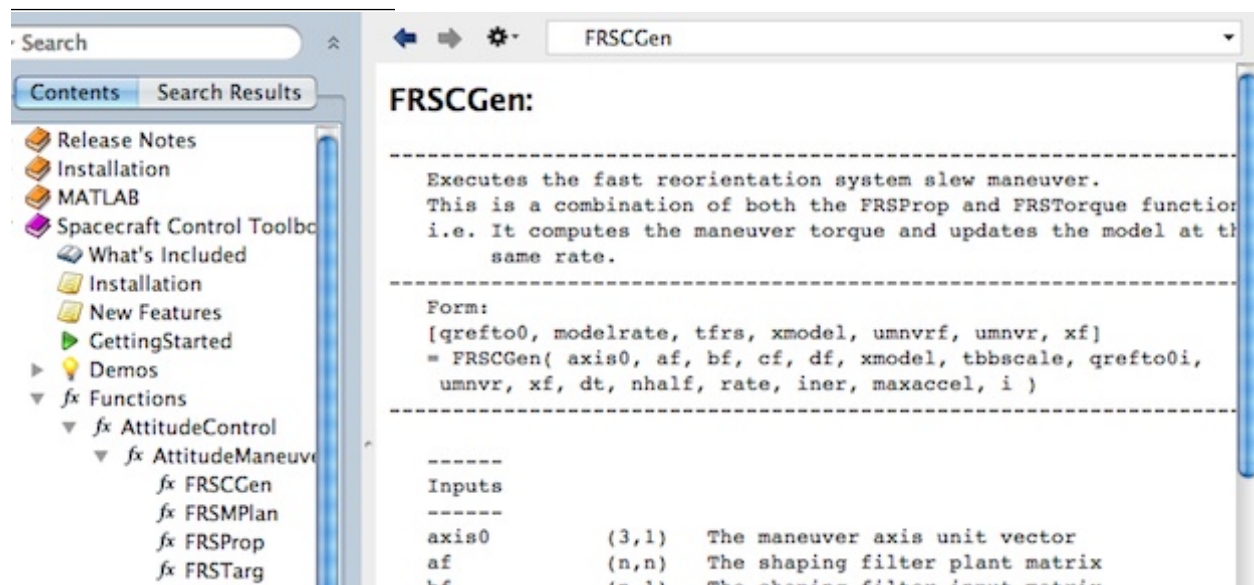
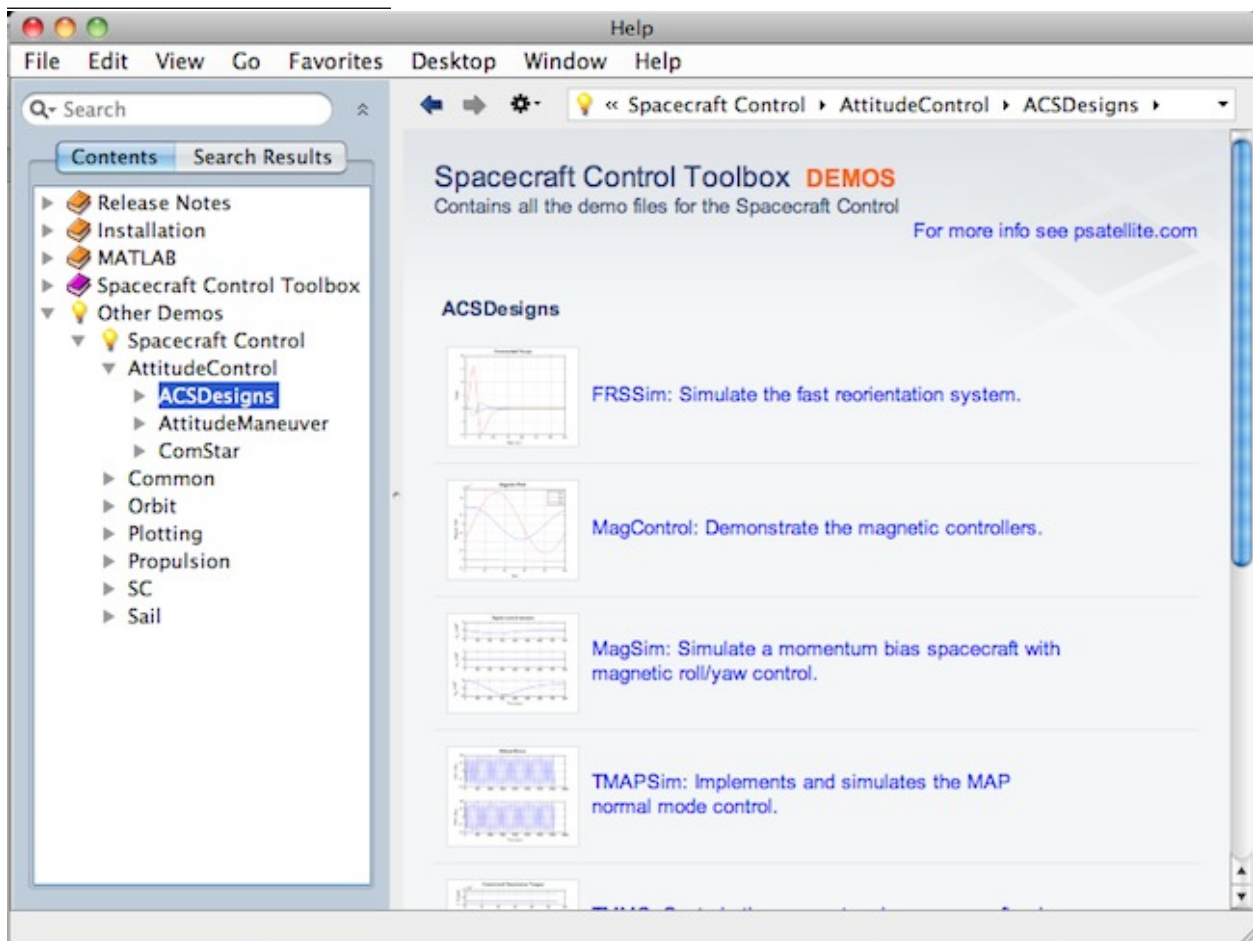


Figure 2.4: Toolbox Demos



## 2.2 Command Line Help

You can get help for any function by typing

```
>> help functionName
```

For example, if you type

```
>> help C2DZOH
```

you will see the following displayed in your MATLAB command window:

```
-----
Create a discrete time system from a continuous system
assuming a zero-order-hold at the input.

Given
.
x = ax + bu

Find f and g where

x(k+1) = fx(k) + gu(k)

-----

Form:
[f, g] = C2DZOH( a, b, T )

-----

-----
Inputs
-----

a          (n,n)  Continuous plant matrix
b          (n,m)  Input matrix
T          (1,1)  Time step

-----

Outputs
-----

f          (n,n)  Discrete plant matrix
g          (n,m)  Discrete input matrix

-----
```

All PSS functions have the standard header format shown above. Keep in mind that you can find out which folder a function resides in using the MATLAB command `which`, i.e.

```
>> which C2DZOH
/Software/Toolboxes/SCT/Common/Control/C2DZOH.m
```

When you want more information about a folder of interest, you can get a list of the contents in any directory by using the `help` command with a folder name. The returned list of files is organized alphabetically. For example,

```
>> help Atmosphere
Common/Atmosphere

S
  SimpAtm - Simplified atmosphere model.
  StdAtm  - Computes atmospheric density based on the standard atmosphere model.
```

If there is a folder with the same name in a Demos directory, the demos will be listed separately. For example,

```
>> help Plugins

Common/Plugins

T
Telemetry                - Generates a set of telemetry pages.
TelemetryOffline         - This plots telemetry files previously saved
                          by Telemetry.
TelemetryPlot            - Plot real time in a single window.
TimePlugIn               - Create a time GUI plug in.

Common/Demos/Plugins

T
TelemetryDemo            - Demonstrate the Telemetry function.
```

In the case of a demo, the command line help will provide a link to the published HTML for that demo, if any exists. Any functions referenced on a See also line will have dynamic links, which will show the help for that function.

```
>> help Attitude3D
Simple sim using a CAD model of the spacecraft to view the attitude.
Demonstrates control design using PIDMIMO, the Disturbances function,
rigid body attitude dynamics with FRB, orbit dynamics with FOrbCart, and
integration using RK4. The CAD model is viewed using DrawSCPlugIn.

Use the flags to turn on or off the disturbances and 3D viewing. If 3D
viewing is off the demo concludes with a quaternion animation.
-----
See also DrawSCPlanPlugIn, PIDMIMO, AU2Q, AnimQ, QLVLH, QMult, QPose,
Constant, NPlot, Plot2D, Plot3D, TimeGUI, RK4, JD2000, E12RV,
Disturbances, SunV1, DrawSCPlugIn, Accel
-----
Published output in the Help browser
showdemo Attitude3D
```

To see the entire contents of a file at the command line, use `type`.

```
>> type Attitude3D

%% Simple sim using a CAD model of the spacecraft to view the attitude.
% Demonstrates control design using PIDMIMO, the Disturbances function,
...
```

Command line help also works with higher level directories, for instance if you ask for help on the Common directory, you will get a list of all the subdirectories.

```
>> help Common

PSS Toolbox Folder Common
Version 2015.1      05-Mar-2015

Directories:
Atmosphere
Classes
CommonData
Control
```

```

ControlGUI
Database
DemoFuns
Demos
Demos/Control
Demos/ControlGUI
Demos/Database
Demos/General
Demos/GeneralEstimation
Demos/Graphics
Demos/Help
Demos/MassProperties
Demos/Plugins
Demos/UKF
Estimation
FileUtils
General
Graphics
Help
Interface
MassProperties
Materials
Plugins
Quaternion
Time
Transform

```

The function `ver` lists the current version of all your installed toolboxes. Each SCT module that you have installed will be listed separately. For instance,

```

-----
MATLAB Version: 8.1.0.604 (R2013a)
...
-----

```

```

MATLAB                               Version 8.1           (R2013a)
PSS Toolbox Folder AeroUtils         Version 2014.1
PSS Toolbox Folder AttitudeControl   Version 2014.1
PSS Toolbox Folder Common             Version 2014.1
PSS Toolbox Folder CubeSat           Version 2014.1
PSS Toolbox Folder Electrical         Version 2014.1
PSS Toolbox Folder Imaging            Version 2014.1
PSS Toolbox Folder Link               Version 2014.1
PSS Toolbox Folder Math               Version 2014.1
PSS Toolbox Folder Orbit              Version 2014.1
PSS Toolbox Folder Plotting           Version 2014.1
PSS Toolbox Folder Propulsion         Version 2014.1
PSS Toolbox Folder SC                 Version 2014.1
PSS Toolbox Folder SCPro              Version 2014.1
PSS Toolbox Folder SpacecraftEstimation Version 2014.1
PSS Toolbox Folder Thermal            Version 2014.1

```

## 2.3 FileHelp

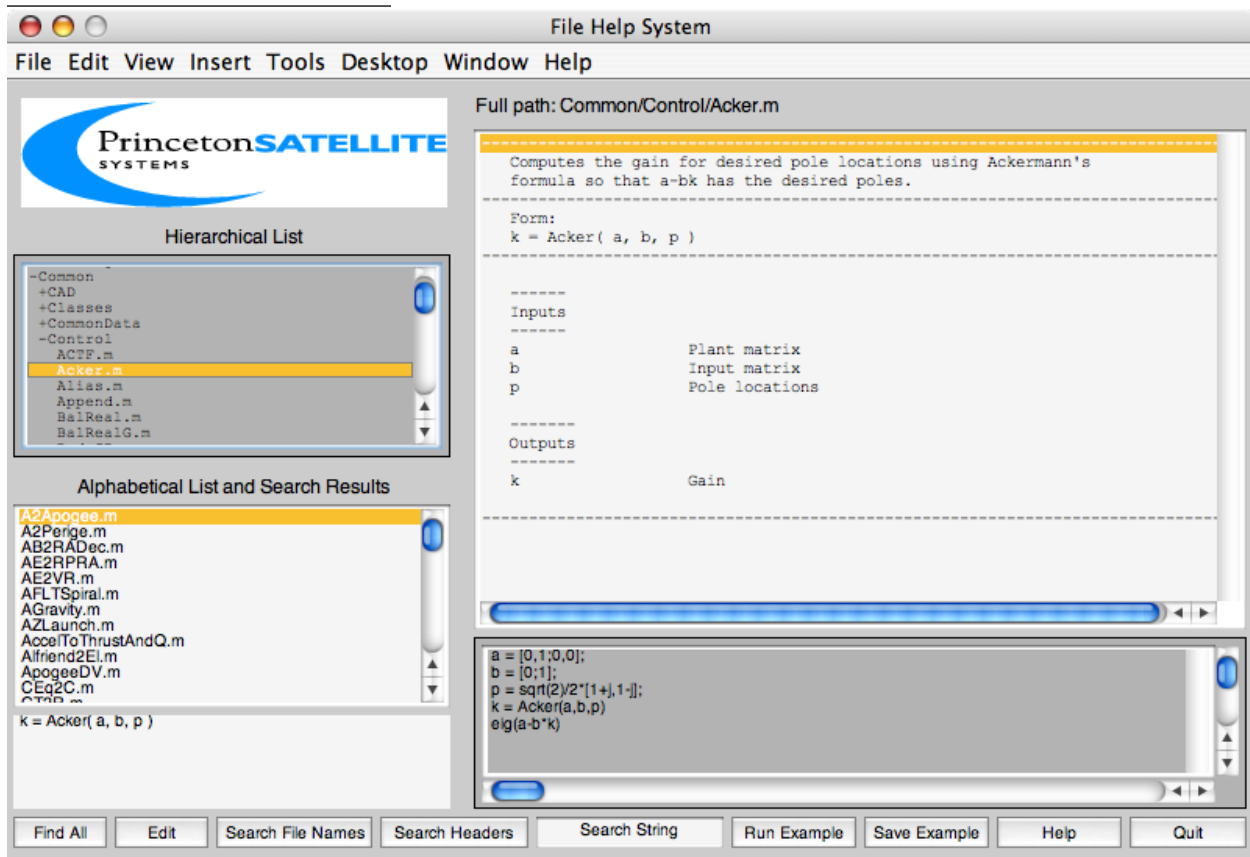
### 2.3.1 Introduction

When you type

```
FileHelp
```

the FileHelp GUI appears, Figure 2.5 on page 12.

**Figure 2.5:** The file help GUI



There are five main panes in the window. On the left hand side is a display of all functions in the toolbox arranged in the same hierarchy as the PSSToolboxes folder. Scripts, including most of the demos, are not included. Below the hierarchical list is a list in alphabetical order by module. On the right-hand-side is the header display pane. Immediately below the header display is the editable example pane. To its left is a template for the function. You can cut and paste the template into your own functions.

The buttons along the bottom provide additional controls along with the search feature. Select the “Search String” text and replace it with your own text, for example “sun”. Then click either the Search File Names button or Search Headers.



### 2.3.2 The List Pane

If you click a file in the alphabetical or hierarchical lists, the header will appear in the header pane. This is the same header that is in the file. The headers are extracted from a .mat file so changes you make will not be reflected in the file. In the hierarchical list, any name with a + or - sign is a folder. Click on the folders until you reach the file you would like. When you click a file, the header and template will appear.

### 2.3.3 Edit Button

This opens the MATLAB edit window for the function selected in the list.

### 2.3.4 The Example Pane

This pane gives an example for the function displayed. Not all functions have examples. The edit display has scroll bars. You can edit the example, create new examples and save them using the buttons below the display. To run an example, push the Run Example button. You can include comments in the example by using the percent symbol.

### 2.3.5 Run Example Button

Run the example in the display. Some of the examples are just the name of the function. These are functions with built-in demos. Results will appear either in separate figure windows or in the MATLAB Command Window.

### 2.3.6 Save Example Button

Save the example in the edit window. Pushing this button only saves it in the temporary memory used by the GUI. You can save the example permanently when you Quit.

### 2.3.7 Help Button

Opens the on-line help system.

### 2.3.8 Quit

Quit the GUI. If you have edited an example, it will ask you whether you want to save the example before you quit.

## 2.4 Searching in File Help

---

### 2.4.1 Search File Names Button

Type in a function name in the edit box and push the button called Search File Names.

### 2.4.2 Find All Button

Find All returns to the original list of the functions. This is used after one of the search options has been used.

### 2.4.3 Search Headers Button

Search headers for a string. This function looks for exact, but not case sensitive, matches. The file display displays all matches. A progress bar gives you an indication of time remaining in the search.

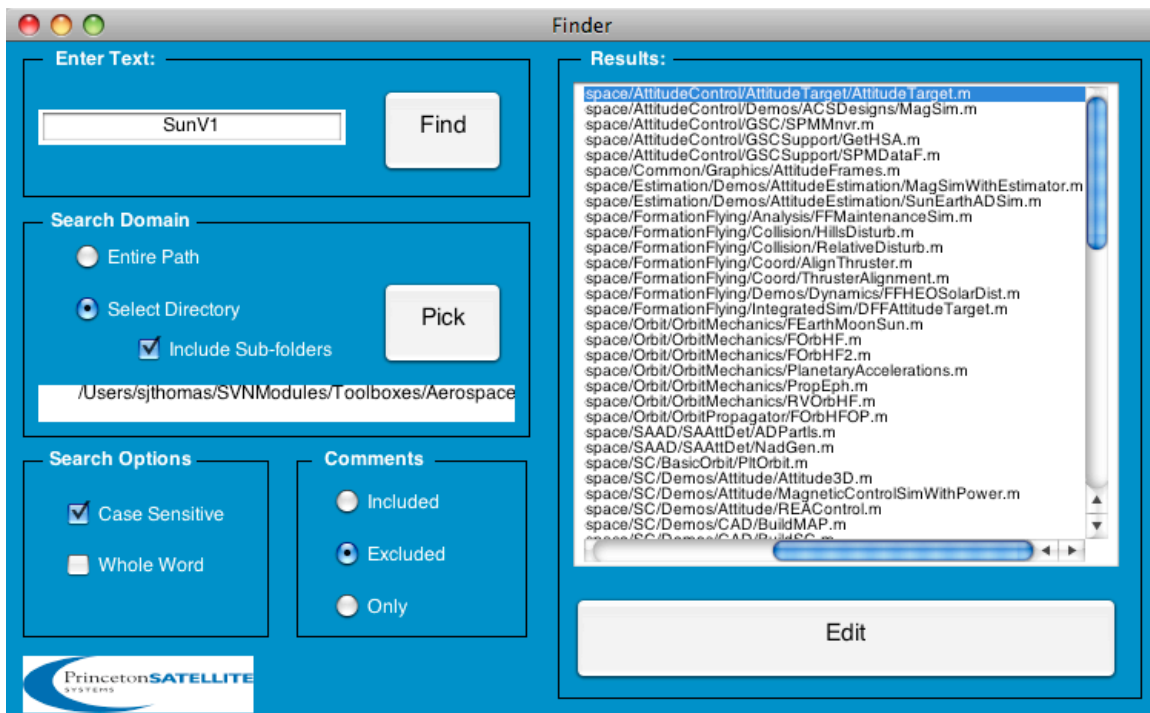
### 2.4.4 Search String Edit Box

This is the search string. Spaces will be matched so if you type attitude control it will not match attitude control (with two spaces.)

## 2.5 Finder

The `Finder` GUI, shown below, is another handy function for searching for information in the toolbox. (It is not included in the CubeSat Toolbox.) You can search for instances of keywords in the entire body of functions and demos, not just the help comments. You can use this function with any toolboxes, not just your PSS toolboxes, since this actively searches the files every time instead of using a parsed version of the headers the way `FileHelp` does. Consequently, it is a little slower to use, but you can use it with your own function libraries too.

The `Finder` function has options for searching the entire path or a selected directory. The `Pick` button brings up a file selection dialog where you can navigate to your desired directory. The search can be case sensitive and you can select whole word matching. You can search on just file help comments, or include or exclude them. For example, you can find all functions and demos that actually use the function `PIDMIMO` by searching with comments excluded. Once your search results are displayed in the Results window, you can open any file by clicking the `Edit` button.

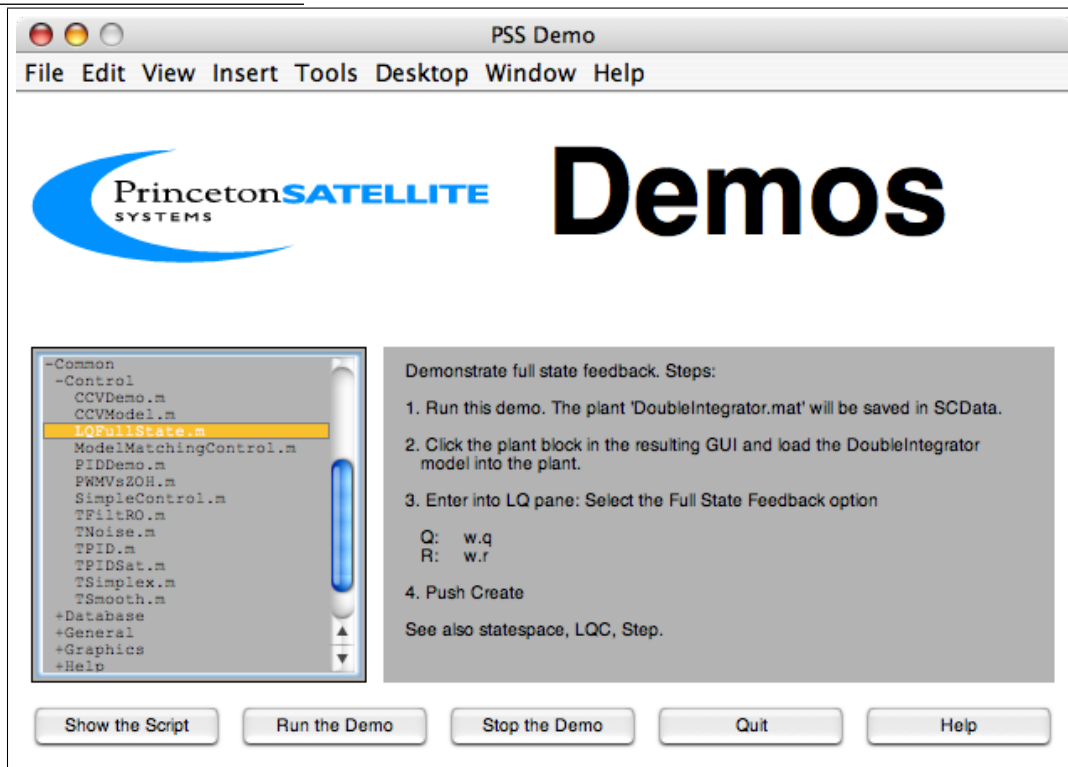


## 2.6 DemoPSS

If you type DemoPSS you will see the GUI in Figure 2.6 on page 15. This predates MATLAB's built-in help feature and provides an easy way to run the scripts provided in the toolbox. The list on the left-hand-side is hierarchical and the top level follows the organization of your toolbox modules. Most folders in your modules have matching folders in Demos with scripts that demonstrate the functions. The GUI checks to see which directories are in the same directory as DemoPSS and lists all directories and files. This allows you to add your own directories and demo files.

Click on the first name to open the directory. The + sign changes to - and the list changes. Figure 2.6 shows the Common/Control folder in the core toolbox. The hierarchical menu shows the highest level folders.

**Figure 2.6:** The demo GUI



Your own demos will appear if they are put in any of the Demos folders. If you would like to look at, or edit, the script, push Show the Script.

You can also access the published version of the demos using MATLAB's help system. On recent versions this is access by selecting Supplemental Software from the main Help window, and then selecting Examples.

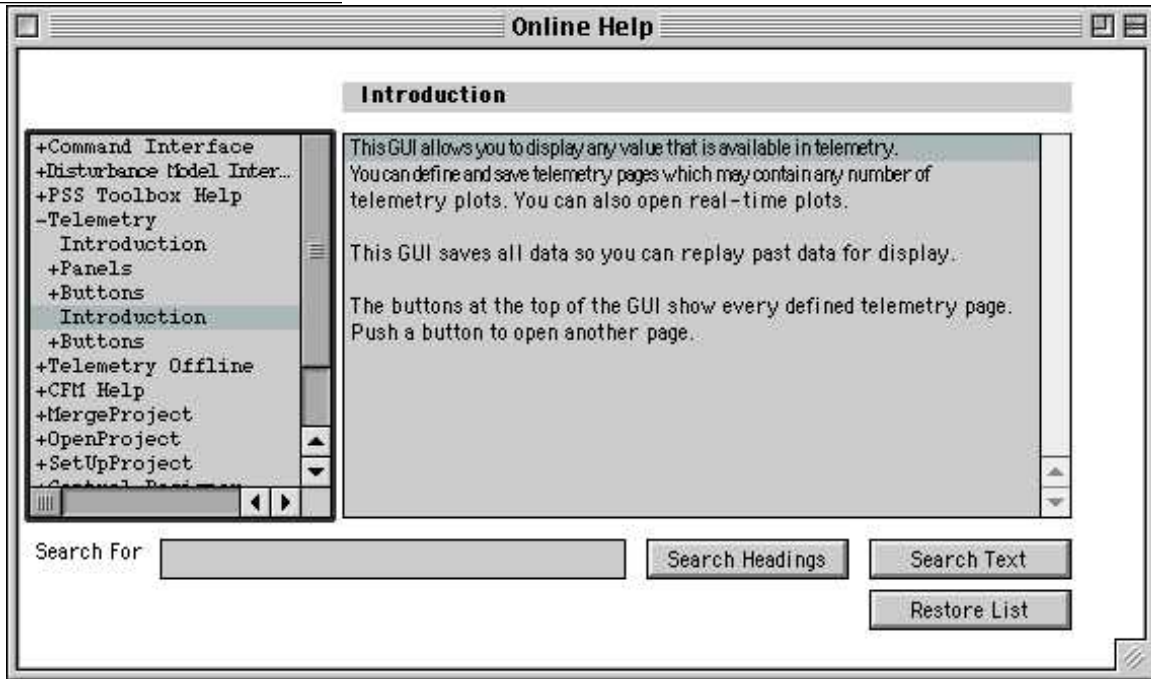
## 2.7 Graphical User Interface Help

Each graphical user interface (GUI) has a help button. If you hit the help button a new GUI will appear. You can access on-line help about any of the GUIs through this display. It is separate from the file help GUI described above. The same help is also available in HTML through the MATLAB help window.

The HelpSystem display is hierarchical. Any list item with a + or - in front is a help heading with multiple subtopics. + means the heading item is closed, - means it is open. Clicking on a heading name toggles it open or closed. Figure 2.7

on the following page shows the display with the Telemetry help expanded. If you click on a topic in the list you will get a text display in the right-hand pane. You can either search the headings or the text by entering a text string

Figure 2.7: On-line Help

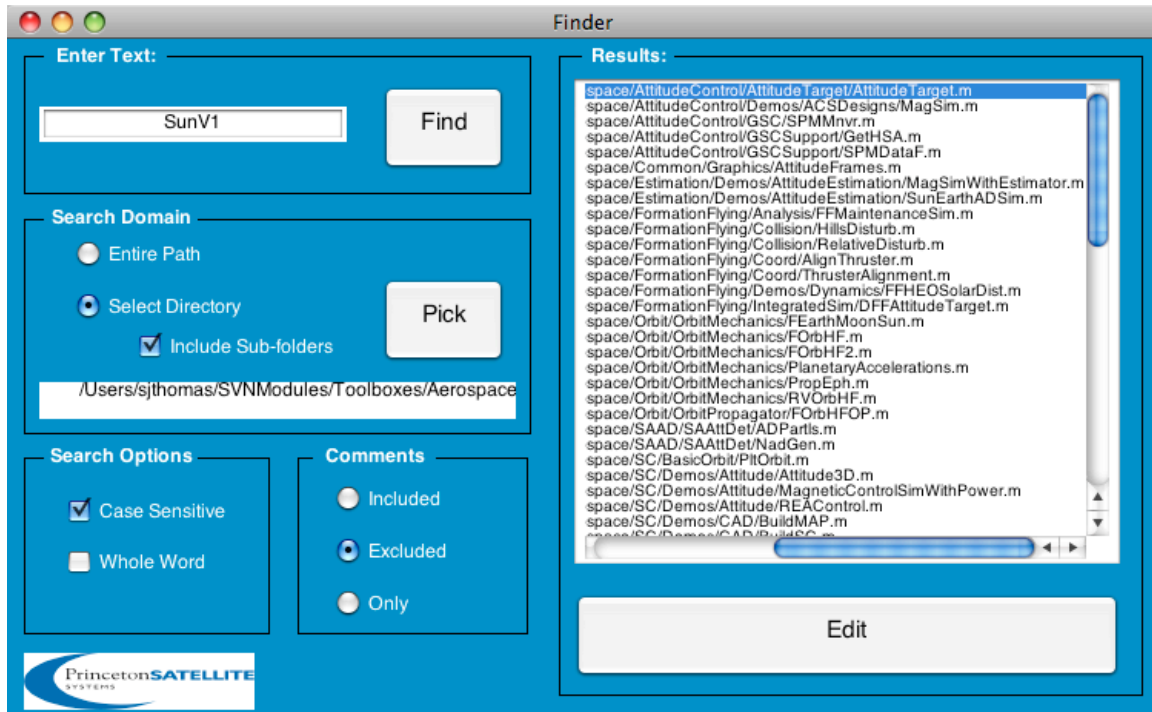


into the Search For edit box and hitting the appropriate button. Restore List restores the list window to its previous configuration.

## 2.8 Finder

The Finder GUI, shown below, is another handy function for searching for information in the toolbox. (It is not included in the CubeSat Toolbox.) You can search for instances of keywords in the entire body of functions and demos, not just the help comments. You can use this function with any toolboxes, not just your PSS toolboxes, since this actively searches the files every time instead of using a parsed version of the headers the way FileHelp does. Consequently, it is a little slower to use, but you can use it with your own function libraries too.

The Finder function has options for searching the entire path or a selected directory. The subfolders of a higher-level directory can be included or not. The Pick button brings up a file selection dialog where you can navigate to your desired directory. The search can be case sensitive and you can select whole word matching. You can search on just file help comments, or include or exclude them. For example, you can find all functions and demos that actually use the function PIDMIMO by searching with comments excluded. Once your search results are displayed in the Results window, you can open any file by clicking the Edit button.



## 2.9 Technical Support

Contact [support@psatellite.com](mailto:support@psatellite.com) for free email technical support. We are happy to add functions and demos for our customers when asked.



# BASIC FUNCTIONS

---

This chapter shows you how to use a sampling of the most basic Spacecraft Control Toolbox functions.

## 3.1 Introduction

---

The Spacecraft Control Toolbox is composed of several thousand MATLAB files. The functions cover attitude control and dynamics, computer aided design, orbit dynamics and kinematics, ephemeris, actuator and sensor modeling, and thermal and mathematics operations. Most of the functions can be used individually although some are rarely called except by other toolbox functions.

This chapter will review some basic features built into the SCT functions and highlight some examples from the folders that you will use most frequently. The last section introduces the GUIs included in the toolbox. You can always build additional GUIs using the plug-ins described in [Appendix C on page 125](#).

## 3.2 Header Format

---

Our functions follow a fixed header format. To view the header of a function in the command line, type, “help functionname”. For example,

```
>> help Dot
-----
Dot product with support for arrays.
The number of columns of w and y can be:
- Both > 1 and equal
- One can have one column and the other any number of columns

Since version 1.
-----
Form:
d = Dot ( w, y )
-----

-----
Inputs
-----
w                (:,:) Vector
```

```

y          (:,:)  Vector
-----
Outputs
-----
d          (1,:)  Dot product of w and y
-----

```

You can see that the header starts with a description of the function. This gives the first version of the toolbox that included the function in the “Since” line. Then, there is the function syntax under the heading “Form”. If there are multiple syntaxes accepted by the function, they will all be listed here. Finally, there is a list of the inputs and outputs with descriptions. If there are specific units required they will be given here. MATLAB variables do not have explicit types, so we have developed a specific format for showing the size or type of inputs and outputs, as shown in Table 3.1 on the following page.

**Table 3.1:** Format Markup

(n,m)	Matrix with row and column dimensions. If these are not a fixed number, : will be used.
(.)	Data structure. The fields will be described on the following lines.
(:)	Data structure array.
{n,m}	A cell array. The dimensions are given the same as for matrices.
(1,:)	A row vector or string.
(1,1)	A scalar or boolean.
(*)	A function handle.

Optional inputs are generally at the end of the input list. An optional input may have a default value available, which will be described in the header. An optional input may be marked with an asterisk.

Function headers should list any “side effects” of the function, such as generating plots or saving files. The header will specify if the function has a built-in demo, which is described further in the next section. Additional content may include references

Here is another example with a demo, inputs with default values, and a reference.

```

>> help el2rv
Converts orbital elements to r and v for an elliptic orbit.

Type El2RV for a demo.
-----
Form:
[r, v] = El2RV( el, tol, mu )
-----

-----
Inputs
-----
el   (6,:) Elements vector [a;i;W;w;e;M]           (angles in radians)
tol  (1,1)* Tolerance for Kepler's equation solver. (default = 1e-14)
mu   (1,1)* Gravitational constant.               (default = 3.9860044e5)

-----
Outputs
-----
r    (3,:) position vector
v    (3,:) velocity vector

```



-----  
 References: Battin, R.H., An Introduction to the Mathematics and  
 Methods of Astrodynamics, p 128.  
 -----

## 3.3 Function Features

### 3.3.1 Introduction

Functions have several features that are helpful to understand. Features that are available in the functions are listed in Table 3.2 on the next page.

**Table 3.2:** Features in Spacecraft Control Toolbox functions

Features
Built-in demos
Default parameters
Built-in plotting
Error checking
Variable inputs

These are illustrated in the examples given below.

### 3.3.2 Built-in demos

Many functions have built in demos. A function with a built-in demo requires no inputs and produces a plot or other output for a range of input parameters to give you a feel for the function.

An example of a function with a built-in demo is `AtmDens2` which generates the plot in Figure 3.1. This plot shows the atmospheric density as a function of altitude over the full range of the model. The inputs for the built-in demo are generally specified near the top of the function so it is easy to check for one by looking at the code. Plots are produced at the bottom of a function.

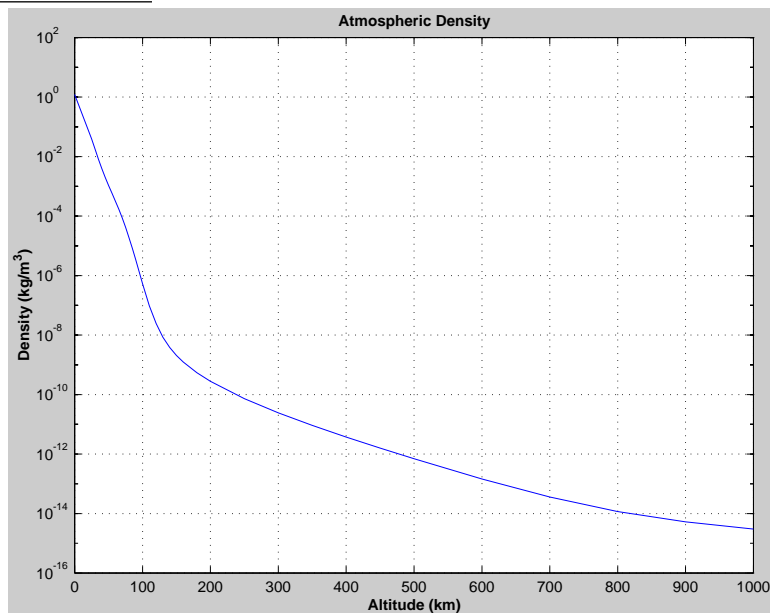
### 3.3.3 Default parameters

Most functions have default parameters. There are two ways to get default parameters. If you pass an empty matrix, i.e.

```
[]
```

as a parameter the function will use a default parameter if defaults are available. This is only necessary if you wish to use a default for one parameter and input the value for the next input. For example, `EarthRot` takes a date in Julian centuries as the first parameter and a flag for equation of the equinoxes as the second parameter. If you look in the function, you will see that the default is to use the current date.

```
>> g = EarthRot( [], 1 )
g =
   -0.9815   -0.1914         0
    0.1914   -0.9815         0
         0         0    1.0000
```

**Figure 3.1:** Atmospheric density from `AtmDens2`

The second way to get defaults is simply to leave off arguments at the end of the input list. For `EarthRot` the second parameter is also optional.

```
>> g = EarthRot( )

g =
   -0.9815   -0.1916         0
    0.1916   -0.9815         0
         0         0    1.0000
```

You should never hesitate to look in functions to see what defaults are available and what the values are. Defaults are always treated at the top of the function just under the header. Remember that the unix command `type` works in MATLAB to display a function's contents, for instance

```
>> type EarthRot

function [g, gMST, gAST] = EarthRot( T, eOfECalc )

%% Computes the Earth greenwich matrix that transforms from ECI to EF.
%% Any input of eOfECalc will cause it to include the equation of the
%% equinoxes.
%%
%% Type EarthRot for a demo.
%%
%% -----
%% Form:
%% [g, gMST, gAST] = EarthRot( T, eOfECalc )
%% -----
%%
%% -----
%% Inputs
%% -----
%% T           (1,1) Julian centuries of 86400s dynamical time from j2000.0
%% eOfECalc    (1,1) Calculate the equation of the equinoxes
%%
%% -----
```

```

% Outputs
% -----
% g          (3,3) Greenwich matrix
% gMST       (1,1) Greenwich mean sidereal time (deg)
% gAST       (1,1) Greenwich apparent sidereal time (deg)
%
% -----
% See also: GMSTime, EOfE, JD2T
% -----
% References: Seidelmann, P. K., The Explanatory Supplement to the
%             Astronomical Almanac, University Science Books, 1992, p. 20.
% -----
%
% Copyright (c) 1993, 2015 Princeton Satellite Systems, Inc.
% All rights reserved.
% -----
% Since 1.1
% -----

if( nargin < 1 )
    T = [];
end

% -----
% $Date: 2016-03-15 15:32:35 -0400 (Tue, 15 Mar 2016) $
% $Revision: 41893 $

```

where we have excerpted the code creating the default input.

### 3.3.4 Built-in plotting

Many of the functions in the toolbox will plot the results if there are no output arguments. In many cases, you do not need any input arguments to get useful plots due to the built in defaults, but you can also generate plots with your own inputs. Calling for example `EarthRad` by itself will generate a plot of the absorbed flux per unit area as shown in Example 3.1 on page 24. If no inputs are given it automatically computes the earth albedo for a range of altitudes. If you want the same altitudes (first input) but different absorptivity and flux (second and third inputs), you can pass empty for the first argument,

```
>> EarthRad( [], 0.3, 320 )
```

and a plot using those values will be generated.

### 3.3.5 Error checking

Many functions perform error checking. However, functions that are designed to be called repeatedly, for example the right-hand-side of a set of differential equations tend not have error checking since the impact on performance would be significant. In that case, if you pass it invalid inputs you will get a MATLAB error message.

### 3.3.6 Variable inputs

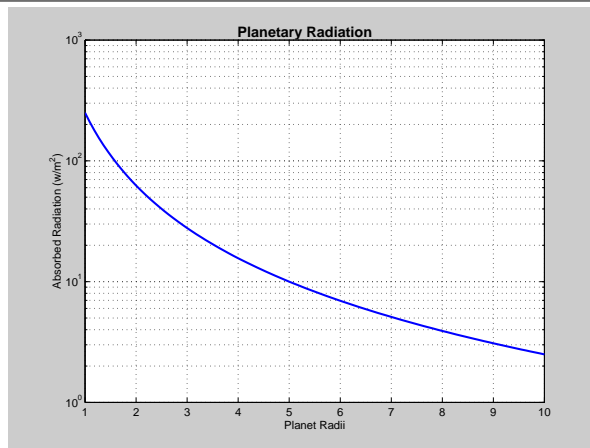
Some functions can take different kinds of inputs. An example is `Date2JD`. You can pass it either an array

```
[ year month day hour minute seconds ]
```

or the data structure

**Example 3.1** Earth radiation

EarthRad



```
d.month
d.day
d.year
d.hour
d.minute
d.second
```

The options are listed in the header.

## 3.4 Example Functions

The following sections gives examples for selected functions from the major folders of the SCT core toolbox.

### 3.4.1 Attitude

Consider stability for a dual spin spacecraft in geosynchronous orbit. Many commercial satellite are dual-spin stabilized since such a design is very robust.

```
>> dSS = DSpnStab( 1, 100, 1000, 1000, 0, 7.291e-5 )
>> DSpnStab( 1, 100, 1000, 1000 )
```

The result is  $dSS = 0$  which means the system is unstable! A stability plot is shown in Figure 3.2 on page 25. One means stable. `DSpnStab` is a typical function in SCT. If no outputs are specified it generates a plot.

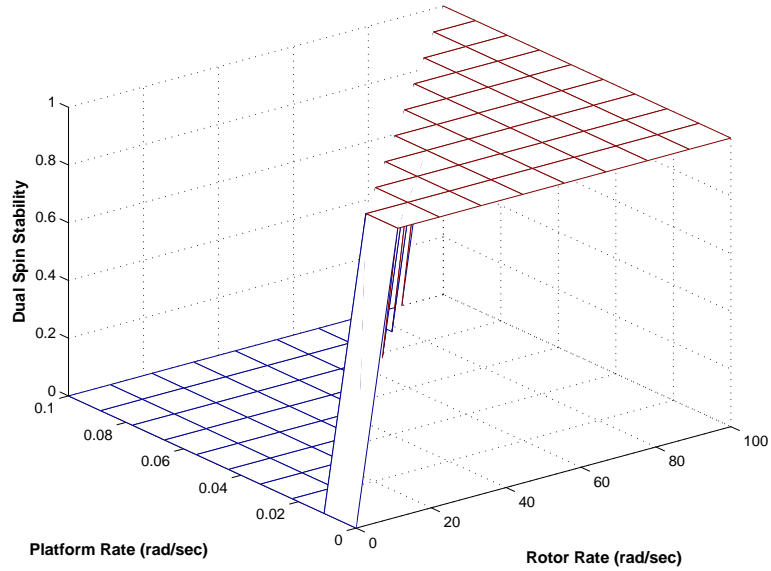
Next, compare magnetic torquers with thrusters at geosynchronous altitude. The specific impulse is 120 sec. and the moment arm is 0.8 m. The plot in Example 3.2 on the next page shows the trade-off between thrusters and magnetic torquers for geosynchronous orbit where the magnetic field is 75 nano-Tesla. The specific impulse assumes very short pulses.

### 3.4.2 Basic Orbit

`RVFromKepler` uses Kepler's equation to propagate the position and velocity vectors. The output of the demo is shown in Figure 3.3 on the facing page.

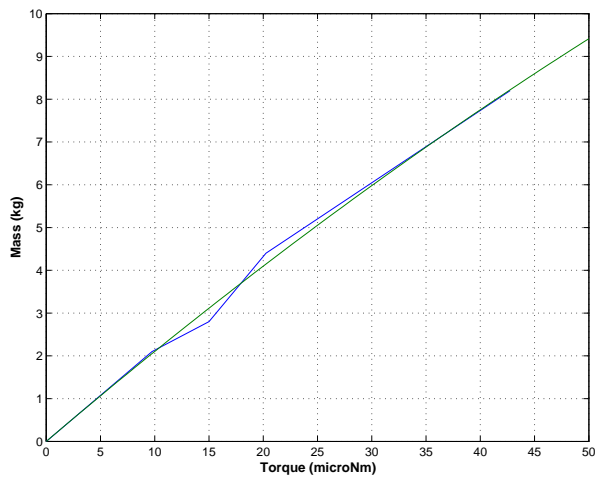
```
>> e1 = [8000,0.2,0,0,0.6,0]; RVFromKepler( e1 )
```

**Figure 3.2:** Dual Spin Stability

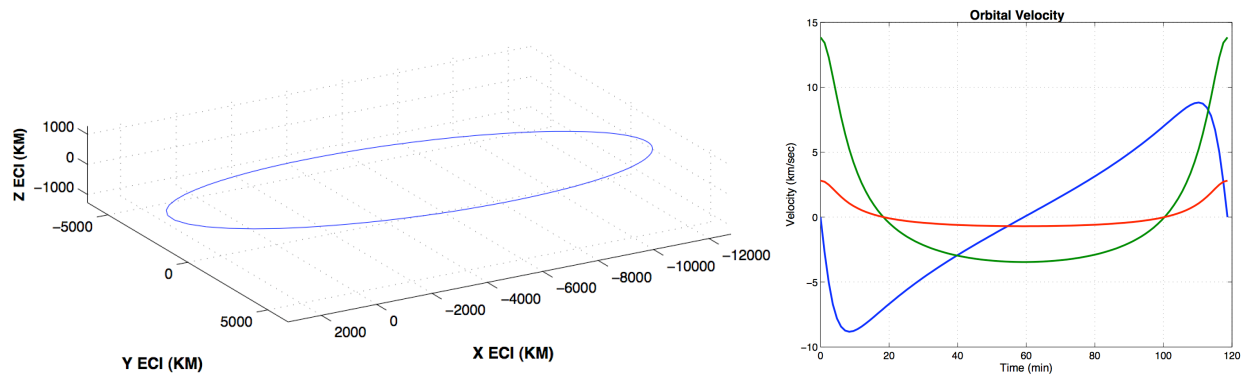


**Example 3.2** Torquer and thruster comparison results

MagTComp( 120, 0.8, 75e-9 )



**Figure 3.3:** Elliptical orbit from RVFromKepler



```
>> [r,v] = E12RV( e1 )

r =
    3200
     0
     0

v =
     0
 13.83596536017765
  2.80468902945837
```

### 3.4.3 Coord

Coord has coordinate transformation functions. Many quaternion functions are included. For example, to transform the vector  $[0;0;1]$  from ECI to LVLH for a spacecraft in a low earth orbit type

```
1 >> q = QLVLH( [7000;0;0],[0;7.5;0])
2 >> QForm( q, [0;0;1] )
3
4 q =
5         0.5
6         0.5
7         0.5
8        -0.5
9 ans =
10     0
11    -1
12     0
```

### 3.4.4 CAD

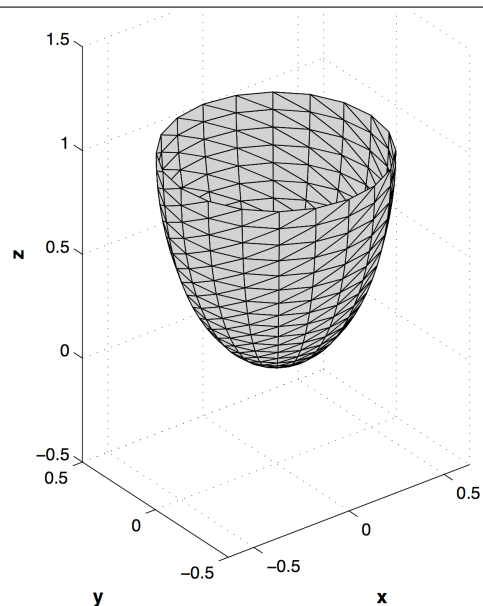
Many CAD functions are available to draw spacecraft components. Type `AntennaPatch` to get an antenna represented as part of an ellipsoid, as in Example 3.3 on page 26.

---

#### Example 3.3 Antenna patch

---

AntennaPatch



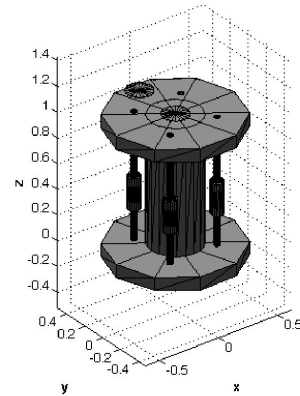
A Hall thruster can be drawn with `HallThrusterModel` as shown in Example 3.4 on the facing page.

---

### Example 3.4 Hall thruster

---

`HallThrusterModel`




---

### 3.4.5 Control

To convert a state space matrix from continuous time to discrete time use `C2DZOH` as shown below. In this case the example is for a double integrator with a time step of 0.5 seconds.

```
>> C2DZOH([0 1;0 0], [0;1], 0.5 )
ans =
           1           0.5
           0           1
```

### 3.4.6 Dynamics

This function can return either a state-space system or the state derivative vector. `MBModel` has analytical expressions for the state space system of a momentum bias spacecraft.

```
>> [a, b, c, d] = MBModel( diag([ 10000 2000 10000]), [0;400;0], [0;7.291e-5;0]);
>> eig(a)
```

The resulting plant has a double integrator for pitch. Roll/yaw has two pairs of complex eigenvalues, one at orbit rate (due to pointing at the earth) and one at the nutation frequency. Both modes are undamped.

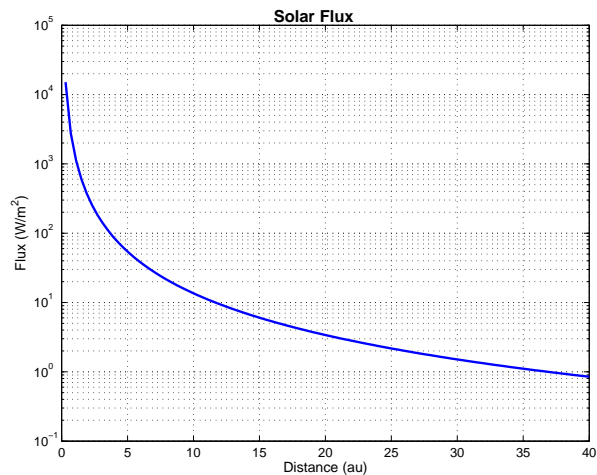
```
ans =
    0 + 0.000072910000000i
    0 - 0.000072910000000i
    0 + 0.039941672000000i
    0 - 0.039941672000000i
    0
    0
```

### 3.4.7 Environs

Models are available to compute solar flux, planetary radiation, magnetic fields and atmospheric properties. `SolarFlx` computes flux as a function of astronomical units from the sun. The function's output is shown in Example 3.5 on the next page.

**Example 3.5** Solar flux from the sun

SolarFlx

**3.4.8 Ephem**

The ephemeris directory has functions that compute the location and the orientation of the Earth and planets.

For example, to locate the inertial Sun vector for a spacecraft orbiting the Earth using an almanac model,

```
>> [u, r] = SunV1( JD2000, [7000;0;0] )
u =
    0.18006
   -0.90249
   -0.39127
r =
  1.4729e+08
```

which returns a unit vector to the sun from the spacecraft and the distance from the origin to the sun. SunV2 provides the same inputs and outputs and uses a higher precision sun model.

**3.4.9 Graphics**

Plot2D is used to plot any two dimensional data. It simplifies your scripts by making most popular plotting options available through a single function. Plot2D will print out a scalar answer if the inputs are scalar. See Figure 3.4 on page 29.

```
>> angle = linspace(0,4*pi);
>> Plot2D(angle, sin(angle), 'Angle_(rad)', 'Sine', 'Sine')
```

There are many other plot support functions such as AddAxes, Plot3D and PlotOrbitPage.

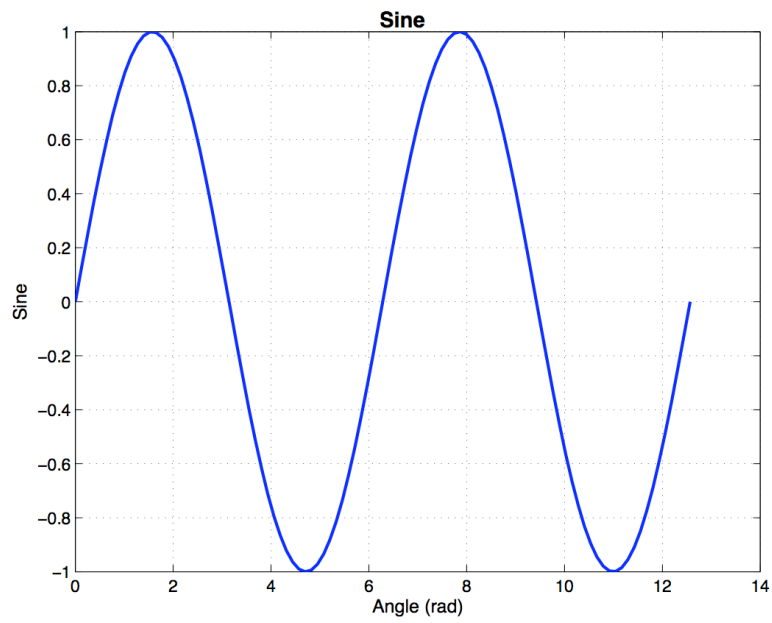
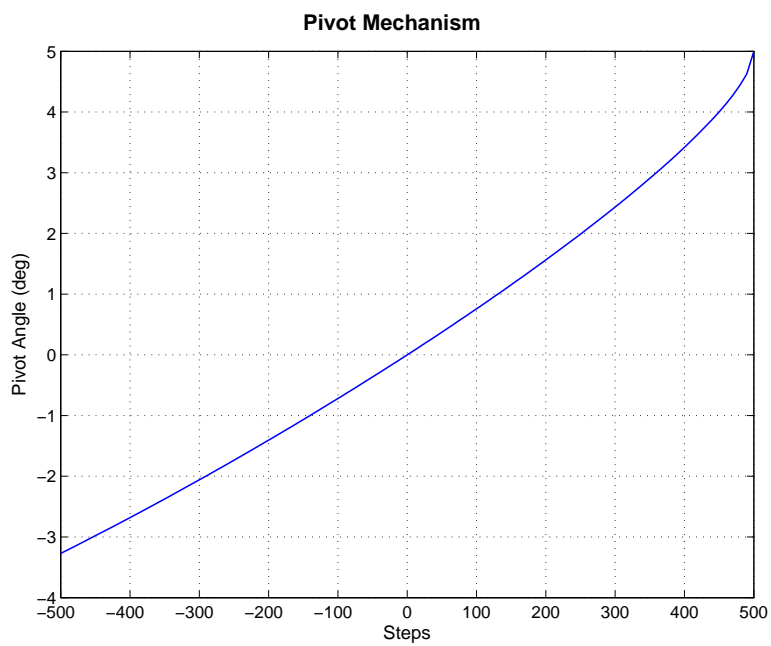
**3.4.10 Hardware**

A pivot is a rotation actuator used to tile spacecraft components. Most pivot mechanisms consist of a hinge and a lead screw driven by a stepping motor via a gear. Pivot mechanisms are nonlinear for large angles. If you type

```
>> PivotMch
```

you get a plot of a pivot mechanism angle versus pivot steps (Figure 3.5 on the next page).



**Figure 3.4:** A sine wave using `Plot2D`**Figure 3.5:** Pivot mechanism

### 3.4.11 Time

The Time directory has functions that convert between various time conventions. The most widely used function is to convert calendar date to Julian Date.

```
>> jD = Date2JD
>> JD2Date(jD)

jD =
    2.451251504154273e+06
ans =
    [1999    3    14  0  0.005  0.0589292138814]
```

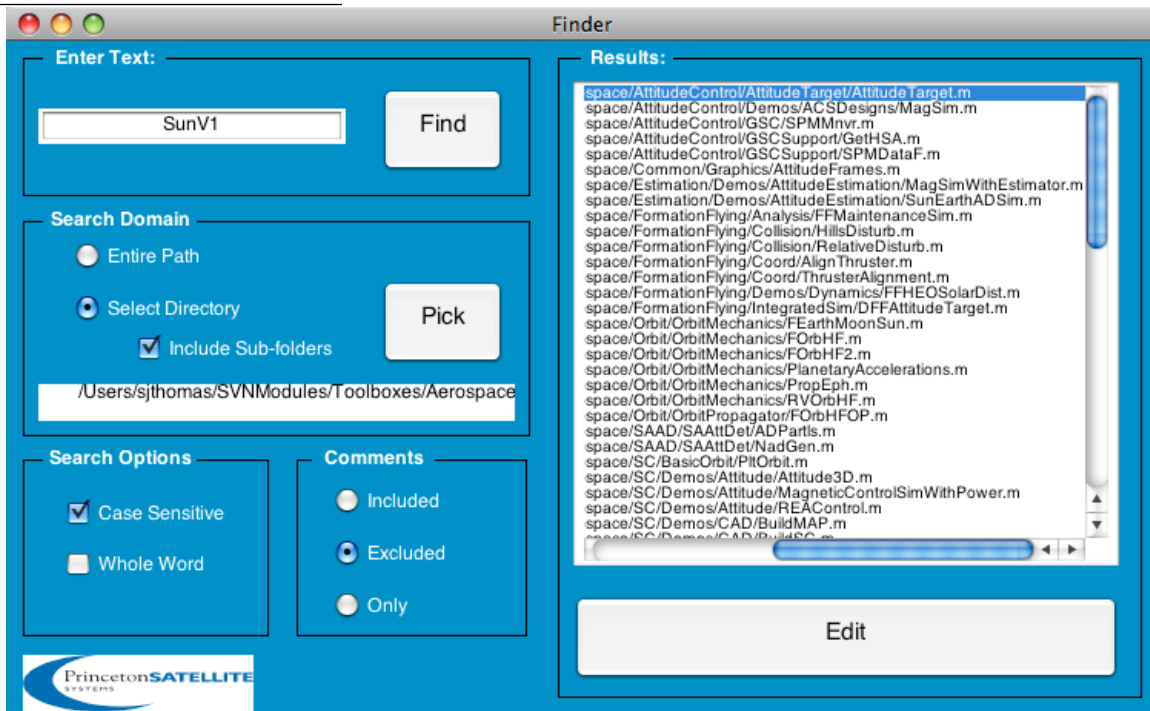
## 3.5 GUI Tools in the Toolbox

### 3.5.1 Finder GUI

The `Finder` GUI, shown in Figure 3.6 on page 31, is a handy function for searching for information in the toolbox. You can search for instances of keywords in the entire body of functions and demos, not just the help comments. You can use this function with any toolboxes, not just your PSS toolboxes, since this actively searches the files every time instead of using a parsed version of the headers the way `FileHelp` does. Consequently, it is a little slower to use, but you can use it with your own function libraries, too.

The `Finder` function has options for searching the entire path or a selected directory. The `Pick` button brings up a file selection dialog where you can navigate to your desired directory. The search can be case sensitive and you can select whole word matching. You can search on just file help comments, or include or exclude them. For example, you can find all functions and demos that actually use the function `PIDMIMO` by searching with comments excluded. Once your search results are displayed in the Results list, you can open any file by clicking the `Edit` button.

Figure 3.6: Finder window





# SPACECRAFT CONTROL TUTORIAL

This tutorial is implemented in the MATLAB function `SCTutorial`.

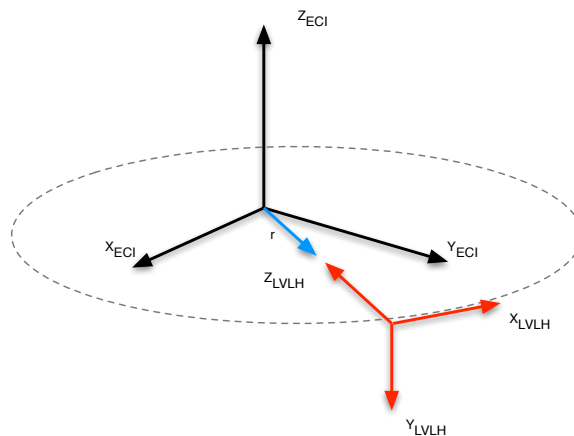
**Step 1 - Dynamics:** We start with the general kinematics and dynamics.

$$\dot{q} = f(q, \omega) \quad (4.1)$$

$$T = I\dot{\omega} + \omega^\times I\omega \quad (4.2)$$

where  $q$  is the spacecraft's attitude quaternion,  $\omega$  is the body rates,  $I$  is the inertia, and  $T$  is the torque applied. We need to linearize these equations in the LVLH (local-vertical-local-horizontal) frame. The inertial (ECI) and LVLH frames are shown in Figure 4.1 on page 33.

**Figure 4.1:** ECI and LVLH Coordinate Frames



Assuming that the spacecraft is earth-pointing, the LVLH body rate in the  $y$  axis is constant and equal to the spacecraft's orbit rate,  $\omega_0$ .

$$\dot{\theta} = \begin{bmatrix} \omega_x + \omega_0\theta_z \\ \omega_0 \\ \omega_z - \omega_0\theta_x \end{bmatrix} \quad (4.3)$$

We see that the  $x$  and  $z$  equations are coupled. However, if we look at the magnitude of the coupling terms, we see that they are quite small, so we drop those terms. The resulting linearized dynamics equation is

$$T = I\dot{\omega} \quad (4.4)$$

This is a simple double integrator ( $1/s^2$ ).

**Step 2 - Control:** We will use PSS' PIDMIMO control design function, which performs automatic pole placement for a double integrator system. It produces a state-space 3 axis PID controller of the form

$$x_{k+1} = Ax_k + Bu_k \quad (4.5)$$

$$y_k = Cx_k + Du_k \quad (4.6)$$

by designing in the frequency domain and converting to discrete time using a zero-order hold. The continuous time equivalent for each axis is

$$y = K_p u + \frac{K_r s}{s + \omega_R} u + K_i \frac{u}{s} \quad (4.7)$$

The function takes the following inputs:

**Table 4.1:** PIDMIMO input parameters

Inputs	Size	Description	Notes
inr	(n,n)	Inertia matrix	
zeta	(n,1)	Vector of damping ratios	A large zeta will cause the system to be sluggish. General practice is to set zeta for critical damping.
omega	(n,1)	Vector of undamped natural frequencies	The bandwidth cant be more than 1/2 the sampling time, and generally is much less. For a 4 Hz sampling rate, we use an omega of 0.1 rad/sec.
tauInt	(n,1)	Vector of integrator time constants	This is the time to cancel a steady disturbance. We need this term because delta-V thrusters are always misaligned, so there is a constant disturbance when we are using them. However, is tauInt is too fast, then the system is hard to stabilize with delays.
omegaR	(n,1)	Vector of derivative term roll-off frequencies	This is the derivative filter. It should be 5 times faster than the bandwidth of the system. Using a derivative filter reduces the need for a low-pass filter.
tSamp	1	Sampling period	Generally 0.25 sec for a spacecraft control system, but should be at least 5-10 times faster than the fastest frequency of the controller.
sType	:	State equation type ('Delta' or 'Z')	Delta ( $x = x + Ax + Bu$ ) is better from a numeric point of view.

The function returns the  $a$ ,  $b$ ,  $c$ , and  $d$  matrices of the state-space control system plus a structure of the forward, rate, and integral gains. The last two inputs are optional and result in a discrete control system; without them the function returns the continuous state-space system. See C2DZOH for more information on the discretization.

The function call with sample output is

```
>> [a, b, c, d, k] = PIDMIMO( 1, 0.7071, 0.1, 100, 1, 0.25, delta)
a =
      0      0
      0 -0.259968823501459
b =
      0.25
  0.25996882350146
c =
  0.000521750104942956 -0.207366539452617
d =
  0.223137533733209
k =
  kR: 10.918482508346
  kF: 0.0157709942805926
  kI: 0.0330828922806096
```

**Step 3 - Simulation:** Now, we can simulate our controller with a real spacecraft model. The inertia matrix is

```

inertia = diag([24.5 10 25])
inertia =
           24.5           0           0
           0           10           0
           0           0           25

```

which is representative of a spacecraft with long solar arrays in the  $y$  axis and a nearly symmetric cubic bus. We'll use rigid body dynamics and a 4th order Runge-Kutta integrator. The dynamics update step looks like

```
x = RK4( 'FRB', x, tSamp, t(k), inertia, invInertia, tExt+tDist );
```

where RK4 and FRB are both SCT functions. FRB implements a rigid-body right-hand-side for a state including the attitude quaternion and the body rates. The controller implementation in delta form looks like

```

accel(1) = c*xRoll + d*angleError(1);
xRoll    = xRoll  + a*xRoll  + b*angleError(1);

accel(2) = c*xPitch + d*angleError(2);
xPitch   = xPitch + a*xPitch + b*angleError(2);

accel(3) = c*xYaw   + d*angleError(3);
xYaw     = xYaw   + a*xYaw   + b*angleError(3);

tExt     = -inertia*accel;

```

where each axis is computed separately and the  $x$  vector for each axis contains 2 states, for angle and rate. Since we simulate the quaternion which transforms from the ECI frame to the body frame, we calculate the angle error from the calculated body-to-LVLH quaternion.

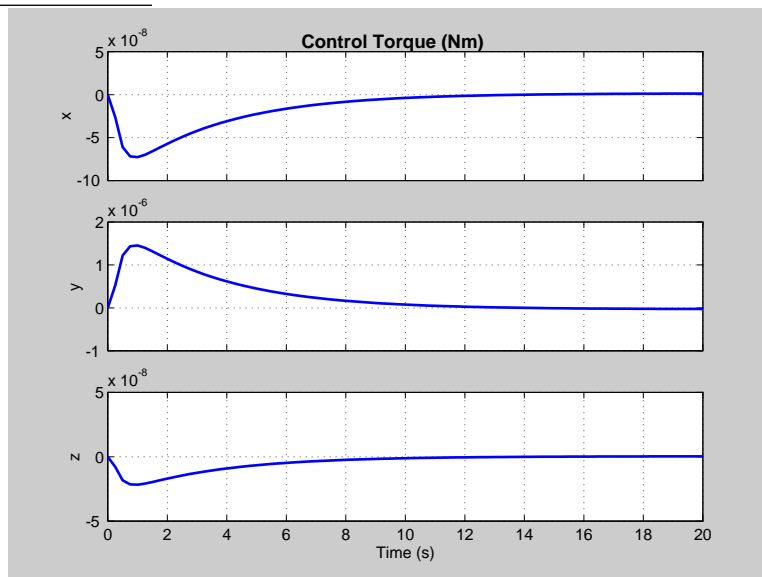
```

qECIToLVLH = QLVLH( rECI, vECI );
qBodyToLVLH = QPose( QMult( QPose(qECIToBody), qECIToLVLH ) );
angleError = -2*qBodyToLVLH(2:4);

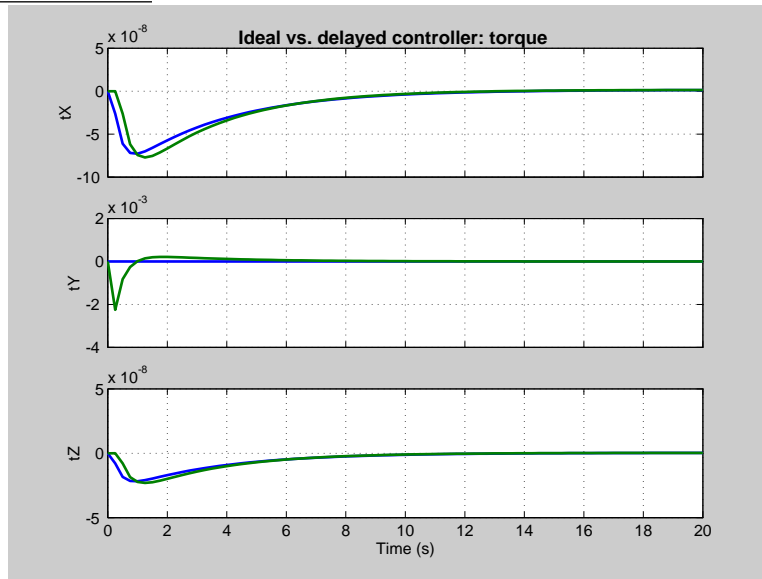
```

To test the controller, we give each axis an initial disturbance torque on the order of  $1 \times 10^{-6}$  Nm. The disturbance torque  $tDist$  is nonzero only for the first step. The resulting control torques are shown in Figure 4.2 on page 35.

**Figure 4.2:** Control torques



Next, we simulate the controller again, this time with a delay equal to one sampling time. We see in Figure 4.3 on the next page that our controller does fine with this delay.

**Figure 4.3:** Control torque, ideal and with a 0.25 sec delay

**Notes on real designs:** In a real stationkeeping system, one must account for a number of effects that are not modeled in this simple example, such as:

- minimum pulsewidth of the control actuators (thrusters)
- noise filters
- unmodelled dynamics (disturbances like drag, etc.)
- delays in the system

The minimum pulsewidth for thrusters is often 20 ms, so any control torque we applied in the simulation resulting in a smaller pulsewidth is actually not physically possible. Accounting for this finite resolution results in limit cycling about the desired control point.

Noise filters are generally needed in a real control system. They can help account for both noise and unmodelled dynamics. Typical types are notch and roll-off, see for example `Notch`.

Delays may come from sensors, other hardware, or software. Once identified, they can be accounted for in the controller gains. For example, PSS once had to upload new gains to a geosynchronous spacecraft when a 2 second delay was discovered in the earth sensor after launch!



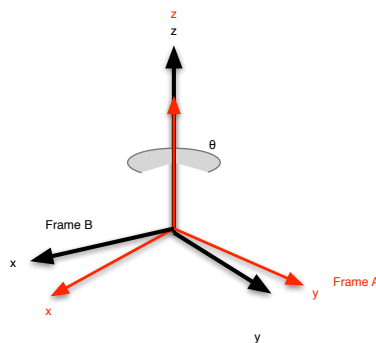
# COORDINATE TRANSFORMATIONS

This chapter shows you how to use Spacecraft Control Toolbox functions for coordinate transformations. There is a very extensive set of functions in the `Common/Coord` folder covering quaternions, Euler angles, transformation matrices, right ascension/declination, spherical coordinates, geodetic coordinates, and more. A few are discussed here. For more information on coordinate frames and representations, please see the Coordinate Systems and Kinematics chapters in the accompanying book *Spacecraft Attitude and Orbit Control*.

## 5.1 Transformation Matrices

Transforming a vector  $u$  from its representation in frame  $A$  to its representation in frame  $B$  is easily done with a transformation matrix. Consider two frames with an angle  $\theta$  between their  $x$  and  $y$  axes.

**Figure 5.1:** Frames A and B



```

1 uA = [1;0;0];
2 theta = pi/6;
3 m = [ cos(theta), sin(theta), 0;...
4       -sin(theta), cos(theta), 0;...
5       0, 0, 1];
6 uB = m*uA

```

Using SCT functions, this code can be written as a function of Euler angles using a rotation about the  $z$  axis:

```
uB = Eul2Mat([0, 0, theta])*uA;
```

Use `Mat2Eul` to switch back to an Euler angle representation.

## 5.2 Quaternions

A quaternion is a four parameter set that embodies the concept that any set of rotations can be represented by a single axis of rotation and an angle. PSS uses the shuttle convention so that our unit quaternion (obtained with `QZero`) is `[1 0 0 0]`. In Figure 5.1 on page 37 the axis of rotation is `[0 0 1]` (the  $z$  axis) and the angle is `theta`. Of course, the axis of rotation could also be `[0 0 -1]` and the angle `-theta`.

Quaternion transformations are implemented by the functions `QForm` and `QTForm`. `QForm` rotates a vector in the direction of the quaternion, and `QTForm` rotates it in the opposite direction. In this case

```
q = Mat2Q(m);
uB = QForm(q,uA)
uA = QTForm(q,uB)
```

We could also get `q` by typing

```
q = Eul2Q([0;0;theta])
```

Much as you can concatenate coordinate transformation matrices, you can also multiply quaternions. If `qAToB` transforms from  $A$  to  $B$  and `qBToC` transforms from  $B$  to  $C$  then

```
qAToC = QMult(qAToB,qBToC);
```

The transpose of a quaternion is just

```
qCToA = QPose(qAToC);
```

You can extract Euler angles by

```
eAToC = Q2Eul(qAToC);
```

or matrices by

```
mAToC = Q2Mat(qAToC);
```

If we convert the three Euler angles to a quaternion

```
qIToB = Eul2Q(e);
```

`qIToB` will transform vectors represented in  $I$  to vectors represented in  $B$ . This quaternion will be the transpose of the quaternion that rotates frame  $B$  from its initial orientation to its final orientation or

```
qIToB = QPose(qBInitialToBFinal);
```

Given a vector of small angles `eSmall` that rotate from vectors from frame  $A$  to  $B$ , the transformation from  $A$  to  $B$  is

```
uB = (eye(3)-SkewSymm(eSmall))*uA;
```

where

```
SkewSymm([1;2;3])
ans =
[0 -3  2;
 3  0 -1;
-2  1  0]
```

Note that `SkewSymm(x)*y` is the same as `Cross(x,y)`.

## 5.3 Coordinate Frames

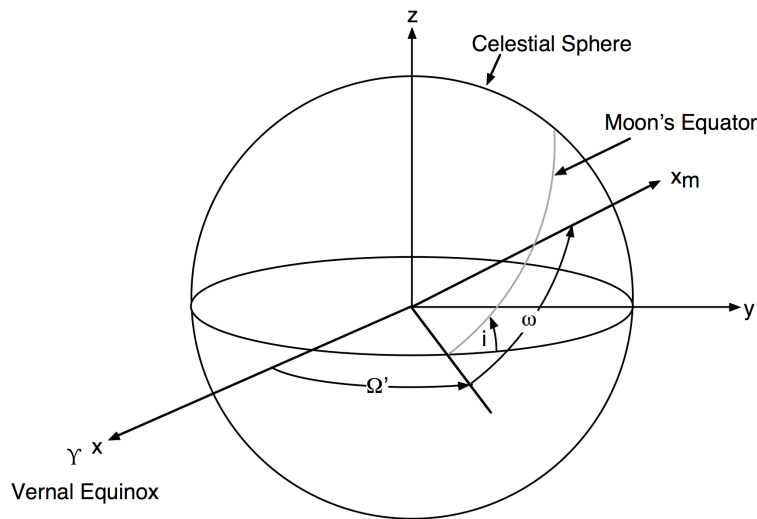
The toolbox has functions for many common coordinate frames and representations, including

- Earth-fixed frame, aerographic (Mars), selenographic (Moon)
- Latitude (geocentric and geodetic), longitude, and altitude
- Earth-centered inertial
- Local vertical, local horizontal
- Nadir and sun-nadir pointing
- Hills frame
- Rotating libration point
- Right ascension and declination
- Azimuth and elevation
- Spherical, cartesian, and cylindrical

Most of these functions can be found in `Coord` and some are in `Ephem` due to their dependence on ephemeris data such as the sun vector. A few relevant functions are in `OrbitMechanics`. For example, the `QLVLH` function in `Coord` computes a quaternion from the inertial to local-vertical local-horizontal frame from the position and velocity vectors. `QNadirPoint`, also in `Coord`, aligns a particular body vector with the nadir vector. The `QSunNadir` function in `Ephem` computes the sun-nadir quaternion from the ECI spacecraft state and sun vector.

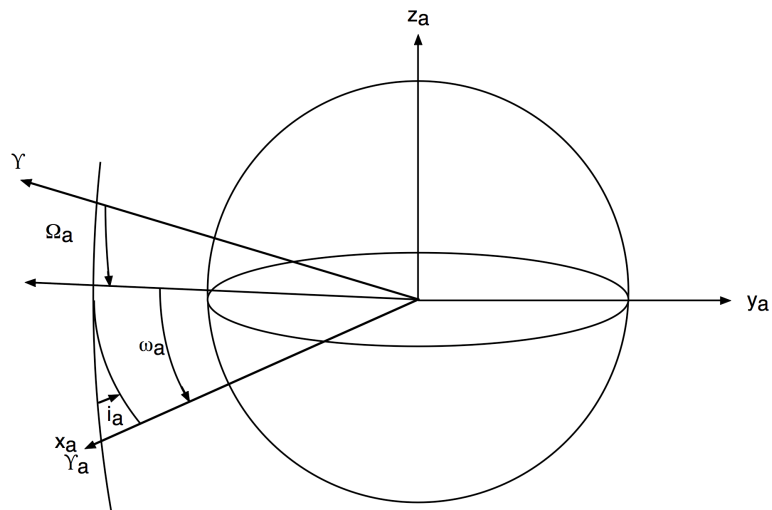
The relationship between the Selenographic and the ECI frame is computed by `MoonRot` and shown in Figure 5.2 on the next page.  $\phi'_m$  is the Selenographic Latitude which is the acute angle measured normal to the Moon's equator

**Figure 5.2:** Selenographic to ECI frame



between the equator and a line connecting the geometrical center of the coordinate system with a point on the surface of the Moon. The angle range is between 0 and 360 degrees.  $\lambda_m$  is the Selenographic Longitude which is the angle measured towards the West in the Moon's equatorial plane, from the lunar prime meridian to the object's meridian. The angle range is between -90 and +90 degrees. North is the section above the lunar equator containing Mare Serenitatis. West is measured towards Mare Crisium.

The areocentric frame computed by `MarsRot` is shown in Figure 5.3 on the following page.  $\Omega_a$ ,  $\omega_a$  and  $i_a$  are the standard Euler rotations of the Mars vernal equinox,  $\Upsilon_a$  with respect to the Earth's vernal equinox,  $\Upsilon$ .

**Figure 5.3:** Areocentric frame

# USING CAD FUNCTIONS

---

## 6.1 Introduction

---

This chapter shows you how to use the CAD functions. These functions allow you to work with 3-dimensional representations of your spacecraft. You can assemble spacecraft from components and use the resulting models for multi-body simulations, disturbance analysis and visualization.

The toolbox provides several CAD functions to simplify your design and simulation tasks. You can create a spacecraft model with `BuildCADModel`, or import it (.dxf, .obj and .3DMF files) using the `LoadCAD` function. You can interactively draw the spacecraft using `DrawSCPlanPlugIn` and visualize its attitude and orbit motion by passing the kinematic and position vectors to `DrawSCPlugIn`.

There are many demos showing you how to create both simple and complex models. Most of the model demos begin with `Build`. There are also demos showing how to use the models for visualization inside simulations and disturbance analysis.

## 6.2 Building a Spacecraft Model Using `BuildCADModel`

---

### 6.2.1 Introduction

`BuildCADModel` is script based and is only used to view the results of your model building. All model building is done in a script. This way it is easy to write loops to add components, etc. The process for building a CAD model is

1. Define spacecraft properties
2. Create bodies
3. Create components and assign them to bodies
4. View with `BuildCADModel`
5. Export, if desired, or save the resulting data structure

All functions are done with `BuildCADModel`. Bodies, rigid assemblies such as a rotating solar array on a strut, are created with the function `CreateBody` and components are created with the function `CreateComponent`. If you call `BuildCADModel` with no inputs, a list of all possible actions for the function will be printed.

```
>> BuildCADModel
```

```

User Actions
'initialize'
'add_units'
'add_name'
'set_aerodynamic_model'
'set_aerodynamic_model_file'
'add_reci'
'add_veci'
'add_qecitobody'
'add_qlvlh'
'add_omega'
'add_mass'
'set_mass'
'add_body'
'add_component'
'lock_mass_properties'
'unlock_mass_properties'
'update_body_mass_properties'
'spacecraftplugin'
'create_body_arrays'
'add_subsystem'
'add_subassembly'
'add_panel'
'compute_paths'
'table'
'export_cad_model'
'get_cad_model'
'quit'
'help'

```

## 6.2.2 Geometry

Before we delve any further into using `BuildCADModel` we should establish the geometry of the models. All surface geometry is specified at the component level and results in a set of triangles. The triangles are defined by two fields, `faces` and `vertices`, in the same format in which patches are stored in MATLAB, see `patch`. For example,

---

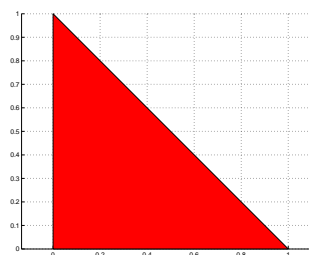
### Example 6.1 Simple patch with one triangle

---

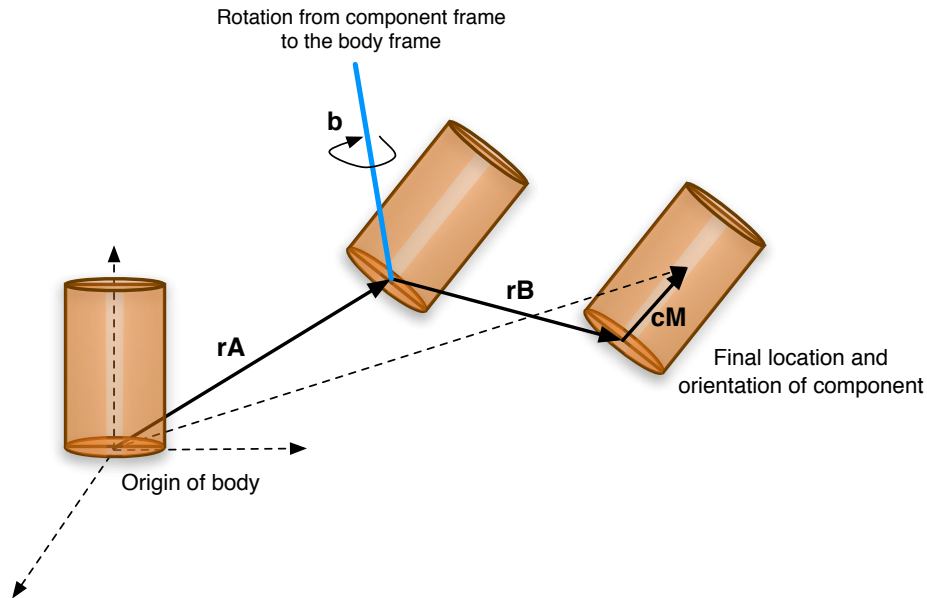
```

1 >> p = patch([0 1 0],[0 0 1], 'r');
2 >> v = get(p, 'vertices')
3 v =
4     0     0
5     1     0
6     0     1
7
8 >> f = get(p, 'faces')
9 f =
10    1     2     3

```



We mentioned above that each model consists of bodies and components. Each component is defined in a local frame, for example the cylinder primitive in the SCT has height along the  $Z$  axis. The component center of mass is defined in this frame. In the case of a cylinder, this is measured from the base,  $cM = [0; 0; h/2]$ . The inertia matrix is defined about the center of mass. Each component can then be placed in a body using two vectors and a rotation in between them. The vectors are depicted in Figure 6.1 on page 43 with a cylinder primitive. The matrix  $b$  rotates from the component frame to the body frame.  $rA$  is the displacement of the unrotated component, i.e. *after* rotation to the body frame, and  $rB$  is *before*. Most components in the SCT examples are placed using  $rA$  and  $b$  without  $rB$ .

**Figure 6.1:** Component geometry and frames

The equation for the geometric location of the component's center of mass ( $r_{CM}$  or  $cM$ ) in the body frame is

$$r_{CM}|_{body} = r_A + b * (r_B + r_{CM}|_{comp})$$

where the component center of mass is measured in its frame as defined by  $b$ .

The bodies are defined in a tree configuration. Each body has a hinge vector defined in the frame of the previous body. The hinge rotation matrix also expresses the rotation into the previous body's frame. The center of mass of the body, computed by `BuildCADModel` from the included components, is referenced from the hinge vector in the body frame.

Assume a second body is attached to the core. The core hinge rotation is  $B_{core}$  and the second body hinge rotation is  $B_H$ . The second body's hinge vector is  $r_H$ . The equation for the location of the second body's components in the core frame is

$$r = B_{core} * (r_H + B_H * (r_A + b * r_B))$$

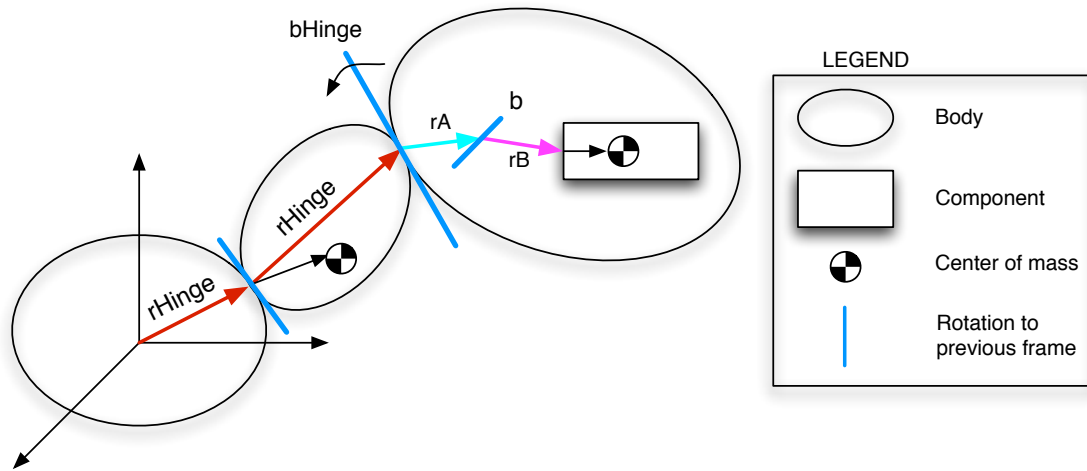
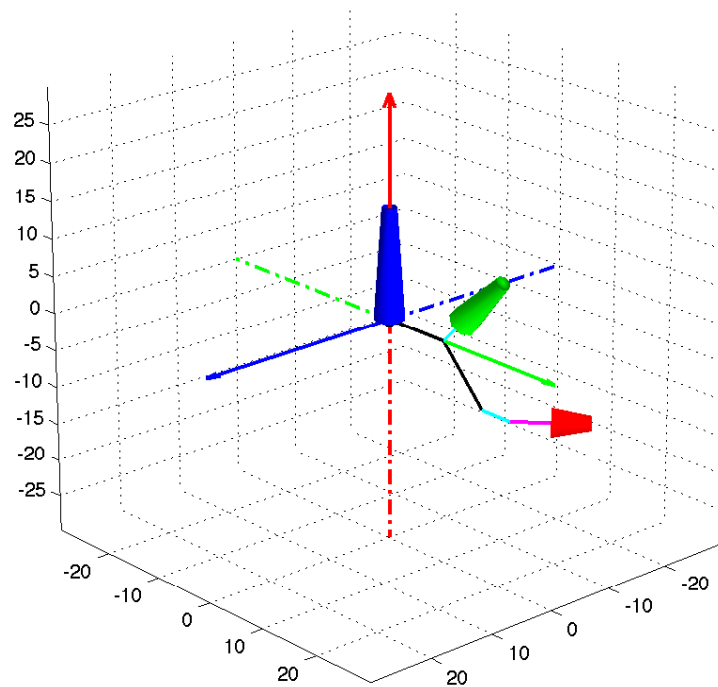
where we see again that the hinge vector is defined in the previous body's frame,  $r_A$  is in the body frame and  $r_B$  and  $cM$  are in the component frame. The body's center of mass in the core frame is

$$r_{CM}|_{core} = r_H + B_H * (r_{CM}|_{body})$$

Figure 6.2 on page 44 shows the topological tree for three bodies with one component drawn.

See the example `BuildCylinders` for a systematic treatment of the vectors and rotations. The resulting plot is shown in Figure 6.3 on the next page.

The vectors and rotations described here are explained further in the sections on bodies and components later in this chapter, Section 6.2.5 and Section 6.2.6.

**Figure 6.2:** Body geometry and frames**Figure 6.3:** Cylinder demo of geometry and frames. Hinge vectors are in black,  $r_A$  is cyan and  $r_B$  is magenta. The axes are coded X (blue), Y (green), and Z (red).



### 6.2.3 A Simple Example

A very simple spacecraft with one body and two components is given in the following listing from `BuildSimpleSat`. The components are a core box and a single sensor. We can use this example to illustrate the steps in building a model and to begin a discussion of the CAD properties and fields.

**Listing 6.1:** Simplest spacecraft CAD script

*BuildSimpleSat.m*

```

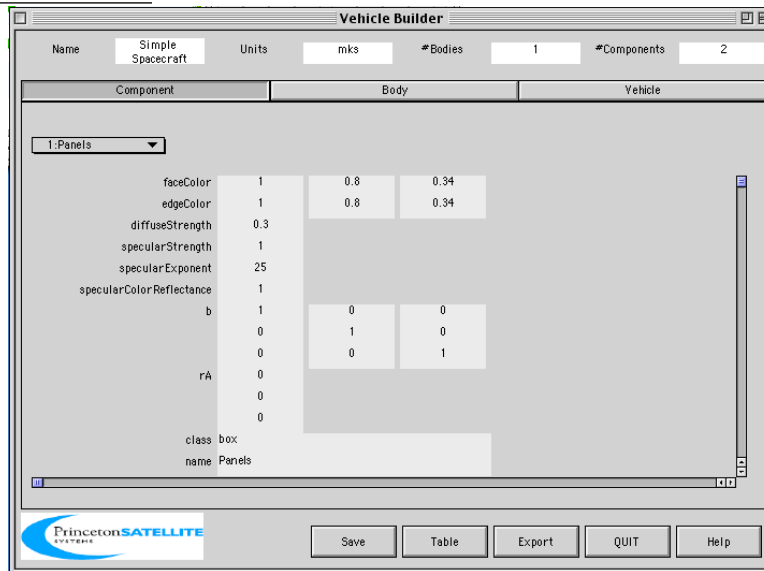
1 %% A very simple spacecraft with one sensor.
2 % -----
3 % See also BuildCADModel, CreateBody, CreateComponent, FindDirectory,
4 % SaveStructure
5 % -----
6 %%
7 % -----
8 % Copyright (c) 2005, 2007 Princeton Satellite Systems, Inc.
9 % All rights reserved.
10 % -----
11
12 %% Initialize
13 %-----
14 BuildCADModel( 'initialize' );
15
16 % Add spacecraft properties
17 %-----
18 BuildCADModel( 'set_name' , 'Simple_Spacecraft' );
19 BuildCADModel( 'set_units', 'mks' );
20 BuildCADModel( 'set_qecitobody', [1;0;0;0] );
21 BuildCADModel( 'set_reci', [7000;0;0] );
22
23 % Create bodies first
24 %-----
25 m = CreateBody( 'make', 'name', 'Core' );
26 BuildCADModel( 'add_body', m );
27
28 % This creates the connections between the bodies
29 %-----
30 BuildCADModel( 'compute_paths' );
31
32 %% Add Components
33 %-----
34
35 % Core
36 %-----
37 m = CreateComponent( 'make', 'box', 'name', 'Panels', 'x', 1, 'y', 1, 'z', 1, ...
38     'faceColor', 'gold_foil', 'inside', 0, ...
39     'rA', [0;0;0], 'mass', 20, 'body', 1 );
40 BuildCADModel( 'add_component', m );
41
42 % Star Camera
43 %-----
44 m = CreateComponent( 'make', 'star_camera', 'model', 'ct633', 'name', 'Camera', ...
45     'rA', [0.4;0.4;0.4], 'body', 1, 'boresight', [1;0;0], ...
46     'faceColor', 'aluminum' );
47 BuildCADModel( 'add_component', m );
48
49 %% Save the model
50 %-----
51 g = BuildCADModel( 'get_cad_model' );
52 BuildCADModel( 'show_spacecraft' );
53 d = FindDirectory( 'SCModels' );
54 SaveStructure( g, fullfile(d, 'SimpleSat' ) )
55
56 % -----
57 % PSS internal file version information

```

*BuildSimpleSat.m*

If you execute this script you will see Figure 6.4 on page 46. You can't change any properties using this GUI but you

**Figure 6.4:** Simple Sat



can view the properties at the component, body and vehicle levels. See Section 6.2.9 for more information on using this GUI.

The `CreateBody` and `CreateComponent` functions are described further in later sections. Each function is passed an action, in this case `make`, and then a set of parameter/value pairs. We can see from this example that the model has the following sections:

1. Properties, in this case the mass, can be grouped at the top of the script for easy reference.
2. The `BuildCADModel` function is initialized and some spacecraft parameters, such as the name and units, are set.
3. The bodies are created.
4. The paths between the bodies are computed. You must add this call even if there is only one body for the model to have the correct fields.
5. The components are created and assigned to bodies.
6. The completed model is extracted and stored as a mat-file.

We can note a few other things from this example. First, all components are considered to be inside the spacecraft by default; these components are ignored for the purposes of disturbance analysis. This enables you to have many components inside, such as reaction wheels and batteries, without all those extra surfaces slowing down the computations. As a result, any components which are *not* inside must be explicitly set to be so using `inside, 0`.

An ECI position vector and quaternion are required as spacecraft properties to draw the spacecraft in an orbit view. The additional fields you can set (`qlv1h`, `veci`, etc) enable you to use the CAD model as a database for default properties.

To view the resulting model structure, type `g` at the command line.

```
>> g
g =
    name: 'Simple_Spacecraft'
    units: 'mks'
    body: [1x1 struct]
```

```

component: [1x2 struct]
  radius: 8.660254037844386e-01
  mass: [1x1 struct]

```

All models, no matter how simple or complex, will have these fields. The name and units are straightforward. The body and component fields are structure arrays. The radius is the radius of a sphere encompassing the model. The mass is a structure with the total mass, inertia, and center of mass of the vehicle.

You might notice that we did not specify a mass for the camera. In fact, there are defaults for all the component parameters. To see what values were used, first get the list of component fields.

```

>> g.component
ans =
1x2 struct array with fields:
  faceColor
  edgeColor
  diffuseStrength
  specularStrength
  specularExponent
  specularColorReflectance
  b
  rA
  v
  f
  a
  n
  r
  radius
  deviceInfo
  class
  name
  optical
  infrared
  thermal
  power
  aero
  magnetic
  mass
  inside
  rF
  body
  manufacturer
  model

```

The components also have a field called mass. The camera was the second component, so we look at its mass:

```

>> g.component(2).mass
ans =
  mass: 2.837
  cM: [3x1 double]
  inertia: [3x3 double]
>> g.component(2).mass.cM
ans =
  0.54224
  0.4
  0.4
>> g.component(2).mass.inertia
ans =
  0.0064267 0 -5.4041e-18
  0 0.030765 0

```

```
-5.4041e-18      0      0.030765
```

The mass and inertia were looked up via `StarCameraModel`, for the CT633 model specified. The center of mass is referenced from the origin of the component's frame.

The mass properties of the spacecraft are automatically computed from the components when we retrieve the model using `get cad model`. We will see that the mass is the total of the two component masses and the inertia and center of mass are offset; the inertia has off-axis terms due to the camera. You will also find that the body properties and the vehicle properties are, in this case, the same.

```
>> g.mass
ans =
    mass: 22.837
    inertia: [3x3 double]
    cM: [3x1 double]
>> g.mass.inertia
ans =
    4.1348    -0.53889    -0.53889
   -0.53889     4.4922    -0.39753
   -0.53889    -0.39753     4.4922
>> g.mass.cM
ans =
    0.067362
    0.049691
    0.049691
```

You can more easily find the default values of properties by examining the function `GenericProperties`. This function is used by `CreateComponent`. The following is an excerpt from the `GenericProperties` header. The function has a built-in demo which prints a list of the properties with descriptions.

```
-----
Generates generic properties for type:

'thermal'
    'absoprtivity', 'emissivity', 'heatGeneratedOn', 'heatGeneratedStandby',
    'temperature', 'surfaceArea', 'uNormal', 'thermalMass'
'power'
    'powerStandby', 'powerOn', 'electricalConversionEfficiency',
    'powerHeater', 'powerIsOn'
'optical'
    'sigmaT', 'sigmaA', 'sigmaD', 'sigmaS'
'infrared'
    'sigmaRT', 'sigmaRA', 'sigmaRD', 'sigmaRS'
'mass'
    'mass', 'cM', 'inertia'
'aero'
    'cD'
'rF'
    'flux', 'u', 'r'
'magnetic'
    'dipole'
'propulsion'
    'thrust' 'iSP' 'feedPressure' 'efficiency'
'graphics'
    'faceColor' 'edgeColor' 'specularStrength' 'diffuseStrength'
    'inside'

You can customize properties by adding property/value pairs.
Type GenericProperties to have a list describing the properties printed to
the command window.
```

For example, we can check the default thermal properties directly.

```
>> p = GenericProperties('thermal')
p =
    absorptivity: 0.1000
    emissivity: 0.8000
    heatGeneratedOn: 0
    heatGeneratedStandby: 0
    temperature: 300
    surfaceArea: 1
    uNormal: [3x1 double]
    thermalMass: 1
```

## 6.2.4 Setting Spacecraft Properties

The following code sets selected spacecraft properties:

```
1 BuildCADModel( 'initialize' );
2 BuildCADModel( 'set_name', 'MySpacecraft' );
3 BuildCADModel( 'set_units', 'mks' );
4 BuildCADModel( 'set_rECI', rECI );
5 BuildCADModel( 'set_vECI', vECI );
6 BuildCADModel( 'set_qLVLH', qLVLH );
7 BuildCADModel( 'set_qECIToBody', q );
8 BuildCADModel( 'set_omega', omega );
9 BuildCADModel( 'set_mass', mass );
```

An orbit can be set for use by the spacecraft visualization tools, for instance by using variables called `rECI` and `vECI` as above. `BuildCADModel` will automatically initialize itself if you do not initialize it. Except for setting the name, the other calls are optional, except that a quaternion and ECI position are required for the orbit view. You can set the mass properties using the 'set mass' action or allow `BuildCADModel` to compute the mass from the component mass properties. This requires that you include all components to get an accurate mass and inertia. The input to mass is the mass data structure which can be assembled using, for example,

```
mass = MassStructure( 2100, 'box', [coreX coreY coreZ] );
```

which returns a data structure with the mass, inertia and center-of-mass.

## 6.2.5 Creating Bodies

The next step after setting the model-wide properties is to define the bodies. Bodies are essentially containers for components that allow you to define rigid assemblies which can be rotated independently. Mass properties including inertia and center of mass are computed at the body level in addition to the vehicle level. Topological trees defined via a CAD model can be simulated dynamically using PSS' `Tree` function. You must always define at least one body. The example below creates a total of four bodies, with the latter three bodies attached to the core (first body).

```
1 % The core
2 %-----
3 m = CreateBody( 'make', 'name', 'Core' );
4 BuildCADModel( 'add_body', m );
5 % Rotating bodies
6 %-----
7 axis = [1 2 3];
8 for k = 1:3
9     m = CreateBody( 'make', 'name', ['Boom_' num2str(k)], 'bHinge', struct( 'b', eye(3), 'axis',
    axis(k) ),
```

```

10 'previousBody', 1, 'rHinge', [0;0;0] );
11 BuildCADModel('add_body', m );
12 end

```

The first input to CreateBody is the action 'make'. The remainder are parameter/value pairs. You can get a list of parameters by typing: CreateBody('parameters')

```

>> CreateBody('parameters')
ans =
    'name'      'rHinge'      'bHinge'      'previousBody'  'mass'      'inertia'      '
      cM'

```

The rHinge parameter defines the origin of the body relative to the origin of the spacecraft or to the previous body in the chain. bHinge defines the rotation to the previous body. All components added to the body will be rotated by bHinge as well so that the body rotates as a single assembly. bHinge is a data structure defines the hinge. The fields are

```

bHinge.b      (3,3) Transformation matrix
bHinge.q      (4,1) Quaternion
bHinge.angle  (1,1) Angle of rotation (radians)
bHinge.axis   (1,1) Axis of rotation 1=x, 2=y, 3=z (default)

```

You can define a rotation using the transformation matrix and either the quaternion or angle. If you choose the angle it defines a single axis rotation about the axis defined by the axis field. For a two angle hinge you need to use the b field. See the BHinge function which uses this structure to define the rotation and the function Eul2Mat which is useful for generating transformation matrices.

The parameter previousBody points to the previous body in the chain. In this case it is the core. You must define bodies before defining components since each component needs to be assigned a body when it is created.

## 6.2.6 Creating Components

The function CreateComponent works in a similar fashion to CreateBody. All parameters needed for disturbance analysis, such as optical properties, are defined at the component level in addition to the surface geometry. There are two supporting functions which help set up the many parameters, GenericProperties and DeviceProperties. Defaults are available for nearly all parameters and there are many types of components to choose from.

Recall the camera component from BuildSimpleSat,

```

m = CreateComponent('make', 'camera', 'model', 'ct633', 'name', 'Camera', ...
                  'rA', [0.4;0.4;0.4], 'body', 1, 'unitVector', [1;0;0], ...
                  'faceColor', 'aluminum' );
BuildCADModel('add_component', m );

```

The first input is the action (make), the second is the component type (camera). The remainder are parameter/value pairs. All such inputs are optional and many depend on the type of component. You can add components in any order. CreateComponent is a complicated function but there are some function calls to help. For example, CreateComponent can output a list of all available types, (it's a long list!)

```

>> CreateComponent('type')
-----
Components
-----
    'sphere'
    'box'
    'cylinder'
    'cubesat'

```

```
'hollow_cylinder'  
'hollow_box'  
'hollow_sphere'  
'antenna'  
'ellipsoid'  
'surface_of_revolution'  
'triangle'  
'empty'  
'generic'  
'angle_of_attack_sensor'  
'battery'  
'camera'  
'current_sensor'  
'earth_sensor'  
'f16_gas_turbine'  
'fuel_tank'  
'gps_receiver'  
'ground_link_antenna'  
'gun'  
'hall_thruster'  
'heater'  
'hydrazine_tank'  
'hydrazine_thruster'  
'imu'  
'isl'  
'lem'  
'magnet'  
'magnetic_torquer'  
'magnetometer'  
'nuclear_reactor'  
'omni'  
'onoff_thruster'  
'pcu'  
'position_sensor'  
'power_relay'  
'radar'  
'radiator'  
'rate_gyro'  
'rea'  
'reaction_wheel'  
'relative_position_sensor'  
'rocket_engine'  
'sail'  
'shunt'  
'single_axis_drive'  
'single_axis_linear_drive'  
'single_axis_stepper_drive'  
'solar_array'  
'solar_array_back'  
'solar_array_front'  
'solar_panel'  
'star_camera'  
'star_tracker'  
'state_sensor'  
'temperature_sensor'  
'two_axis_sun_sensor'  
'wheel'
```

The geometric primitives are listed first, followed by spacecraft part types. A generic component must be defined

using a vertex and face list, but the other components require only a few parameters (such as radius for a sphere) and the shape will be defined for you. All component shapes result in a set of triangles.

There is a set of generic inputs that apply to every component, including the name, position vectors and rotation matrix. To see this list type

```
>> CreateComponent( 'inputs' )
-----
Additional generic inputs
-----
      name      Name of component
      rB        Displacement of component before rotation
      b         Rotation transformation matrix
      rA        Displacement of component after rotation
      body      ID of body to which component is attached
      inside    If a body is inside the model it is not used in
                disturbance calculations
      dataFileName  Data file name, passed to BuildCADModel
      model       Model string
      manufacturer name of manufacturer
```

These are the critical parameters where you place your component in particular location, rotate it as needed, and assign it to a body. The matrix is the rotation required to transform from the component frame to the body frame. Note that  $r_A$  is the displacement of the rotated component, i.e. *after* rotation to the body frame, and  $r_B$  is a displacement *before* the rotation, so  $r_A$  is defined in the body frame and  $r_B$  is defined in the component frame. The center of mass of the component is also in the component frame. Recall from the section on geometry (6.2.2) that

$$r_{CM|body} = r_A + b * (r_B + r_{CM|comp})$$

The default location ( $r_A$  and  $r_B$ ) is simply zero and the body is 1. If you forget to name your component it will be called 'no name'. All components are inside by default.

The only parameters that you truly need to enter when you make a component are the small set of properties needed to define its geometry.

You can find out what parameters are relevant to a particular component by typing, for example,

```
>> CreateComponent('parameters', 'imu')
-----
Parameters for imu
-----
'rUpper'
'rLower'
'h'
'n'
'thermal. absorptivity'
'thermal. emissivity'
'thermal. heatGeneratedOn'
'thermal. heatGeneratedStandby'
'thermal. temperature'
'thermal. surfaceArea'
'thermal. uNormal'
'thermal. thermalMass'
'mass. mass'
'mass. cM'
'mass. inertia'
'power. powerStandby'
'power. powerOn'
'power. electricalConversionEfficiency'
```



```

'power.powerHeater'
'power.powerIsOn'
'optical.sigmaT'
'optical.sigmaA'
'optical.sigmaD'
'optical.sigmaS'
'infrared.sigmaRT'
'infrared.sigmaRA'
'infrared.sigmaRD'
'infrared.sigmaRS'
'aero.cD'
'rf.flux'
'rf.u'
'rf.r'
'magnetic.dipole'
'propulsion.thrust'
'propulsion.iSP'
'propulsion.feedPressure'
'propulsion.eta'
'graphics.faceColor'
'graphics.edgeColor'
'graphics.diffuseStrength'
'graphics.specularStrength'
'graphics.specularColorReflectance'
'graphics.specularExponent'
'deviceInfo.scale'
'deviceInfo.bias'
'deviceInfo.randomWalk'
'deviceInfo.angleRandomWalk'
'deviceInfo.outputNoiseSigma'
'deviceInfo.beta'
'deviceInfo.rateLimit'
'deviceInfo.scaleFactor'
'deviceInfo.lsb'
'deviceInfo.countLimit'

```

The top properties listed are specific to the geometry of the IMU. The structures following are sets of properties needed for disturbance, power, and thermal analysis, for which defaults are obtained from `GenericProperties` as alluded to in Section 6.2.3. The graphics properties are used only for displaying the component in MATLAB. The names are mostly self-explanatory except for the optical and infrared coefficients; the T, A, D, and S stand for transmissivity, absorptivity, diffuse and specular reflectivity, respectively. These coefficients must sum to one. The aerodynamics parameter `cD` is the coefficient of drag. To set any of these properties, you need only enter the field name, i.e. `absorptivity`, not `thermal.absorptivity`. If you do not enter properties, generic properties are used instead.

The properties relevant to the IMU (inertial measurement unit) device are the `deviceInfo.xxx` properties. These properties are defined in the function `DeviceProperties`. Look in this function to see how the needed geometric parameters are used, for example if we search for 'imu' we will find that the geometry is defined using `CylinderModel`.

```

case 'imu'
    p = struct('scale',1,'bias',0,'randomWalk',0,'angleRandomWalk',0,...
              'outputNoiseSigma',0,'beta',0,'rateLimit',0,'scaleFactor',1,...
              'lsb',1e-7,'countLimit',16777215);
    g = {'rUpper', 'rLower', 'h', 'n'};
    if geom
        if ~exist('n','var')
            n = 12;
        end
    end

```

```
m = CylinderModel( rUpper, rLower, h, n, m, computeInertia );
end
```

For more information on the properties `rUpper`, `rLower`, `h`, and `n`, you would look in `CylinderModel`, which is a subfunction of `DeviceProperties`. The geometric function called in turn is `Frustrum`, which has a built-in demo.

`CreateComponent` will accept the list of parameters, compute defaults as needed, and outputs the component structure for passing to `BuildCADModel`. All components end up with a list of vertices and faces defining the geometry. `BuildCADModel` will later compute additional properties of the faces such as their area, normals, and centroids.

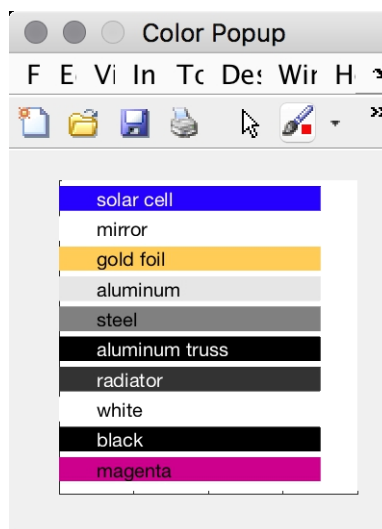
Perhaps the shortest component function call outside of the demo is this:

```
>> m = CreateComponent('make','sphere','radius',5)
m =
    class: 'sphere'
    thermal: [1x1 struct]
    mass: [1x1 struct]
    power: [1x1 struct]
    optical: [1x1 struct]
    infrared: [1x1 struct]
    aero: [1x1 struct]
    rF: [1x1 struct]
    magnetic: [1x1 struct]
    propulsion: [1x1 struct]
    graphics: [1x1 struct]
    inside: 1
    model: 'generic'
    manufacturer: 'none'
    deviceInfo: {}
    v: [5x3 double]
    f: [6x3 double]
```

Here you can see that the sphere has been defined by five vertices and six faces.

You can determine what predefined colors are available by typing `CreateComponent('colors')`. You will see the window in [Figure 6.5 on page 54](#).

**Figure 6.5:** CreateComponent color display



## 6.2.7 Subsystems

You can organize your components into subsystems. The following shows example subsystem commands.

```
1 BuildCADModel( 'add_subsystem', 'propulsion', {'ppt', 'thruster'} );
2 BuildCADModel( 'add_subsystem', 'mechanism', {'drive'} );
```

BuildCADModel searches the component names for the strings in the cell array and associates them with the subsystem. If the string doesn't exist it is ignored. Subsystems are used in the table form of the CAD model and in some of the visualization capabilities.

## 6.2.8 Panels

You can organize your components by panels. This is purely for organizational purposes and does not affect any of the properties of the model.

```
1 BuildCADModel( 'add_panel', 'north', {'ppt', 'thruster'} );
2 BuildCADModel( 'add_panel', 'south', {'drive'} );
```

BuildCADModel searches for the strings in the cell array and associates them with the panel. If the string doesn't exist it is ignored. The result is a list of components stored with the panel name.

## 6.2.9 Viewing the CAD Model

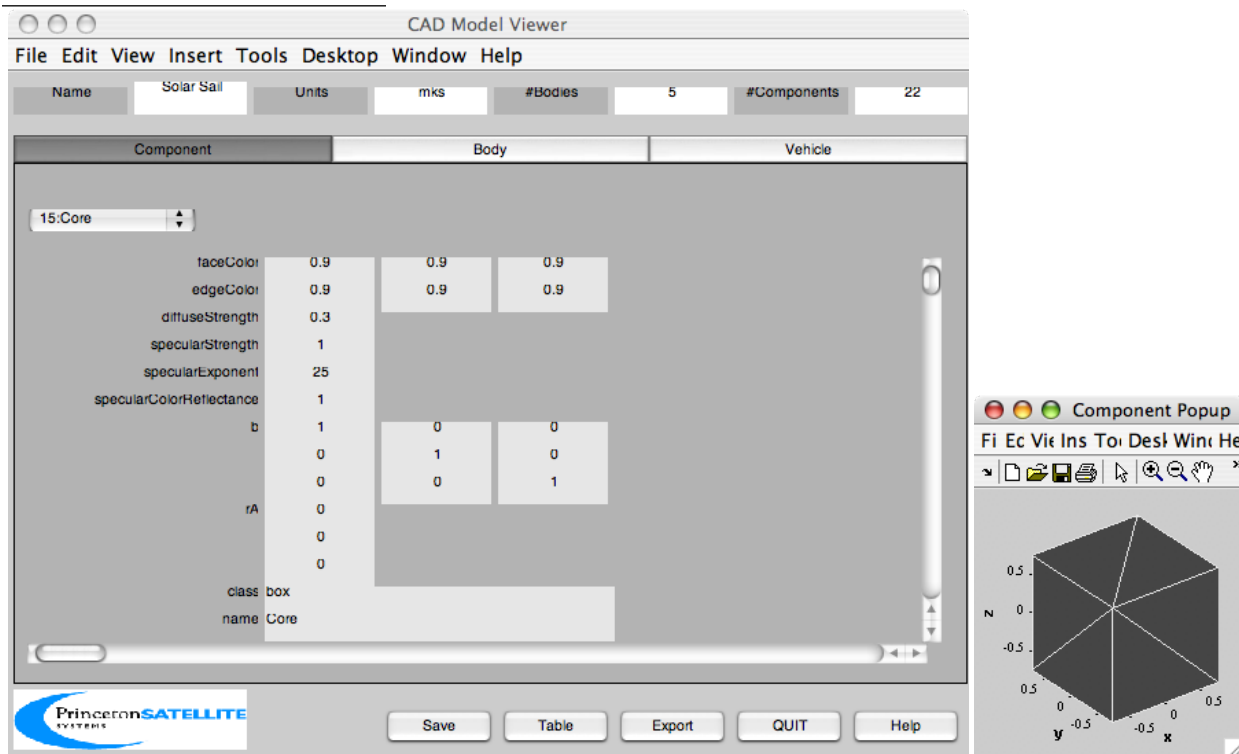
When your CAD script is done, execute the script. The GUI shown in Figure 6.6 on page 56 will appear as in BuildSimpleSat in the beginning of this chapter. For this section we will view the results of BuildSolarSail.m, which builds a solar sail. This demo demonstrates both generic components like the sails, geometric primitive components like the core box, and specific components with a model name like the reaction wheels. The demo also has two subsystems, the bus and the sail.

The GUI has three tab buttons, Component, Body and Vehicle for three different views of the CAD model. When you run a script the Component view is activated. The row above the tabs gives general information about the CAD model. The buttons at the bottom, outside of the tab views, are: Save, for saving the model to a mat-file; Table creates the subsystem table; Export, for exporting the model to a DSIM/Satellite Simulator compatible format; Quit and Help for online help. The scroll bars allow you to scroll through data, which in this view, is individual component data. You cannot modify any data through the GUI. If you select a component from the popup menu, you will also get a 3D view of that component alone. The figure to the right shows the Core component.

The subsystem table is shown below. It is stored in a text file using the name of the CAD model, in this case, SolarSail.mat.

Subsystem	Component	Mass (kg)	Power (W)	Heater	Power (W)	Manufacturer
Model						
Sail Subsystem						
	Panel +X/+Y	2.00	0.00		0.00	none generic
	Panel +X/-Y	2.00	0.00		0.00	none generic
	Panel -X/+Y	2.00	0.00		0.00	none generic
	Panel -X/-Y	2.00	0.00		0.00	none generic
	Drive +X/+Y	0.50	0.00		0.00	none generic
	Drive +X/-Y	0.50	0.00		0.00	none generic
	Drive -X/+Y	0.50	0.00		0.00	none generic
	Drive -X/-Y	0.50	0.00		0.00	none generic
	Sail +X	20.00	0.00	0.00	none	generic
	Sail -X	20.00	0.00	0.00	none	generic
	Sail +Y	20.00	0.00	0.00	none	generic

Figure 6.6: BuildCADModel Component View



Sail	-Y	20.00	0.00	0.00	none	generic
Bus Subsystem						
RWA	X	14.30	0.00	0.00	none	hr60
RWA	Y	14.30	0.00	0.00	none	hr60
RWA	Z	14.30	0.00	0.00	none	hr60
South Array		0.03		0.00	0.00	none generic
North Array		0.03		0.00	0.00	none generic
Core		10.00	0.00	0.00		none generic
Radiator	+Y	2.00		0.00	0.00	none generic
Radiator	-Y	2.00		0.00	0.00	none generic
Chassis		3.00	0.00		0.00	none generic
Battery		12.00	0.00		0.00	none generic
Total		161.95		0.00	0.00	

If you push the Body tab you get the body view with body level data, shown in Figure 6.7 on page 57. The list of components associated with the body is given in a popup menu on the left. In the Vehicle view, if you push the Show Vehicle button you get a 3D window with the spacecraft and a list of components organized by body, see Figure 6.8 on the facing page and Figure 6.9 on page 58.

The radio buttons in the Vehicle tab select the mode. If All is selected all components are shown. If Add is selected, no components are shown and you add them by clicking on the component in the list. If Subtract is selected all components are shown and you remove components by clicking on the component in the list. Within the 3D view, you can highlight certain subsystems. The buttons on the left of the window toggle each subsystem's, or the whole spacecraft's, transparency. See Figure 6.10 on page 58, where the sail subsystem has been made transparent. You can also adjust the Open Model slider to get an "exploded" view of the spacecraft as shown on the right, where the sail components have been subtracted and the remaining bus components have been spaced apart.

The Show Spacecraft in Orbit button shows the spacecraft in orbit around the earth. You need to specify an orbit position and quaternion for this feature, as is done in the following lines:

Figure 6.7: BuildCADModel Body view

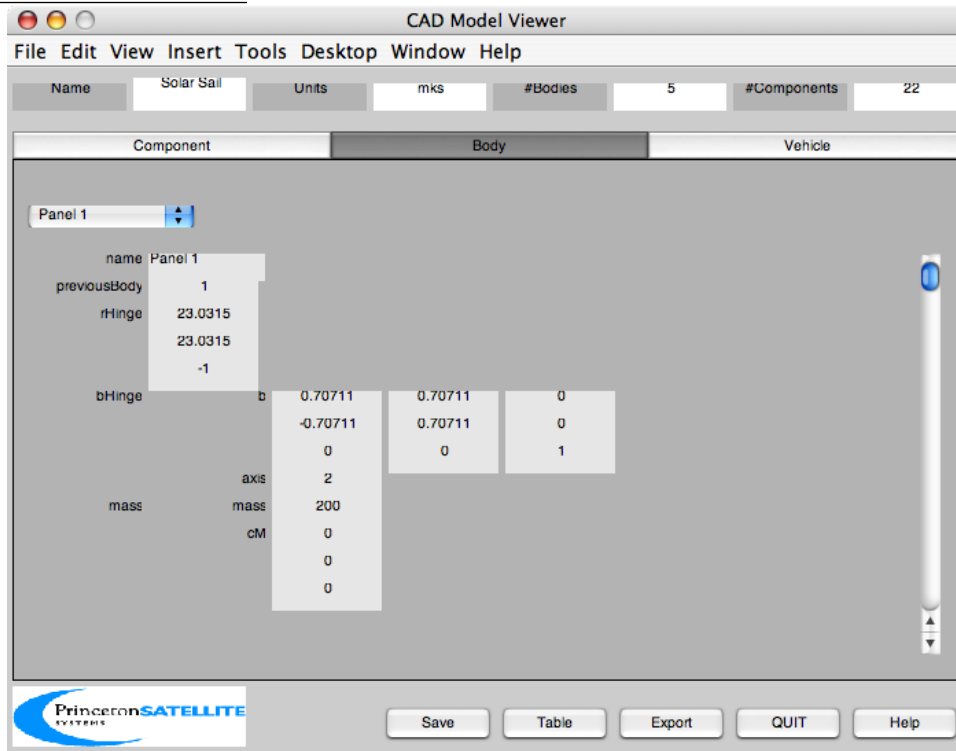


Figure 6.8: Vehicle tab with Show Vehicle pushed

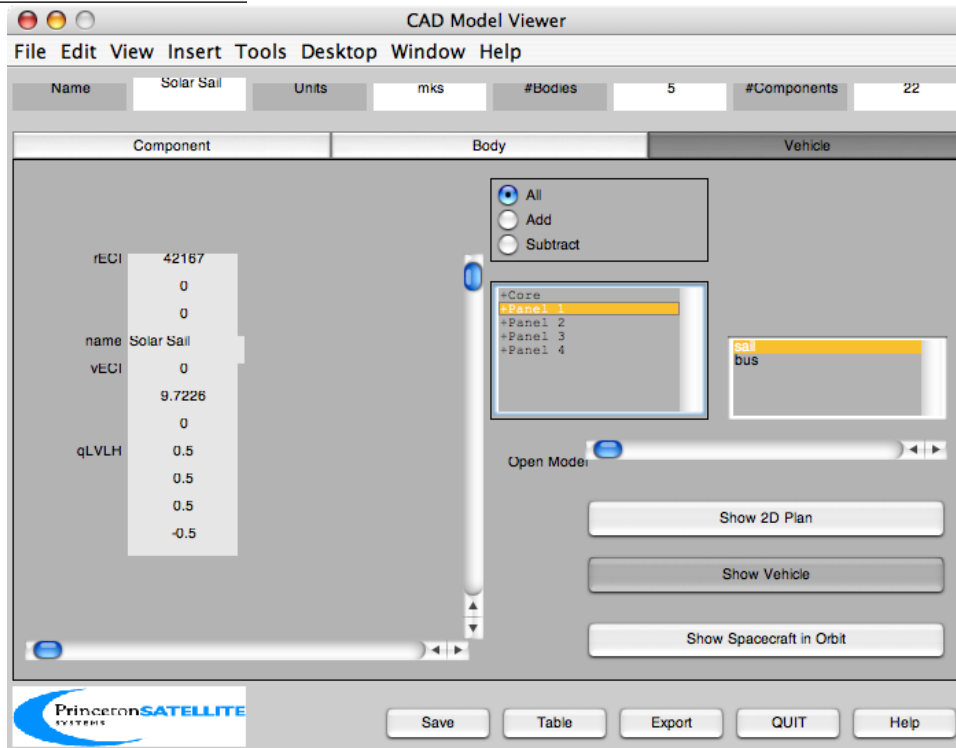


Figure 6.9: 3D View obtained by pushing Show Vehicle, left, and another sail model shown using Show Spacecraft in Orbit, right

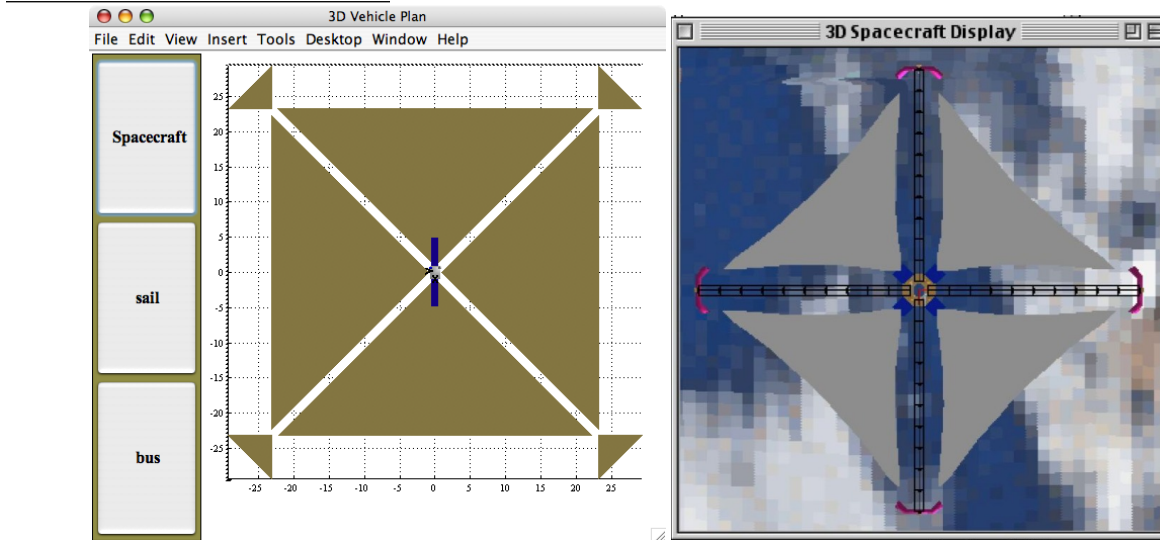
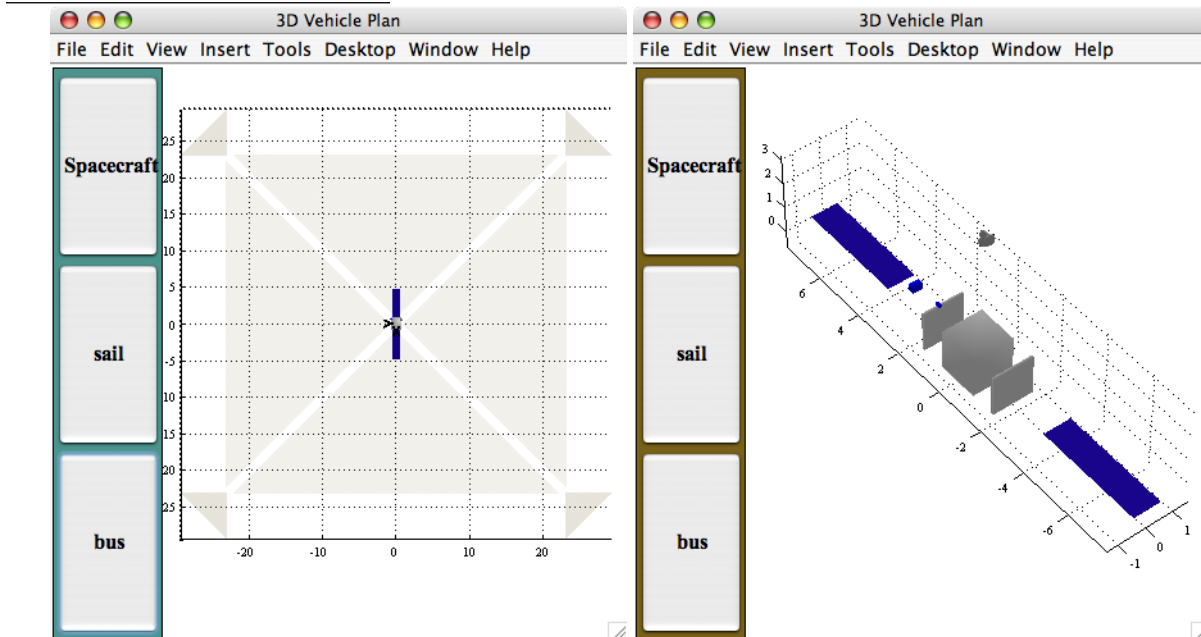


Figure 6.10: Subsystem Transparency and Exploded views



```

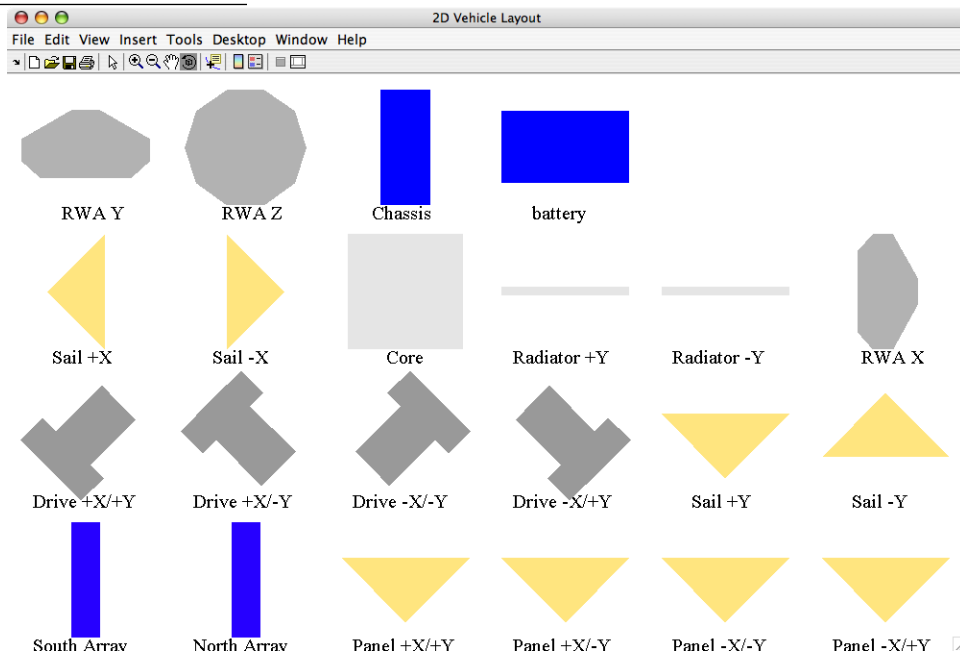
1 BuildCADModel( 'set_rECI' ,      rECI  );
2 BuildCADModel( 'set_vECI' ,      vECI  );
3 BuildCADModel( 'set_qLVLH' ,     qLVLH  );
4 BuildCADModel( 'set_qECIToBody', q     );
5 BuildCADModel( 'set_omega' ,     omega  );

```

An example view of another sail model in orbit is shown on the right in Figure 6.9 on page 58. The quaternions set here are used only for display purposes.

The Show 2D Plan button shows a 2D picture with 2D views of all of the components. The components are scaled to fit inside the same size rectangle.

**Figure 6.11:** 2D View from clicking Show 2D Plan



### 6.2.10 Importing Models

Another model can be imported if it has previously been saved as a .mat file. This uses the 'add\_subassembly' action of BuildCADModel.

```

1 BuildCADModel( 'add_subassembly', 'Bus.mat', gimbaledBoomBody, r, eye(3) )

```

The second argument is the mat file, the third is the body to which the subassembly is to be attached, the fourth is the location in that body and the fifth rotates the subassembly (prior to moving to the specified location.)

You can also import geometry information from other model types, such as dxf files, and use it to create a CAD component. The generic component type allows you to pass in the face and vertex data directly. See LoadCAD. A number of dxf files are included in the SCDData folder as examples, including models of Hubble, Cassini, and the Huygens probe. Some are more fanciful, for instance,

```

>> LoadCAD( 'TFIGHTER.dxf' )

```

draws a picture of a tie fighter, shown in Figure 6.12 on the following page. If you call the function with the output, you will get the following data structure:

```

>> gFighter = LoadCAD('TFIGHTER.dxf')

gFighter =

    name: 'TFIGHTER.dxf'
 component: [1x1 struct]
   radius: 292.3833

>> gFighter.component

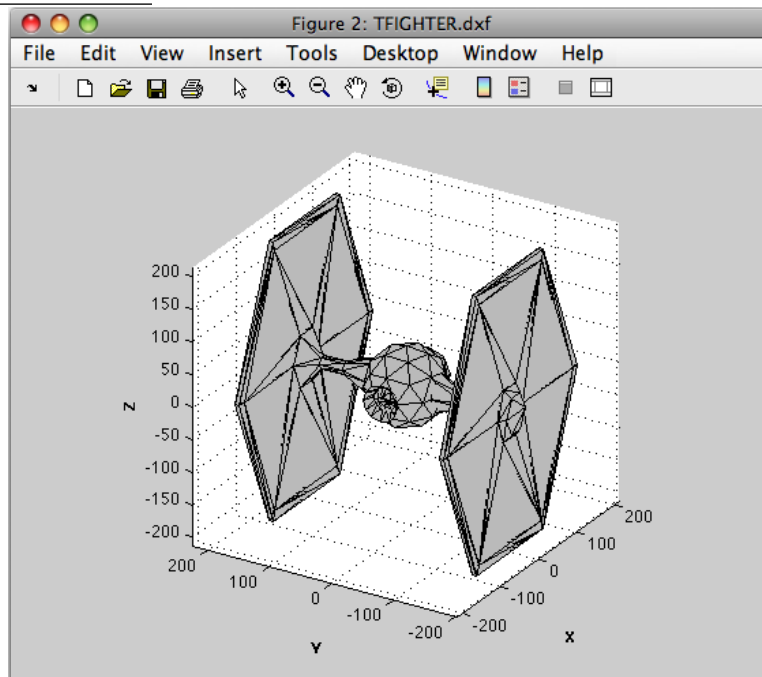
ans =

    name: '0'
 faceColor: [0.7320 0.7320 0.7320]
diffuseStrength: 0.3000
specularStrength: 0.3000
  ambientStrength: 1
specularExponent: 10
specularColorReflectance: 1
        inside: 0
        mass: [1x1 struct]
           v: [2464x3 double]
           f: [616x3 double]
           n: [616x1 double]

```

where you can see that there are 2464 vertices and 616 faces in the model. The Strength, Exponent, and Reflectance fields are for drawing in MATLAB only; they do not represent true optical properties are not used in disturbance modeling.

**Figure 6.12:** Tie fighter as loaded from a dxf file





### 6.2.11 Exporting

Export a model from the GUI by pushing Export. Save the file (\*.cad) to your selected location as prompted by the popup window. This is a text description of your model in which all the vertices and faces which were computed by `CreateComponent` are listed in full. For example, one of the sails from the `BuildSolarSail` results in this text in the .cad file:

```
componentname Sail +Y
componenttype generic
numberofvertices 3
numberoffaces 1
color [ 1.0000 0.9000 0.5000]
magneticdipole [ 0.0000 0.0000 0.0000]
sigmat 0.0000
sigmaa 0.0500
sigmad 0.5500
sigmas 0.4000
absorptivity 0.1000
emissivity 0.8000
heatgeneratedon 0.0000
heatgeneratedstandby 0.0000
poweron 0.0000
powerstandby 0.0000
electricalconversionefficiency 0.0000
dragcoefficient 2.7000
vertex 0.0000 0.8000 -0.7500
vertex 22.3607 23.1607 -0.7500
vertex -22.3607 23.1607 -0.7500
face 1 2 3
```

The sail is a triangle, so there are three vertices and a single face. The export can also be performed in your script using the function `ExportCAD`.

Another available function is `ExportOBJ` (professional edition of the toolbox only). This function will create 3 files:

```
myFile.obj
myFile.mtl
myFile.cad
```

The first two are Wavefront OBJ formatted files. The first contains the geometry and the second the material properties (color). The last file contains the component non-graphic parameters including mass, inertia and specific component properties. It also contains the connection information between the bodies. The first two files are compatible with any graphics package that imports OBJ formatted files. The last is used by Princeton Satellite Systems `VisualCommander`.

`ExportOBJ` requires the CAD data to be passed in as a data structure. This structure can be obtained at the end of a CAD script using the line

```
g = BuildCADModel( 'get_cad_model' );
```

You can save the data structure as a .mat file by pushing the `Save` button. The equivalent function call for a script is

```
SaveStructure( g, 'MyCADFileName' );
```

The structure can then be retrieved simply by loading the mat-file into a variable, i.e.

```
g = load('MyCADFileName.mat');
```

## 6.3 Visualizing your CAD Model

Once you have created a CAD model you will want to use it in other scripts. There are several functions for visualizing models:

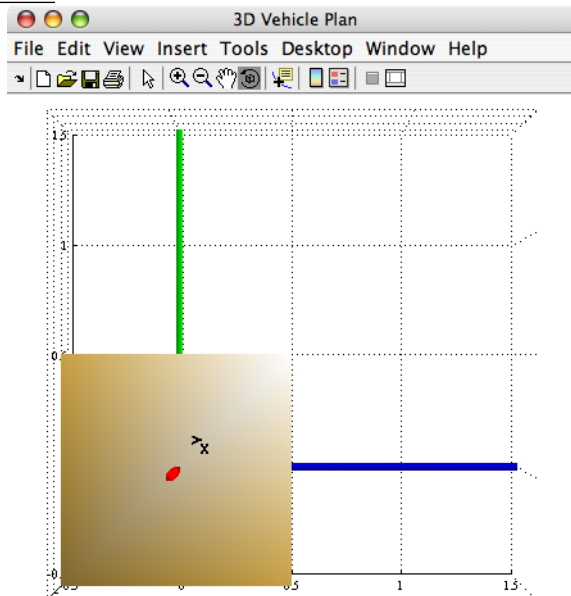
- ShowCAD
- DrawCAD
- DrawSC
- DrawSCPlanPlugIn
- DrawSCPlugIn

Each requires the CAD model to be loaded as a data structure as explained in Section 6.2.11. DrawSCPlanPlugIn draws a spacecraft in a plain 3D box. The simplest way to use it to view a model is

```
g = load('XYZSat.mat');
tag = DrawSCPlanPlugIn( 'initialize', g );
```

See the figure in Figure 6.13 on page 62. This is the same view that can be obtained using the Show Vehicle button in the CAD model GUI.

**Figure 6.13:** DrawSCPlanPlugIn showing the XYZSat model



You can create a figure like this inside a script, then update it iteratively in a loop using the call

```
DrawSCPlanPlugIn( 'update', tag, g );
```

where we are using the `tag` which was returned from the initialization call. There are additional actions for this function explained in its header. For example, you might be rotating the solar arrays of a spacecraft and you want to visualize this. You have to update the fields of `g` as needed between calls to the plug in and the new locations or rotations will be drawn. You can change any value of `g`, for instance we can change the color of the core box. First, examine the `g` structure.

```
g =
    name: 'Cube_Spacecraft'
```

```

units: 'mks'
  rECI: [3x1 double]
  vECI: [3x1 double]
  qLVLH: [4x1 double]
  q: [4x1 double]
  omega: [3x1 double]
  mass: [1x1 struct]
  body: [1x1 struct]
component: [1x4 struct]
radius: 1.500149992500750e+00

```

To see the names of the components in a neat list, type

```

>> {g.component.name}
ans =
  'Core'      'X_Axis'    'Y_Axis'    'Z_Axis'

```

The core is the first component. To view its fields, type

```

>> g.component(1)
ans =
    faceColor: [1 8.000000000000000e-01 3.400000000000000e-01]
    edgeColor: [1 8.000000000000000e-01 3.400000000000000e-01]
    diffuseStrength: 3.000000000000000e-01
    specularStrength: 1
    specularExponent: 25
    specularColorReflectance: 1
        b: [3x3 double]
        rA: [3x1 double]
        v: [8x3 double]
        f: [12x3 double]
        a: [12x1 double]
        n: [12x3 double]
        r: [12x3 double]
    radius: [12x1 double]
    deviceInfo: {}
        class: 'box'
        name: 'Core'
    optical: [1x1 struct]
    infrared: [1x1 struct]
    thermal: [1x1 struct]
    power: [1x1 struct]
    aero: [1x1 struct]
    magnetic: [1x1 struct]
    mass: [1x1 struct]
    inside: 0
    rF: [1x1 struct]
    body: 1
    manufacturer: 'none'
    model: 'generic'

```

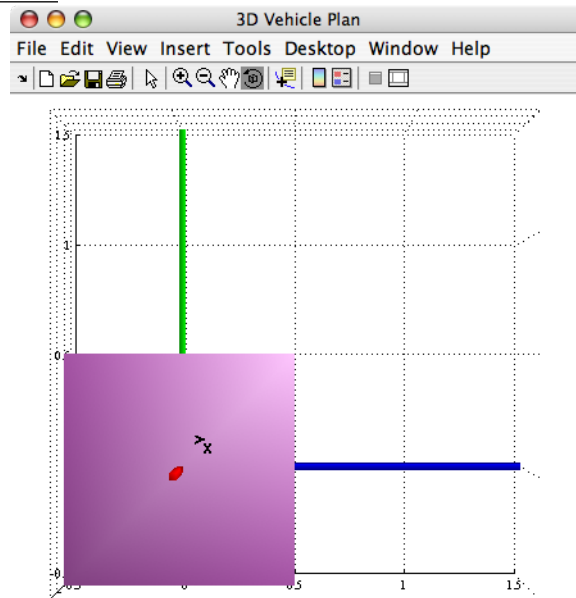
We will rewrite the `faceColor` field. Set the field to any RGB triple, then update the plug in. For example,

```

>> g.component(1).faceColor = [1 0.5 1];
>> DrawSCPlanPlugIn( 'update', tag, g )

```

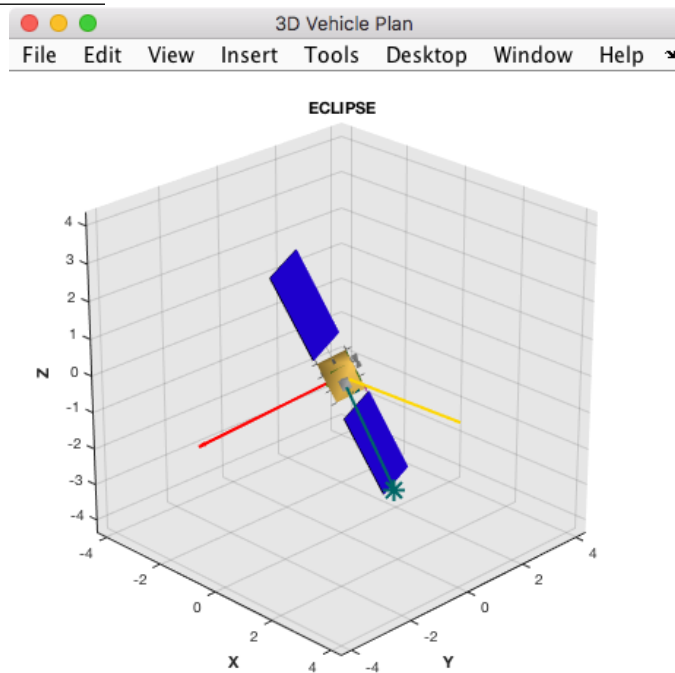
The core is now lavender, as shown in Figure 6.14 on page 64.

**Figure 6.14:** DrawSCPlanPlugIn showing the XYZSat model after a change to  $g$ 

DrawSCPlanPlugIn can display vectors, such as sun and nadir vectors, to aid in visualization. For an example this see `DisturbanceBatchDemo`. The code used to update the vectors is

```
>> DrawSCPlanPlugIn( 'vectors', tag, g, uSun );
```

where the `rECI` and `vECI` fields are used for the position and velocity vectors and a sun vector is passed in. When the satellite is in eclipse the entire axes are shaded gray.

**Figure 6.15:** DrawSCPlanPlugIn showing a satellite with vectors. The green vector is nadir, the yellow vector the sun, and the red vector the orbital velocity.

There is a similar function for viewing your spacecraft in orbit, `DrawSCPlugIn`. This is also the same function called from the CAD GUI for the button `Show Spacecraft in Orbit`. This function has initialization and update calls as well. A good example is `ShuttleSim`, which simulates the opening of the bay doors as the shuttle moves in its orbit. Each door is stored in the model as a separate body. The code is quite compact so we include it here. Note that the Earth is passed in explicitly so that you can view a spacecraft in reference to any planet or moon for which a texture is available. Note the use of the functions `SetCADQuaternion`, `SetCADState`, and `SetCADRotation` to interact with the CAD model struct.

**Listing 6.2:** Shuttle simulation visualizing the opening of the bay doors during an orbit simulation `ShuttleSim.m`

```

1  %% Simulate the space shuttle model.
2  % We rotate the shuttle bay doors and the attached robot arm using the body
3  % hinge structure of the CAD model. The shuttle model is loaded from a mat-file.
4  % This script shows how to interact with DrawSCPlugIn in a simulation loop.
5  % -----
6  % See also DrawSCPlugIn, QLVLH, QPose, StopGUI, RK4,
7  % JD2000, E12RV
8  % -----
9  %%
10 %-----
11 % Copyright 1999 Princeton Satellite Systems, Inc. All rights reserved.
12 %-----
13
14 %% Constants
15 %-----
16 degToRad = pi/180;
17
18 %% Global for the time GUI
19 %-----
20 global simulationAction
21 simulationAction = '┐';
22
23 g = load('ShuttleModel');
24
25 %% Initialize the orbits
26 %-----
27 [r1,v1] = E12RV( [7000 61*degToRad 0 0 0 0] );
28 x1      = [r1;v1];
29 t       = 0;
30 jD      = JD2000;
31
32 %% Initialize the 3D window
33 %-----
34 g = SetCADQuaternion( g, QLVLH( r1, v1 ) );
35 g = SetCADState( g, r1, v1 );
36 g.name      = 'Space_Shuttle_Orbiter';
37 tag3DWindow = DrawSCPlugIn( 'initialize', g, [], [], 'Earth', jD );
38
39 dTSim          = 1;
40
41 %% Generate the two orbits using numerical integration
42 %-----
43 doorAngle = 0;
44 rMSAngle  = 0;
45 StopGUI( 'Space_Shuttle' );
46
47 rMSAxis = [3 2 2 2 2];
48
49 while t<120
50
51 % Transformation matrices
52 %-----
53 g = SetCADQuaternion( g, QLVLH( x1(1:3), x1(4:6) ) );
54 g = SetCADState( g, x1(1:3), x1(4:6) );
55 if( doorAngle > -139 )
56     doorAngle = doorAngle - 1;
57 elseif( rMSAngle < 45 )

```

```
58     rMSAngle = rMSAngle + 1;
59 end
60 g = SetCADRotation( g, 'body', 2, 'axis', 1, 'angle', doorAngle*pi/180 );
61 g = SetCADRotation( g, 'body', 3, 'axis', 1, 'angle', -doorAngle*pi/180 );
62 for j = 1:5
63     g = SetCADRotation( g, 'body', j+3, 'axis', rMSAxis(j), 'angle', -rMSAngle*pi/180 );
64 end
65
66 DrawSCPlugIn( 'update_spacecraft', tag3DWindow, g, jD );
67
68 % Propagate the orbits
69 %-----
70 x1 = RK4( @FOrbCart, x1, dTSim, t, [0;0;0] );
71
72 % Update the time
73 %-----
74 t = t + dTSim;
75 jD = jD + dTSim/86400;
76
77 % Time control
78 %-----
79 switch simulationAction
80     case 'pause'
81         pause
82         simulationAction = '_';
83     case 'stop'
84         return;
85     case 'plot'
86         break;
```

---

*ShuttleSim.m*

# ATTITUDE SIMULATIONS

Several different functions are included in the toolbox for simulating spacecraft attitude dynamics. They range from simple rigid body models to sophisticated multi-body models. This chapter provides an overview of the available models and looks in depth at two of the included demos.

## 7.1 Dynamics Models

The toolbox has numerous dynamical models for spacecraft. These include attitude and orbit dynamics as well as component dynamics. Attitude dynamics range from rigid bodies to multiple bodies in a topological tree. A simple rigid body attitude simulation can be embodied in the function,

```
xDot = ASim( x, t, inertia, torque );
```

which contains the following code,

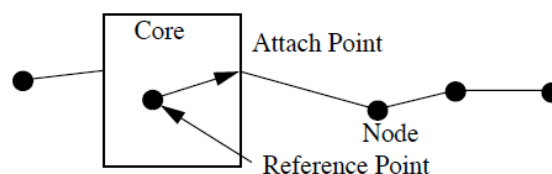
```
xDot = [QIToBDot( x(1:4), x(5:7)); RBModel( x(5:7), inertia, torque)];
```

The first four elements of  $x$  form the quaternion from the inertial frame to the body frame. The last three elements of  $x$  are the inertial body rates measured in the body frame.  $t$  is not used in `ASim`. This function is used in [Example 7.1 on page 68](#). Initializing `xPlot` in the script speeds up the script considerably. Notice the use of brackets in  $x$  to make the script easier to read.

## 7.2 Wire Model

The wire model simulates a spacecraft with deployable wires as is illustrated in [Figure 7.1](#). Two models are included.

**Figure 7.1:** Wire model

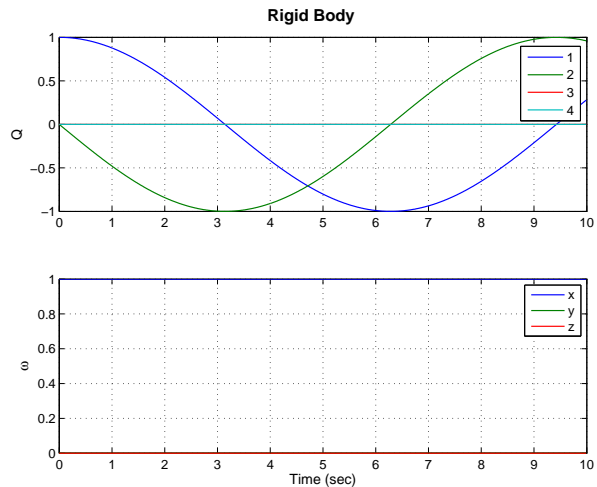


**Example 7.1** Rigid Body Attitude Simulation

```

1 dT      = 0.01;
2 nSimSteps = 1000;
3 inertia  = [3000,0,0;0,1000,0;0,0,1000];
4 torque   = [0;0;0];
5 xPlot    = zeros(7,nSimSteps);
6 tPlot    = zeros(1,nSimSteps);
7 x        = [ [1;0;0;0]; [1;0;0] ];
8 t        = 0;
9 for k = 1:nSimSteps,
10  xPlot(:,k) = x;
11  tPlot(k)   = t;
12  zIO(1:3,4) = [0;0;0];
13  x          = RK4('ASim',x,dT,t, inertia,
14                torque);
14  t          = t + dT;
15 end
16 Plot2D(tPlot,xPlot,'Time_(sec)', {'Q'; '\omega'
17   },...
18   'Rigid_Body','lin',{[1:4],[5:7]});
19 legend('x','y','z')
20 subplot(2,1,1)
21 legend('1','2','3','4')

```



WireFRB includes the extensional dynamics, modeling the stretching of the wire. The second, WireC, applies kinematic constraints to control the length between wire nodes. The former can introduce very high frequency dynamics into the simulation but is useful if you expect the wire to stretch. The latter is useful for most scientific satellites with copper or aluminum wires. Either model can simulate deployment and both account for system center-of-mass motion. A spacecraft can have any number of wires.

The demo script WireSimG, listed below, illustrates the use of the model.

**Listing 7.1:** Wire Simulation: Header

WireSimG.m

```

%-----
% This script demonstrates the deployment of the wire from the
% spacecraft. This model assumes that the center-of-mass of the
% spacecraft does not move as the wires deploy. The simulation
% models the wire as a string of masses connected by springs.
% Orbit dynamics and gravity gradient are included.
%-----
% Copyright 1997 Princeton Satellite Systems, Inc. All rights reserved.
%-----
% Clean up the workspace
%-----
clear all
% Global for the time GUI
%-----
global simulationAction
simulationAction = '_';

% Constants
%-----
false = 0;
true  = 1;

```

WireSimG.m

These parameters manage the script. kConst determines whether the constrained model is to be used.

**Listing 7.2:** Wire Simulation: Initialization

WireSimG.m

```

% Simulation parameters
%-----
tSim      = 120.0;

```



```

dT      = 0.125;
nSim    = tSim/dT;
nPlot   = min([tSim/dT 200]);
nPMax   = floor(nSim/nPlot);
nPlot   = floor(nSim/nPMax);
gGOn    = false;
kConst  = true;

% Print the time to go message every nTTGo steps
%-----
nTTGo   = 1000;

```

WireSimG.m

The core body information follows. The model includes orbit dynamics.

**Listing 7.3:** Wire Simulation: Spacecraft properties

WireSimG.m

```

% Spacecraft properties
%-----
mass     = 800; % kg
r0       = [0;0;0];
inertia  = [104 0 0;0 107.8 0;0 0 125.4];
% Orbital elements [a i W w e M]
%-----
e1       = [7000;0;0;0;0;0];
muEarth  = 3.98600436e5;
% Initial rigid body state
%-----
[rECI, vECI] = El2RV( e1 );
omega       = [0;0;0.5]*pi/30;
q           = [1;0;0;0];
torque      = [0;0;0]; % On the central body
force       = [0;0;0]; % On the central body

```

WireSimG.m

Next, create the wires. You can have as many wires as you like. Each column represents one wire. If you are using the constrained equations you can ignore the spring and damping constants.

**Listing 7.4:** Wire Simulation: Initialization of the wire model

WireSimG.m

```

% The wire model. Each column is one wire
%-----
nNodes   = [ 3   3 ];
rWireBase = [0 0;0.6 -0.6;0 0];
lWireMax  = [0.04 0.08];
massWire  = [0.4 0.4];
kSpring   = [ 3.0  3.0]; % Used only by WireFRB
cSpring   = [ 0.5  0.5 ]; % Used only by WireFRB
cDeploy   = cSpring; % Used only by WireFRB
nodeDeploying = [ 0 0 ]; % To start undeployed set these to 3
vDeploy   = [ 0.001 0.001 ]; % m/sec
% Initialize the wire data structure
%-----
[wireDS, x] = WireInit( nNodes, mass, massWire, lWireMax, kSpring, cSpring, vDeploy, cDeploy,
    nodeDeploying, rWireBase, rECI, vECI, q, omega, r0, inertia, gGOn );

```

WireSimG.m

alpha is a constraint torque gain. mu determines the damping, omega the constraint stiffness. The number of iterations is generally less than 2.

**Listing 7.5:** Wire Simulation: Kinematic constraints

WireSimG.m

```

% If using the kinematic constraints
%-----
penalty.alpha = 1e6;
penalty.mu    = 1;

```

```
penalty.omega = 10;
penalty.nIts = 2;
```

WireSimG.m

Initialize the plotting arrays.

**Listing 7.6:** Wire Simulation: Initialization of plotting

WireSimG.m

```
% Plotting arrays
%-----
xPlot = zeros(length(x), nPlot);
hPlot = zeros(1, nPlot);
tPlot = zeros(1, nPlot);
nP     = 0;
kP     = 0;
t      = 0;
% Initialize the time display
%-----
tToGoMem.lastJD      = 0;
tToGoMem.lastStepsDone = 0;
tToGoMem.kAve        = 0;
ratioRealTime        = 0;

% Initialize the status message function
%-----
[ratioRealTime, tToGoMem] = TimeToGo( nSim, 0, tToGoMem, 0, dT );
```

WireSimG.m

The simulation loop is next. The first part shows the plotting functions.

**Listing 7.7:** Wire Simulation: Simulation loop plotting

WireSimG.m

```
for k = 1:nSim

    % Display the status message
    %-----
    [ratioRealTime, tToGoMem] = TimeGUI( nSim, k, tToGoMem, ratioRealTime, dT );
    % Plotting
    %-----
    if( nP == 0 )
        kP     = kP + 1;
        xPlot(:, kP) = x;
        hPlot(kP) = WireH( x, wireDS );
        tPlot(kP) = t;
        nP     = nPMax - 1;
    else
        nP     = nP - 1;
    end
```

WireSimG.m

This is the simulation. WireDMch simulates the deployment mechanism. It just assigns a nonzero extensional rate to the innermost wire segment.

**Listing 7.8:** Wire Simulation: Dynamics model

WireSimG.m

```
% Choose either the extensional stiffness or constrained wire models
%-----
if( kConst == true )
    x = RK4( 'WireC', x, dT, t, wireDS, muEarth, torque, force, penalty );
else
    x = RK4( 'WireFRB', x, dT, t, wireDS, muEarth, torque, force );
end
t = t + dT;
[wireDS, x] = WireDMch( wireDS, x, t );
% Time control
%-----
switch simulationAction
```

```

case 'pause'
    pause
    simulationAction = '_';
case 'stop'
    return;
case 'plot'
    break;
end
end
end

```

WireSimG.m

The following listing shows the plotting code.

### Listing 7.9: Wire Simulation: Plotting

WireSimG.m

```

% Output
%-----
dOmega = [xPlot(11,:) - xPlot(11,1);...
          xPlot(12,:) - xPlot(12,1);...
          xPlot(13,:) - xPlot(13,1)];

magH = abs(hPlot(1));
hPlot = hPlot / magH;
Plot2D( tPlot, xPlot( 1: 3,:), 'Time_(sec)', 'rECI_(km)' );
Plot2D( tPlot, xPlot( 4: 6,:), 'Time_(sec)', 'vECI_(km/sec)' );
Plot2D( tPlot, xPlot( 7:10,:), 'Time_(sec)', 'q' );
Plot2D( tPlot, dOmega, 'Time_(sec)', 'dOmega_(rad/sec)' );
Plot2D( tPlot, hPlot - 1, 'Time_(sec)', 'H/|H(0)|_1' );

WirePlot( xPlot, tPlot, wireDS );

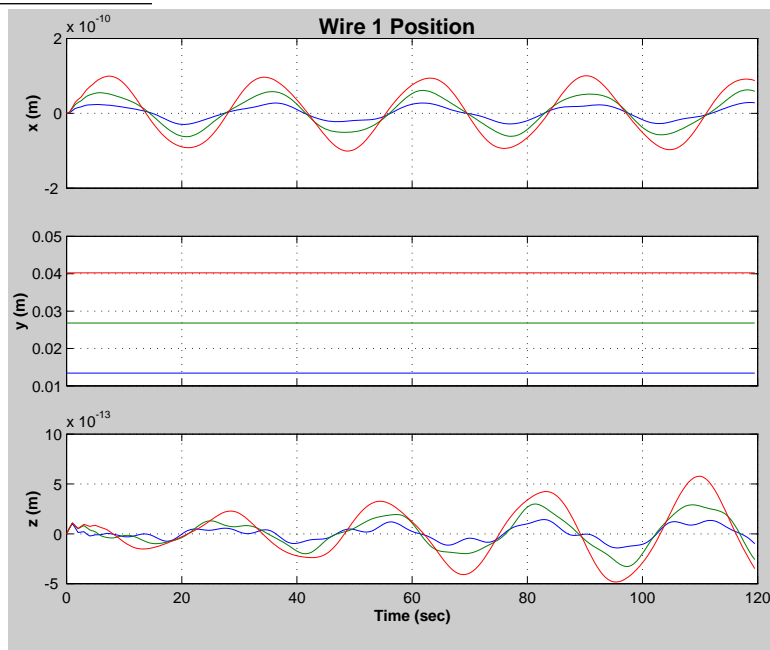
disp(sprintf('Max_momentum_change_=%12.4e_with_dT_=%8.4f_and_tSim_=%8.4f', max(abs(hPlot-1)),
            dT, tSim))

```

WireSimG.m

The results of this wire simulation are shown in Figure 7.2 on page 71.

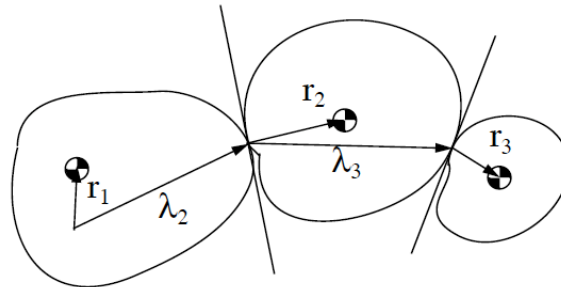
**Figure 7.2:** Wire simulation: Results



## 7.3 Tree Model

The tree model simulates multibody spacecraft arranged in a tree configuration.

**Figure 7.3:** Topological Tree



There can be no closed loops which also means that for any component exactly one hinge leads to the core. All of the sub-bodies are rigid and only rotational degrees of freedom are permitted at each hinge. In addition, the number of degrees of freedom at a hinge can be either 3 or 1. To simulate a 2-dof hinge just interpose a massless and inertialess body.

The demo script `TreeSim`, listed below, illustrates the use of the tree model, beginning with initialization.

**Listing 7.10:** Tree Simulation: Initialization

*TreeSim.m*

```

%-----
%   Three body simulation. The bodies are connected
%
%   1 - y DOF - 2 - zDOF - 3
%
%   Overall there are 5 dof.
%
%   See TreeCAD.m for a demonstration of how to use Tree with the CAD
%   functions.
%
%-----

% Clean up the workspace
%-----
close all

% Vectors from previous body reference to the hinge of the body
%-----
lambda1 = [0;0;0];
lambda2 = [2;0;0];
lambda3 = [1;0;0];
% Vector from body reference to body c.m. For all but the core
% the reference is always the hinge that leads to the core
%-----
r1      = [0;0;0];
r2      = [0;0.5;0];
r3      = [0;0;0.5];
% Mass and inertia
%-----
m1      = 100;
m2      = 2;
m3      = 5;
i1      = diag([200 200 300]);
i2      = diag([ 1  1  1]);
i3      = diag([ 3  3  1]);

```

*TreeSim.m*

TreeAdd adds bodies to the link. You can give them a name.

**Listing 7.11:** Tree Simulation: Adding bodies.

*TreeSim.m*

```
% Add each body to the tree data structure
%-----
body(1) = TreeAdd( i1, r1, lambda1, m1, 0, 0, [], [], [], [], [], 'Core' );
body(2) = TreeAdd( i2, r2, lambda2, m2, 1, 2, [], [], [], [], [], 'Link' );
body(3) = TreeAdd( i3, r3, lambda3, m3, 2, 3, [], [], [], [], [], 'Payload' );

Cause the inner link to accelerate.
Tree Simulation: A hinge acceleration.
% Internal torque
%-----
body(2).torque = 0.1;

Define a low earth orbit.
Tree Simulation: A low earth orbit.
% Initial orbit
%-----
r = [7000;0;0];
v = [0;sqrt(3.98600436e5/7000);0];
```

*TreeSim.m*

This initializes the tree data structures and the state vector. TreePrint prints out all of the information in the data structure.

Initialize the multibody tree and preallocate arrays.

**Listing 7.12:** Tree Simulation: TreeInit

*TreeSim.m*

```
% Initialize the multibody tree
%-----
[treeDS, x] = TreeInit( body, r, v );

% Print out the tree
%-----
TreePrint( body, treeDS );
% Plotting and initialization
%-----
tSim      = 20;
dTSim     = 0.1;
nSim      = floor(tSim/dTSim);
hPlot     = zeros(1,nSim);
tPlot     = zeros(1,nSim);
xPlot     = zeros(length(x),nSim);
t         = 0;

% Initialize the time display
%-----
tToGoMem.lastJD      = 0;
tToGoMem.lastStepsDone = 0;
tToGoMem.kAve       = 0;
ratioRealTime        = 0;
[ ratioRealTime, tToGoMem ] = TimeGUI( nSim, 0, tToGoMem, 0, dTSim, 'Tree_Sim' );
```

*TreeSim.m*

Run the simulation. The angular momentum is computed for informational purposes only.

**Listing 7.13:** Tree Simulation: The simulation loop.

*TreeSim.m*

```
% Run the simulation
%-----
for k = 1:nSim

    % Display the status message
    %-----
```

```

[ ratioRealTime, tToGoMem ] = TimeGUI( nSim, k, tToGoMem, ratioRealTime, dTSim );
% Save for plotting
%-----
hPlot(k) = Mag( TreeH( x, t, treeDS, body ) );
xPlot(:,k) = x;
tPlot(k) = t;

% Update the equations of motion
%-----
x = RK4( 'Tree', x, dTSim, t, treeDS, body );
t = t + dTSim;

% Time control
%-----
switch simulationAction
case 'pause'
    pause
    simulationAction = '┘';
case 'stop'
    return;
case 'plot'
    break;
end
end
end

```

*TreeSim.m*

Plot the results.

**Listing 7.14:** Tree Simulation: Plotting.

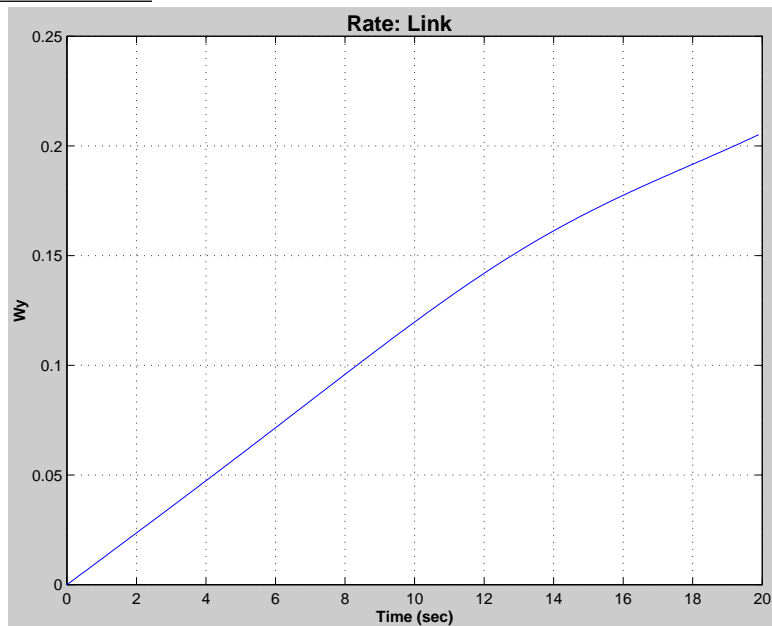
*TreeSim.m*

```
TreePlot( tPlot, xPlot, treeDS, body, hPlot )
```

*TreeSim.m*

An example plot is shown in Figure 7.4 on page 74.

**Figure 7.4:** Tree Simulation: One of several plots.



---

# PLANETARY AND SUN EPHEMERIS

---

## 8.1 Overview

---

The `Ephem` directory provides functions for time-dependent information such as finding planet, moon, and sun locations; coordinate frame transformations; time transformations; solstice, equinox, and eclipses. There also utilities for drawing ground tracks, finding Lagrange points, and computing parallax. Functions for attitude frames related to the sun vector are also found here, such as `sun-nadir` (`QSunNadir`) and `sun-pointing` (`QSunPointing`). Type `help Ephem` for a full listing.

There are demos for finding Earth nutation, precession, and rotation (`TEarth`); eclipses (`TEclipse`); and the simplified planets model (`TPlanets`).

The most recent additions to the toolbox ephemeris capability include an interface to the JPL ephemerides and functions more suited to use in interplanetary orbits, such as `SunVectorECI` in addition to `SunV1` and `SunV2`.

The `CubeSat` Toolbox contains only a subset of these functions, including `SunV1`, `MoonV1`, `Planets`, and `ECIToEF`, which is a computationally fast almanac version of the transformation from ECI to the Earth-fixed frame.

## 8.2 Almanac functions

---

The toolbox has a variety of almanac functions for obtaining the ephemeris of the planets<sup>1</sup>, moon, and sun. For example, the following functions are available in `Ephem`:

- `MoonV1` <sup>2</sup>
- `MoonV2` <sup>3</sup>
- `Planets`, `PlanetPosition` <sup>4</sup>
- `SolarSys`
- `SunV1` <sup>5</sup>
- `SunV2` <sup>6</sup>

---

<sup>1</sup>Explanatory Supplement to the Astronomical Almanac (1992.) Table 5.8.1. p. 316.

<sup>2</sup>The 1993 Astronomical Almanac, p. D46.

<sup>3</sup>Montenbruck, O., Pfleger, T., Astronomy on the Personal Computer, Springer-Verlag, Berlin, 1991, pp. 103-111.

<sup>4</sup>Explanatory Supplement to the Astronomical Almanac, 1992, p. 316.

<sup>5</sup>The 1993 Astronomical Almanac, p. C24.

<sup>6</sup>Montenbruck and Pfleger, p. 36.

## 8.3 JPL Ephemeris

The toolbox has the capability to use a JPL ephemeris file such as the DE405 set within the Toolbox. The functions `PlanetPosJPL` and `SolarSysJPL` provide equivalence to `PlanetPosition` and `SolarSys`.

NASA's Jet Propulsion Laboratory (JPL) freely provides these ASCII Lunar and Planetary Ephemerides which can be downloaded from its website at the ftp site at `ftp://ssd.jpl.nasa.gov/pub/eph/export` with help at `http://ssd.jpl.nasa.gov/?planet_eph_export`. The functions used in the toolbox for the purpose of reading and interpolating these JPL ephemeris files are based on a C implementation by David Hoffman (Johnson Space Center) available on the ftp site. They generate state data (position and velocity) for the sun, earth's moon and the 9 major planets. The MATLAB functions require binary versions of the ephemeris which are system-dependent and therefore must be created by users.

### 8.3.1 Creating and managing binary files of the JPL ephemerides

Users will first need to compile binary versions of the ASCII files on their own system. This can be done with any of a number of tools available from the website. The ASCII files are available in 20 year units, i.e. `ASCP2000.405`, `ASCP2020.405`. The conversion also needs the header file `HEADERPO.405`. If you compile Hoffman's C utility on your computer, you could for example do

```
$ ./convert header.405 ascp2000.405 bin2000.405

Writing record: 25
Writing record: 50
Writing record: 75
Writing record: 100
Writing record: 125
Writing record: 150
Writing record: 175
Writing record: 200
Writing record: 225

Data Conversion Completed.

Records Converted: 230
Records Rejected: 0
```

and similarly create a file for `bin2020.405`. To verify the files, you can use `print_header` and `scan_records`.

```
$ ./print_header bin2020.405

GROUP 1010

JPL Planetary Ephemeris DE405/DE405
Start Epoch: JED= 2305424.5 1599 DEC 09 00:00:00
Final Epoch: JED= 2525008.5 2201 FEB 20
00:00:00

GROUP 1030

2305424.50 2525008.50 32.00

GROUP 1040

156
DENUM LENUM TDATEF TDATEB CENTER CLIGHT AU EMRAT GM1 GM2
GMB GM4 GM5 GM6 GM7 GM8 GM9 GMS RAD1 RAD2
```



RAD4	JDEPOC	X1	Y1	Z1	XD1	YD1	ZD1	X2	Y2
Z2	XD2	YD2	ZD2	XB	YB	ZB	XDB	YDB	ZDB
X4	Y4	Z4	XD4	YD4	ZD4	X5	Y5	Z5	XD5
YD5	ZD5	X6	Y6	Z6	XD6	YD6	ZD6	X7	Y7
Z7	XD7	YD7	ZD7	X8	Y8	Z8	XD8	YD8	ZD8
X9	Y9	Z9	XD9	YD9	ZD9	XM	YM	ZM	XDM
YDM	ZDM	XS	YS	ZS	XDS	YDS	ZDS	BETA	GAMMA
J2SUN	GDOT	MA0001	MA0002	MA0004	MAD1	MAD2	MAD3	RE	ASUN
PHI	THT	PSI	OMEGAX	OMEGAY	OMEGAZ	AM	J2M	J3M	J4M
C22M	C31M	C32M	C33M	S31M	S32M	S33M	C41M	C42M	C43M
C44M	S41M	S42M	S43M	S44M	LBET	LGAM	K2M	TAUM	AE
J2E	J3E	J4E	K2E0	K2E1	K2E2	TAUE0	TAUE1	TAUE2	DROTEX
DROTEY	GMAST1	GMAST2	GMAST3	KVC	IFAC	PHIC	THTC	PSIC	OMGCX
OMGCY	OMGCZ	PSIDOT	MGMIS	ROTEX	ROTEY				

```
$ ./scan_records bin2000.405
```

Record	Start	Stop
1	2451536.50	2451568.50
2	2451568.50	2451600.50
...		
230	2458832.50	2458864.50

You can concatenate the files using `append`. In this case, the data from the second binary file is added to the first. When you scan the records of the combined file the number is increased from 230 to 458. There is also a function to enable you to extract records from binary files to create a custom time interval.

```
$ mv bin2000.405 binEphem.405
$ ./append binEphem.405 bin2020.40
$ ./scan_records binEphem.405
```

Record	Start	Stop
1	2451536.50	2451568.50
2	2451568.50	2451600.50
...		
458	2466128.50	2466160.50

### 8.3.2 The `InterpolateState` function

The function which directly interfaces with the JPL ephemeris files is `InterpolateState`. This function returns the state of a single body at a specific Julian Date. The state is measured from the solar system barycenter and in the mean Earth equator frame. The function call is

```
[X, GM] = InterpolateState( Target, Time, fileName )
```

The numbering convention for the `Target` bodies is given in table [Table 8.1 on page 78](#): Additional computations are required to obtain planet states referenced from the sun, or to obtain the heliocentric positions of the Earth and moon.

### 8.3.3 JPL Ephemeris Demos

The function `InterpolateState` has its own built-in demo. Recall that this state is measured from the solar system barycenter and is in the Earth equatorial frame.

```
State for Mercury on Jan 1, 2001
 2.454786013744712e+07
-5.399651407895338e+07
```

**Table 8.1:** Target Numbering Convention

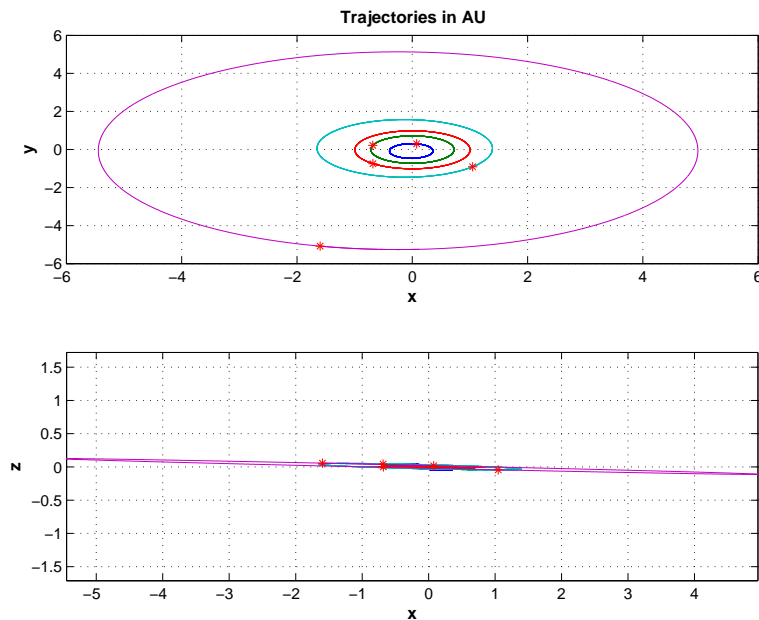
Number	Bodies
1	Mercury
2	Venus
3	Earth-Moon Barycenter
4	Mars
5	Jupiter
6	Saturn
7	Uranus
8	Neptune
9	Pluto
10	Geocentric Moon
11	Sun

```

-3.136647347180613e+07
 3.530110141765838e+01
 1.987507976959802e+01
 6.957125863787630e+00

```

`PlanetPosJPL` and `SolarSysJPL` also have built-in demos. These functions compute the planet states from the sun's geometric position and allow for output in the ecliptic frame. `SolarSysJPL` will create a plot of the trajectories of the first five planets in the ecliptic frame.

**Figure 8.1:** Planet trajectories as generated by the built-in demo of `SolarSysJPL`

The demo `JPLPlanetDemo` demonstrates both `SolarSysJPL` and `PlanetPosJPL`. The demo tests that the planet positions from both functions match by overlaying the output. The geocentric moon state is also plotted.

# PLOTTING TOOL

---

This chapter explains how to use the Plotting Tool to display the results of your multiple body simulations.

## 9.1 Introduction

---

The Plotting Tool is a graphical user interface which enables convenient and customized viewing of simulation data. The tool supports data generated by either MATLAB or external simulations, and allows users to configure the way in which they view and analyze the data.

There are two types of files which may be loaded into the plotting tool:

- **Simulation Output Files:** These files contain raw data from a simulation, consisting of the time history of several different variables. \* Note that these files are treated as “Read-Only” they are never modified while using the tool.
- **Template Files:** These are .mat files which contain user preferences. Each template file acts as a filter to the simulation data. They affect how the variables are displayed and plotted, and how new data is derived.

In addition to loading in a data file, you may load in data directly from the MATLAB workspace.

While using the Plotting Tool, you have several options available to you for configuring how the data is displayed. You may group variables, exclude variables from the display, develop methods for deriving new data, create time history figures, and perform 3-dimensional animations. At any point, these settings may be saved to a template file.

The key feature of the Plotting Tool is that is flexible. Any number of template files may be created and applied to any set of simulation data. The files are organized in the **Plotting** module.

```
>> help Plotting
PSS Toolbox Folder Plotting
Version 7.1      13-Jul-2009

Directories:
Demos
Demos/GUI
Derivation Functions
GUI
Help
Parametric
Sim Results
```

Templates  
Utilities

The remainder of this chapter describes how to use the Plotting Tool, and discusses the various features that you may find useful.

## 9.2 Getting Started

To open the GUI, type `PlottingTool` at the MATLAB prompt.

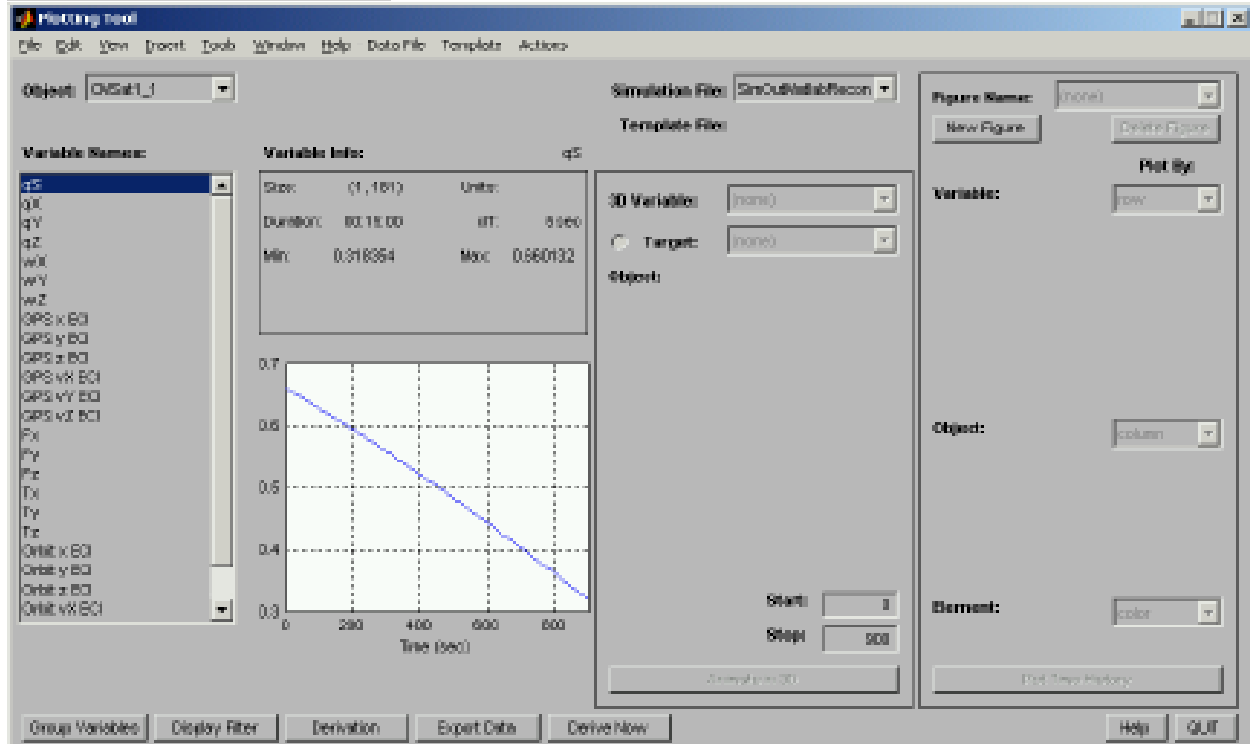
```
>> PlottingTool
```

To begin using the tool, you must first load in data. Data can be loaded in from a file or directly from the MATLAB workspace. In this example we will load a data file. Click on the Data File menu, located at the top of the GUI in the menu bar, and select the Load option from the drop-down menu that appears. The Save As and Close options are initially inactive, since no data file has yet been loaded. As an alternative to using the GUI menubar, you may use the shortcut CTRL+L to load data files.

You will be prompted to select a data file. The first Sim Results folder found in your MATLAB path is the default location used in the browser window. Choose the example data file included with the build, `SimOut-MatlabRecon.mat`.

The simulation data consists of three spacecraft, each with 25 single-dimensional time-history variables. The duration of the simulation is 900 seconds, with a fixed 5 second time-step. A snapshot of the GUI as it should appear after loading the data file is provided below.

**Figure 9.1:** Plotting Tool GUI with Newly Loaded Data File



At this point, one simulation data file has been loaded. You may load as many data files into the tool as you wish, however only one data file is displayed at a time. A pull-down menu located in the top-center portion of the GUI allows you to choose which data file to display.

From this point, you may begin working with the tool. Several options are available to you, including:

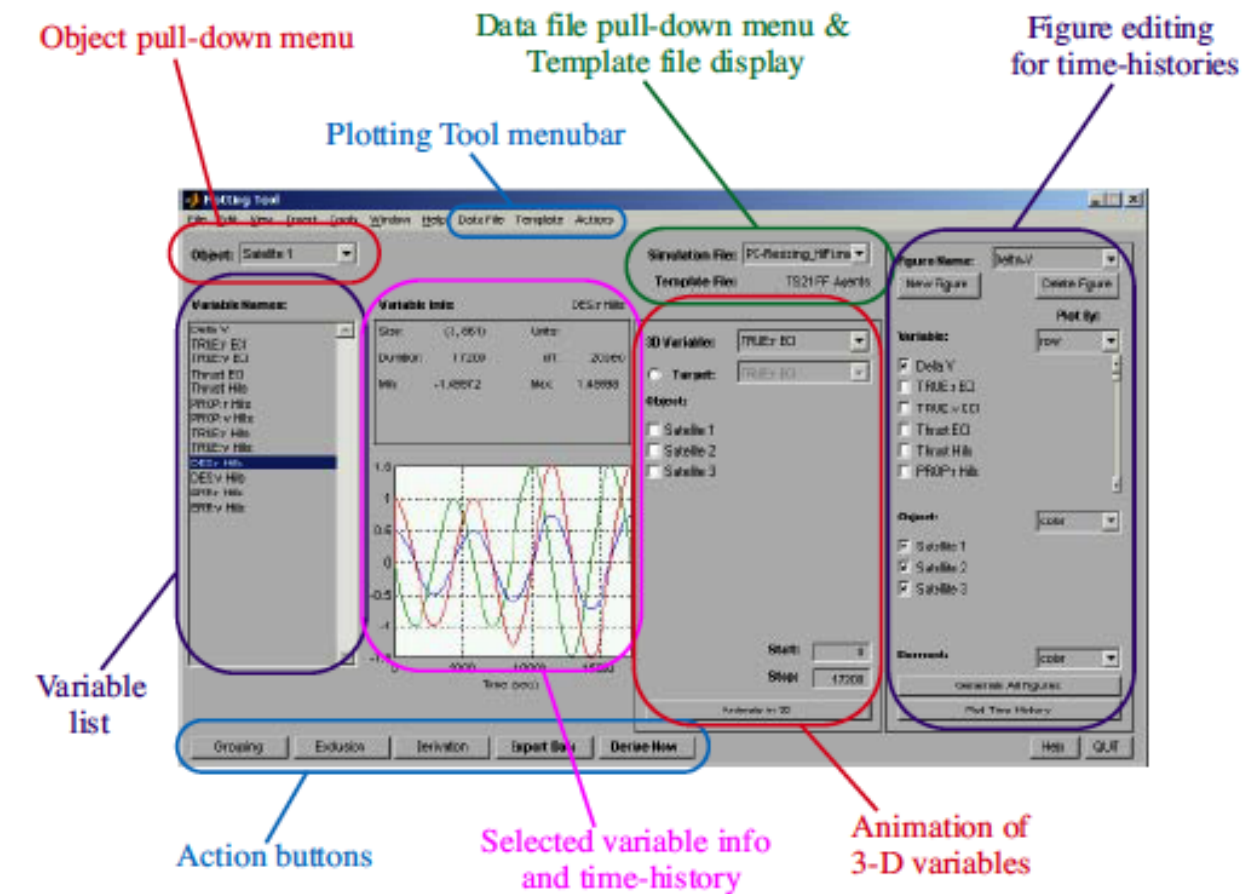
- Grouping and organizing variables
- Selecting which variables should be displayed to the screen
- Generating time-history plots
- Animating 3-dimensional variables
- Deriving new data from the raw data
- Applying predefined templates to perform any of the above actions automatically

The remainder of this document explains how to use these various features.

### 9.3 GUI Layout and Features

This section will describe the general layout and features of the Plotting Tool. The diagram in Figure 11-2 illustrates the layout of the GUI.

**Figure 9.2:** Diagram of the Plotting Tool GUI



### 9.3.1 Data File Pull-Down Menu

Multiple data files may be loaded into the Plotting Tool. Only one data file is displayed at any time, though. This pull-down menu allows you to select one of the loaded data files to be displayed.

Below the data file pull-down menu is the Template file display. Only one template may be applied at any time.

### 9.3.2 Object Pull-Down Menu

The Plotting Tool is designed to conveniently display the simulation results involving multiple objects, i.e. multiple spacecraft or aircraft. The variables associated with each object are displayed within the GUI. Only one object may be displayed at once. This pull-down menu allows you to toggle between the objects.

### 9.3.3 Variable List

This window displays the filtered list of variables associated with the currently selected object and data file. If no template has been applied, then the original set of variables contained in the data file is displayed. When a template is applied, the set of displayed variables may change from the original set. This is part of the display filtering that occurs when applying a template to the raw data.

The list of variables displayed to the screen is built-up as follows:

1. Begin with the original list of variables from the raw simulation data
2. Add all derived variables
3. Add all variable groups
4. Exclude all those variables contained in the exclusion list

The grouping, exclusion, and derivation of variables may be conducted using separate GUI's, as described in Sections [9.7](#), [9.5](#), and [9.6](#).

### 9.3.4 Variable Information & Time-History

Any variable in the list may be selected to display its time history and general information. The general information includes the vector size, units, time duration, time-step, minimum and maximum value. The units are only displayed if they are supplied in the data file.

### 9.3.5 Figure Editing

The right-hand side of the GUI is used for generating time-history figures. Figures may be created, modified, and deleted. Multiple variables of multiple objects may be included in each figure. In addition, the objects and variables may be organized within the figure by row, column and line-color. More details are presented in Section [9.4 on page 83](#).

### 9.3.6 3-D Variable Animation

A center portion of the GUI is used for setting up the animation of 3-dimensional variables. Animation is discussed in detail in Section [9.8 on page 86](#).

### 9.3.7 Action Buttons

Five action buttons are provided at the bottom of the GUI. The first three buttons bring up new GUI's:

- Grouping Brings up the Group Variables GUI
- Exclusion Brings up the Variable Exclusion GUI
- Derivation Brings up the Data Derivation GUI

The remaining two buttons are titled Export and Derive Now. The Export button will export all of the variables associated with the currently selected data file to MATLAB workspace. If multiple objects are loaded, the variable names are appended with “\_1”, “\_2”, etc. The Derive Now button will cause all of the derived variables to be recomputed.

In addition to these 5 action buttons, a Help button and a Quit button are located at the bottom righthand side of the GUI.

### 9.3.8 Plotting Tool Menubar

A set of menu controls specific to the Plotting Tool are provided in the menubar of the figure. The menus include Data File, Template and Actions. From the Data File menu, you can load and close data files, and save them as new `.mat` files with a plotting tool format. From the Template menu, you can apply, save or remove a template file. The Actions menu contains links to the Derivation, Exclusion and Grouping GUI's, and includes additional options to export the data to the workspace, recompute the derived variables, or quit the current session.

### 9.3.9 Shortcuts

Several of the buttons or menu items in the Plotting Tool have shortcut keys. The following table lists all of the shortcuts.

**Table 9.1:** Shortcut Keys for the Plotting Tool

Action	Shortcut (PC)	Shortcut (Mac)
Load Data File	CTRL + L	CMD + L
Close Data File	CTRL + X	CMD + X
Apply Template	CTRL + A	CMD + A
Save Template	CTRL + T	CMD + T
Remove Template	CTRL + R	CMD + R
Export Data to Workspace	CTRL + E	CMD + E
Derive Data	CTRL + D	CMD + D
Quit	CTRL + Q	CMD + Q

## 9.4 Creating Figures

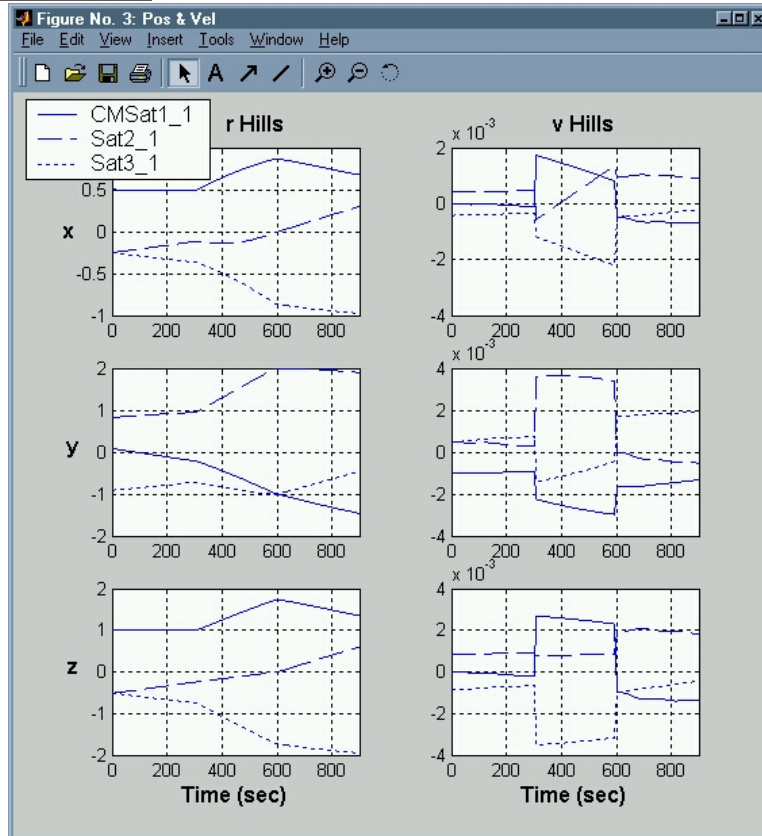
The right hand side of the GUI is used for generating figures. First, click the New Figure button and pick a name for the new figure. You can choose the variables and satellites you wish to plot from the checkbox lists, and indicate how they are distinguished with the pull-down menus under the words, Plot By.

In general, the data has three descriptive components to it: variable, satellite, and element. A given variable may have one or multiple elements, and a unique copy of each variable exists for each satellite. For a given figure, the user may choose which variables and which satellites to include in the plot. The user may also define the manner in which these three components are organized within the figure.

Each component may be assigned one of the following four plotting characteristics: row, column, color, and linestyle. Organizing by row or column causes separate sub-plots to be created within the figure, while the color and linestyle attributes distinguish the components within a single plot.

Figure 9.3 on page 84 provides an example of the type of figures that can be quickly generated with the Plotting Tool. It shows the relative position and velocity of three spacecraft. The  $x$ ,  $y$ , and  $z$  elements are separated by row, the **r Hills** and **v Hills** variables separated by column, and the satellites distinguished by linestyle. The figure could be easily generated again in several different formats. For each figure created, the title, variables, satellites and all plotting characteristics are stored in the template.

**Figure 9.3:** Plotting Tool Generated Figure



The two variables plotted in this figure were not included in the simulation output - they were derived automatically upon applying a template. In this case, a derivation function was used which takes the position and velocity of the satellites in the ECI frame as inputs, performs a coordinate transformation, and returns the relative states in the local Hills frame. The exact process involved with creating these variables is discussed in further detail in Section 9.7.

## 9.5 Grouping Variables

It is often useful to work with a vector of data rather than the individual elements. The plotting tool allows you to group individual variables together.

By pressing the Grouping button at the bottom of the Plotting Tool, a new Group Variables GUI will open up. This GUI may also be opened from the Actions menu. Within the Group Variables GUI, you may press the New Group button to create a new variable group. Once a group has been created, you may scroll through the checkbox list of



simulation elements to assign variables to the group. Groups may always be edited, renamed, or deleted.

If any groups have been created, modified or deleted, the variable display in the Plotting Tool will be updated accordingly only after the Group Variables GUI is closed.

As an example, load the “SimOutMatlabRecon.mat” data file. Create two groups, “**r ECI**” and “**v ECI**”. Add the following variables to the “**r ECI**” group: Orbit:x ECI, Orbit:y ECI, Orbit:z ECI. Similarly, add the following variables to the “**v ECI**” group: Orbit:vX ECI, Orbit:vY ECI, Orbit:vZ ECI. Press the Exit button, and scroll to the bottom of the variable display in the Plotting Tool to view the new groups.

## 9.6 Excluding Variables

---

It is often the case where several types of telemetry is collected, but only a select few are of interest most of the time. Or in some cases, redundant sets of data are stored, and it is usually only necessary to view a unique set. The Plotting Tool allows you to exclude any variables from the display that you do not wish to see.

Press the Exclusion button at the of the Plotting Tool to pull up the Display Filter GUI. Simply select those variables that you do not wish to be displayed, then press the Exit button.

## 9.7 Deriving New Data

---

An often repetitive task involved with analyzing raw data is deriving new, meaningful data from it. One of the aims of this tool is to make the process of creating and viewing derived data more automatic and therefore less time-consuming. Press the Derivation button at the bottom of the Plotting Tool to bring up the Data Derivation GUI.

The Data Derivation GUI is set up such that you can choose new outputs to be created from the raw simulation data. Each new output is created with a *Derivation Function*, which is supplied with your choice of inputs. The steps below provide an example of how to derive new data. In order to follow along with the steps, first load the SimOutMatlabRecon.mat data file and refer to Section 9.5 for an explanation of how to create the “**r ECI**” and “**v ECI**” groups.

The steps for deriving new data are summarized as follows:

1. Choose a name for the output. For example, derive the relative position vector in the Hills frame, “**r Hills**”, from the ECI position and velocity data.
2. Choose a function with which to create the output. In this case, the function to use is `TransformECI2Hills.m`.
3. Choose the inputs for your function. The function `TransformECI2Hills.m` requires 2 inputs: the position in the ECI frame, “**r ECI**”, and the velocity, “**v ECI**”.

Press the New Output button and name your new output “**r Hills**”. The default function assigned to new outputs is `DerivationTemplate`. Change the function to `TransformECI2Hills`.

Now, select the inputs. Choose the following 2 groups: “**r ECI**” and “**v ECI**”. Each input passed into the derivation functions includes the data for all satellites. Press the Exit button to return to the Plotting Tool.

If changes have been made to your derivation settings, all of your derived outputs will automatically be recalculated upon exiting the derivation window. In addition, you may re-derive data at any time by pressing the Derive Now button on the bottom of the main Plotting Tool GUI.

When you exit from the Data Derivation GUI, the derivation functions are called for all outputs that you have created, and the derived data is stored in the PlottingTool GUI. If you have assigned the wrong number of inputs to a function,

or have supplied inputs of an improper size, a dialogue box will be displayed indicating the error. If there is an error inside the function itself, the GUI can be closed once the error has been fixed.

Take a closer look at the `TransformECI2Hills.m` function to see how the ECI states are converted to the Hills frame. Several options are available from within the function itself which dictate how the new data is calculated. In the future, any options associated with derivation functions will be made available from within the Derive Data GUI.

## 9.8 Animation

---

The Plotting Tool also allows you to set up animations of any three-dimensional variables. Any group having three elements is displayed in the list of variables in the animation section of the main GUI.

One of the key applications of the 3-D animation feature is viewing the relative motion of satellites in a local reference frame. In formation flying, the objective is to control the relative position and velocity of the satellites such that a particular shape or configuration is maintained. The manner in which these relative states evolve over time can be viewed nicely through animation.

While it is perhaps most intuitive to view 3-dimensional variables representing spatial coordinates (such as the satellites' relative positions), the tool can be used to animate any 3-dimensional variable. The potential therefore exists for users to gain new insights into their data by experimenting with this feature.

First, select one of the three-dimensional variables. This might be a measured position, for example. If there exists another 3-D variable that represents the desired value, then it may be chosen as the target. Next, choose which of the objects you wish to animate. You may also choose the time window that is animated by changing the start and stop times.

Push the Animate in 3D button to open the `AnimationGUI`. Figure 9.4 on page 87 provides a snapshot of the Animation GUI generated with the relative Hills position derived from the `SimOutMatlabRecon.mat` data file.

## 9.9 Templates

---

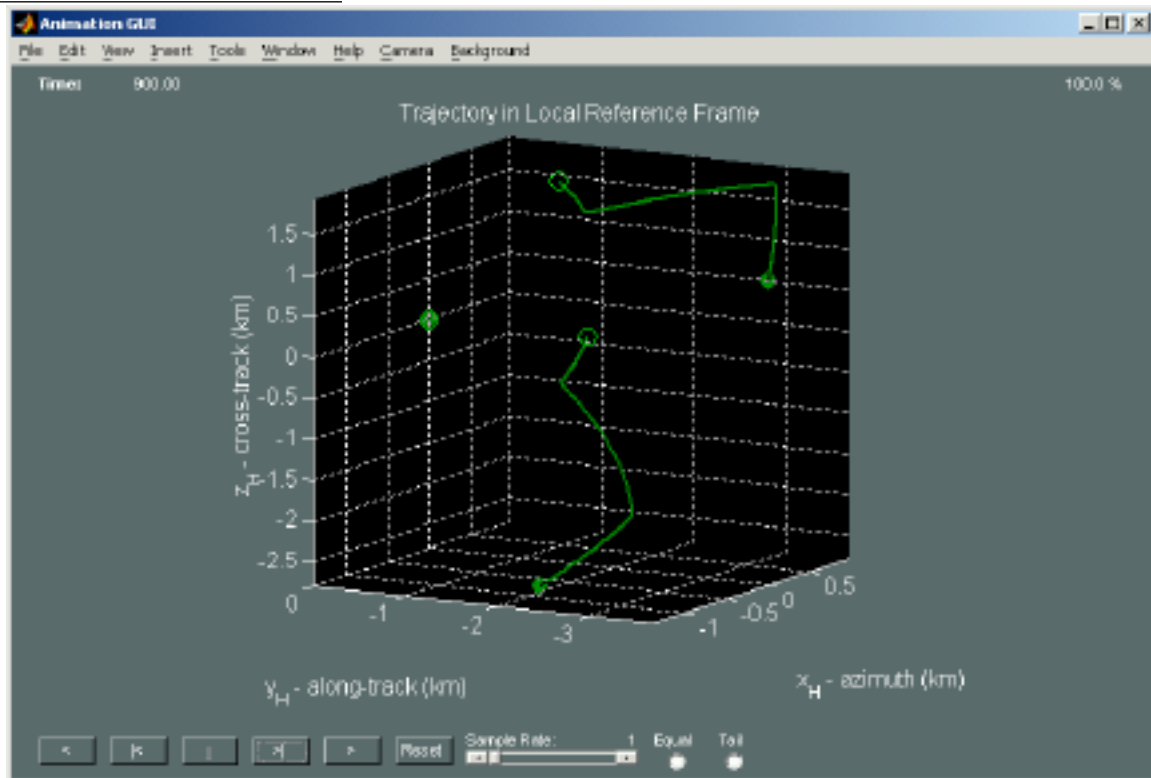
Once a data file is loaded into the Plotting Tool, the user may manipulate the data in a number of different ways. Variables may be grouped or excluded, new variables derived, and figures generated. All of these settings may be saved to a template file. The template may be edited over time, and applied to other simulation files.

Templates are stored as `.mat` files. As you make changes to the display, add figures or derive new variables, you can continue to save the template. Go to the Templates menu, select the Save option, and the old file will be overwritten. Otherwise, you may choose the Save As option and save the template under a different name.

As an example, load the `SimOutMatlabRecon.mat` data file and apply the `FF_Matlab.mat` template to it. First, go to the Data File menu and select Load, or just type CTRL+L. Select and open the `SimOutMatlabRecon.mat` data file. The next step is to apply a plotting template to this raw data. Go to the Template menu, located at the top of the GUI in the menubar, and select the Apply option. Alternatively, use the shortcut CTRL+A.

You will now be prompted to select a template file. The first "Templates" folder found in your MATLAB path is used as the default folder in the browser window. Choose the `FF_Matlab.mat` template file that was included with the build. The template is now applied to the data file in the Plotting Tool.

Figure 9.4: Animation GUI Snap-Shot



## 9.10 Plotting Output from Custom Simulation Software

The Plotting Tool is capable of plotting output formats generated by certain MATLAB simulations or output log files generated by any external application. This section describes how to read a custom output file into the Plotting Tool. This requires the user to write an m-file to read in the data. An alternative approach is to use the standard PSS simulation output file format. The next two subsections describe these alternatives. Users who do not have need of this advanced functionality can skip the rest of this section.

Princeton Satellite Systems has developed a C++ based simulation tool called DSim, which outputs data files into the proper format for the Plotting Tool. The ability to run custom simulations outside of MATLAB and then use MATLAB's graphical analysis capabilities to analyze simulation results has proven extremely valuable.

### 9.10.1 Reading a Custom Output File

The Plotting Tool is designed to make it easy for users to read in their own custom output files. There are three steps to this process: creating the output file, creating a custom m-file to read the output file, and modifying the `PlottingTool.m` file.

#### Creating a Custom Output File

The output file must be of a format that MATLAB can read and text files are probably the easiest to deal with. A number of c-type functions are available within MATLAB to read and parse text files. It is advised that the output file be a text file delimited by either spaces or commas. The remainder of this section assumes the output is a text file.

The only requirement for creating the output file is that the first line contains a description of the type of output. For

example, the first line of each DSim output ends with the string “Simulation Output”. The entire line is a description of the type of simulation that was run, for example: “Formation Flying Simulation Output”. This first line descriptor is used in the `PlottingTool.m` file to determine how to read the rest of the output file.

Aside from that requirement, you can format the file in any fashion you desire. It is strongly encouraged that the file contain not only the raw data but also the time associated with the data and the variable names of each of the data elements. Optional information that can be included in the file is described in the next subsection.

### Creating an M-File for Reading in a Custom Output File

Once you’ve created an output file, you must create an m-file that reads it into the Plotting Tool. This function call should have the following format

```
d = ReadCustomFile( fId, any_other_necessary_parameters );
```

where `d` is an output data structure whose fields are defined in Table 9.2 on page 88, ‘`ReadCustomFile`’ is the name of the m-file, ‘`fId`’ is the MATLAB file identifier for the output file, and ‘`any_other_necessary_parameters`’ is a list of any additional information that must be passed to the m-file that reads your output. For example, the default file reader, `ReadOutputFile.m`, is passed the name of the output file for use in error messages that are generated if it encounters problems reading the file.

Table 9.2 shows the possible fields of the data structure `d`. The only field that must exist is ‘`data`’ although it is strongly recommended that the ‘`dataNames`’ and ‘`time`’ fields also be included. The remaining fields enable you to provide more information about the data as well as control how some of the information is displayed.

**Table 9.2:** Fields of Output Data Structure `d`

Field	Format	Description
<code>data</code>	{1 x nObjects} cell array with each cell containing a (nD x nT) matrix of raw data	Each cell contains the data for a specific spacecraft. The matrix of raw data has one row for each data element and one column for each time point. These data elements can be grouped in the ‘ <code>groups</code> ’ field or by using the Plotting Tool after reading in the data.
<code>dataNames</code>	{1 x nD} cell array of strings	Each string is the name of the corresponding data element. If this field doesn’t exist, the elements are named ‘ <code>e1.1</code> ’, ‘ <code>e1.2</code> ’, etc.
<code>dataUnits</code>	{1 x nD} cell array of strings	Each string describes the units of the corresponding data element. Optional.
<code>excluded</code>	{1 x n} cell array of strings	Each string is the <code>dataName</code> of an element that is not to be displayed in the variable list. Optional.
<code>groups</code>	(: ) array of data structures with fields ‘ <code>name</code> ’ and ‘ <code>elements</code> ’	Each element is a data structure corresponding to a new grouping of individual data elements. The ‘ <code>name</code> ’ field is the name of the group as a string. The ‘ <code>elements</code> ’ field is a cell array of strings that contains the <code>dataName</code> of each data element that is to be included in the group. Optional.
<code>jDEpoch</code>	scalar	Simulation start time as Julian date. Optional.
<code>objNames</code>	{1 x nObjects} cell array of strings	Each string is the name of the corresponding object. If this field doesn’t exist, the objects are named ‘ <code>Obj_1</code> ’, ‘ <code>Obj_2</code> ’, etc.
<code>time</code>	(1 x nT)	Row vector of elapsed simulation time in seconds. If this field doesn’t exist, it is assumed the time step is one second.

How the output data structure `d` is assembled from the raw data in the output file is up to you. Any of the built-in MATLAB functions for file and string manipulation can be used and you are free to write your own functions to massage your data into the appropriate format. As an example of how to read a text file and generate the appropriate output data structure, see the `ReadOutputFile.m` file.

### Modifying the PlottingTool.m File

Once you have created the m-file for reading your data, you must modify the `PlottingTool.m` file to recognize this output file type. In the member function “LoadSimData” (which is in the `PlottingTool.m` file), locate the following comment lines:

```
% User can add additional output file types here using
% elseif( strcmp( line1, 'first_line_in_file_string' ) )
```

After these comment lines, add two lines of the following form:

```
elseif( strcmp( line1, 'first_line_in_output_file' ) )
d = ReadCustomFile( fId, any_other_necessary_parameters );
```

where `first_line_in_output_file` is the unique descriptor you’ve chosen for the first line of the output file. As an alternative to using MATLAB’s `strcmp` function, which compares the entirety of both strings, you may want to use the `findstr` function, which looks for the occurrence of the string ‘`first_line_in_output_file`’ within `line1`. The second line is the function call to the m-file you created. Provided that your m-file outputs a data structure `d` with the appropriate fields, you should now be able to use your custom output file with the Plotting Tool.

### 9.10.2 Using the PSS Simulation Output Format

An alternative to creating your own output file format is to use the format described here. If you follow this format, the Plotting Tool will automatically be able to read your output file. The Spacecraft Control Toolbox comes with a sample simulation output file that is in the format described here. It is called “`SampleSimOutput.txt`” and is located in the `/Common/CommonData/` folder. It can be loaded into the Plotting Tool with the following command:

```
>> PlottingTool( 'load_sim_data', 'SampleSimOutput.txt' );
```

The file format can be most easily understood by viewing the contents of this file in an editor. The first five lines are shown here for reference. The entire fifth line is too long to fit so only the first part is shown.

#### Listing 9.1: Example Format for Simulation Output Text File

*SampleSimOutput.txt*

```
1 Small Agile Satellite Simulation Output
2 Epoch 2.452994833e+06
3 nObjects 1
4 Target
5 // time (secs) [1] // rECI1 (km) [3] {x, y, z} // vECI1 (km/s) [3] {xDot, yDot, zDot} . . .
```

*SampleSimOutput.txt*

The first line of the output file must contain the string ‘Simulation Output’. This must be included in the first line of your output file if the Plotting Tool is going to recognize the file type correctly.

The second line is optional and provides the start time of the simulation as a Julian date. The line format is the word ‘Epoch’ (or ‘epoch’) followed by a space followed by the Julian date.

The third line specifies the number of objects in the simulation. This is required. The format is ‘nObjects’ followed by a space followed by an integer.

The fourth line provides the name of the object. In this case we have only one object, so the name is given on one line. For multiple objects, the name should be provided (in order) on separate lines. Each line is read as the name of a new object so you can use whatever characters you’d like in the name.

The fifth line specifies the names of the various data elements, their units, etc. This line is required. Following this line is the raw data. Let’s look at how to format the raw data first and then come back to the element names line.

The raw data is a matrix with each element separated by spaces. Each column is a different output element and each row is a different timestep. The matrix is arranged such that all the outputs from one object are grouped together.

Hence, all outputs of the first object are listed first by column, followed by the columns of the outputs of the second object, etc.

Each object must have the same number and type of outputs. For example, either all objects have the output ‘x’ or none of the spacecraft have output ‘x’. Furthermore, the order of the output columns for each object is the same. i.e. If the first object’s outputs are ‘x’ then ‘y’ then ‘z’, then all objects have the ‘x’ output column followed by the ‘y’ column followed by the ‘z’ column.

Finally, the first column of the raw data corresponds to the simulation time in seconds. This time is the same across all object output columns.

Given these constraints, if there were three spacecraft each with 12 output elements, there would be a total of 37 output columns. The first column is the time, the next 12 columns are the outputs of the first spacecraft, and these are followed by 12 columns for the second spacecraft and 12 columns for the third spacecraft.

Now back to the header line that provides the name information about each output. There are a number of ways to format this line but the general format is

```
// outputName (units) [number of elements] {elName1, elName2, ...}
// outputName ... //
```

where ‘//’ separates different variables, optional parentheses ( ‘(’ and ‘)’ ) are used to specify units, optional square brackets ( ‘[’ and ‘]’ ) are used to specify the number of elements contained in that variable, and optional braces ( ‘{’ and ‘}’ ) are used to specify the names of the individual elements.

The line must start and end with ‘//’. After that, there are a number of options. The simplest method is to provide the name of each element of data. The format would be

```
// el1Name // el2Name // el3Name // ... //
```

In general, it is necessary to define this line for only one spacecraft. The Plotting Tool only supports multiple objects if all objects have an identical set of outputs. The `ReadOutputFile.m` function uses only the first set of outputs in the line to define the variable list. It first counts the total number of variables in the line, and assigns the value to `nVar`. Next, it modifies the value of `nVar` as follows:

```
nVar = 1 + (nVar-1)/nObjects;
```

If you wish to supply only one list of variable names that applies to all objects, then you would omit this line. According to the example above, you would specify thirteen element names (the first would be ‘time’ followed by the names of the 12 output columns for each spacecraft.)

If you wish to include units, the format would be

```
// el1Name (units) // el2Name // el3Name (units) // ... //
```

It is not necessary to specify the units for each element. Some elements can be unitless. The square brackets and braces are used if you wish to group individual elements into larger variables (such as individual ‘x’, ‘y’, and ‘z’ coordinates into ‘rECI’.) This is equivalent to using the Plotting Tool to group variables together and the Display Filter to exclude the individual elements from the variable list.

The square bracket specifies how many columns are associated with that variable name. For example

```
// time // rECI (km) [3] // vECI [3] (km/s) //
```

would group the second through fourth columns into the variable ‘rECI’ and the fifth through seventh columns into ‘vECI’. These grouped variable names would appear in the Plotting Tool variable list instead of the individual element names. Also note that the order of the optional parameters is not important.

The braces allow you to name the individual elements of the grouped variable. If you don't provide names, the Plotting Tool will automatically append numerals to the group name to come up with the individual element names. i.e. In the example above, the group 'rECI' would have individual elements 'rECI\_1', 'rECI\_2', and 'rECI\_3'. Using the format

```
// time // rECI (km) [3] x, y, z // ... //
```

would cause the individual elements to be named 'rECI\_x', 'rECI\_y', 'rECI\_z'. The common 'rECI\_' will not appear in any plots.

The preceding example would cause the variable list in the Plotting Tool to display 'time', 'rECI', and 'vECI' and exclude 'rECI\_x', 'rECI\_y', etc.

You can use the sample simulation output file to see how to format your own output file.

## 9.11 Summary

---

The Plotting Tool is a MATLAB GUI designed to display the results of raw simulation data in a useful way. The purpose of this tool is to provide users with the ability to quickly, conveniently, and consistently analyze the results of their simulation. If you find that you repeatedly view the same types of plots or derive the same types of data, then the Plotting Tool can help to automate these tasks by storing your settings in template files. The output files of both MATLAB and external simulations may be loaded in to the GUI, while different templates are applied to change how the data is displayed. The result is a flexible, user-configurable method for post simulation analysis.





---

# POINTING BUDGETS

---

This chapter discusses how to generate a pointing budget.

## 10.1 Pointing Budget

---

`PBudget`, in the Attitude folder of SC, generates antenna beam pointing budgets. Its inputs are a  $n$ -by-3 matrix of error contributions, an  $n$ -by- $m$  matrix of categories, an  $n$ -by- $m$  matrix of descriptions, and a 2-by-1 matrix of antenna offsets. For the category and description matrices  $m$  signifies the length of the longest string.

### Listing 10.1: `PBudget` Example

```
errors = [0.01 0.02 0.03;...
         0.02 0.04 0.01;...
         0.04 0.03 0.05;...
         0.05 0.00 0.06]; % degrees
categ = 'Bias';
categ = char(categ, 'Bias');
categ = char(categ, 'Diurnal');
categ = char(categ, 'Diurnal');
desc = 'Thermal';
desc = char(desc, 'Misalignments');
desc = char(desc, 'Thermal');
desc = char(desc, 'Misalignments');
aZ = 0.0;
eL = 0.0;
[cep, r, s, t] = PBudget(errors, categ, desc, aZ, eL, 'MyBudget.txt');
```

`PBudget` always adds categories and combines errors within categories by taking the square root of the sum of the squares. You can have as many categories as you wish. The descriptions have no effect on the computations. The outputs are the 3-sigma circular error, the azimuth and elevation beam pointing errors, the category totals and the totals. The 3-sigma error is the angular error that there is a 0.99865 probability that the beam center is within. If the last input is given, `PBudget` will create a pointing budget file called `MyBudget.txt`.

The circular error is computed by numerically integrating the two-dimensional probability density function for azimuth and elevation. The MATLAB `quad` routine is used to perform the inner integration. A bisection search is used to find the value of angular radius with that marks the 3-sigma boundary. A complete pointing budget is given in the script `ExamplePointingBudget`.

### Listing 10.2: Pointing budget example output

*MyBudget.txt*

```

Pointing budget 10-Aug-2007

Thermal
1 Bias      0.0100  0.0200  0.0300  deg
2 Diurnal   0.0400  0.0300  0.0500  deg
-----
Subtotal   0.0412  0.0361  0.0583  deg

Misalignments
3 Bias      0.0200  0.0400  0.0100  deg
4 Diurnal   0.0500  0.0000  0.0600  deg
-----
Subtotal   0.0539  0.0400  0.0608  deg

-----
Total     0.0951  0.0761  0.1191  deg

CEP = 0.1070 deg

```

*MyBudget.txt*

## 10.2 Pointing Budget GUI

The function `PointingBudgetGUI` creates a GUI for doing budgets. Figure 10.1 on page 94 shows the GUI after it has been opened. The File popup menu allows you to select existing pointing budget files or create a new file. The scrollable window below it shows the budget. To the right is the area where you enter roll, pitch and yaw errors (in degrees) and give the error a name. You can select one of four temporal categories using the popup menu. Use the Add button to add the contribution and Remove to remove it. The scrollable area on the right shows the budget and the resulting pointing error.

Only one methodology is available. The Azimuth and Elevation are of the beam center. Hit the Update button to update the budget after you have entered new information.

**Figure 10.1:** Open the GUI

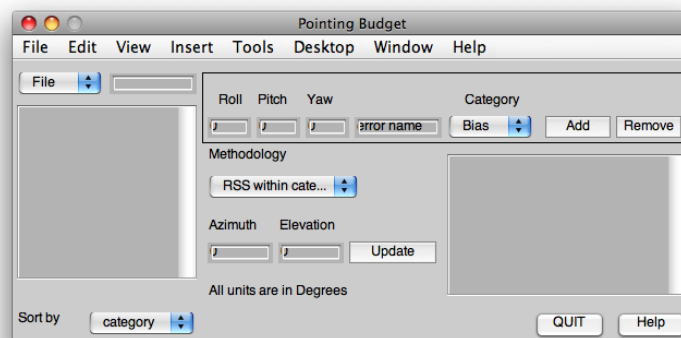


Figure 10.2 on the next page shows the GUI after the first error has been entered. The component appears in both scrollable text windows. The right one also shows the resulting pointing error.

Figure 10.3 on the facing page shows the GUI after the second error has been entered.

Figure 10.4 on the next page shows save file dialog box.

Figure 10.2: Entering the first error

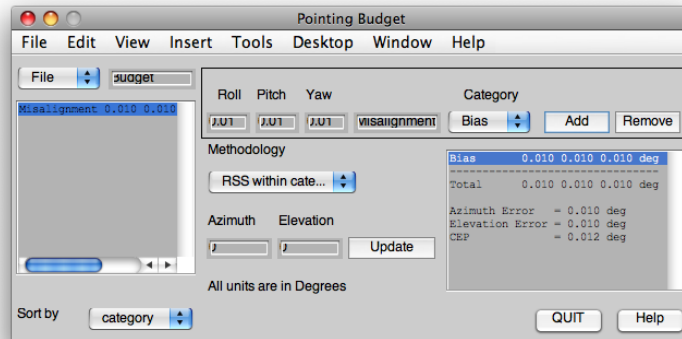


Figure 10.3: Entering a second error

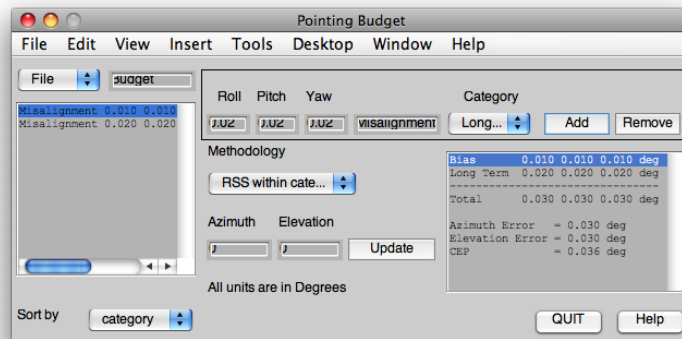
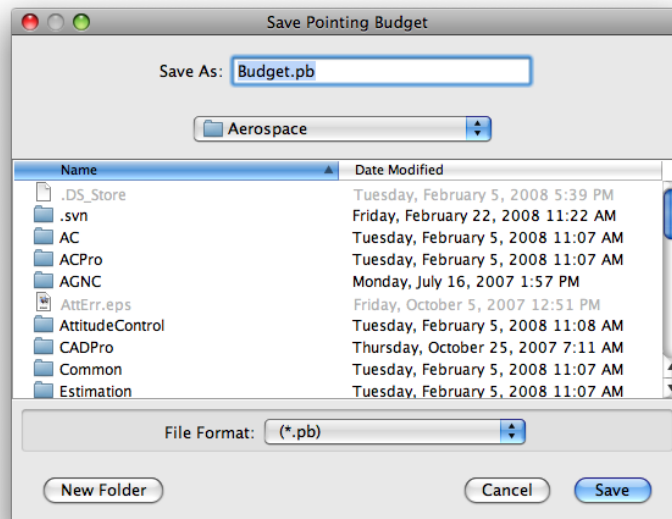


Figure 10.4: Save window







## 11.2 Linear Quadratic Control

In this example we will design a compensator for a double integrator using full-state feedback. A double integrator's states are position and velocity. For full-state feedback, both must be available.

This example is automated using `LQFullState`, shown below.

**Listing 11.1:** Linear Quadratic control design matrices and plant

*LQFullState.m*

```
a = [0 1;0 0];
b = [0;1];
c = eye(2);
d = [0;0];

g = statespace( a, b, c, d, 'Double_Integrator',...
    {'position', 'velocity'}, 'force', {'position', 'velocity'} );

p = FindDirectory( 'CommonData' );
save( fullfile(p,'DoubleIntegrator'), 'g' );

q = eye(2);
r = 1;

w.q = q;
w.r = r;

gC = LQC( g, w, 'lq' );
k = get( gC, 'd' );

[a,b,c,d] = getabcd( g );
inputs = get( g, 'inputs' );
inputs = strvcat( inputs, 'pitch_rate' );
g = set( g, a - b*k*c, 'a' );
Step( g, 1, 0.1, 100 );
```

*LQFullState.m*

The script sets values for the controller design matrices. As you can see, you can also use `LQC` outside of the design GUI. This script also creates the plant model, `DoubleIntegrator.mat`, and saves it to the directory `CommonData`. Run the script and you will get the plot in [Figure 11.2 on page 99](#).

Now type `ControlDesignPlugin`. Select the *plant* block in the diagram (top right) and click `Load Plant` in the bottom of the GUI, then select the file `DoubleIntegrator.mat`. The message window will read `Double Integrator plant loaded`. You can see from the GUI that the plant has one input called `force` and two outputs, `position` and `velocity`.

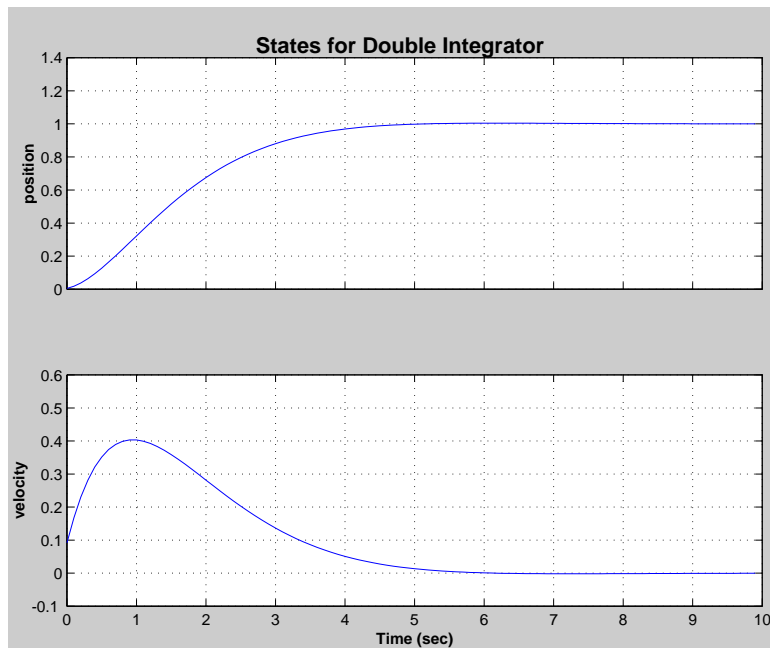
Next, select the *control* block (top center) and then select the `LQ` tab. Select full state feedback, at the top of the list. Input fields for  $Q$ ,  $R$ , and  $N$  will appear. Enter  $q$  and  $r$  into the corresponding input fields. The display will look as follows ([Figure 11.3 on page 100](#)). Push `Create`. The values for  $q$  and  $r$  are read in from the workspace. This eliminates the need to type in potentially large matrices. When you read in a controller these matrices are stored in the workspace.

Next click the *control* block to unhighlight it so that you can work with the whole system. You can now do a step response by pushing `Step`, [Figure 11.4 on page 101](#).

## 11.3 Single-Input-Single-Output

Close and reopen the GUI and load in the double integrator plant. Next select the *control* block and the `SISO` tab. Add the input `position` and output `force`, by typing in the boxes and then clicking `Add`. The results will show up in the list on the left. Then add a transfer function `TF` at the bottom of the tab. Push the button to make `position` the

Figure 11.2: Step response



transfer function input (To Input in the Input section) and `force` the output (To Output), respectively. The names will appear in the lists to the right in the transfer function area. Now select TF and click PD in the SISO List. The GUI will look like that in Figure 11.5 on page 101.

Hit the Save button under the transfer function heading. Select the MapIO tab. You will see that the inputs and outputs of the plant and controller are aligned properly.

Under plant to sensor click velocity and hit remove since it is not used by the SISO controller. When removed, velocity is prefixed by a star to indicate that it is part of the plant but unused. Click the control box to select the whole plant and hit step. You will see the following step response (Figure 11.7 on page 102).

## 11.4 Eigenstructure Assignment

Run the script `CCVDemo`. This script generates the inputs for the eigenstructure assignment example. The model is already stored in `CCVModel.mat`.

Listing 11.2: Control configured vehicle example

CCVDemo.m

```
% Plant matrix
%-----
g = CCVModel;

% Desired eigenvalues and eigenvectors
%-----
lambda = [ -5.6 + j*4.2; -5.6 - j*4.2; -1.0;...
           -19.0; -19.5];
vD = [ 1-j  1+j  0  1  1;...
       -1+j -1-j  1  0  0;...
       0    0  0  0  0];

% We really want to decouple gamma
%-----
w = [ 1    1    1  1  1;...
```

Figure 11.3: LQ GUI

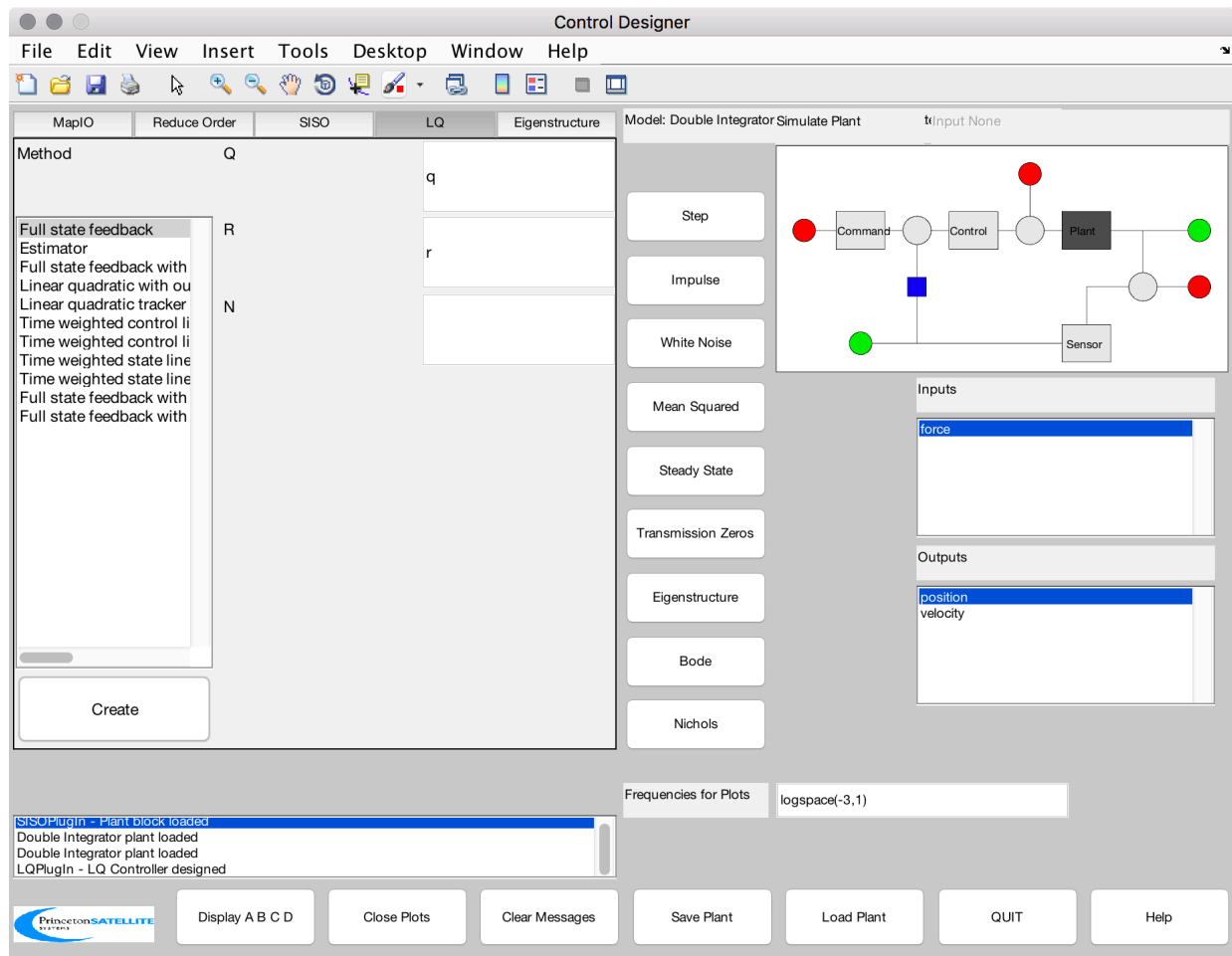




Figure 11.4: Step response from the GUI

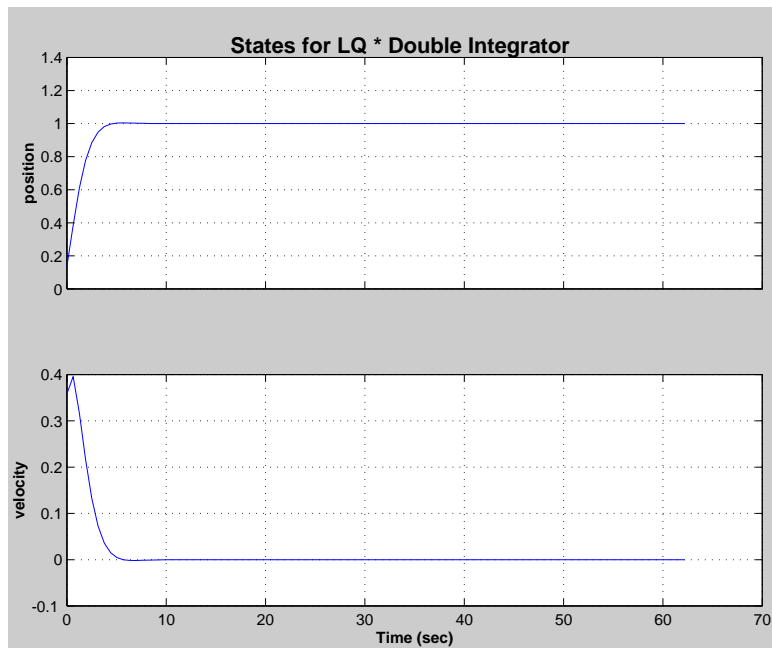


Figure 11.5: SISO inputs

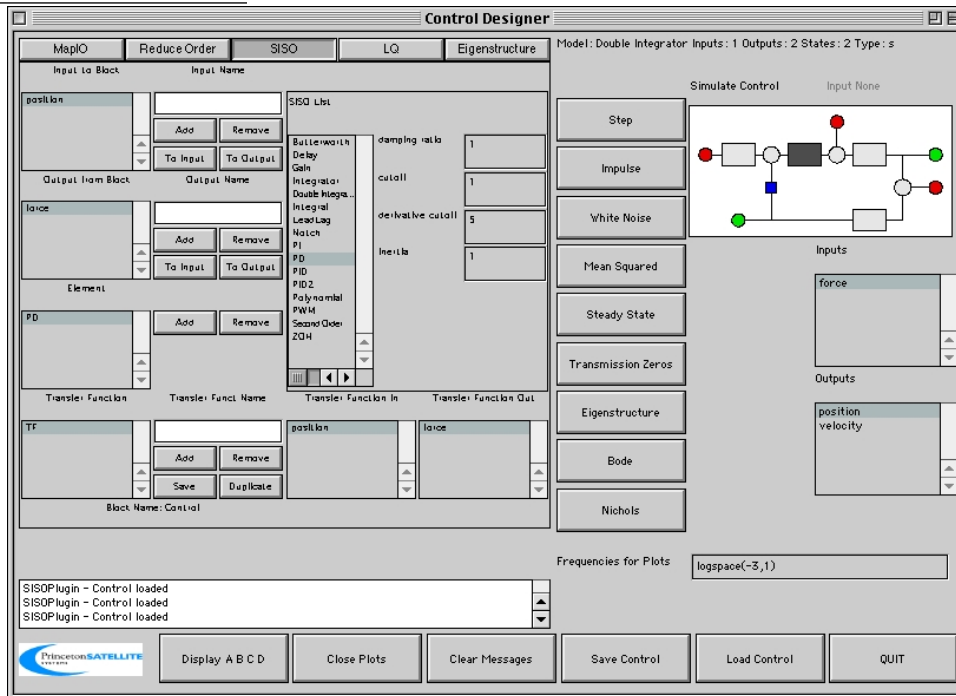


Figure 11.6: MapIO

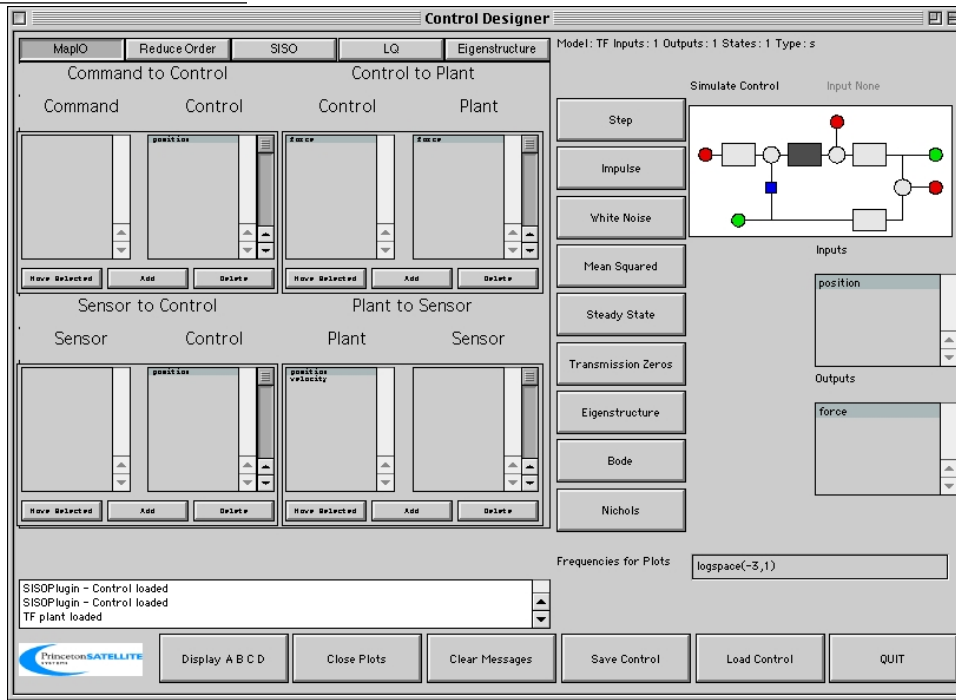
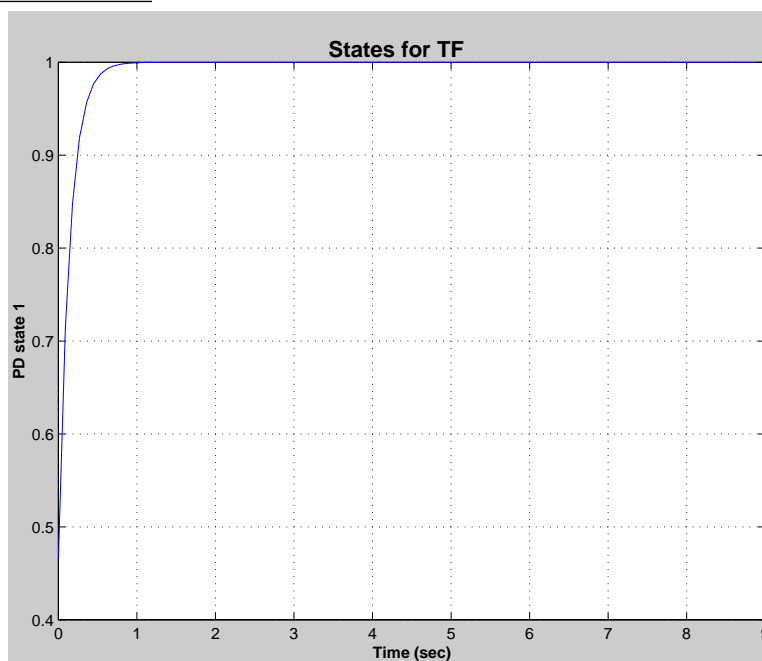


Figure 11.7: SISO step response



```

    1   1   1   1   1;...
    100 100 1   1   1];

% The design matrix.
%-----
d = [eye(3),zeros(3,2);... % Desired structure for eigenvector 1
     eye(3),zeros(3,2);... % Desired structure for eigenvector 2
     0 1 0 0 0;...        % Desired structure for eigenvector 3
     0 0 1 0 0;...        %
     0 0 0 1 0;...        % Desired structure for eigenvector 4
     0 0 0 0 1];         % Desired structure for eigenvector 5

% Rows in d per eigenvalue
% Each column is for one eigenvalue
% i.e. column one means that the first three rows of
% d relate to eigenvalue 1
%-----
rD = [3,3,2,1,1];

% Compute the gain and the achieved eigenvectors
%-----
[k, v] = EVAssgnC( g, lambda, vD, d, rD, w );

```

CCVDemo.m

$\lambda$  gives the desired eigenvalues, something that would be specified for simple pole placement.  $vD$  are the desired eigenvectors which we can assign because we are using multi-input-multi-output control. The weighting matrix shows how important each element of the desired eigenvector is to the control design. Notice that the length of each eigenvector in  $vD$  is not the length of the state. This is because we don't care about most of the eigenvector values. The matrix  $d$  is used to related the desired eigenvector matrix to the actual states.  $rD$  indexes the rows in  $d$  to the eigenvalues. One column per state. Each row relates  $vD$  to the plant matrix For example, rows 7 and 8 relate column 3 in  $vD$  to the plant. In this case  $vD(1, 3)$  relates to state 2 and  $vD(2, 4)$  relates to state 3.

Now open `ControlDesignPlugin`. Click on the plan box and load `CCVModel.mat`. Now click on the Eigenstructure tab and enter  $\lambda$ ,  $vD$ ,  $d$ ,  $rD$  and  $w$  into the corresponding spots. The GUI will look as shown in [Figure 11.8 on page 104](#).

Push Create. Next push Step. You will see the plot in [Figure 11.9 on the following page](#).

Figure 11.8: Eigenstructure design GUI

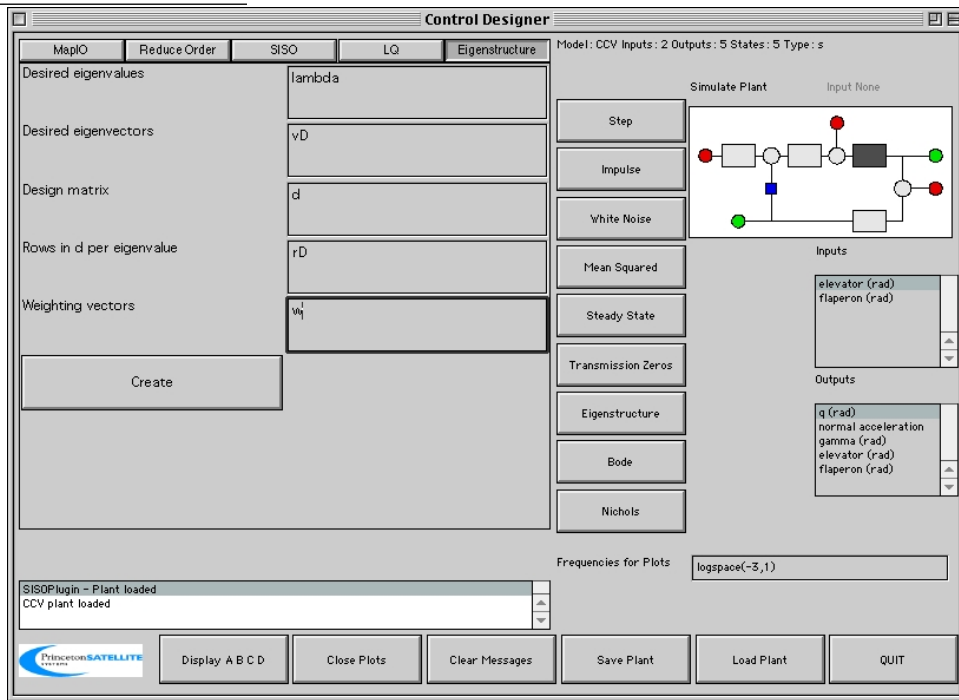
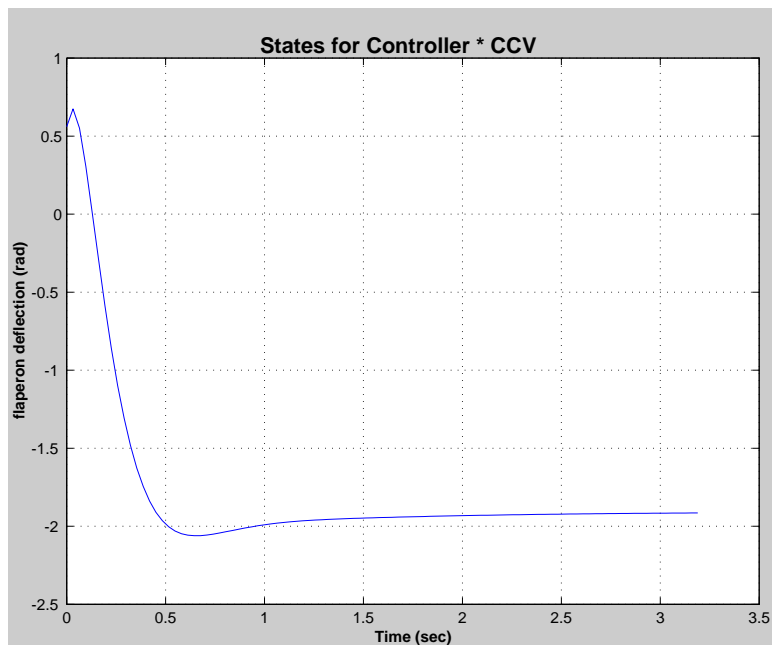


Figure 11.9: Step response with eigenstructure assignment



# CUBESAT

---

This chapter discusses how to use the CubeSat module. This module provides special system modules and mission planning tools suitable for nanosatellite design.

The organization of the module can be seen by typing

```
>> help CubeSat
```

which returns a list of folders in the module. This includes Simulation, MissionPlanning, and Visualization, along with corresponding Demos folders.

The CubeSat integrated simulation includes a simplified surface model for calculating disturbances. The toolbox includes the following features:

- Integrated simulation model including
  - Rigid body dynamics
  - Reaction wheel gyrostat dynamics
  - Point mass
  - Scale height and Jacchia atmospheric models
  - Magnetic dipole, drag, and optical force models
  - Gravity gradient torques
  - Solar cell power model including battery charging dynamics
- Three-axis attitude control using a PID
- Momentum unloading calculations
- Model attitude damping such as with magnetic hysteresis rods
- 2D and 3D visualization including
  - Model visualization with surface normals
  - 2D and 3D orbit plotting
  - 3D attitude visualization
- Ephemeris
  - Convert ECI to Earth-fixed using almanac models
  - Sun vector

- Moon vector
- Advanced orbit dynamics
  - Spherical harmonic gravity model
  - Relative orbit dynamics between two close satellites
- Observation time windows
- Subsystem models
  - Link bit error probabilities
  - Isothermal spacecraft model
  - Cold gas propulsion

## 12.1 CubeSat Modeling

The CubeSat model is generated by `CubeSatModel`, in the `Utilities` folder. The default demo creates a 2U satellite. The model is essentially a set of vertices and faces defining the exterior of the satellite. The function header is below and the resulting model is in Figure 12.1 on page 107. This model has 152 faces.

```
>> help CubeSatModel
Generate vertices and faces for a CubeSat model.
If there are no outputs it will generate a plot with surface normals, or
you can draw the cubesat model using patch:

    patch('vertices',v,'faces',f,'facecolor',[0.5 0.5 0.5]);

type can be '3U' or [3 2 1] i.e. a different dimension for x, y and z.

Type CubeSatModel for a demo of a 3U CubeSat.

This function will populate dRHS for use in RHSCubeSat. The surface
data for the cube faces will be 6 surfaces that are the dimensions of
the core spacecraft. Additional surfaces are added for the deployable
solar panels. Solar panels are grouped into wings that attached to the
edges of the CubeSat.

The function computes the inertia matrix, center of mass and total
mass. The mass properties of the interior components are computed from
total mass and center of mass.

If you set frameOnly to true (or 1), v and f will not contain the
walls. However, dRHS will contain all the wall properties.
-----
Form:
d          = CubeSatModel( 'struct' )
[v, f]     = CubeSatModel( type, t )
[v, f, dRHS] = CubeSatModel( type, d, frameOnly )
Demo:
CubeSatModel
-----

-----
Inputs
-----
type      (1,:) 'nU' where n may be any number, or [x y z]
d         (.) Data structure for the CubeSat
          .thicknessWall    (1,1) Wall thickness (mm)
          .thicknessRail    (1,1) Rail thickness (mm)
          .densityWall      (1,1) Density of the wall material (kg/m3)
          .massComponents   (1,1) Interior component mass (kg)
```

```

.cmComponents      (1,1) Interior components center of mass
.sigma            (3,6) [absorbed; specular; diffuse]
.cd              (1,6) Drag coefficient
.solarPanel.dim   (3,1) [side attached to cubesat, side perpendicular,
thickness]
.solarPanel.nPanels (1,1) Number of panels per wing
.solarPanel.rPanel (3,w) Location of inner edge of panel
.solarPanel.sPanel (3,w) Direction of wing spine
.solarPanel.cellNormal (3,w) Wing cell normal
.solarPanel.sigmaCell (3,1) [absorbed; specular; diffuse] coefficients
.solarPanel.sigmaBack (3,1) [absorbed; specular; diffuse]
.solarPanel.mass  (1,1) Panel mass

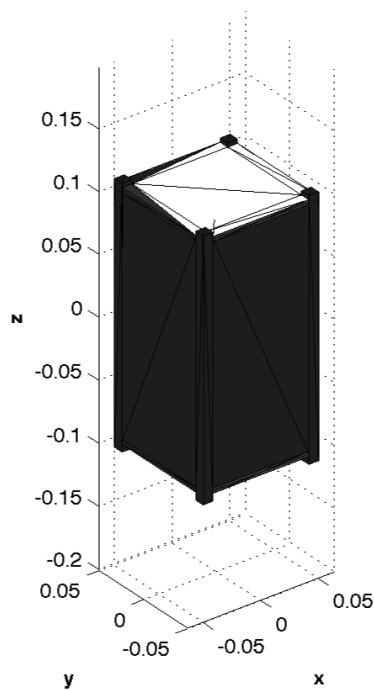
- OR -
t                (1,1) Wall thickness (mm)
frameOnly       (1,1) If true just draw the frame, optional

-----
Outputs
-----
v                (:,3) Vertices
f                (:,3) Faces
dRHS            (1,1) Data structure for the function RHSCubeSat

-----
Reference: CubeSat Design Specification (CDS) Revision 9
-----

```

**Figure 12.1:** Model of 2U CubeSat



For the purposes of disturbance analysis, the CubeSat module uses a simplified model of areas and normals. See `CubeSatFaces`. The `CubeSatModel` function will output a data structure with the surface model in addition to the vertices and faces, which are strictly for visualization. This function does have the capability to model deployable solar wings. The solar areas and normals for power generation are specified separately from the satellite surfaces, as they may be only a portion of any given surface. See `SolarCellPower` for the power model.

The `CubeSatRHS` function documents the simulation data model. The function returns a data structure by default for initializing simulations. The surface data from the `CubeSatModel` function is in the `surfData` and `power` fields. The default data assumes no reaction wheels, as can be seen below since the `kWheels` field is empty. The `atm` data

structure contains the atmosphere model data for use with `AtmJ70`. If this structure is empty, the simpler and faster scale height model in `AtmDens2` will be used instead.

```
>> d = RHSCubeSat
d =
  struct with fields:

      jD0: 2.4552e+06
      mass: 1
      inertia: 0.0016667
      dipole: [3?1 double]
      power: [1?1 struct]
      surfData: [1?1 struct]
      aeroModel: @CubeSatAero
      opticalModel: @CubeSatRadiationPressure
      atm: [1x1 struct]
      kWheels: []
      inertiaRWA: []
      tRWA: []
```

Note in the above structure that the aerodynamics and optical force models are function handles. These functions are designed to accept the surface model data structure within `RHSCubeSat`.

The key functions for modeling CubeSats are summarized in [Table 12.1 on page 108](#).

**Table 12.1:** CubeSat Modeling Functions

<code>AddMass</code>	Combine component masses and calculate inertia and center-of-mass.
<code>InertiaCubeSat</code>	Compute the inertia for standard CubeSat types.
<code>CubeSatFaces</code>	Compute surface areas and normals for the faces of a CubeSat.
<code>CubeSatModel</code>	Generate vertices and faces for a CubeSat model.
<code>TubeSatModel</code>	Generate a TubeSat model.

## 12.2 Simulation

Example simulations are in the `Demos/RelativeOrbit` and `Demos/Simulation` folders. The first has a formation flying demo, `FFSimDemo`. The second has a variety of simulations including an attitude control simulation demo, `CubeSatSimulation`. `CubeSatRWASimulation` demonstrates a set of three orthogonal reaction wheels. `CubeSatGGStabilized` shows how to set up the mass properties for a gravity-gradient boom.

Attitude control loops can be designed using the `PIDMIMO` function and implemented using `PID3Axis`. These are included from the standard `Spacecraft Control Toolbox`.

The orbit simulations, `TwoSpacecraftSimpleOrbitSimulation` and `TwoSpacecraftOrbitSimulation`, simulate the same orbits, but the simple version uses just the central force model and the second adds a variety of disturbances. Both use MATLAB's `ode113` function for integration, so that integration occurs on a single line, without a for loop. `ode113` is a variable step propagator that may take very long steps for orbit sims. The integration line looks like

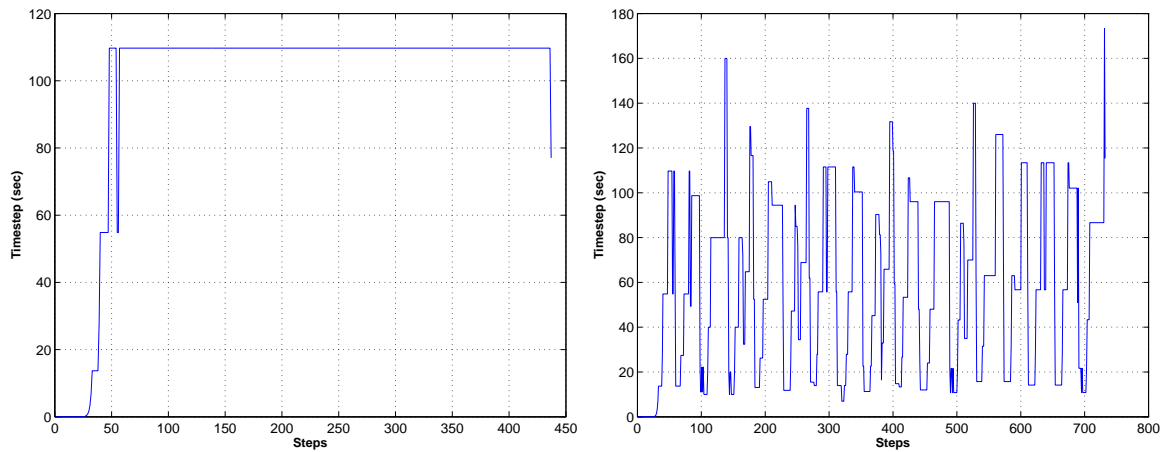
```
% Numerically integrate the orbit
%-----
[t,x] = ode113( @FOrbitMultiSpacecraft, [0 tEnd], x, opt, d );
```

The default simulation length is 12 hours, and the simple sim results in 438 timesteps while the one with disturbances computes 733 steps; [Figure 12.2 on the next page](#) compares the time output of the two examples, where we can see

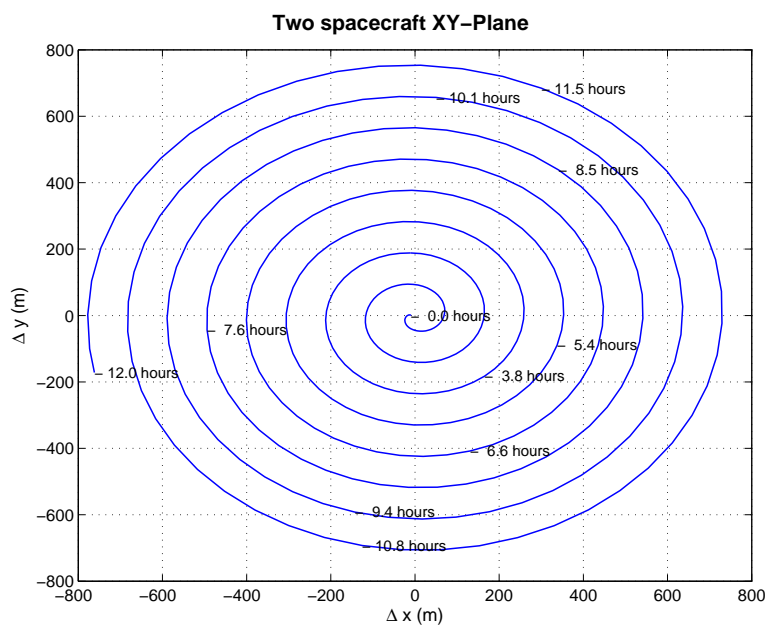


that the simple simulation had mostly a constant step size. Figure 12.3 on the facing page shows the typical orbit results.

**Figure 12.2:** Orbit simulation timestep results, simple on the left with with disturbances on the right.



**Figure 12.3:** Orbit evolution for an initial separation of 10 meters



CubeSatSimulation simulates the attitude dynamics of the CubeSat in addition to a point-mass orbit and the power subsystem. This simulation includes forces and torques from drag, radiation pressure, and an magnetic torque, such as from a torquer control system. Since this simulation can include control, it steps through time discretely in a for loop. RK4 is used for integration. In this case, the integration lines look like

```
for k = 1:nSim
    % Control system placeholder - apply constant dipole
    %-----
    d.dipole    = [0.01;0;0]; % Amp-turns m^2

    % A time step with 4th order Runge-Kutta
    %-----
    x          = RK4( @RHSCubeSat, x, dT, t, d );
```

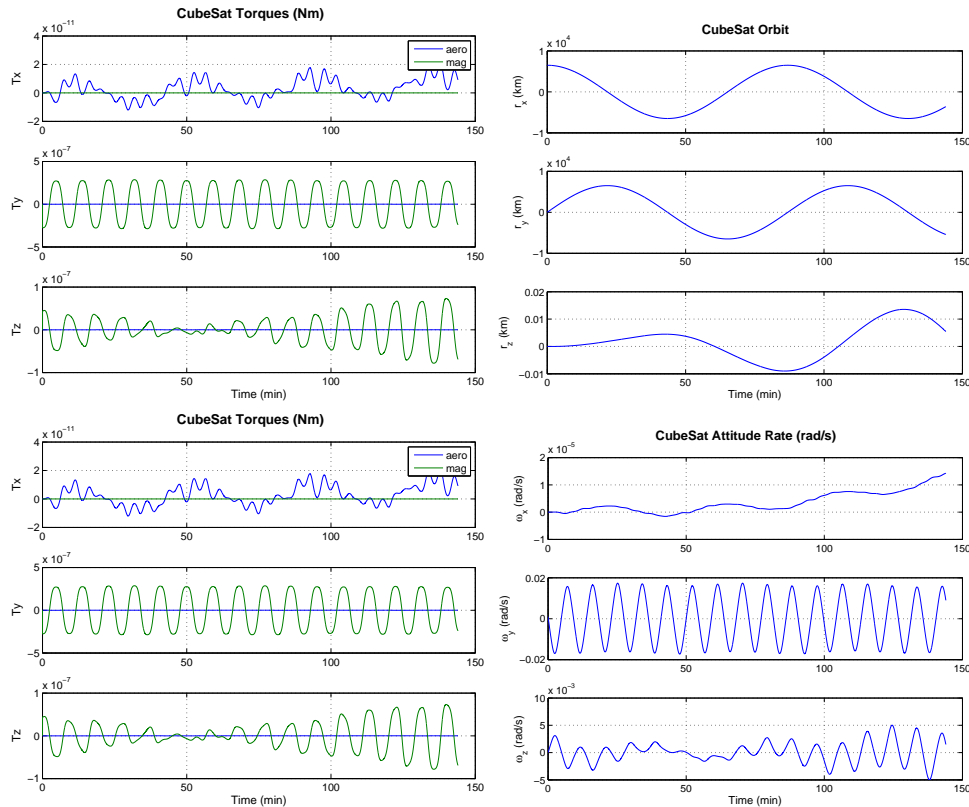
```

% Update plotting and time
%-----
xPlot(:,k+1) = x;
t           = t + dT;

end
    
```

where the control, `d.dipole`, would be computed before the integration at every step for a fixed timestep (1 second). See Figure 12.4 on page 110 for sample results. The simulation takes about one minute of computation time per low-Earth orbit.

Figure 12.4: CubeSatSimulation example results



The key functions used in simulations are summarized in Table 12.2 on the facing page. The space environment calculations from `CubeSatEnvironment` are then passed to the force models in `CubeSatAero` and `CubeSatRadiationPressure`.

Table 12.2: CubeSat Simulation Functions

<code>RHSCubeSat</code>	Dynamics model including power and optional reaction wheels
<code>CubeSatEnvironment</code>	Environment calculations for the CubeSat dynamical model.
<code>CubeSatAero</code>	Aerodynamic model for a CubeSat.
<code>CubeSatRadiationPressure</code>	Radiation pressure model for a CubeSat around the Earth.
<code>CubeSatAttitude</code>	Attitude model with either ECI or LVLH reference
<code>SolarCellPower</code>	Compute the power generated for a CubeSat.

`RHSCubeSat` also models battery charging if the `batteryCapacity` field of the power structure has been appropriately set. Power beyond the calculated consumption will be used to charge the battery until the capacity is reached.

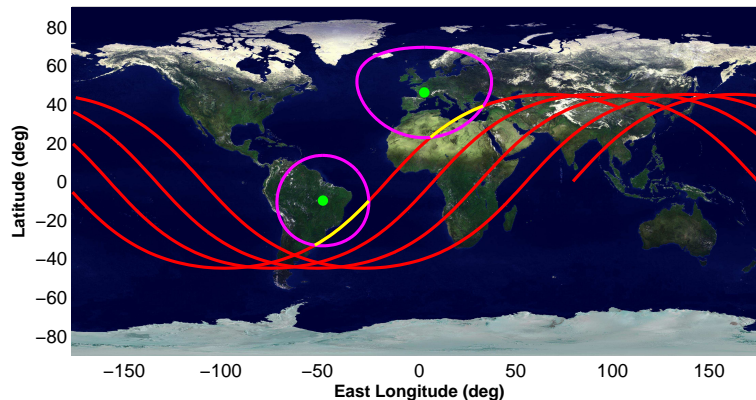
The battery charge is always be the last element of the spacecraft state (after the states of any optional reaction wheels).

## 12.3 Mission Planning

The `MissionPlanning` folder provides several tools for planning a CubeSat mission. These include generating attitude profiles and determining observation windows.

`ObservationTimeWindows` has a built-in demo which also demonstrates `ObservationTimeWindowsPlot`, as shown in Figure 12.5 on the next page. There are two ground targets, one in South America and one in France

**Figure 12.5:** Observation time windows



(large green dots). The satellite is placed in a low Earth orbit, given a field of view of 180 degrees, and the windows are generated over a 7 hour horizon. The figure shows the field of view in magenta and the satellite trajectory segments when the target is in the field of view are highlighted in yellow. This function can operate on a single Keplerian element set or a stored trajectory profile.

`RapidSwath` also has a built-in demo. The demos uses an altitude of 2000 km and a field of view half-angle of 31 degrees. The function allows you to specify a pitch angle between the sensor boresight axis and the nadir axis, in this case 15 degrees. When called with no outputs, the function generates a 3D plot. In Figure 12.6 on the following page we have used the camera controls to zoom in on the sensor cone. The nadir axis is drawn in green and the boresight axis in yellow.

The `AttitudeProfileDemo` shows how to assemble several profile segments together and get the resulting observation windows. The segments can be any of a number of modes, such as latitude/longitude pointing, nadir or sun pointing, etc.

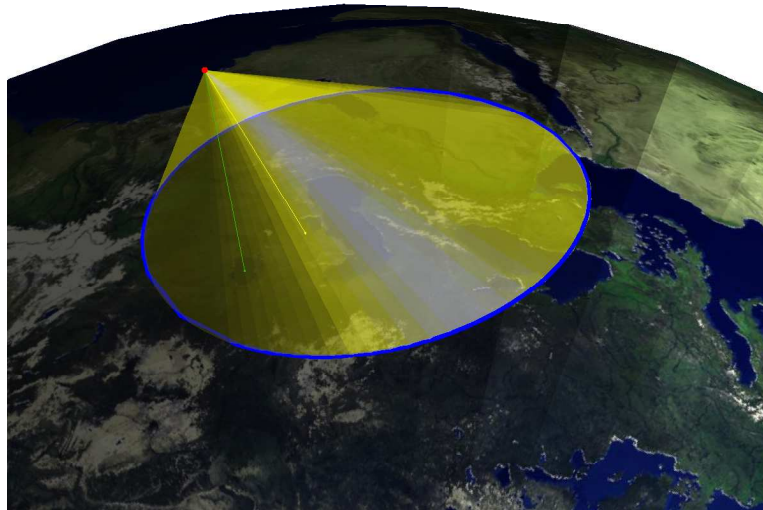
## 12.4 Visualization

The CubeSat Toolbox provides some useful tools to visualize orbits, field of view, lines of sight, and spacecraft orientations.

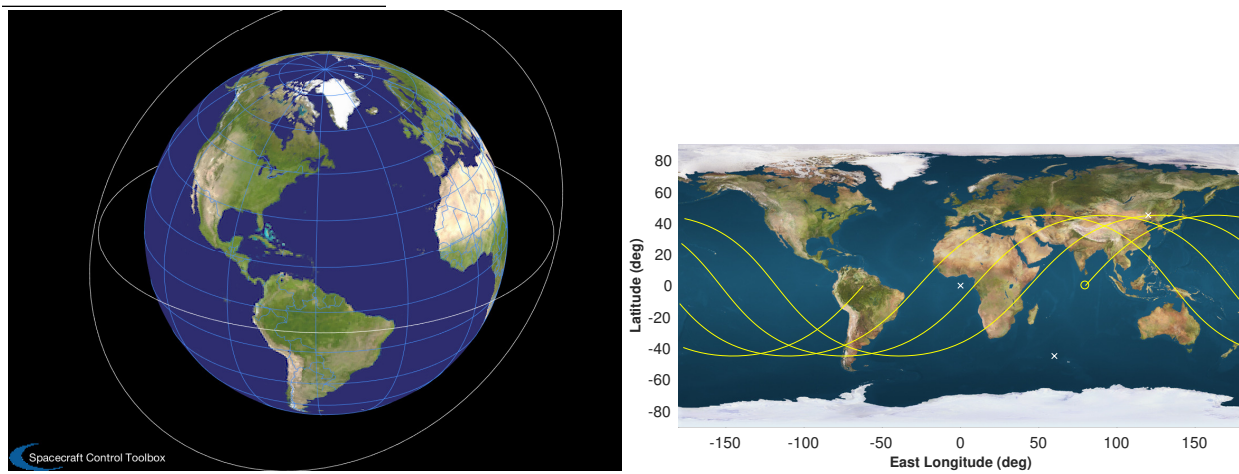
Use `PlotOrbit` to view a spacecraft trajectory in 3D with an Earth map. The `GroundTrack` function plots the trajectory in 2D and has the option of marking ground station locations.

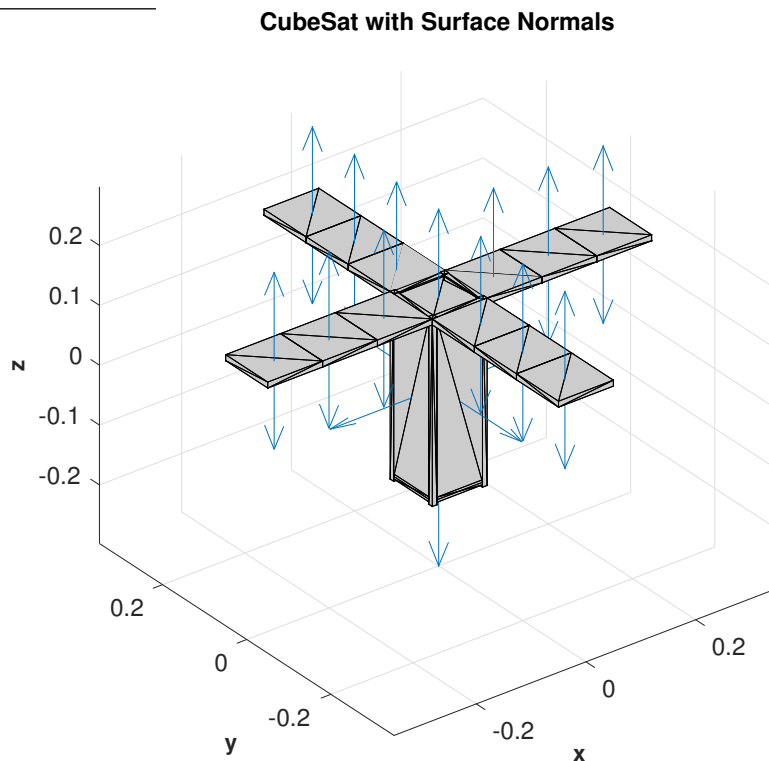
The spacecraft model from `CubeSatModel` can be viewed with surface normals using `DrawCubeSat`. The vertex and face information is not retained with the dynamical data, so `DrawCubeSatSolarAreas` can be used to verify the solar cell areas directly from the RHS data structure.

**Figure 12.6:** RapidSwath built-in demo results

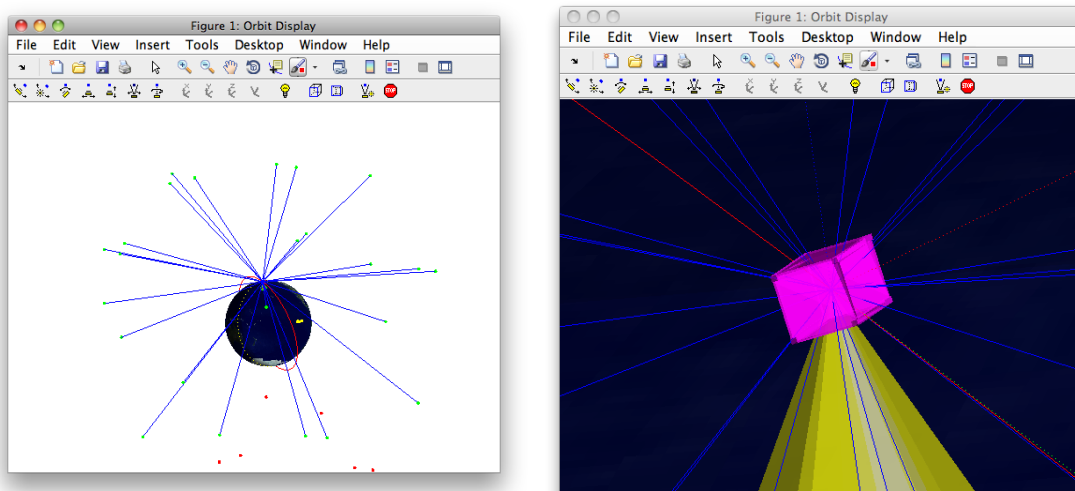


**Figure 12.7:** Orbit visualization with PlotOrbit and GroundTrack



**Figure 12.8:** CubeSat model with deployable solar panels viewed with DrawCubeSat

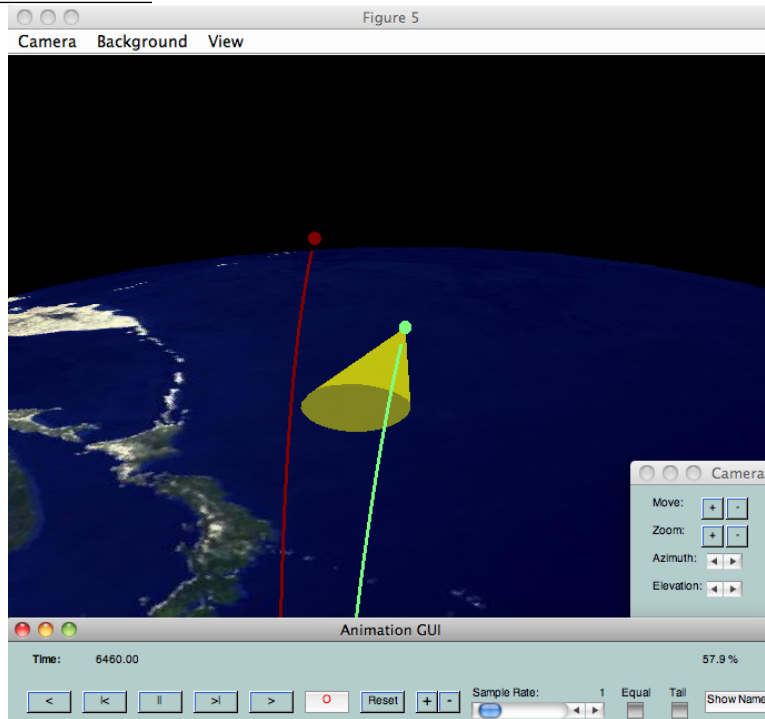
A representative model of the spacecraft may also be viewed in its orbit, along with a sensor cone and lines of sight to all of the visible GPS satellites. Use `DrawSpacecraftInOrbit.m` to generate this view. An example is shown in Figure 12.9 on page 113. The image on the left shows the spacecraft orbit, its sensor cone projected on the Earth, the surrounding GPS satellites, and lines of sight to the visible GPS satellites. The image on the right is a zoomed-in view, where the spacecraft CAD model may be clearly seen.

**Figure 12.9:** Spacecraft visualization with sight lines using `DrawSpacecraftInOrbit`

Run the `OrbitAndSensorConeAnimation.m` mission planning demo to see how to generate simulated orbits, compute sensor cone geometry, and package the data for playback using `PackageOrbitDataForPlayback` and

`PlaybackOrbitSim`. The playback function loads two orbits into the `AnimationGUI`, which provides VCR like controls for playing the simulation forward and backward at different speeds. Set the background color to black and point the camera at a spacecraft, then use the camera controls to move in/out, zoom in/out, and rotate the camera around the spacecraft within a local coordinate frame. The screenshot in Figure 12.10 on page 114 shows the 3D animation window, the `AnimationGUI` playback controls, and the camera controls. Press the Record button (with the red circle) to save the frames to the workspace so that they may be exported to an AVI movie.

**Figure 12.10:** Playback demo using `PlaybackOrbitSim`



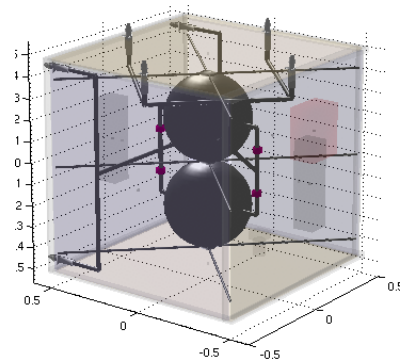
## 12.5 Subsystems Modeling

The CubeSat Toolbox contains select models for key subsystems. The relevant functions and demos are:

- `BatterySizing` - compute power storage requirements given a spacecraft power model and an orbit
- `LinkOrbitAnalysis` - Compute bit error probability along an orbit
- `IsothermalCubeSatDemo` - modeling the CubeSat as a block at a single temperature, calculate the fluctuations over an orbit including the effect of eclipses.
- `CubeSatPropulsion` - Returns the force, torque and mass flow for a cold gas system.
- `DesignMagneticTorquer` - Design an air coil magnetic torquer for a CubeSat.

## PART II

# ATTITUDE CONTROL DESIGN EXAMPLES







# ATTITUDE CONTROL

---

## 13.1 ACS Overview

---

There are attitude control system design examples in SC, SCPro, and the Missions module (*Missions*). These design examples are implemented as demonstration scripts. One, for the Microwave Anisotropy Satellite, MAP, is explained in detail in the next chapter.

## 13.2 Design Examples

---

### 13.2.1 PID Design

`Attitude3D` in `SC/Demos/Attitude` provides a simple example of designing a three-axis PID control system using `PIDMIMO`. A full CAD model is used with the `Disturbances` function in the simulation for calculating disturbances. `REAControl` extends this example to a satellite with thrusters, where equivalent pulsewidths are calculated.

### 13.2.2 Magnetic Control

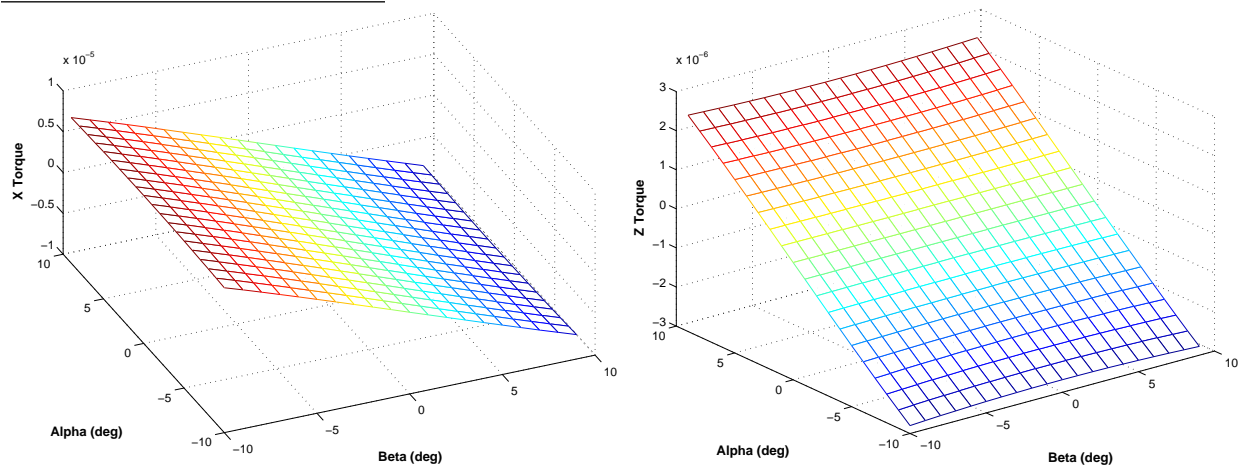
The SC module has both two-axis and three-axis magnetic control functions, `MagControlTwoAxis` and `MagControlThreeAxis`. The three axis version take the integral of the magnetic field over the time vector. Both are functions with built-in demos and are also featured in the demo `MagControl`. See `SC/Demos/MagneticControl`.

### 13.2.3 Microwave Anisotropy Satellite

This is a zero-momentum spacecraft using reaction wheels for control. As part of its mission it must spin and moves its spin axis at a fixed angle with respect to an inertial reference. The script with the MAP implementation is `MAPControlSim` in `SCPro/Demos/ProControl`.

### 13.2.4 Momentum Unloading

This design uses solar pressure on a double-gimballed solar array for two-axis momentum control. The script, `SolarMomentumControl.m` in `ProControl`, maps out the torque produced over a range of gimbal angles before performing the simulation. See [Figure 13.1 on the next page](#).

**Figure 13.1:** Solar pressure torque map for double-gimballed solar array

### 13.2.5 Sun Nadir Pointing

A sun nadir pointing spacecraft points its yaw-axis at the earth and rotates about yaw and steers its solar array so that the solar array normal is aligned with the sun. GPS is an example of a sun-nadir pointing spacecraft. This design uses reaction wheels for attitude control and magnetic torquers for momentum management. See `SunNadirPointingSim.m` in the Missions module.

### 13.2.6 Roll Yaw Control Using a Double Pivot

This design controls the roll and yaw axes of a momentum bias spacecraft by moving a pivoted momentum wheel. It is implemented in the script `RollYawDoublePivot.m` in `Missions/Demos/Comsat`.

### 13.2.7 ComStar

ComStar is a complete geosynchronous bias momentum design. It uses magnetic torquers for roll/yaw control and a fixed momentum wheel for pitch control. During stationkeeping it uses thrusters for roll, yaw and pitch control. When not in stationkeeping mode it uses only an earth sensor which gives roll and pitch information. Yaw is controlled via the roll measurement. Since only the derivative of yaw can be observed in roll it is not possible to control body fixed yaw disturbances. The spacecraft has a flexible solar array.

The design is contained in numerous scripts all found in the `Missions/Demos/Comsat` folder. The following analyses are performed:

- Simulate the AKM firing
- Acquisition sequence
- Generate mass properties
- Simulate a dual spin turn.
- Computes the disturbances including solar, gravity gradient, RF and residual dipole.
- Design earth sensor noise filters
- Demonstrate a pointing budget
- Design a PD controller for pitch
- Tests the MWA pitch loop

- Design and test the low frequency roll/yaw control system.
- Design and test the nutation compensator.
- Compensate the flexible solar array on the ComStar Spacecraft (orbit rate and nutation modes).
- Simulate a spin precession maneuver
- Demonstrate the automatic spin up
- Simulates transfer orbit



---

# A WORKED DESIGN EXAMPLE

---

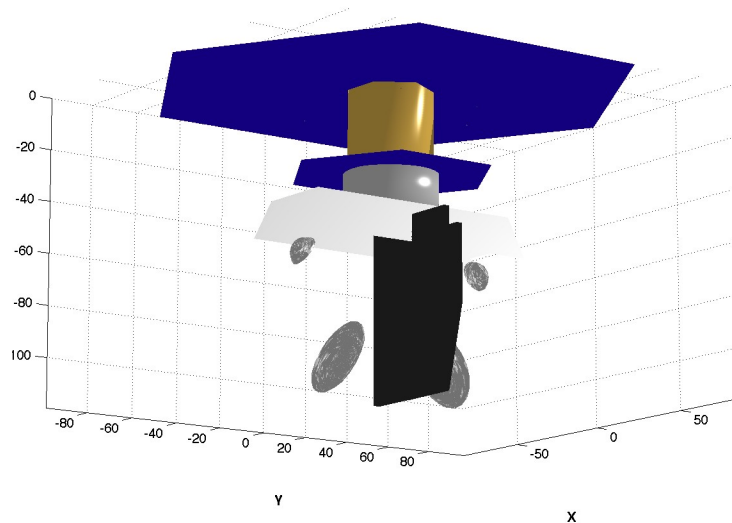
This section walks you through a complete design example, that of the Microwave Anisotropy Satellite normal and acquisition modes.

### 14.1 The MAP Mission

---

The Microwave Anisotropy Probe is a three-axis controlled spacecraft. It employs reaction wheels for nominal attitude control with star trackers and gyros for sensing. Ten hydrazine thrusters are available for backup attitude control, orbit adjust maneuvers and momentum unloading. The rendering below was created using the toolbox CAD tools in the script `BuildMAP`.

**Figure 14.1:** The Microwave Anisotropy Probe



## 14.2 Control Modes

---

The MAP spacecraft requires three control modes:

- Mission control mode using reaction wheels with thrusters for momentum unloading. This mode would also include the acquisition function.
- Backup mission control mode with thrusters for three-axis attitude control
- Orbit adjust mode using thrusters for three-axis attitude control. The reaction wheels are kept in a tachometer mode during orbit changes.

Optionally, a safe hold mode could be added.

## 14.3 Sensing and Actuation

---

Nominally, MAP uses gyros for attitude determination and a star tracker to correct for gyro drift and for absolute attitude information. The gyro and star tracker data would be integrated using an extended iterated Kalman filter. A backup mode using just the star tracker would also be available. During most of the mission the reaction wheels are used for attitude control. Secular momentum growth due to solar pressure would be controlled using the thrusters. Momentum unloading could be autonomous or ground commanded. During orbit adjust modes the thrusters would be used for attitude control since the disturbances due to the orbit adjust thrusters would quickly saturate the reaction wheels.

## 14.4 Control System Design

---

The control system design is a modification of Princeton Satellite Systems' standard three axis control system. The control system employs tachometer inner loops for the three reaction wheels. The outer loops are two Proportional Integral Differential (PID) loops for the transverse axes and a Proportional Integral (PI) loop for the spin axis. The integral term in the PID controllers insures accurate tracking of the spin axis target. The inner reaction wheel loops are PI controllers. The outer loops output an acceleration demand that is converted to a wheel speed demand and passed to the reaction wheel tachometer loops. The inner loops insure proper reaction wheel response regardless of bearing friction magnitude and uncertainty. A sampling rate of 4 Hz was chosen for the digital implementation of this control system.

## 14.5 Simulation Results

---

Normal mode acquisition was simulated using the `TMAPSim` script. The mass properties and the actuator characteristics for the baseline MAP spacecraft were not available so generic parameters were used in the simulation. The spacecraft model is for a gyrostat with three reaction wheels. All nonlinear terms are included. The simulation models do not include sensor noise or external disturbances. The spacecraft is initially spinning at 0.464 rpm with the spin axis aligned with the inertial Z axis. The desired precession angle command of 22.5 deg is fed into the control loops through a low pass filter to eliminate transients. The gains of the filter are chosen so that acquisition is completed within ten minutes.

## 14.6 The TMAPSim Script

In this section we will walk through the TMAPSim script (now available as MAPControlSim in recent versions of the toolbox). Once you understand how it was written you will be able to easily design and build your own control systems and simulations. Listing 14.1 on page 125 shows the header which defines a few useful constants. The global variable is used to control the time GUI. We routinely close all windows.

**Listing 14.1:** File header

*TMAPSim.m*

```

1  %-----
2  %   Implements and simulates the MAP normal mode control. This is not the
3  %   NASA design but one done in house. It shows how to integrate control
4  %   design and simulation into a single script.
5  %
6  %       This script also generates an STK attitude file.
7  %-----
8  %   Copyright (c) 1996-2001 Princeton Satellite Systems, Inc. All rights reserved.
9  %-----
10
11 % Clean up the workspace
12 % -----
13 close all
14
15 % Global for the time GUI
16 %-----
17 global simulationAction
18 simulationAction = '┘';
19
20 % Constants
21 % -----
22 degToRad = pi/180;
23 radToDeg = 180/pi;
24 rPMTorRPS = pi/30;

```

*TMAPSim.m*

The sampling interval is both for the controller and the simulation. This is fine if your controller effectively sets the highest frequencies in the simulation. If your model includes higher frequency terms (which would cause jitter) you would need to use a smaller integration step. The rest of the quantities are used to cause the simulation plot less frequently than once per integration step.

**Listing 14.2:** Sampling intervals

*TMAPSim.m*

```

1  % The control sampling period and the simulation integration time step
2  % -----
3  tSamp      = 0.25;
4
5  % Number of sim steps
6  % -----
7  nSim      = 4*3600;
8
9  % Plot every nPMax steps
10 % -----
11 nPMax     = 40;
12 nPlot     = nSim/nPMax;
13
14 % Print the time to go message every nTTGo steps
15 % -----
16 nTTGo     = 1000;

```

*TMAPSim.m*

Listing 14.3 on page 126 defines the spacecraft mass properties. This set is not based on the actual MAP properties but is for demonstration purposes only. We are using a gyostat (rigid body + wheels) model that allows for any number of reaction wheels so we must include the spin axis inertias of the wheels. We will use three orthogonal wheels.

**Listing 14.3:** Mass properties

TMAPSim.m

```

1 % Spacecraft Inertias
2 % -----
3 inr      = [2000,0,0;0,2000,0;0,0,4000];
4 inrRWA   = 1;
5 inrW     = [inrRWA,inrRWA,inrRWA];
6 invInr   = inv(inr);
7 tDist    = [0;0;0];
8
9 % Wheel spin axis unit vectors
10 % -----
11 uW       = eye(3);

```

TMAPSim.m

First design the inner tach loops for the reaction wheels. Our attitude control loops will actually command rate changes to the wheels. The inner loops will handle uncertainty in friction, wheel motor scale factors, etc. PIDesign uses pole placement to design a PI loop. In this example we will implement the controllers in delta form so that state space equations produce deltas to the control commands, not the actual value.

**Listing 14.4:** Tachometer loop design

TMAPSim.m

```

1 % -----
2 % Design the control loops
3 % -----
4
5 % RWA Tach loops
6 % -----
7 zeta      = 0.7071; % Damping ratio
8 wN       = 1.0;    % Closed loop undamped natural frequency
9
10 [aTL,bTL,cTL,dTL] = PIDesign( zeta, wN, inrW(3), tSamp, 'Delta' );

```

TMAPSim.m

Next, design the attitude loops. We are only controlling the spin axis orientation so we only need two loops, roll and pitch (we are spinning about yaw.) PIDMIMO designs a PID controller with a rate filter.

**Listing 14.5:** ACS design

TMAPSim.m

```

1 % Attitude Loops
2 % -----
3 zeta      = 0.7071; % Damping ratio
4 wN       = 0.5;    % Closed loop undamped natural frequency
5 wR       = 4.0;    % Rate filter break frequency
6 tau      = 50;    % Integrator time constant
7
8 [aRoll ,bRoll, cRoll, dRoll] = PIDMIMO( inr(1,1), zeta, wN, tau, wR, tSamp, 'Delta');
9 [aPitch,bPitch,cPitch,dPitch] = PIDMIMO( inr(2,2), zeta, wN, tau, wR, tSamp, 'Delta');

```

TMAPSim.m

Listing 14.6 on page 126 gives the yaw rate loop that will maintain the MAP spin rate. This uses the same PI controller as the reaction wheels but with a slower time constant.

**Listing 14.6:** Rate loop

TMAPSim.m

```

1 % Rate Loops
2 % -----
3 zeta      = 1.0; % Damping ratio
4 wN       = 0.5; % Closed loop undamped natural frequency
5
6 [aYaw, bYaw, cYaw, dYaw] = PIDesign( zeta, wN, inr(3,3), tSamp, 'Delta' );

```

TMAPSim.m

Now that the controller design is complete it is time to run a simulation to test the system. Of course we should also do frequency domain analysis of the entire controller. Even though we used SISO design techniques this is really a



MIMO design. The controller has fast inner loops to control the wheels and slower outer loops to control the attitude. Several tools exist in SCT for analyzing such systems.

The first step is to initialize the control system. We've turned off the momentum management system.

**Listing 14.7:** Initialize the control system

*TMAPSim.m*

```

1  % Initialize the control system
2  % -----
3  xTL          = zeros(3,1);
4  xRoll       = [0;0];
5  xPitch      = [0;0];
6  xYaw        = 0;
7  tC          = [0;0;0];
8  spinRate    = 0.464*rPMTORPS;
9  precRate    = 2*pi/3600;
10 precAngleDemand = 22.5;
11 precAngle    = 0;
12 precAngleGain = 0.995;
13
14 % Momentum Management gain
15 % -----
16 kMM          = 0.000;
17
18 % The control distribution matrix converts
19 % torque demand to angular acceleration demand
20 % -----
21 aRWA         = eye(3)/inrW(3);
22 wRWAC        = [0;0;0];

```

*TMAPSim.m*

The following code sets up the simulation. We allocate memory for plotting arrays to save processing time. We also set up the time statistics function which will display time statistics.

**Listing 14.8:** Set up the simulation

*TMAPSim.m*

```

1  % Plotting arrays
2  % -----
3  cPlot       = zeros(3,nPlot);
4  hPlot       = zeros(3,nPlot);
5  tPlot       = zeros(1,nPlot);
6  xPlot       = zeros(10,nPlot);
7  zPlot       = zeros(3,nPlot);
8  rPlot       = zeros(3,nPlot);
9  pPlot       = zeros(2,nPlot);
10 ePlot       = zeros(2,nPlot);
11 wPlot       = zeros(3,nPlot);
12 yPlot       = zeros(3,nPlot);
13
14 % Initial conditions
15 % -----
16 %           q           w           wRWA
17 x          = [[1;0;0;0];[0;0;spinRate];[0;0;0]];
18
19 dTSim      = tSamp;
20 t          = 0;
21 nP         = 0;
22 kP         = 0;
23 tW         = zeros(3,1);
24 roll       = 0;
25 pitch      = 0;
26 yaw        = 0;
27
28 % Initialize the time display
29 %-----
30 tToGoMem.lastJD      = 0;
31 tToGoMem.lastStepsDone = 0;

```

```

32 tToGoMem.kAve          = 0;
33 ratioRealTime          = 0;
34 [ ratioRealTime, tToGoMem ] = TimeGUI( nSim, 0, tToGoMem, 0, tSamp, 'TMAPSim' );

```

TMAPSim.m

This is the start of the simulation loop. The first function displays the status message in the command window.

**Listing 14.9:** Start of the simulation loop

TMAPSim.m

```

1 % Run the simulation
2 % -----
3 for k = 1:nSim
4
5 % Display the status message
6 % -----
7 [ ratioRealTime, tToGoMem ] = TimeGUI( nSim, k, tToGoMem, ratioRealTime, tSamp );

```

TMAPSim.m

We are using very simple sensor models for this demonstration. Just the true states without sensor dynamics, scale factor errors, noise or any other effects that are quite important in practice.

**Listing 14.10:** Sensor models

TMAPSim.m

```

1 % -----
2 % Sensors
3 % -----
4
5 % RWA Tachometer
6 % -----
7 wTach = x(8:10);
8
9 % Gyros
10 % -----
11 wCore = x(5:7);
12
13 % Attitude
14 % -----
15 qIToB = x(1:4);

```

TMAPSim.m

Finally, we get to the control system implementation. The first part is the momentum management system which is just a proportional controller.

**Listing 14.11:** Control system

TMAPSim.m

```

1 % -----
2 % The Attitude Control System
3 % -----
4
5 % Momentum Management
6 % -----
7
8 % Neglect the rate errors in the body component-assume exact tracking
9 % -----
10 hTotal = QForm( qIToB, inr*[wCore(1);wCore(2);wCore(3)] + inrRWA*uW*(wTach+wCore) );
11
12 % Proportional controller for momentum
13 % We could feedforward this to the controller
14 % -----
15 tMM = QForm( qIToB, -kMM*hTotal );

```

TMAPSim.m

We next compute the errors. These are what we will be controlling. Notice that the commanded precession angle is filtered. This makes the system a two-degree-of-freedom control system and prevents the operator from putting in a command that would saturate the actuators.

**Listing 14.12:** Error computation

TMAPSim.m

```

1  % Compute the errors
2  % -----
3  precAngle = precAngleGain*precAngle + (1-precAngleGain)*precAngleDemand;
4  cP        = CosD(precAngle);
5  sP        = SinD(precAngle);
6  xT        = [sP*cos(precRate*t); sP*sin(precRate*t); cP];
7  xTB       = QForm( qIToB, xT );
8  rollError = asin(xTB(2));
9  pitchError = -asin(xTB(1));
10 yawError  = wCore(3) - spinRate;

```

TMAPSim.m

These loops control the attitude and spin rate. The output is torque demand ( $t_C$ ).

**Listing 14.13:** Attitude loop

TMAPSim.m

```

1  % The attitude control loops
2  % -----
3  tC(1) = -cRoll*xRoll - dRoll*rollError;
4  xRoll = xRoll + aRoll*xRoll + bRoll*rollError;
5
6  tC(2) = -cPitch*xPitch - dPitch*pitchError;
7  xPitch = xPitch + aPitch*xPitch + bPitch*pitchError;
8
9  tC(3) = -cYaw*xYaw - dYaw*yawError;
10 xYaw = xYaw + aYaw*xYaw + bYaw*yawError;

```

TMAPSim.m

We convert torque into a demand on the wheels which we integrate to get wheel speed demand.

**Listing 14.14:** Compute wheel speed demand

TMAPSim.m

```

1  % Convert torque demand to RWA angular acceleration demand
2  % -----
3  wDRWA = -aRWA*tC;
4
5  % Integrate to get wheel speed demand
6  % -----
7  wRWAC = wRWAC + tSamp*wDRWA;

```

TMAPSim.m

The wheels are commanded through the tach loops.

**Listing 14.15:** RWA tach loops

TMAPSim.m

```

1  % The RWA Tach Loops
2  % -----
3  wError = wTach - wRWAC;
4  tRWA   = -dTl*wError - cTl*xTL;
5  xTL    = xTL + aTl*xTL + bTl*wError;

```

TMAPSim.m

This is the simulation. We are using the gyrostat model for this simulation. The gyrostat model FGs is called by RK4. RK4 is a Fourth-Order Runge-Kutta method that works well for most spacecraft control simulation problems. It does not do automatic error correction or computation. This means that you cannot tell how accurately it is integrating the equations of motion from a single run. To check on numerical errors you should run your simulation at half the time step and compare the results. If they are significantly different you will need to go to a smaller step size. The RK45, with somewhat greater overhead, can be used in situations where you need to control the precision of your simulation.

**Listing 14.16:** Simulation

TMAPSim.m

```

1  % -----
2  % Update the equations of motion
3  % -----

```

```

4 x      = RK4('FGs',x,dTSim,t,inr,invInr,tDist+tMM,inrW,uW,tRWA');
5 t      = t + dTSim;

```

TMAPSim.m

In the following listing we are saving information for the plots. The code under the comment “time control” allows you to control the script from the time GUI.

Listing 14.17: Saving plot information

TMAPSim.m

```

1  % Plotting
2  % -----
3  if nP == 0,
4      kP      = kP + 1;
5      xPlot(:,kP) = x;
6      tPlot(1,kP) = t;
7      cPlot(:,kP) = tC;
8      hPlot(:,kP) = hTotal;
9      zPlot(:,kP) = tMM;
10     xI      = QTForm(qIToB,[0;0;1]);
11     rPlot(:,kP) = xI;
12     pPlot(:,kP) = acos([xI(3);xT(3)]);
13     ePlot(:,kP) = [rollError;pitchError];
14     wPlot(:,kP) = xT;
15     yPlot(:,kP) = xTB;
16     nP      = nPMax - 1;
17 else
18     nP      = nP - 1;
19 end
20
21 % Time control
22 % -----
23 switch simulationAction
24 case 'pause'
25     pause
26     simulationAction = '_';
27 case 'stop'
28     return;
29 case 'plot'
30     break;
31 end
32
33 end

```

TMAPSim.m

After the simulation is done we plot the results.

Listing 14.18: Plotting

TMAPSim.m

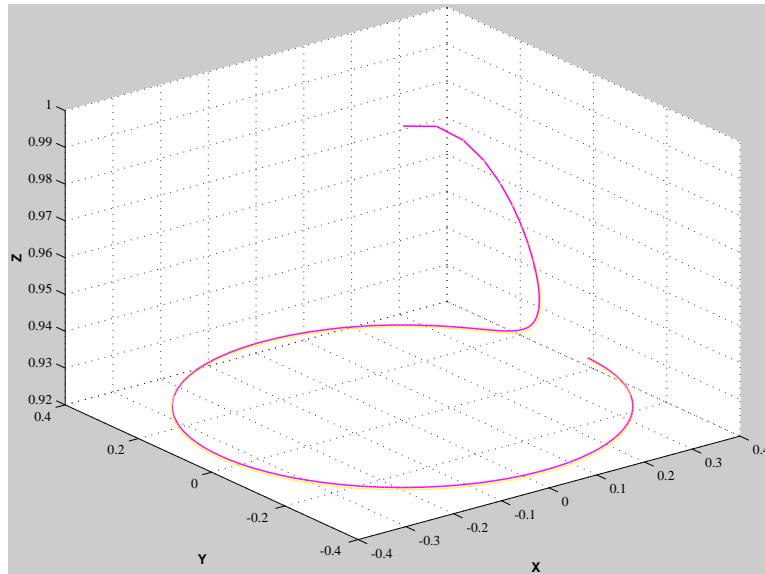
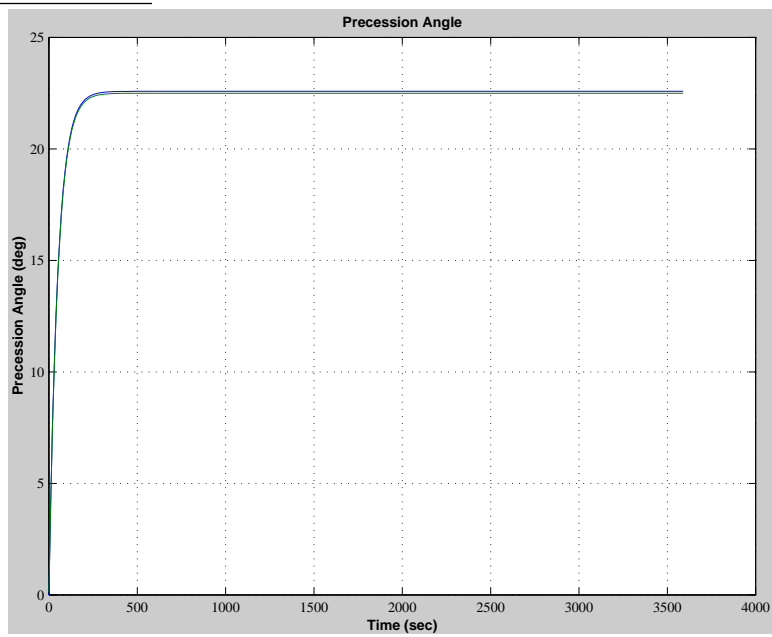
```

1 Plot2D(tPlot,xPlot( 1: 4, j), 'Time_(sec)', ['Qs';'Qx';'Qy';'Qz'],'Quaternion')
2 Plot2D(tPlot,xPlot( 5: 7, j), 'Time_(sec)', ['Wx';'Wy';'Wz'],'Body_Rates')
3 Plot2D(tPlot,xPlot( 8:10, j), 'Time_(sec)', ['W1';'W2';'W3'],'Reaction_Wheels')
4 Plot2D(tPlot,cPlot(:, j), 'Time_(sec)', ['X';'Y';'Z'],'Control_Torque_Demand')
5 Plot2D(tPlot,hPlot(:, j), 'Time_(sec)', ['X';'Y';'Z'],'Body_Momentum')
6 Plot2D(tPlot,zPlot(:, j), 'Time_(sec)', ['X';'Y';'Z'],'Momentum_Management_Torque')
7 Plot2D(tPlot,pPlot(:, j)*radToDeg, 'Time_(sec)', 'Precession_Angle_(deg)', 'Precession_Angle')
8 PlotV([rPlot(:, j);wPlot(:, j)], 'X', 'Y', 'Z', 'Spin_Axis_and_Target')
9 Plot2D(tPlot,ePlot(:, j)*radToDeg, 'Time_(sec)', ['Roll_(deg)';'Pitch_(deg)'], 'Attitude_Errors')

```

TMAPSim.m

Figure 14.2 on page 131 shows the results of `PlotV`. It shows the spin axis unit vector. It starts as  $[0;0;1]$  (the highest point on the plot) and as the spacecraft acquires picks up  $y$  and  $x$  components until it traces out a circle in the  $xy$ -plane. Figure 14.3 on page 131 shows the precession of the unit vector.

**Figure 14.2:** Motion of the unit vector**Figure 14.3:** Precession of the unit vector

## 14.7 Summary

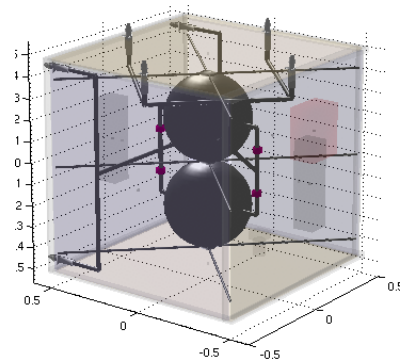
---

As can be seen from the plots, the control system successfully acquires and tracks the target. The overall pointing accuracy is better than 0.2 deg. All of the plots were generated using Spacecraft Control Toolbox functions. All of the design functions and simulated functions fit in a single MATLAB script. The second plot shows the unit vector for the spin-axis of the spacecraft in three dimensional space. Initially, it is aligned with the z-axis so its x and y components are zero. After acquisition the x and y-axes trace out a circle in the xy-plane and the z value is constant.

The control system presented here is only part of the overall control architecture that includes not only the other modes but also the attitude determination function, the momentum unloading system and the telemetry and command functions. Although the spacecraft is mostly single string, a fault detection and isolation system might also be included to protect the spacecraft against temporary problems.

## PART III

# ORBIT PROPAGATION AND MANEUVERS







# ORBIT MODULE

---

## 15.1 Overview

---

This module (*Orbit*) contains software for orbit mechanics, orbit coordinate transformations, maneuver analysis and planning, and simulation. There is a special GUI for performing high-fidelity orbit propagations using MATLAB's `ode113` propagator with over 30 stopping conditions. The gravity functions allow user-selectable gravity models such as GEM-T1, JGM-2, and WGS-84. A lunar gravity model is included as well.

The maneuver functions encompass a variety of types, including:

- Impulsive transfer analysis
- Low-thrust spirals and simulations
- Lambert law solver for large orbit changes
- Circumnavigation and glideslopes using pulses
- Bielliptic and Hohmann transfers
- Stationkeeping analysis

Additional analysis tools include a two-line element propagator, linearized orbit model, Gauss' variational equations, fuel budgets, and  $J_2$  effects. Equinoctial and Keplerian orbit mechanics are supported as well as cartesian, cylindrical, and spherical frames. If a tool or algorithm you want isn't available, please let us know!



# ORBIT PROPAGATION EXAMPLES

---

This chapter gives examples from textbooks. One additional example showing the effects of the ode113 integration tolerance parameters is given at the end. Each example is included as a demo file in the demo directory.

## 16.1 Low Thrust Orbit Raising

---

### Reference

Wiesel, W. E. (1989.) Spaceflight Dynamics. McGraw-Hill. pp. 89-90.

### Introduction

If a small thrust is tangential to the orbit and its magnitude is small and constant, the semi-major axis time derivative is given by Wiesel in Equation (3.61)

$$\frac{da}{dt} = \frac{2}{\sqrt{\mu}} a^{3/2} A \quad (16.1)$$

where  $a$  is the semi major axis and  $A$  is the tangential acceleration. The solution is

$$a^{-1/2} = a_0^{-1/2} - \frac{A}{\sqrt{\mu}}(t - t_0) \quad (16.2)$$

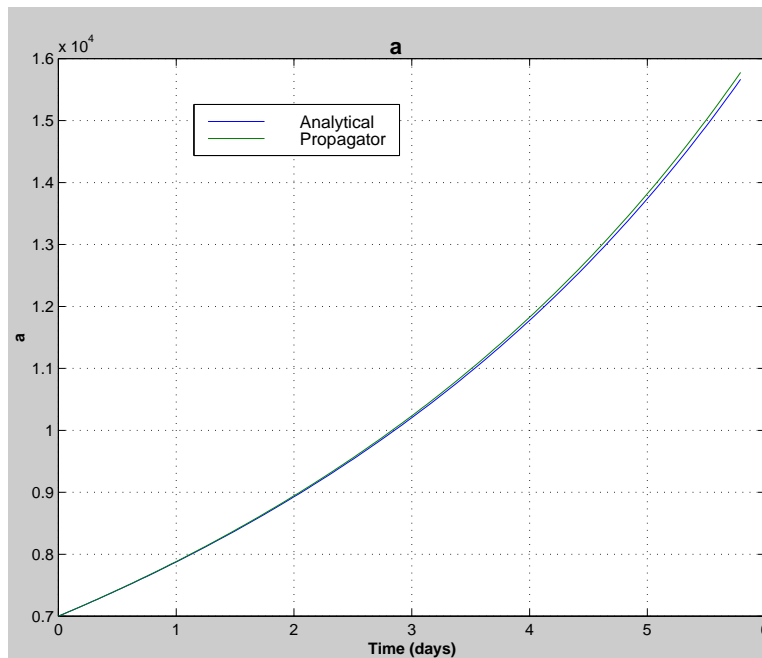
### Results

Type

LowThrustDemo

in the command window. The last plot will show the difference between the propagator and the analytical model (Figure 16.1 on page 138).

The results are very close showing both the accuracy of the integrator when a perturbation is introduced and the

**Figure 16.1:** Comparison of an analytical low thrust model with the propagated solution.

accuracy of the approximation. An important point is that the numerical result is very dependent on the tolerances chosen. When the default relative tolerance was used, the numerical solution showed a much greater increase in semi-major axis. Consequently, before certifying a result you should always make runs with different tolerances. Ideally, you should find the closest analytical solution and choose the tolerance based on how close the numerical results come to the analytical.

## 16.2 Reentry Due to Drag

### Reference

Wiesel, W. E. (1989.) Spaceflight Dynamics. McGraw-Hill. pp. 83-84.

### Introduction

Given an exponential atmosphere, the time derivative of the semi-major axis is

$$\frac{da}{dt} = -\sqrt{\mu a} \frac{C_D A}{m} \rho_0 e^{-(a-R_E)/h} \quad (16.3)$$

where  $C_D$  is the drag-coefficient,  $A$  the effective area,  $m$  the spacecraft mass,  $R_E$  the radius of the earth and  $h$  the atmospheric scale height.

This can be analytically integrated. However, a simpler form emerges if the dependent variable is changed from  $a$  to  $H$ , the altitude, and  $H$  is recognized as being small relative to  $R_E$ . The solution to the differential equation then becomes

$$H = h \ln \left( e^{H_0/h} - \frac{\rho_0 \sqrt{\mu R_E} \left( \frac{C_D A}{m} \right)}{h} (t - t_0) \right) \quad (16.4)$$

This equation does not account for the motion of the earth's atmosphere. In this example we use the Standard Atmosphere and select the scale height for the analytical model so that the atmospheric density given by the exponential model is the same as the density given by the standard atmosphere.

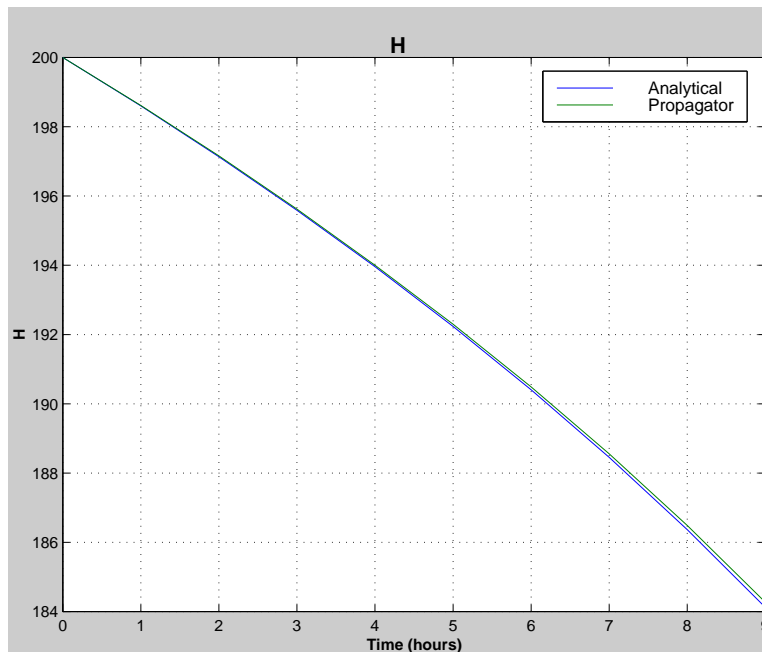
## Results

Type

DragDemo

As expected, the propagator shows slightly less decay due to the rotation of the atmosphere (Figure 16.2 on page 139).

**Figure 16.2:** Reentry Due to Drag



## 16.3 Topex Ephemerides

### Reference

Vallado, D. A. and W. D. McClain. (1997.) Fundamentals of Astrodynamics and Applications. McGraw-Hill. p.536.

## Introduction

In this example we compare Topex data with Keplerian orbit data and the results of the `PropagateOrbitPlugin`. Topex is a sun-nadir pointing spacecraft in a 1300 km altitude orbit. The perturbations of significance will be

- Gravitational harmonics
- Solar pressure
- Sun and moon perturbations

A sun-nadir pointing spacecraft maneuvers about yaw and rotates its solar array so that the solar array is always pointing at the sun. This fits in well with the `FSolar` model which assumes that the spacecraft (approximated as a flat plate) normal is aligned with the sun vector. That aside, the optical surface properties are unknown and the geometry of the rest of the spacecraft is not known which will limit the accuracy of the model.

## Results

Type

TopexDemo

The Topex ECI ephemerides are shown in the Figure 16.3 on page 140. The range of  $R$  and  $V$  do not vary much over the period of the orbit.

**Figure 16.3:** Topex Ephemeris Cycle 10

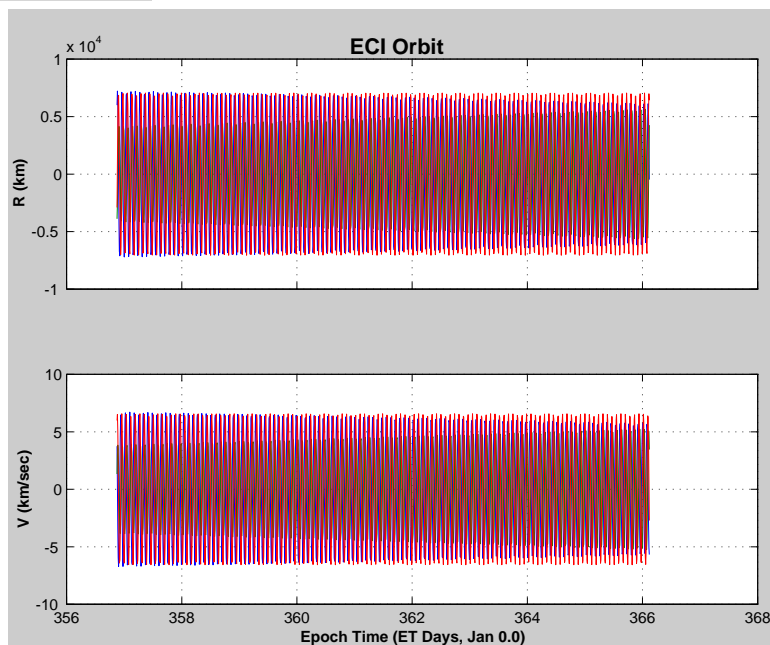


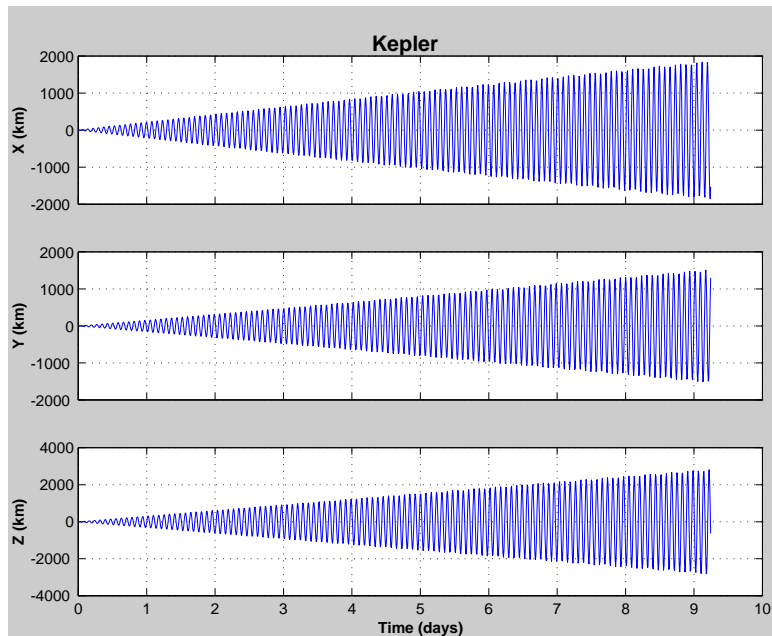
Figure 16.4 on page 141 shows the error between the Topex orbit and a perfect Keplerian orbit. The osculating elements at the start of the Topex file were used for the Kepler propagator. The osculating elements are given in Table 16.1 on page 141. The differences are enormous by the end of the simulation. Essentially, the Topex orbit is completely out of phase with the spherical model.

Figure 16.5 on the facing page shows the difference between the Topex Ephemeris and `PropagateOrbitPlugin` when just the GEM-T1 spherical earth model was used. The results (as expected) are the same.

**Table 16.1:** Topex Osculating Elements at the Beginning of the Simulation.

Element	Value
$a$	7719.2 km
$i$	1.1528 rad
$\omega$	-0.39392 rad
$\Omega$	-0.94544 rad
$e$	0.00025586
$M$	0.52091 rad

**Figure 16.4:** Difference Between the Topex Ephemeris and a Keplerian Orbit



**Figure 16.5:** Difference Between the Topex Ephemeris and a Propagated Spherical Earth Model

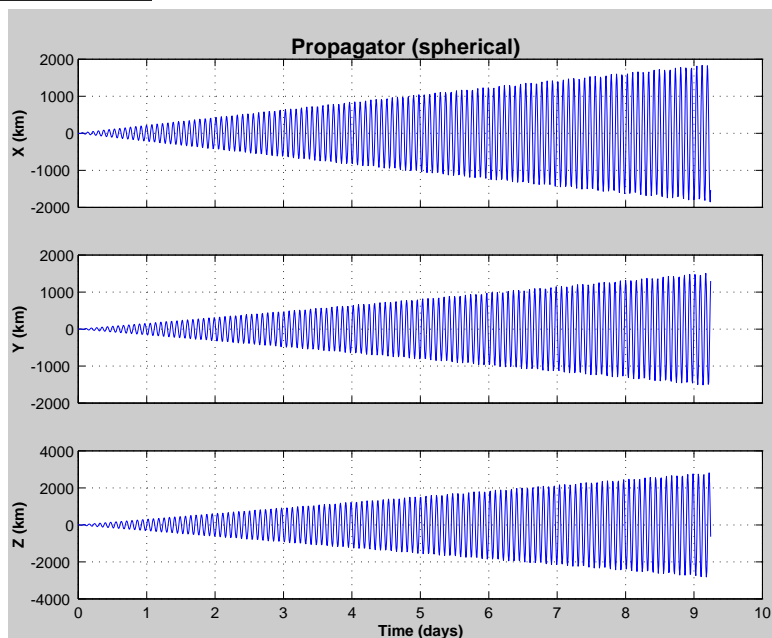


Figure 16.6 on page 142 shows the Topex Orbit in 3D referenced to the earth-fixed frame.

**Figure 16.6:** Topex Orbits in 3D

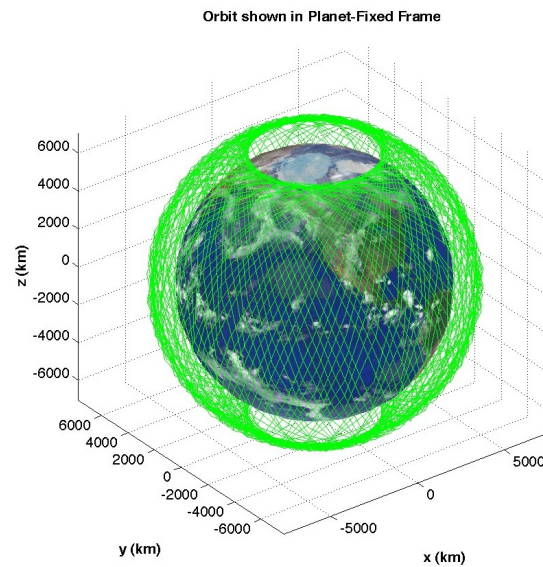
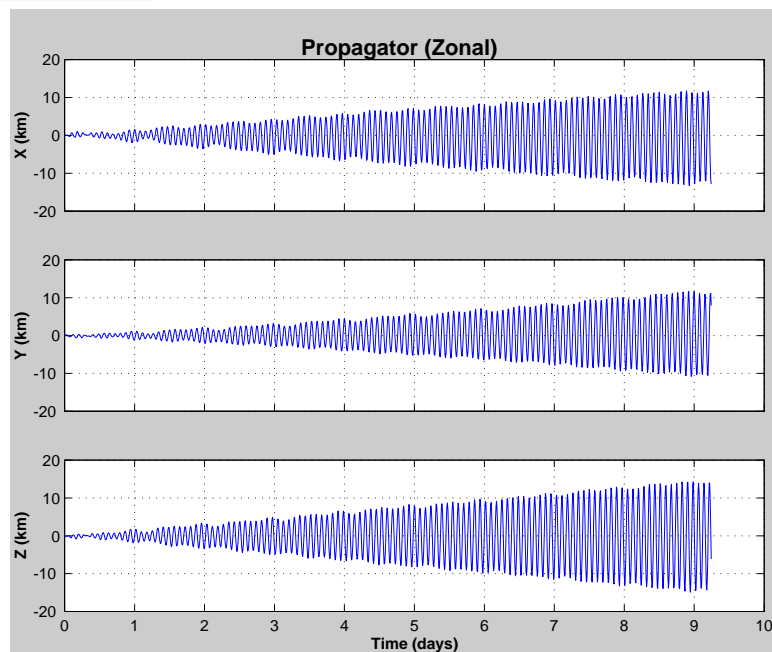


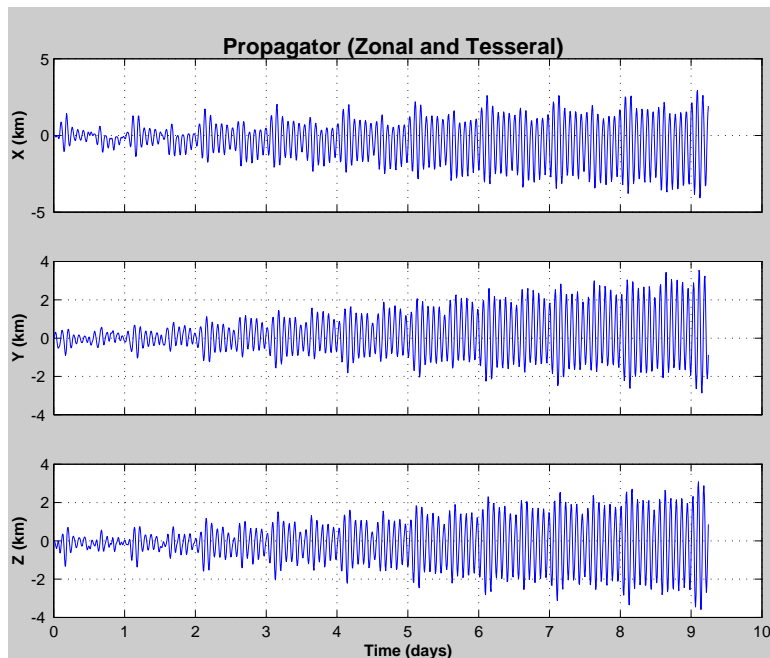
Figure 16.7 on page 142 shows the difference when 4 GEM-T1 zonal harmonics are added. The results show a dramatic improvement in the tracking of the orbit. Finally, 4 tesseral harmonics are added in Figure 16.8 on page 143.

**Figure 16.7:** Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal Harmonics



Again, an improvement is noted.



**Figure 16.8:** Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal and 4 Tesseral Harmonics

## 16.4 Integration Tolerance

### Reference

MATLAB Users Guide.

### Introduction

There are two integration tolerance parameters for `ode113`; one is relative tolerance and one is absolute tolerance.

### Results

Type

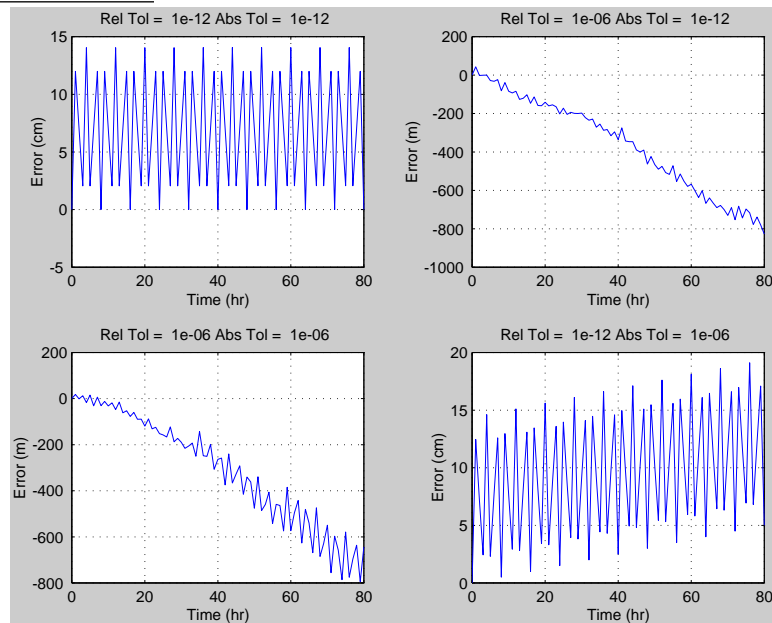
`IntegrationAccuracyDemo`

Figure 16.9 on the following page shows the effects of varying the integration tolerances for a 7000 km circular orbit.

The main diagonal plot units, with relative tolerances of  $1e-12$ , are in centimeters while the off-diagonal plots, with relative tolerances of  $1e-6$ , are in meters.

The differences are substantial. If you used the tolerances in the upper right hand corner, and didn't know that the orbit was unperturbed, you might think that the orbit was decaying! This also shows the danger of comparing one simulation against another. Unless you know the error tolerances for both runs, and have a good idea how the error correction algorithms work in both cases, your comparison could be meaningless.

You always want to get your simulations done as fast as possible. This is particularly true if you are doing Monte Carlo type analyses. Tighter tolerances mean longer simulation times. A balance must be made between integration

**Figure 16.9:** Effect of Integration Tolerances on Orbit Propagation

accuracy and time.

The number of samples output does not have an effect on simulation time.

The durations for the four simulations are given in the following table. These durations include all graphics generated at the end of the simulations.

**Table 16.2:** Simulation Times for a 140 hour Circular Orbit Simulation run on a 500 MHz Apple PowerBook G3

Rel Tol	Abs Tol	Duration (sec)	Accuracy
1e-12	1e-12	76.3	15 cm
1e-06	1e-12	36.2	1000 m
1e-06	1e-06	37.6	1000 m
1e-12	1e-06	62.4	20 cm

---

## ORBIT MANEUVERS

---

This chapter shows you how to plan orbit maneuvers.

### 17.1 Orbit Maneuver Functions

---

The following listing from `OrbMnvrDemo` will result in a demo of many of the orbit maneuver functions.

**Listing 17.1:** Orbit maneuver demonstration script

```
OrbMnvrBiElliptic
OrbMnvrHohmann
OrbMnvrInclination
OrbMnvrOneTangent
OrbMnvrLongitude
OrbMnvrLongAndIncl
OrbMnvrSemimajor
```

The following listing shows the results of running the script. Each function has a built-in demonstration to illustrate the inputs and outputs of the function. Each implements a relatively simple maneuver and they are useful for developing maneuver strategies.

**Listing 17.2:** Results

```
Bi-elliptic Transfer
-----
Initial Orbit Radius = 6569.5100
Final Orbit Radius = 382689.9000
Intermediate Orbit Radius = 510253.2000
-----
Delta V Total = 3.9040
Delta V at A = 3.1562
Delta V at B = 0.6774
Delta V at C = 0.0705
Time of Flight = 593.9238 hours

Hohmann Transfer
-----
Initial Orbit Radius = 6569.5091
Final Orbit Radius = 42159.5136
-----
Delta V Total = 3.9352
Delta V at A = 2.4570
Delta V at B = 1.4782
```

```

Time of Flight = 5.2567 hours

Inclination Correction
-----
Velocity = 3.5680
Eccentricity = 0.3000
True anomaly = 150.0000 deg
Inclination Correction = 15.0000 deg
-----

Delta V Total = 0.9129
Flight Path Angle = 11.4559 deg

One-Tangent Transfer
-----
Initial Orbit Radius = 6569.5091
Final Orbit Radius = 42159.5136
True anomaly = 160.0000 deg
Type = periapsis
-----

Delta V Total = 4.6993
Delta V at A = 2.5754
Delta V at B = 2.1239
Eccentric Anomaly = 127.8053 deg
Time of Flight = 3.4574 hours

Longitude Correction
-----
Velocity = 5.8923
Inclination = 55.0000 deg
Longitude Correction = 45.0000 deg
-----

Delta V Total = 3.6942
Initial latitude = 103.3647 deg
Final latitude = 76.6353 deg

Longitude Correction
-----
Velocity = 5.8923
Inclination = 0.0000 deg
Longitude Correction = 45.0000 deg
-----

Delta V Total = 3.6159
Initial latitude = 128.9041 deg
Final latitude = 97.3803 deg

Semimajor Axis Correction
-----
Velocity = 7.7843
Semimajor axis = 6578.0000
Semimajor axis Correction = 13.0000
-----

Delta V Total = 0.0077

```

---

## 17.2 Launch and Early Orbit Operations

---

The toolbox can be used for Launch and Early Orbit Operations (LEOP) analysis. `LEOPAnalysis.m` demonstrates these capabilities. Liquid apogee engines (LAEs) allow for precise control of insertion burns leading to less of a need to use the mission orbit system. However, because LAEs have much lower thrusts than solid motors it is necessary to perform multiple burns (compared to one for the solid). The script allows you to select a burn sequence for geosynchronous orbit transfer using a single array

```
burn = [0 0.5 0 0.3 0 0.15 0 0.05 0]
```

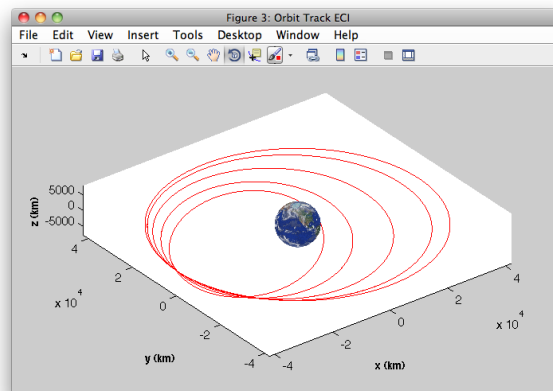
In this case nine revolutions (orbits) are in the plan and there are burns on rev 2, 4, 6 and 8. The number is the percentage of the total delta- $v$  to be done on each burn. The script uses the rocket equation to compute the total amount of fuel needed for the spacecraft. All burns are done around apogee so the required delta- $v$  vector is

$$\Delta v = v_{drift} - v_{transfer}$$

where drift refers to the orbit after the final burn because the insertion point is usually not the final station and the spacecraft “drifts” to station. Transfer orbit is the orbit upon separation from the launch vehicle.

Figure 17.1 shows the transfer orbit and the semi-major axis, inclination and eccentricity. The five distinct orbital paths correspond to the initial transfer orbit followed by the four burn sequence. The script uses MATLAB’s `ode113` for orbit propagation and uses `ode113`’s “events” capability to find apogee on each burn rev. The entire script is only 172 lines long and can be easily customized to provide more sophisticated burn planning.

**Figure 17.1:** Launch and early orbit operations analysis



The dynamical model uses a point mass Earth model, although higher fidelity models can easily be used instead (the toolbox includes a 36<sup>th</sup> order gravity model). In this example the final burn leaves the spacecraft on station. In practice, it would be left with some drift velocity. In addition, the burn vector is held constant when small adjustments might need to be made due to orbit determination errors and orbit perturbations.

## 17.3 Drag Correction Example

This example shows how to compute the compensation for drag. The script is shown in Listing 17.3. The demo first compares an analytical model of drag with the orbit propagator. It then computes osculating elements for each  $(r, v)$  pair. The osculating elements are plotted. The trend shows a steady decrease in semimajor axis. This delta is then compensated for by a burn at perigee computed using one of the maneuver functions.

**Listing 17.3:** Drag correction script

*DragCompensationDemo.m*

```
mu      = Constant('mu_earth');
rE      = Constant('equatorial_radius_earth');
area    = 1; % m^2
mass    = 100;
cD      = 2.7;
bStar   = cD*area/mass;
H0      = 200;
H1      = 200;
H2      = 180;
```

```

rho1      = AtmDens2( 200 );
rho2      = AtmDens2( 180 );
h         = (H2-H1)/log(rho1/rho2);
rho0      = rho1/exp(-H1/h);
dT        = 3600;
nSim      = 10;

t         = (0:(nSim-1))*dT;
xPlot     = zeros(2,length(t));
xPlot(1,:) = h*log( exp(H0/h) - sqrt(mu*rE)*bStar*rho0*1e3*t/h );

% Run the orbit propagator in batch
%-----
d         = load('DragCompensationDemo.mat');
tag       = PropagateOrbitPlugIn( 'initialize' );
PropagateOrbitPlugIn( 'set_data', tag, d );
PropagateOrbitPlugIn( 'propagate', tag );

r         = PropagateOrbitPlugIn( 'get_r', tag );
v         = PropagateOrbitPlugIn( 'get_v', tag );
nP        = size(r,2);

if( ~isempty(r) & ~isempty(v) & nP > 0 )
    t      = t(1:nP);
    xPlot  = xPlot(:,1:nP);
    xPlot(2,:) = RV2AE( r, v, mu ) - rE;
    [t, xLabl] = TimeLabl( t );
    [h, hA]   = Plot2D( t, xPlot, xLabl, 'H' );
    legend( hA.h, 'Analytical', 'Propagator' );

    n = nP;
    for k = 1:n
        el(:,k) = RV2E1( r(:,k), v(:,k) );
    end

    yL = ['a';'i';'w';'w';'e'];
    Plot2D( t, el(1:5,:), 'Time', yL, 'Elements' );

% Compute the change in semimajor axis
%-----
deltaA = el(1,1) - el(1,end);
deltaV = OrbMnvrSemimajor( Mag(v(:,1)),el(1,1),deltaA);
disp(sprintf('Drag_compensation_delta-V_is_%12.4f_km/sec',deltaV.total));
end

```

*DragCompensationDemo.m*

This demo also shows you how to use the orbit propagator in batch mode. When you execute the script the Orbit Propagator GUI appears as shown in Figure 17.2 on the next page. The aero model inputs are shown as well.

The model assumes the vehicle is a sphere with a simple drag coefficient. The first plot (Figure 17.3 on the facing page) shows the analytical and propagated results. The next plot shows the variation in the orbital elements.

Since the orbit is circular the argument of perigee is not well-defined. The variation in the semimajor axis is nearly linear. This variation is used to compute the required delta-V.

The required delta-V is displayed in the command window.

```

>> DragCompensationDemo
Drag compensation delta-V is      0.0077 km/sec

```

Figure 17.2: Orbit propagation GUI and aero model inputs

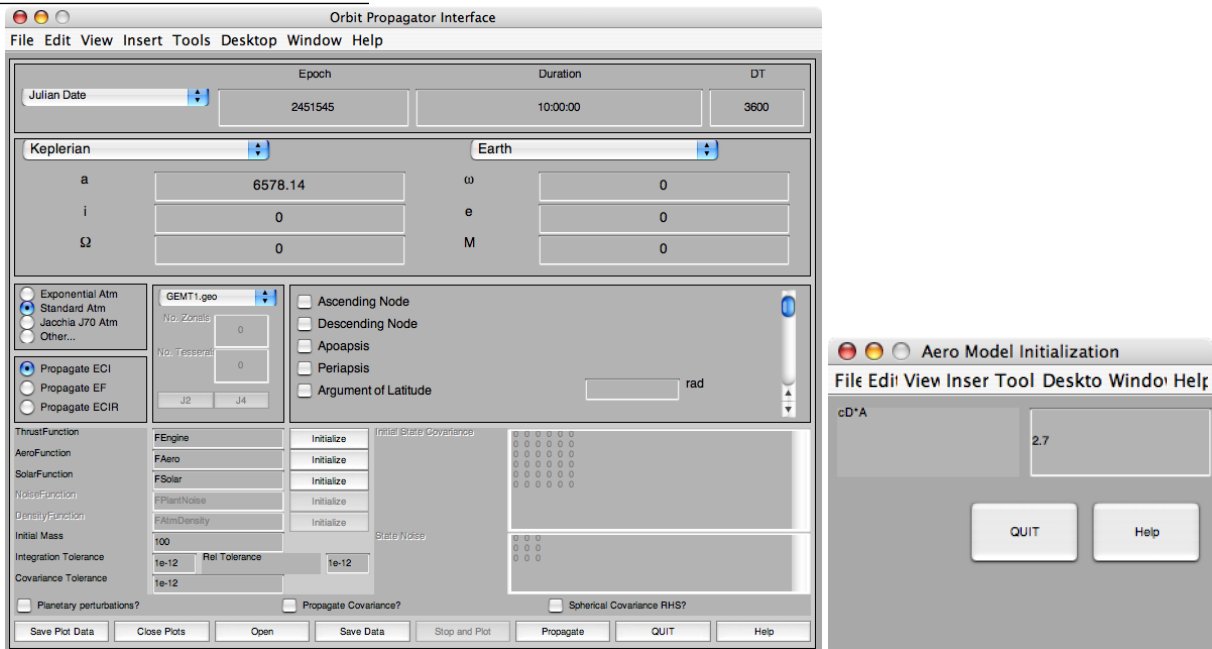


Figure 17.3: Comparison of analytical drag with propagated

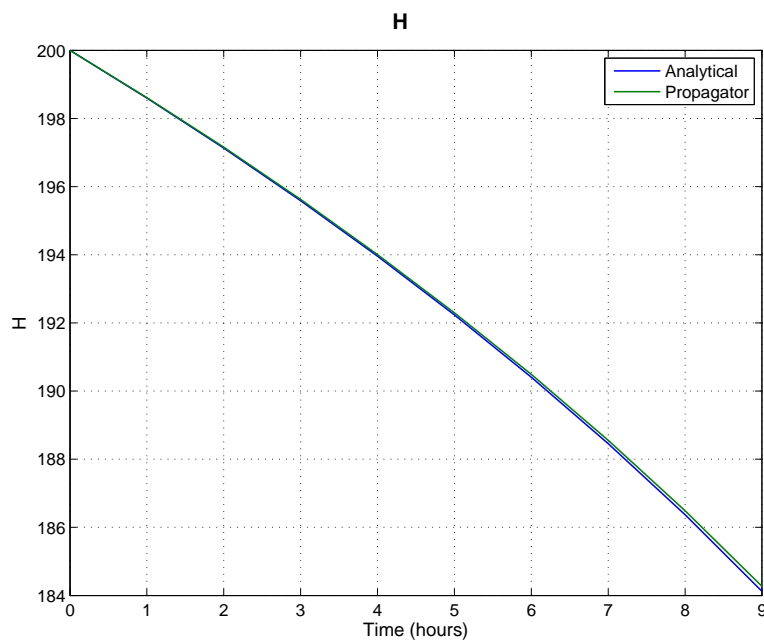
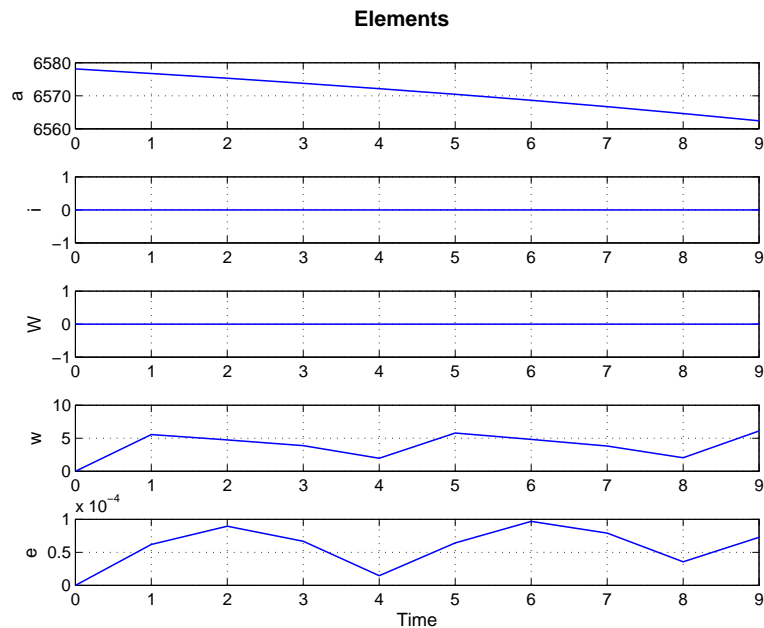


Figure 17.4: Orbital elements



## 17.4 Glideslopes

A glideslope is a trajectory where range rate is proportional to range. This can be used for rendezvous and proximity operations, however, these algorithms are only applicable to circular orbits. The toolbox has functions for generating maneuvers for inbound and outbound glideslopes. The same functions can also be used for circumnavigation in a set number of pulses.

The functions in glideslope have built-in demos. The function `GlideslopePlanner` calculates sequential glideslope maneuvers. The default data contains three segments, an inbound slope, a circumnavigation, and an outbound slope. The Clohessy-Wiltshire equations are used to propagate in between segments.

Listing 17.4: Default data for the built-in demo

*GlideslopePlanner.m*

```

%-----
% Default data for a planning demo
%-----
function d = DefaultData

% Inbound slope
d(1).x0      = [1000; 1000; 1000; 0; 0; 0];
d(1).rTarget = [300; 0; 0];
d(1).V       = -[17.56 0.084];
d(1).T       = 10*60;
d(1).N       = 4;

% Circumnavigation
d(2).rTarget = 300;
d(2).V       = [];
d(2).T       = 30*60;
d(2).N       = 8;
d(2).a       = [0;1;0];

% Outbound slope
d(3).rTarget = [750; 750; 750];
d(3).T       = 6*60;
d(3).N       = 4;
d(3).V       = [1 10];

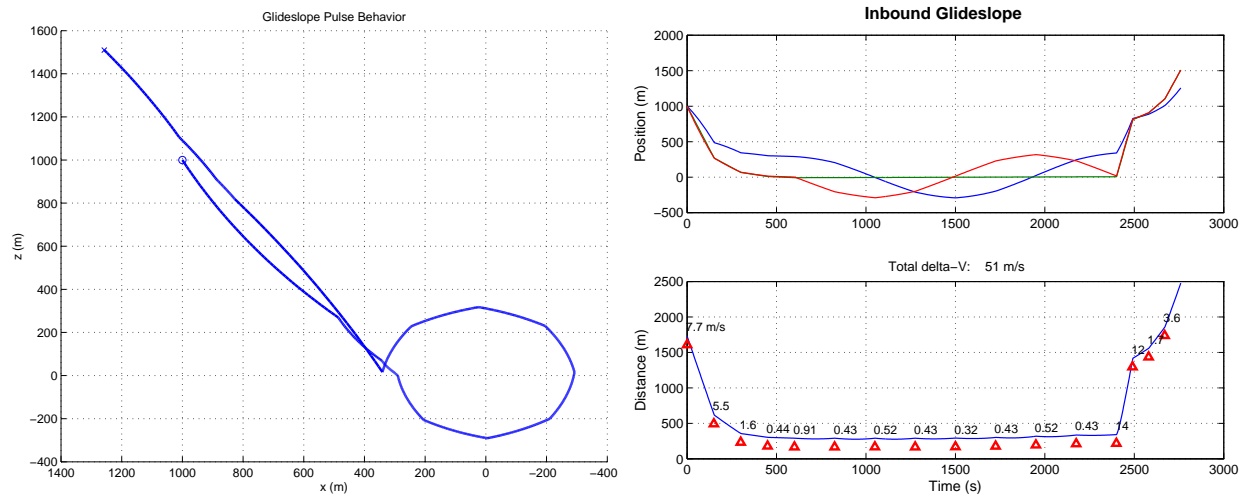
```

*GlideslopePlanner.m*



The results of this demo are shown in Figure 17.5. The maneuvers are indicated by red triangles.

**Figure 17.5:** Glideslope demo results





# ORBIT PROPAGATOR GUI PLUGIN

---

This chapter shows you how to propagate orbits and perform related tasks using the Propagator Plug In tool, `PropagateOrbitPlugIn`. This tool enables you to create some very elaborate propagations without writing lengthy scripts. The functions can be found in the `OrbitPropagator` folder.

## 18.1 The Orbit Propagator Tool

---

### 18.1.1 Introduction

The function `PropagateOrbitPlugIn` allows for 3 degree-of-freedom propagation of orbits. It can be used interactively or in batch mode. It has the following features:

- User selectable gravity models. GEM-T1, JGM-2, JGM-3 and WGS-84 are supported. Other models can be downloaded.
- Ballistic propagation of covariance matrix for state vector. A plug in function allows the user to control the plant noise covariance matrix.
- Propagate in ECI, ECR (earth-fixed) or ECIR. The last is an inertial frame that is coincident with EF at time 0.
- The user has complete control accuracy of numerical integration. The propagator uses the MathWorks' state-of-the-art `ode113` propagator.
- The user can select the time step. The time step can be variable and negative. It can be defined by any MATLAB expression or function.
- The user can specify any one of over thirty stopping conditions.
- Flight path angle, altitude, geodetic latitude and longitude are automatically computed.
- Outputs can be saved to a mat-file for further analysis.
- Sun/moon/earth perturbations can be added if you are propagating in an earth-centered or moon-centered frame.

### 18.1.2 Limitations

The following are some limitations that will be addressed in a future version:

- The propagator does not switch centers when the trajectory leaves one planet's sphere of influence and enters another.

- Only the moon and sun are added as perturbations when the earth is the center and the sun and earth are entered as perturbations when the moon is the center.
- Only centers for which the toolbox has pictures are included. This leaves out many moons.
- Only Mean-of-Aries 2000, Earth-Fixed and inertial coincident with earth-fixed frames are available.
- The noise function is not computed automatically.

## 18.2 Using the Graphical User Interface

---

The graphical user interface displays all options for using the propagator.

Figure 23-1 Orbit Propagation GUI

### 18.2.1 Overview

The orbit propagator interface, which is created by `PropagateOrbitPlugIn.m`, allows you to propagate any type of orbit using `ode113`, the MathWork's state-of-the-art propagator.

The GUI has several different "panes." These are summarized below. The following sections discuss each in more detail.

- Time Pane - set the time
- Elements Pane - set the initial orbital elements. These are always referenced to the ECI frame
- Atmosphere Pane - select one of four atmosphere models
- Propagation Pane - select the coordinate frame in which to propagate
- Gravity Model Pane - select the gravity model and the order of the model
- Stopping Conditions Pane - select the stopping condition for the simulation
- Customization Pane - select simulation options and functions you wish to plug into the simulation
- Buttons - control the simulation

The propagator can be run interactively or as part of a MATLAB script.

### 18.2.2 Panes

#### Time Pane

The time plug-in, in the upper right corner of the Propagator window, contains the values for the epoch, maximum duration, and time step of the next propagation run. This plug-in allows you to convert the epoch between Julian Date and UTC Calendar date. The duration field allows you to enter a maximum duration for the run. If the specified stopping condition is not met within the specified duration the run will stop. The default duration is one hour. The DT field allows you to specify the time step of the propagation. The default DT is 100 seconds. DT may be negative.

A set of times can be entered for the duration. For example you could enter:

```
0:25 hr
```

or

```
linspace(0,60000,1000) sec
```

or

```
[0 1 7 9 11 22 43 900] min
```

or

```
[900 43 22 11 9 7 1 0] hr
```

or

```
MyTimeSequence hr
```

or

```
3 orbits
```

Any MATLAB command or function can be entered into duration. If anything except a time is entered, DT is ignored. The output will be computed at these specific times only. The numerical integration routine will select the appropriate time step to achieve the specified numerical accuracy independent of the entered times.

### Elements Pane

The elements plug-in allows you to enter and convert between Keplerian elements, equinoctial elements, and RV elements. The planet used for the orbit center is also shown. Element sets are given in the following table. The RV elements are position and velocity vector.

**Table 18.1:** Element sets

Keplerian		RP/RA		Equinoctial		RV
a	Semi-major axis	$R_p$	Perigee radius	a	Semi-major axis	$r_1$
i	Inclination	i	Inclination	$P_1$		$r_2$
$\Omega$	Right Ascension of the Ascending Node	$\Omega$	Right Ascension	$P_2$		$r_3$
$\omega$	Argument of Perigee	$\omega$	Argument of Perigee	$Q_1$		$v_1$
e	Eccentricity	$R_a$	Apogee radius	$Q_2$		$v_2$
M	Mean anomaly	M	Mean anomaly	l	Mean longitude	$v_3$

### Atmosphere Pane

You can select any one of four models. The first two are altitude dependent and are based on a spherical planet model. The third will enable the default Atmospheric density plugin function, `FAtmDensity`, which implements the Jacchia J70 model. Other... allows you to enter your own atmosphere model.

### Propagate Pane

This selects the frame in which to propagate. The ECIR frame is an inertial frame which coincides with the EF frame at the start of the simulation.

### Gravity Model Pane

The Database model is a spherical planet model with the gravitational parameter extracted from the `Constant` function. Currently, this the only model you can use for planets or moons other than the earth.

You can download GEMT\*, JGM-\* and WGS84 models from various websites. Any file with the suffix `.GEO` will be displayed in the pull-down menu. You can enter the number of zonal and tesseral harmonics you wish to use. If you just want the J2 or J4 zonal, you can hit the button to set the appropriate model order.

### Stopping Conditions Pane

You can select any combination of stopping conditions listed in the pane. For each you can enter a tolerance. Some require data. The stopping conditions are described in the following table. Some of the conditions aren't relevant to true Keplerian orbits in which the orbital elements do not change.

**Table 18.2:** Stopping Conditions

Stopping Condition	Explanation
Impact	Stop when orbit crosses the central body ellipsoid. This is the same as setting the geodetic altitude to zero.
Ascending Node	Stop at the ascending node.
Descending Node	Stop at the descending node.
Apoapsis	Stop at apoapsis.
Periapsis	Stop at periapsis
Argument of Latitude	Stop when $\omega + \nu = \text{value}$
XZ Crossing	Stop when ECI $y = 0$
XY Crossing	Stop when ECI $z = 0$
YZ Crossing	Stop when ECI $x = 0$
True Anomaly	Stop when $\nu = \text{value}$
Approaching Distance	Stop when $r = \text{value}$ and is decreasing
Receding Distance	Stop when $r = \text{value}$ and is increasing.
Increasing Velocity	Stop when $v = \text{value}$ and is decreasing
Decreasing Velocity	Stop when $v = \text{value}$ and is increasing.
Flight Path Angle	Stop when the flight path angle = the value
Perigee Height	Stop when perigee radius = value
Apogee Height	Stop when apogee radius = value
Geodetic Latitude	Stop when geodetic latitude = value
Geocentric latitude	Stop when geocentric latitude = value
Geodetic Altitude	Stop when geodetic altitude = value
Semi major Axis	Stop when $a = \text{value}$
Mean Anomaly	Stop when $M = \text{value}$
Orbital Period	Stop when orbital period = value
Right Ascension of Ascending Node	Stop when $\Omega = \text{value}$ and only at the descending node
Right Ascension of Descending Node	Stop when $\Omega = \text{value}$ and only at the ascending node
Change in Inertial Velocity Magnitude	Stop when $ v  = \text{value}$
Vehicle Longitude	Stop with earth-fixed longitude = value
Argument of Perigee	Stop when argument of perigee = value
Eccentric Anomaly	Stop when $E = \text{value}$

### Customization Pane

The first four buttons are the names of functions the propagator should call. The default names in the GUI are default functions. When you hit Initialize, the function will display an initialization GUI.

The Noise Function returns the plant noise matrix,  $q$ , for the covariance propagation:

$$p\text{Dot} = f * p + p * f' + q$$

where  $f$  is the matrix of partials of the right-hand-side of the orbit.

The initial mass is the mass of the spacecraft.

The next two parameters relate to the integration accuracy.

Rel Tolerance gives the relative error tolerance. The estimated error in each integration step satisfies

```
e(i) <= max(RelTol*abs(y(i)), Tol(i))
```

Integration Tolerance gives the absolute error tolerance which applies to all components of the solution vector.

The next two checkboxes allow you to select planetary perturbations and/or to propagate the covariance. When you select planetary perturbations you will get the moon and the sun if the earth is selected as the center through the elements pane or the earth and the sun if the moon is selected as the center. If you select other planets or moons you will not get any perturbations.

When you select Propagate Covariance the initial covariance input will be enabled. You should enter the initial state covariance into the box. You need only enter upper or lower triangular information. The propagator automatically does

$$p = 0.5 * (p + p')$$

to insure that the covariance matrix is symmetric. You have the option to select Spherical Cov. This uses a spherical earth model just to propagate the covariance. This is much faster than using the full model.

### Buttons

The following is the list of buttons in the GUI.

- Save Plot Data saves the plot data in a mat file of type \*.OPP. The plot is saved as a matrix with rows: [x;y;z;vX;vY;vZ;mass;pXX;...pVZVZ;pXY;pYZ...] p is saved in columns starting with the main diagonal and moving up. For example, a 3x3 matrix would be saved as [a11;a22;a33;a12;a23;a13];
- Close PlotsClose the plot windows.
- OpenOpen a gui dat file of type \*.OPG.
- Save DataSave the gui data in a mat file of type \*.OPG.
- PropagateRun the propagator.
- QUITQuit. It will ask you to save unsaved plot and GUI data.
- HelpGet help.

## 18.3 Plugin Functions

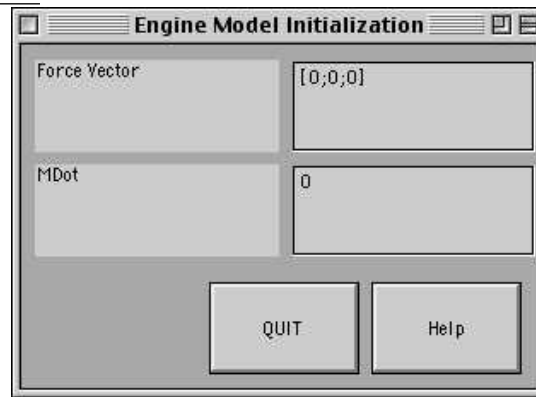
The orbit propagator has four plugin in functions that allow you to customize the simulation. When you first bring up the GUI four default functions are shown. They are described in Table 18.3.

**Table 18.3:** Plugin Functions

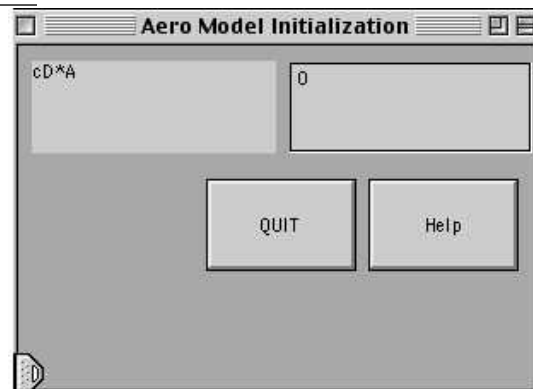
Function	Description
FEngine	Computes engine force and spacecraft mass. The force is in the LVLH frame.
FAero	Computes aero forces in the ECI frame.
FSolar	Computes solar forces in the ECI frame.
FPlantNoise	The Q matrix in the covariance propagation equation.
FAtmDensity	Atmospheric density. The baseline model is Jacchia J70.

Each model supplied with the toolbox has its own GUI. These popup when you hit the Initialize button next to each function's name.

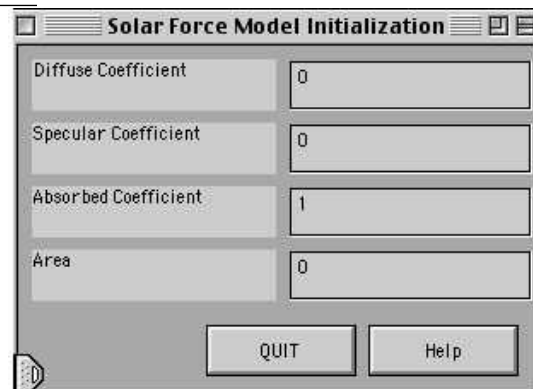
The FEngine model, Figure 18.1 on the next page, allows you to enter a constant force in the ECI frame and a constant mass change rate.

**Figure 18.1:** FEngine GUI

The FAero model, Figure 18.2, assumes a flat plate spacecraft with the plate always normal to the velocity vector. The input is the product of the drag coefficient and plate area.

**Figure 18.2:** FAero GUI

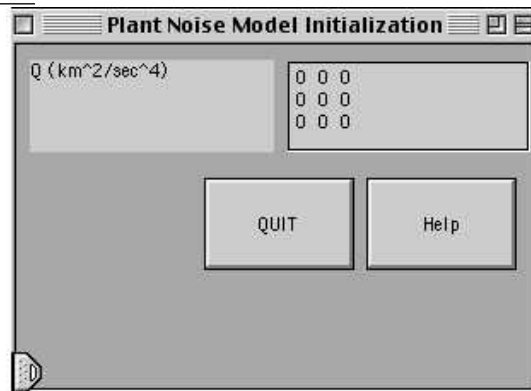
The FSolar model, Figure 18.3, is a flat plat model in which the plate is always normal to the sun vector.

**Figure 18.3:** FSolar GUI

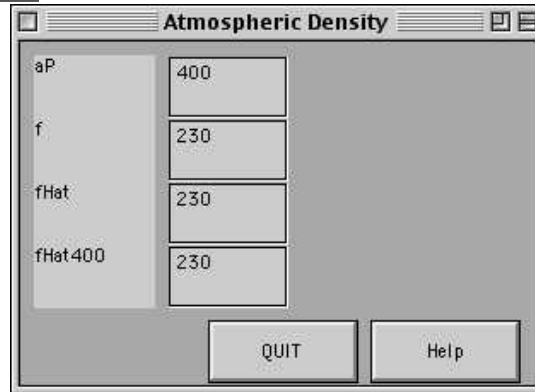
FPlantNoise function, Figure 18.4 on the facing page, assumes a constant plant noise covariance matrix. The inputs are the uncertainty in body accelerations.

The FAtmDensity GUI, Figure 18.5 on the next page, gives inputs for the Jacchia J70 model. The J70 inputs are



**Figure 18.4:** FPlantNoise GUI

given in Table 18.4.

**Figure 18.5:** FAtmDensity GUI**Table 18.4:** Jacchia J70 Atm inputs

Input	Description
aP	Geomagnetic index 6.7 hours before the computation
f	Daily 10.7 cm solar flux (e-22 watts/m <sup>2</sup> /cycle/sec)
fHat	81-day mean of f (e-22 watts/m <sup>2</sup> /cycle/sec)
fHat400	fHat 400 days before computation date

## 18.4 Building a Plugin Function

All plugin in functions have the same structure. In this section, we will walk you through the structure of FAero, the simplest function.

**Listing 18.1:** Function Definition

FAero.m

```
1 function x = FAero( action, modifier, u, t )
```

FAero.m

Listing 18.2 on the next page shows the switch statement that makes up the main function. `action` is the argument for the switch statement and `modifier` is the GUI tag, which identifies the GUI.

Listing 18.2: Main Function Body

FAero.m

```

1  if( nargin < 1 )
2      action = 'initialize';
3  end
4
5  switch action
6
7      case 'initialize'
8          x = Initialize( modifier );
9
10         case 'update'
11             x = Update( modifier, u, t );
12
13         case 'get_data'
14             x = GetData( modifier );
15
16         case 'store_data'
17             h = GetH( modifier );
18             h.d = GetData( modifier );
19             PutH( h );
20
21         case 'set_data'
22             SetData( modifier, u );
23
24         case 'help'
25             HelpSystem( 'initialize', 'OnlineHelp' );
26
27         case 'quit'
28             h = GetH( modifier );
29             if( ishandle( h.propagateOrbitPluginTag ) )
30                 PropagateOrbitPlugIn( 'close_function', h.propagateOrbitPluginTag, 'FAero' );
31             end
32             CloseFigure( h.fig )
33
34 end

```

FAero.m

The Initialize subfunction draws the GUI and returns the tag for the figure.

Listing 18.3: Initialize

FAero.m

```

1  function tag = Initialize( propagateOrbitPluginTag )
2
3  name      = 'Aero_Model_Initialization';
4  position  = [5 5 300 200];
5  tag       = GetNewTag( name );
6  h.fig     = figure( 'position', position, 'color', [0.66 0.66 0.66], 'NumberTitle', 'off', ...
7              'name', name, 'resize', 'off', 'tag', tag, 'CloseRequestFcn',
8              CreateCallback( 'quit', tag ) );
9  fontSize  = position(4)*9/185;
10 v         = { 'parent', h.fig, 'fontunits', 'pixels', 'fontsize', fontSize, '
11             horizontalalignment', 'left' };
12 space     = position(3)/40;
13 x0        = space;
14 dX        = position(3)/2;
15 xW        = dX - space;
16 dY        = (position(4) - 2*space)/3;
17 yW        = dY - space;
18
19 x         = x0;
20 y         = position(4) - dY;
21
22 % The orbit propagator tag
23 %-----
24 h.propagateOrbitPluginTag = propagateOrbitPluginTag;
25
26 hMatlab6Bug = uicontrol('visible','off'); % Added to fix a Matlab 6 bug

```

```

25
26 % Inputs
27 %-----
28 h.gui.cDAText = uicontrol( v{:}, 'position', [x y xW yW], 'string', 'cD*A', 'style', 'text
   ' ); x = x + xW + space;
29 h.gui.cDA = uicontrol( v{:}, 'position', [x y xW yW], 'string', '0', 'style', 'edit
   ' ); x = x0; y = y - dY;
30
31 % GUI controls
32 %-----
33 xW = (position(3) - 5*space)/3;
34 x = position(3) - xW - space;
35 y = y - space;
36
37 h.gui.help = uicontrol( v{:}, 'position', [x y xW yW], 'string', 'Help', 'callback',
   CreateCallback( 'help', tag ) ); x = x - xW - space;
38 h.gui.quit = uicontrol( v{:}, 'position', [x y xW yW], 'string', 'QUIT', 'callback',
   CreateCallback( 'quit', tag ) ); x = x - xW - space;
39
40 PutH( h )

```

FAero.m

The Inputs and GUI controls lines show you how to write uicontrols for edit boxes seen in the GUI. uicontrols require you to pass them parameters in pairs. To simplify the code we put a lot of the parameters in the cell array `v`. The `CreateCallback` subfunction (Listing 18.4) generates a call back to the `FAero` function. For example the last line creates the call:

```
FAero( quit', tag )
```

Listing 18.4: CreateCallback

FAero.m

```

1 function s = CreateCallback( action, modifier )
2
3 s = ['FAero(_' action '' modifier ')'];

```

FAero.m

See the MATLAB documentation on how to build GUIs for more information. The `get` data function gets data stored in the GUI and `set` data sets the data. `SetData` is used by the orbit propagation function when you pass it the saved propagator data structure. `GetData` is used internally as shown in the next listing.

Listing 18.5: Getting and setting data

FAero.m

```

1 %-----
2 %   Get the data from the gui
3 %-----
4 function d = GetData( modifier )
5
6 h = GetH( modifier );
7
8 d.cDA = str2num( get( h.gui.cDA, 'string' ) );
9
10 %-----
11 %   Set the data in the gui
12 %-----
13 function SetData( modifier, d )
14
15 h = GetH( modifier );
16
17 set( h.gui.cDA, 'string', num2str( d.cDA ) );

```

FAero.m

The `Update` subfunction computes the aerodynamic drag. It gets the figure handle, gets the data from the figure handle, and combines it with the velocity stored in data structure `u`, to compute the force.

The next listing completes our review of `FAero`.

Listing 18.6: The Update subfunction

FAero.m

```

1 function x = Update( modifier, u, t )
2
3 h = GetH( modifier );
4 d = GetData( modifier );
5
6 x = -0.5*d.cDA*u.rho*Mag(u.v)*u.v*1e3; % kg-km/sec^2: m^2*(kg/m^3)*km^2/sec^2, kg-km^2/m
   sec^2

```

FAero.m

## 18.5 Reference Frames

### 18.5.1 Coordinate Transformations

CoordinateTransform, in the Common/Coord portion of the Core module, transforms between:

- ECI
- ECR
- Geodetic latitude, longitude and altitude

For example

```

r = [6524.834;6862.875;6448.296];
x = CoordinateTransform( 'eci', 'ef', r, 2449773 )
x =
    1.0e+03 *
    2.85101441661017
    9.03243563995358
    6.44514208556912

x = CoordinateTransform( 'eci', 'llr', r, 2449773 )
x =
    1.0e+03 *
    0.00059923067421
    0.00126505172630
    5.08520910899187

```

### 18.5.2 Finding ECI/ECR coincidence

EarthRotationZero, in the SC/Ephem portion of the Core module, finds a Julian date for which the Greenwich Apparent Sidereal Time (GAST) is zero.

For example

```

x = EarthRotationZero( JD2000 )
x =
    2.4515e+06

```

gives a Julian Date within a day of JD2000. For this date

```

GASTime( x )
ans =
    7.2978e-09

```

You can also find the transformation matrix from ECI to ECR at any time by typing

```
m = ECIToECIR( jD )
```

$m$  transforms from ECI to the frame aligned with EF at time  $jD$ . This constant offset frame can be used to propagate equations in the ECIR frame.

## 18.6 Sun

SunV1, also in SC/Ephem, provides a moderate precision (0.01 deg) sun angle.

```
uSun = SunV1( jD );
[rA, dec] = U2RADec( uSun )
```

The second function gives the right ascension and declination for the sun.

## 18.7 Interfacing to STK

You can interface to STK using the function STKOrbit. STKOrbit allows you to exchange data with STK by creating an input file for STK.

```
[err, message] = STKOrbit( fileName, ver, epoch, nPoints, time, position,
    velocity, type )
```

**Table 18.5:** I/O for STKOrbit

Variable	Description
fileName	Filename for output. Will be overwritten if it already exists.
ver	STK version number (string!)
epoch	Beginning date and time: [Y M D H M S]
nPoints	Number of orbit data points
time	Time in seconds for each data point
position	Position data in km (if type = ECF or ECI) or in [degrees,degrees,km] (if type = LLR)
velocity	Velocity data in km/sec (if type = ECF or ECI) or [degrees,degrees,km]/sec (if type = LLR)
type	'ECI','ECF',or 'LLR'
err	err = 0 means an error occurred
message	Error message

## 18.8 Ground Coverage

Ground coverage is found using this function from OrbitMechanics:

```
x = GroundCoverage( xECI, jD, d )
```

where  $xECI$  is the ECI state vector  $[r; v]$ ,  $jD$  is the Julian date and  $d$  is the data structure in which  $d.fOV.x$  is the  $x$  axis field-of-view,  $d.fOV.y$  is the  $y$ -axis field-of-view and  $d.m$  transforms from the body frame to the sensor frame. The sensor is assumed to have its boresight along  $+z$  in the sensor frame. If  $d.m$  were the identity matrix the sensor would point along  $+z$  in the body frame. For this function, the satellite body frame is assumed to be aligned with the LVLH frame.

$d.e$  is a 3-by-1 vector with the [x dimension;y dimension;z dimension] of the ellipsoid. If the planet is spherical the three dimensions are the same.

## 18.9 Orbit Propagation API

### 18.9.1 Batch Runs

You can run `PropagateOrbitPlugIn` from a script. This allows you to set up large numbers of cases and run them automatically. `OPBatchDemo` shows how to set up a batch run.

```
d = load('OPDemo.mat');
d.epoch.dT = 3600;
tag = PropagateOrbitPlugIn( 'initialize' );
    PropagateOrbitPlugIn( 'set_data', tag, d );
    PropagateOrbitPlugIn( 'propagate', tag );
    pause
    PropagateOrbitPlugIn( 'close_plots', tag );
    PropagateOrbitPlugIn( 'quit', tag );
```

In the first line a saved data file is loaded. This file was created and saved using the GUI interactively. In the second the time step is customized. Any field in the data structure may be customized. In the third step the GUI is initialized. In the fourth the datastructure `d` is loaded. The orbit is then propagated. Plots are closed after the pause and the final step is to quit. `tag` identifies the GUI. Possible commands are listed in Table 18.6.

**Table 18.6:** `PropagateOrbitPlugIn` API

Prototype	Purpose
<code>tag = PropagateOrbitPlugIn( 'initialize' );</code>	Initialize the plugin.
<code>PropagateOrbitPlugIn( 'set_data', tag, d );</code>	Set the data in the GUI
<code>PropagateOrbitPlugIn( 'propagate', tag );</code>	Propagate the orbit.
<code>PropagateOrbitPlugIn( 'close_plots', tag );</code>	Close the plots
<code>PropagateOrbitPlugIn( 'save', tag, fileName );</code>	Save the GUI data.
<code>PropagateOrbitPlugIn( 'save plot data', tag, fileName );</code>	Save the plot data
<code>PropagateOrbitPlugIn( 'quit', tag );</code>	Close the GUI.

### 18.9.2 Data Structure

The data structure `d` is defined in the following table.

**Table 18.7:** Input data structure for `PropagateOrbitPlugIn`

Field	SubField	Type	Description
<code>f</code>		Struct	Plugin function names and data
	<code>thrustFunction</code>	String (1,:)	Name of the thrust function
	<code>aeroFunction</code>	String (1,:)	Name of the aerodynamics function
	<code>solarFunction</code>	String (1,:)	Name of the solar pressure function
	<code>noiseFunction</code>	String (1,:)	Name of the covariance noise function
	<code>densFunction</code>	String (1,:)	Name of the atmospheric density function
	<code>thrustData</code>	User defined	Data for the users thrust function
	<code>aeroData</code>	User defined	Data for the users aerodynamic function
	<code>solarData</code>	User defined	Data for the users solar pressure function
	<code>noiseData</code>	User defined	Data for the users covariance noise function
	<code>densData</code>	User defined	Data for the users density function
	<code>planetaryDisturbancesOn</code>	(1,1)	1 turns on planetary disturbances
	<code>atmosphere</code>	String (1,:)	String for atmosphere model name
	<code>nZonal</code>	(1,1)	Number of zonal harmonics
	<code>nTesseral</code>	(1,1)	Number of tesseral harmonics
	<code>stopping.condition</code>	String (1,:)	Stopping condition string

covariancePropagationOn		(1,1)	Turns on covariance propagation
q		Struct	
	simpleCovarianceRHSON	(1,1)	Simple covariance right-hand-side enabled
propagate		String (1,:)	Propagation method
mass		(1,1)	Spacecraft mass
epoch		Struct	Time epoch
	jDEpoch	(1,1)	Julian date (days)
	dT	(1,1)	Time step (sec)
	duration	(1,1)	Duration (sec)
tol		(1,1)	Absolute tolerance for ode113
reitol		(1,1)	Relative tolerance for ode113

### 18.9.3 Function Inputs

The following table lists all inputs to `PropagateOrbitPlugIn`.

**Table 18.8:** Inputs to `PropagateOrbitPlugIn`

Action	Inputs	Output	Description
initialize	None	None	Tells the GUI to initialize itself.
propagate	None	None	Propagate the orbit
propagate covariance	None	None	Propagate the covariance
atmosphere	None	None	Open the atmosphere GUI
gravity	None	None	Open the gravity GUI
set data	Data structure (see above)	None	Provide a data structure to the GUI
get data	None	GUI data structure	Returns the GUI data structure
get results	None	Plot data	Returns the plot data
get r	None	r(3,:)	Returns the position vector array
get v	None	v(3,:)	Returns the velocity vector array
get mass	None	(1,1)	Returns the spacecraft mass
save	String (1,:)	None	Save the data in the GUI
open	None	None	Open a saved file
save plot data	String (1,:)	None	Plot data file name
initialize function	None	None	Initialize the plugin functions
close function	String (1,:)	None	Close the input plugin
j2	None	None	Set the number of zonal harmonics to 2 and tesseral to 0
j4	None	None	Set the number of zonal harmonics to 4 and tesseral harmonics to 0
help	None	None	Open the help GUI
quit	None	None	Close the GUI

### 18.9.4 Function Outputs

The plot outputs for `PropagateOrbitPlugIn` are given in the following table.

**Table 18.9:** Plot outputs for `PropagateOrbitPlugIn`

Plot Data	Description	Units
x(1:3,:)	Position	km
x(4:6,:)	Velocity	km/sec
x(7,:)	Mass	kg
x(8,:)	Geodetic Latitude	deg
x(9,:)	Longitude	deg

---

x(10,:)	Altitude	km
x(11,:)	Flight path angle	rad
x(12:17,:)	Covariance diagonal	



---

# FUEL BUDGETS

---

This chapter discusses how to generate fuel budgets. These budgets are some of the most important elements of spacecraft system design.

## 19.1 Fuel Budget

---

Computation of fuel budgets is a straightforward bookkeeping exercise. The most important point to remember is that the fuel consumption of the thrusters is a function of system pressure. Consequently, fuel budgets must be done chronologically.

FBudget generates fuel budgets. There are four types of maneuvers supported. They are spin precession maneuvers, spin changes, AKM firings, and stationkeeping maneuvers. You set up each maneuver using the FBudget data structures as shown in the following example.

### Listing 19.1: Fuel Budget: Moment Arms

```
% You need position vectors and moment arms.
r = [ -0.8 -0.8 -0.8 -0.8 0.8 0.8 0.8 0.8 0.8 -0.8 -0.8 0.8 0.2 -0.2 -0.2 0.2;...
      -0.9 -0.8 0.7 0.8 -0.8 -0.8 0.8 0.8 -0.8 -0.9 -0.8 -0.7 -0.8 -0.8 -0.8 -0.8;
       0.8 -0.8 -0.8 0.8 0.9 -0.8 -0.8 0.7 0.8 0.8 -0.8 -0.8 0.15 -0.2 -0.2 0.2];
u = [ -1 -1 -1 -1 1 1 1 1 0 0 0 0 0 0 0 0;...
       0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1;...
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

---

In the next listing the thruster performance data is collected into data structures.

### Listing 19.2: Fuel Budget: Thruster performance data

```
for k = 1:12
    f(k).iSP = [100 100 10]; % See REA.m for use of this array
    f(k).thrust = 2.07e-6; % See REA.m for use of this array
    f(k).riseTime = 0.016; % In seconds
    f(k).fallTime = 0.016;
    f(k).u = u(:,k);
    f(k).r = r(:,k);
    f(k).type = 'liquid';
    f(k).systemID = Odd(k) + 1; % You can have as many fuel systems as you want
end
for k = 13:16
    f(k).iSP = [100 250 10];
    f(k).thrust = 0.828e-6; % In this case just multiplying this by pressure
    f(k).riseTime = 0.016;
```

```

f(k).fallTime = 0.016;
f(k).u = u(:,k);
f(k).r = r(:,k);
f(k).type = 'liquid';
f(k).systemID = Odd(k) + 1;
end

```

The next listing has general spacecraft data. This includes details about the thruster system. In this case, there are two systems (often called half systems) each using liquid propellant thrusters with equal volumes, pressures and temperatures.

### Listing 19.3: Fuel Budget: General spacecraft data

```

% Assemble general spacecraft data.
x.regulatedPressure = [250 250]*6895;
x.systemTemp = [298 298];
x.systemType = ['liquid';'liquid'];
x.systemVol = [1.6505e-01 1.6505e-01];
x.systemFuel = [1.1789e+02 1.1789e+02];
x.pressurantMass = [1.8372e-01 1.8372e-01];
x.pressurantR = MolWt2R( 0.004 )*[1 1];
x.dryMass = 9.5514e+02;
x.fuelDensity = [1000 1000];
x.thrusterData = f;
x.controlDT = 1;
k = 0;

```

The next set of listings define the fuel burning events for the budget. The event structure fields are defined in the header of FBudget, along with an indication of which events require which data. The definitions are copied below for reference.

### Listing 19.4: Fuel Budget: Event structure fields

```

% event (:) Event data structure
% event.description (1,:) String describing the event
% event.type (1,:) 'initial', 'delta-v', 'delta-h', 'acs', 'pulses',
% 'circular'
% event.aCSThruster (1,:) List of ACS thrusters to use <- all
% event.dVThruster (1,:) List of deltaV thrusters to use <- 'delta-V'
% event.dVTotals (1,1) Total delta-V <- 'delta-V'
% event.dTControl (1,1) Control period <- 'delta-V', 'acs'
% event.burnEff (1,1) Burn efficiency <- 'delta-V'
% event.dHTotals (1,1) Total delta-H <- 'delta-H'
% event.disturbance (3,1) Attitude disturbance <- 'acs'
% event.pulsewidth (1,1) Pulsewidth to be used <- all
% event.nPulses (1,1) Number of pulses <- pulse
% event.duration (1,1) Used for attitude control <- 'acs'
% event.regulated (1,1) 1 if regulated at following level
% event.cM (3,1) Center of mass
% event.circular (1,1) struct('period',p,'radius',r,'duration',t)
% <- 'circular'

```

The first event shows the initialization of the arrays.

### Listing 19.5: Fuel Budget: Array initialization

```

% Event 1: Initialization
%-----
k = k + 1;
event(k).type = 'initial';
event(k).description = 'Initial_State';
event(k).aCSThruster = [];
event(k).dVThruster = [];

```

```

event(k).dVTotal = [];
event(k).dHTotal = [];
event(k).disturbance = [];
event(k).pulsewidthACS = 0;
event(k).pulsewidthDV = 0;
event(k).nPulses = [];
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];

```

The next shows an entry for a continuous Delta-V burn.

#### Listing 19.6: Fuel Budget: Continuous delta-v burn

```

% Event 2: A continuous delta-V burn using different ACS and DV thrusters
%-----
k = k + 1;
event(k).type = 'delta-v';
event(k).description = 'Delta_V_Continuous';
event(k).acSThruster = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster = [13 14];
event(k).dVTotal = 100; % Total delta V in m/sec
event(k).dHTotal = [];
event(k).dTControl = 1;
event(k).disturbance = [];
event(k).nPulses = [];
event(k).duration = [];
event(k).regulated = 0; % If not regulated use blowdown curves
event(k).cM = [0;0;0];

```

The next listing shows the entry for a delta-v burn with off-pulsing.

#### Listing 19.7: Fuel Budget: Delta-v with off-pulsing

```

% Event 3: Delta-V off-pulsing some thrusters
%-----
k = k + 1;
event(k).type = 'delta-v';
event(k).description = 'Delta_V_Off_Pulse';
event(k).acSThruster = [1 2 3 4 9 10 11 12];
event(k).dVThruster = [1 2 3 4];
event(k).dVTotal = 100;
event(k).dHTotal = [];
event(k).dTControl = 1;
event(k).nPulses = [];
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];

```

The next listing shows an unloading momentum entry.

#### Listing 19.8: Fuel Budget: Unloading momentum

```

% Event 4: Unloading momentum
%-----
k = k + 1;
event(k).type = 'delta-h';
event(k).description = 'Delta_H';
event(k).acSThruster = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster = [];
event(k).dVTotal = [];
event(k).dHTotal = [1;0;0];
event(k).dTControl = 1;
event(k).disturbance = [];

```

```

event(k).nPulses = [];
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];

```

This listing shows firing of the ACS thrusters in response to a disturbance.

#### Listing 19.9: Fuel Budget: Using ACS thrusters

```

% Event 5: Using thrusters for ACS
%-----
k = k + 1;
event(k).type = 'acs';
event(k).description = 'ACS_Disturbance';
event(k).acSThruster = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster = [];
event(k).dVTTotal = [];
event(k).dHTTotal = [];
event(k).dTControl = 1;
event(k).disturbance = [0;1;0];
event(k).nPulses = [];
event(k).duration = 10000;
event(k).regulated = 0;
event(k).cM = [0;0;0];

```

The next listing shows an example of firing a few pulses.

#### Listing 19.10: Fuel Budget: Firing a few pulses

```

% Event 6: Firing a few pulses
%-----
k = k + 1;
event(k).type = 'pulsed';
event(k).description = 'Short_Pulse';
event(k).acSThruster = [1 2];
event(k).dVThruster = [];
event(k).dVTTotal = [];
event(k).dHTTotal = [];
event(k).disturbance = [];
event(k).nPulses = []; % Wont use any fuel this way
event(k).pulsewidth = 0.05;
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];

```

The results for these events are shown in Listing 19.11.

```

FBudget ( event, x ) % Generates the following results

```

#### Listing 19.11: Fuel Budget Example, Six Events

```

TestBudget Propellant budget 19-Sep-1997
Item Description Total Fuel Remaining Fuel Used Pressure ACS Isp DV Isp
1 Initial State 1190.92 117.89 117.89 0.000 0.000 2.4e+06 2.4e+06 0.00 0.00
2 Delta V Continuous 1153.87 99.36 99.36 18.525 8.525 2.4e+06 2.4e+06 100.02 360.00
3 Delta V Off Pulse 1099.18 72.02 72.02 27.346 27.346 1.7e+06 1.7e+06 100.00 210.00
4 Delta H 1099.18 72.02 72.02 0.001 0.001 1.2e+06 1.2e+06 100.41 0.00
5 ACS Disturbance 1087.45 66.16 66.16 5.862 5.862 1.2e+06 1.2e+06 100.39 0.00
6 Short Pulse 1087.45 66.16 66.16 0.000 0.000 1.2e+06 1.2e+06 100.50 0.00

```

For east/west stationkeeping the delta V required because of longitude drift, due to the earths tesseral gravitational harmonics is computed by

```
[dVEW,dTEW] = DVLDrift( box, scLon );
```

where the first number is the size of the stationkeeping box in degrees and the second is the spacecraft longitude. `DVLDrift` outputs the delta V required per maneuver and the time between maneuvers. For north/south stationkeeping the corresponding function is

```
[dVNS,dTNS] = DVIDrift( box, year );
```

where the second argument is the year for which the calculation is done. The inclination drift is a function of the earth/moon geometry which is included in `DVIDrift`.



# ORBIT REFERENCES

---

This chapter gives selected orbit resources.

## 20.1 Orbits

---

Excellent applied navigation and interesting mathematical approaches to orbit dynamics. Prof. Battin gives extensive information on the Apollo navigation system.

Battin, R. H. “An Introduction to the Mathematics and Methods of Astrodynamics.” AIAA Education Series.

The most comprehensive and up-to-date book on applied orbit dynamics.

Vallado, David. A. and W. D. McClain. *Fundamentals of Astrodynamics and Applications, Second Edition*. Microcosm Press, El Segundo, CA, 2001.

A good introduction with many physical insights.

Wiesel, W. E. *Spaceflight Dynamics*. McGraw-Hill, 1989.

## 20.2 Web Sites

---

Gravity models have been available at

<http://cddis.nasa.gov/926/egm96/>

[http://www.iges.polimi.it/pagine/services/repo/repo\\_model.asp](http://www.iges.polimi.it/pagine/services/repo/repo_model.asp)

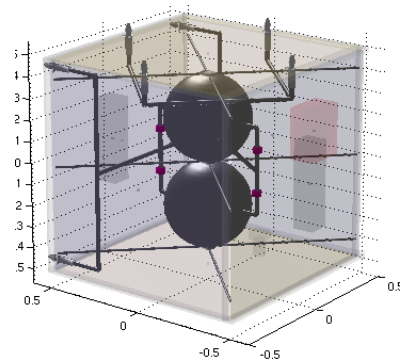
JPL solar systems dynamics information, including the Horizons ephemeris tool and astrodynamics constants, is available at

<http://ssd.jpl.nasa.gov/>



# PART IV

# SUBSYSTEMS ANALYSIS





# IMAGING

---

This chapter discusses how to use the Imaging module.

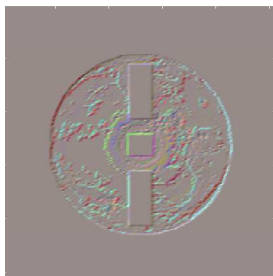
## 21.1 ImageMatching

---

ImageMatching has the Canny Edge Detection algorithm, image derivative and a function for fitting a circle to points in an image.

**Figure 21.1:** CannyEdgeDetector demo

---



## 21.2 ImageProcessing

---

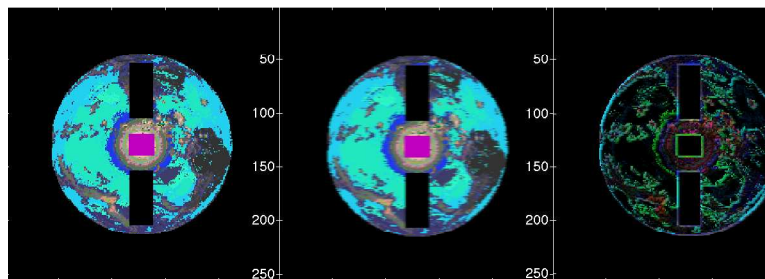
This folder includes utilities for image processing

1. GaussianImageFilter

2. `IndexedImageToRGB`
3. `MedianImageFilter`

For example `GaussianImageFilter`.

**Figure 21.2:** `GaussianImageFilter` demo



The first image shows the original, the second the filtered and the third the difference. This function handles images of the form  $a(n, m, 3)$ . The matrices must be doubles.

# LINK MODULE

---

The link module provides basic functions for computing link margins. Link margins are the signal to noise ratios for communication links between spacecraft and between spacecraft and the ground. The module is divided into utilities, design functions, and RF and optical link analysis functions. The list of folders is shown below.

```
>> help Link
PSS Toolbox Folder Link
Version 8.0      03-Dec-2009

Directories:
Coverage
Demos
Demos/Optical
Demos/RF
Help
Ladar
LinkDesign
LinkUtilities
Optical
RF
Radar
```

## 22.1 Link Utilities

---

Link utilities are for conversions and other basic link calculations. For the complete listing type

```
>> help LinkUtilities
```

and you will get the list of functions with a help comment.

```
Link/LinkUtilities

B
  BEP - Computes the bit error probability given C/N and the bit rate.

C
  CNRComp - Compute the carrier to noise ratio.

D
  DBPower - Power conversion to decibels. dB = 20 log10(s).
  DBSignal - Signal conversion to decibels. dB = 10 log10(s).
```

```

DBSignalToPower - Convert decibels to power.
DBWToNW - Convert dBW (decibels-Watts) to nanowatts.
DBWToPW - Convert dBW to picowatts.
DopplerBlind - Finds Doppler blind speeds.
DopplerShift - Doppler shift

F
FPD - Function called by fzero to compute the carrier to noise ratio.
FPFA - Right hand side for probability of false alarm computation.
Frequency - Outputs the frequencies for specified bands of electromagnetic
radiation.
FrequencyToWavelength - Converts frequency to wavelength.

P
ProbDetection - Probability of detection for a diffuse target.

S
Shannon - The Shannon information theorem relating S/N and C/B.

T
TToPower - Convert temperature to power.

W
Wavelength - Outputs the wavelengths for bands of electromagnetic radiation.
WavelengthToFrequency - Converts wavelength to frequency.

Y
YBComp - yB computation used as part of the probability of detection
algorithm.

```

For example to get nanowatts from dB,

```

>> power = DBWToNW(-106)
power =
    2.5119e-02

```

or picowatts

```

>> power = DBWToPW(-106)
power =
    2.5119e+01

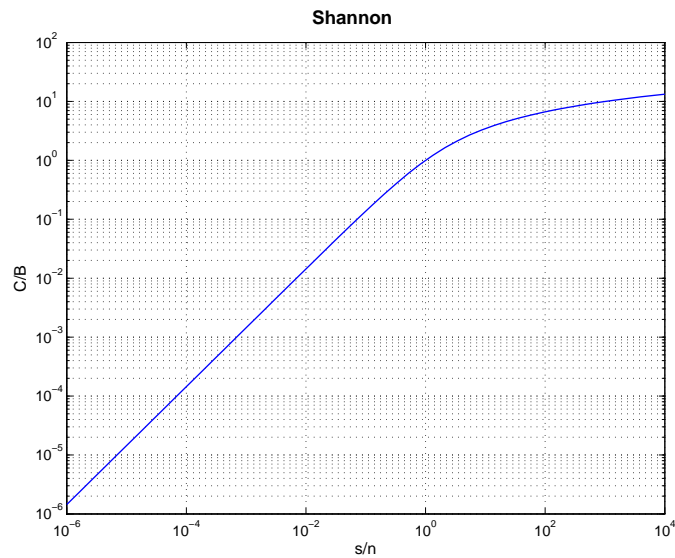
```

If you type `Shannon` you get [Figure 22.1 on the next page](#) (a demo of the Shannon function). This relates the information content to the signal to noise ratio.

## 22.2 Optical

Two functions are available for optical calculation. The first, `OpticalBackgroundNoise`, computes the optical background noise for

- Solar
- Lunar
- Mercury
- Venus
- Mars

**Figure 22.1:** Shannon demo

- Jupiter
- Saturn
- Sunlit clouds
- Sunlit snow and ice
- Starfield

The second, `OpticalReceiverNoiseDensity`, computes optical receiver noise density for coherent and direct receivers.

## 22.3 RF

The RF functions compute various losses and noise temperatures. For example typing

```
LossAtmosphericGas
```

gives you path losses due to atmospheric gas, see [Figure 22.2 on the following page](#). Other loss functions include `LossFreeSpace` and `LossPrecipitation`. Typing

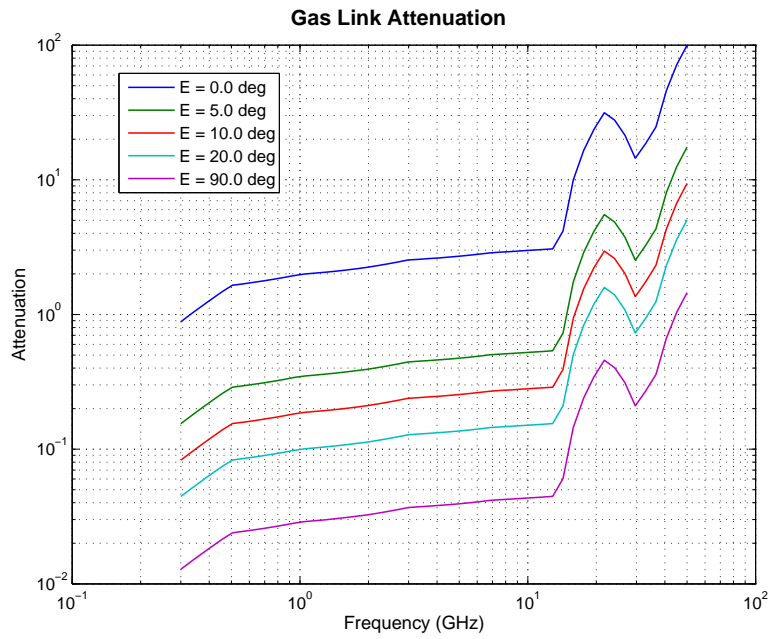
```
TSky
```

gives you a sky temperature demo, see [Figure 22.3 on the next page](#). You can also get temperatures for the moon, sun, receiver, attenuator, etc.

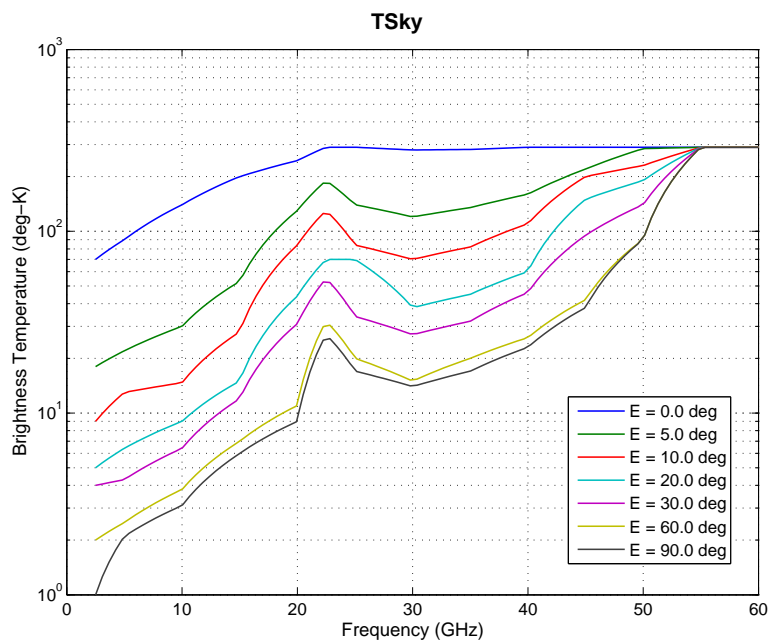
You can compute losses and noise inputs using a variety of functions in the toolbox. All of these tools are integrated into the `LinkAnalysisTool`. The built-in demo of this function is shown below and the results in [Figure 22.4 on page 183](#).

```
% Space link
%-----
spaceLink.theta3DB      = 2; % (deg)
spaceLink.fIllumination = 70; % (deg)
spaceLink.effTransmit   = 0.55;
```

**Figure 22.2:** Atmospheric Gas Loss



**Figure 22.3:** Sky temperature





```

spaceLink.powerTransmit      = 10; % dBW
spaceLink.lossEdge           = 3; % dB
spaceLink.lossTransmitterFeed = 0.5;
spaceLink.frequency         = 12;
spaceLink.diameterAperture  = 1;
spaceLink.thetaDepointing   = 0.1; % (deg)

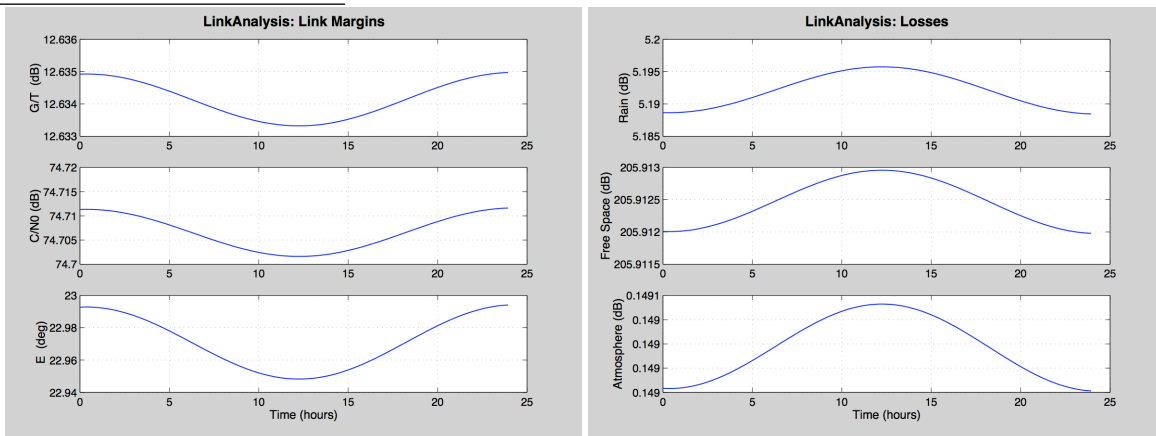
% Ground link
%-----
groundLink.antenna.type      = 'circularAperture';
groundLink.antenna.area     = pi*2^2/4;
groundLink.latitude         = 45;
groundLink.longitude        = 0;
groundLink.altitude         = 0;
groundLink.horizonAngle     = 0;
groundLink.tGround          = 45; % Ambient ground temperature
groundLink.tAttenuatorAmbient = 290; % Ambient attenuator temperature
groundLink.lossAttenuator    = 1; % Attenuator loss (dB)
groundLink.lossFeeder       = 0.5;
groundLink.climateZone      = 'e';
groundLink.precipitationFractionOfTime = 0.01;
groundLink.polarizationTiltAngle = 0;

% Orbit
%-----
orbit.el = [42164 0 0 0 0 0];
orbit.jD = DateStringTOJD( '11/04/2004' );
orbit.t = linspace(0,Period(orbit.el(1)));

LinkAnalysisTool( spaceLink, groundLink, orbit );

```

Figure 22.4: Results of LinkAnalysisTool



## 22.4 Downlink Analysis Example

This example shows how to do a downlink analysis. The reference is Maral, G. and M. Bousquet. *Satellite Communications Systems*, Third Edition. John Wiley (1998). See this and more examples in `LinkExamples.m`.

```

%-----
% Example 2 pp. 24-25
%-----

```

```

fD      = 12; % GHz
pT      = DBSignal(10); % dBW
r       = 40000; % km
theta3DB = 2;
effT    = 0.55;
effR    = 0.6;
dR      = 4;
fillumination = 70;

gT      = Gain3dB( theta3DB, fillumination, effT );
eIRPSL  = pT + gT;
gR      = AntennaGain(struct('type','circularAperture','area',pi*dR^2/4,'
    eff',effR),fD);
lFS     = LossFreeSpace( fD, r );
pR      = eIRPSL - lFS + gR;

disp(sprintf('\n\n2.3.2_Example_2:_The_Downlink\n-----'))
disp(sprintf('Gain_Transmit_Antenna_=_%10.2f_(dBW)',gT));
disp(sprintf('EIRP_SL_=_%10.2f_(dBW)',eIRPSL));
disp(sprintf('Free_Space_Loss_=_%10.2f_(dB)',lFS));
disp(sprintf('Gain_Receive_Antenna_=_%10.2f_(dB)',gR));
disp(sprintf('Power_Received_=_%10.2f_(dBW)',pR));
disp(sprintf('Power_Received_=_%10.2f_(pW)',DBWToPW(pR)));

```

The only loss is the free space loss which is due to the  $1/r^2$  dissipation of the radio waves with distance in free space. The result of this example is

```

2.3.2 Example 2: The Downlink
-----
Gain Transmit Antenna = 38.23 (dBW)
EIRP SL = 48.23 (dBW)
Free Space Loss = 206.07 (dB)
Gain Receive Antenna = 51.81 (dB)
Power Received = -106.03 (dBW)
Power Received = 24.94 (pW)

```

With a distance of 40,000 km the free space loss dominates the losses. Only 24.94 picowatts is delivered to the receiver. This analysis ignores other losses such as receiver loss, loss due to atmosphere and loss due to rain. In addition it is not a complete example since we really need to know the signal to noise ratio at the receiver to determine how much information we can transmit.

## 22.5 LADAR

Two main functions are available for analyzing LADAR systems. `OpticalLinkBudget` is used to compute link budgets for optical systems. The following example shows how to assign values to the data structure. Note that losses are from 0 to 1 not dB. 1 is lossless and 0 is infinite losses.

```

disp('Pluto_Mission');
d.wavelength      = 532e-9; % nd:YLF (Neodymium doped: Ytterbium Lithium
    Floride)
d.transmitApertureDiameter = 0.2; % M
d.thetaOffset     = 0; % No offset
d.transmitPower   = 100; % W
d.wavefrontError  = 0; % Loss is exp(-2*pi*error/wavelength)
d.range          = 41*149.6e9; % 41 AU
d.receiverLoss   = 1; % 0 to 1, 1 means no loss
d.receiverPointingLoss = 1; % 0 to 1, 1 means no loss

```

```
d.receiveApertureDiameter = 1; % Telescope diameter (m)
OpticalLinkBudget( d );
```

This example results in for a total link loss of -292.7 dB. Compare this with the results given in the previous section!

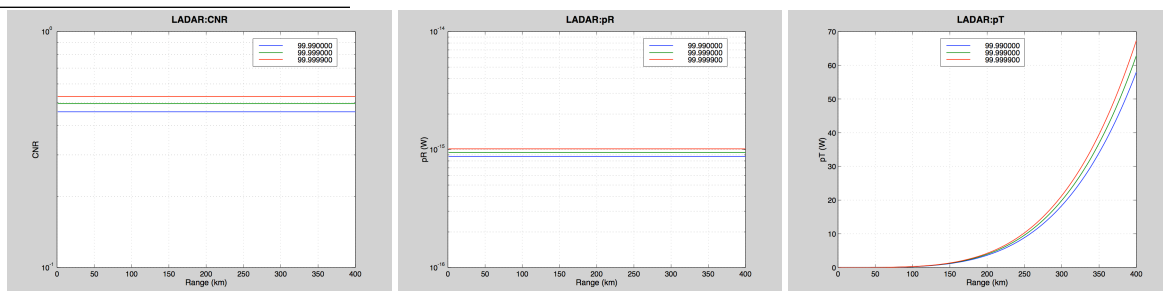
```
Pluto Mission
Transmit Power      =      100.0000 W
Wavelength          =      532.0000 nm
Theta Divergence    =      3.3868e-06 rad
Beamwidth at Range  =      2.0773e+07 m
Pointing Loss       =      1.0000e+00
Transmit Antenna Gain =      1.3949e+12
Range Loss          =      4.7640e-41
Wavefront Loss      =      1.0000e+00
Receive Antenna Gain =      3.4872e+13
Receiver Pointing Loss =      1.0000e+00
Receiver Loss       =      1.0000e+00
Receive Power       =      2.3173e-13 W
Losses are 0 to 1. 1 means no loss
```

The transmit power can be computed using LADARTransmitPower.

```
d.b      = 30e3;
d.n      = 500;
d.etaM    = 0.5;
d.d      = 0.2;
d.etaR    = 0.7;
d.theta   = 90e-6;
d.k      = 2;
d.etaT    = 0.7;
d.etaQ    = 0.8;
d.etaE    = 0.7;
d.pFA     = 1e-6;
d.pD      = 1 - [1e-4 1e-5 1e-6];
d.xC      = 1;
d.lambda  = 11.149e-6;
d.r       = linspace(1,400)*1000;
d.tA      = 1;
d.thetaT  = 0;
d.x       = d.r/1000;
d.xLabel  = 'Range_(km)';
LADARTransmitPower( d );
```

This function generates 3 figures shown in Figure 22.5.

**Figure 22.5:** Results of LADARTransmitPower



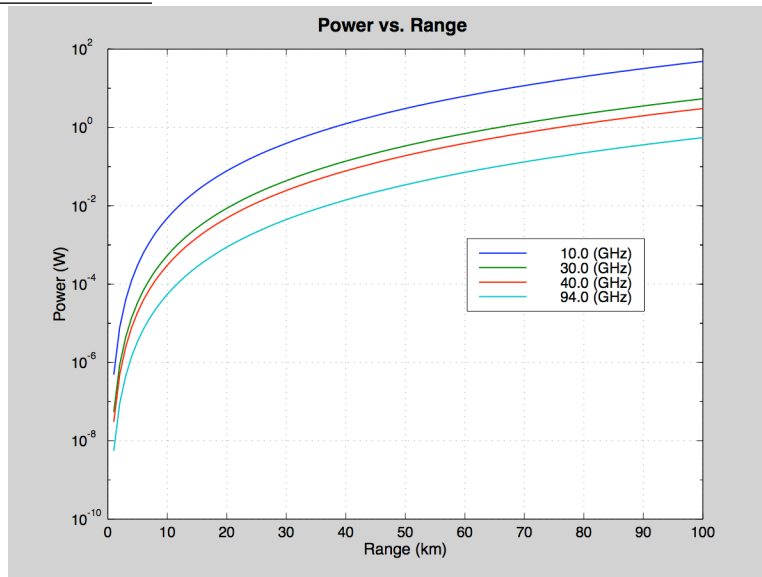
## 22.6 RADAR

There are many tools available for radar analysis. The radar equation may be computed in various forms by:

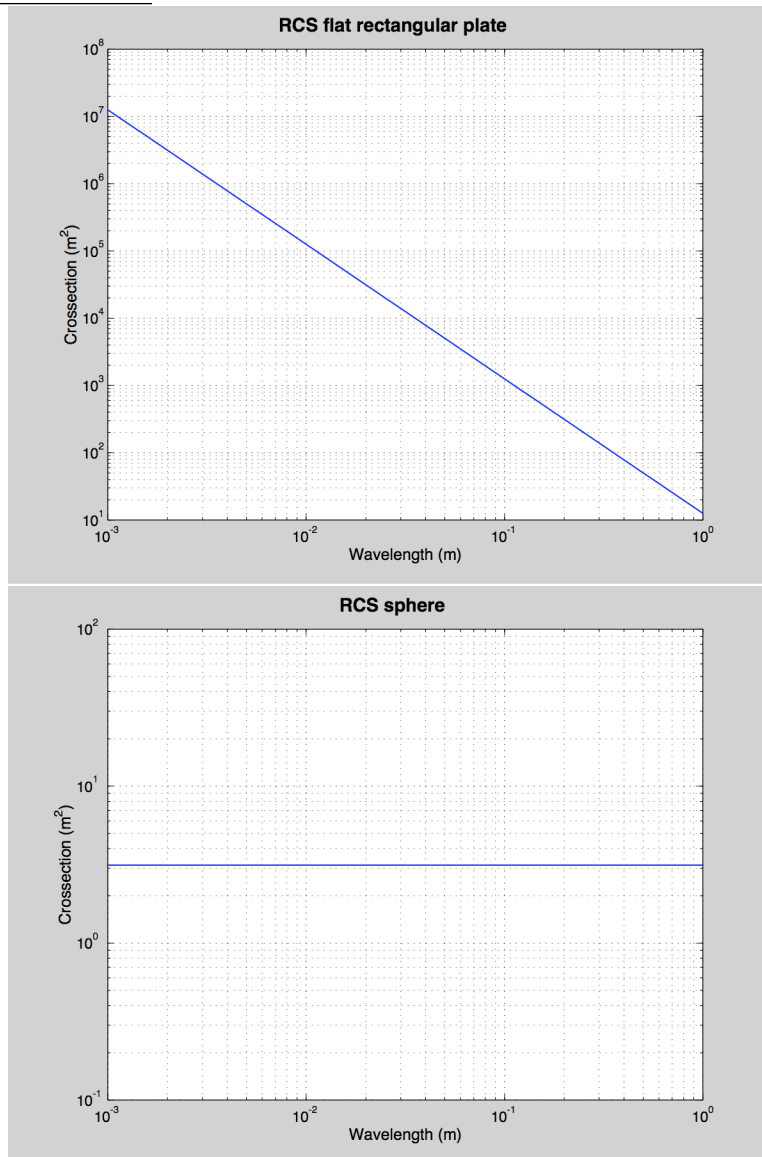
```
RadarEquationPulse.m  
RadarEquationSN.m  
RadarEquation.m
```

Most have demos. For example if you type `RadarEquation` you will see Figure 22.6.

**Figure 22.6:** Results of `RadarEquation`



Type `RadarCrosssection` and you will get the radar crosssections of a plate and sphere.

**Figure 22.7:** Results of RadarCrosssection



---

# POWER

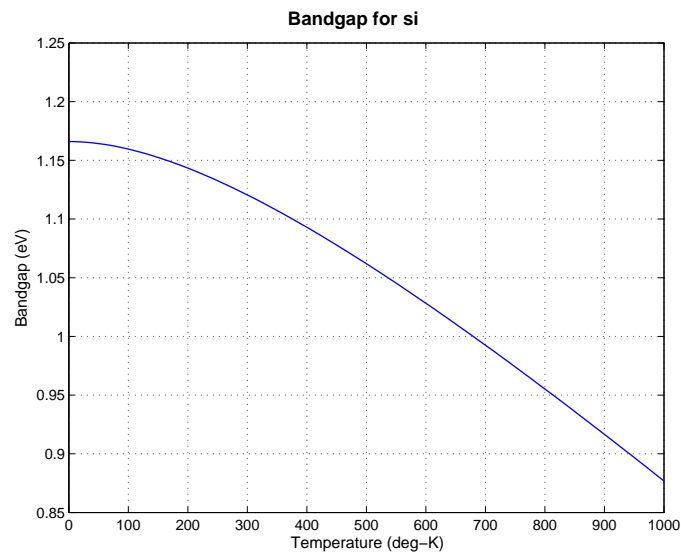
---

This chapter discusses how to generate use the power module.

The `Source` folder provides functions for computing parameters for solar cells.

`BandGap` computes the band gap for solar cells as a function of temperature, as shown in Figure 23.1.

**Figure 23.1:** Band gap



`SolarCell` computes the I-V curves for solar cells as in Figure 23.2 on the following page.

`SolarCellEff` computes the efficiency for solar cells as in Figure 23.3 on the next page.

Figure 23.2: Solar cell I-V curves

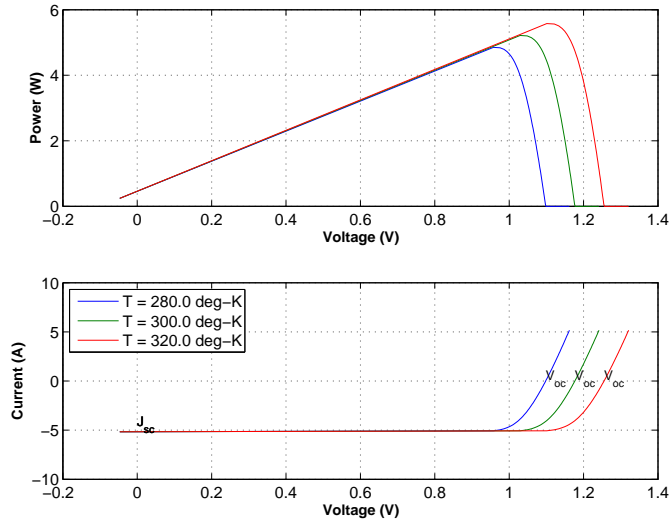
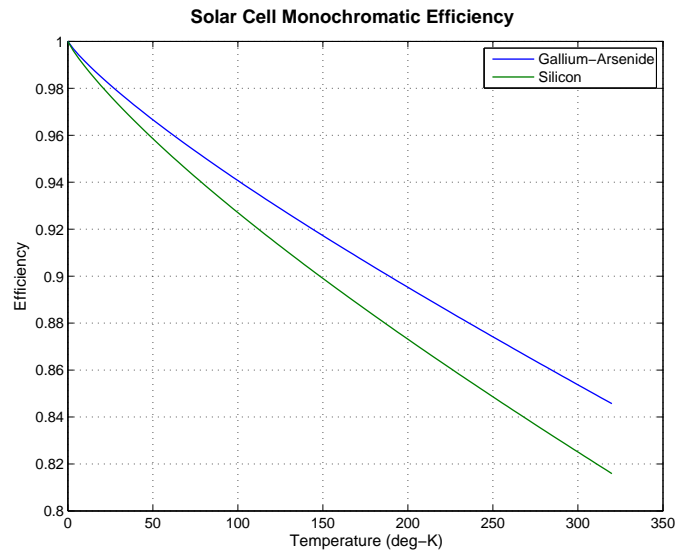


Figure 23.3: Solar cell efficiency





# PROPULSION MODULE

---

This chapter presents the propulsion module.

## 24.1 Module Overview

---

The propulsion module has several broad categories

- Chemical propulsion
- Electric propulsion
- Multistage rocket analysis
- Rocket

The chemical category includes functions for modeling and designing blowdown systems and functions for analyzing the chemistry or chemical rockets. The electric propulsion category includes functions for analyzing and optimizing electric propulsion systems. The multistage rocket analysis functions are for launch vehicle design and the rocket functions are functions that apply to all types of rockets.

The complete list of folders is shown below.

```
>> help Propulsion
PSS Toolbox Folder Propulsion
Version 8.0      03-Dec-2009

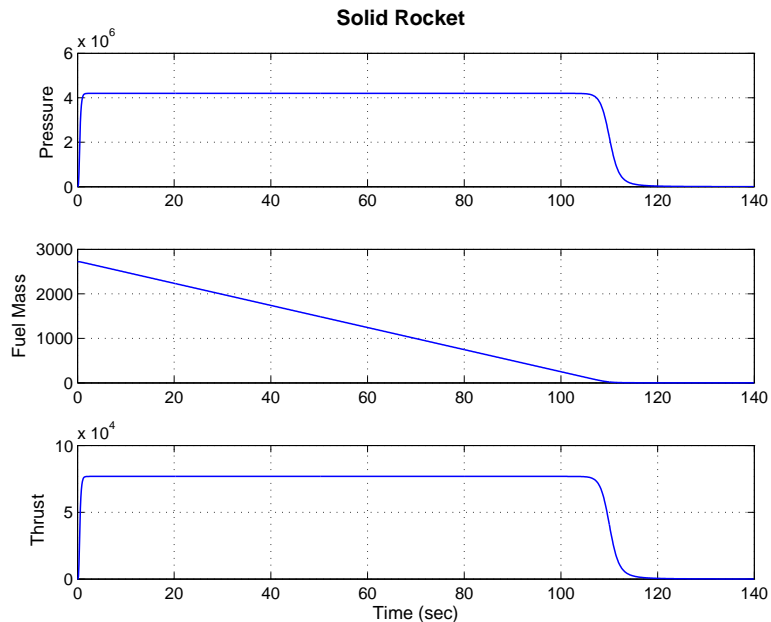
Directories:
Airbreathing
Chemical
Demos
Demos/Airbreathing
Demos/Chemical
Demos/LVModels
Demos/MultiStage
Demos/Rocket
Electric
Help
LVModels
LaunchSim
MultiStage
```

Nuclear  
PropulsionData  
Rocket

## 24.2 Solid Rocket Motor

`TSolid` is an example of a solid motor simulation. The function `SolidR`, which models the dynamics of the burning motor, is numerically integrated in a loop. See Figure 24.1 for the demo results.

Figure 24.1: Solid motor firing



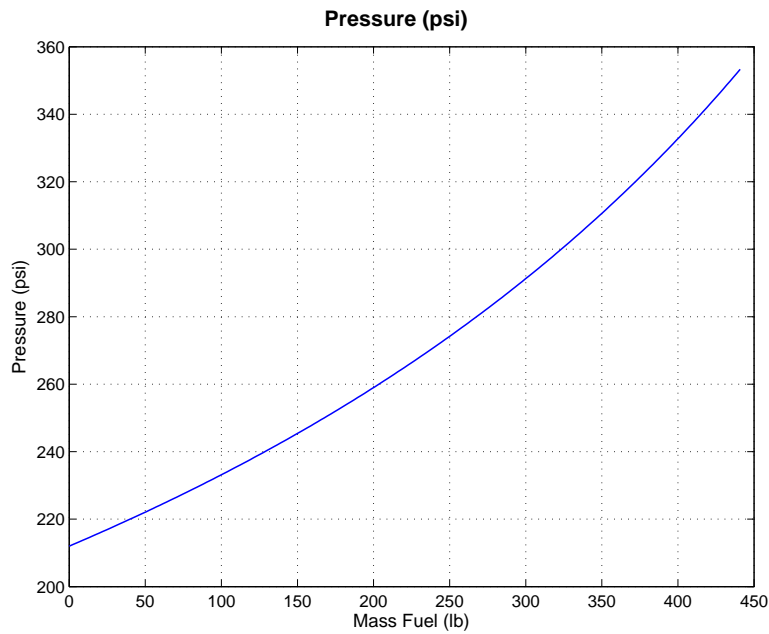
## 24.3 Blowdown Curve

Designing a blowdown system is of critical importance for most satellites. You need to specify the mass of fuel, initial pressure and final pressure. The pressure are set by the thrusters. The following code designs a blowdown system.

```
Nm2ToPSI = Constant('N/m^2_to_PSI');
KgToLbF = Constant('Kg_to_Lb_Force');
molWt = 0.004;
rPress = MolWt2R(molWt);
mFuel = 0:200;
rhoFuel = 1000;
T = 293;
vTank = 0.5;
mPress = 1.2;
p = BloDown(mPress, rhoFuel, vTank, rPress, T, mFuel);
Plot2D(mFuel*KgToLbF, p*Nm2ToPSI, 'Mass_Fuel_(lb)', 'Pressure_(psi)')
```

See Figure 24.2 on the next page. With no fuel, the system pressure is 210 psi. With the full fuel load the pressure is 350 psi.

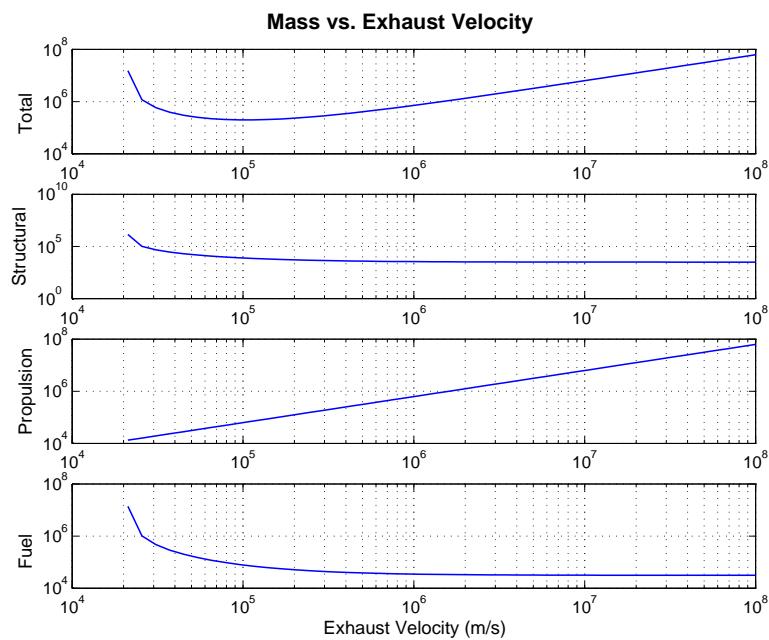
Figure 24.2: Blowdown curve



## 24.4 Electric propulsion

Type `ElectricPropulsion` and you will get the plot in Figure 24.3. This shows the mass of an electrically propelled vehicle as a function of exhaust velocity. The optimal exhaust velocity is surprisingly low.

Figure 24.3: Optimal electric vehicle



The vehicle characteristics are printed in the MATLAB command window:

```

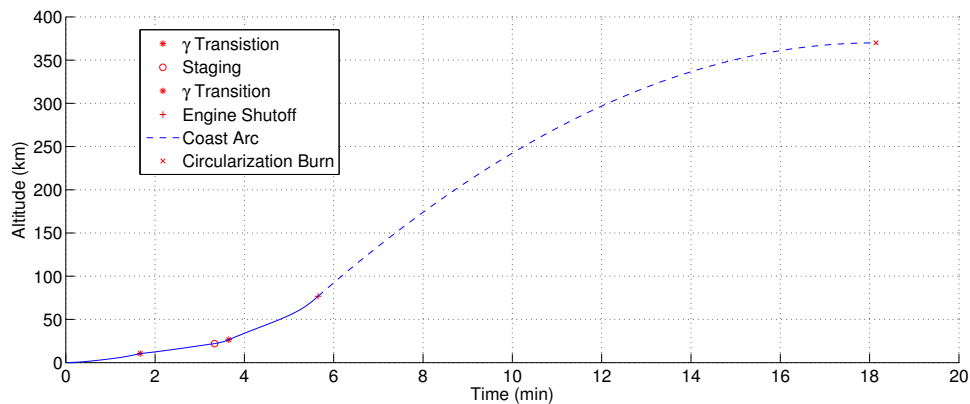
mP: 50000
thrust: 15000
dV: 50000
pR: 20000
sR: 1.0000e-01
eff: 6.0000e-01
uE: 1.0381e+05
power: 1.2976e+09
mR: 6.4880e+04
mF: 7.5771e+04
mS: 7.5771e+03
mD: 1.2246e+05
mT: 1.9823e+05
mDot: 1.4450e-01

```

## 24.5 Launch vehicle analysis

Type `TSTODemo` for a simulation of a two-stage-to-orbit horizontal takeoff/horizontal landing launch vehicle. The first stage is an air-breathing vehicle powered by two turbofan and two hydrogen-fueled ramjet engines. The transition from turbofan to ramjet power occurs at Mach 1.5. The second stage is powered by a hydrogen rocket. Type `help TSTODemo` for a description of the mission. The flight path of both stages and also the staging conditions have been optimized to maximize the mass to orbit. The optimization has been previously performed with `OptimizeTSTO`, which requires MATLAB's Optimization Toolbox. Figure 24.4 shows the results of the demo, in which a payload of 593 kg is delivered to orbit.

**Figure 24.4:** Two-State-to-Orbit Simulation





# THERMAL MODULE

---

This chapter introduces the thermal analysis module.

## 25.1 Module Overview

---

The thermal module provides basic functions for performing thermal analysis of spacecraft. It is not meant to be a substitute for packages such as TRASYS or SINDA. Rather, it is useful to help GN&C analysts uncover potential thermal problems caused by GN&C activity.

There are two classes of functions. One class is the properties functions. The other is the thermal analysis functions. The complete list of folders is shown below.

```
>> help Thermal
PSS Toolbox Folder Thermal
Version 8.0      03-Dec-2009

Directories:
Demos
Demos/ThermalAnalysis
Demos/ThermalGraphics
ThermalAnalysis
ThermalData
ThermalGraphics
ThermalProperties
```

## 25.2 Thermal Properties

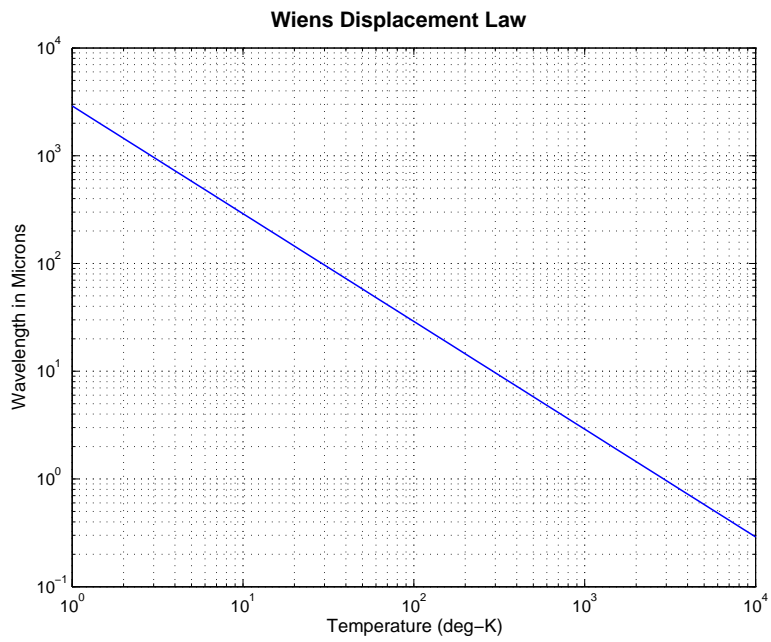
---

These functions generate information about the thermal properties of objects. For example typing `WiensDsp` computes the peak black body wavelength as a function of temperature, as shown in Figure 25.1 on the following page.

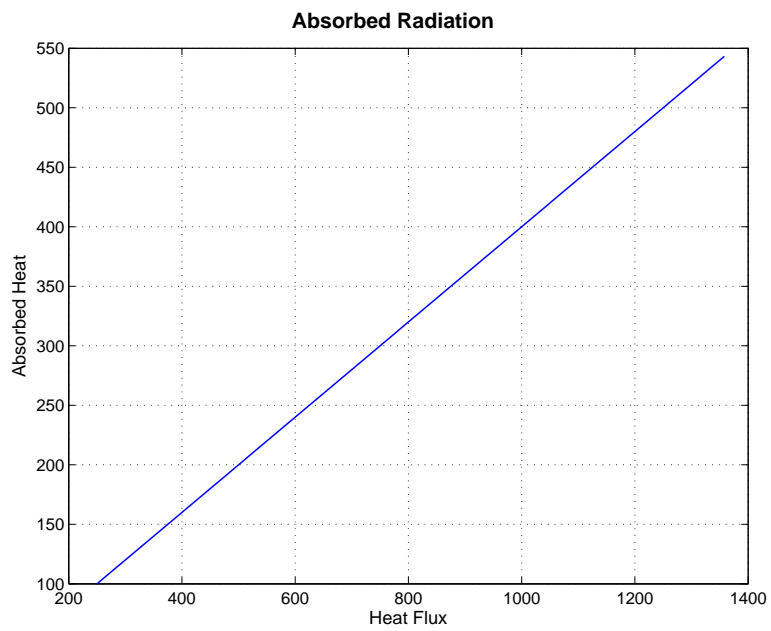
`AbsDa` computes the absorbed heat as a function of absorptivity and incoming flux. See Figure 25.2 on the next page. Other functions compute the temperate of simple geometric objects under incoming radiation.

Another function is `PlanckL` which gives Plancks Law. See Figure 25.3 on page 199.

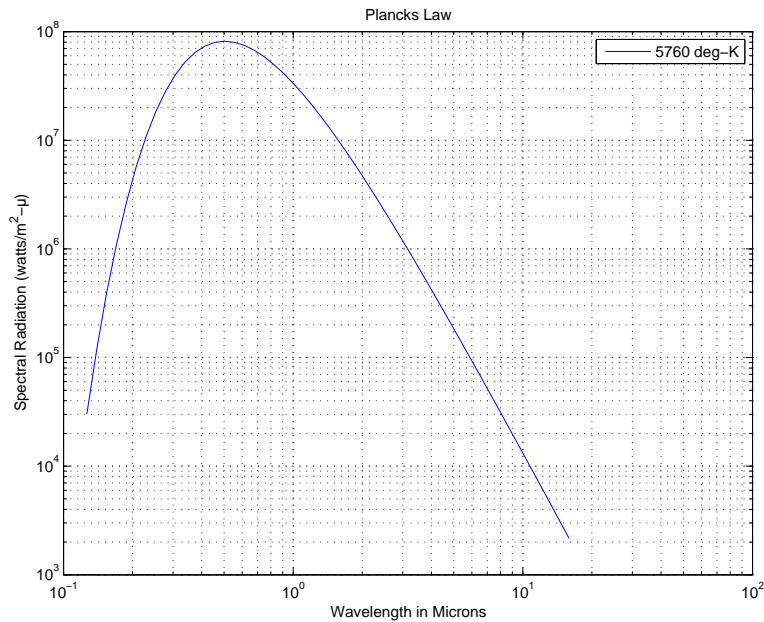
**Figure 25.1:** Black body wavelength as a function of temperatures



**Figure 25.2:** AbsDa demo





**Figure 25.3:** Plancks Law

## 25.3 Thermal Analysis

There are two thermal analysis functions. One is `Thermnet` which generates a state space system for a general thermal system including radiation heat transfer. This is done by analytically linearizing the equations about a nominal temperature. You can specify any connections between nodes and declare external nodes for heat flux into the system.

The second function is `ThermalEquilibrium`. This function computes the thermal equilibrium for a system governed by the static equation

$$lT^4 + kT = bq \quad (25.1)$$

using MATLAB's sparse matrix functions.

### 25.3.1 Thermnet Example

`Thermnet` creates a set of first order differential equations to model a thermal circuit. Inputs are the nodal masses, a conductive node map, a radiative node map, an input node map and an output node map. The output is a set of first order differential equations in the form

$$\begin{aligned} \dot{T} &= aT + bq \\ y &= cT \end{aligned} \quad (25.2)$$

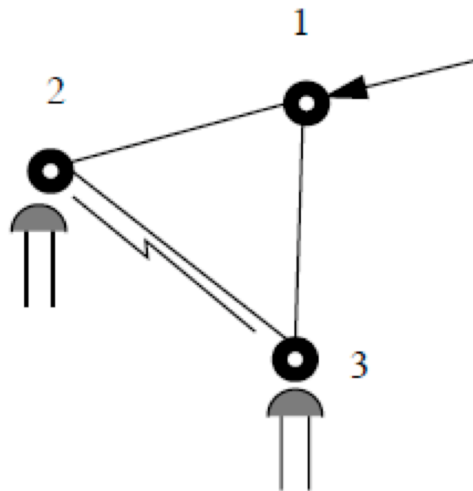
where  $T$  is the nodal temperature,  $q$  is the thermal input and  $y$  is the vector of measurements.

A node map has one row for each node and as many columns as are necessary to specify the connections. Each node map is accompanied by a coefficient map that gives either the conductive or radiative heat transfer coefficients for the node pair. `Thermnet` will automatically make the a matrix symmetric for conduction. `Thermnet` will warn you if you entered a coefficient twice and the two were different. You only have to specify the connection in one direction.

Assume that we want to model the system shown in Figure 25.4 on the next page.

External flux is coming in on node 1 and measurements are taken on node 3 and 2. There is conduction heat transfer between all nodes, but radiation transfer only between nodes 3 and 2, as indicated by the zigzag line.

Figure 25.4: Thermal network



Assume that the thermal mass of each node is one and that the nominal temperature is 300 deg-K. Then

```
massNode = [1 1 1];
tNominal = [300 300 300];
```

The input node is 1 so

```
inputNodeMap = 1;
```

and the output nodes are 2 and 3

```
outputNodeMap = [2 3];
```

There is one row per node in the conduction node map. All nodes are connected to all other nodes so the conduction node map is

```
conductionNodeMap = [2 3;1 3;1 2];
```

and the radiation node map is

```
radiationNodeMap = [inf;3;2];
```

The first node radiates to infinity. If there are different numbers of connections per node, then fill in zeros to make the matrix full. The coefficients are

```
conductionCoeff = [1 2;1 2;2 2];
radiationCoeff = [1;1;1]*0.25e-6
```

Again, each row is for a node and the column corresponds to the index in the node map. For example, the conduction coefficient between node 1 and 2 is 1. Type

```
[a, b, c] = Thermnet( massNode,...
                    conductionNodeMap, conductionCoeff,...
                    inputNodeMap, outputNodeMap,...
                    radiationNodeMap, radiationCoeff,...
                    tNominal )
```

and the result is

```
a =
   -30     1     2
     1   -30    29
     2    29   -31
```

```
b =
    1
    0
    0
c =
    0    1    0
    0    0    1
```

### 25.3.2 Thermal Equilibrium Example

The simplest case is two nodes connected by a conductor. Heat is coming through node 1. Both nodes are radiating to free space.  $k$  is the linear term coefficient and  $l$  is the quartic coefficient.  $b$  is the input matrix. The input flux  $q$  is set to the typical value of the solar flux at the Earth. The input node temperatures  $t$  are guesses or prior values. The default number of iterations is 40 and the default tolerance is MATLAB's `eps` parameter ( $2.2204e-16$ ).

```
d.k = 10*[1 -1;-1 1];
d.l = 5.67e-8*[1 0;0 1];
q   = 1358;
t   = [300;300];
d.b = [1;0];

[t, delta] = ThermalEquilibrium( d, q, t )

t =
    3.5236e+02
    3.0396e+02
delta =
    8.5508e-16
```

As expected, node 1 is hotter than node 2, by about 50 degrees. This is a good model for a solar array.

### 25.3.3 Thermal Imaging

`SCThermalDemo` demonstrates thermal imaging code. The script draws a view of the spacecraft as seen in “natural” lighting and then draws one which shows an image as function of temperature. Figure 25.5 on the following page shows the four windows that are generated.

This simulation employs the model generated in `BuildSCModelForImaging` which is a simple spacecraft model.

```
g = load('SCForImaging');
```

Two different cameras are used. `SpaceCamera` shows the spacecraft in solar lighting. In figure ambient lighting is set high so that it is clearly visible. `ThermalImager` generates the thermal color data. This script employs the standard PSS CAD model format with the fields for position and quaternion set in the script.

```
% CAD body structure
%-----
g.body(1).bHinge.q      = QPose(qLVLH);
g.body(2).bHinge.angle = theta + pi;
g.body(3).bHinge.angle = theta + pi;
g.rECI                  = rECI(:,k);
g.qLVLH                 = qLVLH;
```

While the script is running you can end it and plot all temperatures by pushing the plot button.

Figure 25.5: Thermal imaging demo

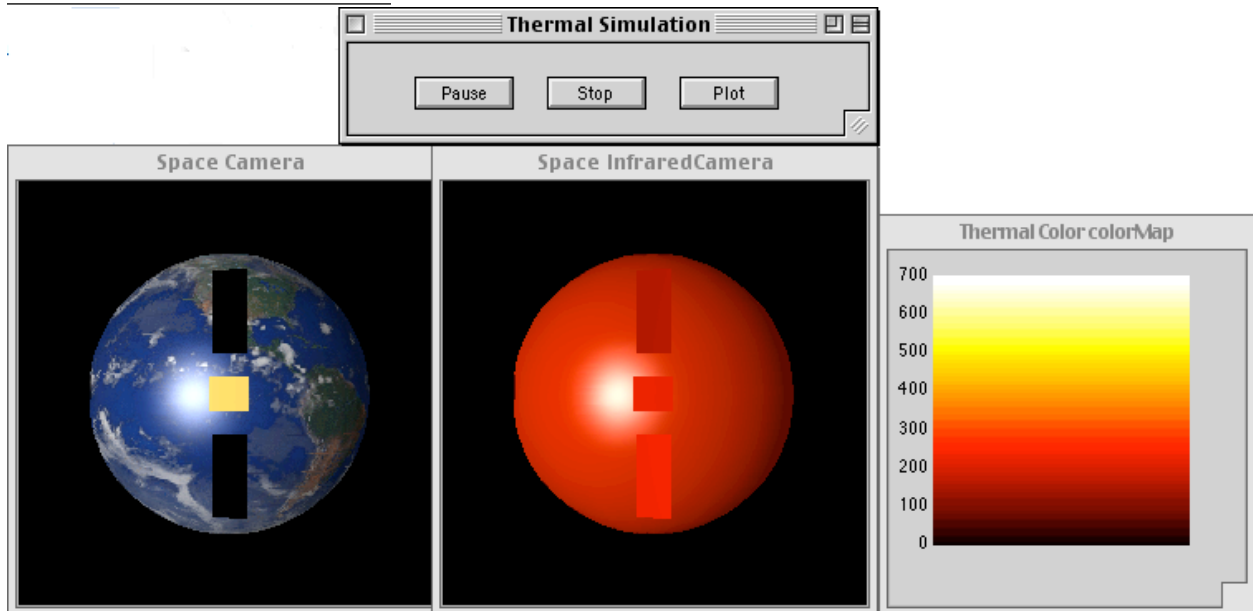
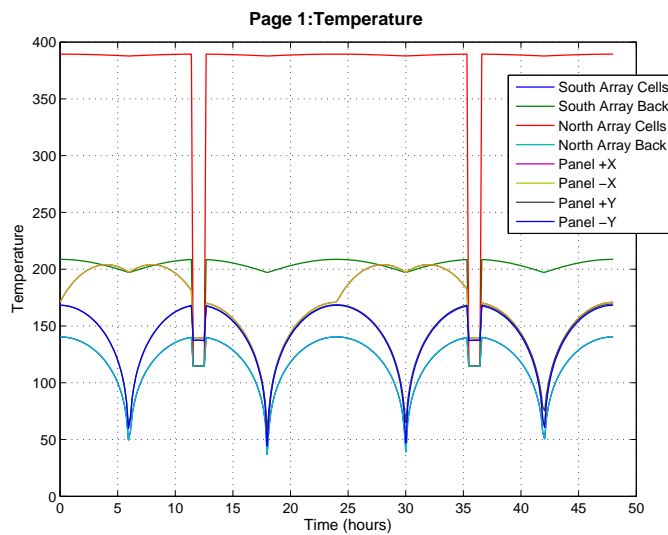
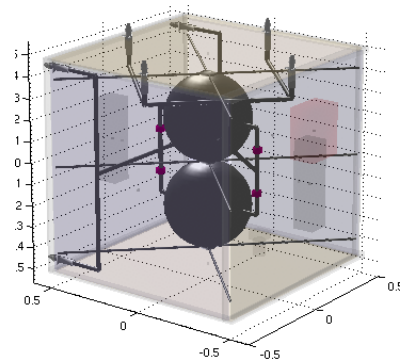


Figure 25.6: Thermal imaging demo temperature plot



## PART V

# ATTITUDE AND ORBIT ESTIMATION





# INTRODUCTION

---

## 26.1 Overview

---

The Estimation module contains software for developing estimation algorithms. The algorithms can be used to solve any estimation problem. Particular technologies included are

1. Fixed Gain Kalman Filters
2. Variable Gain Kalman Filters
3. Extended Kalman Filters
4. Unscented Kalman Filters
5. Fault Detection

Topic areas discussed in this part of the guide are

1. Recursive attitude determination
2. Recursive orbit determination
3. Stellar attitude determination

This guide covers the basics of operating the software. Estimation theory is covered in the accompanying book, *Spacecraft Attitude and Orbit Control*.





# GENERAL ESTIMATION

---

## 27.1 Introduction

---

This module focuses on recursive estimation. Recursive estimation is simply taking a measurement and using it to update an existing estimate of the state of the system. The measurement need not be a state, but for all states to be observed the measurement must be related to the states or derivatives of the states.

In this module, recursive estimators can be divided into six classes:

1. Fixed gain estimators
2. Variable gain estimators for linear systems
3. Variable gain estimators for nonlinear systems in which the plant (the state dynamics) are linearized about the current state and propagated as a linear system. This type of system is used for attitude estimation. This is also called the Extended Kalman Filter.
4. Variable gain estimators for nonlinear systems in which the plant is numerically integrated. This type of system is used for orbit estimation. This is also called the Continuous Discrete Extended Kalman Filter.
5. Unscented Kalman Filters. These filters compute multiple states and covariances for each in effect computing the statistics online.

This chapter will discuss estimation in general with examples from attitude and orbit estimation.

## 27.2 Fixed Gain Estimators

---

Fixed gain estimators are written in the form

$$x_{k+1} = ax_k + bu_k + k(y - cx_k) \quad (27.1)$$

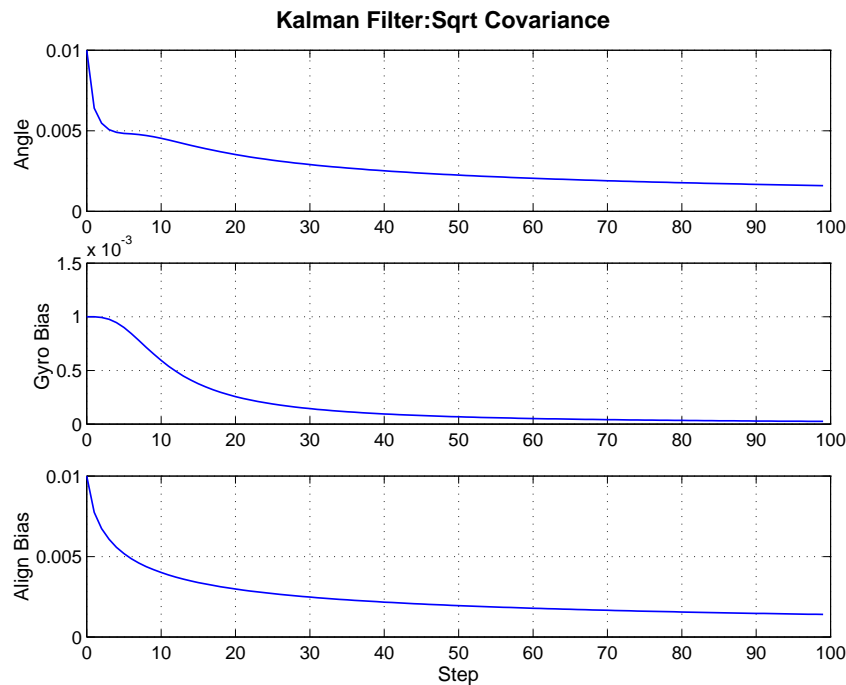
where  $x_k$  is the state vector at step  $k$ ,  $a$  is the state transition matrix,  $b$  is the input matrix,  $u_k$  is the input vector,  $y$  is the measurement vector and  $c$  is the measurement matrix that relates the states to the measurements.

The gain matrix,  $k$  may be computed in many different ways. Note that this, in contrast to a conventional noise filter, requires an estimate of the inputs. However, the filter returns the vector of all observable states which is needed for linear quadratic controllers.

This kind of estimator can be designed using the toolbox function `QCE`, for continuous systems, and `DQCE` for discrete systems. If the former is used, the filter must be converted to discrete time in a second step. If the latter is used you must be careful that the filter state matrix is balanced and well-conditioned.

An interesting example is given in `KalmanFilterDemo` in which two angle measurements are available for single degree of freedom case. One of the states is the unknown misalignment between the two sensors. Figure 27.1 shows the covariance for this problem.

**Figure 27.1:** Fixed gain Kalman filter



## 27.3 Variable Gain Estimators

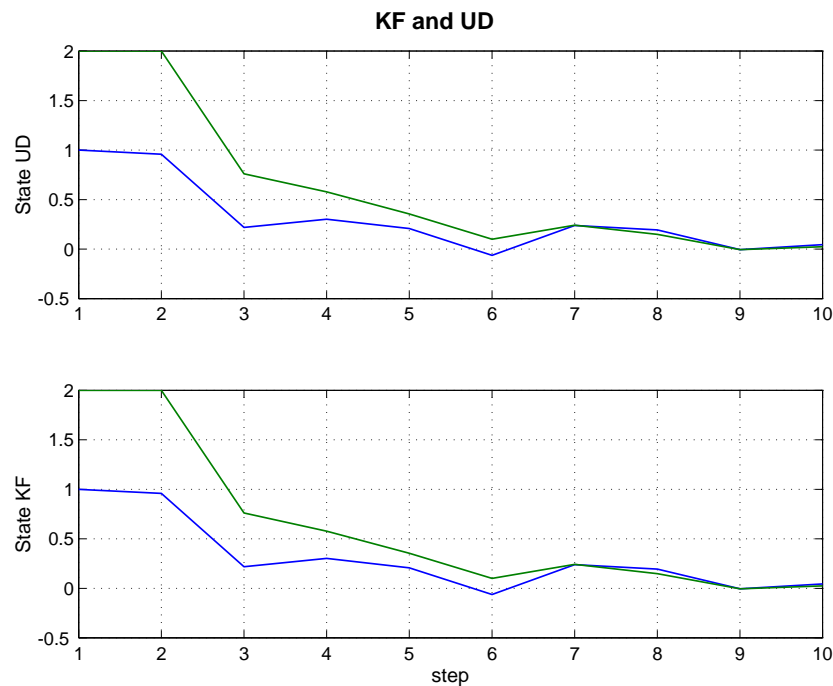
Two routines are available for variable gain estimators. These are `KFilter` for conventional Kalman filters and `UDKalmanFilter`, for filters in which the plant matrix is formulated in Upper Diagonal (UD) form. The latter has better numerical properties. Both filters are demonstrated for double integrator plant in the demo `UDKFDemo`. The results are compared in Figure 27.2 on the facing page.

As expected with double precision arithmetic both filters produce the same result. With single precision arithmetic the UD form should be better results. Note that `KFilter` eliminates the problem of instabilities due to asymmetric covariance matrices by forcing it to be symmetric each step.

## 27.4 Extended Kalman Filter

The extended Kalman Filter is used for attitude determination and many other applications. In an extended Kalman filter the state transition matrix and measurement matrix is linearized about the current estimated state. This is in contrast to the linearized Kalman Filter in which the measurement and state transition matrices are linearized about a predetermined state or state trajectory. In the context of attitude determination, attitude measurements are unit

Figure 27.2: UD filter



vector measurements. Since a unit vectors magnitude is 1 there are only 2 independent quantities. Thus a unit vector provides only a two-axis measurement. Since complete attitude knowledge requires at least 3-measurements a multiple unit vector measurements need to be used. Sources for measurements are

- Sun
- Planets
- Moon
- Magnetic field
- Stars
- Features on a planetary surface

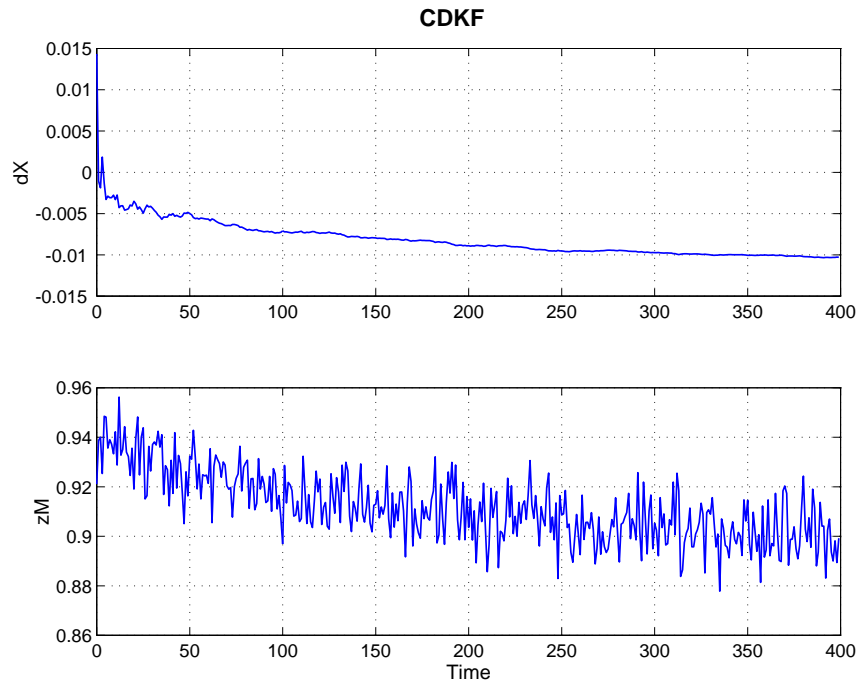
In each case a reference catalog for the object is maintained in the software. The first four provide unambiguous references. However, stars and planetary features need to be identified. The next chapter goes into stellar attitude determination in great detail. The abstracting of all attitude measurements into unit vectors in the toolbox makes it possible to have a unified approach to attitude determination. There are two approaches to the plant model. One is to run gyros all the time as model only the gyro noise. This approach is used on high precision pointing spacecraft. The other approach is to model the dynamics of the spacecraft. This approach requires a reasonable plant model. If the spacecraft is under closed loop control, a low order model can be extracted from the closed loop dynamics and used when control is active. Otherwise it is necessary to accurately model the spacecraft dynamics, including all control inputs. Two examples are given in this module, `SunMagAttDet` used in `MagSimWithEstimator`, and `SunEarthADSim`. The first combines sun and magnetic field measurements and the latter, sun and earth measurements. Another demo, `KFSunAndEarthTest2` compares different types of sensors.

## 27.5 Continuous Discrete Extended Kalman Filter

### 27.5.1 Introduction

The script `CDKFDemo` demonstrates the use of a Continuous Discrete Kalman Filter with a nonlinear spring model, and a nonlinear position measurement. The first plot in Figure 27.3 shows that the estimate converges to the true value of the spring position less a bias. The second shows the noisy measurement.

**Figure 27.3:** Continuous discrete Kalman filter



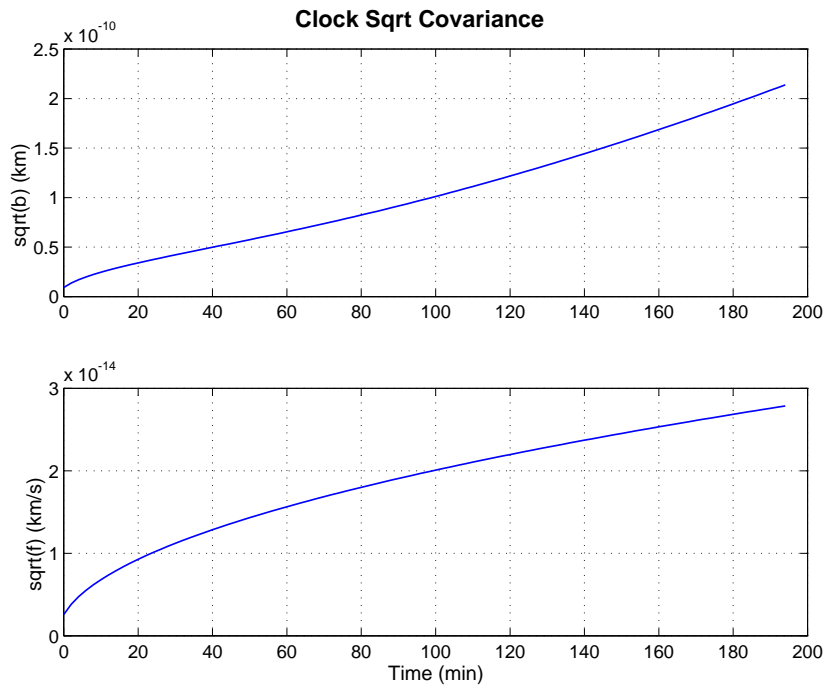
### 27.5.2 Orbit Determination

The orbit determination system used in this module assumes that all measurements used for orbit determination are range measurements. These can be from GPS satellites or ground stations. Both range and range rate are assumed to be available. One measurement gives enough information to estimate position and velocity along a single axis. At least three measurements are needed for a position fix. The filter incorporates each measurement as it arrives and propagates the orbit between measurements by numerically integrating the orbital equations of motion.

An orbit estimator is demonstrated in the script `OrbitKalmanFilter`. The clock error covariance is shown in Figure 27.4 on the facing page.

## 27.6 Unscented Kalman Filter

The Unscented Kalman Filter (UKF) is able to achieve greater estimation performance than the Extended Kalman Filter (EKF) through the use of the unscented transformation (UT). The UT allows the UKF to capture first and second order terms of the nonlinear system. Instead of just propagating the state, the filter propagates the state and additional sigma points which are the states plus the square roots of rows or columns of the covariance matrix. Thus the state

**Figure 27.4:** Clock error covariance for an orbit Kalman Filter

and the state plus a standard deviation are propagated. This captures the uncertainty in the state. It is not necessary to numerically integrate the covariance matrix.

Unscented Kalman filters are discussed in more detail in Chapter 29.



---

# STELLAR ATTITUDE DETERMINATION

---

## 28.1 Introduction

---

This chapter describes how to use the stellar attitude determination routines in the Estimation module to develop attitude determination systems for your applications. The process of stellar attitude determination is shown in [Figure 28.1 on the next page](#).

A simulation generates an ECI to body attitude quaternion and body rates. These are used for attitude determination. The body rates are only needed if the attitude determination system is gyro based. That means the model used by the filter is the model of the gyro, not of the spacecraft.

The simulated quaternion determines the pointing direction. If you want to test star centroiding you need a catalog of star images that cover the entire sky. For many purposes you can bypass that step and pass the measured unit vectors from the pinhole camera directly to the star identification software. At that point the pyramid algorithm, that uses four stars at a time, is used to determine which stars are in the field-of-view. This information is passed to the Kalman filter or least squares algorithm to determine the attitude.

## 28.2 Star Catalogs

---

The toolbox includes the catalogs given in [Table 28.1](#).

**Table 28.1:** Star catalog

Catalog	Number of Stars	Minimal Visual Magnitude	Description
Hipparcos	118218	14	Stars measured by the Hipparcos
FK5	1536	8.35	Standard star catalog
FK5 Polar	14	7.17	Polar stars from FK5
FK5 Short	768	7.75	Subset of FK5

You load a catalog by typing:

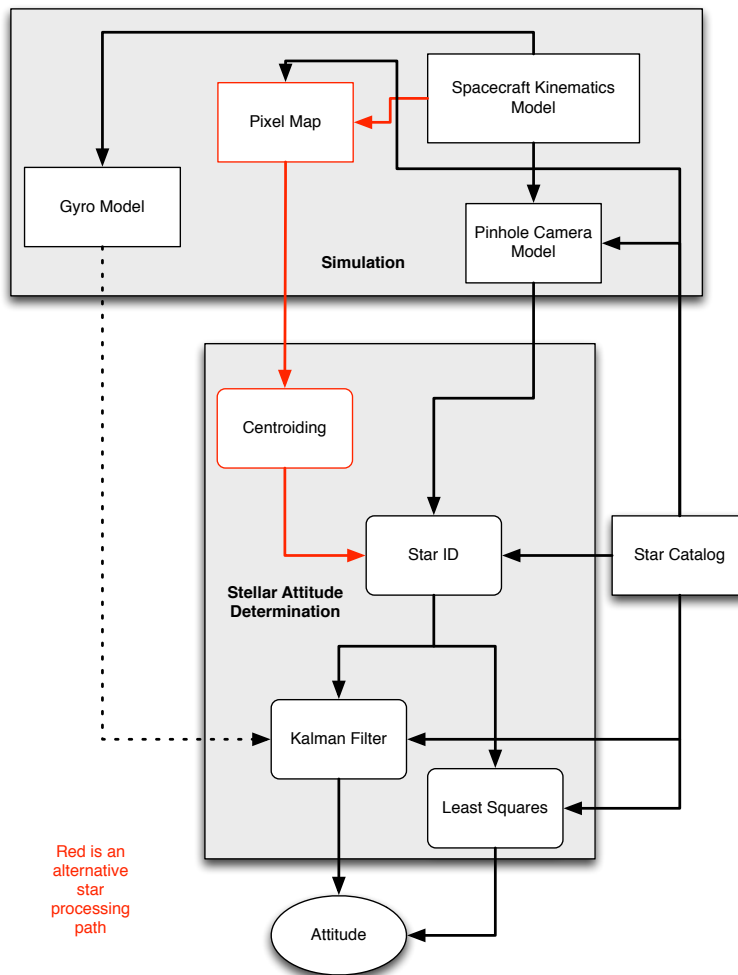
**Listing: Catalog**

```

1 starCatalog = LoadCatalog( 'FK5_Short', 'all' )
2
3 starCatalog =
4 name1: [768x8 char ]
5 name2: [768x8 char ]

```

**Figure 28.1:** Stellar attitude determination process.





```

6 name3: [768x8 char ]
7 rA: [ 1x768 double]
8 dec: [ 1x768 double]
9 pMRA: [ 1x768 double]
10 pMDec: [ 1x768 double]
11 parallax: [ 1x768 double]
12 rV: [ 1x768 double]
13 vM: [ 1x768 double]

```

'all' may be replaced by a maximum visual magnitude which is determined by the sensitivity of the star camera that you plan to use in your application. If you type

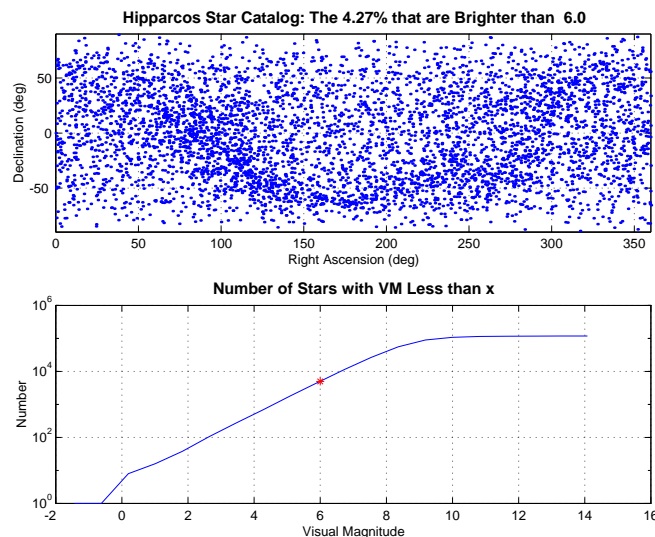
```
LoadCatalog
```

you will get the default Hipparcos catalog with stars brighter than visual magnitude 6. With no outputs, Figure 28.2 is generated.

**Table 28.2:** Star catalogs

Catalog	Number of Stars	Minimum visual magnitude	Description
Hipparcos	118218	14	Stars measured by the Hipparcos satellite.
FK5	1536	8.35	Standard star catalog.
FK5 Polar	14	7.17	Polar stars from FK5.
FK5 Short	768	7.7500	Subset of FK5.

**Figure 28.2:** LoadCatalog default



The red mark shows the selected visual magnitude. The upper picture shows the star map. After generating a catalog you need to generate a dot product catalog. The dot products are used in the star identification procedures. This is done using `StarDataGeneration`,

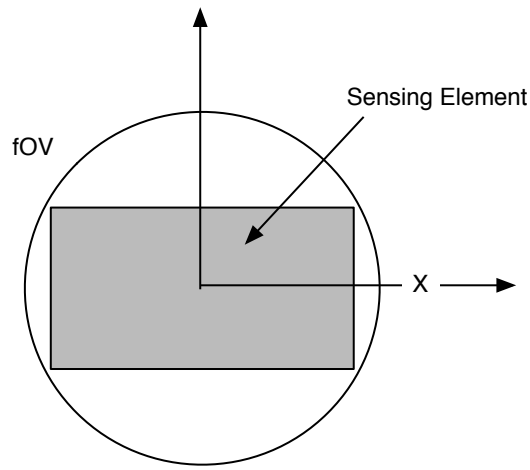
```
d = StarDataGeneration( starCatalog, fieldOfView, angularResolution )
```

where `starCatalog` is the data structure generated above, `fieldOfView` is the field-of-view in radians and `angularResolution` is the minimum resolvable angle.

The data structure `d` has several fields. They are given in the following table. `nearMatrix` is a 3-by-m matrix with dot products for all stars and their neighbors within the `fieldOfView` of the camera. That is, if a star is in the center of the field-of-view, it gives dot products for all stars that it can see. `fieldOfView` should be chosen as in illustration

in Figure 28.3. The `fieldOfView` should circumscribe the sensing element. `StarDataGeneration` does not

**Figure 28.3:** FOV determination



store duplicate pairs. Once you have run `StarDataGeneration` you can save the results in a mat file and use that. Unless you change the star catalog, `dPTol` or `fOV` there is no reason to rerun the function.

**Table 28.3:** `StarDataGeneration` output fields

Field	Size	Units or Type	Description
<code>starMatrix</code>	(4,:)	[intensity;rad;rad;rad]	An array with intensity and unit vectors for all stars in star catalog.
<code>nearMatrix</code>	(3,m)	[dot product;star 1;star 2]	The first row is the dot product and the next two rows are the two stars corresponding to the dot product pair.
<code>starCatalog</code>	(1,1)	Data structure	The reduced star catalog.
<code>angularResolution</code>	(1,1)	rad	The input angular resolution.
<code>fieldOfView</code>	(1,1)	rad	The input field of view fOV Sensing element .

## 28.3 Generating Star Data

There are two functions that you can use to generate star data. One, `PixelMapCentroid`, generates star data from input images. The second, `PinholeStarCamera`, generates star data from a star catalog. The latter is useful for simulation. Both output the same data structure shown in Table 28.4.

**Table 28.4:** `StarMeasurement` data structure

Name	Size	Units or Type	Description
<code>pixelLocation</code>	(2,m)	pixels	The star centroid in pixels
<code>intensity</code>	(2,m)	intensity	The intensity of the star

### 28.3.1 `PixelMapCentroid`

`PixelMapCentroid` is invoked with the call

```
starMeasurement = PixelMapCentroid( pixelMap, threshold, show )
```

pixelMap may be a pixel map, a string with the name of an image file, or an empty matrix in which case the function brings up a file selection dialog. threshold is from 0 to 1 and tells the function to only include stars whose intensity is that fraction of the maximum intensity measured. Any argument for show will generate a set of plots.

You can generate a pixel map by typing

```
pixelMap = double(imread( fileName ) );
```

where fileName is the name of an image file. See imread for more information. Doing this outside of PixelMapCentroid can save time if you want to experiment with different thresholds.

### 28.3.2 PinholeStarCamera

PinholeStarCamera is invoked with the call

```
[starMeasurement, camera] = PinholeStarCamera( bCatalogToCamera, camera,
starCatalog, drawImage );
```

starCatalog is the star catalog data structure generated by LoadCatalog. camera is the camera data structure. The fields in the camera data structure are, wX, the pixel width in  $x$ , wY the pixel width in  $y$  and fScale which can be found from

```
f = ComputePixelMapScale( wX, xFieldOfView )
```

Any value for drawImage cause an image to be drawn.

## 28.4 Centroiding

The first step in star identification is to take the pixel map from the CCD camera and to identify stars and compute centroids. This can be done with the function

```
PixelMapCentroid
```

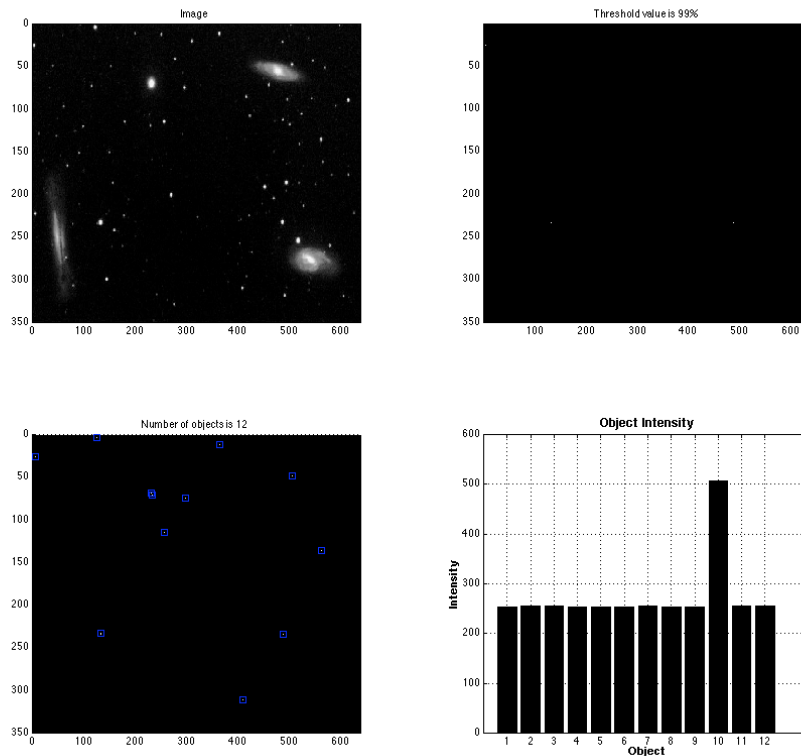
If you just type in the above name in a MATLAB command window and then select the file “StarFieldDemo.jpg” you will get the display shown in [Figure 28.4 on the following page](#).

12 objects are identified. It is important to set the minimum tolerance appropriately. Every pixel will have some finite value due to dark current and noise. If the minimum tolerance is set too low these pixels will be identified as stars. If this is combined with relatively low resolutions, it may result in misidentified stars.

Another approach is StarCentroidCOM. This finds the centroid given intensities on  $n$  pixels and their coordinates using the center of mass method. The center of mass will have larger errors if the width of the region of interest goes outside the pixel mask. Generally, star measurements on the edge will not be reliable due to optical errors so should be discarded if you have enough stars that for than  $n$  pixels from the edges. [Figure 28.5 on page 219](#) the result from the built-in demo.

```
>> StarCentroidCOM
Measured Centroids (microns)
    70.0000    90.0000  12665.3031
   167.8630   386.0009  14374.9016
   250.2159   230.0000  19686.0882
   359.9821  1240.1813  37047.1646
   383.9552   321.9790  19828.4600
   530.2069   881.9847  29506.0417
   879.9857  1041.9738  37011.9382
```

Figure 28.4: Centroiding demo



```

1160.1834    439.9843    17161.5555

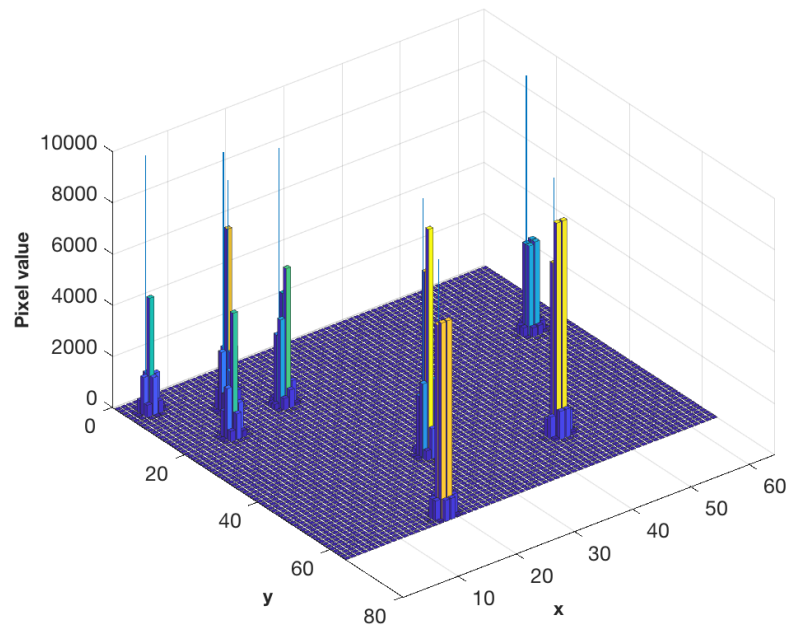
True Centroids (The order may be different)
  70.0000    90.0000    12666.9016    14.0000
 168.0000    386.0000    14476.4591    16.0000
 250.0000    230.0000    19905.1312    22.0000
 360.0000    1240.2000    37079.0291    41.0000
 384.0000    322.0000    19905.1310    22.0000
 530.0000    882.0000    29857.6965    33.0000
 880.0000    1042.0000    37095.9263    41.0000
1160.2000    440.0000    17190.7948    19.0000

```

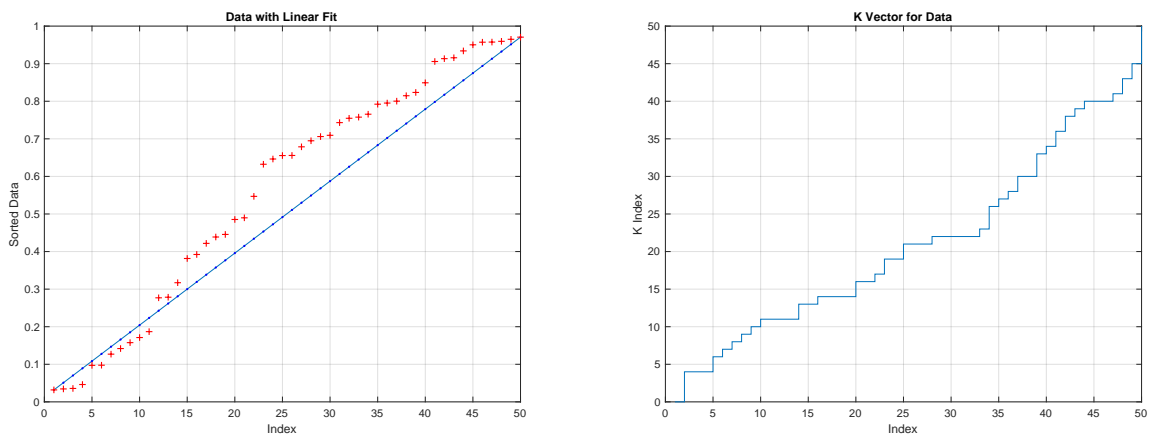
## 28.5 Star Identification

The Spacecraft Control Toolbox employs the Pyramid algorithm for star identification. The star ID algorithm begins with proper processing of the star catalog. We have implemented the  $k$ -vector search less technique for looking up star pairs based on measured angles. This is combined with a pyramid star algorithm that find star triads and then matches them with an additional fourth star, nearly eliminating incorrect star identification.

The  $k$ -vector range search technique allows a range of values from the near matrix to be extracted without searching, by using indexing. The star angles of the star pairs in the near matrix are sorted in ascending order, producing an array of star angles  $s$ , and an additional integer index value is assigned to each pair. A slope and intercept is calculated for an equivalent straight line through the data. The indices of  $s$  can then be calculated for a given star angle range and all values within the range easily retrieved. Practically, the index  $k$  gives the number of elements in the angle array below the value dictated by the equivalent line.

**Figure 28.5:** StarCentroidCOM demo

The script `CreateKVector` produces a demo in MATLAB with the plots in Figure 28.6. This is for a simple set of random numbers on the interval (0,1). The plot on the left shows the steps in  $k$ , in a stair plot, as compared against a linear progression. For example, at index 4,  $k$  jumps from 2 to 4, indicating that there are 4 data points with values below the line value at that index. On the right, the data is printed against the straight line fit to the endpoints, with a range calculated of 0.5 to 0.75. The range is marked with the magenta solid lines. The data points that are circled are the values returned as in the desired range. We can see in this particular example that one data point below 0.5 was returned, because it is also above the value of the closest line point, 0.4642. The line points bracketing the range are indicated by the black dashed lines.

**Figure 28.6:** K-Vector Example in MATLAB

When using  $k$ -vector with measured star pairs, the angle range will be the measured angle  $\theta$  plus and minus some  $\delta$ . This should be a high confidence interval such as  $4\sigma$ . This could be a constant value for the entire pixel array or calculated based on the star locations or angle as is explained in the next section.

The pyramid algorithm<sup>1</sup> begins with star triads but adds a fourth star to the pattern for more precise identification. It has been tested on orbit on the HETE spacecraft.

The demo `StarIDPyramidDemo`.

```

%% Parameters
% Constant
degToRad      = pi/180;

% Camera parameters
fOV           = 30*degToRad;
fScale        = 1;
nPixels       = 1024;

qBToS         = [1;0;0;0];
uS            = [0 0 1]';

%% Create a random star catalog
nStars        = 1000;
angRes        = 5*fOV/nPixels;
[,starCatalog] = RandSC(nStars);
d              = StarDataGeneration( starCatalog, fOV*sqrt(2), angRes );

%% Identify stars form random initial conditions
disp('Demo_of_StarIDPyramid_--')
options       = StarIDPyramid;
options.pixelMapScale = fScale;
for k = 1:100
    q          = QRand;
    s          = StarSensor( q, qBToS, uS, d.starMatrix, fOV, fScale )
    ;
    starMeas.pixelIntensity = s(1,:);
    starMeas.pixelLocation = s(2:3,:);
    starID          = StarIDPyramid( starMeas, d, options );
    fprintf('Quaternion: [%f_%f_%f_%f]\n',q)
    fprintf('Star_IDs: %s\n',num2str(starID));
end

```

shows how to use the pyramid algorithm. Only the first few quaternions are shown below.

```

>> StarIDPyramidDemo

StarDataGeneration: Eliminating stars that cannot be separated based on the input
                    angular resolution.

StarDataGeneration:      2 stars will be eliminated that are too close to other
                    stars.
                    998 stars will be in the reduced catalog.

Demo of StarIDPyramid --
Quaternion: [0.578263 -0.410593 0.341393 -0.616828]
Star IDs:   36 156 208 249 256 322 350 363 368 389 397 482
Quaternion: [0.708439 0.147505 -0.272617 -0.634064]
Star IDs:   9 22 176 210 238 353 391 413 427 432 465 518
Quaternion: [0.549248 -0.491669 -0.015056 -0.675545]
Star IDs:   4 25 29 93 146 162 165 193 237 271 340 343

```

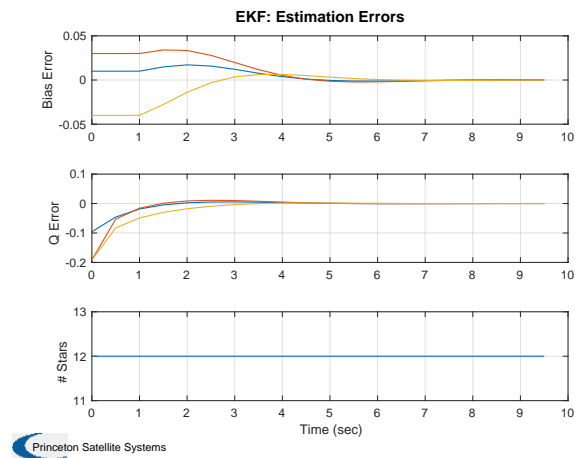
The pre-processing of the catalog removes closely spaced stars.

<sup>1</sup>Mortari, Samaan, Bruccoleri, and Junkins. *The Pyramid Star Identification Technique*, Navigation, Vol. 51, No. 3, Sept. 2004

## 28.6 Kalman Filtering

The final step is to integrate these functions with filtering. Both gyro based and gyro-less algorithms are available. You can use the Extended Kalman Filter, Least Squares or the Unscented Kalman Filter. UKF is useful with nonlinear sensor models. `AttDetEKFSim` is a simulation using gyros and the EKF. `StellarAttDetEKF` implements stellar attitude determination with the EKF. Figure 28.7 shows the convergence of the attitude errors.

**Figure 28.7:** Use of the EKF for stellar attitude determination.







---

# UNSCENTED KALMAN FILTER

---

## 29.1 Introduction

---

The unscented Kalman Filter is an alternate approach to estimating states and parameters in a nonlinear system. In fact, it is not necessary to distinguish between states and parameters at all. A major advantage, albeit at the cost of additional computation, is that it can employ any nonlinear measurement model or plant model.

This chapter gives an example of an unscented Kalman Filter.

## 29.2 Pressure Kalman Filter

---

This demonstration in `PressureKF` for a fuel mass Kalman filter for a blowdown system. In a blowdown system, the fuel is pressurized by a gas, such as helium. As fuel is consumed the gas expands into the empty volume and the pressure decreases. The estimator uses a pulsewidth model and a pressure measurement. The pressure measurements on a spacecraft are generally low resolution, typically 8 bits over the entire pressure range (of 350 psi to 100 psi.) Every time a thruster fires, the model computes the fuel used based on a model of fuel consumption versus pulsewidth. This is the dynamical model. The measurement of pressure is then incorporated into the estimate using an Unscented Kalman Filter (UKF) implemented in `UKF`. The UKF can use a nonlinear measurement and plant (dynamics) model directly without any linearization.

The script computes a random pulsewidth from 0 to 8 seconds in length when a second random number between 0 and 1, is greater than 0.95. The pressure measurement has 12 bits resolution and random noise.

The script is fairly short so we show it all here.

**Listing 29.1:** Fuel mass Kalman filter example

*PressureKF.m*

```

1 %-----
2 %   Demonstrate a fuel mass Kalman filter for a blowdown system.
3 %   In a blowdown system, the fuel is pressurized by a gas, such as
4 %   helium. As fuel is consumed the gas expands into the empty volume
5 %   and the pressure decreases. The estimator
6 %   uses a pulsewidth model and a pressure measurement. The pressure
7 %   measurements on a spacecraft are generally low resolution, typically
8 %   8 bits over the entire pressure range (of 350 psi to 100 psi.)
9 %   Every time a thruster fires, the model computes the fuel used
10 %   based on a model of fuel consumption versus pulsewidth. This is
11 %   the dynamical model. The measurement of pressure is then
12 %   incorporated into the estimate using an Unscented Kalman Filter

```

```

13 % (UKF). The UKF can use a nonlinear measurement and plant (dynamics)
14 % model directly without any linearization.
15 %
16 % The script computes a random pulsewidth from 0 to 8 seconds in
17 % length when a second random number between 0 and 1, is greater than
18 % 0.95.
19 %
20 % The pressure measurement has 12 bits resolution and random noise.
21 %
22 % The script plots true pressure, fuel mass estimated errors, pulsewidth
23 % and the pressure measurement.
24 %-----
25 % Copyright (c) 2006 Princeton Satellite Systems, Inc.
26 % All rights reserved.
27 %-----
28
29 uE      = 200*9.806;
30 thrust  = 1;
31 dT      = 8;
32 mDot    = thrust/uE;
33
34 T       = 300;
35 vTank   = 0.5;
36 rhoFuel = 1000;
37 mPress  = 0.5;
38 rPress  = MolWt2R( 0.004 ); % He
39
40 nSim    = 1000;
41
42 p       = 350*6895;
43 mFuel   = BloDownMass( mPress, rhoFuel, vTank, rPress, T, p );
44 xPlot   = zeros(6, nSim);
45 LSB     = p/2^12;
46 sigQ    = sqrt(NQuant( LSB, 'truncate' ));
47 sigN    = LSB/2;
48
49 % Estimation parameters
50 %-----
51 d.x      = 0.9997*mFuel;
52 d.rHSFun = 'RHSPressure';
53 d.measFun = 'GPressure';
54 d.measFunData = struct('vTank', vTank, 'rhoFuel', rhoFuel, 'mPress', mPress, 'rPress', rPress, 'T', T)
55 ;
56 d.alpha  = 0.5;
57 d.kappa  = 0;
58 d.beta   = 2;
59 d.dY     = 1;
60 d.dT     = dT;
61 d.rM     = sigQ^2 + sigN^2;
62 d.rP     = 0.00001;
63 d.p      = 40*d.rP;
64 d        = UKF('initialize', d);
65
66 for k = 1:nSim
67     if( rand > 0.95 )
68         pw = rand*8;
69     else
70         pw = 0;
71     end
72     p      = BloDown( mPress, rhoFuel, vTank, rPress, T, mFuel );
73     pMeas  = LSB*floor(p/LSB) + sigN*randn;
74     xPlot(:,k) = [p;mFuel;pw;pMeas;d.x;d.p];
75     mFuel   = mFuel - mDot*pw;
76     d.t     = 0;
77     d.rHSFunData = struct('mDot', mDot, 'pw', pw, 'dT', dT);
78     d      = UKF( 'update', pMeas, d );
79 end

```

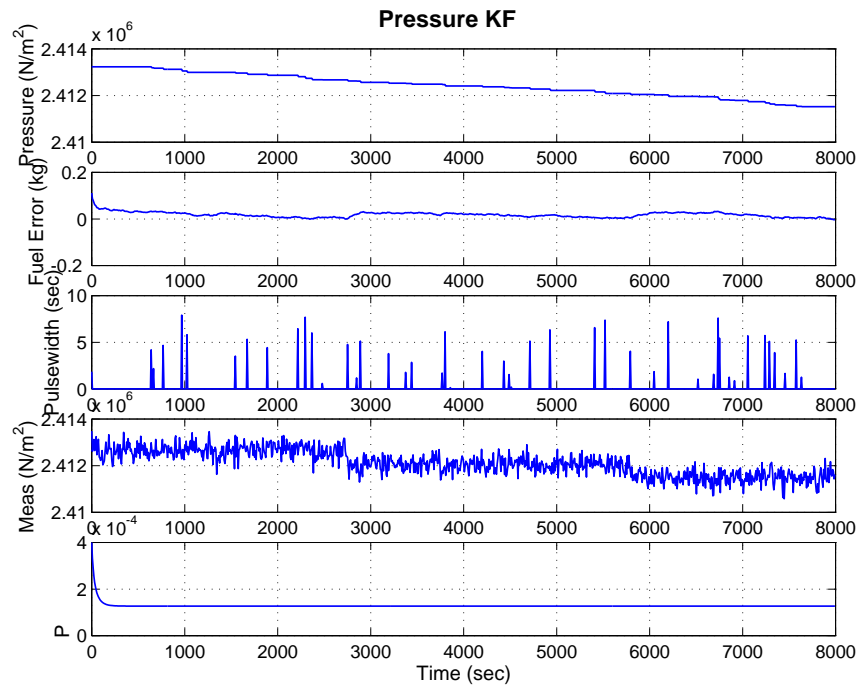
```

80 yL = {'Pressure_(N/m^2)' 'Fuel_Error_(kg)' 'Pulsewidth_(sec)' 'Meas_(N/m^2)' , 'P'};
81 Plot2D( (0:(nSim-1))*dT, [xPlot(1,:);xPlot(2,:) - xPlot(5,:);xPlot([3 4 6],:)] , 'Time_(sec)',
    yL, 'Pressure_KF')

```

PressureKF.m

Figure 29.1: Pressure Kalman Filter



The top plot in Figure 29.1 shows the pressure. The second shows the fuel mass estimation error. The next plot shows the pulsewidth of thruster firings which are consuming the fuel. The noisy pressure estimate is shown below that. The covariance, shown in the last plot, decreases rapidly and then is constant. This is confirmed by the mass error which reaches steady state quickly.



# APPENDICES



## GOES INTERFACE

This chapter presents the GOES interface functions.

### A.1 Introduction

The GOES functions provides a convenient and easy-to-use interface to the GOES CD-ROM. The Toolbox can read GOES binary files and either display the information in plots, or load it into matrices for further manipulation. The Toolbox formats the data and removes telemetry glitches automatically. The CD-ROM includes data from GOES-2 through GOES-7 starting in January 1986. The data is available as one and five minute averages. The five minute averages take up between 288K and 320K while the one minute averages range from 1,440K to 1,600K. Manipulating the five minute averages requires about 12,000K allocated to MATLAB and manipulating the one minute averages requires in excess of 20,000K.

### A.2 Loading GOES Data

To load GOES data type the command

```
>> LoadGOES
```

You will then see a dialog box resembling that shown in [Figure A.1 on page 117](#).

All files with the `.bin` suffix will be displayed and/or highlighted. Double click on the only file name that appears. The file names have the format `DSSRYMM.BIN;1` where

**Table A-1:** GOES data definition

Digits	Description
D	Data Version: G, X-ray, Magnetic Field, Electrons & Uncorrected Proton Channels Z, X-ray, Magnetic Field, Electrons & Corrected Proton Channels I, X-ray, Magnetic Field, Electrons & Corrected Integral Proton Channels H, X-ray, Magnetic Field, Electrons & HEPAD A, X-ray, Magnetic Field, Electrons & Uncorrected Alpha particles
SS	Satellite ID (02 through 07)
R	Time Resolution (5 = 5-minute averages, 1 = 1-minute averages)
YY	Year

MM	Month
----	-------

Double click on the filename in the window or hit the OPEN button. All of the GOES data will be displayed in a series of three plots. The first shows magnetometer data, the second X-ray and electron data and the third will show proton and alpha particle data.

## A.3 Getting GOES Longitude

---

Type the command

```
>> LoadSatP
```

A dialog box will appear listing all files with the suffix .TXT. A plot of the longitude for all GOES satellites from 1986 through the present will be displayed.

## A.4 GOES functions

---

### A.4.1 LoadMag

LoadMag loads multiple magnetic field files into one array. For example put the GOES CD in your CD drive and type:

```
b = LoadMag(7, 'GOES_94', 11, 88, 5, 89);  
Plot2D([], b, Sample, Magnetic Field)
```

You will get the plot in [Figure A.2 on page 117](#).

### A.4.2 LoadSers

LoadSers loads GOES data. For example, with the CD in the drive, type

```
[b,e,x,p,file] = LoadSers(7, 'A', 'GOES_94', 11, 88, 11, 88);  
Plot2D([], e, Sample, Electrons)
```

[Figure A.3 on page 118](#) shows the resulting plot.

### A.4.3 Weather

Weather gives a summary plot of all GOES data. For example, type

```
>> Weather
```

and select the file in the GOESData folder. You will then see the plot in [Figure A.4 on page 118](#).

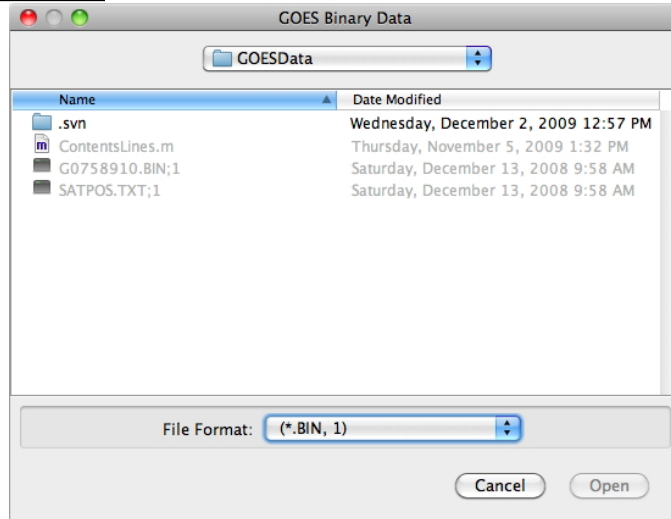
## A.5 For More Information

---

The GOES CD-ROM is a product of the National Oceanic and Atmospheric Administration National Geophysical Data Center Solar-Terrestrial Physics Division. The CD-ROM includes data from January 1986-present for all of the



**Figure A-1:** Loading GOES data



**Figure A-2:** GOES Magnetic Field Data

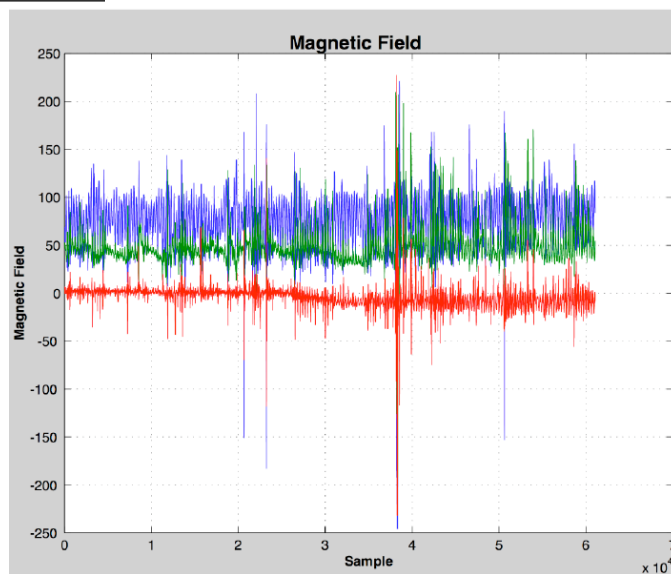


Figure A-3: GOES Electrons

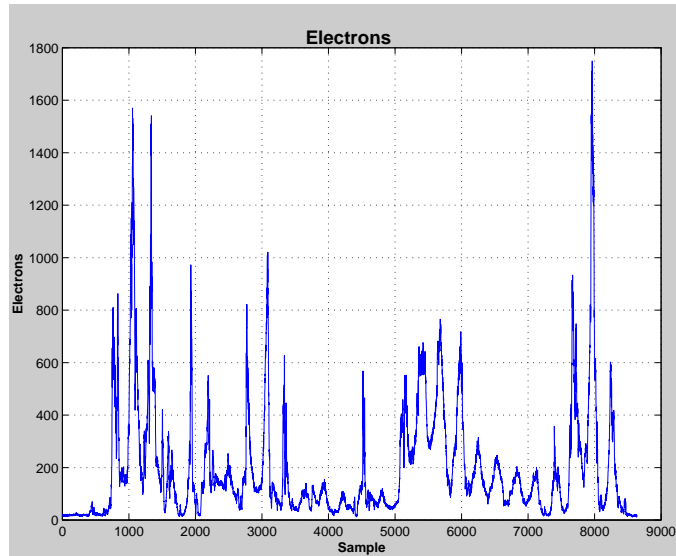
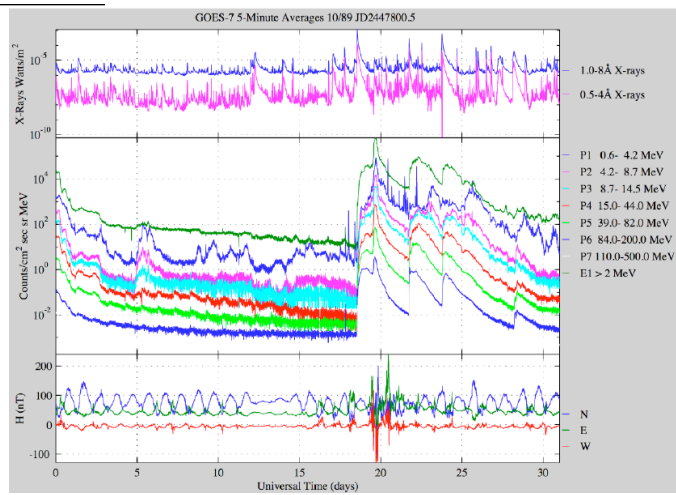


Figure A-4: GOES Weather



GOES satellites. The CD-ROM disk is updated periodically. The most up-to-date information is also available on 3.5” floppy disks.

## A.6 References

---

- Wilkinson, D. and G. Ushomirskiy. (1994).
- GOES Space Environment Monitor CD-ROM 1-Minute and 5- Minute Averages January 1986-April 1994
- User Documentation. National Oceanic and Atmospheric Administration National Geophysical Data Center Solar-Terrestrial Physics Division, July 18, 1994.



# USING DATABASES

This chapter shows you how to use the database and constant functions in the toolboxes. These functions allow you to manage the various constants and parameters used in your projects and ensure that all of your engineers are using consistent numbers in their analyses.

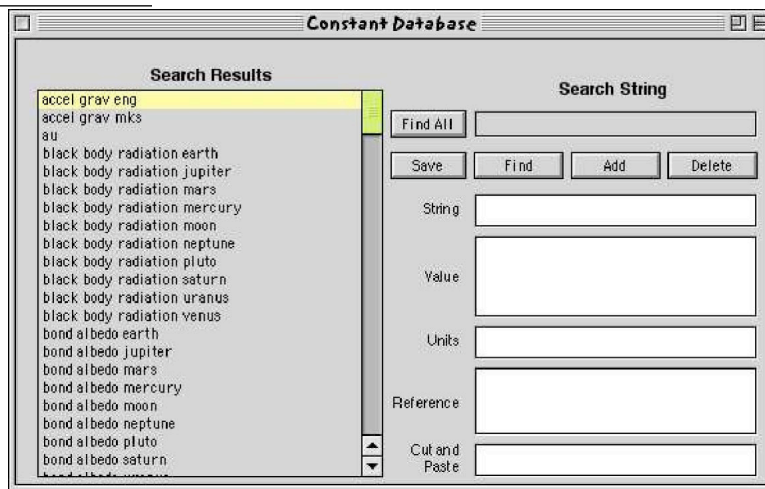
## B.1 The Constant Database

The constant database gives a substantial selection of useful constants. If you type

Constant

you will get the GUI in Figure B.1 on page 121.

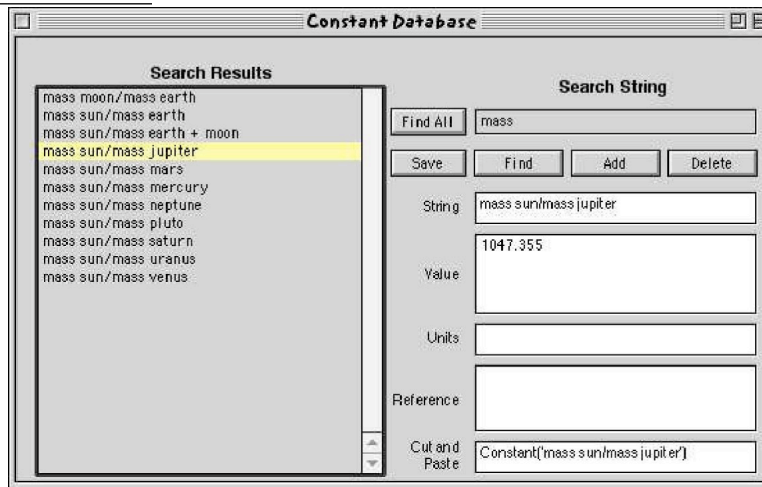
Figure B-1: Constant database



The list on the left is a list of all of the constants in the database. You can enter a search string and look for matches by hitting Find. If you then click one of the selections the GUI looks like it does in the following figure. This function always loads its constants from the .mat file SCTConstants.mat

The string field shows the parameter name. Directly below it is the value for the parameter. The value may be any MATLAB construct. Directly below that is a field for units, then a field for reference information and finally a field

Figure B-2: Searching for mass



that gives a template for the function. You can cut and paste this into any function or script. Searches are case and whitespace insensitive. To add a new constant, type a constant name in the String field, a value in the value field and optionally, units and reference information. Hit Add. You will get a warning if you try to replace an existing constant. To modify the value of an existing constant, select the constant you wish to modify. Edit the value and hit the Add button. You can delete a constant by hitting the Delete button. You can access the database through the command line by passing the name of the desired constant to the function. For example:

```
1 Constant('mass_sun/mass_jupiter')
2 ans =
3     1.0474e+03
```

The database loads its constants from a database the first time it is launched. Once it is launched, it will not load a new database. However there is a fair amount of overhead involved in searching for a constant so we recommend that whenever possible you get the constant once from the database and store it in a local variable.

## B.2 Merging Constant Databases

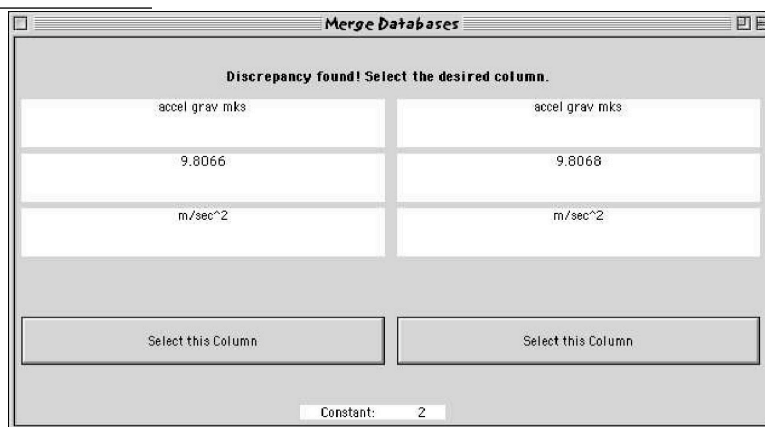
Periodically, PSS will release new constant databases. If you have customized your own database you can merge it with the PSS database using the `MergeConstantDB` function. Just type

```
MergeConstantDB( initialize, a, b )
```

where `a` and `b` are the `.mat` files to be merged. The standard PSS database is called `SCTConstants.mat`. In this example we have modified the value of `'accel grav mks'` to be 9.8068. We type

```
MergeConstantDB('initialize', 'SCTConstants.mat', 'SCTConstantsOld.mat')
```

You will see the display in Figure B.3 on page 123. Just click the button for the column you wish to include in `SCTConstants.mat`.

**Figure B-3:** MergeConstantDB function





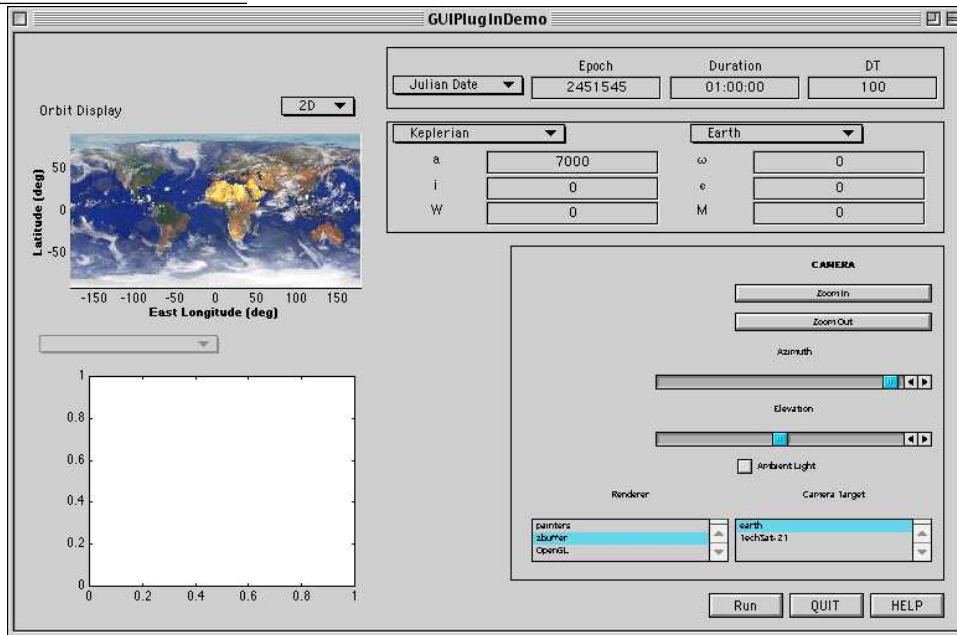
## GUI PLUG INS

This section shows you how to use GUI plug-ins. Such plug-ins are used in a number of displays created by Princeton Satellite Systems and you can use them to build your own custom displays for performing simulations and analysis. These functions date from MATAB version 5.2 but are still supported in version 7. See `Common/Plugins` and `SC/GUIPlugIn` for available plug-ins.

### C.1 GUI PlugIn Demo

`PlugInDemo` is a m-file function that implements an orbit simulation using several GUI PlugIn elements. This can be used as a template for your simulations.

**Figure C-1:** PlugInDemo on starting



The GUI PlugIns shown are from top to bottom starting on the left:

- `OrbitDisplayPlugin`

- PlotPlugIn
- TimePlugIn
- ElementsPlugin
- DrawSCPlugin

The Run, QUIT and HELP buttons are part of the `PlugInDemo` function.

You can change any of the displayed properties in the three frames on the right. Whenever you change the properties, the changes are passed to the rest of the GUI. For example, if you select the earth as the camera center and zoom out you get the display in Figure C.2 on page 126.

**Figure C-2:** PlugInDemo with the earth as center



You can change planets as shown in Figure C.3 on page 127.

If you hit Run, the simulation will run. The results at the end are shown in Figure C.4 on page 127.

The 3D window (which is animated) looks like Figure C.5 on page 128.

## C.2 Writing Your Own GUI Function

`PlugInDemo` is implemented as shown below.

**Listing C.1:** PlugInDemo top-level switch statement

*PlugInDemo.m*

```

1 function PlugInDemo( action )
2 % Process the input arguments
3 %-----
4 if( nargin < 1 )
5 action = 'create_gui';
6 end
7 % Perform actions
8 %-----
9 switch action

```

Figure C-3: Selecting planets in the elements plugin

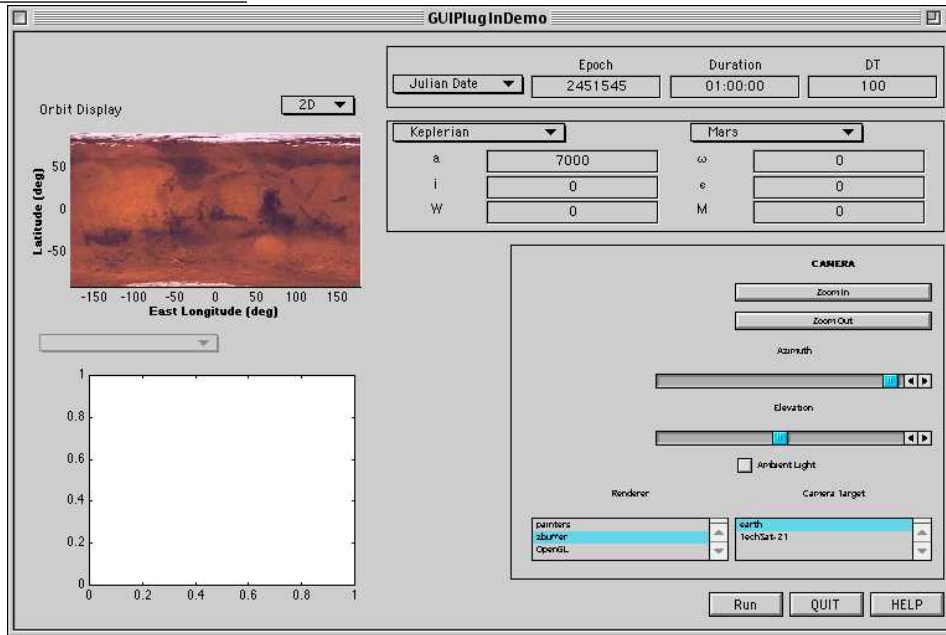
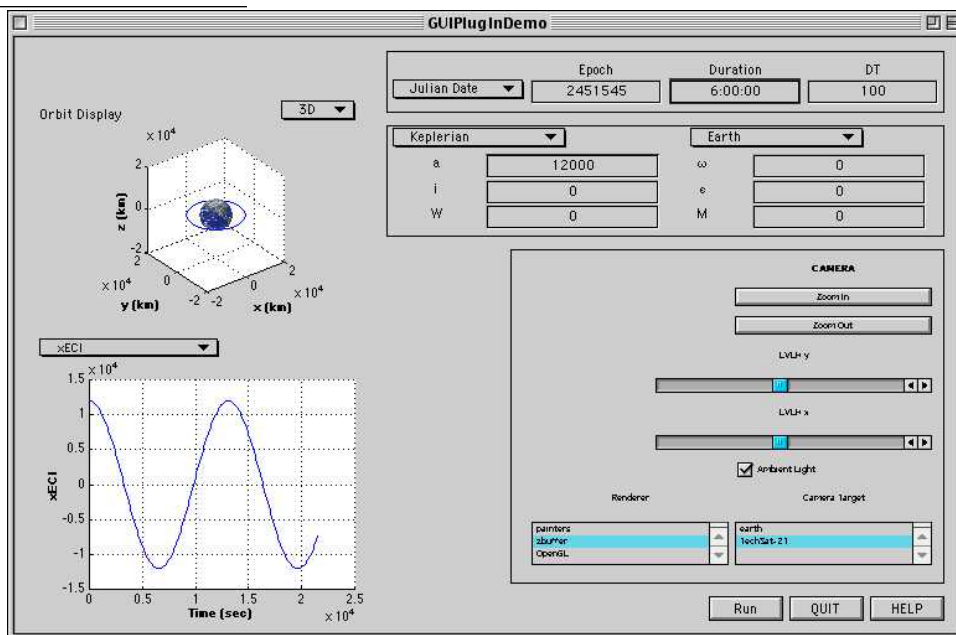
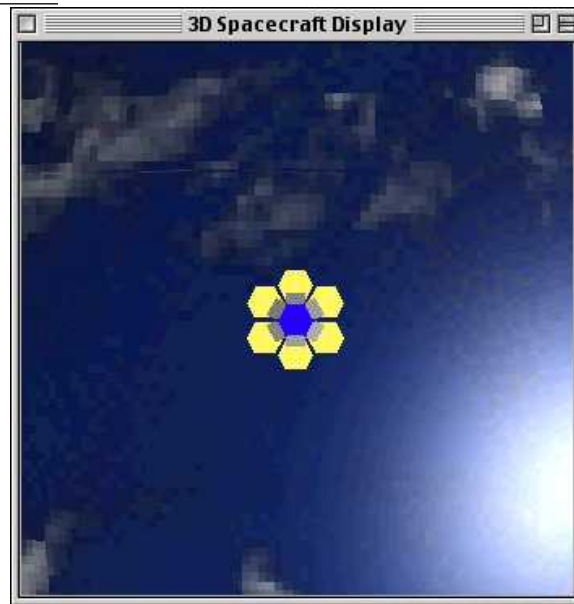


Figure C-4: PlugInDemo at the end of the simulation



**Figure C-5:** PlugInDemo 3D Spacecraft Display

```

10 case 'help'
11 HelpSystem( 'initialize', 'SCHelp' );
12 case 'create_gui'
13 h = GetH;
14 if( isempty(h) )
15 CreateGUI;
16 else
17 figure( h.fig );
18 end
19 case 'run'
20 Run;
21 case 'changed'
22 Update;
23 case 'quit'
24 h = GetH;
25 CloseFigure( h.fig );
26 end

```

*PlugInDemo.m*

This first part uses a switch statement to get the right action when PlugInDemo is called. All of the actions come about when you click buttons on the window. You run PlugInDemo by typing

```
PlugInDemo
```

If a PlugInDemo window already exists it will bring it to the front.

The CreateGUI subfunction draws the window and initializes all the plugins.

**Listing C.2:** CreateGUI*PlugInDemo.m*

```

1 function CreateGUI
2
3 % The figure window
4 %-----
5 p = [5 5 760 480];
6 h.fig = figure( 'name','GUIPlugInDemo','Units','pixels', 'Position',[40 p(4) - 600
7 p(3:4)],'resize','off', 'NumberTitle', 'off','tag', 'GUI_PlugIn_Demo',
8 'CloseRequestFcn', CreateCallback( 'quit' ) );
9

```

```

10 % Buttons
11 %-----
12 v = {'parent', h.fig, 'units', 'pixels', 'fontunits', 'pixels'};
13 r = p(1) + p(3);
14 h.run = uicontrol( v{:}, 'Position', [r-205 10 60 20], 'callback',
15 CreateCallback( 'run' ), 'string','Run' );
16 h.quit = uicontrol( v{:}, 'Position', [r-140 10 60 20], 'callback',
17 CreateCallback( 'quit' ), 'string','QUIT' );
18 h.help = uicontrol( v{:}, 'Position', [r- 75 10 60 20], 'callback',
19 CreateCallback( 'help' ), 'string','HELP' );
20
21 % Initialize the plugins
22 %-----
23 cB = 'PlugInDemo(␣'changed'␣)';
24 h.orbitDisplayTag = OrbitDisplayPlugIn( 'initialize', [], h.fig, [ 5 250 290
25 185], [ ] );
26 h.plotPlugInTag = PlotPlugIn( 'initialize', [], h.fig, [ 5 10 290 235] );
27 h.timePlugInTag = TimePlugIn( 'initialize', [], h.fig, [300 420 450 50], cB );
28 h.elementsPlugInTag = ElementsPlugIn( 'initialize', [], h.fig, [300 320 450 90],
29 cB );
30
31 % Initialize the 3D window
32 %-----
33 sim = GetSimData( h );
34 h.g = load('TechSat-21');
35 if( isfield( sim.orbit, 'r' ) & isfield( sim.orbit, 'v' ) )
36     h.g.body(1).bHinge.q = QLVLH(sim.orbit.r, sim.orbit.v );
37 else
38     h.g.body(1).bHinge.q = [1;0;0;0];
39 end
40
41 h.g.name = 'TechSat-21';
42 if( isfield( sim.orbit, 'r' ) )
43     h.g.rECI = sim.orbit.r;
44 else
45     h.g.rECI = [sim.orbit.el(1);0;0];
46 end
47 h.g.qLVLH = h.g.body(1).bHinge.q;
48 h.scWindowTag = DrawSCPlugIn( 'initialize', h.g, h.fig, [400 40 350 270], 'earth',
49 sim.time.jDEpoch );
50
51 PutH( h );

```

---

*PlugInDemo.m*

figure and uicontrol are MATLAB functions. figure creates a new figure window and uicontrol creates a new user interface control. In this case the uicontrols are the Run, HELP and QUIT buttons. Arguments are passed to figure and uicontrol in pairs. The first argument of each pair describes the next argument. For example,

'position'

tells MATLAB that

[r-205 10 60 20]

is

[left bottom width height]

in MATLAB screen coordinates. You can store parameter pairs in a cell array

v = 'parent', h.fig, 'units', 'pixels', 'fontunits', 'pixels'

and pass them to a uicontrol as

```
uicontrol( v{:}, ...
```

the `{:}` expands the values in the cell array so that this is the equivalent of

```
uicontrol('parent', h.fig, 'units', 'pixels', 'fontunits', 'pixels',...
```

The rest of the code initializes the plug-ins. They all have the same format. The spacecraft model is also read-in from a .mat file and used to initialize the DrawSCPlugIn.

The following code is the orbit simulation. It gets the data from the plug-ins and updates the 3D display.

**Listing C.3:** Initializing the simulation

*PlugInDemo.m*

```

1 function Run
2
3 % Get the simulation data
4 %-----
5 h = GetH;
6 sim = GetSimData( h );
7
8 % Duration
9 %-----
10 duration = datenum(sim.time.duration)*86400;
11
12 % Check duration
13 %-----
14 if( duration == 0 )
15     msgbox('The_duration_is_zero._Will_not_run_the_simulation.');
```

```

16     return
17 end
18
19 % Create the basic state vector from required plug-ins
20 %-----
21 x = [sim.orbit.r; sim.orbit.v];
22 jD = sim.time.jDEpoch;
23 nSim = duration/sim.time.dT;
24 y{1} = zeros(6,nSim);
25 t = 0;
26 DrawSCPlugIn( 'bring_to_front', h.scWindowTag );
```

*PlugInDemo.m*

The simulation is run with the following code. `FOrbCart` returns the state derivatives for the orbit model.

**Listing C.4:** The simulation loop

*PlugInDemo.m*

```

1 for k = 1:nSim
2     % Plotting
3     %-----
4     y{1}(:,k) = x;
5     u(k) = t;
6
7     % Transformation matrices
8     %-----
9     qLVLH = QLVLH( x(1:3), x(4:6) );
10    h.g.body(1).bHinge.q = QPose( qLVLH );
11    h.g.rECI = x(1:3);
12    h.g.qLVLH = qLVLH ;
13    DrawSCPlugIn( 'update_spacecraft', h.scWindowTag, h.g, jD );
14    % Propagate the orbits
15    %-----
16    x = RK4( 'FOrbCart', x, sim.time.dT, t, sim.orbit.mu );
17    % Update the time
18    %-----
19    t = t + sim.time.dT;
20    jD = jD + sim.time.dT/86400;
21 end
```

The plots are created with the following code.

**Listing C.5: Plotting**

PlugInDemo.m

```

1 % Plotting
2 %-----
3 if( ~isempty( y ) )
4   PlotPlugIn( 'clear_plots', h.plotPlugInTag, k );
5   p.xLabel = {'Time_(sec)'};
6   p.yLabel = {'xECI' 'yECI' 'zECI' 'vXECI' 'vYECI' 'vZECI'};
7   p.title = p.yLabel;
8   PlotPlugIn( 'update_labels', h.plotPlugInTag, p );
9   PlotPlugIn( 'add_points', h.plotPlugInTag, struct('x', u, 'y', y) );
10  PlotPlugIn( 'plot', h.plotPlugInTag );
11 end
12
13 jD = sim.time.jDEpoch + (0:(nSim-1))*sim.time.dT/86400;
14 OrbitDisplayPlugIn( 'draw', h.orbitDisplayTag, {y{1}(1:3,:)} , jD,
15 sim.orbit.planet );
16
17 PutH( h );

```

PlugInDemo.m

When you change anything in a plugin the following code is executed. Three plugins are called as part of this code. DrawSCPlugin is called twice, once to update the spacecraft and the second time to update the planet.

**Listing C.6: Update PlugInDemo**

PlugInDemo.m

```

1 function Update
2 % Get the simulation data
3 %-----
4 h = GetH;
5 sim = GetSimData( h );
6
7 % Update the spacecraft state
8 %-----
9 if( isfield( sim.orbit, 'r' ) & isfield( sim.orbit, 'v' ) )
10  qLVLH = QLVLH(sim.orbit.r, sim.orbit.v );
11 else
12  qLVLH = [1;0;0;0];
13 end
14
15 h.g.body(1).bHinge.q = QPose( qLVLH );
16 if( isfield( sim.orbit, 'r' ) )
17 h.g.rECI = sim.orbit.r;
18 else
19 h.g.rECI = [sim.orbit.el(1);0;0];
20 end
21 h.g.qLVLH = qLVLH ;
22 DrawSCPlugin( 'update_spacecraft', h.scWindowTag, h.g, sim.time.jDEpoch
23 );
24 OrbitDisplayPlugIn( 'clear_plot', h.orbitDisplayTag, sim.orbit.planet );
25 DrawSCPlugin( 'update_planet', h.scWindowTag, sim.orbit.planet );
26 PutH( h );

```

PlugInDemo.m

The function GetSimData gets the data from the elements and time plug-ins. The first argument to each plugin is an action and the second is the tag for the plugin. The tag tells MATLAB which copy of the plugin to call.

**Listing: Get data from the plug-ins**

```

1 function sim = GetSimData( h )
2
3 % Get the data from the plug ins
4 %-----

```

---

```

5 sim.orbit = ElementsPlugIn( 'get', h.elementsPlugInTag );
6 sim.time = TimePlugIn ( 'get', h.timePlugInTag );

```

---

The following are utility functions for getting data from the figure handle, putting data into the figure handle and creating uicontrol callback strings.

**Listing C.7: Utilities***PlugInDemo.m*

```

1  %-----
2  % Put the data into the figure handle
3  %-----
4  function PutH( h )
5
6  set( h.fig, 'UserData', h );
7
8  %-----
9  % Get the data from the figure handle
10 %-----
11 function h = GetH
12
13 hFig = findobj( allchild(0), 'flat', 'Tag', 'GUI_PlugIn_Demo' );
14 h = get( hFig, 'userdata' );
15
16 %-----
17 % Create a callback string
18 %-----
19 function c = CreateCallback( action )
20
21 c = ['PlugInDemo(_'' action ''_)'];

```

*PlugInDemo.m*



---

# INDEX

---

- AbsDa, 197
- AddAxes, 30
- AnimationGUI, 88, 115
- AntennaPatch, 28
- ASim, 69
- AtmDens2, 23
- AtmJ70, 110
- AttitudeProfileDemo, 113
  
- BandGap, 189
- BatterySizing, 116
- BHinge, 52
- BloDown, 193, 223
- BloDownMass, 223
- BuildCADModel, 43
- BuildCylinders, 45
- BuildMAP, 123
- BuildSCModelForImaging, 201
- BuildSimpleSat, 47
  
- C2DZOH, 11, 29, 36
- CAD, 28
- CCVDemo, 101
- CDKFDemo, 210
- Constant, 155, 235
- constant database, 235
- ControlDesignGUI, 99
- coordinate transformation, 28
- CoordinateTransform, 162
- CreateBody, 43, 51, 52
- CreateComponent, 43, 52
- cross, 40
- CubeSatAero, 112
- CubeSatFaces, 109
- CubeSatGGStabilized, 110
- CubeSatModel, 108, 109
- CubeSatPropulsion, 116
- CubeSatRadiationPressure, 112
- CubeSatRHS, 109
- CubeSatRWASimulation, 110
- CubeSatSimulation, 110
  
- database, 235
- Date2JD, 25
  
- DemoPSS, 4, 17
- DesignMagneticTorquer, 116
- DeviceProperties, 52
- DQCE, 208
- DragCompensationDemo, 147
- DragDemo, 139
- DrawSCPlanPlugIn, 43, 64
- DrawSCPlugIn, 43, 67
- DrawSCPlugin, 244
- DrawSpacecraftInOrbit.m, 114
- DSpnStab, 26
- DVIDrift, 171
- DVLDrift, 171
  
- EarthRad, 25
- EarthRot, 24
- EarthRotationZero, 162
- ECIToECIR, 163
- ECIToEF, 77
- ElectricPropulsion, 194
- ElementsPlugIn, 245
- ephemeris, 30
- Eul2Mat, 39, 52
- Eul2Q, 40
- Euler angle, 40
- EVAssgnC, 101
- ExamplePointingBudget, 95
- ExportCAD, 63
- ExportOBJ, 63
  
- FAero, 157, 159
- FAtmDensity, 155, 157
- FBudget, 167
- FEngine, 157
- FFSimDemo, 110
- FGs, 129
- FileHelp, 14
- Finder, 16, 18, 33
- FOrbCart, 244
- FPlantNoise, 157
- FRB, 37
- Frustrum, 55
- FSolar, 157
  
- GenericProperties, 50

- GlideslopePlanner, 150  
 GroundCoverage, 163  
 GroundTrack, 114  
  
 Hall thruster, 29  
 HallThrusterModel, 29  
 help, 7  
  
 IntegrationAccuracyDemo, 143  
 InterpolateState, 79  
 IsothermalCubeSatDemo, 116  
  
 JD2Date, 32  
 JPLPlanetDemo, 80  
 Julian Date, 32  
  
 KalmanFilterDemo, 208  
 Kepler's Equation, 26  
 KFilter, 208  
 KFSunAndEarthTest2, 209  
  
 LADARTransmitPower, 185  
 LEOPAnalysis.m, 146  
 LinkAnalysisTool, 181  
 LinkExamples.m, 183  
 LinkOrbitAnalysis, 116  
 LoadCAD, 43, 61  
 LoadCatalog, 213, 217  
 LoadGOES, 229  
 LoadMag, 230  
 LoadSatP, 230  
 LoadSers, 230  
 LossAtmosphericGas, 181  
 LossFreeSpace, 181  
 LossPrecipitation, 181  
 LowThrustDemo, 137  
 LQC, 100  
 LQFullState, 100  
  
 MagSimWithEstimator, 209  
 MagTComp, 26  
 MAPControlSim, 119, 125  
 MarsRot, 41  
 Mat2Eul, 39  
 MBModel, 29  
 MergeConstantDB, 236  
 MoonRot, 41  
 MoonV1, 77  
 MoonV2, 77  
 multibody dynamics  
     tree, 74  
     wires, 69  
  
 Notch, 38  
  
 OBJ, 63  
  
 ObservationTimeWindows, 113  
 ObservationTimeWindowsPlot, 113  
 OPBatchDemo, 164  
 OpticalBackgroundNoise, 180  
 OpticalLinkBudget, 184  
 OpticalReceiverNoiseDensity, 181  
 OptimizeTSTO, 195  
 OrbitAndSensorConeAnimation.m, 115  
 OrbitKalmanFilter, 210  
 OrbMnvrDemo, 145  
  
 PackageOrbitDataForPlayback, 115  
 PBudget, 95  
 PID3Axis, 110  
 PIDesign, 126  
 PIDMIMO, 36, 110, 126  
 PinholeStarCamera, 216, 217  
 PivotMch, 30  
 PlanckL, 197  
 PlanetPosition, 77, 78  
 PlanetPosJPL, 78, 80  
 Planets, 77  
 PlaybackOrbitSim, 115  
 Plot2D, 30, 73  
 Plot3D, 30  
 PlotOrbit, 114  
 PlotOrbitPage, 30  
 PlotPlugIn, 245  
 PlottingTool, 82  
 PlotV, 130  
 PlugInDemo, 239, 240  
 PointingBudgetGUI, 96  
 PressureKF, 223  
 PropagateOrbitPlugIn, 153  
 PropagateOrbitPlugin, 140  
 PSSSetPaths, 4  
  
 Q2Eul, 40  
 Q2Mat, 40  
 QCE, 208  
 QForm, 28, 40  
 QLVLH, 28, 37, 41  
 QMult, 40  
 QNadirPoint, 41  
 QPose, 40  
 QSunNadir, 41  
 QTForm, 40  
 quaternion, 28, 40  
 QZero, 40  
  
 RadarCrosssection, 186  
 RadarEquation, 186  
 RapidSwath, 113  
 ReadOutputFile.m, 90  
 rigid body, 69

RK4, [37](#), [72](#), [75](#), [129](#)  
RK45, [129](#)  
RollYawDoublePivot.m, [120](#)  
RVFromKepler, [26](#)

SCThermalDemo, [201](#)  
SCTTutorial, [35](#)  
selenographic latitude, [41](#)  
selenographic longitude, [41](#)  
Shannon, [180](#)  
ShuttleSim, [67](#)  
SkewSymm, [40](#)  
solar flux, [29](#)  
SolarCell, [189](#)  
SolarCellEff, [189](#)  
SolarCellPower, [109](#)  
SolarFlx, [29](#)  
SolarSys, [77](#), [78](#)  
SolarSysJPL, [78](#), [80](#)  
SolidR, [193](#)  
SpaceCamera, [201](#)  
StarDataGeneration, [215](#)  
STKOrbit, [163](#)  
SunEarthADSim, [209](#)  
SunMagAttDet, [209](#)  
SunNadirPointingSim.m, [120](#)  
SunV1, [30](#), [77](#), [163](#)  
SunV2, [30](#), [77](#)  
SunVectorECI, [77](#)

TEarth, [77](#)  
TEclipse, [77](#)  
ThermalEquilibrium, [199](#)  
ThermalImager, [201](#)  
Thermnet, [199](#)  
time, [32](#)  
TimeGUI, [72](#)  
TimePlugIn, [245](#)  
TMAPSim, [124](#)  
TopexDemo, [140](#)  
TPlanets, [77](#)  
TransformECI2Hills.m, [87](#)  
Tree, [51](#), [75](#)  
TreeAdd, [75](#)  
TreeInit, [75](#)  
TreePlot, [76](#)  
TreePrt, [75](#)  
TSky, [181](#)  
TSolid, [193](#)  
TSTODemo, [195](#)  
TwoSpacecraftOrbitSimulation, [110](#)  
TwoSpacecraftSimpleOrbitSimulation, [110](#)

UDKalmanFilter, [208](#)  
UDKFDemo, [208](#)  
UKF, [223](#)  
VisualCommander, [63](#)  
Wavefront OBJ, [63](#)  
Weather, [230](#)  
WiensDsp, [197](#)  
WireC, [70](#), [72](#)  
WireFRB, [70](#), [72](#)