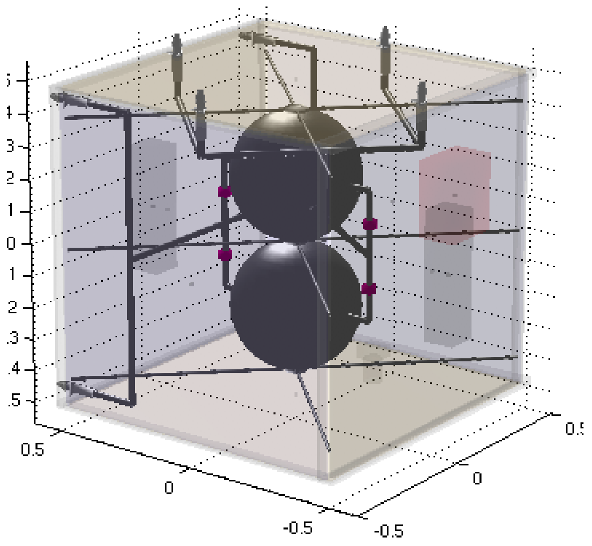
Princeton SATELLITE SYSTEMS

Spacecraft Control Toolbox Academic Edition

Release 2017.1



SCT Academic Edition

Compiled on June 26, 2017

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

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

Contents

CODE EXAMPLES

List of Figures

CHAPTER 1

INTRODUCTION

This chapter shows you how to install the Spacecraft Control Toolbox Academic Edition and how the product is organized. This product is intended for use in a classroom computer lab during undergraduate and graduate attitude control and spacecraft design courses.

1.1 Organization

The Spacecraft Control Toolbox (SCT) 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 special simulation plotting tools are in *Plotting*.

Table 1.1: SCT Academic 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 nanonsatellite 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.

The Professional Edition adds the modules in the next next table, as well as the capability to add on additional modules. The Academic Edition consists of the full *Common*, *CubeSat*, *SC*, and *Plotting* modules, plus a few files from the Professional modules needed to run certain CubeSat demos.

Table 1.2: Spacecraft Control Toolbox Professional Modules

Module		Function
CADPro		More elaborate models and graphics functions
SCPro		Additional high-fidelity models for environment, sensors, actuators.
SpacecraftEstimation		Attitude and orbit estimation. Stellar attitude determination and Kalman filtering.
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.

Propulsion		Electric and chemical propulsion. Launch vehicle analysis.
Thermal		Basic thermal modeling
Formation Flying	Add-On	Formation flying dynamics and control
FusionPropulsion	Add-On	Fusion rockets
LaunchVehicle	Add-On	Staged conventional rockets, also required the Aircraft Control Toolbox
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

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

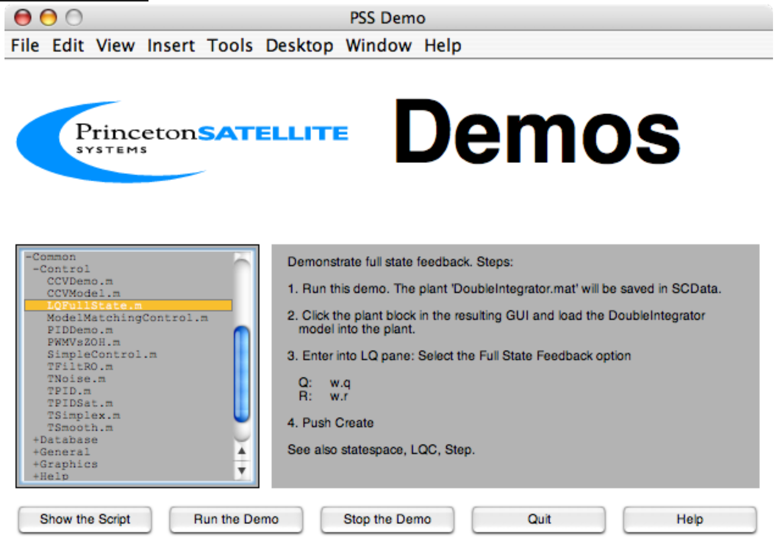
There are two sets of demos created specifically for Academic Edition customers that you should review. Each set combines a CAD model with a simulation script. These simple simulations combine multiple toolbox elements including control laws, visualization, and disturbances.

- `BuildSatWThrusters` and `REAControl` - these scripts build and simulate a spacecraft, REASat, for analyzing control using thrusters and checking the sun angle of a single sensor. (REA stands for rocket engine assembly.) This script does not use the full `Thruster` model, but calls `Simplex` using the control command for analyzing the resulting pulsewidth demands. The PSS function `PIDMIMO` is used to design a control law. The simulation shows how to use two simulation functions, `FRB` for rigid body dynamics and `FOrbCart` for simple orbit dynamics.
- `BuildXYZSat` and `Attitude3D` - this CAD model script and corresponding simulation are similar to the previous demo but has options for 3D graphics and disturbances and excludes the sensors and actuators. The alternative to the 3D graphics includes our `Plot3D` function for viewing orbits and `AnimQ` for animating a quaternion. This CAD model, XYZSat, has color-coded axes to aid the 3D visualization.

Two functions that you should also be familiar with generally are `DemoPSS` and `FileHelp`. `FileHelp` and `DemoPSS` provide the best way to get an overview of the Spacecraft Control Toolbox. The `FileHelp` function, discussed in more detail in the next chapter, 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.

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 GUI display in Figure 1.1 shows some demos in the Common module.

Figure 1.1: `DemoPSS` window


PSS Demo
File Edit View Insert Tools Desktop Window Help
PrincetonSATELLITE SYSTEMS
Demos
Show the Script
Run the Demo
Stop the Demo
Quit
Help

```
-Common
 -Control
   CCVDemo.m
   CCVModel.m
   LQFullState.m
   ModelMatchingControl.m
   PIDDemo.m
   PWMVsZOH.m
   SimpleControl.m
   TFiltRO.m
   TNoise.m
   TPID.m
   TPIDSat.m
   TSimplex.m
   TSmooth.m
 +Database
 +General
 +Graphics
 +Help
```

Demonstrate full state feedback. Steps:

1. Run this demo. The plant 'DoubleIntegrator.mat' will be saved in SCData.
2. Click the plant block in the resulting GUI and load the DoubleIntegrator model into the plant.
3. Enter into LQ pane: Select the Full State Feedback option

   Q: w.q
   R: w.r
4. Push Create

See also statespace, LQC, Step.

The Common/Control demos are visible in the hierarchical menu to the left. The highest level of this menu shows the folders within the toolbox. You can add your own demo scripts to the demo folders so that they can appear in the display.

CHAPTER 2

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 the next page. The index page of the SCT documentation is shown in Figure 2.2 on page 7.

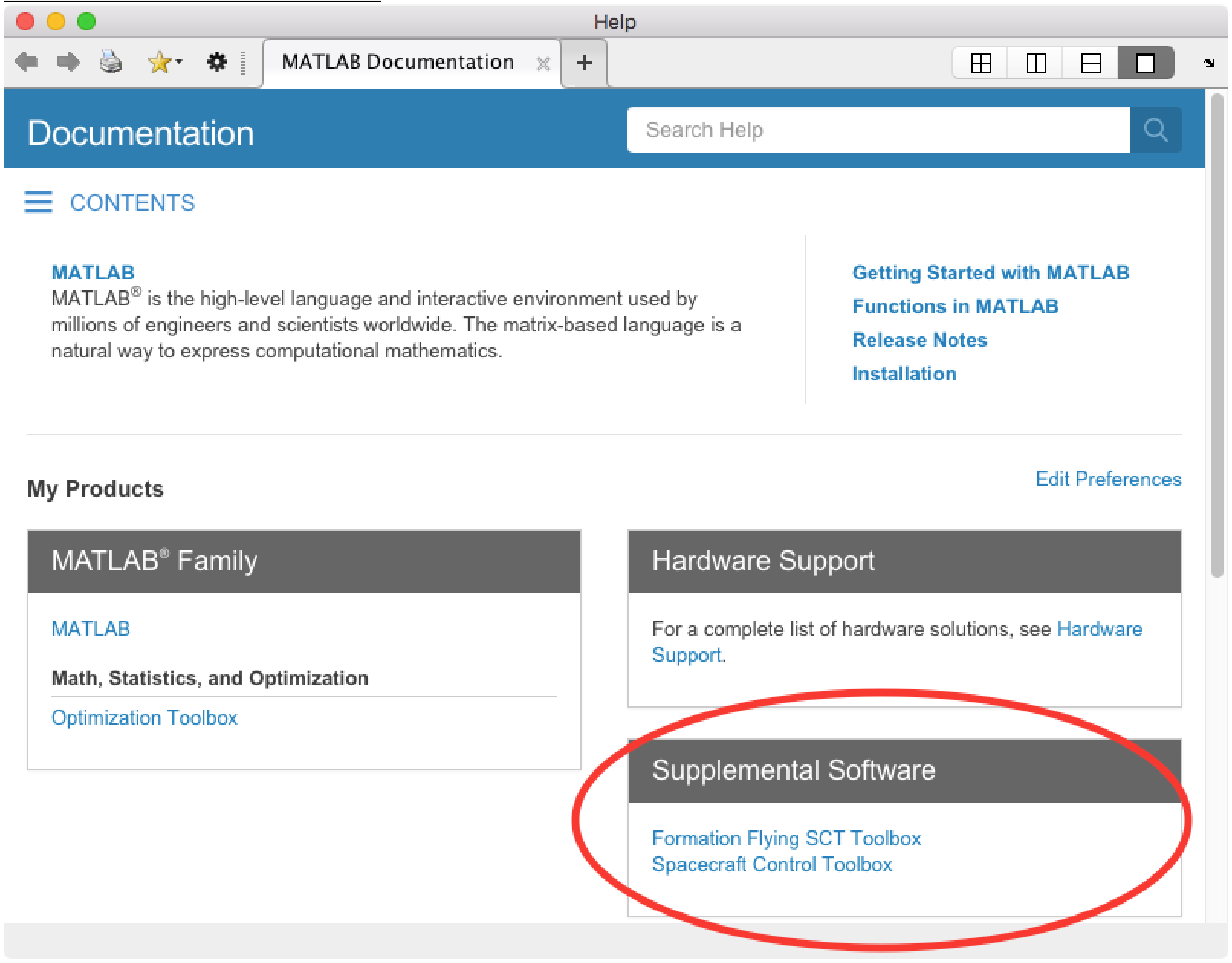
The help window from R2011b and earlier is depicted in Figure 2.3 on page 7.

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



MATLAB
MATLAB® is the high-level language and interactive environment used by millions of engineers and scientists worldwide. The matrix-based language is a natural way to express computational mathematics.

Getting Started with MATLAB
Functions in MATLAB
Release Notes
Installation

My Products

Edit Preferences

MATLAB® Family

MATLAB

Math, Statistics, and Optimization

Optimization Toolbox

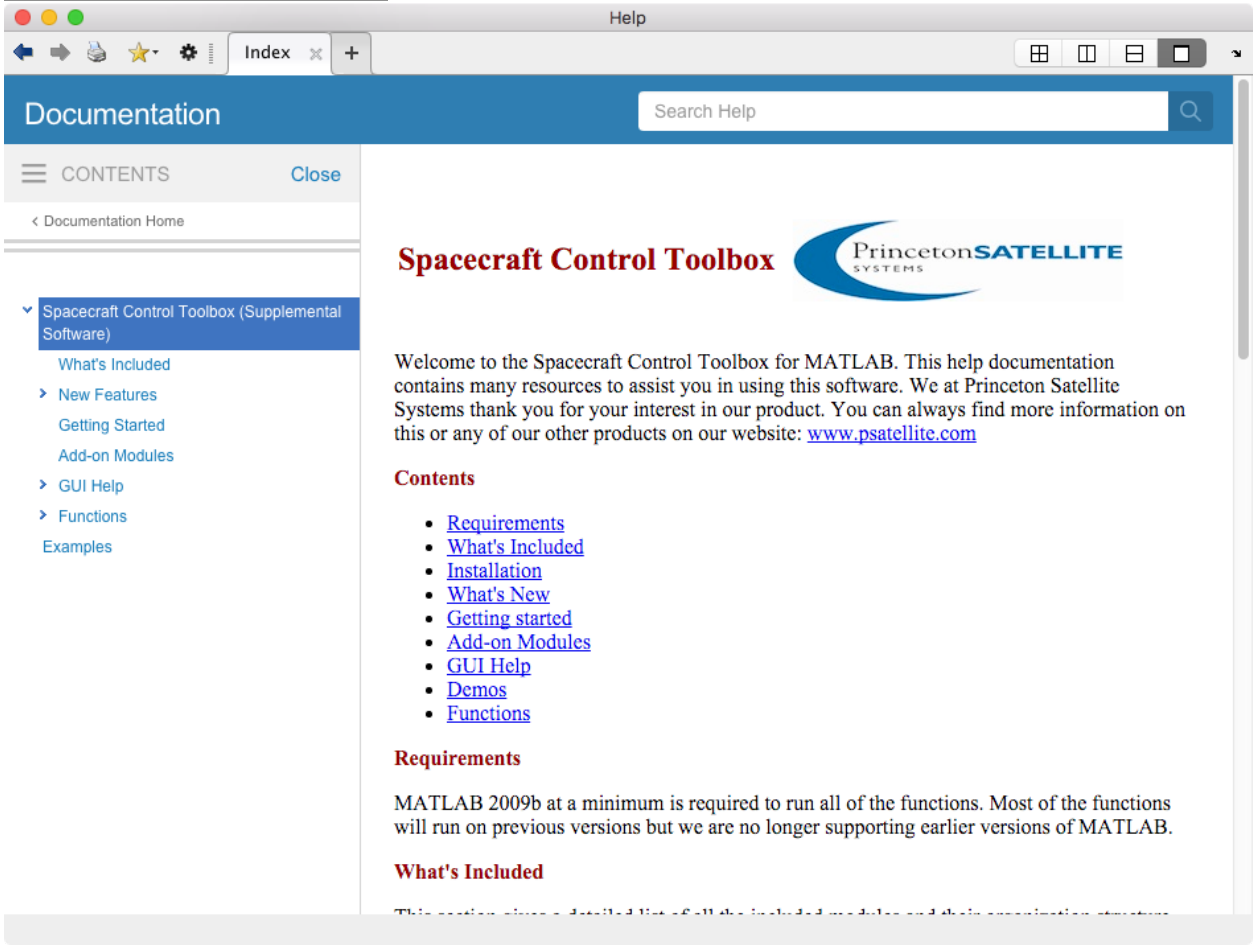
Hardware Support

For a complete list of hardware solutions, see Hardware Support.

Supplemental Software

Formation Flying SCT Toolbox
Spacecraft Control Toolbox

Figure 2.2: Toolbox Documentation Main Page, R2016b



Spacecraft Control Toolbox

Welcome to the Spacecraft Control Toolbox for MATLAB. This help documentation contains many resources to assist you in using this software. We at Princeton Satellite Systems thank you for your interest in our product. You can always find more information on this or any of our other products on our website: www.psatellite.com

Contents

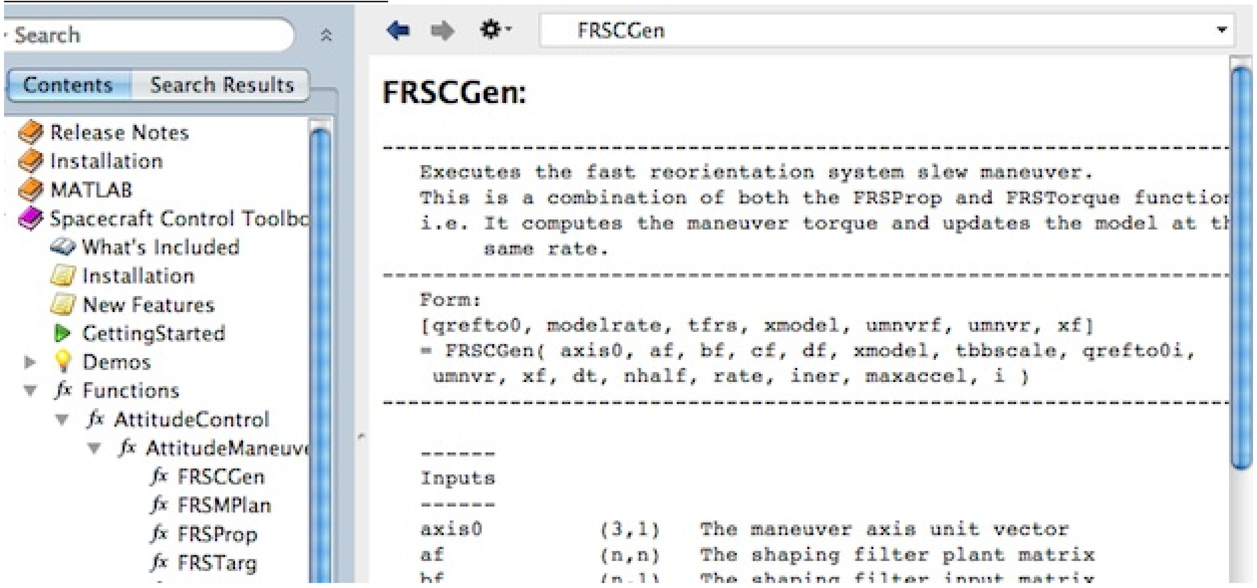
- Requirements
- What's Included
- Installation
- What's New
- Getting started
- Add-on Modules
- GUI Help
- Demos
- Functions

Requirements

MATLAB 2009b 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 earlier versions of MATLAB.

What's Included

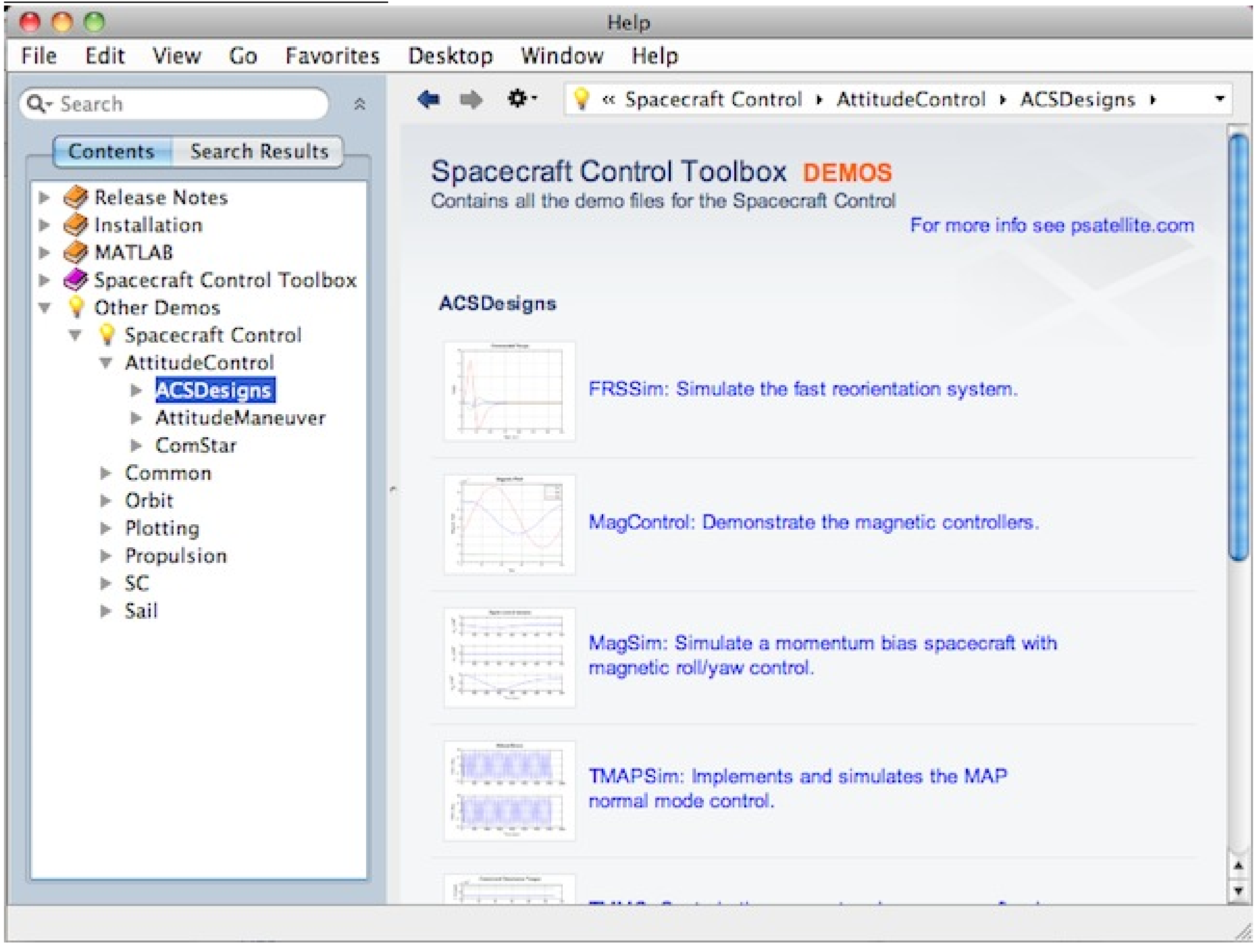
Figure 2.3: Toolbox Documentation, R2011b



FRSCGen:

```
--------------------------------------------------------------------
   Executes the fast reorientation system slew maneuver.
   This is a combination of both the FRSProp and FRSTorque functio
   i.e. It computes the maneuver torque and updates the model at th
        same rate.
--------------------------------------------------------------------
   Form:
   [qrefto0, modelrate, tfrs, xmodel, umnvrf, umnvr, xf]
   = FRSCGen( axis0, af, bf, cf, df, xmodel, tbbscale, qrefto0i,
    umnvr, xf, dt, nhalf, rate, iner, maxaccel, i )
--------------------------------------------------------------------


   ------
   Inputs
   ------
   axis0          (3,1)    The maneuver axis unit vector
   af             (n,n)    The shaping filter plant matrix
```

Figure 2.4: Toolbox Demos



Spacecraft Control Toolbox DEMOS

Contains all the demo files for the Spacecraft Control

For more info see psatellite.com

ACSDesigns

FRSSim: Simulate the fast reorientation system.

MagControl: Demonstrate the magnetic controllers.

MagSim: Simulate a momentum bias spacecraft with magnetic roll/yaw control.

TMAPSim: Implements and simulates the MAP normal mode control.

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, El2RV,
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

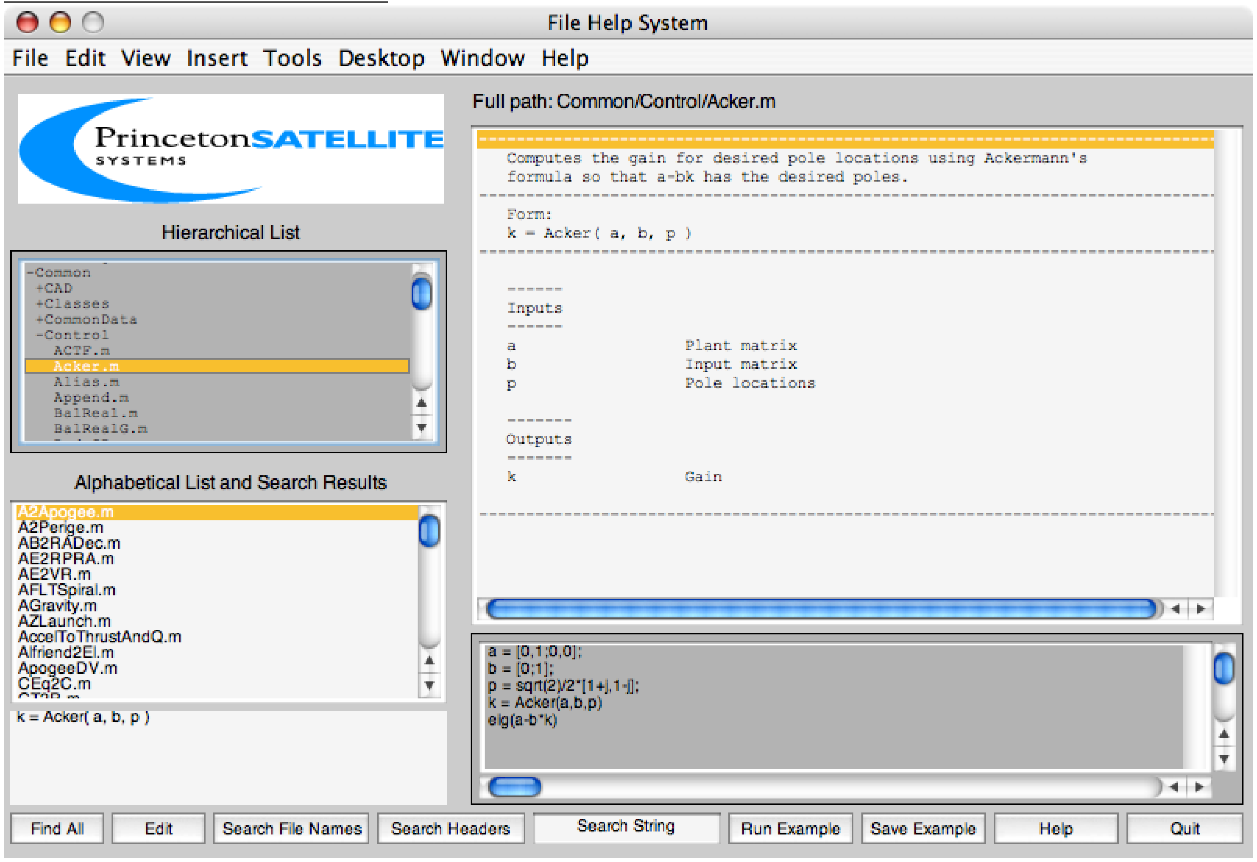
2.3.1 Introduction

When you type

```
FileHelp
```

the FileHelp GUI appears, Figure 2.5.

Figure 2.5: The file help GUI



Full path: Common/Control/Acker.m

```
-Common
 +CAD
 +Classes
 +CommonData
 -Control
   ACTF.m
   Acker.m
   Alias.m
   Append.m
   BalReal.m
   BalRealG.m
```

```
A2Apogee.m
A2Perige.m
AB2RADec.m
AE2RPRA.m
AE2VR.m
AFLTSpiral.m
AGravity.m
AZLaunch.m
AccelToThrustAndQ.m
Alfriend2El.m
ApogeeDV.m
CEq2C.m
```

```
k = Acker( a, b, p )
```

```
-------------------------------------------------------------------------
  Computes the gain for desired pole locations using Ackermann's
  formula so that a-bk has the desired poles.
-------------------------------------------------------------------------
  Form:
  k = Acker( a, b, p )
-------------------------------------------------------------------------

  ------
  Inputs
  ------
  a                   Plant matrix
  b                   Input matrix
  p                   Pole locations

  -------
  Outputs
  -------
  k                   Gain

-------------------------------------------------------------------------
```

```
a = [0,1;0,0];
b = [0;1];
p = sqrt(2)/2*[1+j,1-j];
k = Acker(a,b,p)
eig(a-b*k)
```

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.

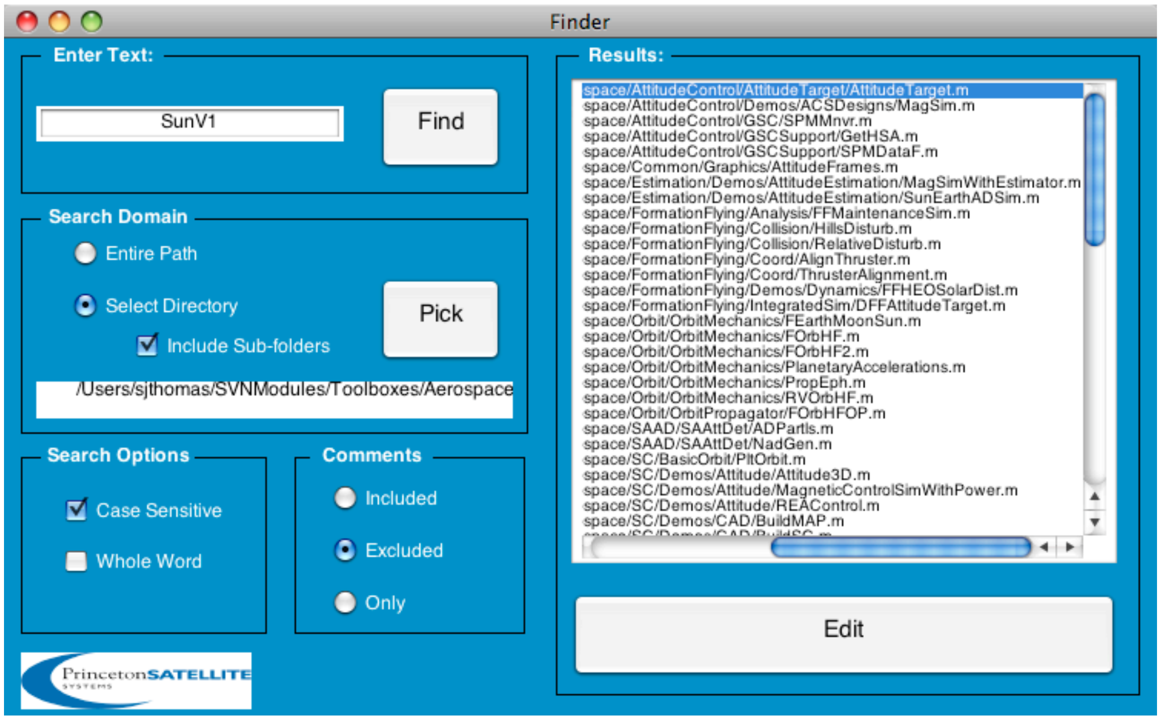
2.4.4 Search String Edit Box

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 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.



Enter Text: SunV1

Search Domain: Entire Path [✓ Select Directory]; Include Sub-folders ✓

/Users/sjthomas/SVNModules/Toolboxes/Aerospace

Search Options: Case Sensitive ✓; Whole Word ☐

Comments: Included [✓ Excluded] Only

Results:

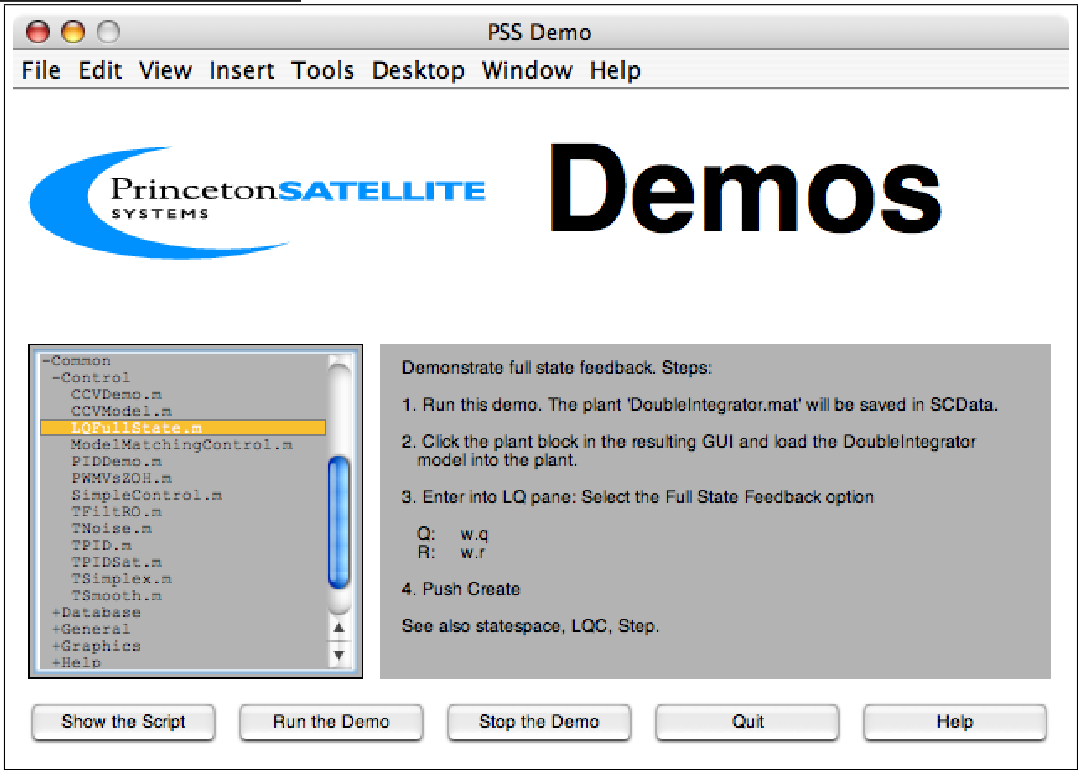
```
space/AttitudeControl/AttitudeTarget/AttitudeTarget.m
space/AttitudeControl/Demos/ACSDesigns/MagSim.m
space/AttitudeControl/GSC/SPMMnvr.m
space/AttitudeControl/GSCSupport/GetHSA.m
space/AttitudeControl/GSCSupport/SPMDataF.m
space/Common/Graphics/AttitudeFrames.m
space/Estimation/Demos/AttitudeEstimation/MagSimWithEstimator.m
space/Estimation/Demos/AttitudeEstimation/SunEarthADSim.m
space/FormationFlying/Analysis/FFMaintenanceSim.m
space/FormationFlying/Collision/HillsDisturb.m
space/FormationFlying/Collision/RelativeDisturb.m
space/FormationFlying/Coord/AlignThruster.m
space/FormationFlying/Coord/ThrusterAlignment.m
space/FormationFlying/Demos/Dynamics/FFHEOSolarDist.m
space/FormationFlying/IntegratedSim/DFFAttitudeTarget.m
space/Orbit/OrbitMechanics/FEarthMoonSun.m
space/Orbit/OrbitMechanics/FOrbHF.m
space/Orbit/OrbitMechanics/FOrbHF2.m
space/Orbit/OrbitMechanics/PlanetaryAccelerations.m
space/Orbit/OrbitMechanics/PropEph.m
space/Orbit/OrbitMechanics/RVOrbHF.m
space/Orbit/OrbitPropagator/FOrbHFOP.m
space/SAAD/SAAttDet/ADPartls.m
space/SAAD/SAAttDet/NadGen.m
space/SC/BasicOrbit/PltOrbit.m
space/SC/Demos/Attitude/Attitude3D.m
space/SC/Demos/Attitude/MagneticControlSimWithPower.m
space/SC/Demos/Attitude/REAControl.m
space/SC/Demos/CAD/BuildMAP.m
```

2.6 DemoPSS

If you type `DemoPSS` you will see the GUI in Figure 2.6. 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


PSS Demo
File Edit View Insert Tools Desktop Window Help
PrincetonSATELLITE SYSTEMS
Demos
Show the Script
Run the Demo
Stop the Demo
Quit
Help

```
-Common
 -Control
   CCVDemo.m
   CCVModel.m
   LQFullState.m
   ModelMatchingControl.m
   PIDDemo.m
   PWMVsZOH.m
   SimpleControl.m
   TFiltRO.m
   TNoise.m
   TPID.m
   TPIDSat.m
   TSimplex.m
   TSmooth.m
 +Database
 +General
 +Graphics
 +Help
```

Demonstrate full state feedback. Steps:

1. Run this demo. The plant 'DoubleIntegrator.mat' will be saved in SCData.

2. Click the plant block in the resulting GUI and load the DoubleIntegrator model into the plant.

3. Enter into LQ pane: Select the Full State Feedback option

   Q: w.q
   R: w.r

4. Push Create

See also statespace, LQC, Step.

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.

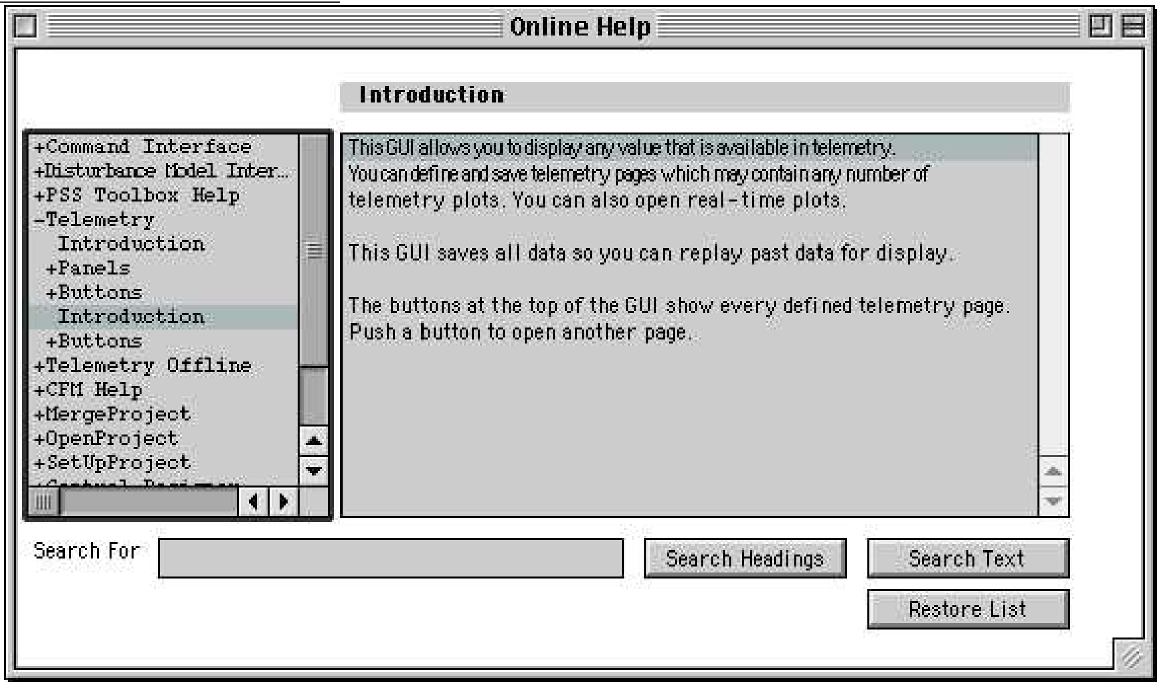
2.7 Graphical User Interface Help

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 next 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 into

Figure 2.7: On-line Help



This GUI allows you to display any value that is available in telemetry. You can define and save telemetry pages which may contain any number of telemetry plots. You can also open real-time plots.

This GUI saves all data so you can replay past data for display.

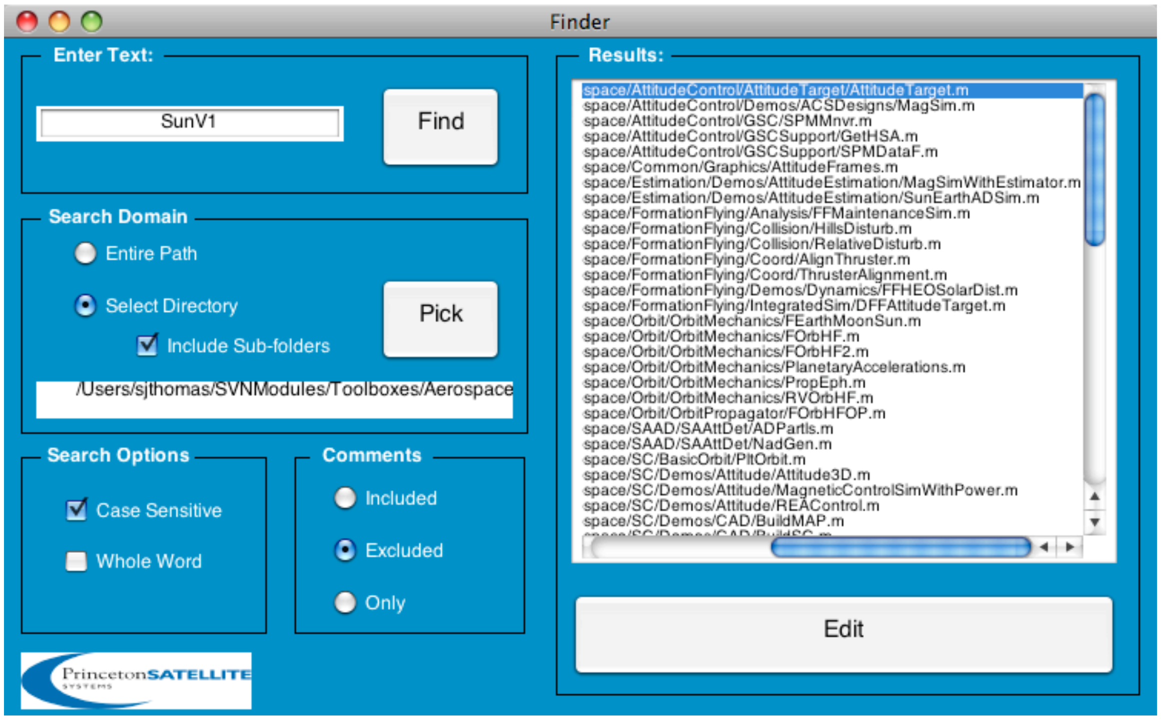
The buttons at the top of the GUI show every defined telemetry page. Push a button to open another page.

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.



Enter Text: SunV1

Search Domain: Entire Path [✓ Select Directory]

[✓] Include Sub-folders

/Users/sjthomas/SVNModules/Toolboxes/Aerospace

Search Options: [✓] Case Sensitive [ ] Whole Word

Comments: Included [✓ Excluded] Only

Results:

```
space/AttitudeControl/AttitudeTarget/AttitudeTarget.m
space/AttitudeControl/Demos/ACSDesigns/MagSim.m
space/AttitudeControl/GSC/SPMMnvr.m
space/AttitudeControl/GSCSupport/GetHSA.m
space/AttitudeControl/GSCSupport/SPMDataF.m
space/Common/Graphics/AttitudeFrames.m
space/Estimation/Demos/AttitudeEstimation/MagSimWithEstimator.m
space/Estimation/Demos/AttitudeEstimation/SunEarthADSim.m
space/FormationFlying/Analysis/FFMaintenanceSim.m
space/FormationFlying/Collision/HillsDisturb.m
space/FormationFlying/Collision/RelativeDisturb.m
space/FormationFlying/Coord/AlignThruster.m
space/FormationFlying/Coord/ThrusterAlignment.m
space/FormationFlying/Demos/Dynamics/FFHEOSolarDist.m
space/FormationFlying/IntegratedSim/DFFAttitudeTarget.m
space/Orbit/OrbitMechanics/FEarthMoonSun.m
space/Orbit/OrbitMechanics/FOrbHF.m
space/Orbit/OrbitMechanics/FOrbHF2.m
space/Orbit/OrbitMechanics/PlanetaryAccelerations.m
space/Orbit/OrbitMechanics/PropEph.m
space/Orbit/OrbitMechanics/RVOrbHF.m
space/Orbit/OrbitPropagator/FOrbHFOP.m
space/SAAD/SAAttDet/ADPartls.m
space/SAAD/SAAttDet/NadGen.m
space/SC/BasicOrbit/PltOrbit.m
space/SC/Demos/Attitude/Attitude3D.m
space/SC/Demos/Attitude/MagneticControlSimWithPower.m
space/SC/Demos/Attitude/REAControl.m
space/SC/Demos/CAD/BuildMAP.m
```

2.9 Technical Support

Contact support@psatellite.com for free email technical support. We are happy to add functions and demos for our customers when asked.

CHAPTER 3

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.

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.

Table 3.2: Features in Spacecraft Control Toolbox functions

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

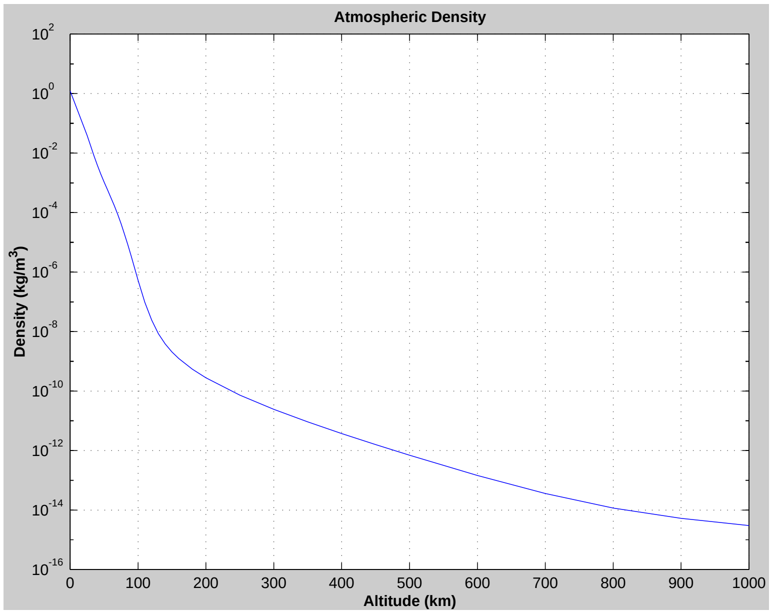
These are illustrated in the examples given below.

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

Figure 3.1: Atmospheric density from `AtmDens2`



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
```

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 the following page. 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

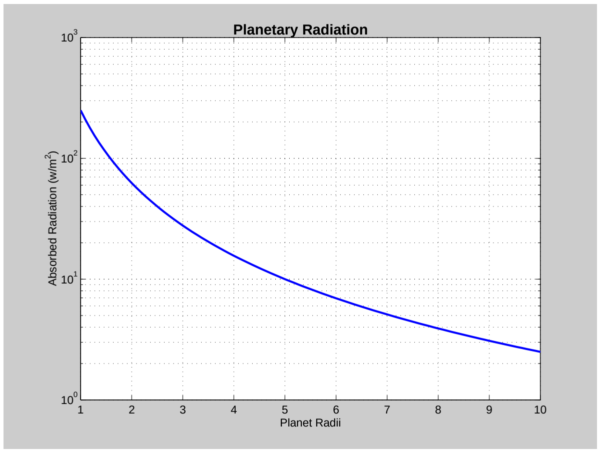
Some functions can take different kinds of inputs. An example is Date2JD. You can pass it either an array

```
[ 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 the next page. One means stable. `DSpnStab` is a typical function in SCT. If no outputs are specified it generates a plot.

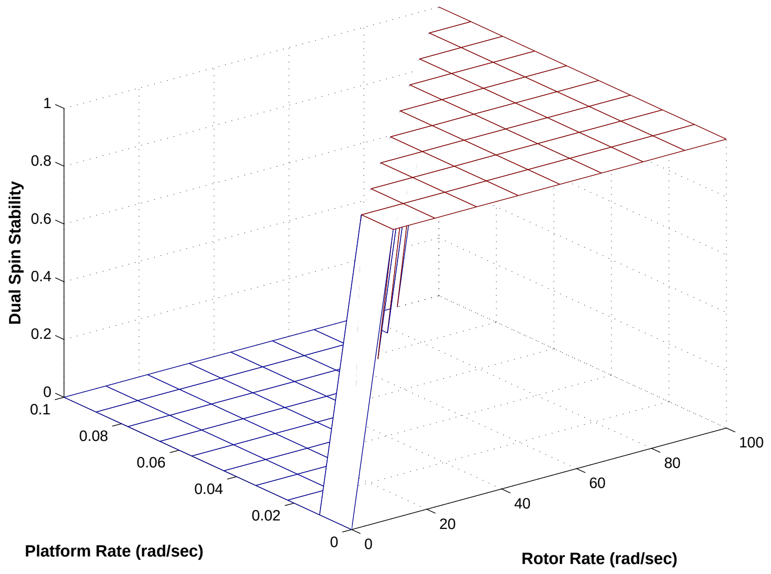
Next, compare magnetic torquers with thrusters at geosynchronous altitude. The specific impulse is 120 sec. and the moment arm is 0.8 m. The plot in Example 3.2 on the facing page shows the trade-off between thrusters and magnetic torquers for geosynchronous orbit where the magnetic field is 75 nano-Tesla.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

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

Figure 3.2: Dual Spin Stability



Example 3.2 Torquer and thruster comparison results

```
MagTComp( 120, 0.8, 75e-9 )
```

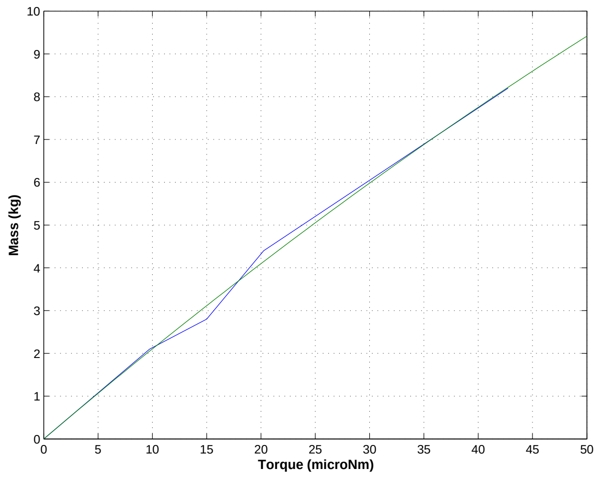
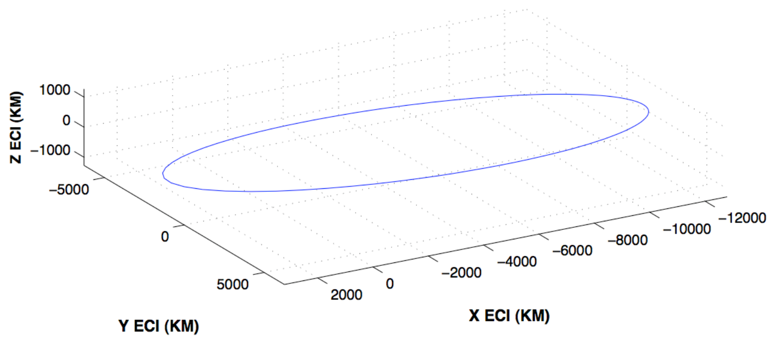
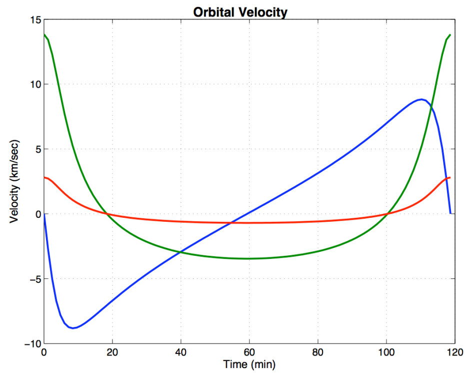


Figure 3.3: Elliptical orbit from `RVFromKepler`





```
>> [r,v] = El2RV( el )

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

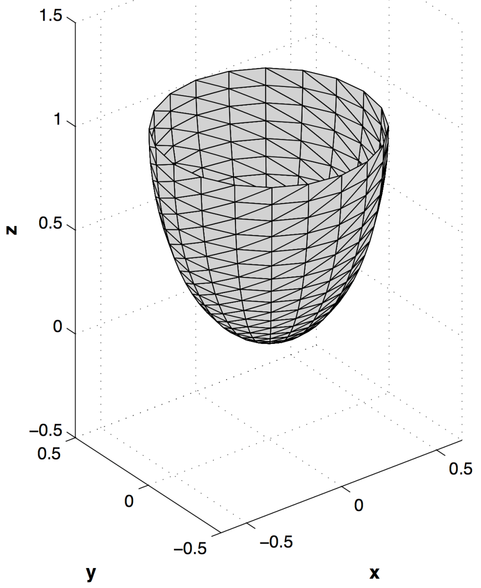
```
>> q = QLVLH( [7000;0;0],[0;7.5;0])
>> QForm( q, [0;0;1] )

q =
                       0.5
                       0.5
                       0.5
                      -0.5
ans =
     0
    -1
     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.

Example 3.3 Antenna patch

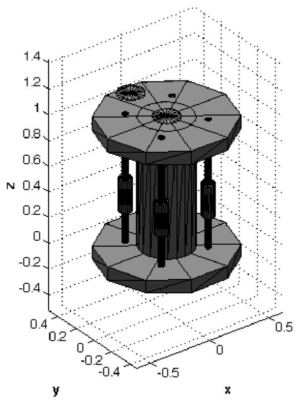
```
AntennaPatch
```



A Hall thruster can be drawn with `HallThrusterModel` as shown in Example 3.4.

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.

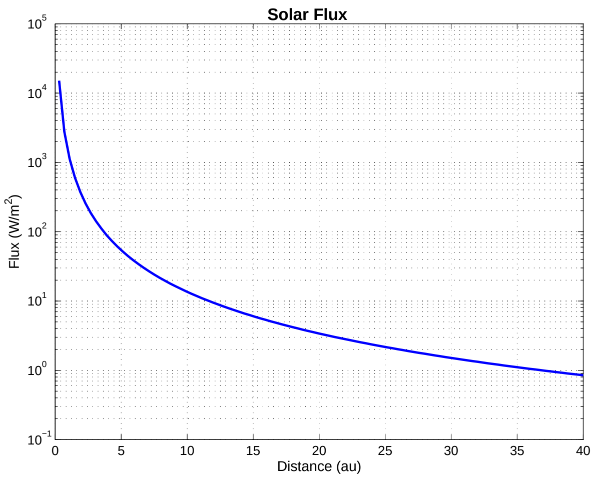
```
ans =
        0 + 0.00007291000000i
        0 - 0.00007291000000i
        0 + 0.03994167200000i
        0 - 0.03994167200000i
        0
        0
```

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 following 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 the next page.

```
>> 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 facing page).

Figure 3.4: A sine wave using `Plot2D`

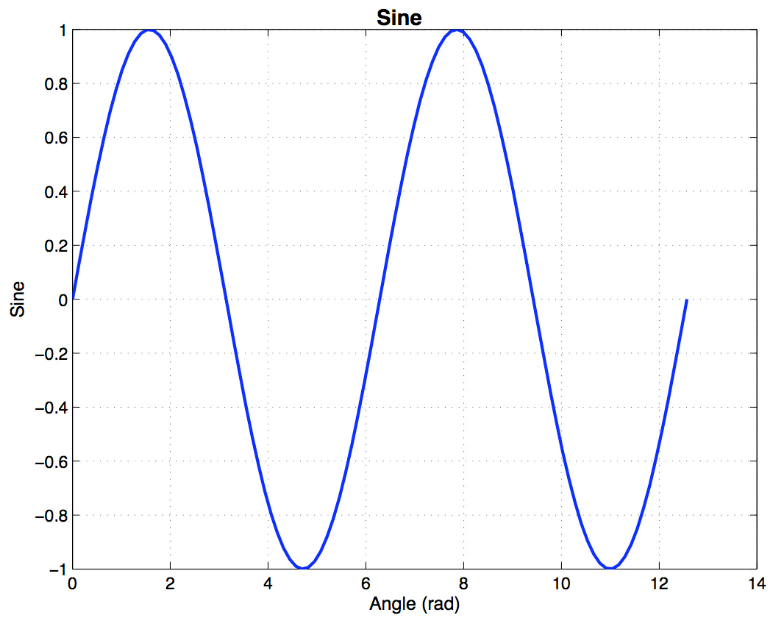
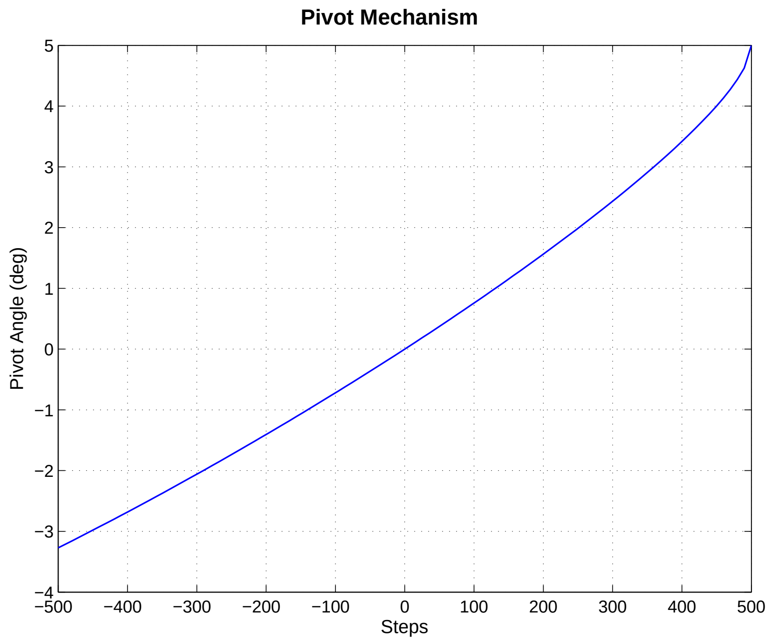


Figure 3.5: Pivot mechanism



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]
```

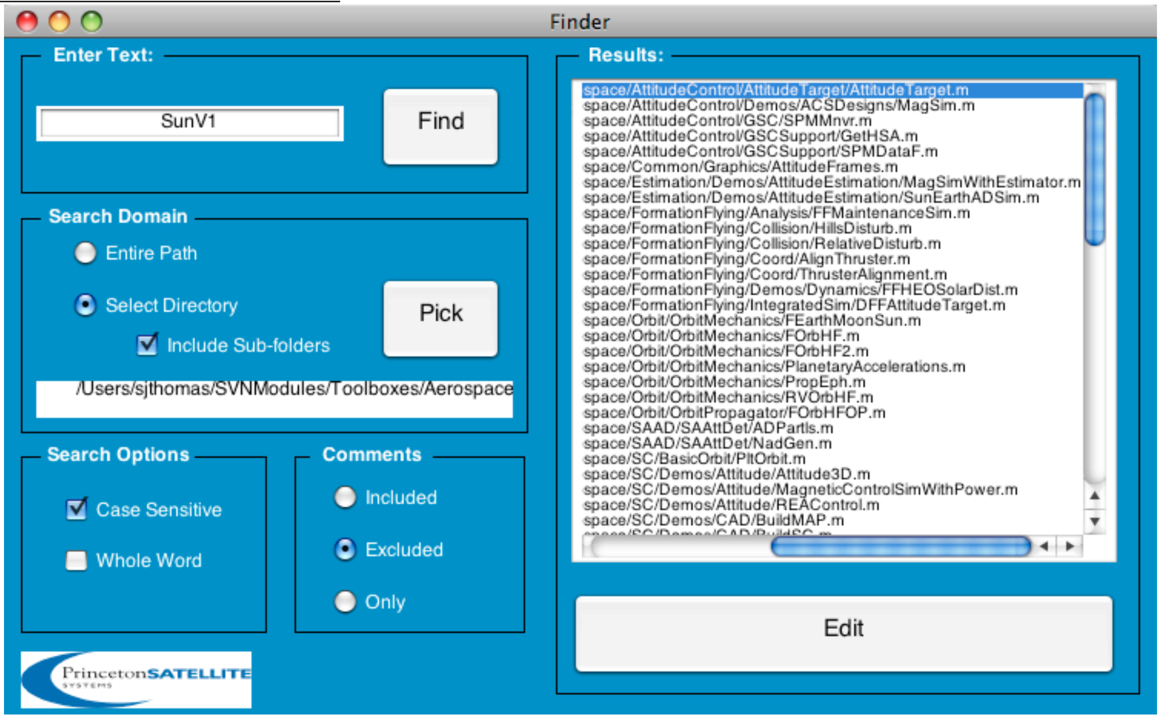
3.5 GUI Tools in the Toolbox

3.5.1 Finder GUI

The `Finder` GUI, shown in Figure 3.6, 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 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 list, you can open any file by clicking the Edit button.

Figure 3.6: Finder window



Enter Text: SunV1

Search Domain: Entire Path [✓ Select Directory]

[✓] Include Sub-folders

/Users/sjthomas/SVNModules/Toolboxes/Aerospace

Search Options: [✓] Case Sensitive, [ ] Whole Word

Comments: Included [✓ Excluded] Only

Results:

```
space/AttitudeControl/AttitudeTarget/AttitudeTarget.m
space/AttitudeControl/Demos/ACSDesigns/MagSim.m
space/AttitudeControl/GSC/SPMMnvr.m
space/AttitudeControl/GSCSupport/GetHSA.m
space/AttitudeControl/GSCSupport/SPMDataF.m
space/Common/Graphics/AttitudeFrames.m
space/Estimation/Demos/AttitudeEstimation/MagSimWithEstimator.m
space/Estimation/Demos/AttitudeEstimation/SunEarthADSim.m
space/FormationFlying/Analysis/FFMaintenanceSim.m
space/FormationFlying/Collision/HillsDisturb.m
space/FormationFlying/Collision/RelativeDisturb.m
space/FormationFlying/Coord/AlignThruster.m
space/FormationFlying/Coord/ThrusterAlignment.m
space/FormationFlying/Demos/Dynamics/FFHEOSolarDist.m
space/FormationFlying/IntegratedSim/DFFAttitudeTarget.m
space/Orbit/OrbitMechanics/FEarthMoonSun.m
space/Orbit/OrbitMechanics/FOrbHF.m
space/Orbit/OrbitMechanics/FOrbHF2.m
space/Orbit/OrbitMechanics/PlanetaryAccelerations.m
space/Orbit/OrbitMechanics/PropEph.m
space/Orbit/OrbitMechanics/RVOrbHF.m
space/Orbit/OrbitPropagator/FOrbHFOP.m
space/SAAD/SAAttDet/ADPartls.m
space/SAAD/SAAttDet/NadGen.m
space/SC/BasicOrbit/PltOrbit.m
space/SC/Demos/Attitude/Attitude3D.m
space/SC/Demos/Attitude/MagneticControlSimWithPower.m
space/SC/Demos/Attitude/REAControl.m
space/SC/Demos/CAD/BuildMAP.m
```

CHAPTER 4

SPACECRAFT CONTROL TUTORIAL

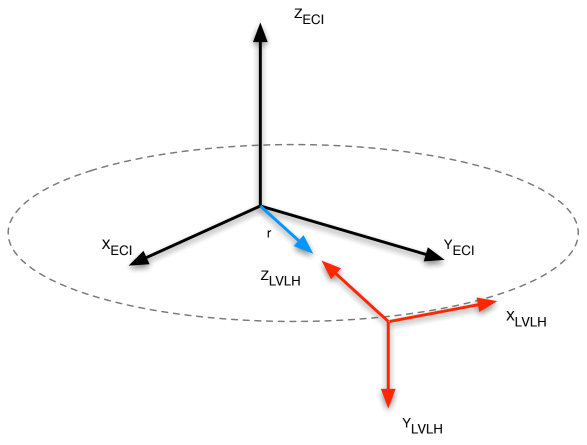
This tutorial is implemented in the MATLAB function `SCTTutorial`.

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

$$\begin{aligned} \dot{q} &= f(q,\omega) && (4.1) \\ T &= I\dot{\omega} + \omega^{\times} I\omega && (4.2) \end{aligned}$$

where q is the spacecraft's attitude quaternion, ω 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.

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, ω_0.

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

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

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

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

$$x_{k+1} = Ax_k + Bu_k \tag{4.5}$$
$$y_k = Cx_k + Du_k \tag{4.6}$$

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

$$y = K_p u + \frac{K_r s}{s + \omega_R} u + K_i \frac{u}{s} \tag{4.7}$$

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. Well 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

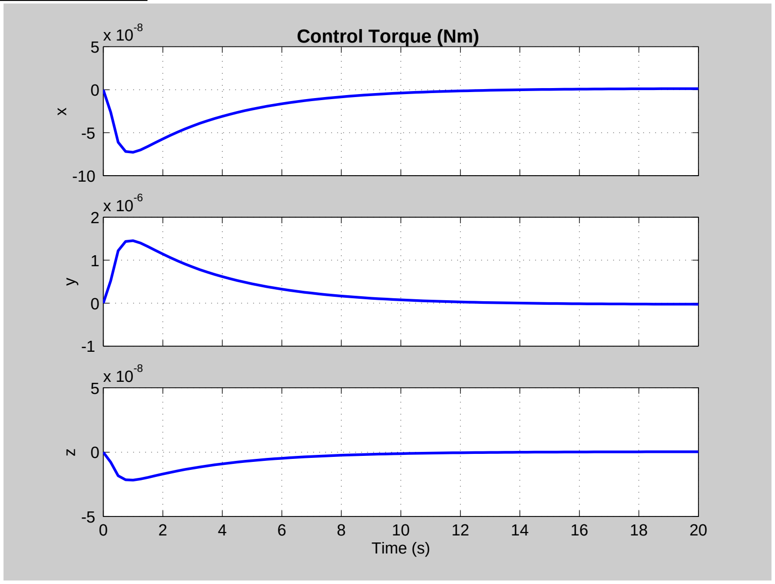
```
    accel(1) =          c*xRoll  + d*angleError(1);
    xRoll    = xRoll  + a*xRoll  + b*angleError(1);

    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);
```

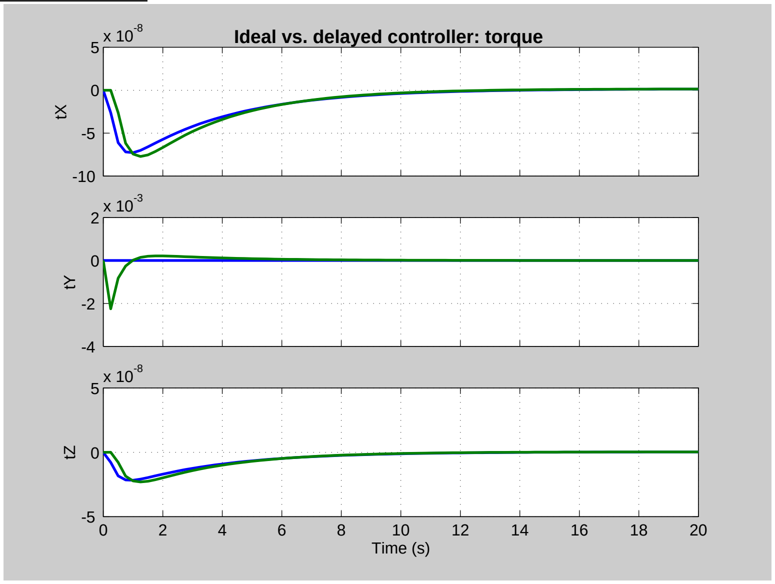
To test the controller, we give each axis an initial disturbance torque on the order of 1x10^{-6} Nm. The disturbance torque tDist is nonzero only for the first step. The resulting control torques are shown in Figure 4.2.

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 following 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!

CHAPTER 5

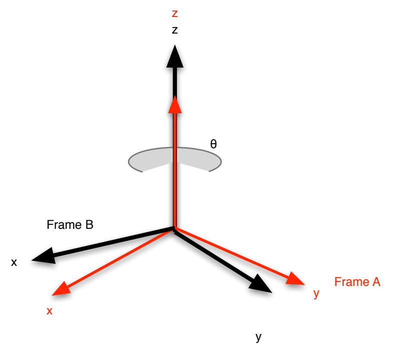
COORDINATE TRANSFORMATIONS

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 θ between their x and y axes.

Figure 5.1: Frames A and B



```
uA    = [1;0;0];
theta = pi/6;
m = [ cos(theta),sin(theta),0;...
     -sin(theta),cos(theta),0;...
      0,0,1];
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 the previous page 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

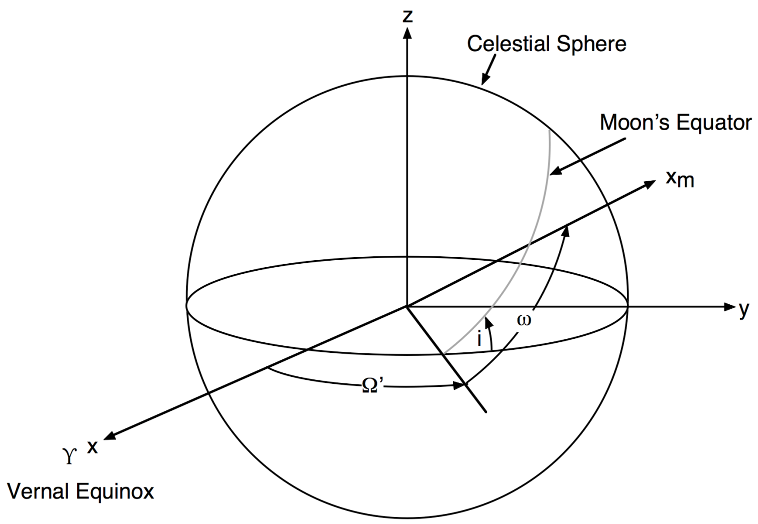
The toolbox has functions for many common coordinate frames and representations, including

- 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. ϕ'_m is the Selenographic Latitude which is the acute angle measured normal to the Moon's equator between the

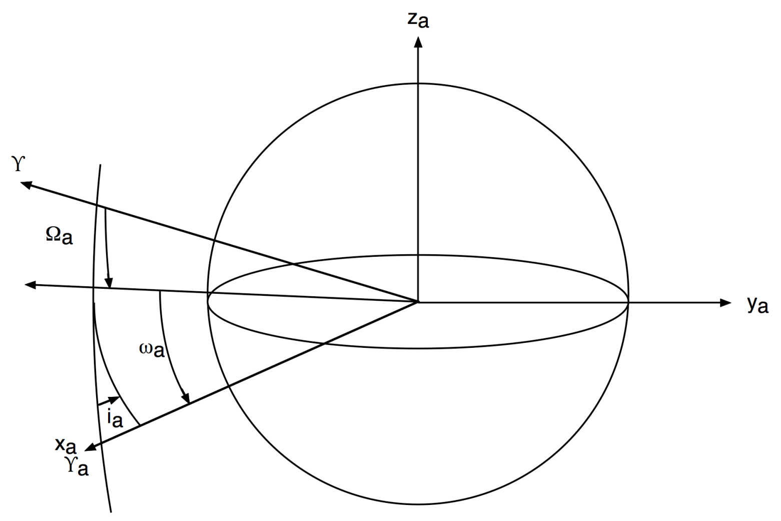
Figure 5.2: Selenographic to ECI frame



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. λ_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 next page. Ω_a, ω_a and i_a are the standard Euler rotations of the Mars vernal equinox, Υ_a with respect to the Earth's vernal equinox, Υ.

Figure 5.3: Areocentric frame



CHAPTER 6

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
```

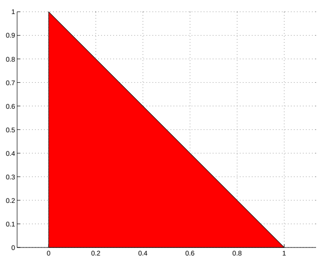
```
User Actions
    'initialize'
    'add␣units'
    'add␣name'
    'set␣aerodynamic␣model'
    'set␣aerodynamic␣model␣file'
    'add␣reci'
    'add␣veci'
    'add␣qecitobody'
    'add␣qlvlh'
    'add␣omega'
    'add␣mass'
    'set␣mass'
    'add␣body'
    'add␣component'
    'lock␣mass␣properties'
    '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,

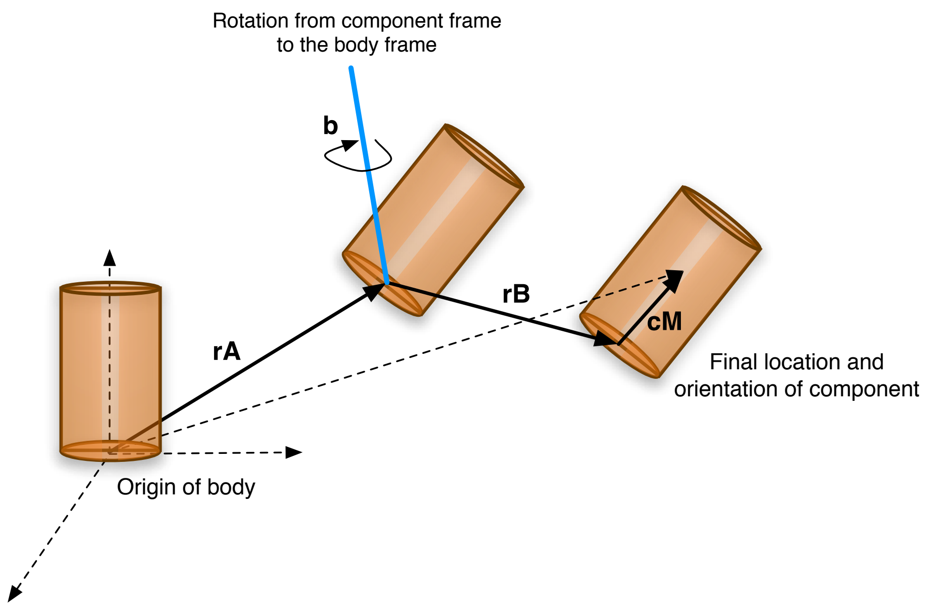
Example 6.1 Simple patch with one triangle

```
>> p = patch([0 1 0],[0 0 1],'r');
>> v = get(p,'vertices')
v =
     0     0
     1     0
     0     1

>> f = get(p,'faces')
f =
     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 the facing page 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, `rA` is in the body frame and `rB` 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 the next page 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 following 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

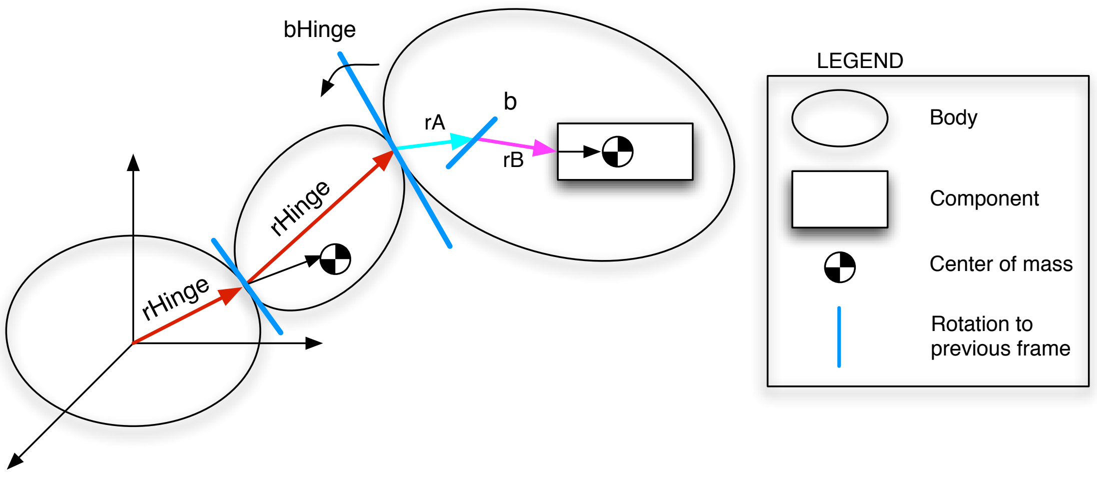
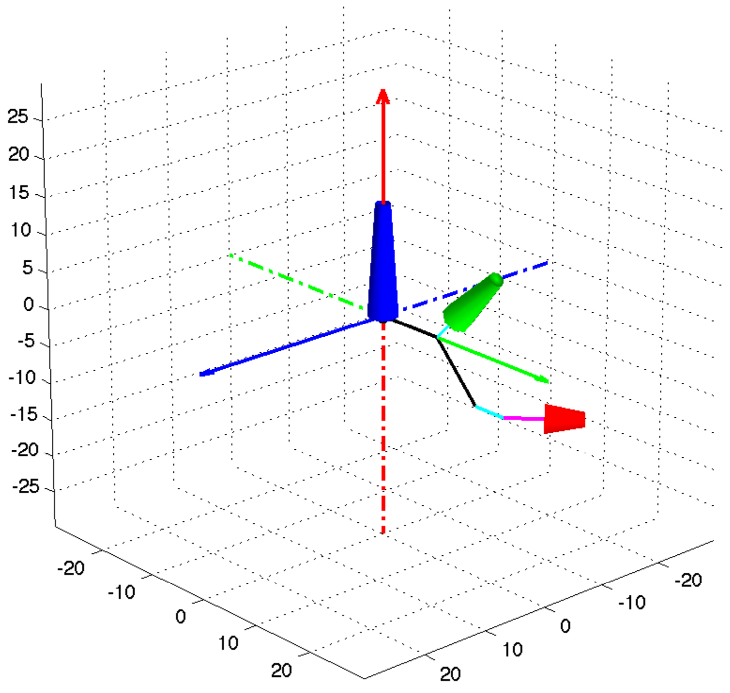


Figure 6.3: Cylinder demo of geometry and frames. Hinge vectors are in black, rA is cyan and rB is magenta. The axes are coded X (blue), Y (green), and Z (red).



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.

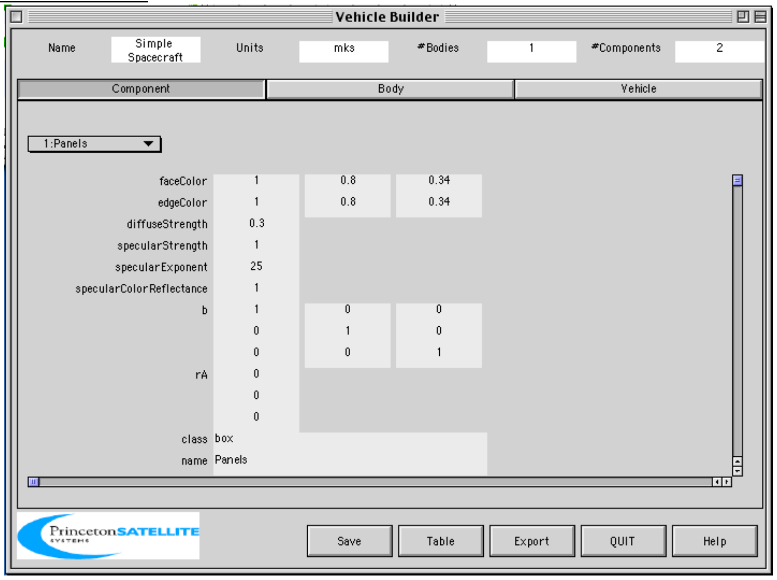
Listing 6.1: Simplest spacecraft CAD script *BuildSimpleSat.m*

```
%% A very simple spacecraft with one sensor.
% -------------------------------------------------------------------------
%  See also BuildCADModel, CreateBody, CreateComponent, FindDirectory,
%  SaveStructure
% -------------------------------------------------------------------------
%%
%--------------------------------------------------------------------------
%  Copyright (c) 2005, 2007 Princeton Satellite Systems, Inc.
%  All rights reserved.
%--------------------------------------------------------------------------

%% Initialize
%------------
BuildCADModel( 'initialize' );

% Add spacecraft properties
%--------------------------
BuildCADModel( 'set_name' , 'Simple_Spacecraft' );
BuildCADModel( 'set_units', 'mks'  );
BuildCADModel( 'set_qecitobody', [1;0;0;0]  );
BuildCADModel( 'set_reci', [7000;0;0]  );

% Create bodies first
%--------------------
m = CreateBody( 'make', 'name', 'Core' );
BuildCADModel('add_body', m );

% This creates the connections between the bodies
%------------------------------------------------
BuildCADModel( 'compute_paths' );

%% Add Components
%----------------

% Core
%------
m = CreateComponent( 'make', 'box', 'name', 'Panels', 'x', 1, 'y', 1, 'z', 1,...
                     'faceColor', 'gold_foil','inside',0,...
                     'rA', [0;0;0], 'mass', 20, 'body', 1 );
BuildCADModel( 'add_component', m );

% Star Camera
%-------------
m          = CreateComponent( 'make', 'star_camera', 'model', 'ct633', 'name', 'Camera',...
                              'rA', [0.4;0.4;0.4], 'body', 1, 'boresight',[1;0;0], ...
                              'faceColor', 'aluminum' );
BuildCADModel( 'add_component', m );

%% Save the model
%----------------
g = BuildCADModel( 'get_cad_model' );
BuildCADModel( 'show_spacecraft' );
d = FindDirectory('SCModels');
SaveStructure( g, fullfile(d,'SimpleSat') )

%--------------------------------------
% PSS internal file version information
```

BuildSimpleSat.m

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

Figure 6.4: Simple Sat



Name	Units	#Bodies	#Components
Simple Spacecraft	mks	1	2

1:Panels

faceColor	1	0.8	0.34
edgeColor	1	0.8	0.34
diffuseStrength	0.3		
specularStrength	1		
specularExponent	25		
specularColorReflectance	1		
b	1	0	0
	0	1	0
	0	0	1
rA	0		
	0		
	0		
class	box		
name	Panels		

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 (`qlvlh`, `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:

```
BuildCADModel( 'initialize' );
BuildCADModel( 'set_name' ,     'MySpacecraft' );
BuildCADModel( 'set_units',      'mks'   );
BuildCADModel( 'set_rECI' ,      rECI    );
BuildCADModel( 'set_vECI' ,      vECI    );
BuildCADModel( 'set_qLVLH',      qLVLH   );
BuildCADModel( 'set_qECIToBody', q       );
BuildCADModel( 'set_omega',      omega   );
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).

```
% The core
%---------
m = CreateBody( 'make', 'name', 'Core' );
BuildCADModel('add_body', m );
% Rotating bodies
%--------------
axis = [1 2 3];
for k = 1:3
  m = CreateBody( 'make', 'name', ['Boom ' num2str(k)], 'bHinge', struct( 'b', eye(3),'axis',
       axis(k) ),
  'previousBody', 1, 'rHinge', [0;0;0] );
  BuildCADModel('add_body', m );
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 `rA` is the displacement of the rotated component, i.e. *after* rotation to the body frame, and `rB` is a displacement *before* the rotation, so `rA` is defined in the body frame and `rB` 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 (`rA` and `rB`) 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.efficiency'
    'graphics.faceColor'
    'graphics.edgeColor'
    'graphics.diffuseStrength'
    'graphics.specularStrength'
    'graphics.specularColorReflectance'
    'graphics.specularExponent'
    'deviceInfo.scale'
    'deviceInfo.bias'
    'deviceInfo.randomWalk'
    'deviceInfo.angleRandomWalk'
    'deviceInfo.outputNoise1Sigma'
    '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`. f 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,...
               'outputNoise1Sigma',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
      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.

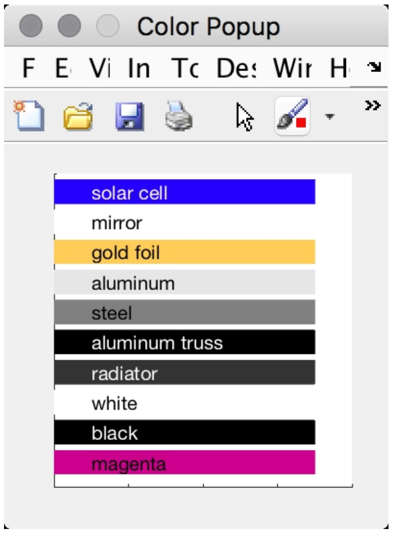
Perhaps the shortest component function call outside of the demo is this:

```
>> 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.

Figure 6.5: CreateComponent color display



6.2.7 Subsystems

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

```
BuildCADModel( 'add_subsystem', 'propulsion', {'ppt', 'thruster'} );
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 doesnt 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.

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

`BuildCADModel` searches for the strings in the cell array and associates them with the panel. If the string doesnt exist it is ignored. The result is a list of components stored with the panel name.

6.2.9 Viewing the CAD Model

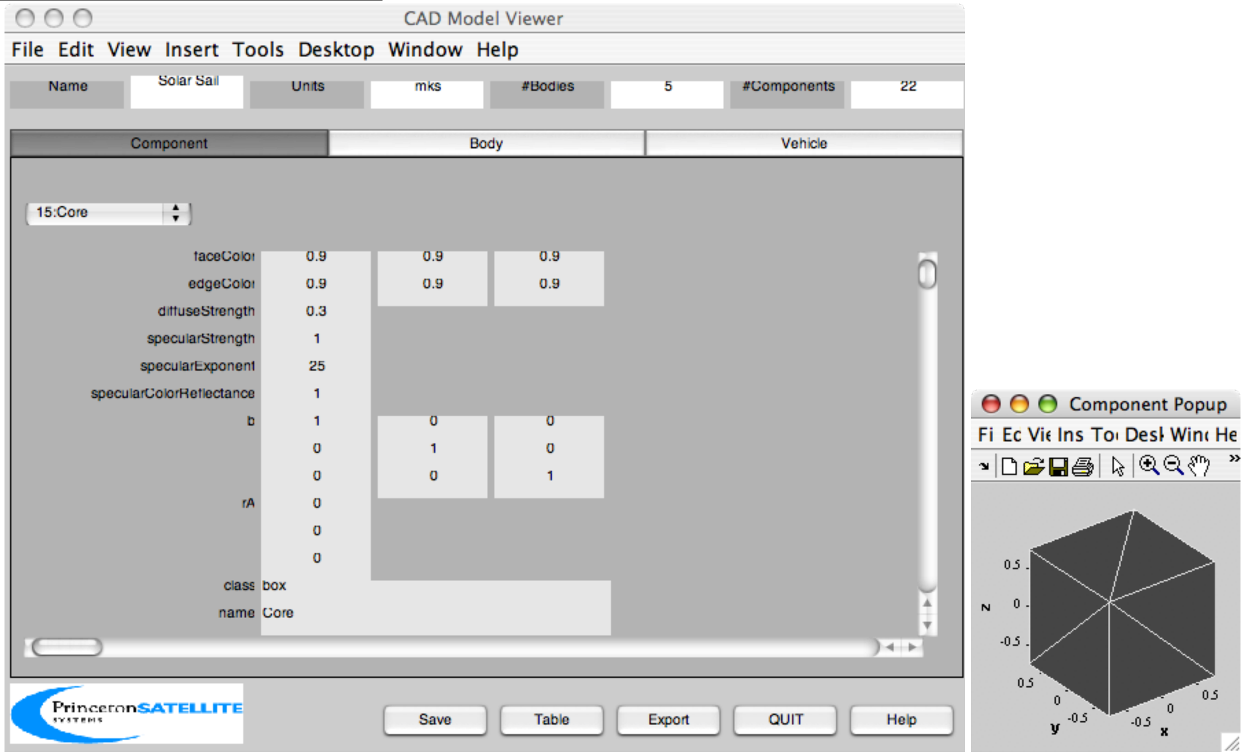
When your CAD script is done, execute the script. The GUI shown in Figure 6.6 on the next page will appear as in `BuildSimpleSat` in the beginning of this chapter. For this section we will view the results of `BuildSolarSail.m`, which builds a solar sail. This demo demonstrates both generic components 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, Solar Sail.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


CAD Model Viewer
File Edit View Insert Tools Desktop Window Help
Component
Body
Vehicle
15:Core
PrincetonSATELLITE SYSTEMS
Save
Table
Export
QUIT
Help
Component Popup
Fi Ec Vie Ins To Desl Wine He
0.5
0
-0.5
N
y
x

Name	Solar Sail	Units	mks	#Bodies	5	#Components	22

faceColor	0.9	0.9	0.9
edgeColor	0.9	0.9	0.9
diffuseStrength	0.3		
specularStrength	1		
specularExponent	25		
specularColorReflectance	1		
b	1	0	0
	0	1	0
	0	0	1
rA	0		
	0		
	0		
class	box		
name	Core		

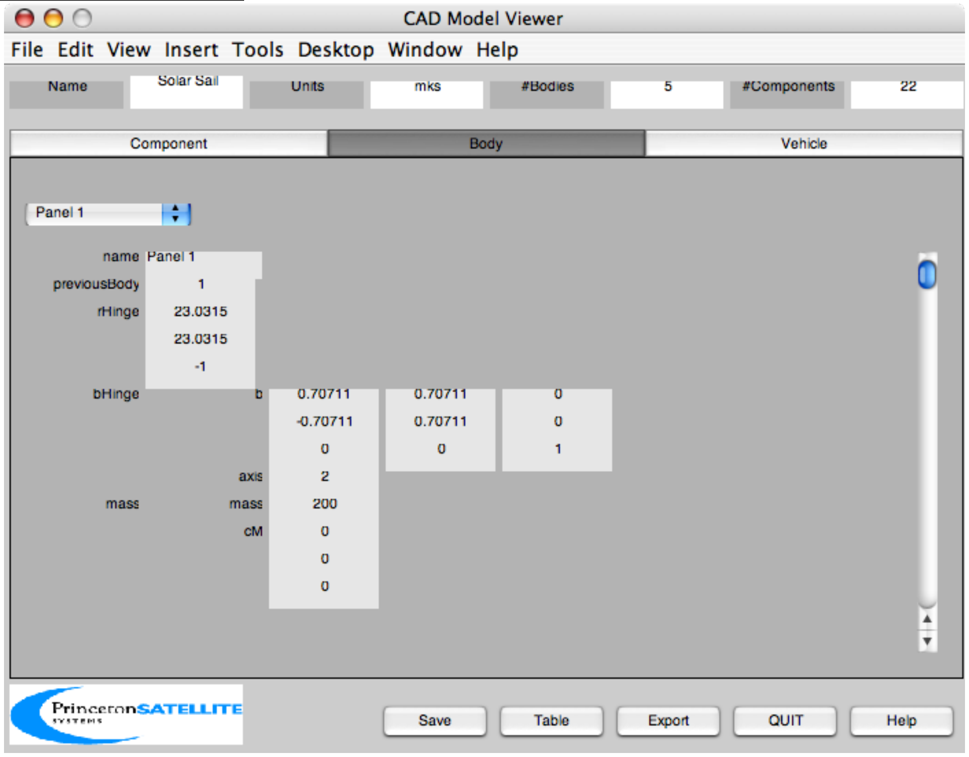
```
         Sail -Y 20.00       0.00          0.00       none generic
Bus Subsystem
         RWA X 14.30         0.00          0.00       none hr60
         RWA Y 14.30         0.00          0.00       none hr60
         RWA Z 14.30         0.00          0.00       none hr60
         South Array 0.03          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 the facing page. 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 next 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

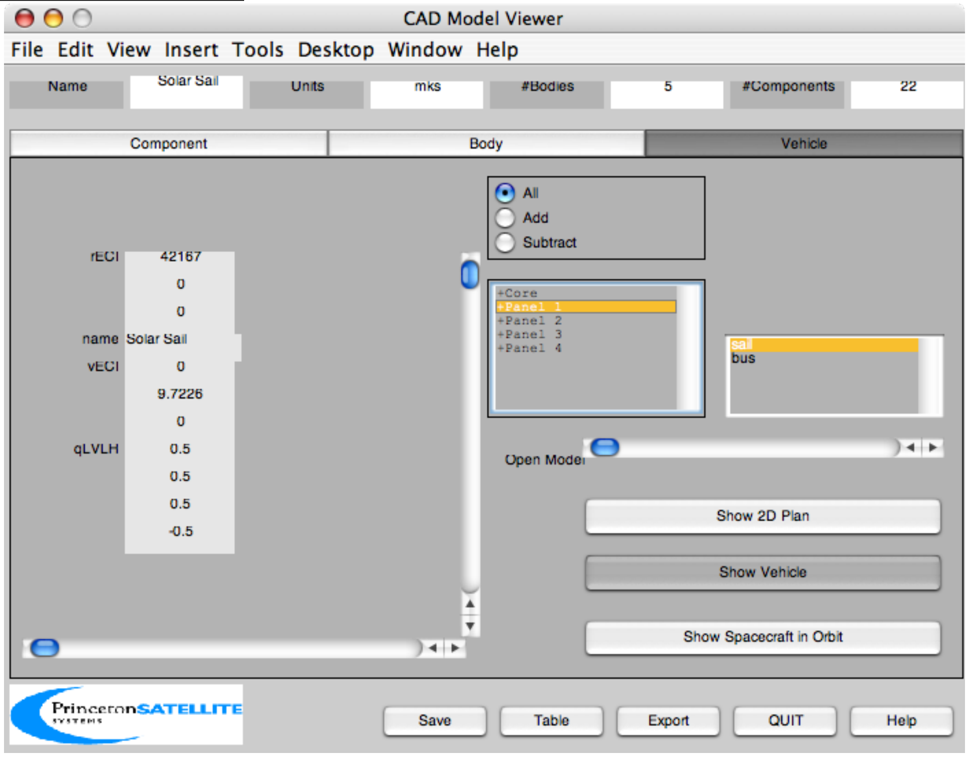

CAD Model Viewer
File Edit View Insert Tools Desktop Window Help
Component
Body
Vehicle
Save
Table
Export
QUIT
Help
PrincetonSATELLITE SYSTEMS

Name	Solar Sail	Units	mks	#Bodies	5	#Components	22

Panel 1

name	Panel 1			
previousBody	1			
rHinge	23.0315			
	23.0315			
	-1			
bHinge	b	0.70711	0.70711	0
		-0.70711	0.70711	0
		0	0	1
	axis	2		
mass	mass	200		
	cM	0		
		0		
		0		

Figure 6.8: Vehicle tab with Show Vehicle pushed



Name	Solar Sail	Units	mks	#Bodies	5	#Components	22

[✓ All] Add Subtract

```
+Core
+Panel 1
+Panel 2
+Panel 3
+Panel 4
```

sail
bus

rECI	42167
	0
	0
name	Solar Sail
vECI	0
	9.7226
	0
qLVLH	0.5
	0.5
	0.5
	-0.5

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

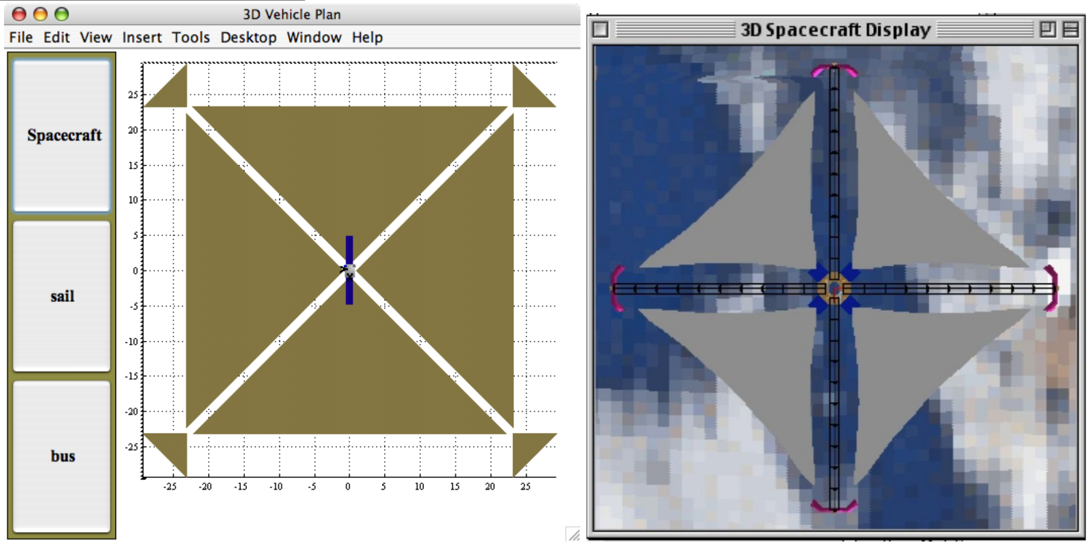
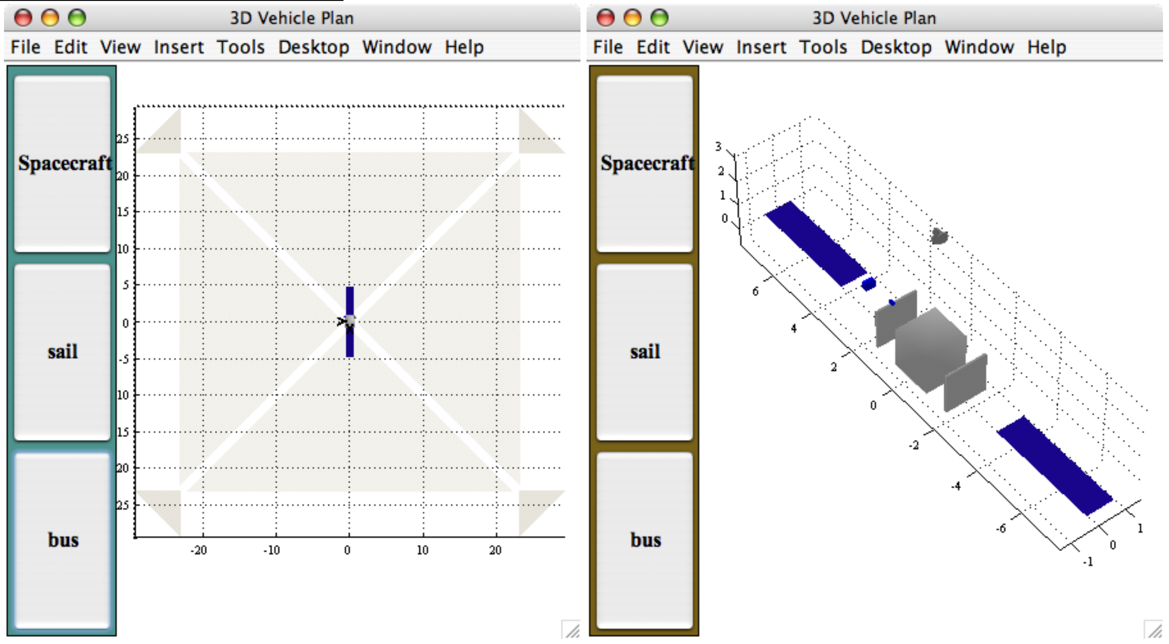


Figure 6.10: Subsystem Transparency and Exploded views

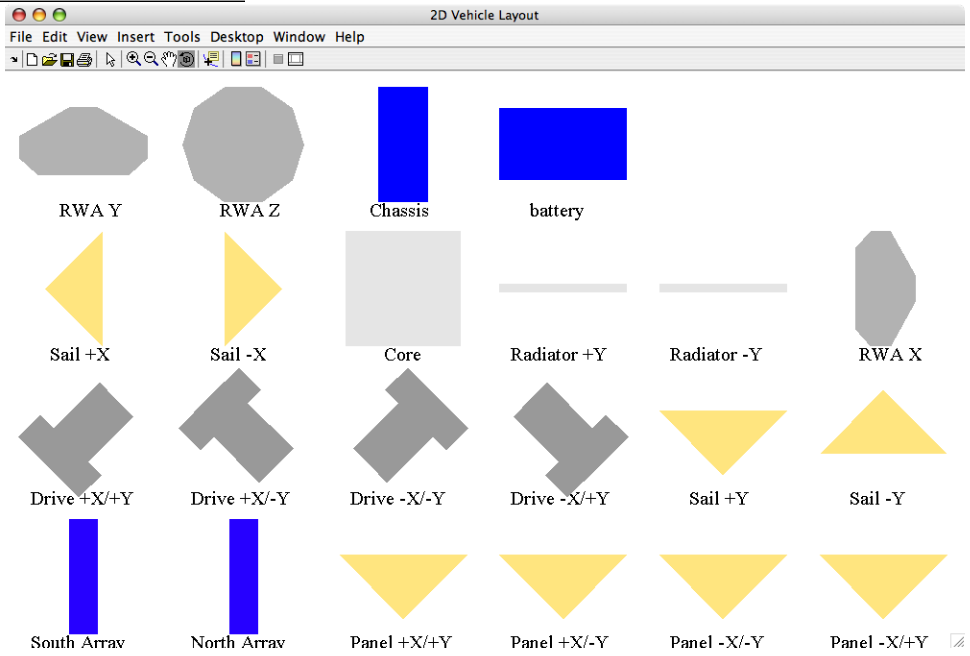

3D Vehicle Plan
File Edit View Insert Tools Desktop Window Help
Spacecraft
sail
bus
3D Vehicle Plan
File Edit View Insert Tools Desktop Window Help
Spacecraft
sail
bus

```
BuildCADModel( 'set_rECI' ,      rECI    );
BuildCADModel( 'set_vECI' ,      vECI    );
BuildCADModel( 'set_qLVLH',      qLVLH   );
BuildCADModel( 'set_qECIToBody', q       );
BuildCADModel( 'set_omega',      omega   );
```

An example view of another sail model in orbit is shown on the right in Figure 6.9 on the facing page. 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.

```
BuildCADModel( 'add_subassembly', 'Bus.mat', gimballedBoomBody, 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 SCData folder as examples, including models of Hubble, Cassini, and the Huygens probe. Some are more fanciful, for instance,

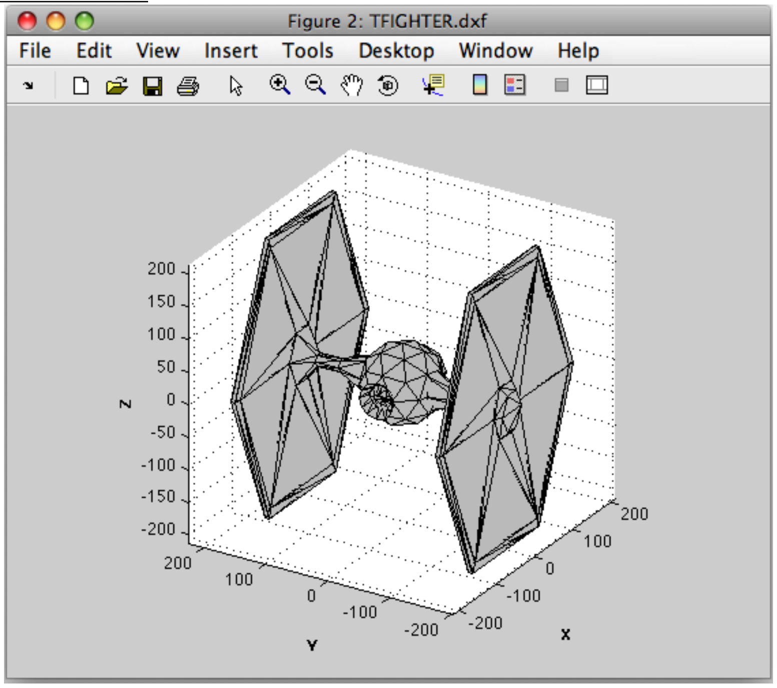
```
>> LoadCAD('TFIGHTER.dxf')
```

draws a picture of a tie fighter, shown in Figure 6.12 on the next 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:

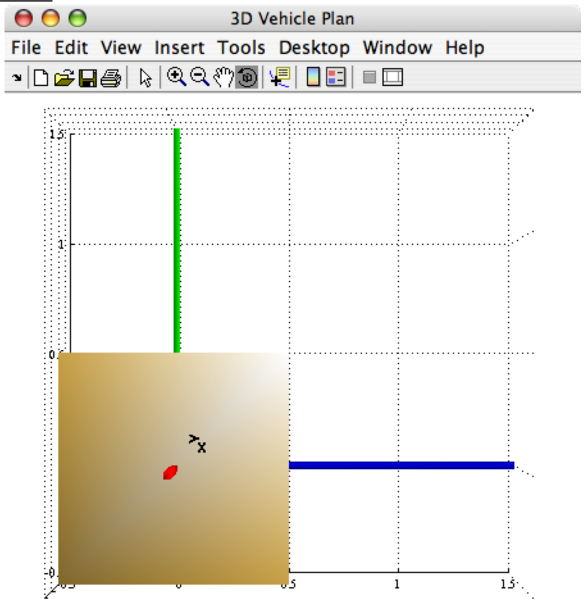
- ShowCAD
- DrawCAD
- DrawSC
- DrawSCPlanPlugIn
- DrawSCPlugIn

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. 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

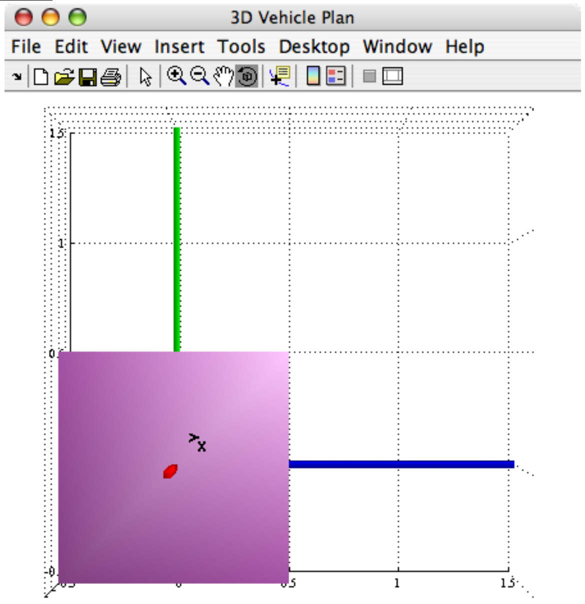
```
>> g.component(1)
ans =
                  faceColor: [1 8.000000000000000e-01 3.400000000000000e-01]
                  edgeColor: [1 8.000000000000000e-01 3.400000000000000e-01]
            diffuseStrength: 3.000000000000000e-01
           specularStrength: 1
           specularExponent: 25
    specularColorReflectance: 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, the 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 the next page.

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

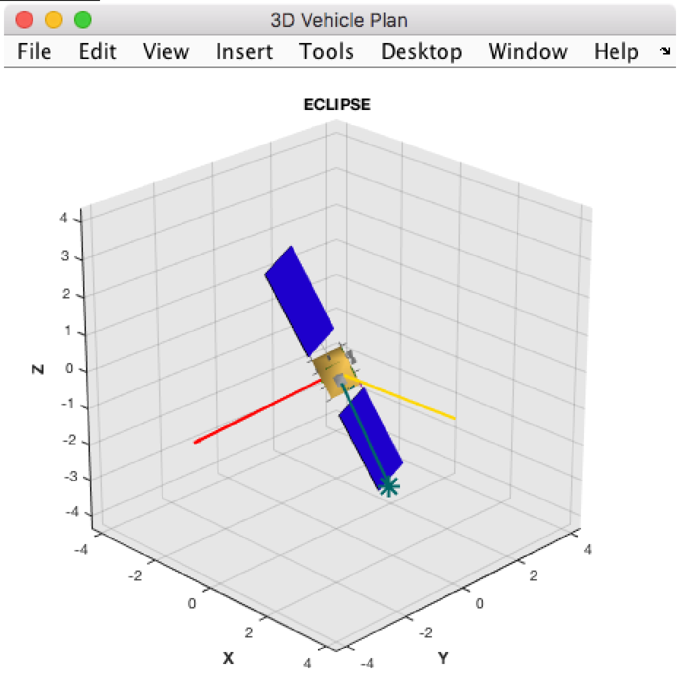


`DrawSCPlanPlugin` can display vectors, such as sun and nadir vectors, to aid in visualization. For an example this see `DisturbanceBatchDemo`. The code used to update the vectors is

```
>> 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*

```
%% Simulate the space shuttle model.
% We rotate the shuttle bay doors and the attached robot arm using the body
% hinge structure of the CAD model. The shuttle model is loaded from a mat-file.
% This script shows how to interact with DrawSCPlugIn in a simulation loop.
%  ------------------------------------------------------------------------
%  See also DrawSCPlugIn, QLVLH, QPose, StopGUI, RK4,
%  JD2000, El2RV
%  ------------------------------------------------------------------------
%%
%--------------------------------------------------------------------------
%   Copyright 1999 Princeton Satellite Systems, Inc. All rights reserved.
%--------------------------------------------------------------------------

%% Constants
%----------
degToRad = pi/180;

%% Global for the time GUI
%------------------------
global simulationAction
simulationAction = '␣';

g = load('ShuttleModel');

%% Initialize the orbits
%----------------------
[r1,v1] = El2RV( [7000 61*degToRad 0 0 0 0]);
x1      = [r1;v1];
t       = 0;
jD      = JD2000;

%% Initialize the 3D window
%-------------------------
g = SetCADQuaternion( g, QLVLH( r1, v1 ) );
g = SetCADState( g, r1, v1 );
g.name      = 'Space␣Shuttle␣Orbiter';
tag3DWindow = DrawSCPlugIn( 'initialize', g, [], [], 'Earth', jD );

dTSim                = 1;

%% Generate the two orbits using numerical integration
%----------------------------------------------------
doorAngle = 0;
rMSAngle  = 0;
StopGUI( 'Space␣Shuttle' );

rMSAxis = [3 2 2 2 2];

while t<120

  % Transformation matrices
  %------------------------
  g = SetCADQuaternion( g, QLVLH( x1(1:3), x1(4:6) ) );
  g = SetCADState( g, x1(1:3), x1(4:6) );
  if( doorAngle > -139 )
    doorAngle = doorAngle - 1;
  elseif( rMSAngle < 45 )
    rMSAngle = rMSAngle + 1;
  end
  g = SetCADRotation( g, 'body', 2, 'axis', 1, 'angle', doorAngle*pi/180 );
  g = SetCADRotation( g, 'body', 3, 'axis', 1, 'angle', -doorAngle*pi/180 );
  for j = 1:5
    g = SetCADRotation( g, 'body', j+3, 'axis', rMSAxis(j), 'angle', -rMSAngle*pi/180 );
  end

  DrawSCPlugIn( 'update␣spacecraft', tag3DWindow, g, jD );

  % Propagate the orbits
  %---------------------
  x1 = RK4( @FOrbCart, x1, dTSim, t, [0;0;0] );

  % Update the time
  %----------------
  t  =  t + dTSim;
  jD = jD + dTSim/86400;

  % Time control
  %-------------
  switch simulationAction
    case 'pause'
      pause
      simulationAction = '␣';
    case 'stop'
      return;
    case 'plot'
      break;
```

ShuttleSim.m

CHAPTER 7

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,

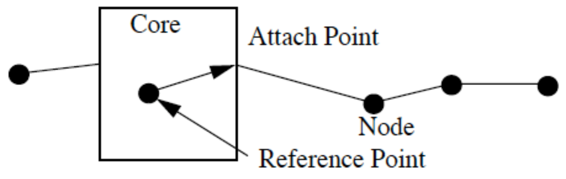
```
xDot = [QIToBDot( x(1:4), x(5:7)); RBModel( x(5:7), inertia, torque)];
```

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 the following page. 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

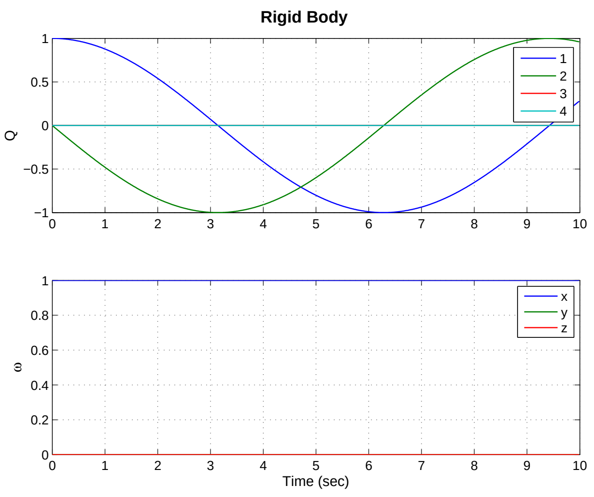
The wire model simulates a spacecraft with deployable wires as is illustrated in Figure 7.1. Two models are included.

Figure 7.1: Wire model



Example 7.1 Rigid Body Attitude Simulation

```
dT            = 0.01;
nSimSteps     = 1000;
inertia       = [3000,0,0;0,1000,0;0,0,1000];
torque        = [0;0;0];
xPlot         = zeros(7,nSimSteps);
tPlot         = zeros(1,nSimSteps);
x             = [ [1;0;0;0]; [1;0;0] ];
t             = 0;
for k = 1:nSimSteps,
  xPlot(:,k)  = x;
  tPlot(k)    = t;
  zIO(1:3,4)  = [0;0;0];
  x           = RK4('ASim',x,dT,t, inertia,
      torque);
  t           = t + dT;
end
Plot2D(tPlot,xPlot,'Time␣(sec)',{'Q';'\omega'
    },...
       'Rigid␣Body','lin',{[1:4],[5:7]});
legend('x','y','z')
subplot(2,1,1)
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]
%--------------------------------
el            = [7000;0;0;0;0;0];
muEarth       = 3.98600436e5;
% Initial rigid body state
%-------------------------
[rECI, vECI]  = El2RV( el );
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
```

WireSimG.m

The following listing shows the plotting code.

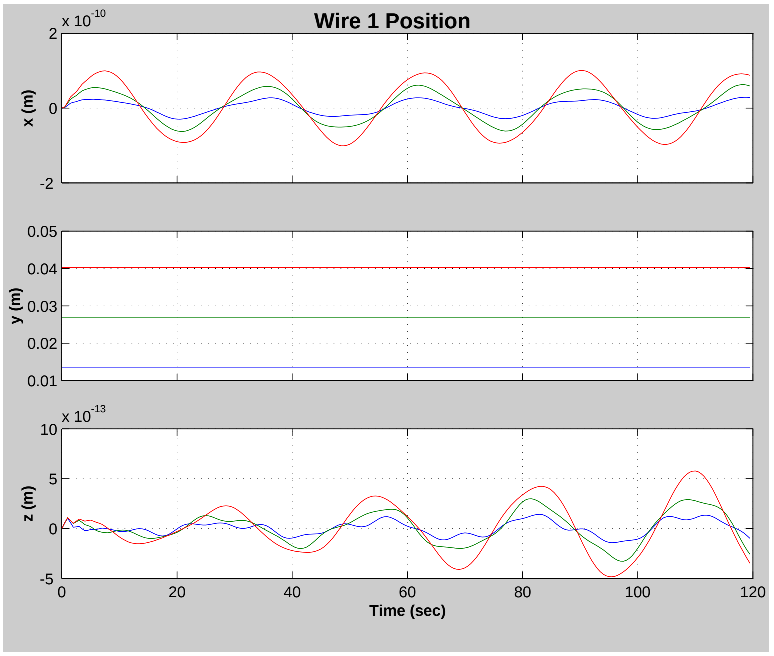
Listing 7.9: Wire Simulation: Plotting *WireSimG.m*

```
% 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.

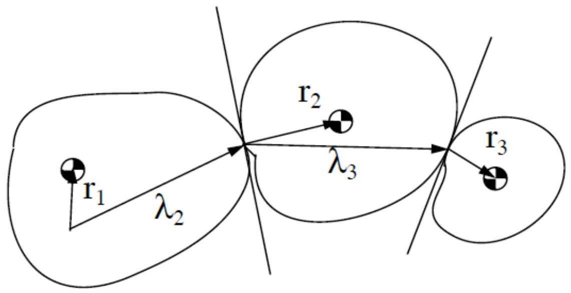
Figure 7.2: Wire simulation: Results



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. TreePrnt 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
%-------------------
TreePrnt( 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
```

TreeSim.m

Plot the results.

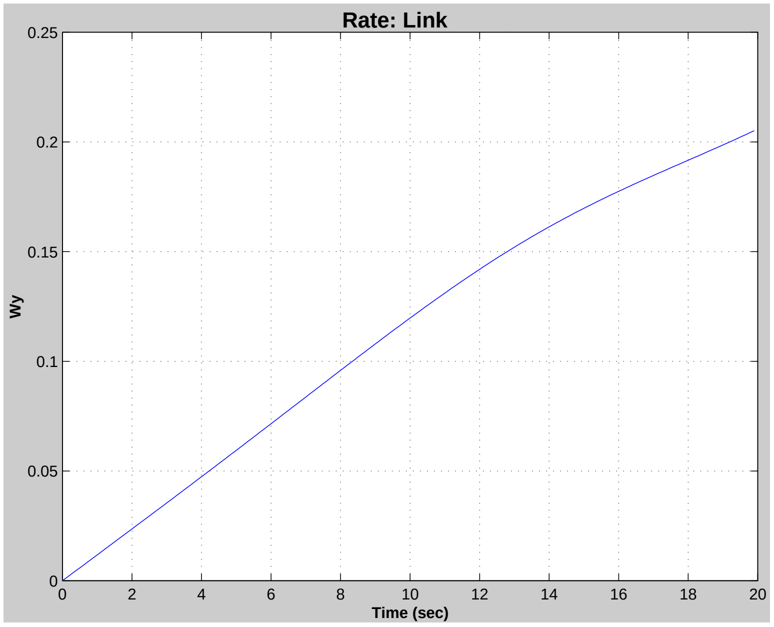
Listing 7.14: Tree Simulation: Plotting. *TreeSim.m*

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

TreeSim.m

An example plot is shown in Figure 7.4.

Figure 7.4: Tree Simulation: One of several plots.



CHAPTER 8

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[1], moon, and sun. For example, the following functions are available in Ephem:

- `MoonV1` [2]
- `MoonV2` [3]
- `Planets, PlanetPosition` [4]
- `SolarSys`
- `SunV1` [5]
- `SunV2` [6]

[1]Explanatory Supplement to the Astronomical Almanac (1992.) Table 5.8.1. p. 316.
[2]The 1993 Astronomical Almanac, p. D46.
[3]Montenbruck, O., Pfleger, T., Astronomy on the Personal Computer, Springer-Verlag, Berlin, 1991, pp. 103-111.
[4]Explanatory Supplement to the Astronomical Almanac, 1992, p. 316.
[5] The 1993 Astronomical Almanac, p. C24.
[6]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 the following page: 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
```

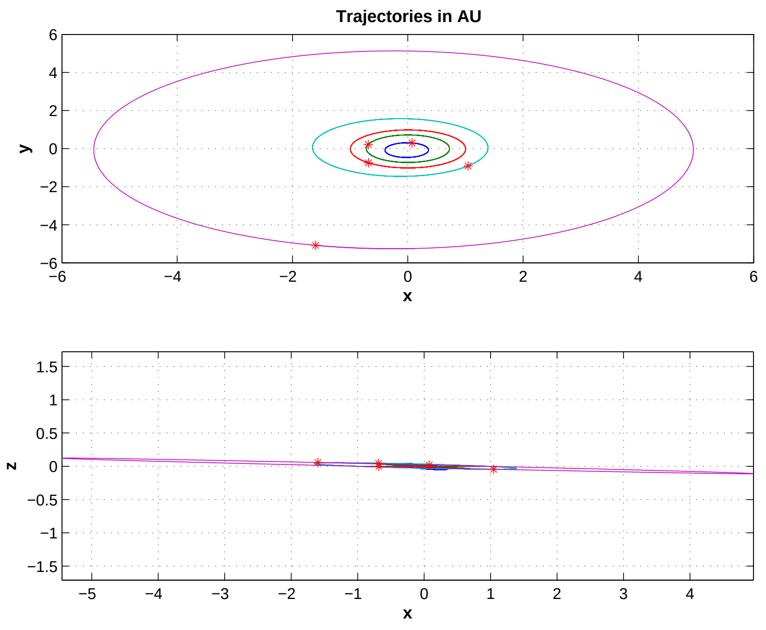
Table 8.1: Target Numbering Convention

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.

CHAPTER 9

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.

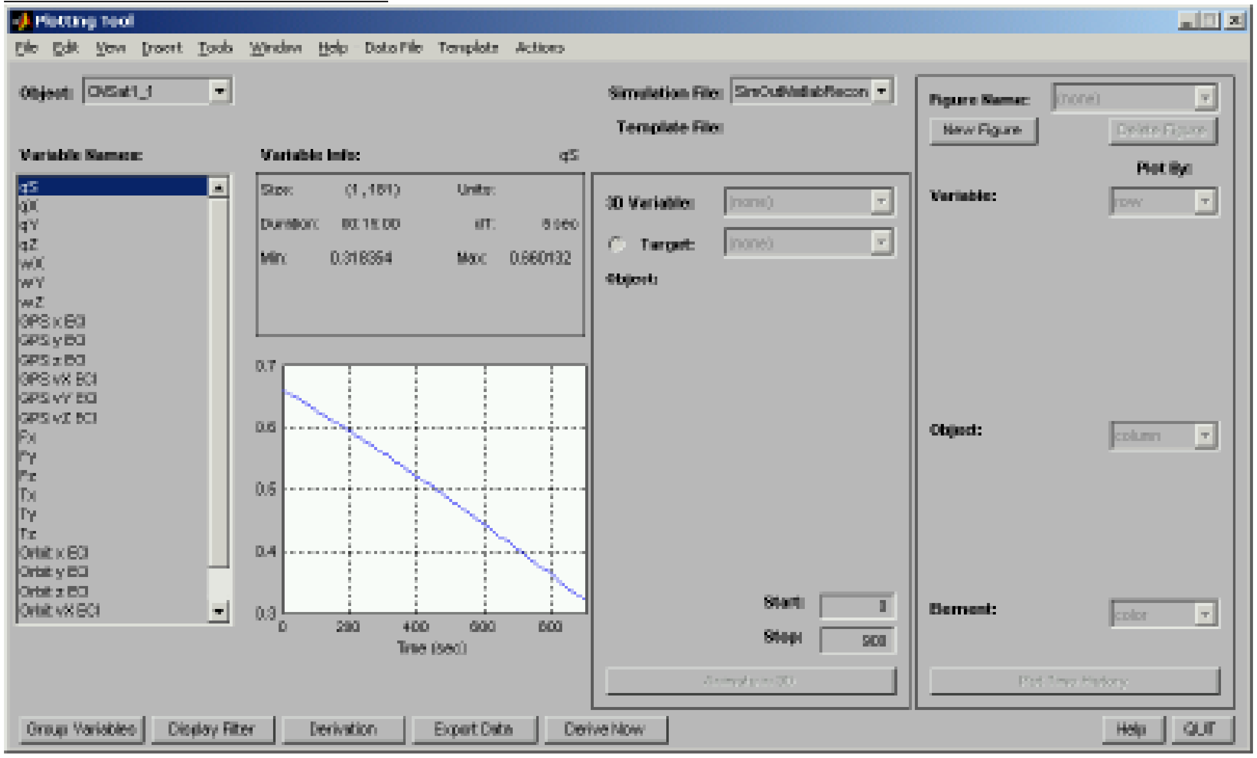
```
>> PlottingTool
```

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



Object: CNSat1_1
Simulation File: SimOutMatlabRecon
Variable Info: qS
Size: (1 ,181) Units:
Duration: 00:15:00 dT: 5 sec
Min: 0.318364 Max: 0.660132
3D Variable: (none)
Target: (none)
Start: 1
Stop: 900
Figure Name: (none)
Variable: row
Object: column
Element: color

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:

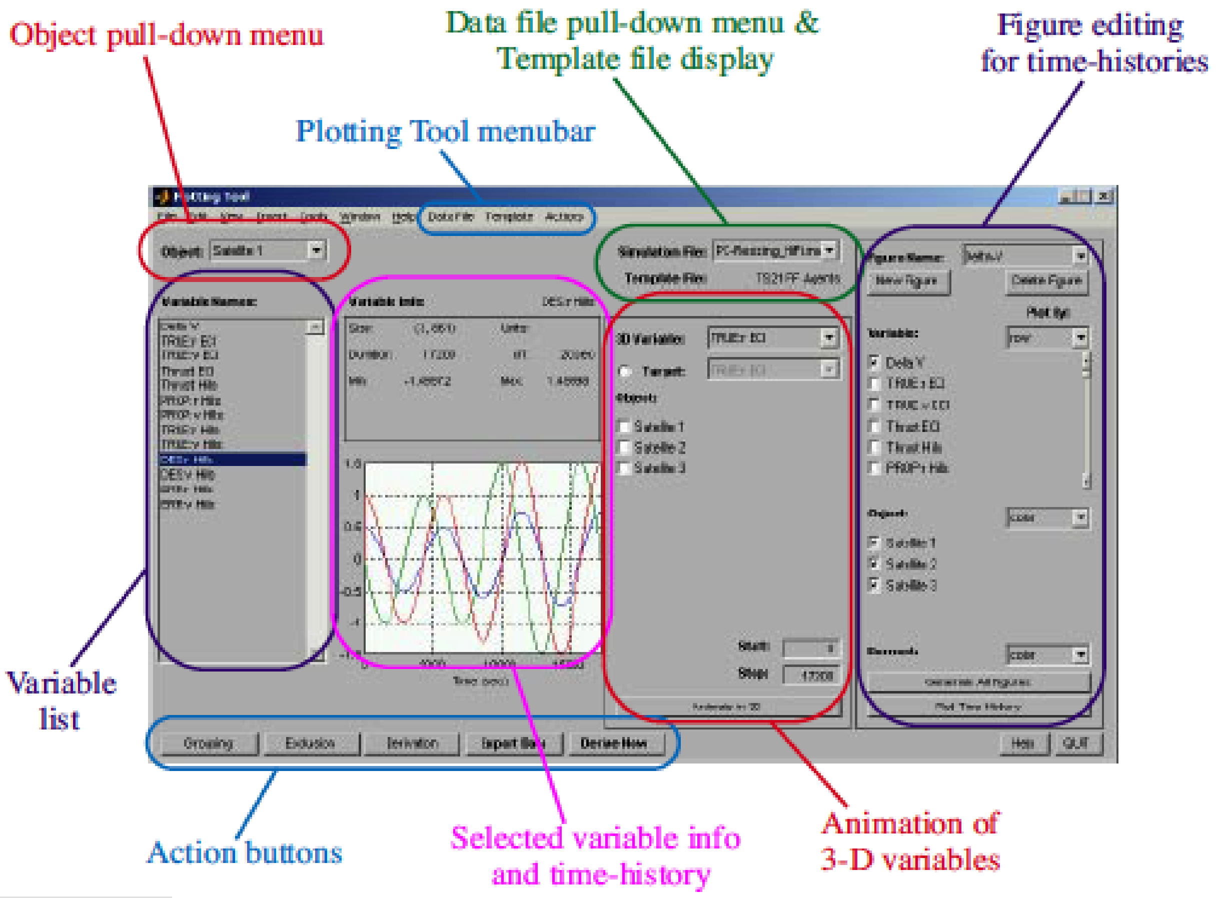
- Grouping and organizing variables
- Selecting which variables should be displayed to the screen
- Generating time-history plots
- Animating 3-dimensional variables
- Deriving new data from the raw data
- Applying predefined templates to perform any of the above actions automatically

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


Object pull-down menu
Data file pull-down menu & Template file display
Figure editing for time-histories
Plotting Tool menubar
Variable list
Action buttons
Selected variable info and time-history
Animation of 3-D variables

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 the next page.

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

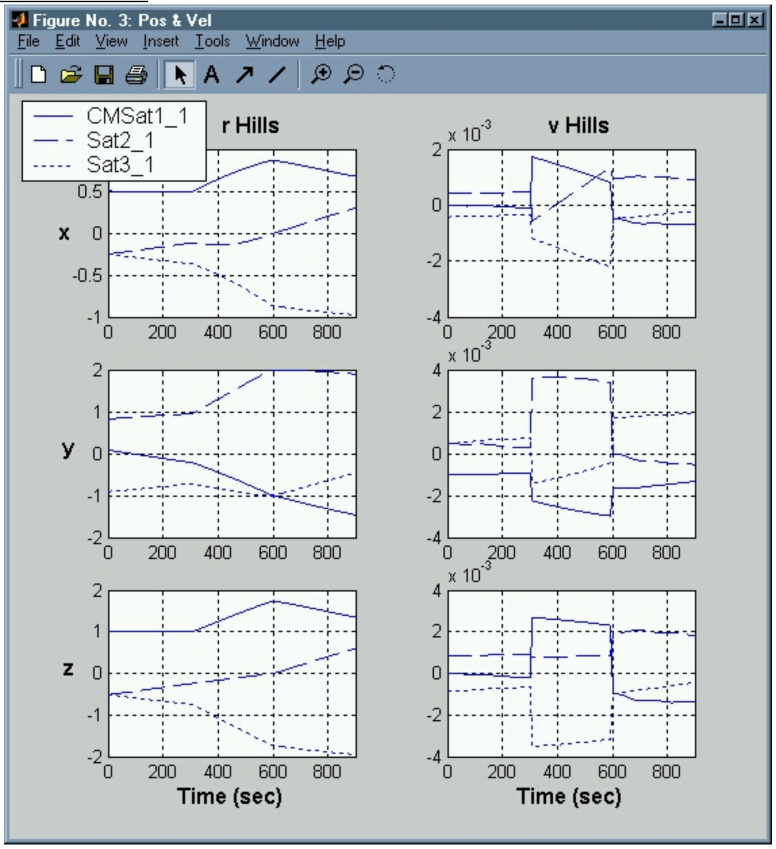
The right hand side of the GUI is used for generating figures. First, click the New Figure button and pick a name for the new figure. You can choose the variables and satellites you wish to plot from the checkbox lists, and indicate how they are distinguished with the pull-down menus under the words, 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 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 the facing page provides a snapshot of the Animation GUI generated with the relative Hills position derived from the `SimOutMatlabRecon.mat` data file.

9.9 Templates

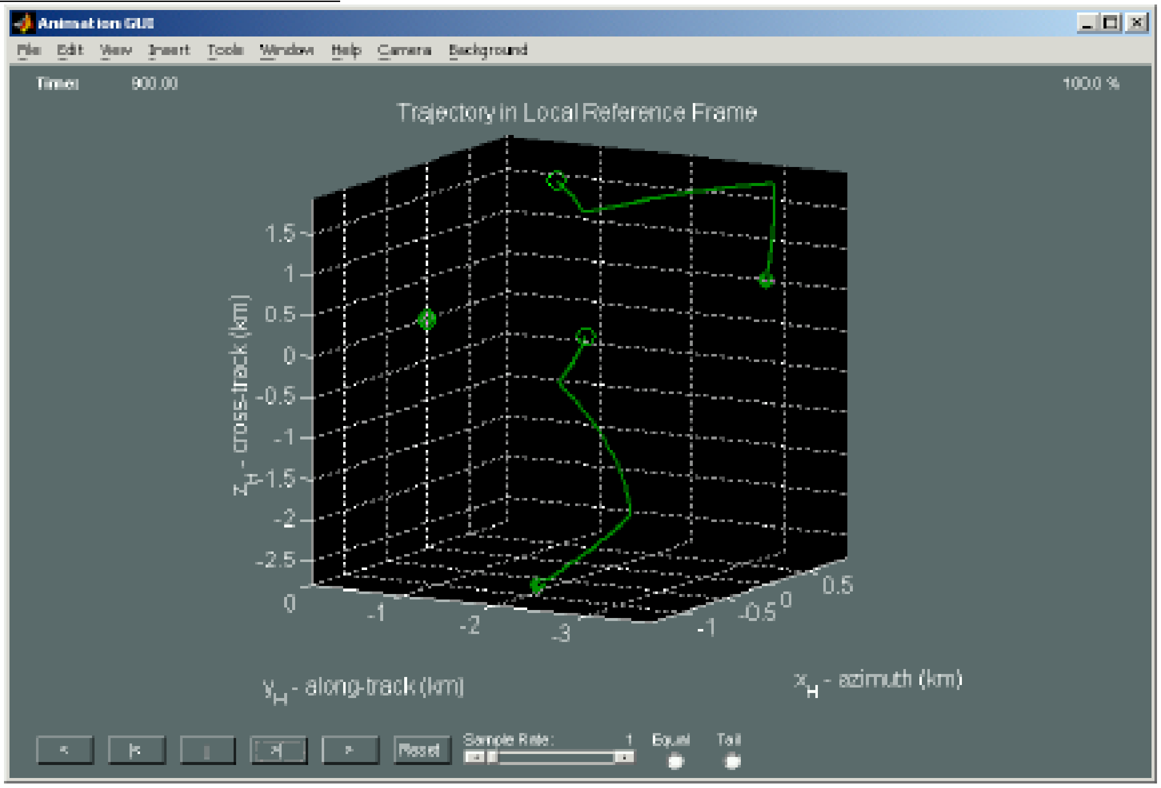
Once a data file is loaded into the Plotting Tool, the user may manipulate the data in a number of different ways. Variables may be grouped or excluded, new variables derived, and figures generated. All of these settings may be saved to a template file. The template may be edited over time, and applied 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, ‘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
`data`	{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 ‘groups’ field or by using the Plotting Tool after reading in the data.
`dataNames`	{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 ‘el_1’, ‘el_2’, etc.
`dataUnits`	{1 x nD} cell array of strings	Each string describes the units of the corresponding data element. Optional.
`excluded`	{1 x n} cell array of strings	Each string is the dataName of an element that is not to be displayed in the variable list. Optional.
`groups`	(:) array of data structures with fields ‘name’ and ‘elements’	Each element is a data structure corresponding to a new grouping of individual data elements. The ’name’ field is the name of the group as a string. The ’elements’ field is a cell array of strings that contains the dataName of each data element that is to be included in the group. Optional.
`jDEpoch`	scalar	Simulation start time as Julian date. Optional.
`objNames`	{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 ‘Obj_1’, ‘Obj_2’, etc.
`time`	(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*

```
Small Agile Satellite Simulation Output
Epoch 2.452994833e+06
nObjects 1
Target
// 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.

CHAPTER 10

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 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

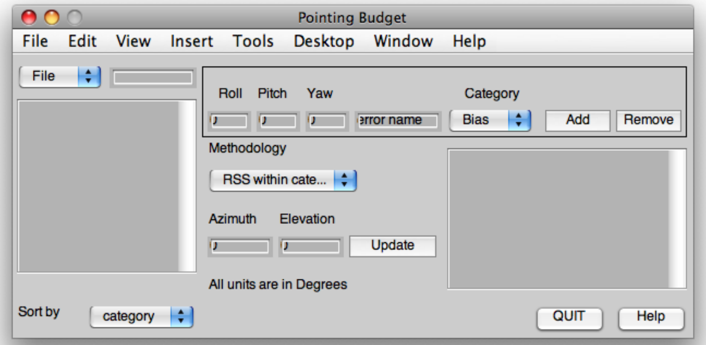
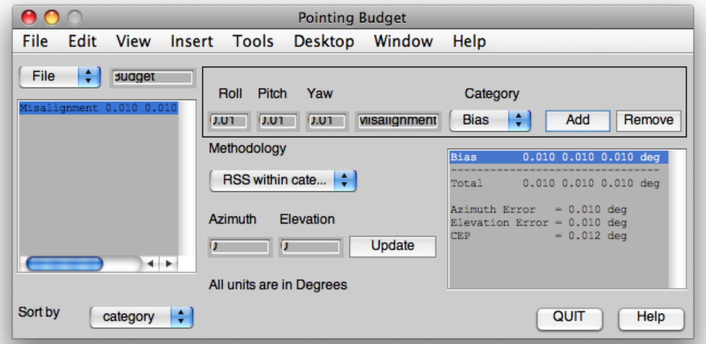


Figure 10.2 on the facing page shows the GUI after the first error has been entered. The component appears in both scrollable text windows. The right one also shows the resulting pointing error.

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

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

Figure 10.2: Entering the first error



Roll 0.01, Pitch 0.01, Yaw 0.01, Misalignment, Category: Bias

Azimuth 0, Elevation 0

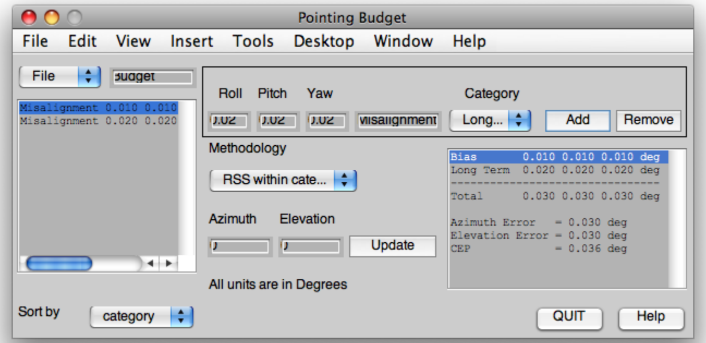
All units are in Degrees

Sort by: category

```
Misalignment 0.010 0.010
```

```
Bias         0.010 0.010 0.010 deg
-----------------------------------
Total        0.010 0.010 0.010 deg

Azimuth Error   = 0.010 deg
Elevation Error = 0.010 deg
CEP             = 0.012 deg
```

Figure 10.3: Entering a second error



Roll 0.02, Pitch 0.02, Yaw 0.02, Misalignment, Category: Long...

Azimuth 0, Elevation 0

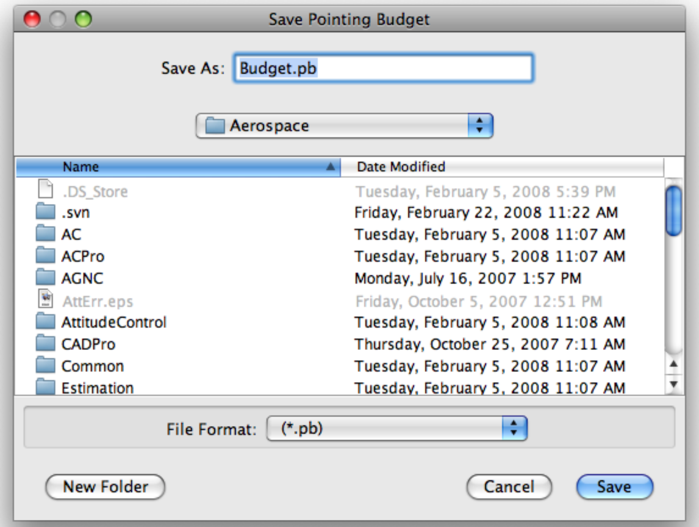
All units are in Degrees

Sort by: category

```
Misalignment 0.010 0.010
Misalignment 0.020 0.020
```

```
Bias         0.010 0.010 0.010 deg
Long Term    0.020 0.020 0.020 deg
-----------------------------------
Total        0.030 0.030 0.030 deg

Azimuth Error   = 0.030 deg
Elevation Error = 0.030 deg
CEP             = 0.036 deg
```

Figure 10.4: Save window



Save As: Budget.pb

Aerospace

Name	Date Modified
.DS_Store	Tuesday, February 5, 2008 5:39 PM
.svn	Friday, February 22, 2008 11:22 AM
AC	Tuesday, February 5, 2008 11:07 AM
ACPro	Tuesday, February 5, 2008 11:07 AM
AGNC	Monday, July 16, 2007 1:57 PM
AttErr.eps	Friday, October 5, 2007 12:51 PM
AttitudeControl	Tuesday, February 5, 2008 11:08 AM
CADPro	Thursday, October 25, 2007 7:11 AM
Common	Tuesday, February 5, 2008 11:07 AM
Estimation	Tuesday, February 5, 2008 11:07 AM

File Format: (*.pb)

CHAPTER 11

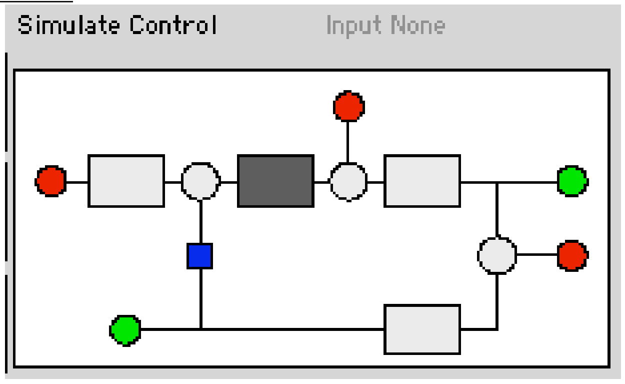
DESIGNING CONTROLLERS

This chapter shows how to design controllers using the `ControlDesignGUI`. The three major methodologies discussed are: Linear Quadratic, Eigenstructure Assignment, and Single-Input-Single-Output. This section focuses on how to use the Control Designer GUI.

11.1 Using the block diagram

The block diagram from the control designer GUI is shown in the following figure.

Figure 11.1: Block diagram



When you select a block, all operations (including all of the simulation buttons, loading and saving), apply only to that block. To work with the entire diagram click the highlighted block so that none are highlighted. The blue box opens and closes the control loops. When it is blue (the default) the system is closed. To open the loops, click the box, and it will turn white.

The red circles are inputs and the green are outputs. When you are working with the entire system you can select the input and output points by clicking on the red and green circles. The red circle on the left is the command input, the one on the top is the disturbance input and the one on the right is the noise input. The green output on the right is the state output and the green output on the left is the measurement output.

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 the next page.

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.

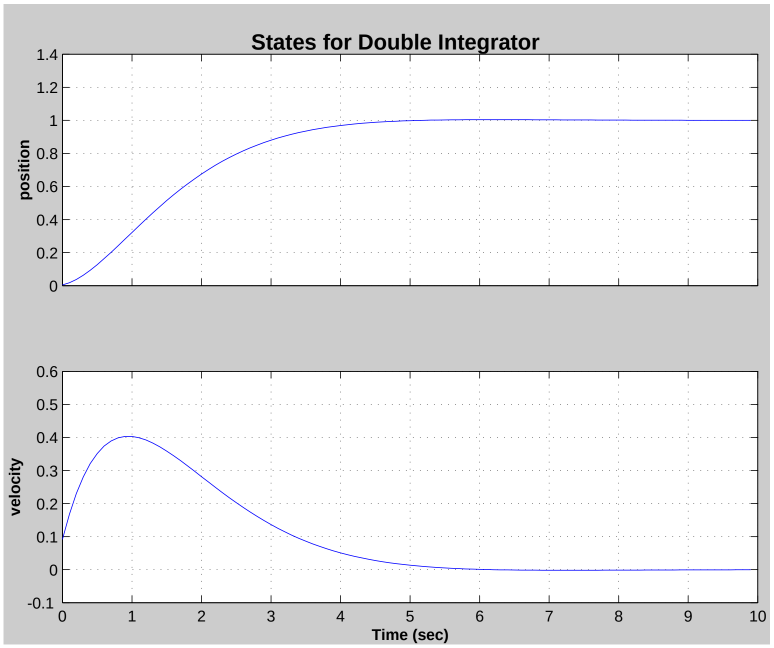
Next, select the *control* block (top center) and then select the LQ tab. Select full state feedback, at the top of the list. Input fields for Q, R, and N will appear. Enter q and r into the corresponding input fields. The display will look as follows (Figure 11.3 on page 100). Push Create. The values 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 buy 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).

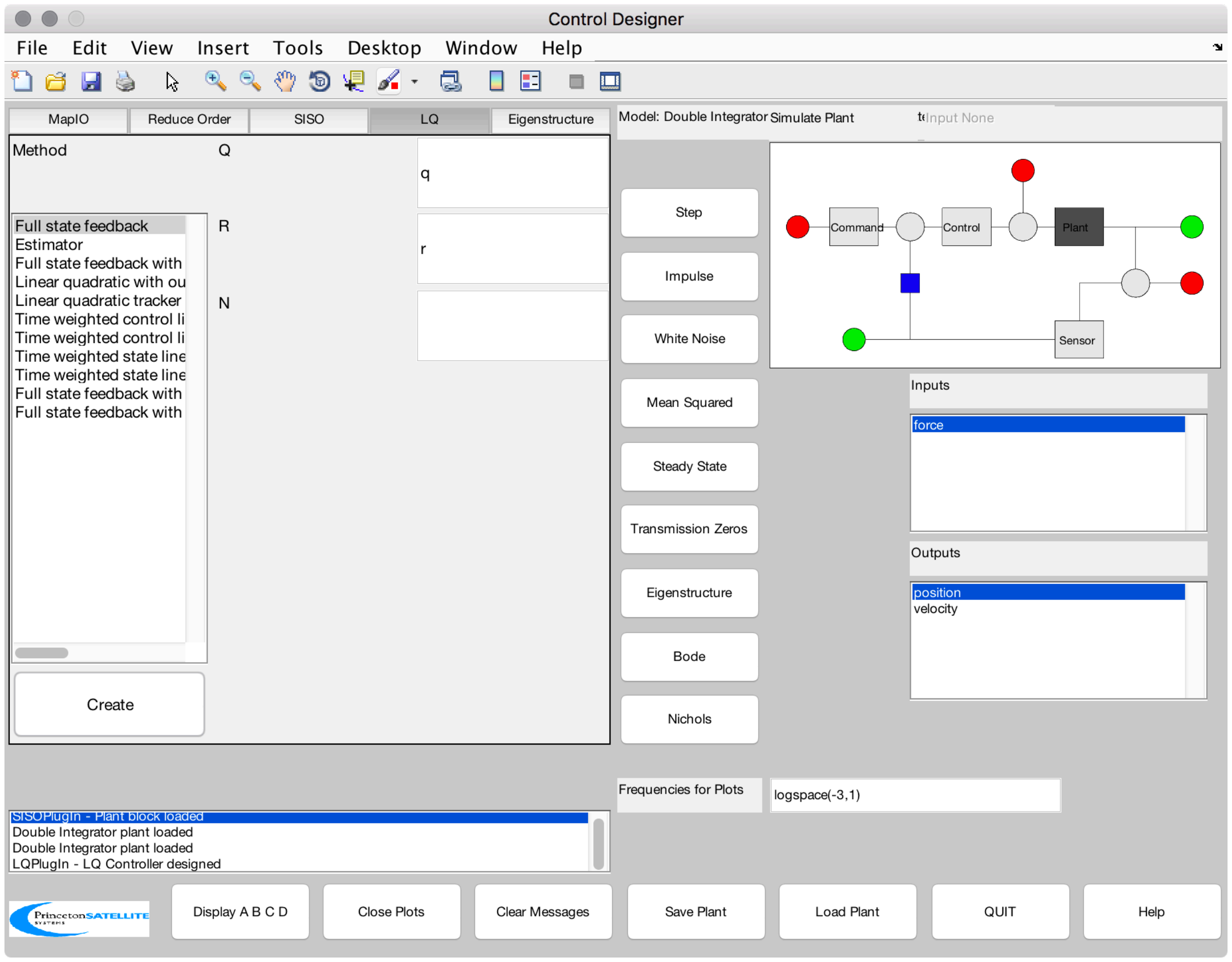
11.4 Eigenstructure Assignment

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



Full state feedback
Estimator
Full state feedback with
Linear quadratic with ou
Linear quadratic tracker
Time weighted control li
Time weighted control li
Time weighted state line
Time weighted state line
Full state feedback with
Full state feedback with

force

position
velocity

logspace(-3,1)

```
SISOPlugIn - Plant block loaded
Double Integrator plant loaded
Double Integrator plant loaded
LQPlugIn - LQ Controller designed
```

Figure 11.4: Step response from the GUI

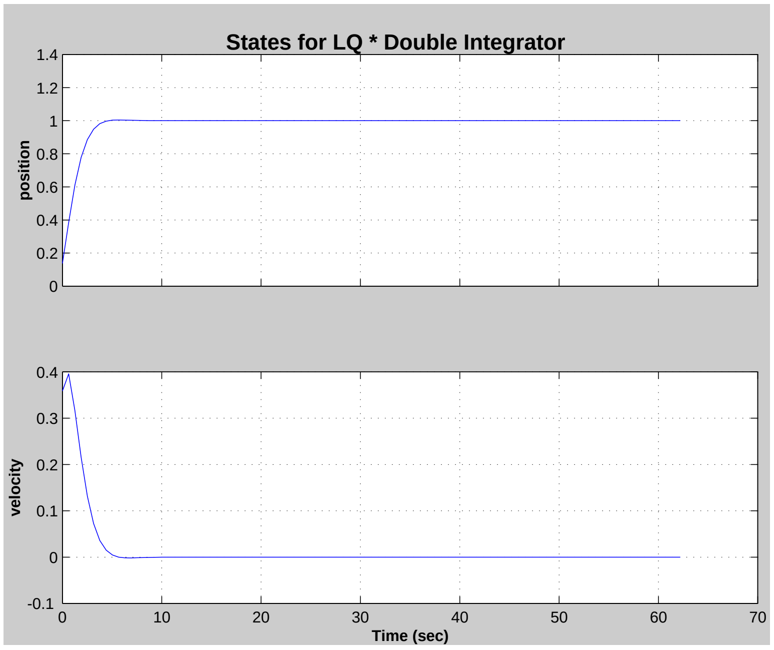
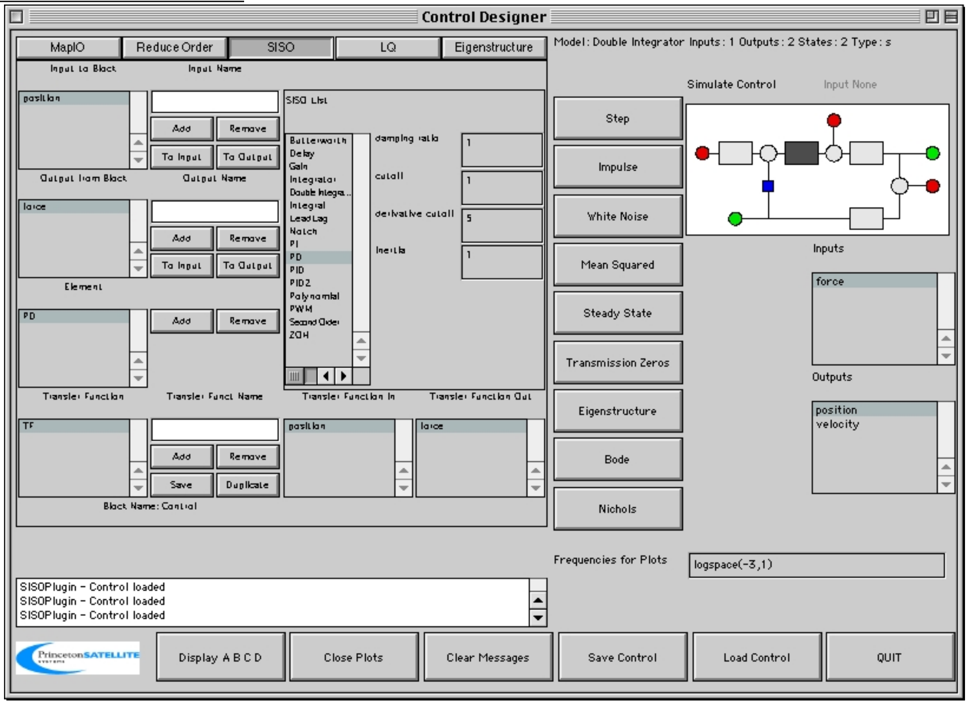


Figure 11.5: SISO inputs



Model: Double Integrator Inputs: 1 Outputs: 2 States: 2 Type: s

Input to Block: position

Output from Block: force

Element: PD

Transfer Function: TF

Transfer Function In: position

Transfer Function Out: force

Block Name: Control

damping ratio: 1

cutoff: 1

derivative cutoff: 5

inertia: 1

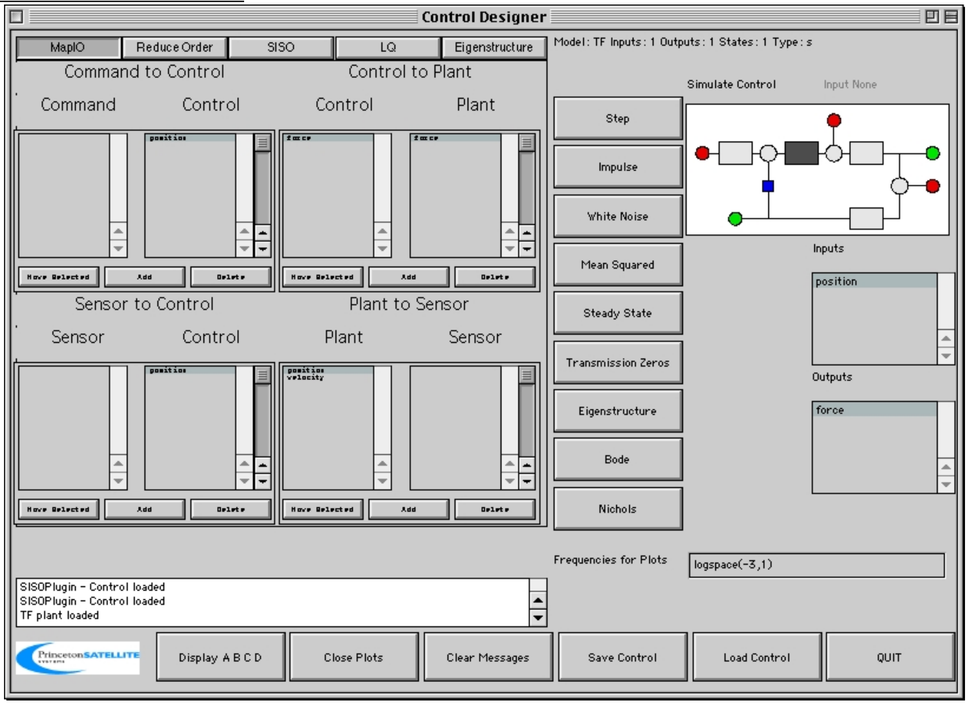
Inputs: force

Outputs: position, velocity

Frequencies for Plots: logspace(-3,1)

```
SISOPlugin - Control loaded
SISOPlugin - Control loaded
SISOPlugin - Control loaded
```

Figure 11.6: MapIO



Model: TF Inputs: 1 Outputs: 1 States: 1 Type: s

position

force

position
velocity

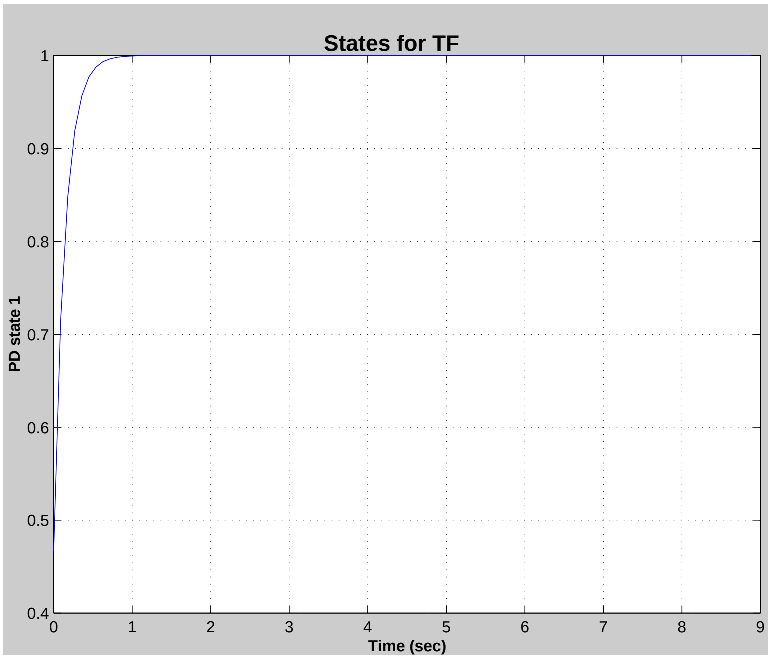
Inputs: position

Outputs: force

Frequencies for Plots: logspace(-3,1)

```
SISOPlugin - Control loaded
SISOPlugin - Control loaded
TF plant loaded
```

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 );
```

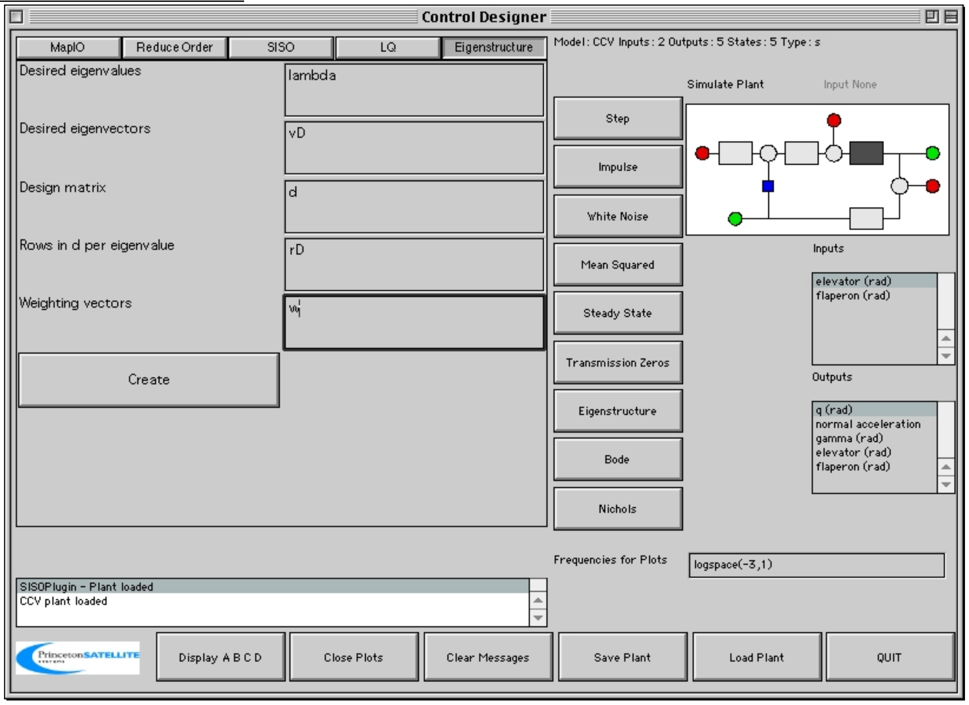
CCVDemo.m

`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 the following page.

Push Create. Next push Step. You will see the plot in Figure 11.9 on the next page.

Figure 11.8: Eigenstructure design GUI



Model: CCV Inputs: 2 Outputs: 5 States: 5 Type: s

Desired eigenvalues: lambda
Desired eigenvectors: vD
Design matrix: d
Rows in d per eigenvalue: rD
Weighting vectors: w

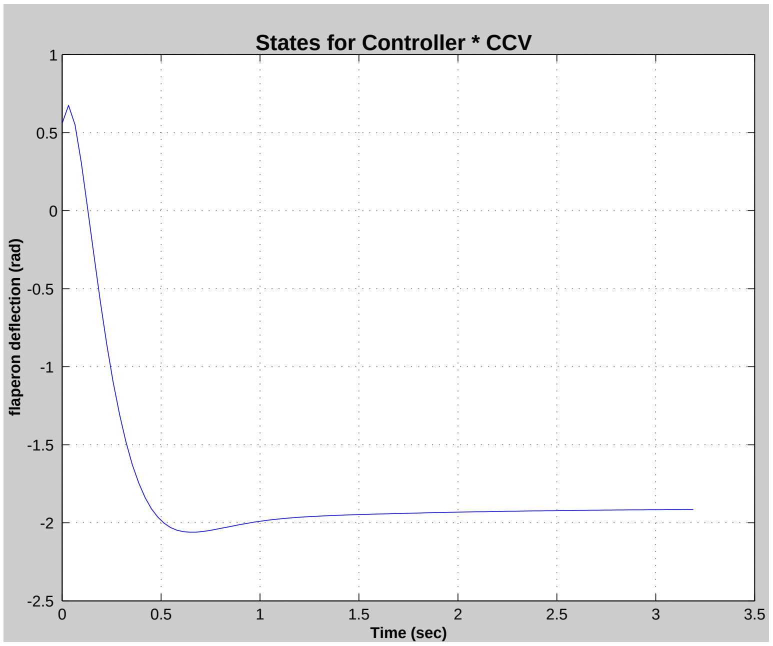
Inputs:
elevator (rad)
flaperon (rad)

Outputs:
q (rad)
normal acceleration
gamma (rad)
elevator (rad)
flaperon (rad)

Frequencies for Plots: logspace(-3,1)

```
SISOPlugin - Plant loaded
CCV plant loaded
```

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

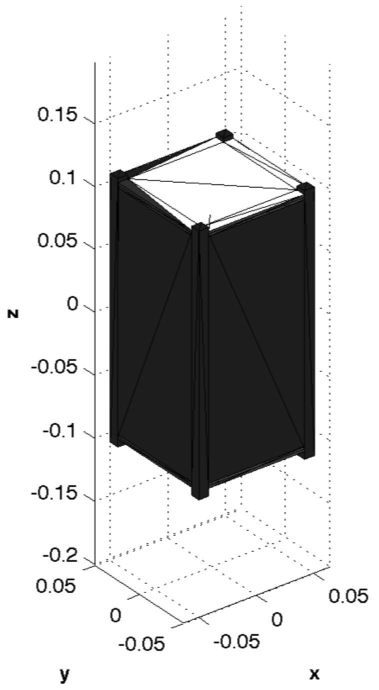
12.1 CubeSat Modeling

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 the facing page. 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.

Table 12.1: CubeSat Modeling Functions

AddMass	Combine component masses and calculate inertia and center-of-mass.
InertiaCubeSat	Compute the inertia for standard CubeSat types.
CubeSatFaces	Compute surface areas and normals for the faces of a CubeSat.
CubeSatModel	Generate vertices and faces for a CubeSat model.
TubeSatModel	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 orthgononal 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 facing 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 next page shows the typical orbit results.

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

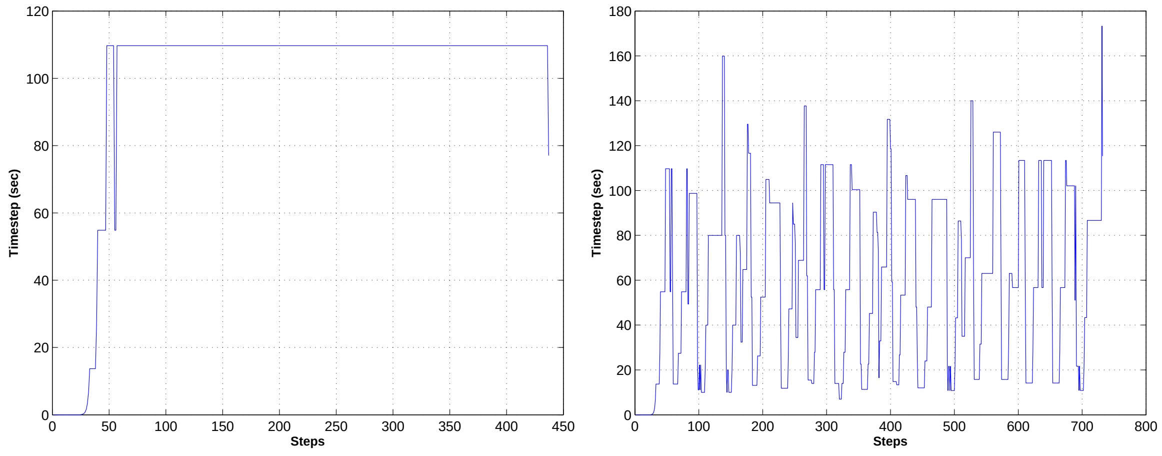
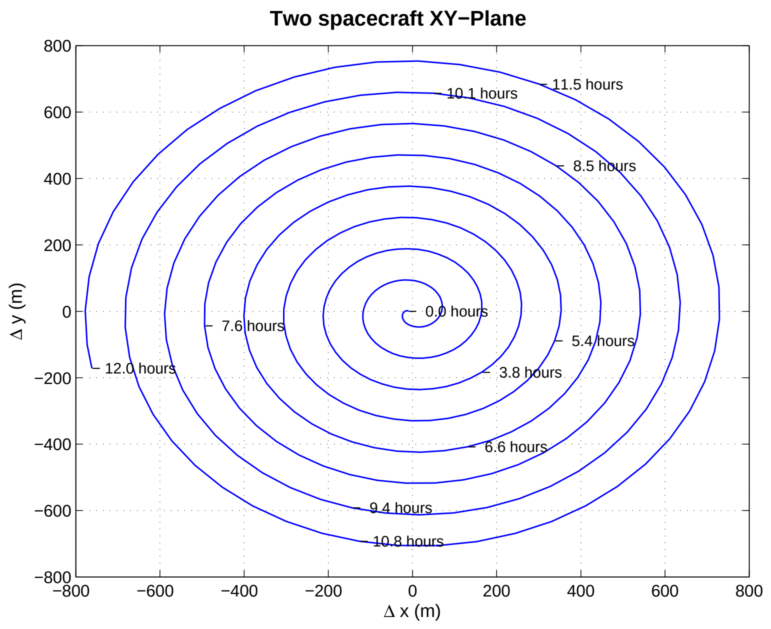


Figure 12.3: Orbit evolution for an initial separation of 10 meters

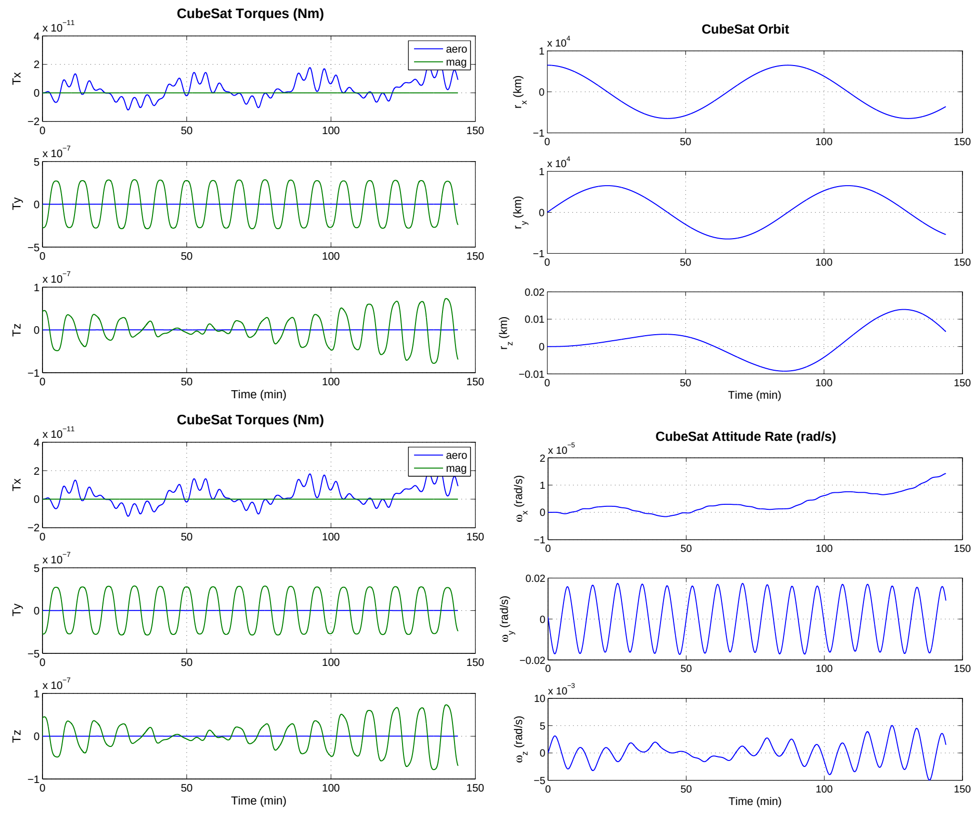


`CubeSatSimulation` simulates the attitude dynamics of the CubeSat in addition to a point-mass orbit and the power subsystem. This simulation includes forces and torques from drag, radiation pressure, and an magnetic torque, such as from a torquer control system. Since this simulation can include control, 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 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 next page. The space environment calculations from `CubeSatEnvironment` are then passed to the force models in `CubeSatAero` and `CubeSatRadiationPressure`.

Table 12.2: CubeSat Simulation Functions

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

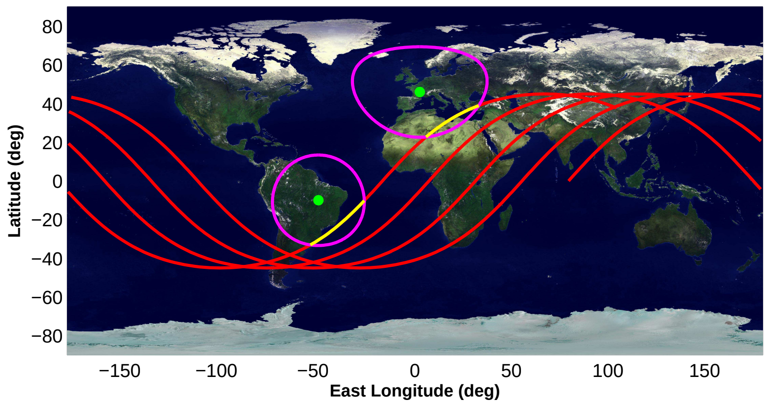
`RHSCubeSat` also models battery charging if the `batteryCapacity` field of the `power` structure has been appropriately set. Power beyond the calculated consumption will be used to charge the battery until the capacity is reached. The battery charge is always be the last element of the spacecraft state (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. There are two ground targets, one in South America and one in France (large green dots).

Figure 12.5: Observation time windows

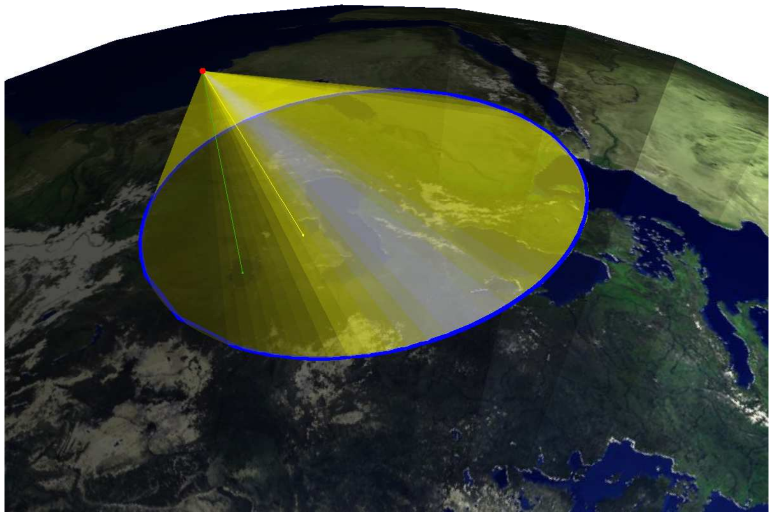


The satellite is placed in a low Earth orbit, given a field of view of 180 degrees, and the windows are generated over a 7 hour horizon. The figure shows the field of view in magenta and the satellite trajectory segments when the target is in the field of view are highlighted in yellow. This function 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 next 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.

Figure 12.6: `RapidSwath` built-in demo results

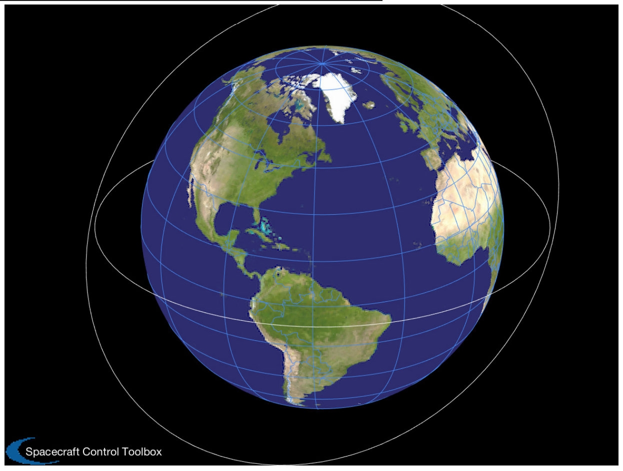


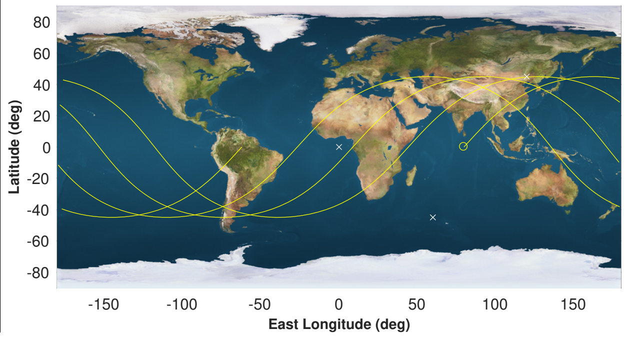
12.4 Visualization

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.

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

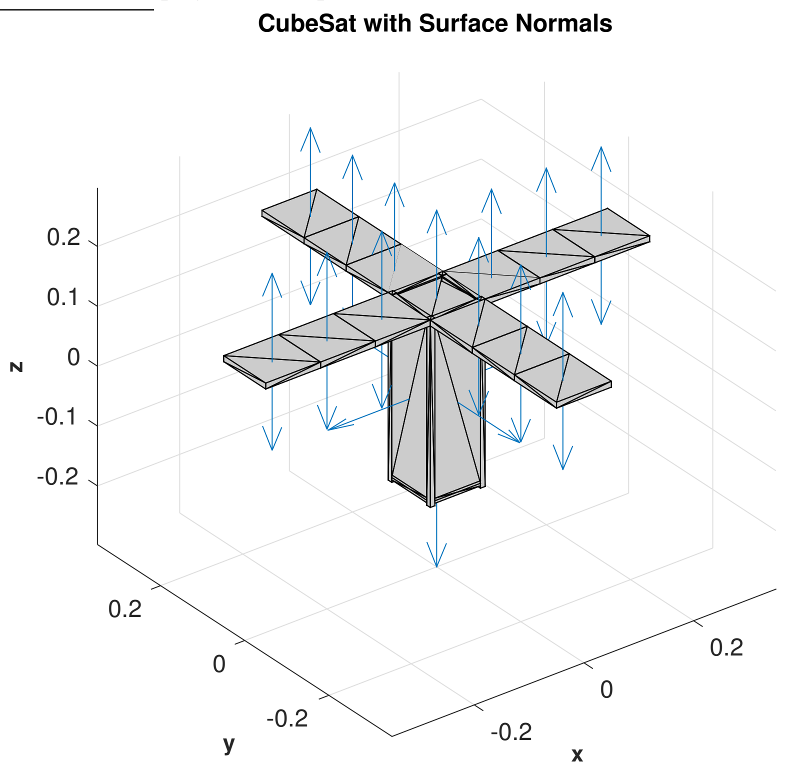




The spacecraft model from `CubeSatModel` can be viewed with surface normals using `DrawCubeSat`. The vertex and face information is not retained with the dynamical data, so `DrawCubeSatSolarAreas` can be used to verify the solar cell areas directly from the RHS data structure.

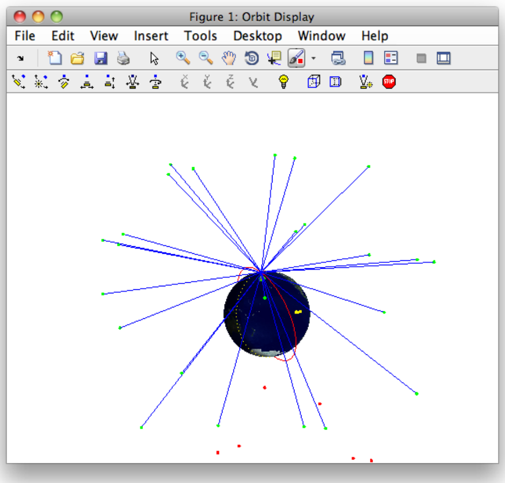
A representative model of the spacecraft may also be viewed in its orbit, along with a sensor cone and lines of sight to all of the visible GPS satellites. Use `DrawSpacecraftInOrbit.m` to generate this view. An example is shown in Figure 12.9 on the facing page. The image on the left shows the spacecraft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

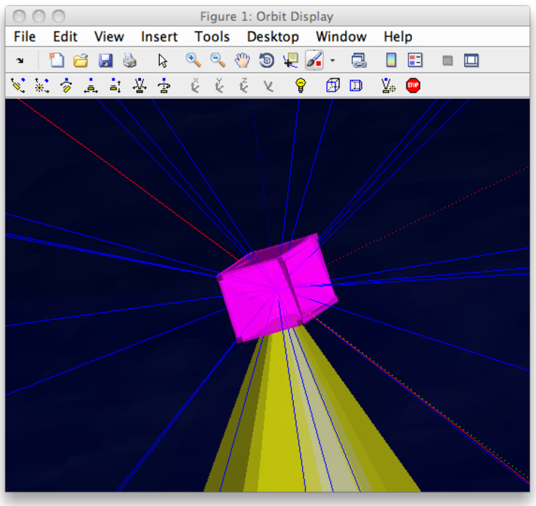
Figure 12.8: CubeSat model with deployable solar panels viewed with `DrawCubeSat`



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.

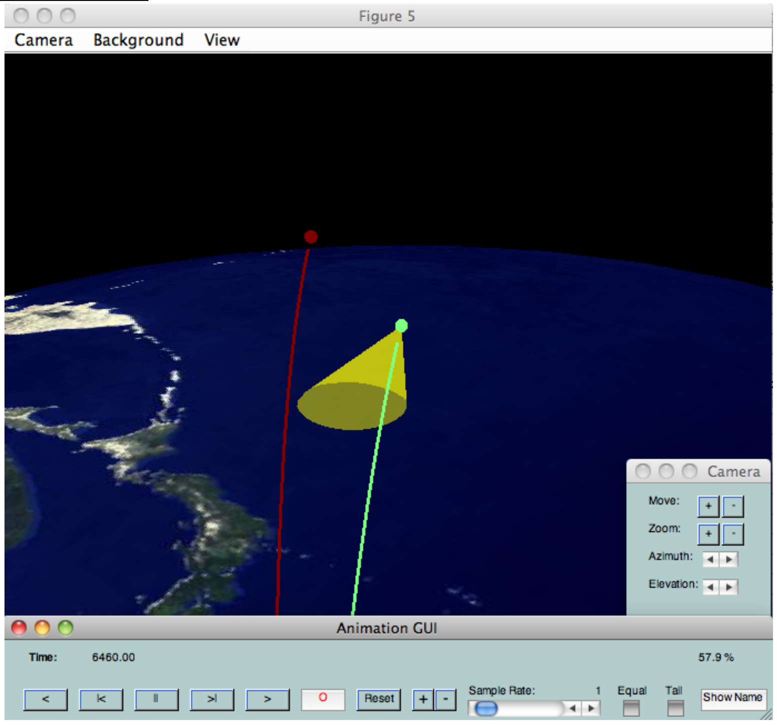
Figure 12.9: Spacecraft visualization with sight lines using `DrawSpacecraftInOrbit`





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 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.

APPENDIX A

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 `DSSRYYMM.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 the next page.

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

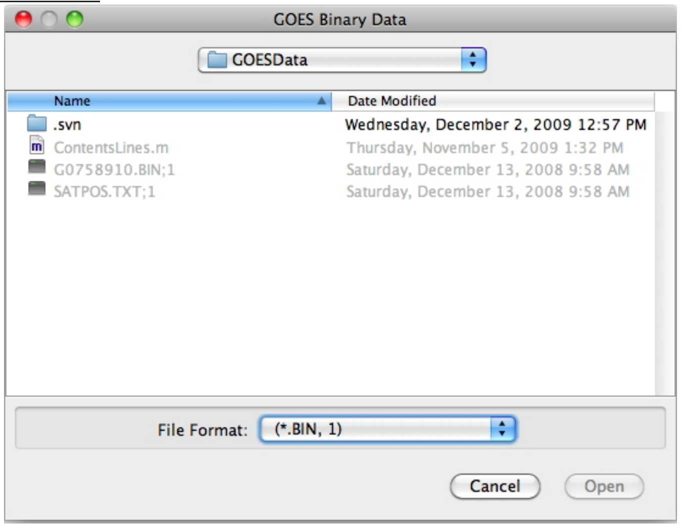
```
>> Weather
```

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



Name	Date Modified
.svn	Wednesday, December 2, 2009 12:57 PM
ContentsLines.m	Thursday, November 5, 2009 1:32 PM
G0758910.BIN;1	Saturday, December 13, 2008 9:58 AM
SATPOS.TXT;1	Saturday, December 13, 2008 9:58 AM

File Format: (*.BIN, 1)

Figure A.2: GOES Magnetic Field Data

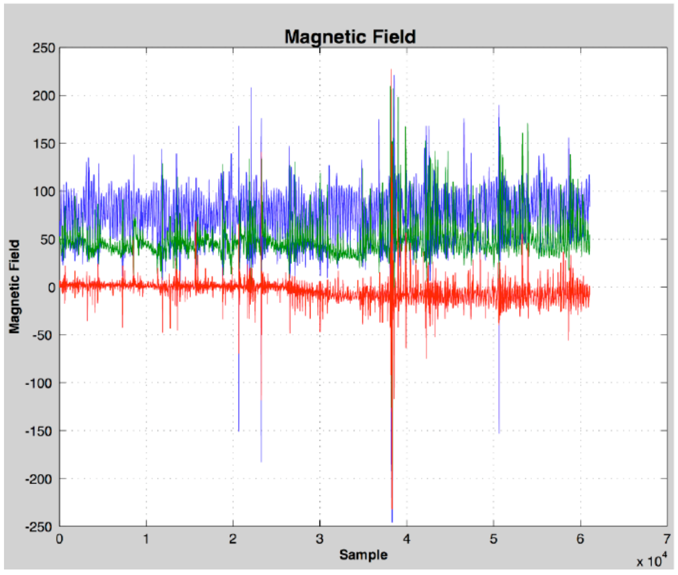


Figure A.3: GOES Electrons

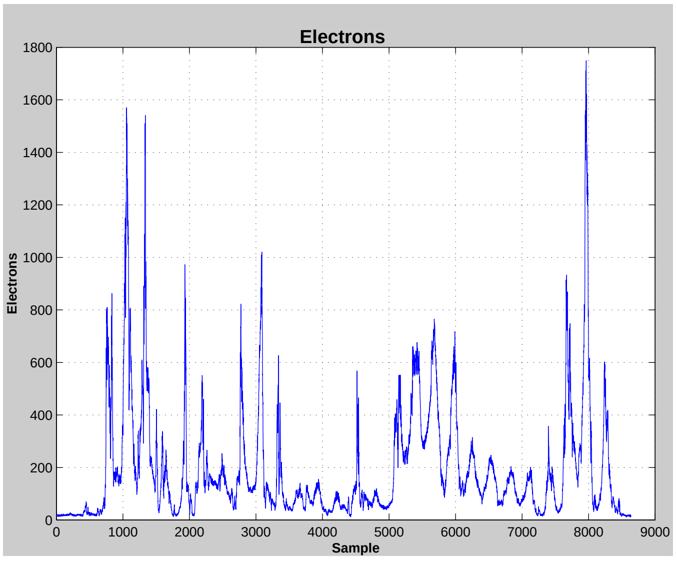
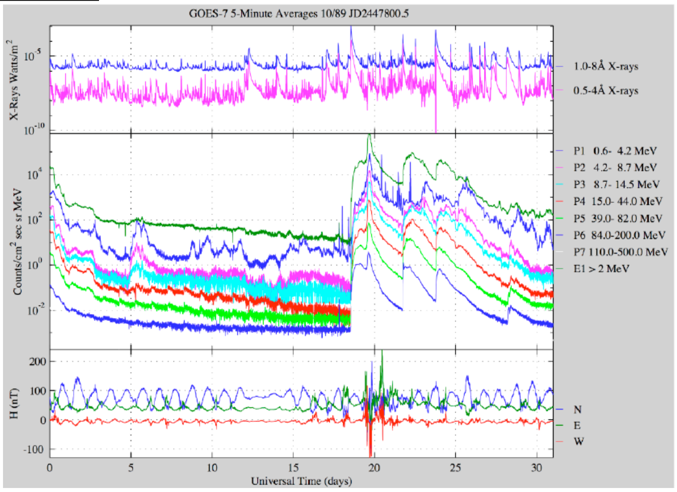


Figure A.4: GOES Weather



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 Documention. National Oceanic and Atmospheric Administration National Geophysical Data Center Solar-Terrestrial Physics Division, July 18, 1994.

APPENDIX B

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.

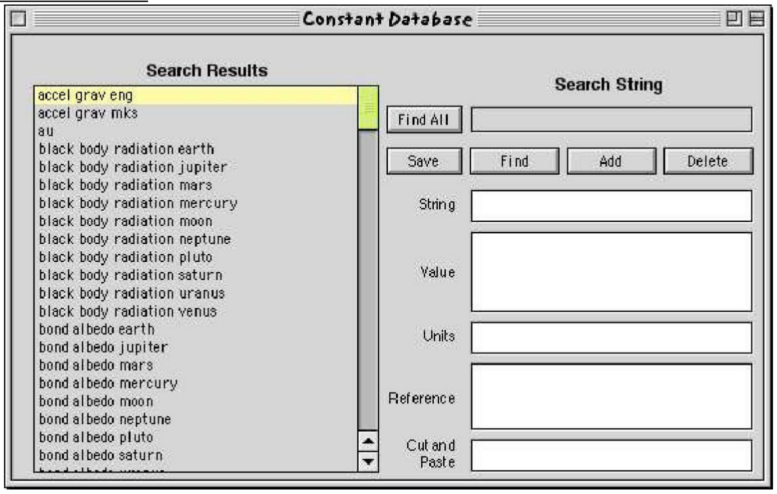
B.1 The Constant Database

The constant database gives a substantial selection of useful constants. If you type

```
Constant
```

you will get the GUI in Figure B.1.

Figure B.1: Constant database

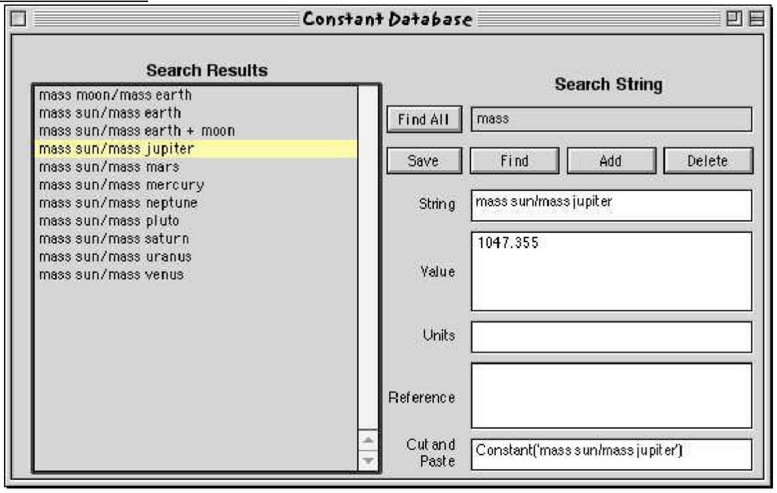


```
accel grav eng
accel grav mks
au
black body radiation earth
black body radiation jupiter
black body radiation mars
black body radiation mercury
black body radiation moon
black body radiation neptune
black body radiation pluto
black body radiation saturn
black body radiation uranus
black body radiation venus
bond albedo earth
bond albedo jupiter
bond albedo mars
bond albedo mercury
bond albedo moon
bond albedo neptune
bond albedo pluto
bond albedo saturn
```

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



mass moon/mass earth
mass sun/mass earth
mass sun/mass earth + moon
mass sun/mass jupiter
mass sun/mass mars
mass sun/mass mercury
mass sun/mass neptune
mass sun/mass pluto
mass sun/mass saturn
mass sun/mass uranus
mass sun/mass venus

Search String: mass
String: mass sun/mass jupiter
Value: 1047.355
Cut and Paste: Constant('mass sun/mass jupiter')

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:

```
Constant('mass␣sun/mass␣jupiter')
ans =
   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

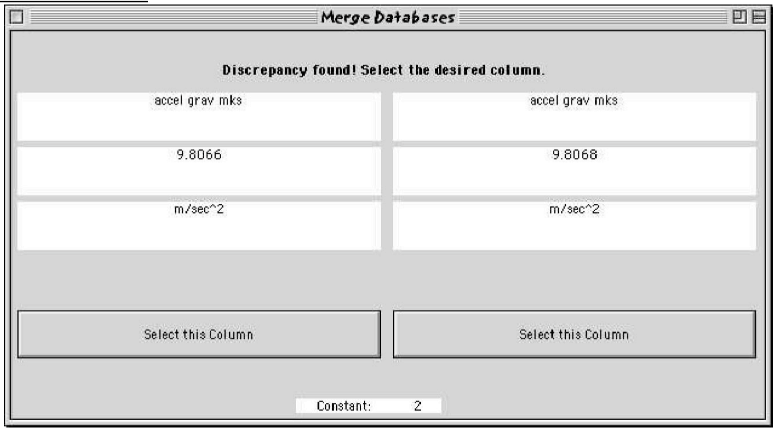
```
MergeConstantDB( initialize, a, b )
```

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 the next page. Just click the button for the column you wish to include in `SCTConstants.mat`.

Figure B.3: `MergeConstantDB` function



Discrepancy found! Select the desired column.

accel grav mks	accel grav mks
9.8066	9.8068
m/sec^2	m/sec^2

Constant: 2

APPENDIX C

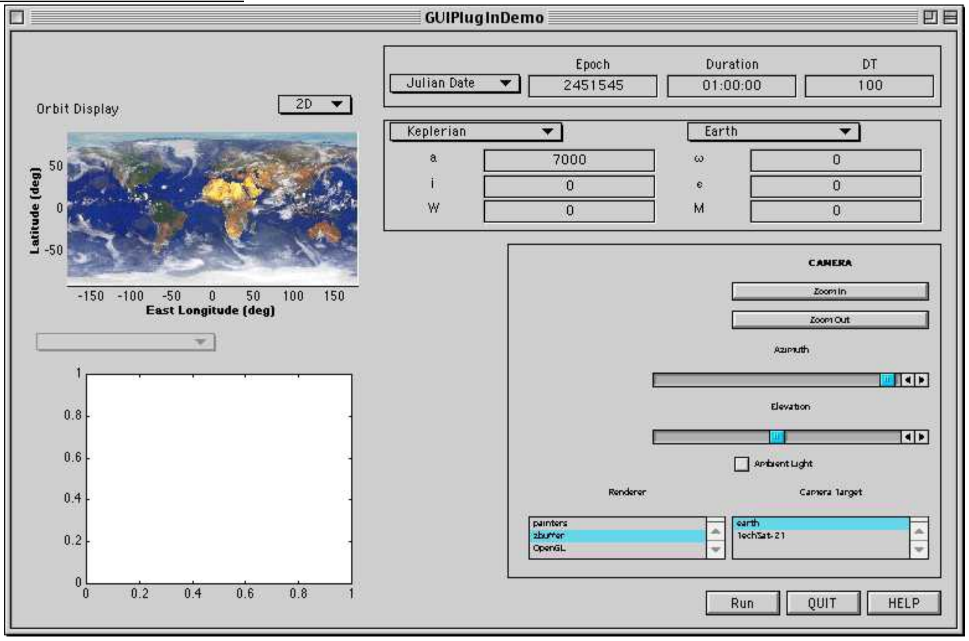
GUI PLUG INS

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



Julian Date
Epoch 2451545
Duration 01:00:00
DT 100
Keplerian
Earth
a 7000
i 0
W 0
ω 0
e 0
M 0
Orbit Display 2D
Renderer: painters, zbuffer, OpenGL
Camera Target: earth, TechSat-21

The GUI PlugIns shown are from top to bottom starting on the left:

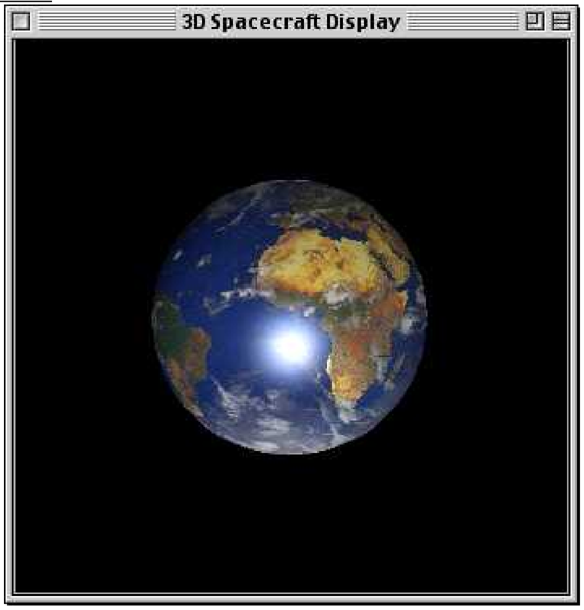
- OrbitDisplayPlugin

- 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.

Figure C.2: PlugInDemo with the earth as center



You can change planets as shown in Figure C.3 on the next page.

If you hit Run, the simulation will run. The results at the end are shown in Figure C.4 on the facing page.

The 3D window (which is animated) looks like Figure C.5 on page 128.

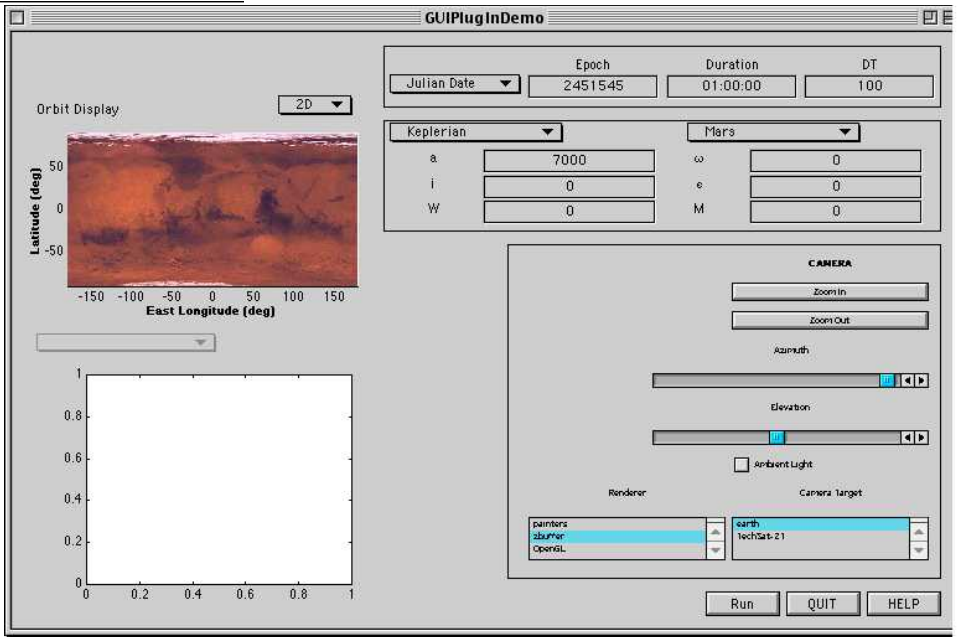
C.2 Writing Your Own GUI Function

`PlugInDemo` is implemented as shown below.

Listing C.1: PlugInDemo top-level switch statement *PlugInDemo.m*

```
function PlugInDemo( action )
% Process the input arguments
%----------------------------
if( nargin < 1 )
action = 'create␣gui';
end
% Perform actions
%----------------
switch action
```

Figure C.3: Selecting planets in the elements plugin



Orbit Display: 2D

Julian Date	Epoch	Duration	DT
	2451545	01:00:00	100

Keplerian — Mars

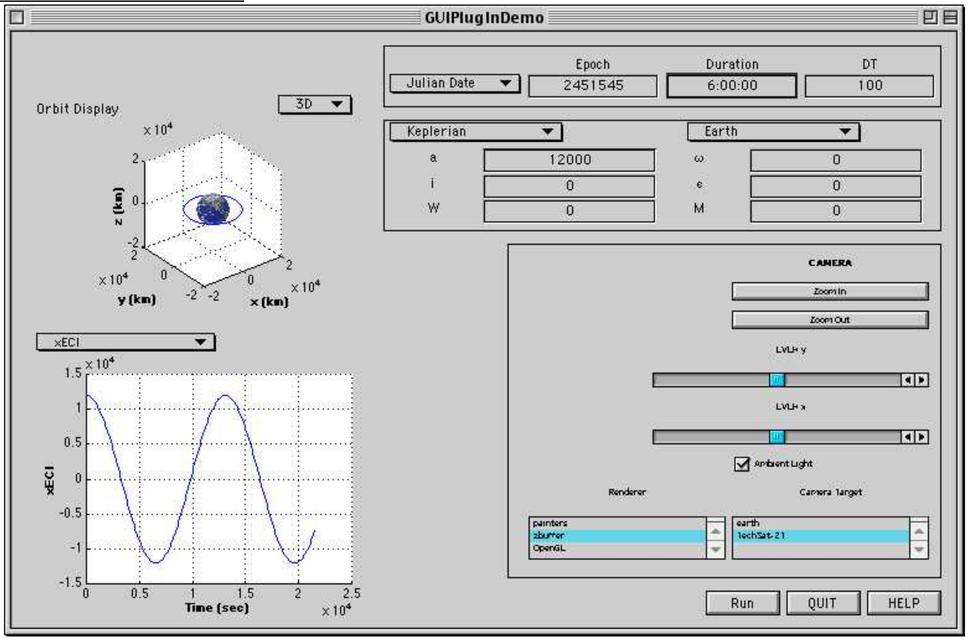
a	7000	ω	0
i	0	e	0
W	0	M	0

Ambient Light: ☐

Renderer: painters, zbuffer, OpenGL

Camera Target: earth, TechSat-21

Figure C.4: PlugInDemo at the end of the simulation



Orbit Display: 3D

Julian Date	Epoch	Duration	DT
	2451545	6:00:00	100

Keplerian — Earth

a	12000	ω	0
i	0	e	0
W	0	M	0

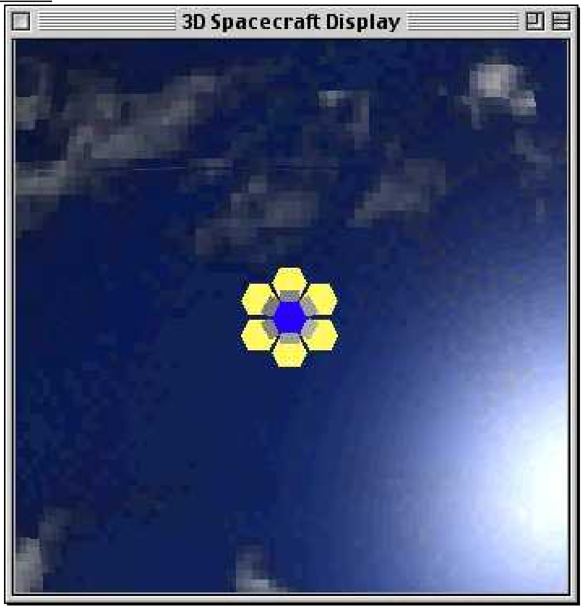
Plot: xECI

Ambient Light: ☑

Renderer: painters, zbuffer, OpenGL

Camera Target: earth, TechSat-21

Figure C.5: PlugInDemo 3D Spacecraft Display



```
case 'help'
HelpSystem( 'initialize', 'SCHelp' );
case 'create_gui'
h = GetH;
if( isempty(h) )
CreateGUI;
else
figure( h.fig );
end
case 'run'
Run;
case 'changed'
Update;
case 'quit'
h = GetH;
CloseFigure( h.fig );
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*

```
function CreateGUI

% The figure window
%-------------------
p = [5 5 760 480];
h.fig = figure( 'name','GUIPlugInDemo','Units','pixels', 'Position',[40 p(4) - 600
p(3:4)],'resize','off', 'NumberTitle', 'off','tag', 'GUI_PlugIn_Demo',
'CloseRequestFcn', CreateCallback( 'quit' ) );

```

```
% Buttons
%--------
v = {'parent', h.fig, 'units', 'pixels', 'fontunits', 'pixels'};
r = p(1) + p(3);
h.run = uicontrol( v{:}, 'Position', [r-205 10 60 20], 'callback',
CreateCallback( 'run' ), 'string','Run' );
h.quit = uicontrol( v{:}, 'Position', [r-140 10 60 20], 'callback',
CreateCallback( 'quit' ), 'string','QUIT');
h.help = uicontrol( v{:}, 'Position', [r- 75 10 60 20], 'callback',
CreateCallback( 'help' ), 'string','HELP');

% Initialize the plugins
%-----------------------
cB = 'PlugInDemo(␣''changed''␣)';
h.orbitDisplayTag = OrbitDisplayPlugIn( 'initialize', [], h.fig, [ 5 250 290
185], [] );
h.plotPlugInTag = PlotPlugIn( 'initialize', [], h.fig, [ 5 10 290 235] );
h.timePlugInTag = TimePlugIn( 'initialize', [], h.fig, [300 420 450 50], cB );
h.elementsPlugInTag = ElementsPlugIn( 'initialize', [], h.fig, [300 320 450 90],
cB );

% Initialize the 3D window
%-------------------------
sim = GetSimData( h );
h.g = load('TechSat-21');
if( isfield( sim.orbit, 'r' ) & isfield( sim.orbit, 'v' ) )
  h.g.body(1).bHinge.q = QLVLH(sim.orbit.r, sim.orbit.v );
else
  h.g.body(1).bHinge.q = [1;0;0;0];
end

h.g.name = 'TechSat-21';
if( isfield( sim.orbit, 'r' ) )
  h.g.rECI = sim.orbit.r;
else
  h.g.rECI = [sim.orbit.el(1);0;0];
end
h.g.qLVLH = h.g.body(1).bHinge.q;
h.scWindowTag = DrawSCPlugIn( 'initialize', h.g, h.fig, [400 40 350 270], 'earth',
sim.time.jDEpoch );

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*

```
function Run

% Get the simulation data
%-------------------------
h = GetH;
sim = GetSimData( h );

% Duration
%---------
duration = datenum(sim.time.duration)*86400;

% Check duration
%---------------
if( duration == 0 )
  msgbox('The␣duration␣is␣zero.␣Will␣not␣run␣the␣simulation.');
  return
end

% Create the basic state vector from required plug-ins
%------------------------------------------------------
x = [sim.orbit.r; sim.orbit.v];
jD = sim.time.jDEpoch;
nSim = duration/sim.time.dT;
y{1} = zeros(6,nSim);
t = 0;
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*

```
for k = 1:nSim
  % Plotting
  %---------
  y{1}(:,k) = x;
  u(k) = t;

% Transformation matrices
%-------------------------
qLVLH = QLVLH( x(1:3), x(4:6) );
h.g.body(1).bHinge.q = QPose( qLVLH );
h.g.rECI = x(1:3);
h.g.qLVLH = qLVLH ;
DrawSCPlugin( 'update␣spacecraft', h.scWindowTag, h.g, jD );
% Propagate the orbits
%---------------------
x = RK4( 'FOrbCart', x, sim.time.dT, t, sim.orbit.mu );
% Update the time
%----------------
t = t + sim.time.dT;
jD = jD + sim.time.dT/86400;
end
```

PlugInDemo.m

The plots are created with the following code.

Listing C.5: Plotting *PlugInDemo.m*

```
% Plotting
%---------
if( ~isempty( y ) )
  PlotPlugIn( 'clear␣plots', h.plotPlugInTag, k );
  p.xLabel = {'Time␣(sec)'};
  p.yLabel = {'xECI' 'yECI' 'zECI' 'vXECI' 'vYECI' 'vZECI'};
  p.title = p.yLabel;
  PlotPlugIn( 'update␣labels', h.plotPlugInTag, p );
  PlotPlugIn( 'add␣points', h.plotPlugInTag, struct('x', u, 'y', y ) );
  PlotPlugIn( 'plot', h.plotPlugInTag );
end

jD = sim.time.jDEpoch + (0:(nSim-1))*sim.time.dT/86400;
OrbitDisplayPlugIn( 'draw', h.orbitDisplayTag, {y{1}(1:3,:)}, jD,
sim.orbit.planet );

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*

```
function Update
% Get the simulation data
%------------------------
h   = GetH;
sim = GetSimData( h );

% Update the spacecraft state
%----------------------------
if( isfield( sim.orbit, 'r' ) & isfield( sim.orbit, 'v' ) )
  qLVLH = QLVLH(sim.orbit.r, sim.orbit.v );
else
  qLVLH = [1;0;0;0];
end

h.g.body(1).bHinge.q = QPose( qLVLH );
if( isfield( sim.orbit, 'r' ) )
h.g.rECI = sim.orbit.r;
else
h.g.rECI = [sim.orbit.el(1);0;0];
end
h.g.qLVLH = qLVLH ;
DrawSCPlugin( 'update␣spacecraft', h.scWindowTag, h.g, sim.time.jDEpoch
);
OrbitDisplayPlugIn( 'clear␣plot', h.orbitDisplayTag, sim.orbit.planet );
DrawSCPlugin( 'update␣planet', h.scWindowTag, sim.orbit.planet );
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

```
function sim = GetSimData( h )

% Get the data from the plug ins
%-------------------------------
```

```
sim.orbit = ElementsPlugIn( 'get', h.elementsPlugInTag );
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*

```
%-----------------------------------------------------------------------
% Put the data into the figure handle
%-----------------------------------------------------------------------
function PutH( h )

set( h.fig, 'UserData', h );

%-----------------------------------------------------------------------
% Get the data from the figure handle
%-----------------------------------------------------------------------
function h = GetH

hFig = findobj( allchild(0), 'flat', 'Tag', 'GUI␣PlugIn␣Demo' );
h = get( hFig, 'userdata' );

%-----------------------------------------------------------------------
% Create a callback string
%-----------------------------------------------------------------------
function c = CreateCallback( action )

c = ['PlugInDemo(␣''' action '''␣)'];
```

PlugInDemo.m

INDEX