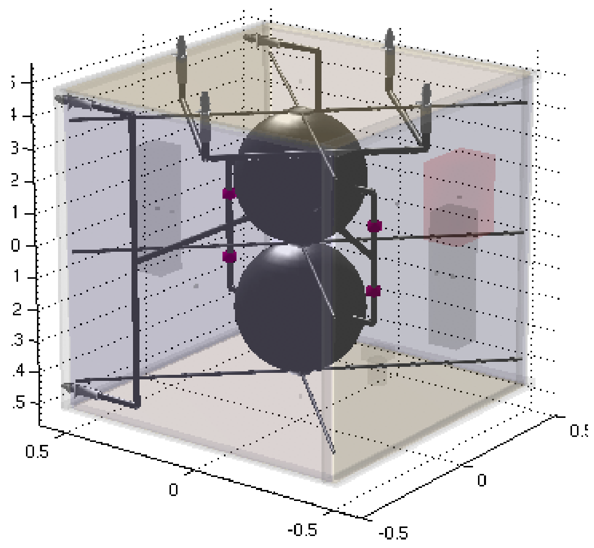
PrincetonSATELLITE SYSTEMS

Spacecraft Control Toolbox

User's Guide

Release 2019.1



Spacecraft Control Toolbox

December 26, 2019

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

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

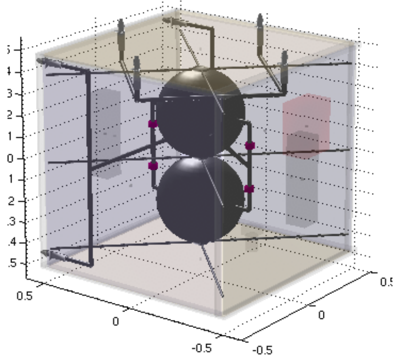
CONTENTS

CODE EXAMPLES

LIST OF FIGURES

PART I

TOOLBOX BASICS



CHAPTER 1

INTRODUCTION

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

1.1 Organization

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

Table 1.1: Spacecraft Control Toolbox Modules

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

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

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

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

1.2 Requirements

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

1.3 Installation

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

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

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

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

1.4 Getting Started

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

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

Another useful utility function you should try is `Finder`. This GUI allows you to search your entire path, or selected folders, for instances of any keyword. While `FileHelp` allows you to search headers for keywords, `Finder` allows you to search for instances of a function or variable in the code itself. This GUI is described in Section 4.5.1. For example, you can use it to find all the demos that show the use of the function `SunV1`, which computes the sun vector from the date using an almanac.

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

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 following page. The index page of the SCT documentation is shown in Figure 2.2 on page 9.

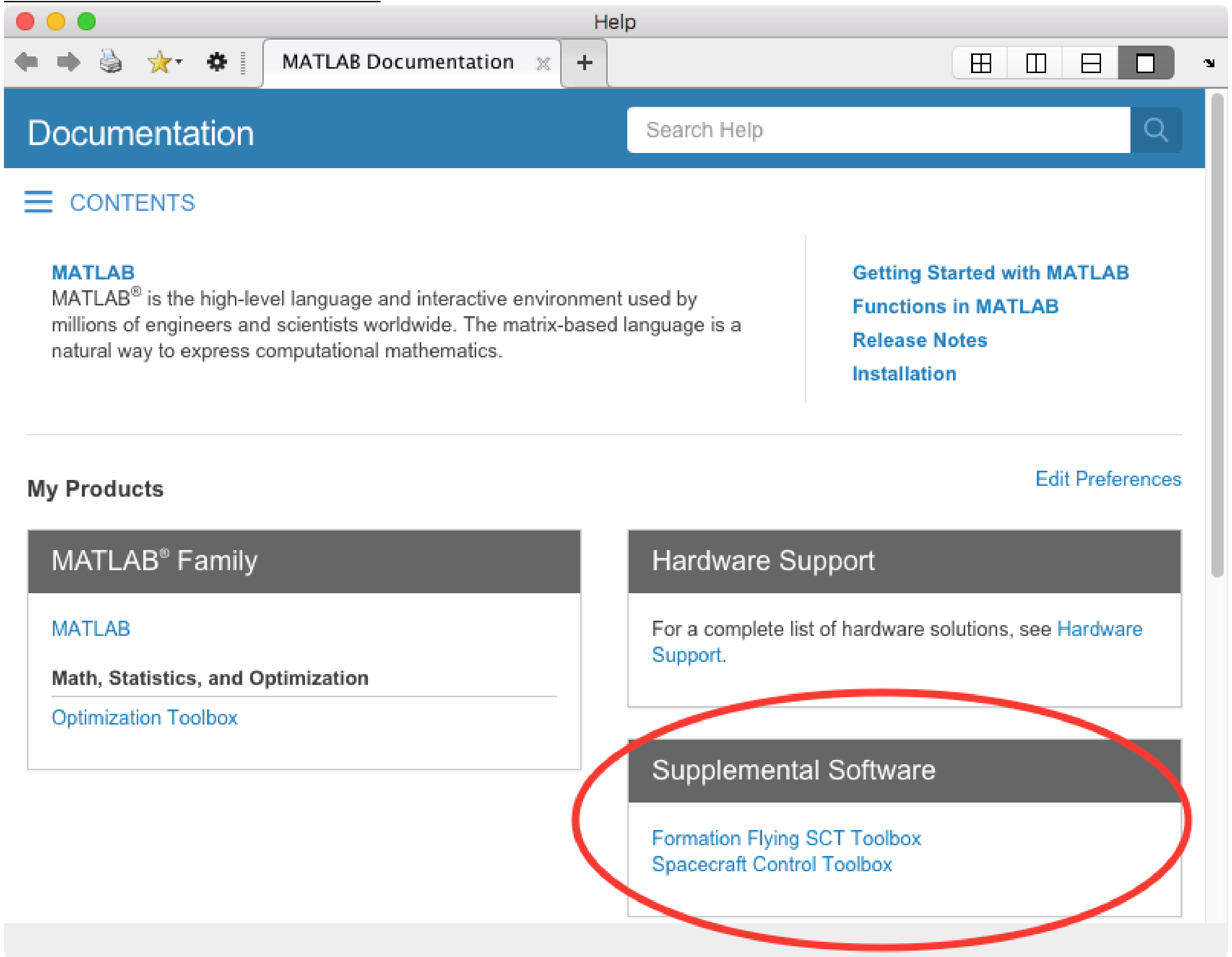
The help window from R2011b and earlier is depicted in Figure 2.3 on page 9.

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.

My Products

MATLAB® Family

MATLAB

Math, Statistics, and Optimization

Optimization Toolbox

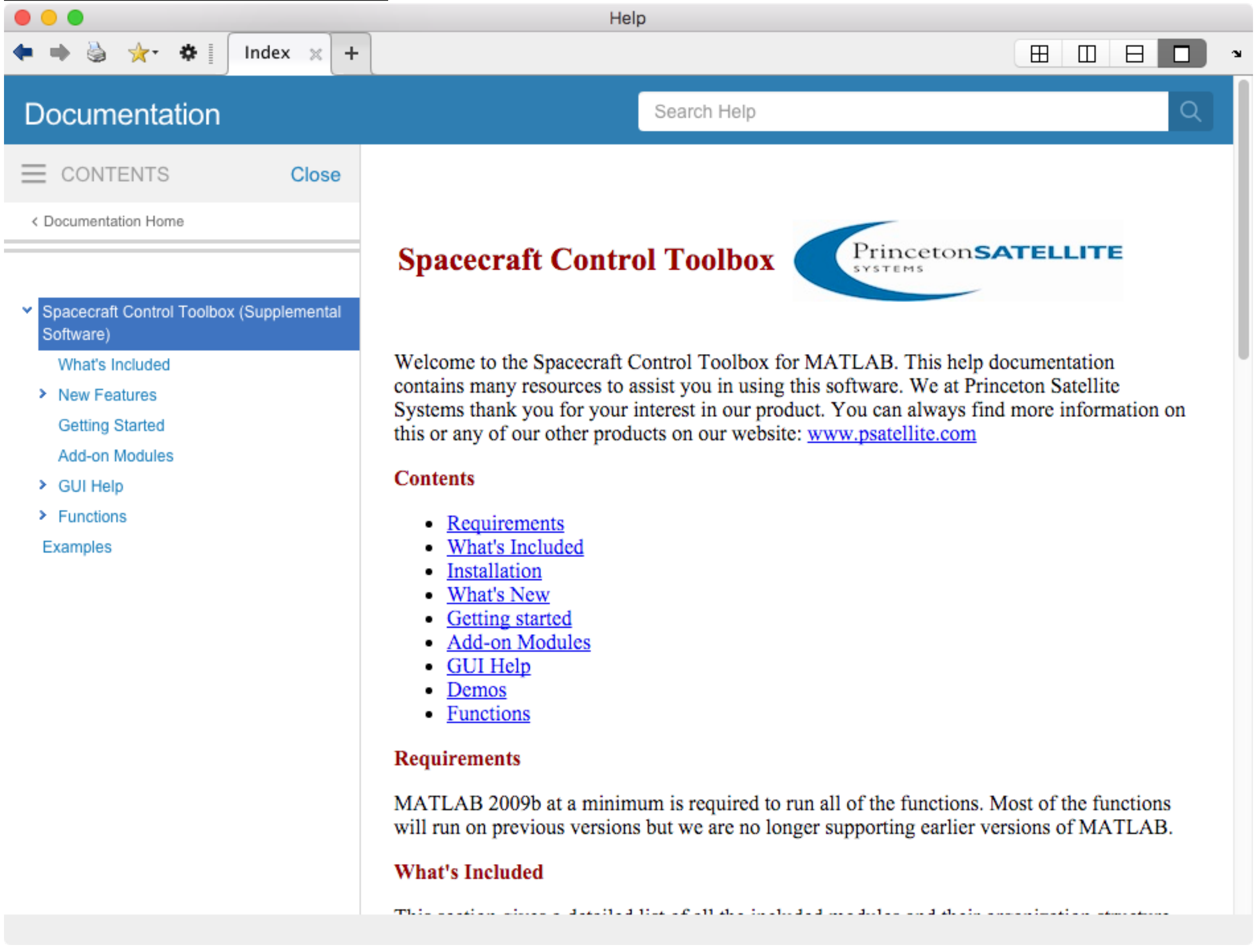
Hardware Support

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

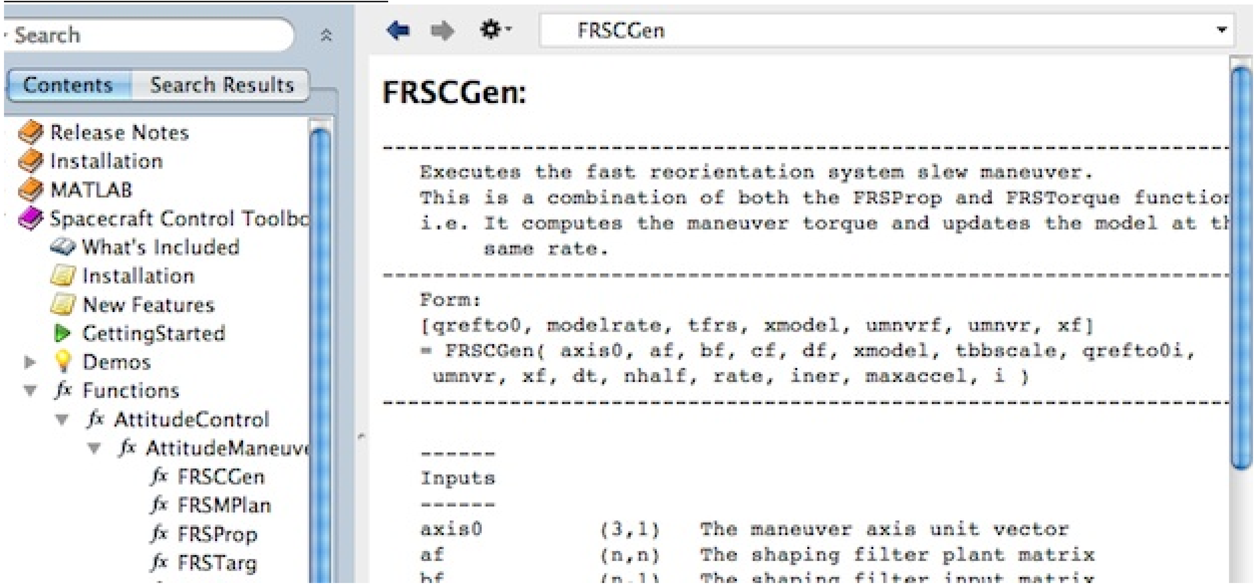
- Requirements
- What's Included
- Installation
- What's New
- Getting started
- Add-on Modules
- GUI Help
- Demos
- Functions

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

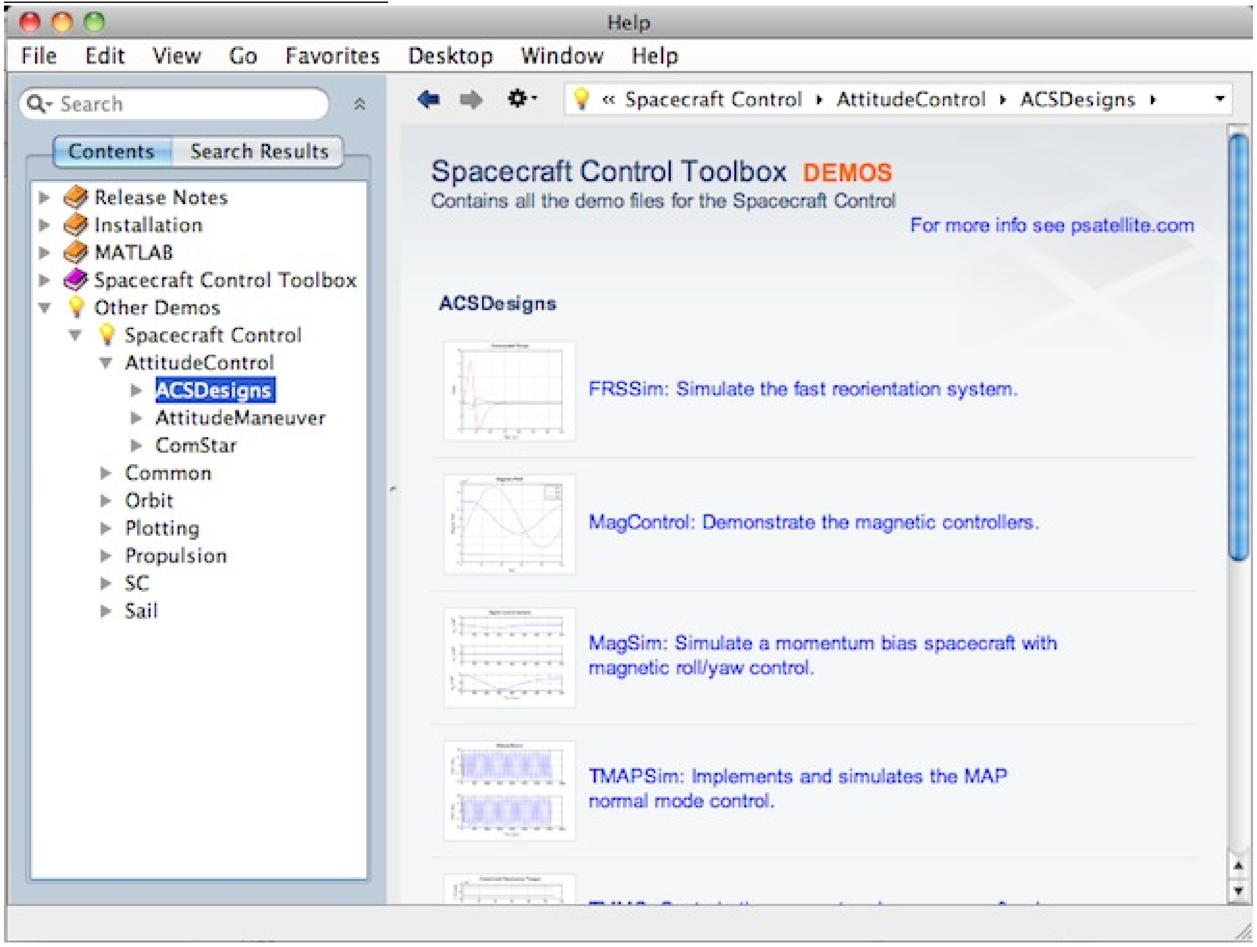
Figure 2.3: Toolbox Documentation, R2011b



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 using a zero order hold.

   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

-------------------------------------------------------------------------
   References: Van Loan, C.F., Computing Integrals Involving the Matrix
               Exponential, IEEE Transactions on Automatic Control
               Vol. AC-23, No. 3, June 1978, pp. 395-404.
-------------------------------------------------------------------------
```

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
Core/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
>> help Common
  PSS Toolbox Folder Common
  Version 2019.1      23-Dec-2019
  Directories:
  Atmosphere
  Classes
  CommonData
  ComponentModels
  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
  GUIs
  General
  Graphics
  Interface
  MassProperties
  Materials
  Plugins
  Quaternion
  Time
  Transform
```

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

```
>> ver
----------------------------------------------------------------------------------------------------

MATLAB Version: 9.6.0.1072779 (R2019a)
MATLAB License Number: 346509
Operating System: Mac OS X  Version: 10.15.2 Build: 19C57
Java Version: Java 1.8.0_181-b13 with Oracle Corporation Java HotSpot(TM) 64-Bit
    Server VM mixed mode
----------------------------------------------------------------------------------------------------

MATLAB                                              Version 9.6         (R2019a
    )
Deep Learning Toolbox                               Version 12.1        (R2019a
    )
Image Acquisition Toolbox                           Version 6.0         (R2019a
    )
Image Processing Toolbox                            Version 10.4        (R2019a
    )
Instrument Control Toolbox                          Version 4.0         (R2019a
    )
Optimization Toolbox                                Version 8.3         (R2019a
    )
PSS Toolbox Folder AerospaceUtils                   Version 2019.1
PSS Toolbox Folder Common                              Version 2019.1
PSS Toolbox Folder CubeSat                             Version 2019.1
PSS Toolbox Folder Electrical                          Version 2019.1
PSS Toolbox Folder Imaging                             Version 2019.1
PSS Toolbox Folder Link                                Version 2019.1
PSS Toolbox Folder LunarCube                           Version 2019.1
PSS Toolbox Folder Math                                Version 2019.1
PSS Toolbox Folder Missions                            Version 2019.1
PSS Toolbox Folder Orbit                               Version 2019.1
PSS Toolbox Folder Plotting                            Version 2019.1
PSS Toolbox Folder Propulsion                          Version 2019.1
PSS Toolbox Folder SC                                  Version 2019.1
PSS Toolbox Folder SCPro                               Version 2019.1
PSS Toolbox Folder SpacecraftEstimation                Version 2019.1
PSS Toolbox Folder Thermal                             Version 2019.1
```

2.3 FileHelp

2.3.1 Introduction

When you type

```
FileHelp
```

the FileHelp GUI appears, Figure 2.5 on the next page.

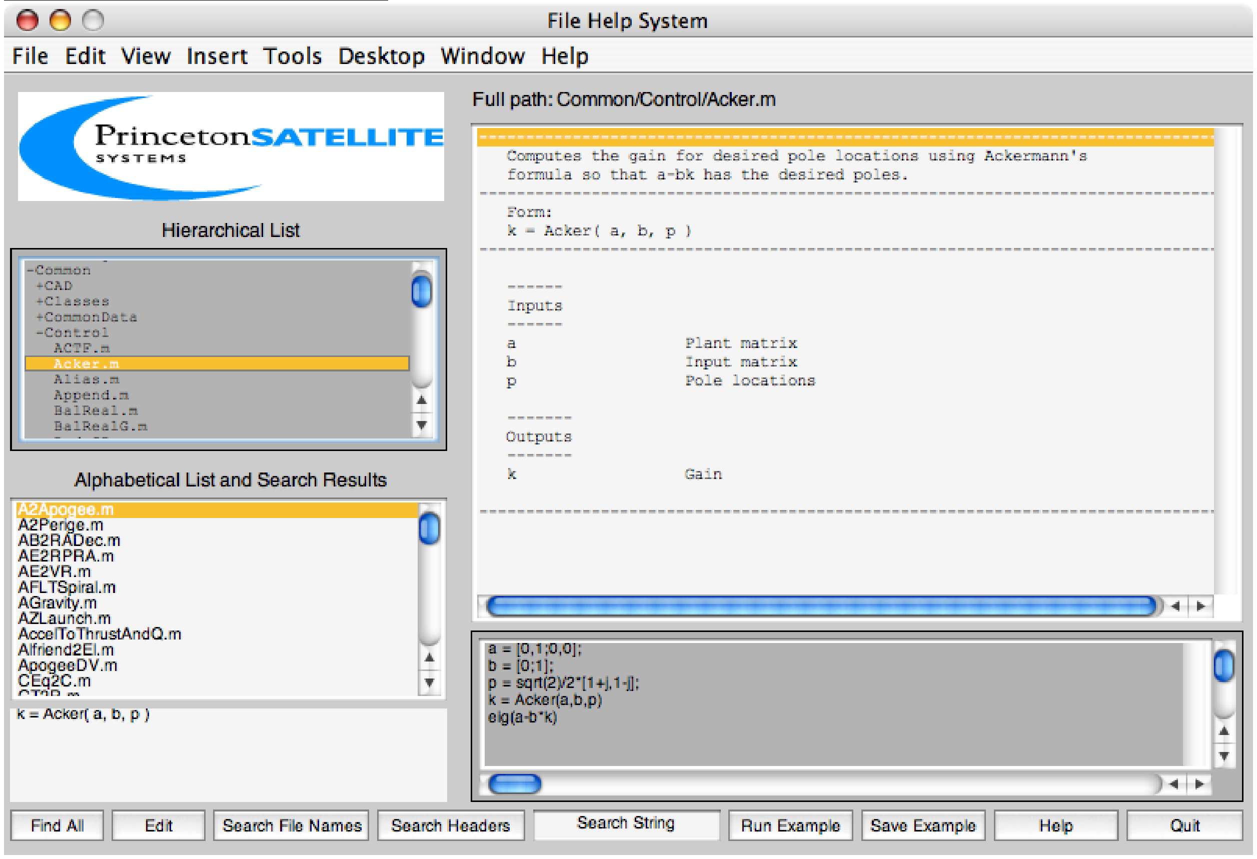
There are five main panes in the window. On the left hand side is a display of all functions in the toolbox arranged in the same hierarchy as the PSSToolboxes folder. Scripts, including most of the demos, are not included. Below the hierarchical list is a list in alphabetical order by module. On the 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.

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)
```

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

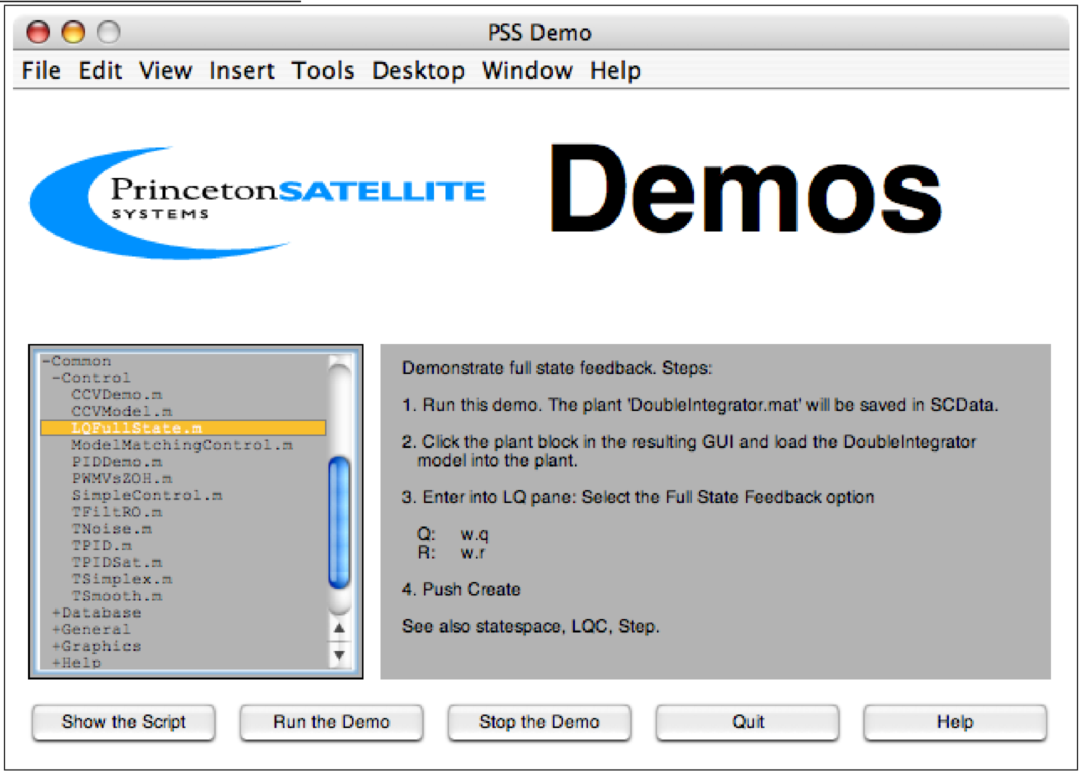
This is the search string. Spaces will be matched so if you type “attitude control” it will not match “attitude control” (with two spaces.)

2.5 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.

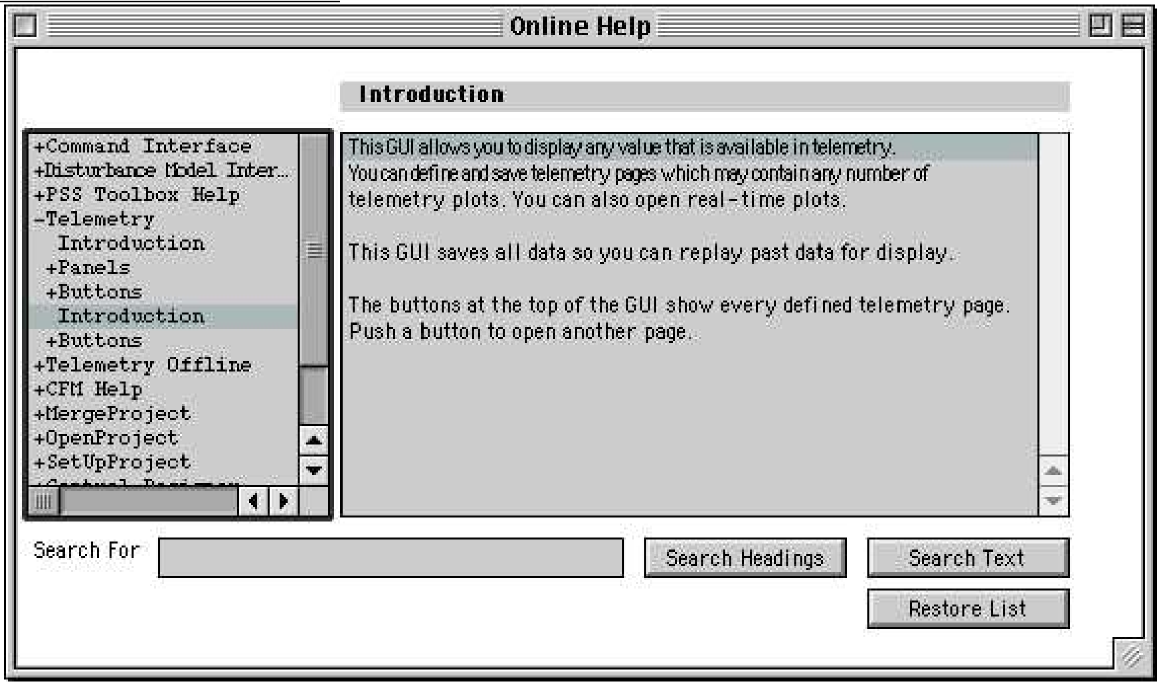
You can also access the published version of the demos using MATLAB’s help system. On recent versions this is access by selecting Supplemental Software from the main Help window, and then selecting Examples.

2.6 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 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 the Search For edit box

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.

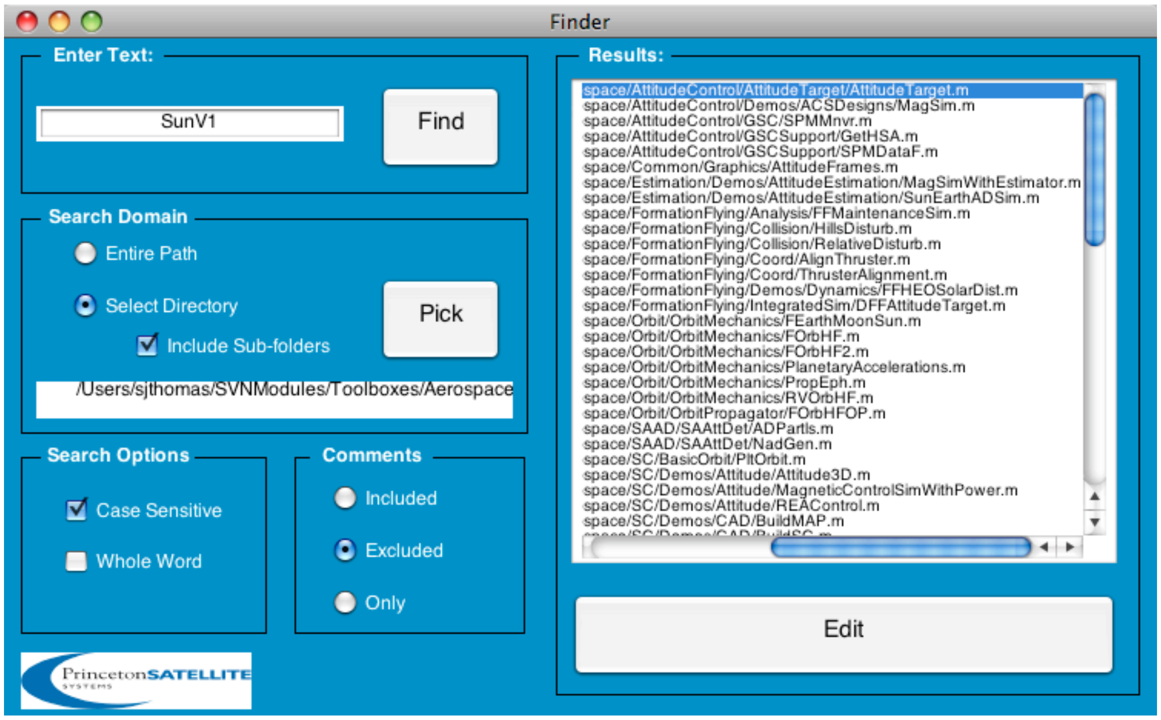
The buttons at the top of the GUI show every defined telemetry page. Push a button to open another page.

and hitting the appropriate button. Restore List restores the list window to its previous configuration.

2.7 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

```
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
```

2.8 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

WORKFLOW

This chapter shows you how the Spacecraft Control Toolbox fits into a spacecraft design workflow.

3.1 Overall Workflow

Figure 3.1 shows the workflow using PSS products. Preliminary work including disturbance budgets, pointing budgets, control system design and simulation are done using the Spacecraft Control Toolbox entirely in the MATLAB environment. MATLAB code is converted to C/C++ code using MatrixLib.

Figure 3.1: Workflow. Work progresses from MATLAB to flight.

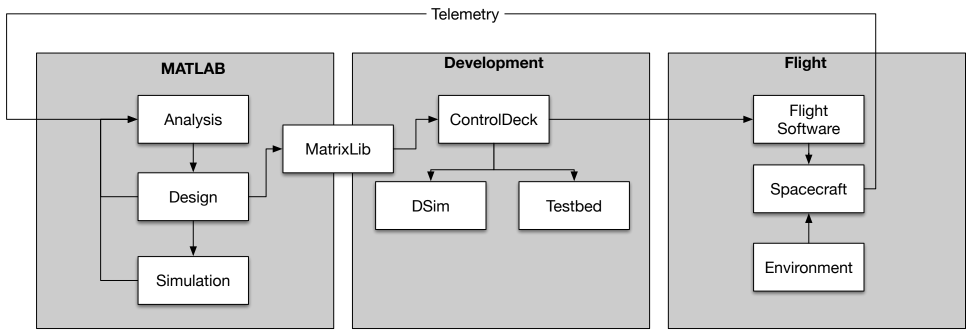


Figure 3.2 on the following page shows a simulation of the NASA Messenger spacecraft near Mercury. This was used as part of a NASA Phase II SBIR to develop and optical navigation system. The single axis testbed used for a U.S. Army satellite contract is also shown in the figure. It includes a simulation of an "Earth-Fixed" spacecraft. This allows checking out of the control system including star camera, sun sensor, magnetometer, GPS and RWA. Stellarium is used to generate star fields. The Army satellite is a 30 cm cube.

3.2 Working in the Spacecraft Control Toolbox

Figure 3.3 on the next page shows the workflow within MATLAB. The entire spacecraft can be designed. Figure 3.4 on the following page shows a typical CAD model done in MATLAB. The layout can be done in SCT. The CAD functions will compute the total mass and inertia. Surface models can be produced for use by disturbances.

Figure 3.5 on page 23 shows a complete GN&C system. Table 3.1 on page 23 gives the SCT functions used to

Figure 3.2: All software simulations and testbed. The testbed interacts with a simulation on the MacBook Pro.

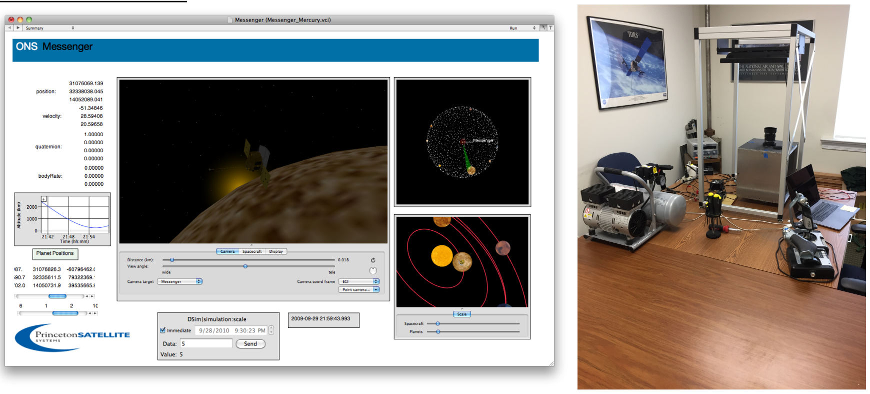


Figure 3.3: MATLAB SCT workflow. SCT provides all of the software needed to analyze and design spacecraft.

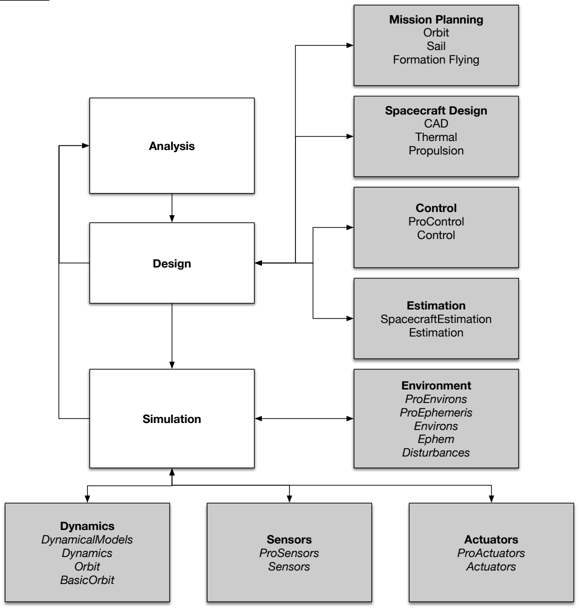
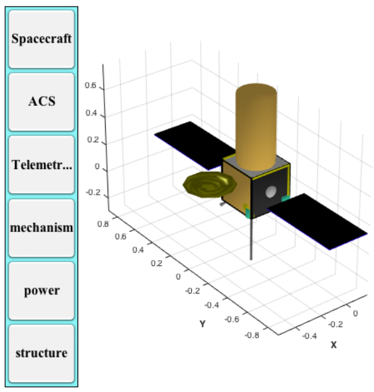
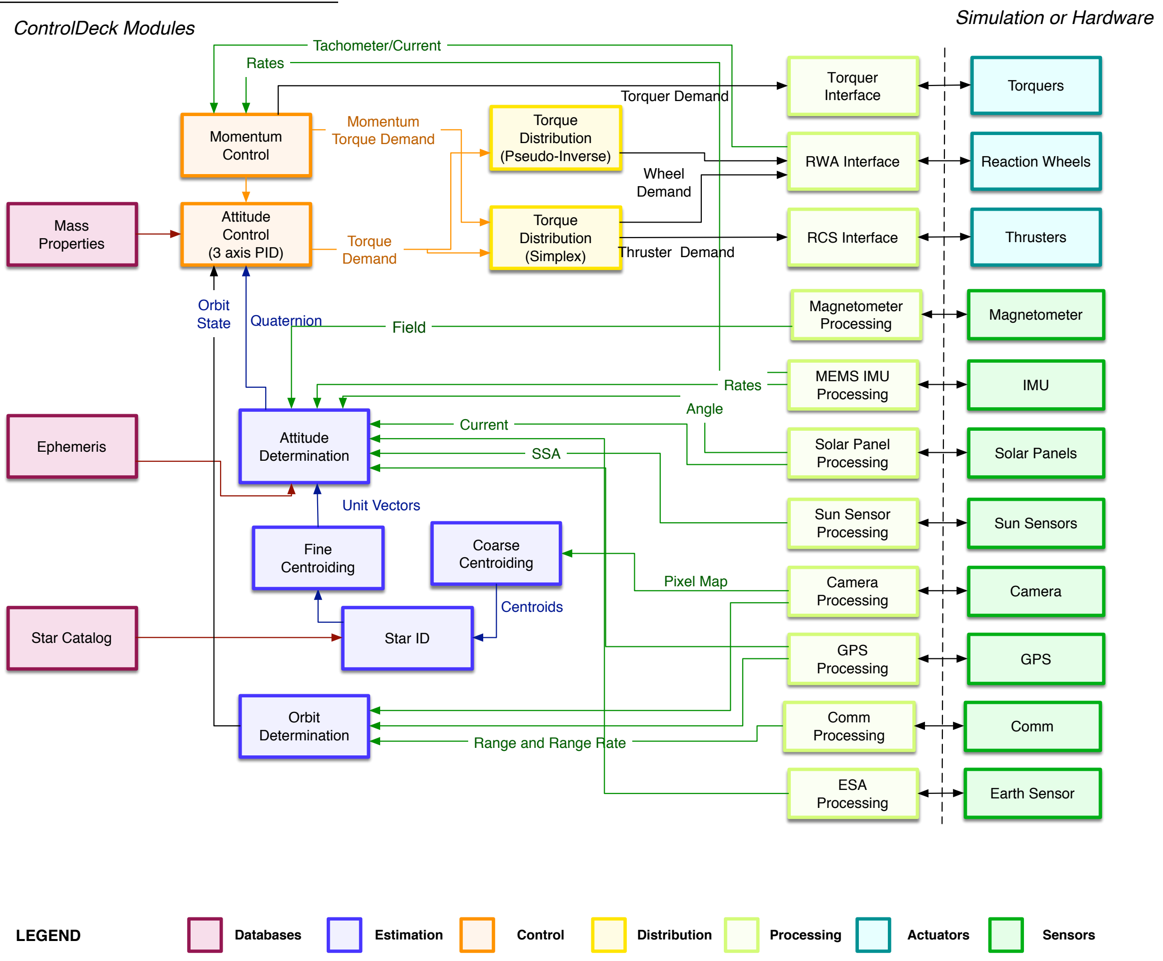


Figure 3.4: Typical CAD model.



implement the system. Note that you don't have to build a complete system in MATLAB. You can work with selected blocks by making a simulation script that just addresses those functions. There are multiple functions that can be used in nearly every block of the diagram. The control system uses state space matrices to implement the PID controller.

Figure 3.5: ACS system with autonomous navigation.



Other control design options include SISO, Eigenstructure assignment, phase plane and quadratic regulators.

Table 3.1: SCT ACS for a 3 axis control system . The row colors correspond the block colors in Figure 3.5. Items with "None" have simple interfaces that do not require a function.

Block	MATLAB function
MassProperties	MassBudget .m
Ephemeris	SolarSysJPL.m
Star Catalog	LoadCatalog.m
Momentum Control	MagControlTwoAxis.m
Attitude Control	PID3Axis.m
Attitude Determination	StellarAttDetUKF.m
Fine Centroiding	CentroidCOM.m
Coarse Centroiding	CoarseCentroids.m
Star ID	StarIDPyramid.m
Orbit Determination	OpticalNavPlanetStar.m
Torque Distribution (Pseudo Inverse)	pinv.m
Torque Distribution (Simplex)	simplex2.m
Torquer Interface	None

RWA Interface	None
RCS Interface	ThrusterCommand.m
Magnetometer Processing	None
MEMS IMU Processing	None
Solar Panel Processing	None
Sun Sensor Processing	SunSensor.m
Camera Processing	None
GPS Processing	GPSReceiver.m
Comm Processing	ToneRanging.m
ESA Processing	MeasEarthChord.m
Torquers	MagneticTorquer.m
Reaction Wheels	RWA.m
Thrusters	BloDown.m
Magnetometer	MeasMagnetometerEarth.m
IMU	RIGModel.m
Solar Panels	SolarPanelThermal.m
Sun Sensors	MeasSunSensorDigital.m
Camera	PinholeCamera.m
GPS	MeasGPS.m
Comm	PowerReceived.m
Earth Sensor	ESA.m

3.3 Porting Spacecraft Control Toolbox MATLAB to C++

3.3.1 Introduction

Flight software development can be done using the Princeton Satellite Systems ControlDeck C++ class library for implementing the control system, MatrixLib for porting MATLAB code and VisualCommander for simulating the spacecraft. The three products are discussed in the following sections.

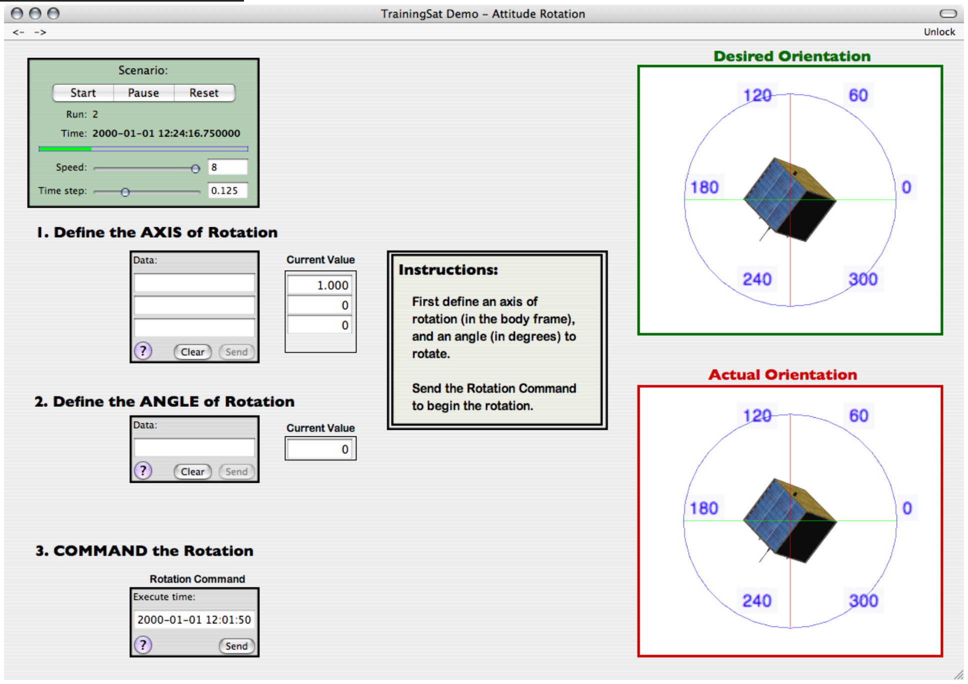
3.3.2 VisualCommander

VisualCommander is an extensible client-server application for data visualization, command, and control. Designed to seamlessly integrate multiple distinct operations (including missions, experiments, simulations, and others), VisualCommander provides reliable data storage, access control, command processing, and an extensive suite of built in visualization and analysis tools. VisualCommander was initially developed under Air Force SBIR funding. An example display from a satellite operator training application is shown in Figure 3.6 on the facing page.

VisualCommander receives data from and sends commands to external entities via a lightweight plugin interface. This approach enables virtually any system to be connected to VisualCommander with minimal programming effort. Examples of such systems include simulations, hardware experiments, space operations (telemetry), medical instruments, web sources, and more. Furthermore, VisualCommander can manage multiple active connections involving distinct, unrelated systems. For example, data from a hardware testbed could be simultaneously integrated with data from both a software simulation and an operational unmanned air vehicle (UAV). The data and command capabilities of each active plugin are combined within a unified “Data Hierarchy”. VisualCommander clients can, in turn, access this data hierarchy as a collective whole or by individual system.

The VisualCommander client application enables users to construct custom graphical user interfaces (GUIs) using a library of built in displays ranging from simple numeric to plots to three dimensional OpenGL visualization tools. Interfaces consist of one or more windows containing multiple, layered display “pages”. Displays are added to these

Figure 3.6: TrainingSat attitude display



Run: 2
Time: 2000-01-01 12:24:16.750000
Speed: 8
Time step: 0.125

Current Value: 1.000, 0, 0

Current Value: 0

Execute time: 2000-01-01 12:01:50

Instructions:

First define an axis of rotation (in the body frame), and an angle (in degrees) to rotate.

Send the Rotation Command to begin the rotation.

pages and then linked with data and/or commands from the data hierarchy, all via standard drag and drop actions. Extensive configuration functionality allows users to easily create a custom "look and feel" for each interface. Multiple screen configurations are easily leveraged via multiple display windows.

In addition to visualization and command, clients have powerful tools for dynamic data analysis known as "Data Processors". Data processors generate new data in the data hierarchy by operating on existing data points. VisualCommander includes a built-in data processor capable of interpreting a significant number of instructions. This can be used to perform extensive real-time calculations on data acquired from external systems.

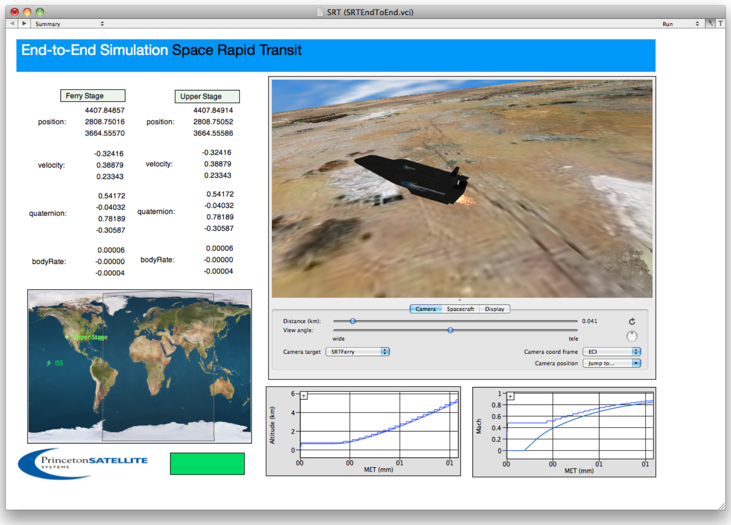
VisualCommander is designed, first and foremost, to be an extensible package. As stated above, lightweight plugins are used to connect external sources of data to the system. In addition, the client can be extended by creating new data display tools and data processors. Both of these component types are loaded by the client as plugins, and the application programming interfaces (APIs) for developing them have been designed to simplify use by customers and third-party vendors. In particular, existing OpenGL visualization tools can be encapsulated in VisualCommander display plugins with minimal effort.

External simulations can be connected to VisualCommander via the previously described plugin interface. In addition, VisualCommander includes a built-in simulation engine, DSim, which is capable of high-fidelity dynamics simulations. In DSim, simulations are constructed by connecting pre-defined models of various components, such as sensors, actuators, batteries, power networks, etc. New simulations can be created simply by modifying the configuration of existing simulations using included graphical tools for manipulation. In addition, completely new models can be added to the existing library programmatically, through an API for describing model properties and behaviors.

An example involving a spacecraft field-of-view is shown in Figure 3.7 on the next page. This shows the Space Rapid Transit launch vehicle under turbofan power and a telemetry page for the attached Upper Stage.

The orbiting satellite picture was created with VisualCommander graphics tools. VC provides a library of utilities that simplify the display of complex 3D objects. This display shows orbit position, attitude and thruster firings from a spacecraft simulation.

Figure 3.7: Space Rapid Transit vehicle simulation



	Ferry Stage		Upper Stage
position:	4407.84857 2808.75016 3664.55570	position:	4407.84914 2808.75052 3664.55586
velocity:	-0.32416 0.38879 0.23343	velocity:	-0.32416 0.38879 0.23343
quaternion:	0.54172 -0.04032 0.78189 -0.30587	quaternion:	0.54172 -0.04032 0.78189 -0.30587
bodyRate:	0.00006 -0.00000 -0.00004	bodyRate:	0.00006 -0.00000 -0.00004

Distance (km): 0.041
View angle:
Camera target: SRTFerry
Camera coord frame: ECI
Camera position: Jump to...

3.3.3 ControlDeck

ControlDeck is an object-oriented asynchronous control system framework. ControlDeck modules communicate through a robust messaging architecture. This provides tremendous flexibility when implementing complex multi-rate control systems. Modules can run off timers, based on events, or using background threads.

ControlDeck is organized into a three-level structure, shown in Figure 3.8 on the facing page. At the top is the ControlDeck framework itself. Below that are organizational groupings known as Systems, followed by the software modules. ControlDeck can easily interface with hardware and with the DSim simulation framework. DSim variables can be linked directly to ControlDeck variables. When hardware is in the loop, individual ControlDeck modules will handle interactions with that hardware. It's easy to use ControlDeck with hardware in the loop. It has been used with Phidgets USB components including motors, temperature sensors and PH sensors, and with Saitek flight simulator controllers, Labjack I/O Boards and other hardware.

Because the source of individual variables is unimportant to other modules wishing to make use of the variables, it is easy to gradually move from simulated to real hardware: as hardware is added into the loop, the variables representing that hardware in the simulation are removed from mapping, and instead a ControlDeck module interfacing with that hardware is written that creates variables with the same group and path. Other modules can interact with either simulated or 'real' variables without any changes.

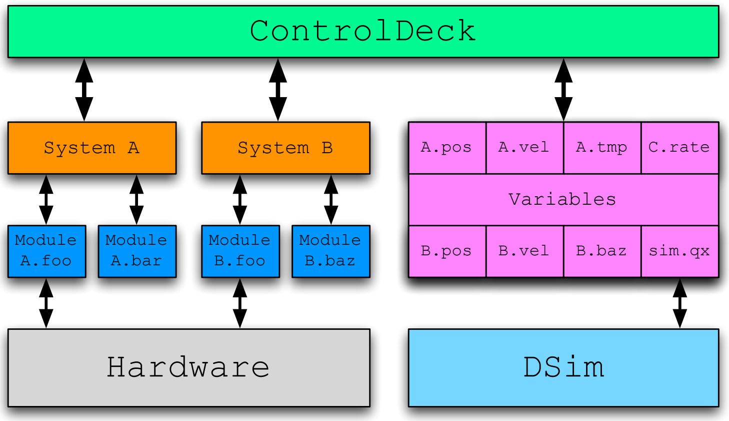
ControlDeck has been run on a Beagleboard for a CubeSat control application.. The board connected to a simulation running on a MacBook using TCP/IP. It also runs on MacOS X, Linux and Windows.

3.3.4 MatrixLib

Introduction

MatrixLib is a C++ implementation of matrix operations. Ideal for science and engineering applications, MatrixLib provides an easy to use C++ matrix class with an extensive body of functions and operations. MatrixLib links against LAPACK (Linear Algebra package) and BLAS (Basic Linear Algebra Subprograms) for many of its internal calculations providing efficiency.

Figure 3.8: ControlDeck. The structure provides a flexible and robust software architecture for multi-processor control applications. It can run on a Beagleboard, right.



Features

- *Cross Platform*: Supported on Windows NT/XP/2000, MacOS, Linux and Unix.
- *Networking Support*: Built in serialization functions exist for transferring matrix data between different architectures (big-endian, little endian) over a network interface.
- *BLAS/LAPACK Interface*: An interface to these libraries allow users to access standard. highly efficient routines in these libraries.
- *Display and Inspection*: Powerful display functions exist to view the matrix and control the precision of display.
- *Built in Debugging*: MatrixLib protects against invalid operations by checking matrix dimensions, element values, data pointers, etc. Each matrix has an internal error field which is flagged when invalid operations are attempted. In addition, it provides warning messages and stack tracing tools for locating the precise point at which an invalid operation originated.
- *Ease of code conversion from Matlab to C++*: Has hundreds of C++ functions for doing matrix operations with function calls similar to Matlab.
- *Free Demo*: A demo version is available for free download complete with API and sample code.

Function categories

The functions contained in MatrixLib can be grouped under the following categories:

- *Construction and I/O*: Packaging to string and binary formats as well as a display to standard output.
- *Matrix Manipulation*: Easy extraction and incorporation of sub-matrices, stack and append operations, dynamic resizing etc.
- *Inspection and Assignment*: Functions for comparing matrices, locating elements in a matrix, sorting the elements, applying functions on an element-by-element basis etc.
- *Arithmetic*: Standard arithmetic operations such as addition and multiplication, both matrix-matrix and matrix-scalar.
- *Algebraic operations and Trigonometry*: Standard algebraic and trigonometric operations on each element of the matrix.

- *Linear Algebra*: Functions include transposes, inverses, singular value decompositions, simplex methods, linear solvers and more.

3.3.5 MATLAB and C++ Example

MatrixLib can speed porting from MATLAB to C++. ControlDeck modules typically use C++ classes defined in MatrixLib and in the SCControl C++ libraries.

This is a MATLAB function from SCT:

```
function r = LatLonAltToEF( x, f, a )
eSq = f*(2-f);
sPhi  = sin( x(1,:) );
d  = sqrt( 1 - eSq*sPhi.^2 );
xC = (a./d + x(3,:)).*cos(x(1,:));
zC = ((1-eSq)*a./d + x(3,:)).*sPhi;
c = cos( x(2,:) );
s = sin( x(2,:) );
r = [xC.*c;xC.*s;zC];
```

This is the C++ version from the PSS SCControl library. It returns an `ml_matrix` object.

```
ml_matrix lat_lon_alt_to_pos(double lat, double lon, double h, double f, double a
    )
{
        double eSq = f*(2.0 - f);
        double c = cos(lat);
        double s = sin(lat);
        double xC, zC;
        double q = a/sqrt(1 - eSq*s*s);
        xC = (q+h)*c;
        zC = (q*(1 - eSq) + h)*s;
        ml_matrix pos(3,1);
        pos(1,1) = xC*cos(lon);
        pos(2,1) = xC*sin(lon);
        pos(3,1) = zC;
        return pos;
}
```

CHAPTER 4

BASIC FUNCTIONS

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

4.1 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 245.

4.2 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 4.1.

Table 4.1: 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 = 3.98600436e5)

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

 --------------------------------------------------------------------------
   References: Battin, R.H., An Introduction to the Mathematics and
```

```
                 Methods of Astrodynamics, p 128.
-------------------------------------------------------------------------------
```

4.3 Function Features

4.3.1 Introduction

Functions have several features that are helpful to understand. Features that are available in the functions are listed in Table 4.2.

Table 4.2: Features in Spacecraft Control Toolbox functions

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

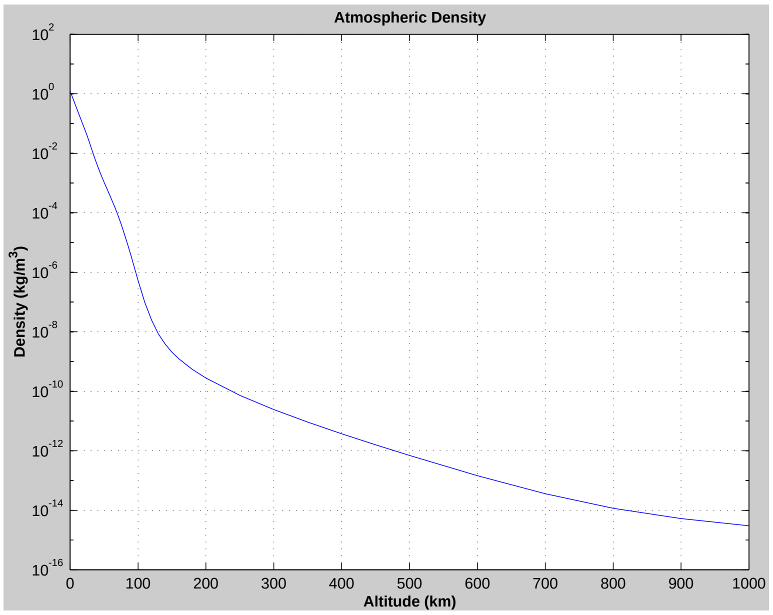
These are illustrated in the examples given below.

4.3.2 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 4.1. 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 4.1: 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.

4.3.3 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

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

%% 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

if( isempty( T ) )
  T = JD2T(Date2JD);
end

% Find Greenwich Mean Sidereal Time
gMST = GMSTime(36525*T + 2451545);

% Add the equation of the equinoxes to get apparent sidereal time
if( nargin == 2 )
  gAST = gMST + EOfE( T );
else
  gAST = gMST;
end

gAST = (pi/180)*gAST; % Convert to radians

sGAST = sin( gAST );
cGAST = cos( gAST );

g     = [cGAST, sGAST, 0; -sGAST, cGAST, 0; 0, 0, 1];

%--------------------------------------
% $Date: 2017-05-05 14:28:24 -0400 (Fri, 05 May 2017) $
% $Revision: 44494 $
>>
```

where we have excerpted the code creating the default input.

4.3.4 Built-in plotting

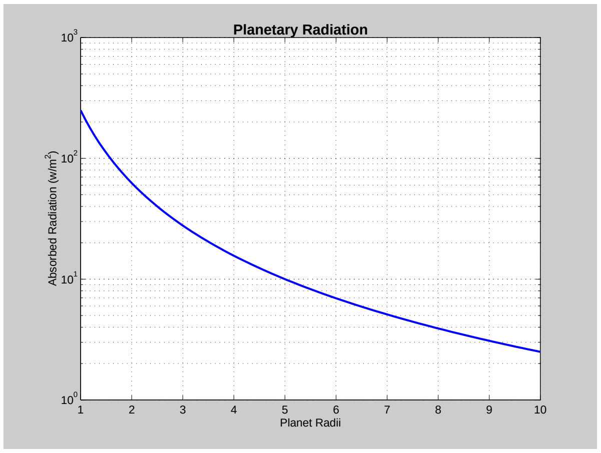
Many of the functions in the toolbox will plot the results if there are no output arguments. In many cases, you do not need any input arguments to get useful plots due to the built in defaults, but you can also generate plots with your own inputs. Calling for example `EarthRad` by itself will generate a plot of the absorbed flux per unit area as shown in Example 4.1 on the next 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.

Example 4.1 Earth radiation

```
EarthRad
```



4.3.5 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.

4.3.6 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

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

The options are listed in the header.

4.4 Example Functions

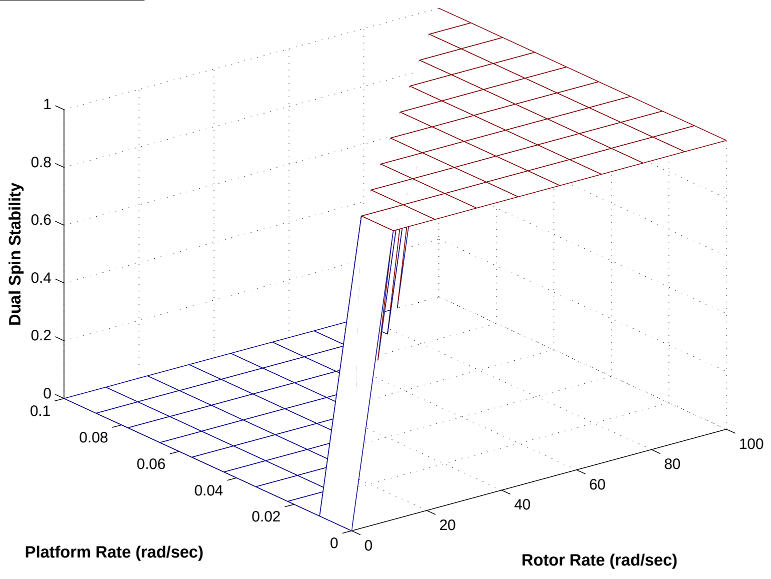
The following sections gives examples for selected functions from the major folders of the SCT core toolbox.

4.4.1 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 )
```

Figure 4.2: Dual Spin Stability

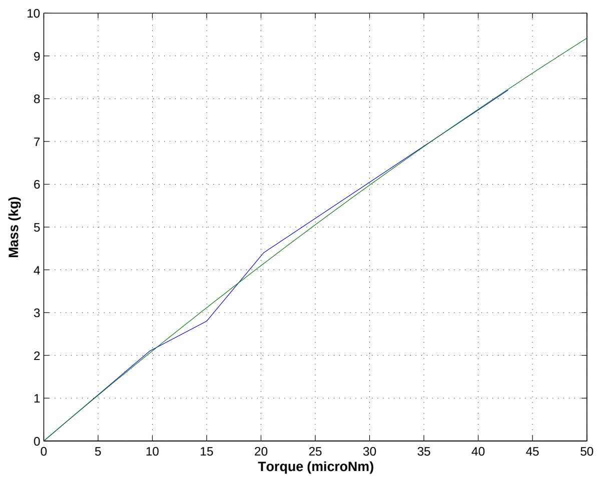


The result is `dSS = 0` which means the system is unstable! A stability plot is shown in Figure 4.2. One means stable. `DSpnStab` is a typical function in SCT. If no outputs are specified it generates a plot.

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 4.2 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.

Example 4.2 Torquer and thruster comparison results

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



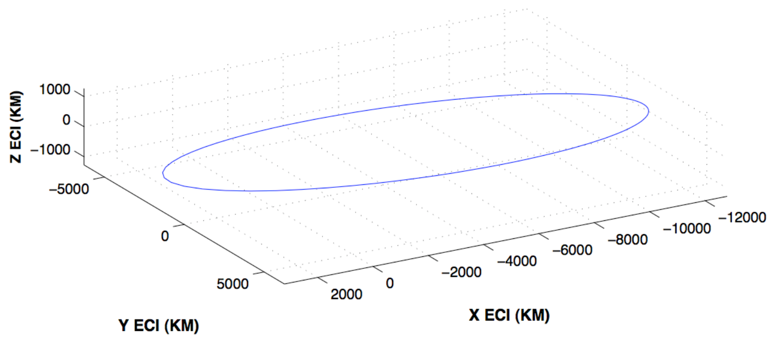
4.4.2 Basic Orbit

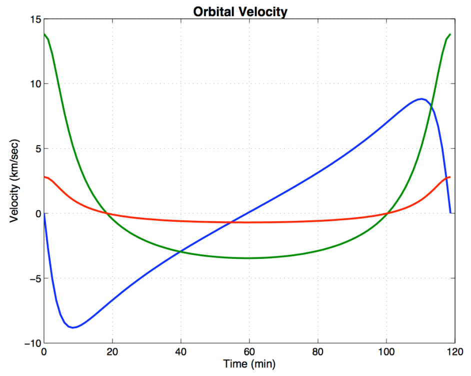
`RVFromKepler` uses Kepler's equation to propagate the position and velocity vectors. The output of the demo is shown in Figure 4.3 on the following page.

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

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

Figure 4.3: Elliptical orbit from `RVFromKepler`





```
r =
        3200
           0
           0

v =
           0
  13.83596536017765
   2.80468902945837
```

4.4.3 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
```

4.4.4 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 4.3 on the next page.

A Hall thruster can be drawn with `HallThrusterModel` as shown in Example 4.4 on the facing page.

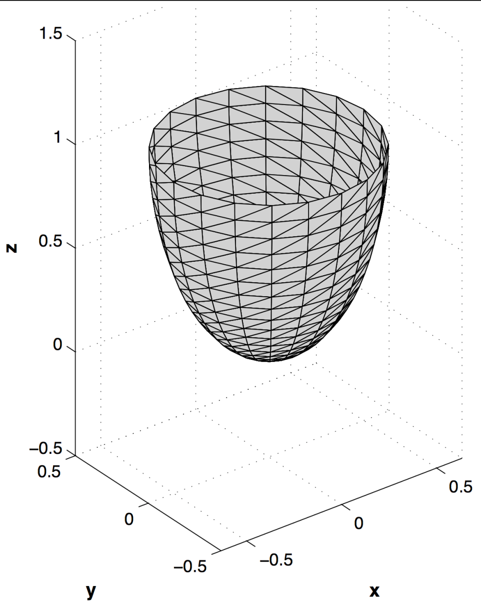
4.4.5 Control

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 )
```

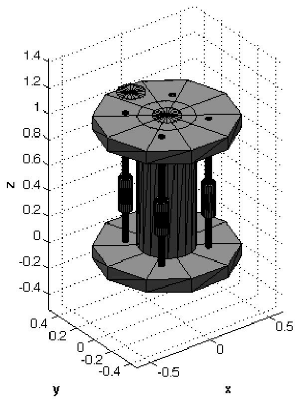
Example 4.3 Antenna patch

```
AntennaPatch
```



Example 4.4 Hall thruster

```
HallThrusterModel
```



```
ans =
                                  1                         0.5
                                  0                           1
```

4.4.6 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.

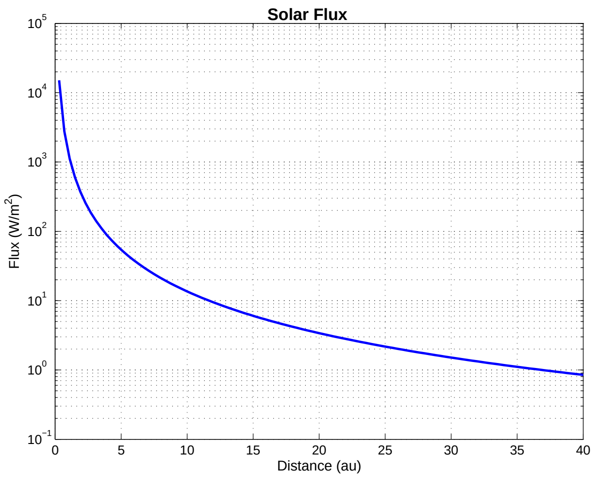
```
ans =
        0 + 0.00007291000000i
        0 - 0.00007291000000i
        0 + 0.03994167200000i
        0 - 0.03994167200000i
        0
        0
```

4.4.7 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 4.5.

Example 4.5 Solar flux from the sun

```
SolarFlx
```



4.4.8 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
```

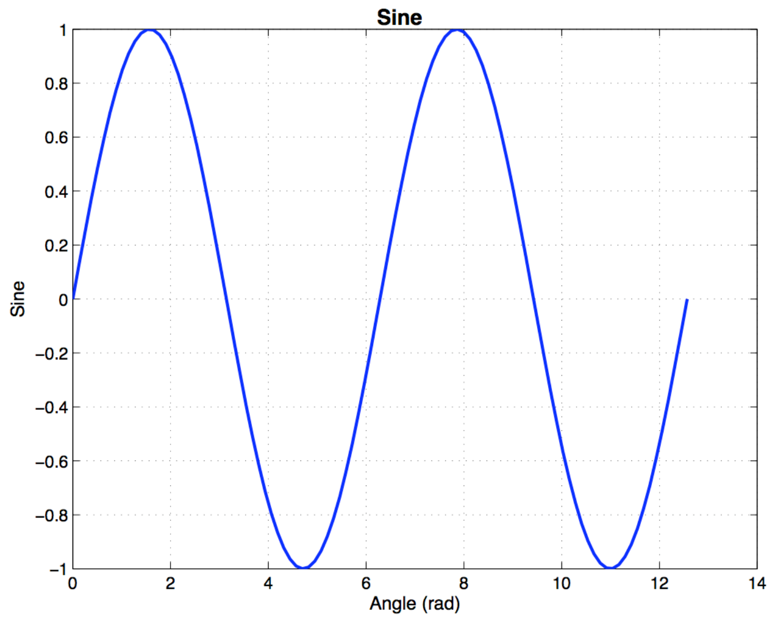
which returns a unit vector to the sun from the spacecraft and the distance from the origin to the sun. `SunV2` provides the same inputs and outputs and uses a higher precision sun model.

4.4.9 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 4.4.

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

Figure 4.4: A sine wave using `Plot2D`



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

4.4.10 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

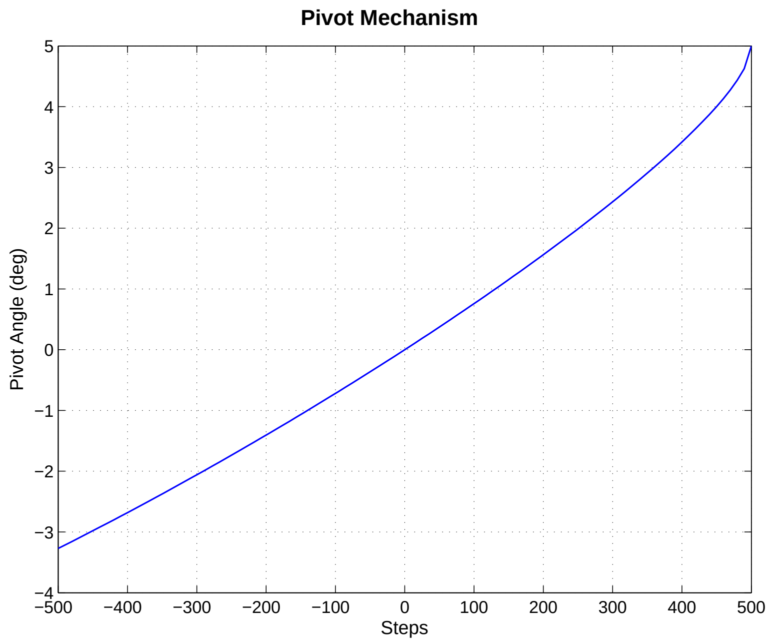
```
>> PivotMch
```

you get a plot of a pivot mechanism angle versus pivot steps (Figure 4.5 on the next page).

4.4.11 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.

Figure 4.5: Pivot mechanism



```
>>  jD  = Date2JD
JD2Date(jD)
jD =
     2.458844133015539e+06
ans =
   1.0e+03 *
   2.019000000000000   0.012000000000000   0.026000000000000   0.015000000000000
         0.011000000000000   0.032542612552643
```

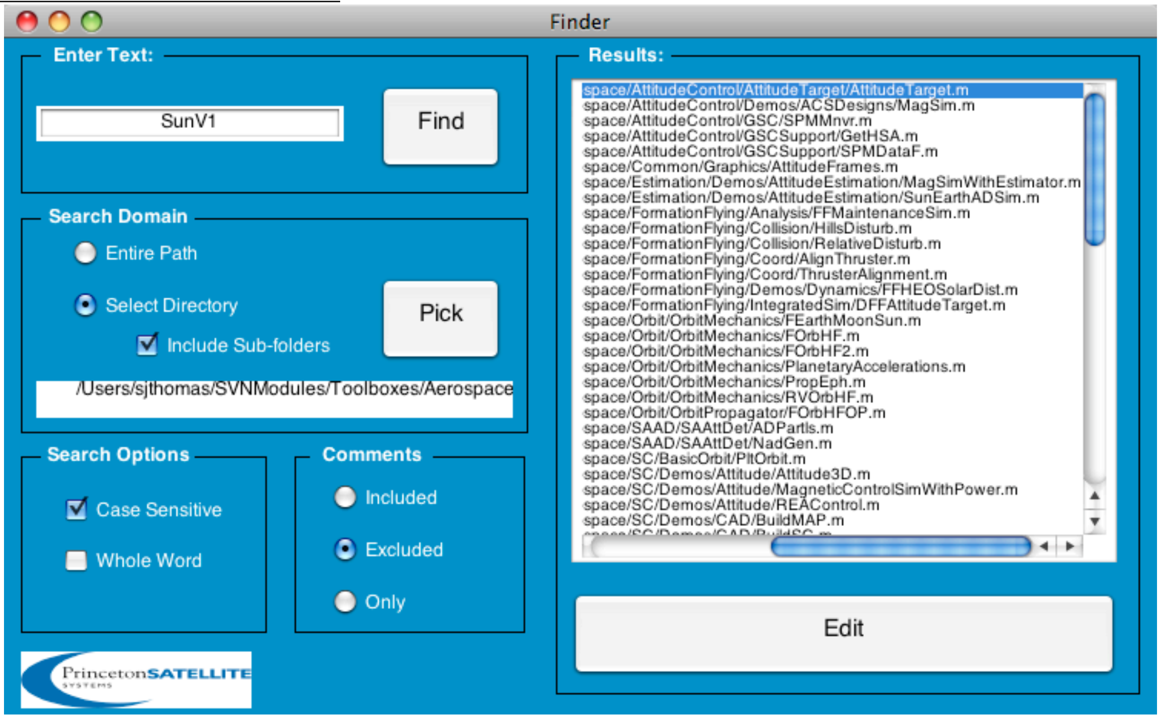
4.5 GUI Tools in the Toolbox

4.5.1 Finder GUI

The `Finder` GUI, shown in Figure 4.6, 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 4.6: 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 5

SPACECRAFT CONTROL TUTORIAL

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

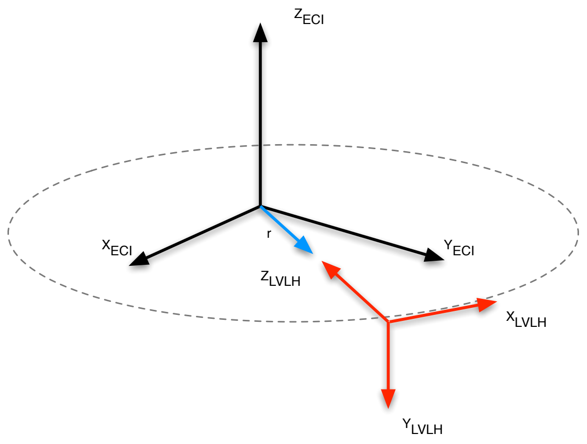
Step 1 - Dynamics: We start with the general kinematics and dynamics.

$$\dot{q} = f(q,\omega) \tag{5.1}$$

$$T = I\dot{\omega} + \omega^{\times} I\omega \tag{5.2}$$

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 5.1.

Figure 5.1: 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.

$$\dot{\theta} = \begin{bmatrix} \omega_x + \omega_0\theta_z \\ \omega_0 \\ \omega_z - \omega_0\theta_x \end{bmatrix} \tag{5.3}$$

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

$$T = I\dot{\omega} \tag{5.4}$$

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

$$x_{k+1} = Ax_k + Bu_k \tag{5.5}$$
$$y_k = Cx_k + Du_k \tag{5.6}$$

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

$$y = K_p u + \frac{K_r s}{s + \omega_R} u + K_i \frac{u}{s} \tag{5.7}$$

The function takes the following inputs:

Table 5.1: 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,  d e l t a )
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

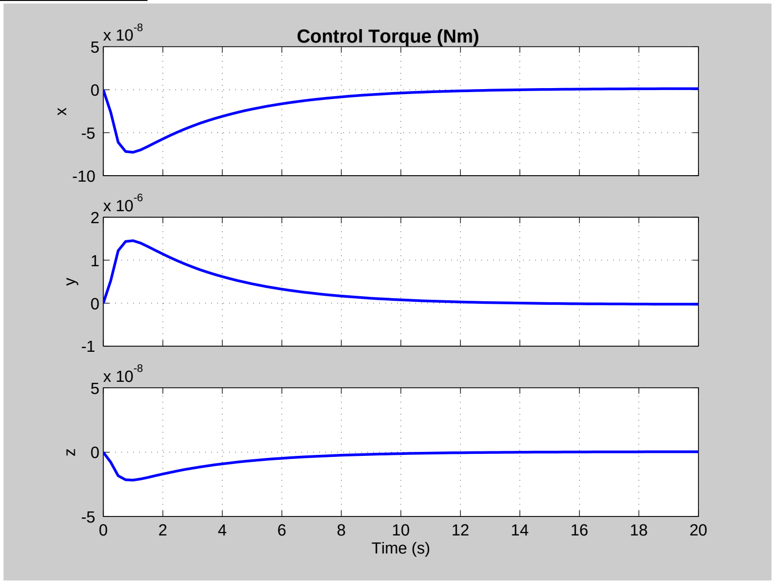
```
    accel(1) =          c*xRoll  + d*angleError(1);
    xRoll    = xRoll  + a*xRoll  + b*angleError(1);

    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);
```

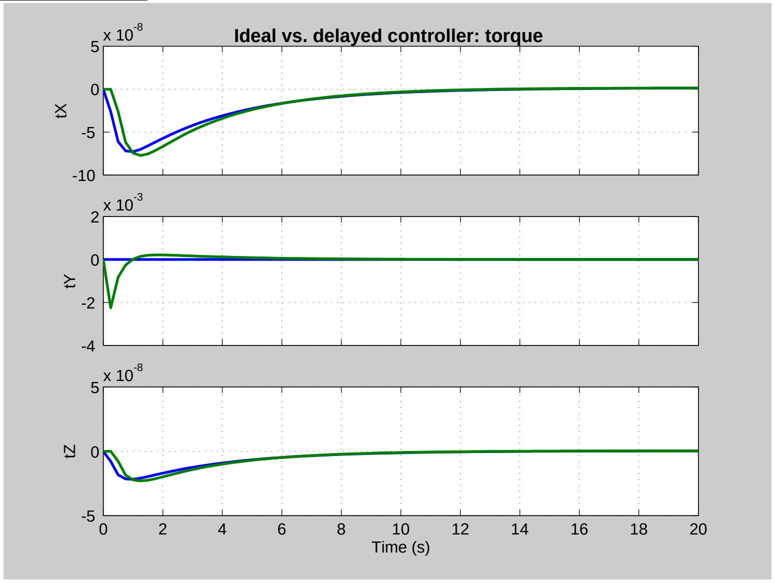
To test the controller, we give each axis an initial disturbance torque on the order of 1x10^{-6} Nm. The disturbance torque tDist is nonzero only for the first step. The resulting control torques are shown in Figure 5.2.

Figure 5.2: Control torques



Next, we simulate the controller again, this time with a delay equal to one sampling time. We see in Figure 5.3 on the next page that our controller does fine with this delay.

Figure 5.3: 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 6

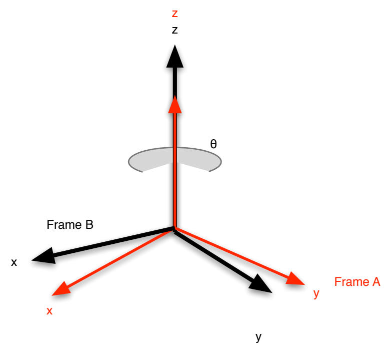
Coordinate Transformations

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*.

6.1 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 6.1: 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.

6.2 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 6.1 on the preceding 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)`.

6.3 Coordinate Frames

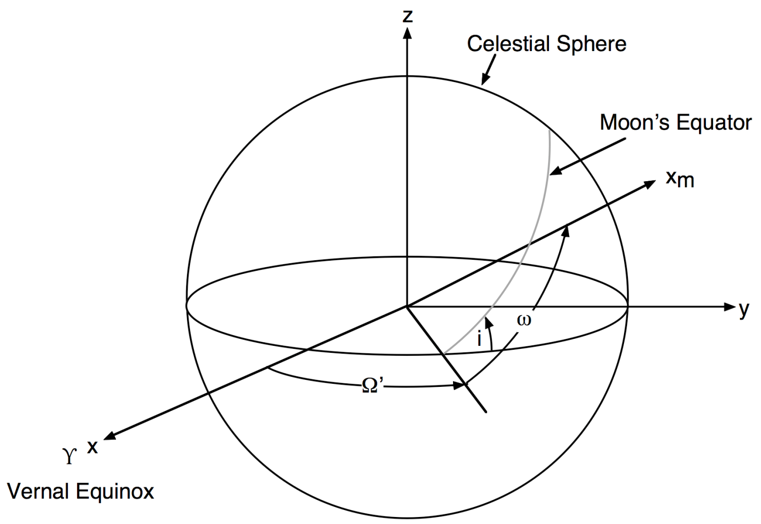
The toolbox has functions for many common coordinate frames and representations, including

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

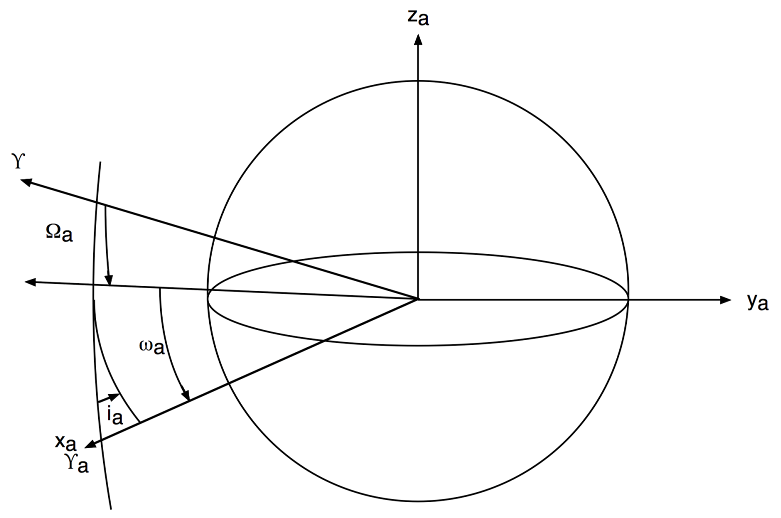
Figure 6.2: Selenographic to ECI frame



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 6.3 on the following 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 6.3: Areocentric frame



CHAPTER 7

USING CAD FUNCTIONS

7.1 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.

7.2 Building a Spacecraft Model Using **`BuildCADModel`**

7.2.1 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'
```

7.2.2 Geometry

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 7.1 on the next 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`.

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`.

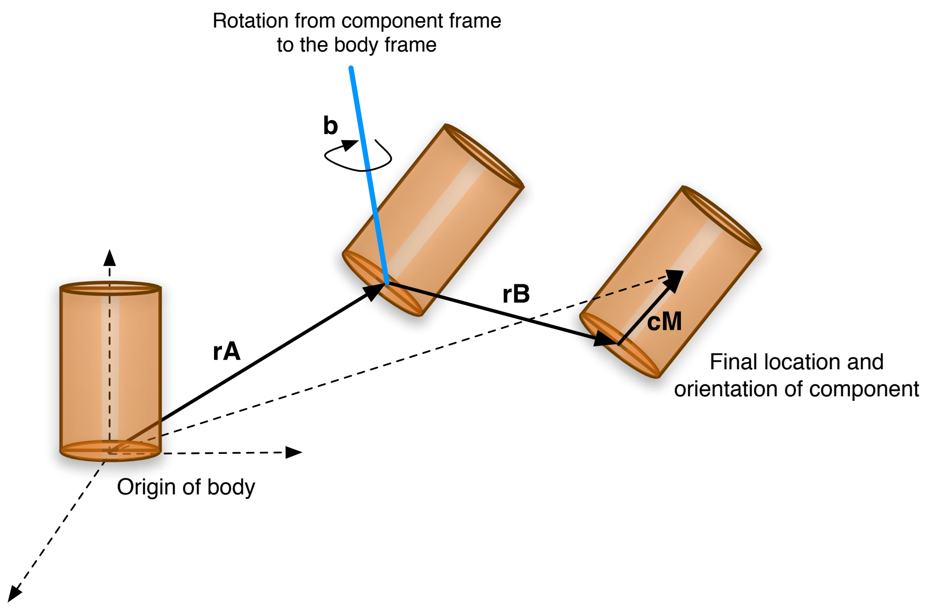
The bodies are defined in a tree configuration. Each body has a hinge vector defined in the frame of the previous body. The hinge rotation matrix also expresses the rotation into the previous body's frame. The center of mass of the body, computed by `BuildCADModel` from the included components, is referenced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

Figure 7.1: Component geometry and frames

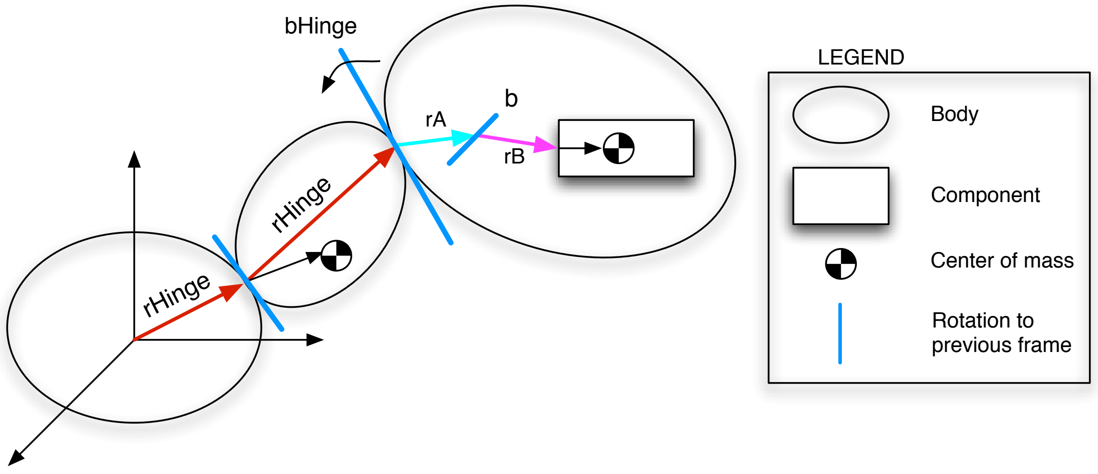


`cM` are in the component frame. The body's center of mass in the core frame is

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

Figure 7.2 shows the topological tree for three bodies with one component drawn.

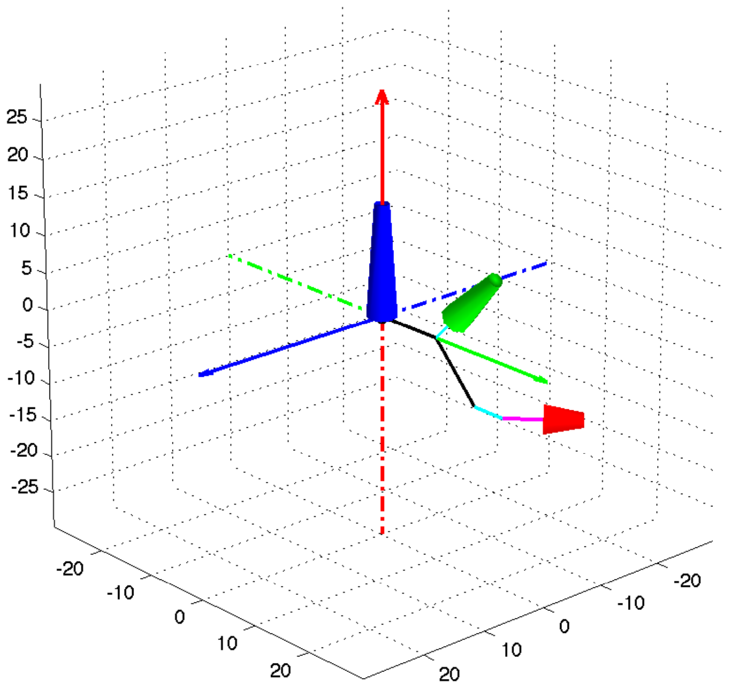
Figure 7.2: Body geometry and frames



See the example `BuildCylinders` for a systematic treatment of the vectors and rotations. The resulting plot is shown in Figure 7.3 on the next page.

The vectors and rotations described here are explained further in the sections on bodies and components later in this chapter, Section 7.2.5 and Section 7.2.6.

Figure 7.3: 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).



7.2.3 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.

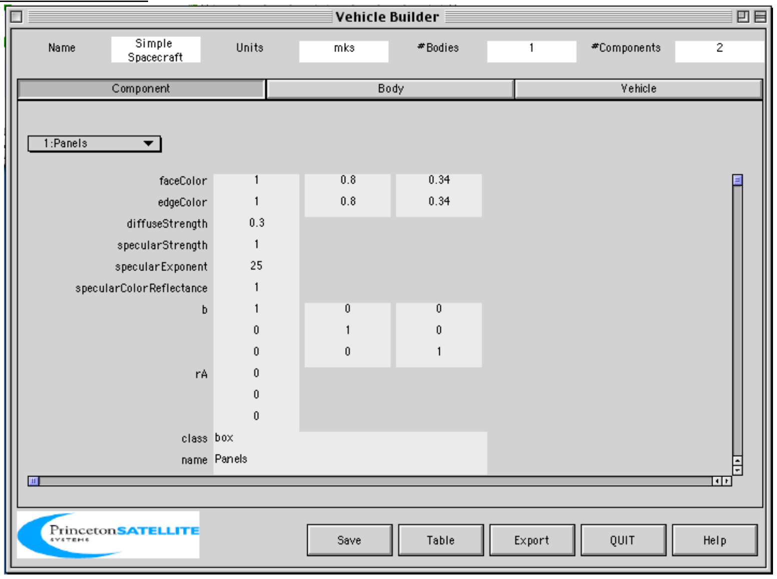
Listing 7.1: Simplest spacecraft CAD script *BuildSimpleSat.m*

```
%% 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'  );

% 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') )

%--------------------------------------
% $Date: 2019-08-05 16:08:47 -0400 (Mon, 05 Aug 2019) $
% $Revision: 49452 $
```

BuildSimpleSat.m

If you execute this script you will see Figure 7.4 on the following page. You can't change any properties using this

Figure 7.4: Simple Sat



Name	Simple Spacecraft	Units	mks	#Bodies	1	#Components	2

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		

GUI but you can view the properties at the component, body and vehicle levels. See Section 7.2.9 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
```

7.2.4 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.

7.2.5 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).

```

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

% Add spacecraft properties
%---------------------------
BuildCADModel( 'set␣name' , 'Simple␣Spacecraft' );
BuildCADModel( 'set␣units', 'mks'  );

% The core
%---------
q = CreateBody( 'make', 'name', 'Core' );
BuildCADModel('add␣body', q );
```

```
% 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 =
  1  7  cell array
    {'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.

7.2.6 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 (7.2.2) 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 7.2.3. 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.

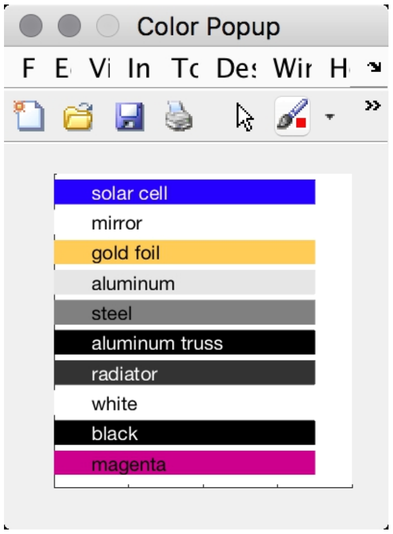
Perhaps the shortest component function call outside of the demo is this:

```
>> m = CreateComponent('make','sphere','radius',5)
m =
  struct with fields:

            name: 'no␣name'
           class: 'sphere'
              rB: [3  1  double]
               b: [3  3  double]
              rA: [3  1  double]
            body: 1
          inside: 1
    dataFileName: ''
           model: 'N/A'
    manufacturer: 'N/A'
      deviceInfo: []
               v: [26  3  double]
               f: [48  3  double]
         thermal: [1  1  struct]
            mass: [1  1  struct]
           power: [1  1  struct]
         optical: [1  1  struct]
        infrared: [1  1  struct]
            aero: [1  1  struct]
              rf: [1  1  struct]
        magnetic: [1  1  struct]
      propulsion: [1  1  struct]
        graphics: [1  1  struct]
              nV: [48  1  double]
               a: []
               n: []
               r: []
          radius: []
```

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 7.5 on the next page.

Figure 7.5: CreateComponent color display



7.2.7 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.

7.2.8 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'} );
```

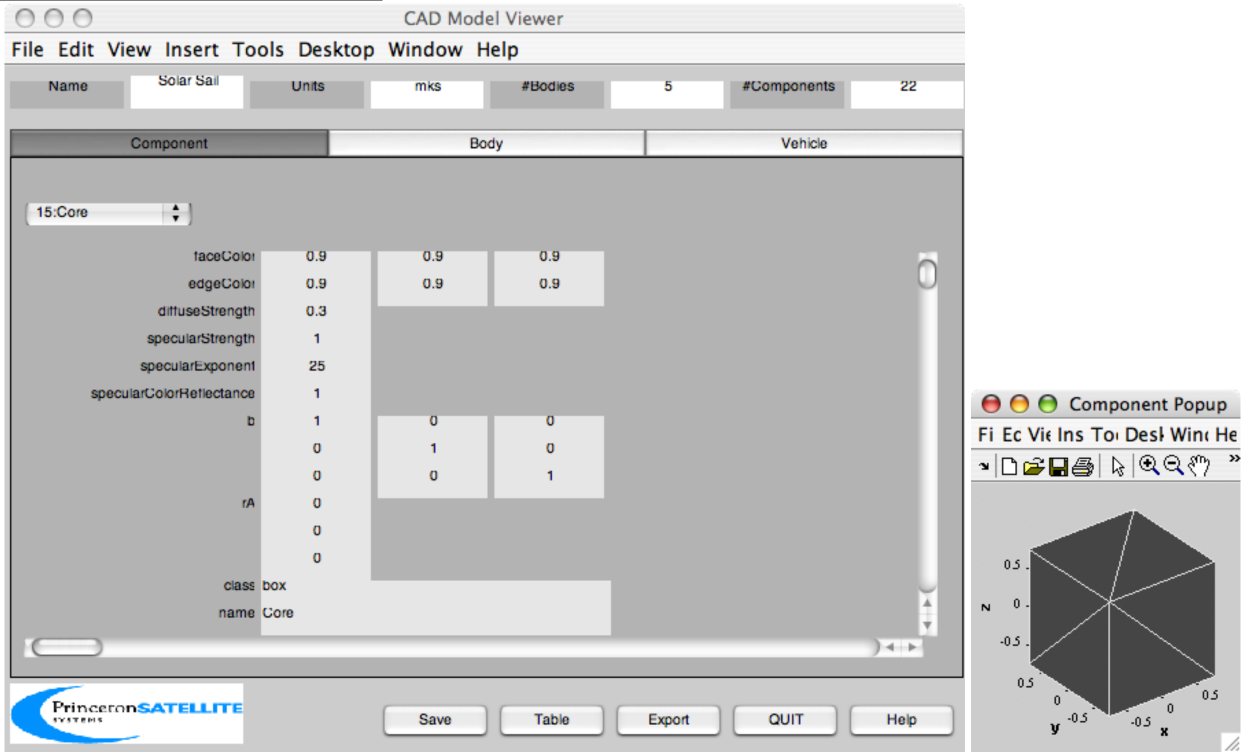
`BuildCADModel` searches for the strings in the cell array and associates them with the panel. If the string doesn't exist it is ignored. The result is a list of components stored with the panel name.

7.2.9 Viewing the CAD Model

When your CAD script is done, execute the script. The GUI shown in Figure 7.6 on the following 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

Figure 7.6: `BuildCADModel` Component View


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

Name Solar Sail Units mks #Bodies 5 #Components 22

15:Core

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		

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.

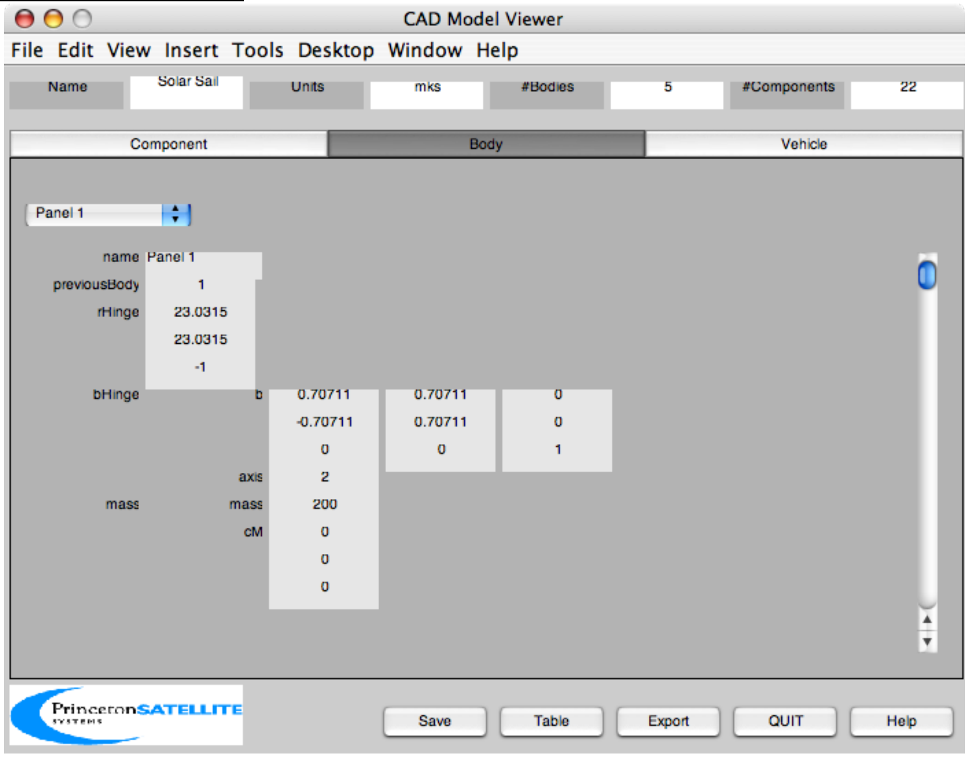
The subsystem table is shown below. It is stored in a text file using the name of the CAD model, in this case, Solar Sail.mat.

```
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
          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 7.7. 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

Figure 7.7: `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		

button you get a 3D window with the spacecraft and a list of components organized by body, see Figure 7.8 on the following page and Figure 7.9 on the next page.

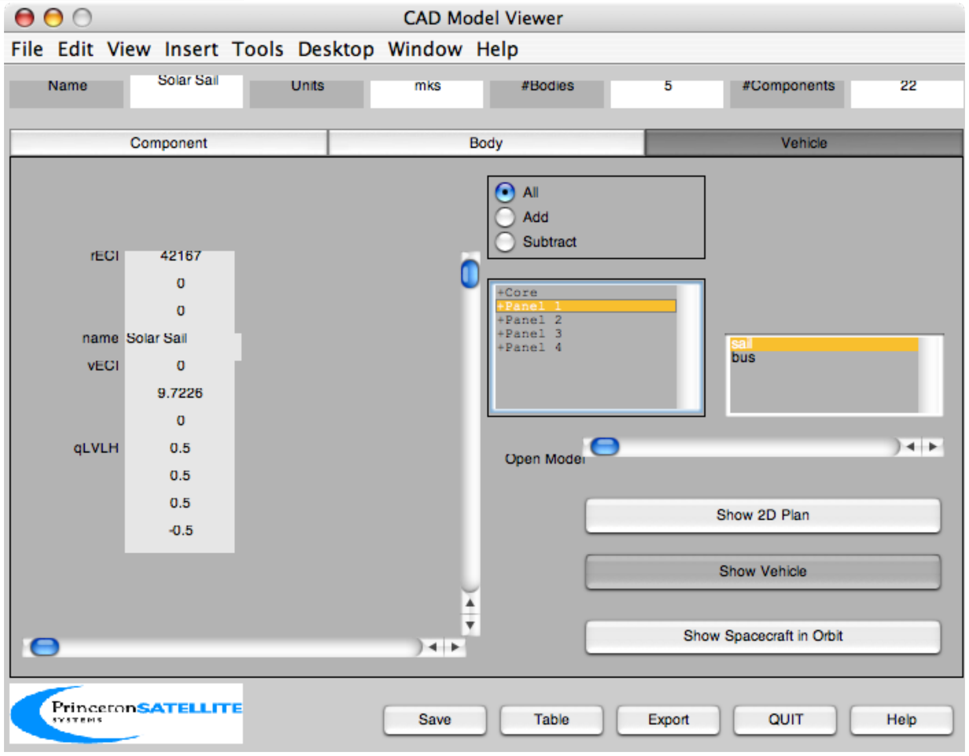
The radio buttons in the Vehicle tab select the mode. If All is selected all components are shown. If Add is selected, no components are shown and you add them by clicking on the component in the list. If Subtract is selected all components are shown and you remove components by clicking on the component 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 7.10 on page 69, 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:

```
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 7.9 on the next page. The quaternions set here are used only for display purposes.

Figure 7.8: Vehicle tab with Show Vehicle pushed



Name	Solar Sail	Units	mks	#Bodies	5	#Components	22

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

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

```
sail
bus
```

Figure 7.9: 3D View obtained by pushing Show Vehicle, left, and another sail model shown using Show Spacecraft in Orbit, right

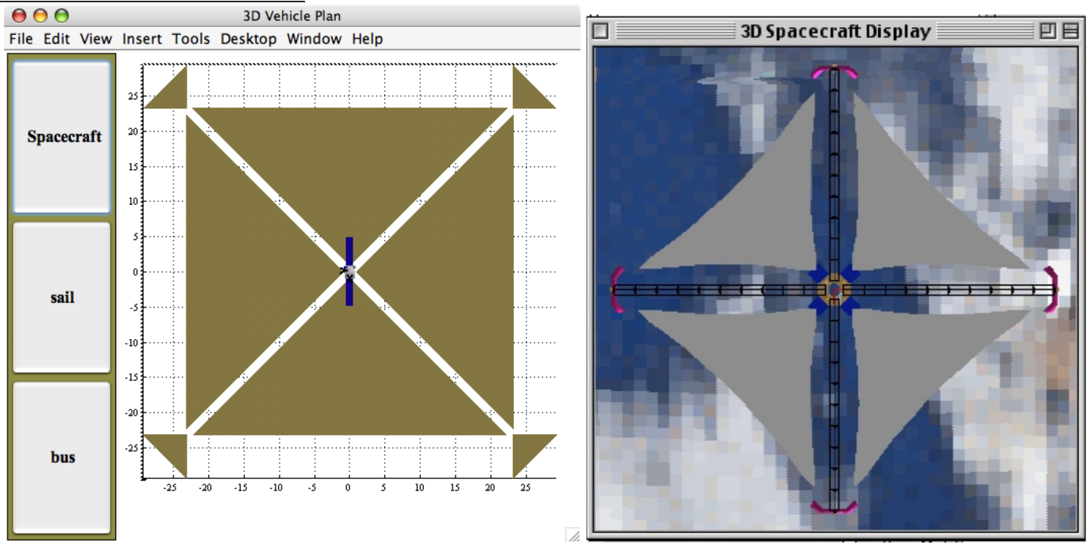
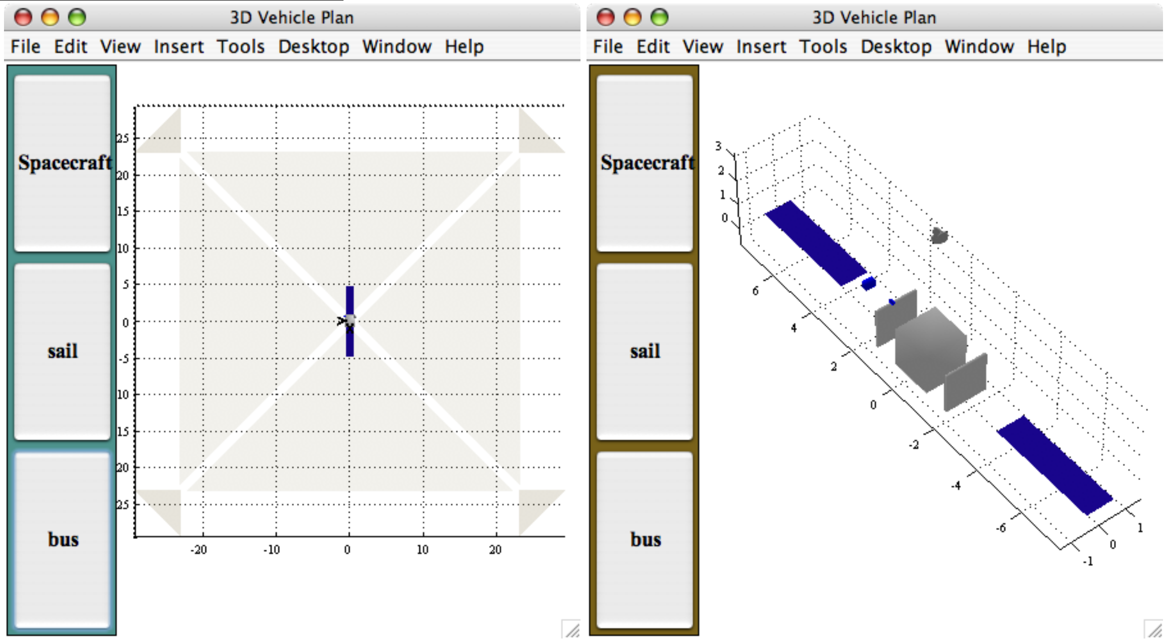
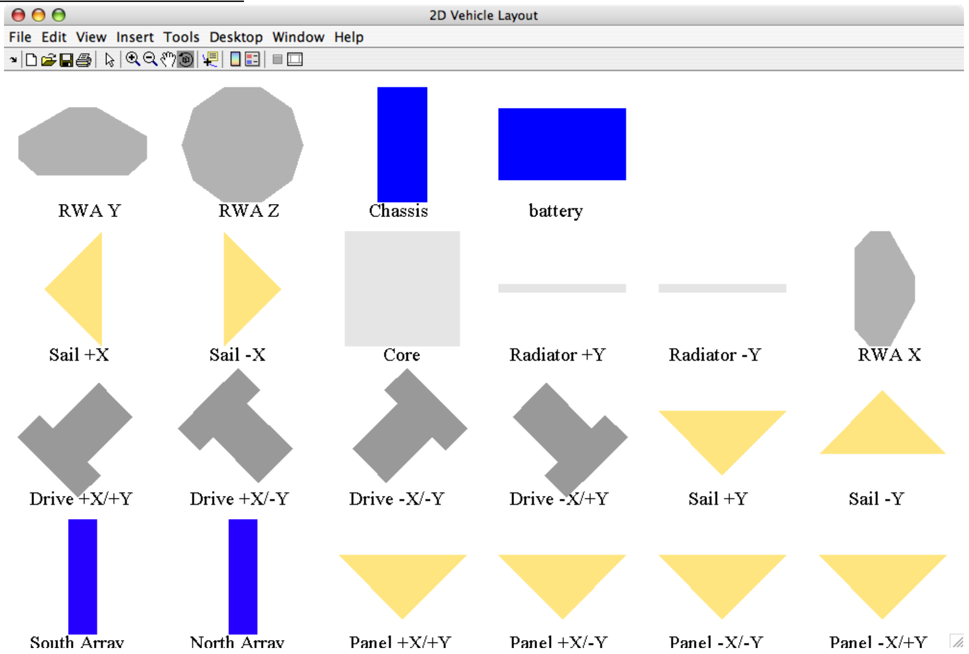


Figure 7.10: Subsystem Transparency and Exploded views


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

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 7.11: 2D View from clicking Show 2D Plan



7.2.10 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 7.12 on the facing 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]
```

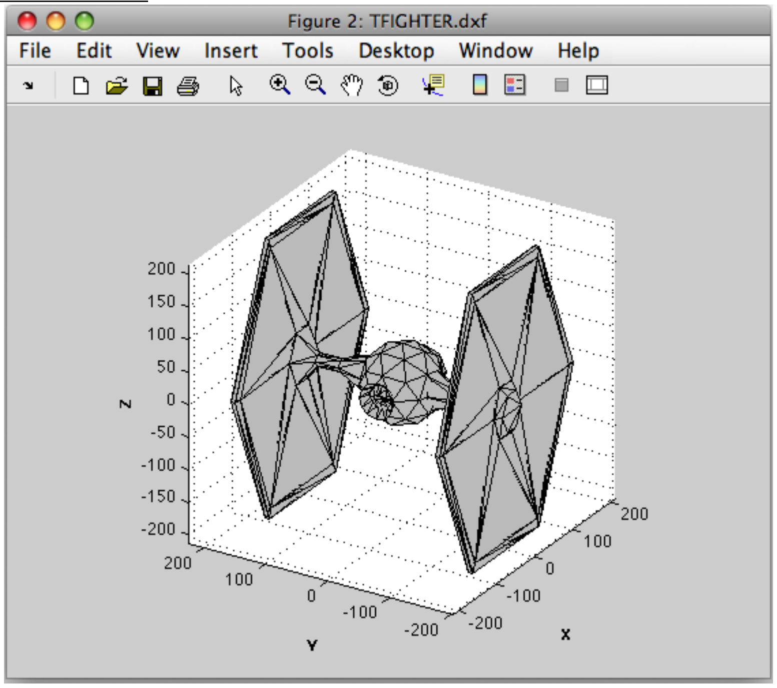
where you can see that there are 2464 vertices and 616 faces in the model. The Strength, Exponent, and Reflectance fields are for drawing in MATLAB only; they do not represent true optical properties are not used in disturbance modeling.

7.2.11 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
```

Figure 7.12: Tie fighter as loaded from a dxf file



```
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');
```

7.3 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:

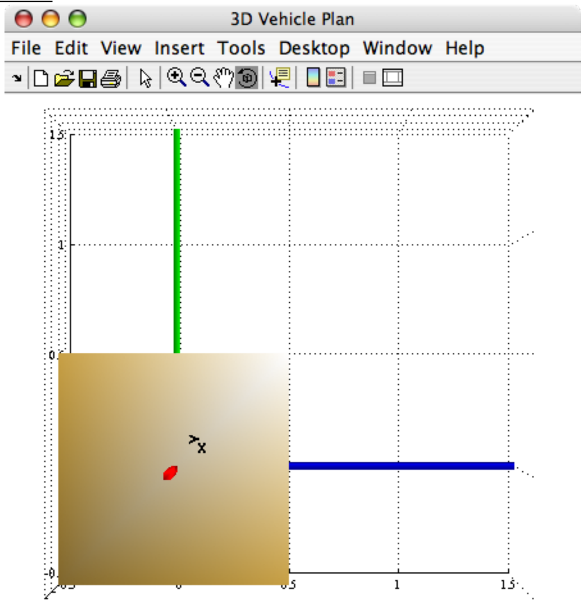
- ShowCAD
- DrawCAD
- DrawSC
- DrawSCPlanPlugIn
- DrawSCPlugIn

Each requires the CAD model to be loaded as a data structure as explained in Section 7.2.11. `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 7.13. This is the same view that can be obtained using the Show Vehicle button in the CAD model GUI.

Figure 7.13: `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

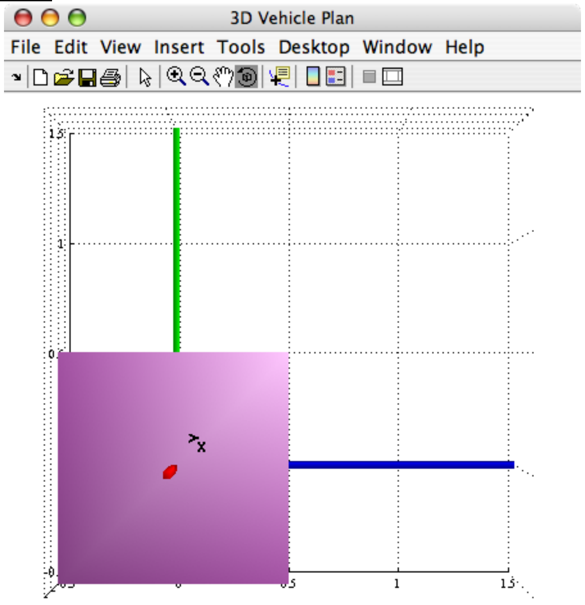
```
>> g.component(1)
ans =
                   faceColor: [1 8.000000000000000e-01 3.400000000000000e-01]
                   edgeColor: [1 8.000000000000000e-01 3.400000000000000e-01]
             diffuseStrength: 3.000000000000000e-01
            specularStrength: 1
            specularExponent: 25
    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 7.14.

Figure 7.14: `DrawSCPlanPlugIn` showing the XYZSat model after a change to `g`

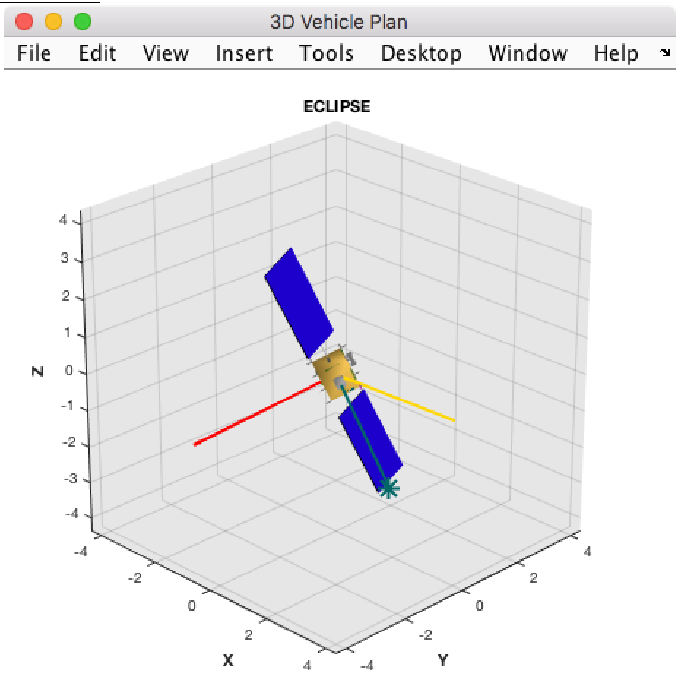


`DrawSCPlanPlugin` can display vectors, such as sun and nadir vectors, to aid in visualization. For an example this see `DisturbanceBatchDemo`. The code used to update the vectors is

```
>> 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 7.15: `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 7.2: 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 8

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.

8.1 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,

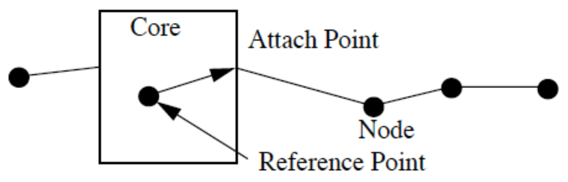
```
xDot = [QIToBDot( x(1:4), x(5:7)); RBModel( x(5:7), inertia, torque)];
```

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 8.1 on the next 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.

8.2 Wire Model

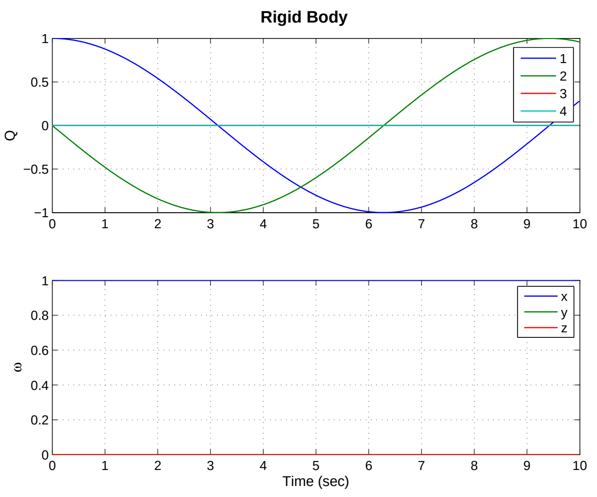
The wire model simulates a spacecraft with deployable wires as is illustrated in Figure 8.1. Two models are included.

Figure 8.1: Wire model



Example 8.1 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 8.1: 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 8.2: 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 8.3: 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 8.4: 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 8.5: 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 8.6: 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;

% 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 8.7: 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 8.8: 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.

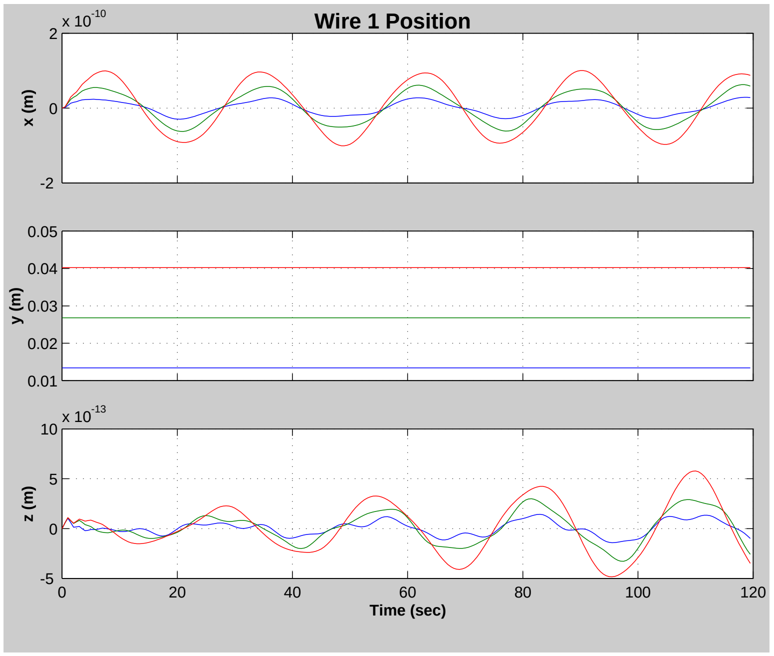
Listing 8.9: Wire Simulation: Plotting *WireSimG.m*

```
% 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 );

fprintf('Max␣momentum␣change␣=␣%12.4e␣with␣dT␣=␣%8.4f␣and␣tSim␣=␣%8.4f\n',max(abs(hPlot-1)), dT,
    tSim)
```

WireSimG.m

The results of this wire simulation are shown in Figure 8.2.

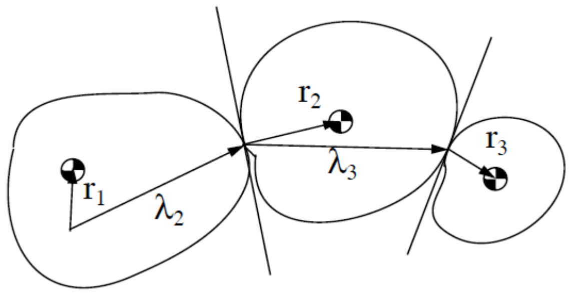
Figure 8.2: Wire simulation: Results



8.3 Tree Model

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

Figure 8.3: 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 8.10: 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 8.11: 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 8.12: 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 8.13: 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.

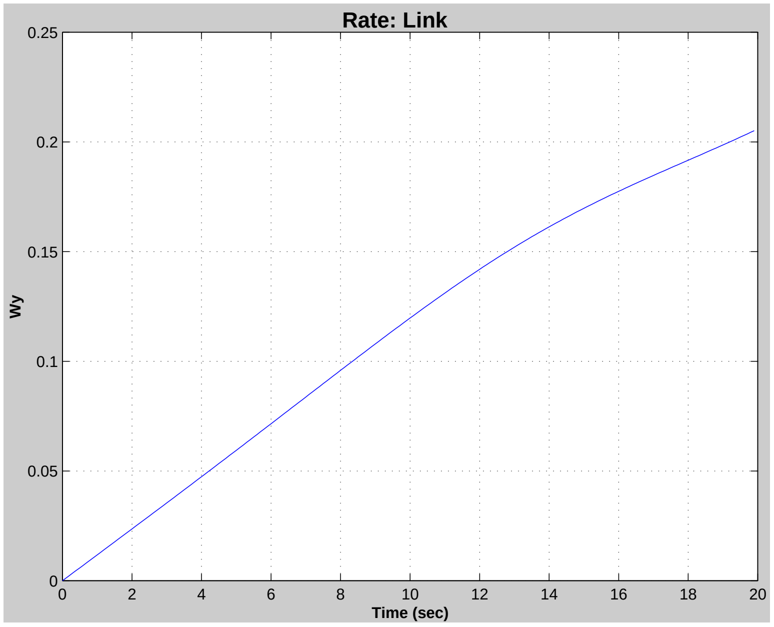
Listing 8.14: Tree Simulation: Plotting. *TreeSim.m*

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

TreeSim.m

An example plot is shown in Figure 8.4.

Figure 8.4: Tree Simulation: One of several plots.



CHAPTER 9

PLANETARY AND SUN EPHEMERIS

9.1 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 and sun-pointing. 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`.

9.2 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]
- `PlanetPosition`
- `SolarSys`
- `SunV1` [4]
- `SunV2` [5]

[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] The 1993 Astronomical Almanac, p. C24.
[5]Montenbruck and Pfleger, p. 36.

9.3 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.

9.3.1 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
```

9.3.2 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 9.1 on the next page: Additional computations are required to obtain planet states referenced from the sun, or to obtain the heliocentric positions of the Earth and moon.

9.3.3 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
```

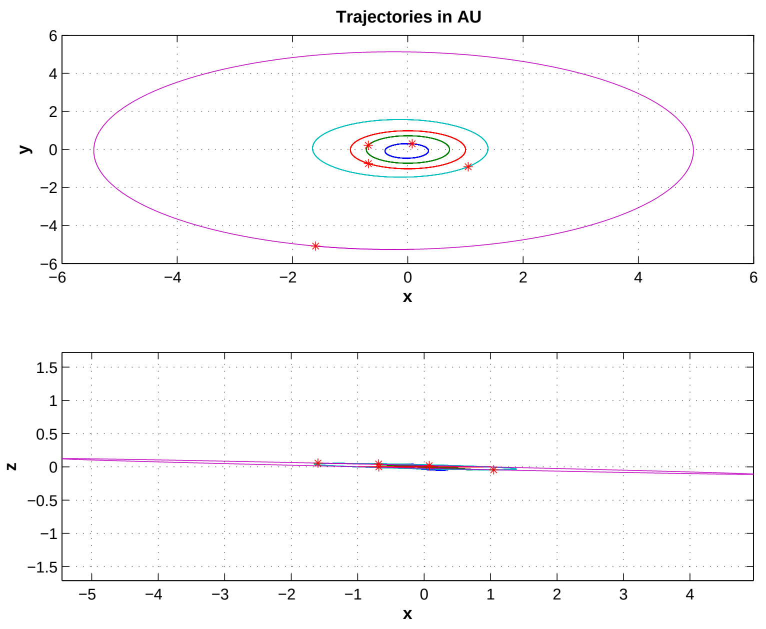
Table 9.1: Target Numbering Convention

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 9.1: Planet trajectories as generated by the built-in demo of `SolarSysJPL`



The demo `PlanetDemo` 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 10

PLOTTING TOOL

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

10.1 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 2019.1      23-Dec-2019

  Directories:
  Demos
  Demos/GUI
  Derivation Functions
  GUI
  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.

10.2 Getting Started

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

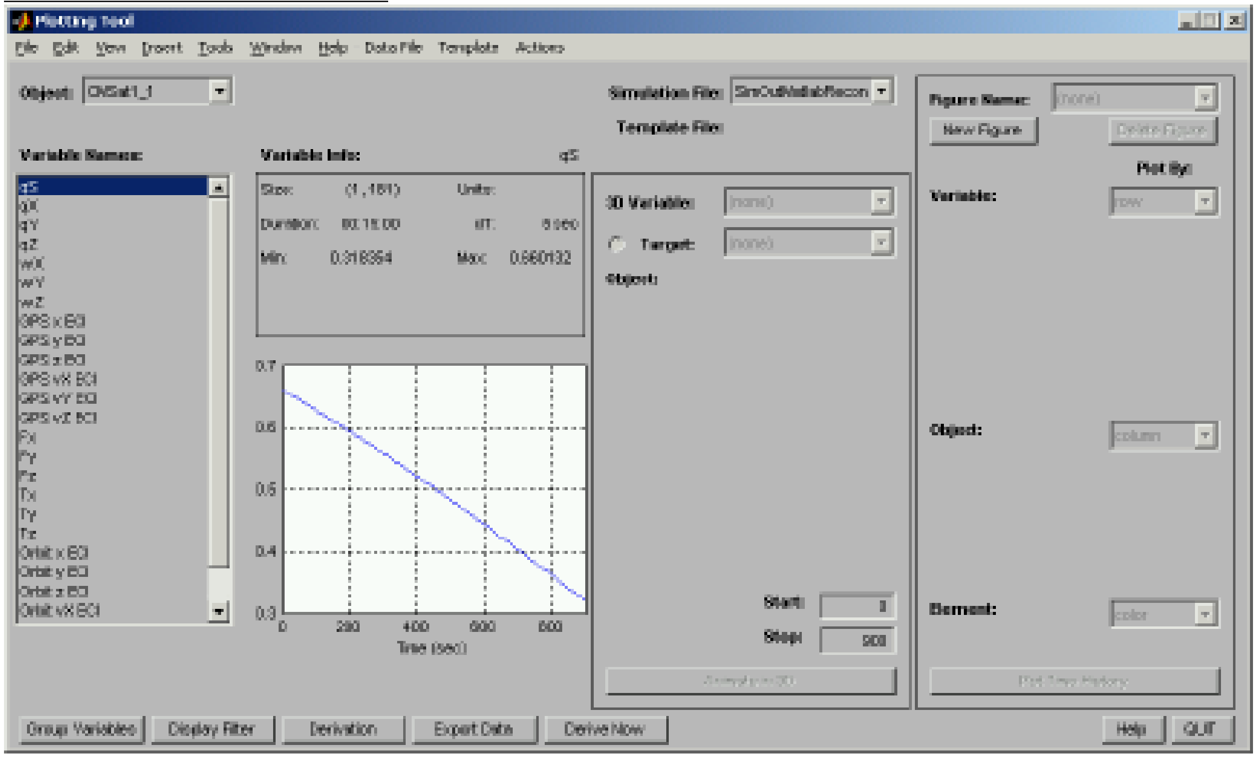
```
>> PlottingTool
```

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 10.1: Plotting Tool GUI with Newly Loaded Data File



Object: CNSat1_1

Simulation File: SimOutMatlabRecon

Variable Names: qS, qX, qY, qZ, wX, wY, wZ, GPS x ECI, GPS y ECI, GPS z ECI, GPS vX ECI, GPS vY ECI, GPS vZ ECI, Fx, Fy, Fz, Tx, Ty, Tz, Orbit x ECI, Orbit y ECI, Orbit z ECI, Orbit vX ECI

Variable Info: qS

Size: (1,181) Units:
Duration: 00:15:00 dT: 5 sec
Min: 0.318354 Max: 0.660132

3D Variable: (none)
Target: (none)
Figure Name: (none)
Variable: row
Object: column
Element: color
Start: 1
Stop: 900

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:

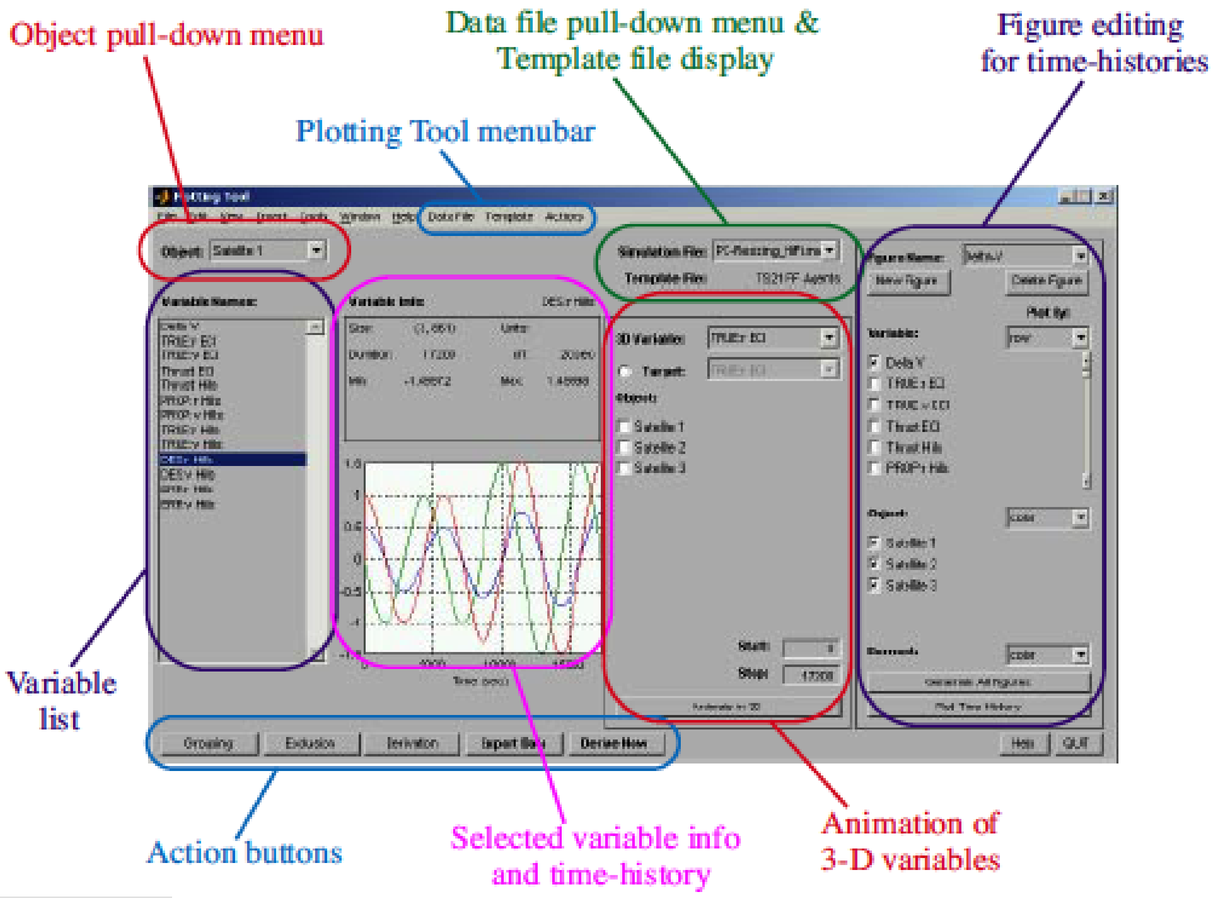
- Grouping and organizing variables
- Selecting which variables should be displayed to the screen
- Generating time-history plots
- Animating 3-dimensional variables
- Deriving new data from the raw data
- Applying predefined templates to perform any of the above actions automatically

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

10.3 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 10.2: Diagram of the Plotting Tool GUI



10.3.1 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.

10.3.2 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.

10.3.3 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 10.7, 10.5, and 10.6.

10.3.4 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.

10.3.5 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 10.4 on the facing page.

10.3.6 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 10.8 on page 98.

10.3.7 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.

10.3.8 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.

10.3.9 Shortcuts

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

Table 10.1: 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

10.4 Creating Figures

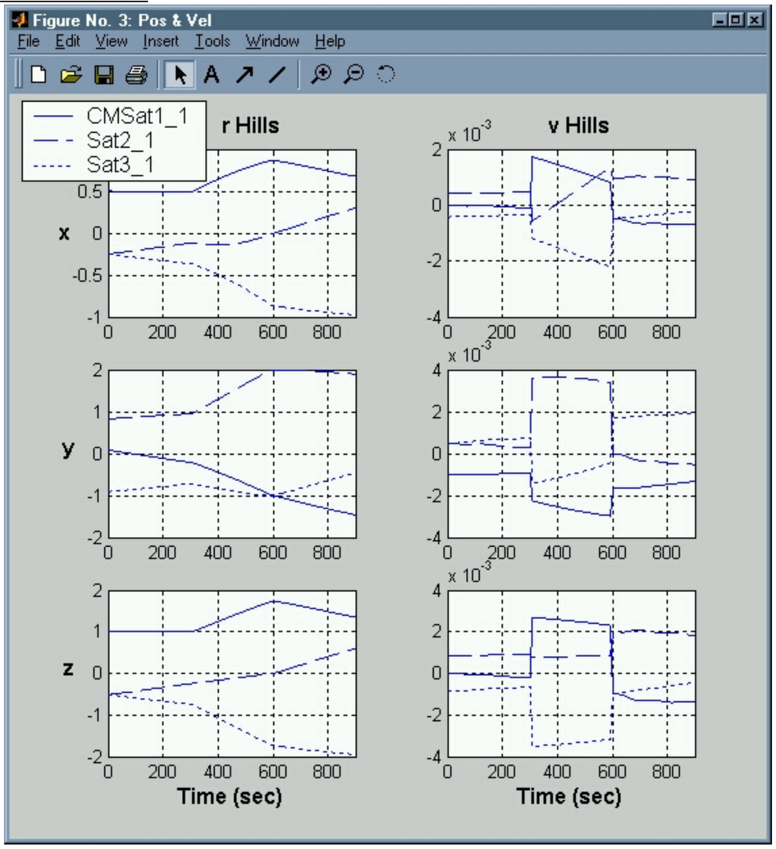
The right hand side of the GUI is used for generating figures. First, click the New Figure button and pick a name for the new figure. You can choose the variables and satellites you wish to plot from the checkbox lists, and indicate how they are distinguished with the pull-down menus under the words, 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 10.3 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 10.3: 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 10.7.

10.5 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.

10.6 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.

10.7 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 10.5 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.

10.8 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 10.4 on the next page provides a snapshot of the Animation GUI generated with the relative Hills position derived from the `SimOutMatlabRecon.mat` data file.

10.9 Templates

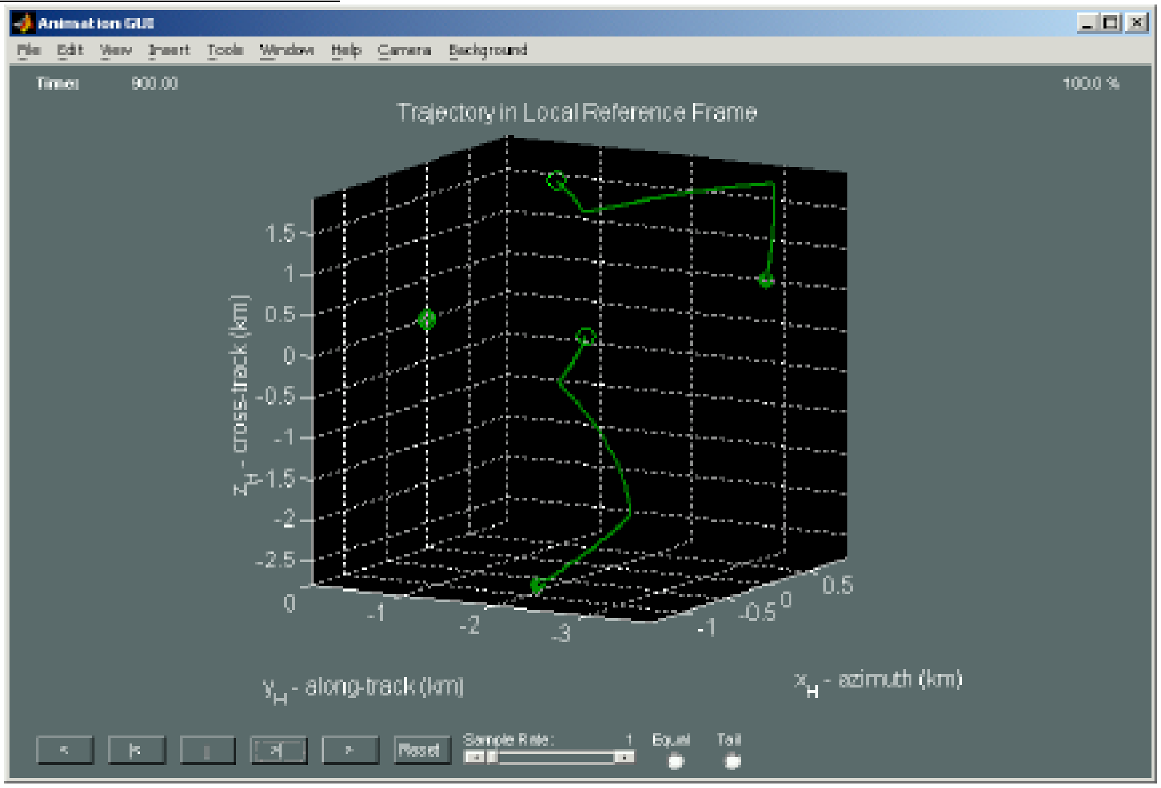
Once a data file is loaded into the Plotting Tool, the user may manipulate the data in a number of different ways. Variables may be grouped or excluded, new variables derived, and figures generated. All of these settings may be saved to a template file. The template may be edited over time, and applied 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 10.4: Animation GUI Snap-Shot



10.10 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.

10.10.1 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 10.2, ‘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 10.2 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 10.2: 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.

10.10.2 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 10.1: 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.

10.11 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 11

POINTING BUDGETS

This chapter discusses how to generate a pointing budget.

11.1 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 11.1: `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 = str2mat(categ,'Bias');
categ = str2mat(categ,'Diurnal');
categ = str2mat(categ,'Diurnal');
desc  = 'Thermal';
desc  = str2mat(desc,'Misalignments');
desc  = str2mat(desc,'Thermal');
desc  = str2mat(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 11.2: 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

11.2 Pointing Budget GUI

The function `PointingBudgetGUI` creates a GUI for doing budgets. Figure 11.1 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 11.1: Open the GUI

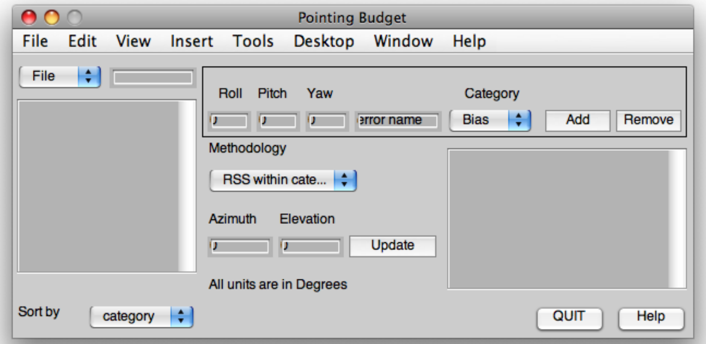
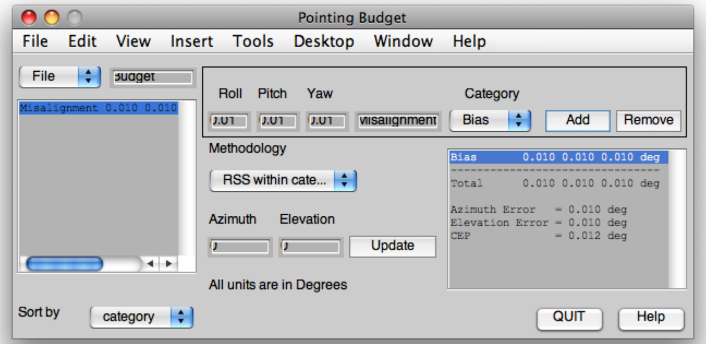


Figure 11.2 on the next page shows the GUI after the first error has been entered. The component appears in both scrollable text windows. The right one also shows the resulting pointing error.

Figure 11.3 on the facing page shows the GUI after the second error has been entered.

Figure 11.4 on the next page shows save file dialog box.

Figure 11.2: Entering the first error



Budget

```
Misalignment 0.010 0.010
```

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

Methodology: RSS within cate...

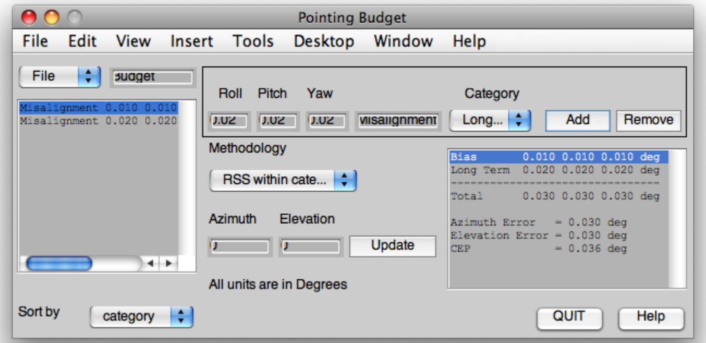
Azimuth: 0 Elevation: 0

All units are in Degrees

Sort by: category

```
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 11.3: Entering a second error



Budget

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

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

Methodology: RSS within cate...

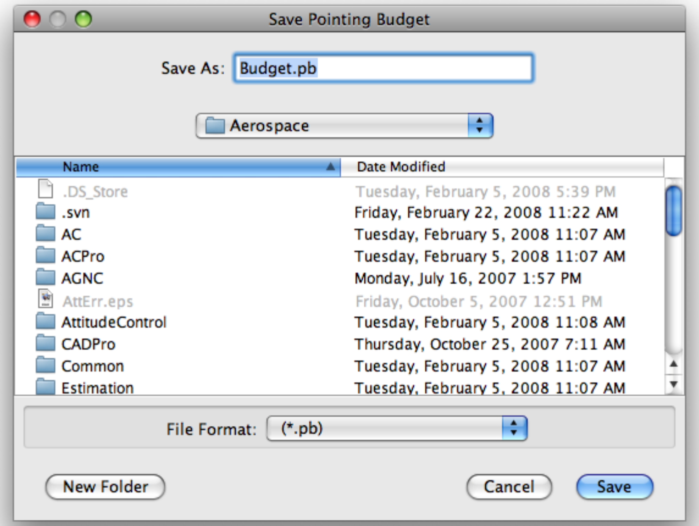
Azimuth: 0 Elevation: 0

All units are in Degrees

Sort by: category

```
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 11.4: 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 12

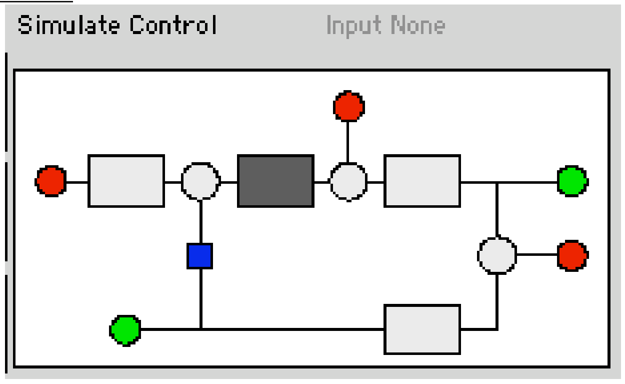
DESIGNING CONTROLLERS

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.

12.1 Using the block diagram

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

Figure 12.1: 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.

12.2 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 12.1: 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 12.2 on the facing page.

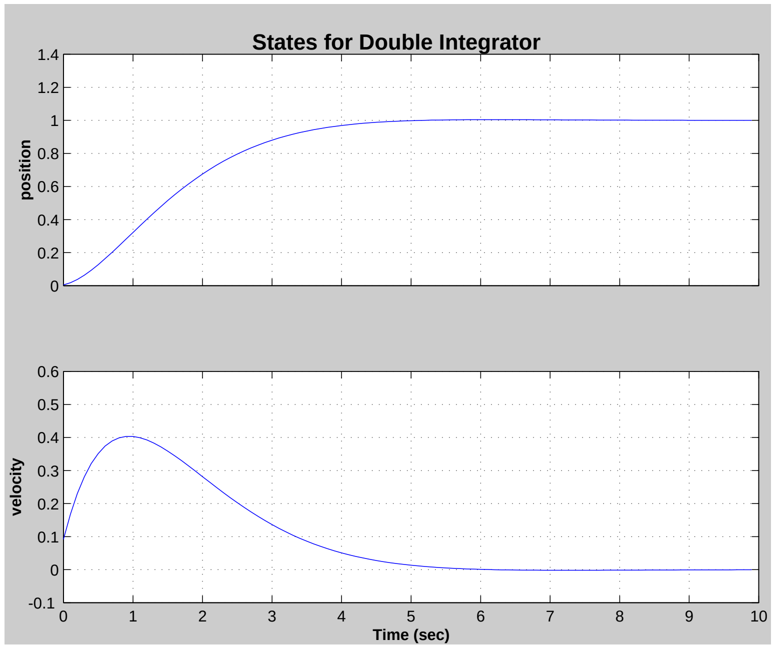
12.3 Eigenstructure Assignment

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

Listing 12.2: 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
%---------------------------------
```

Figure 12.2: Step response



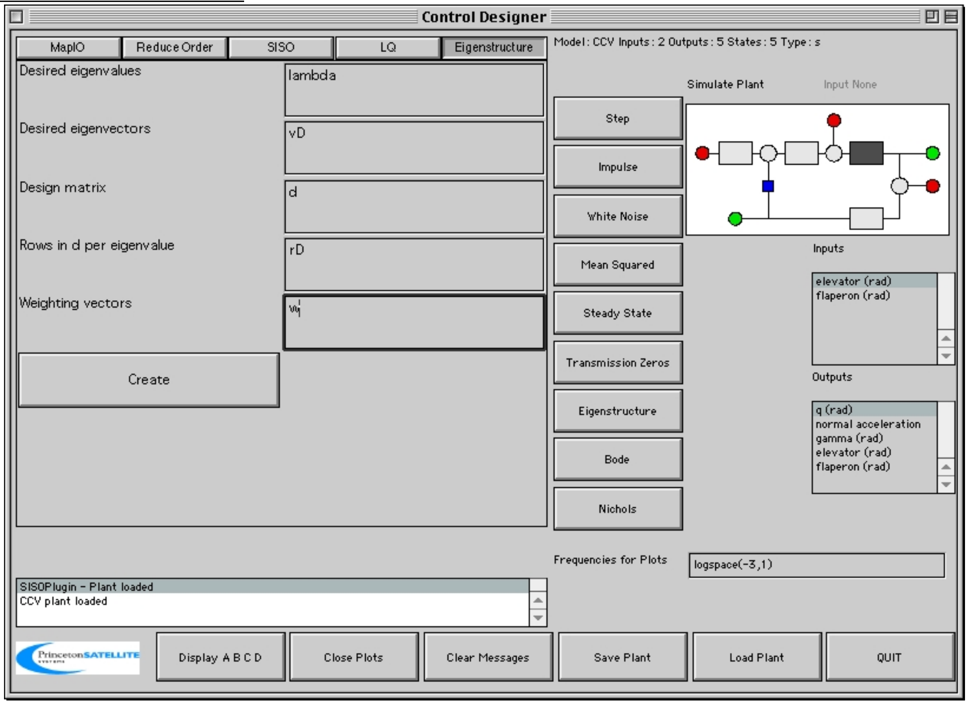
```
w  = [ 1    1    1  1  1;...
       1    1    1  1  1;...
       100  100  1  1  1];

% 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 12.3 on the next page.

Figure 12.3: 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)

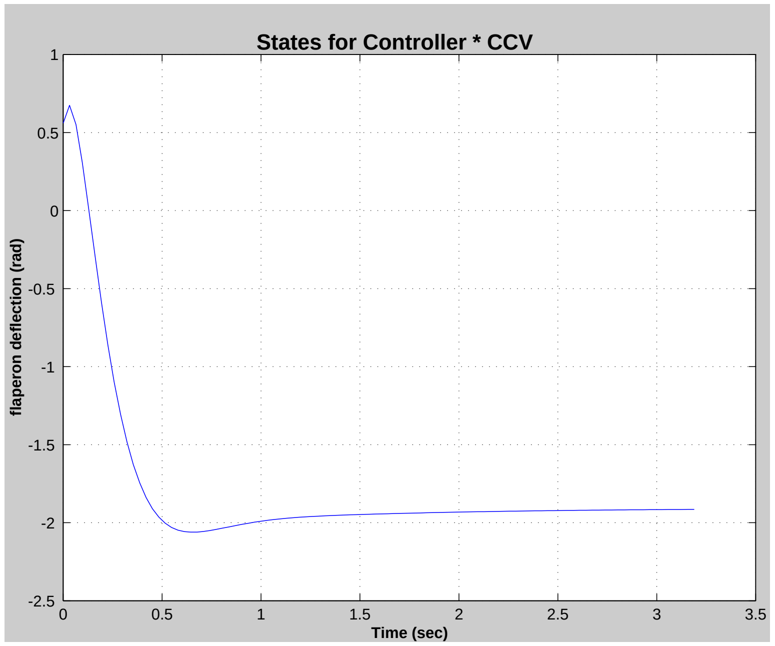
Outputs: q (rad), normal acceleration, gamma (rad), elevator (rad), flaperon (rad)

Frequencies for Plots: logspace(-3,1)

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

Push Create. Next push Step. You will see the plot in Figure 12.4.

Figure 12.4: Step response with eigenstructure assignment



CHAPTER 13

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

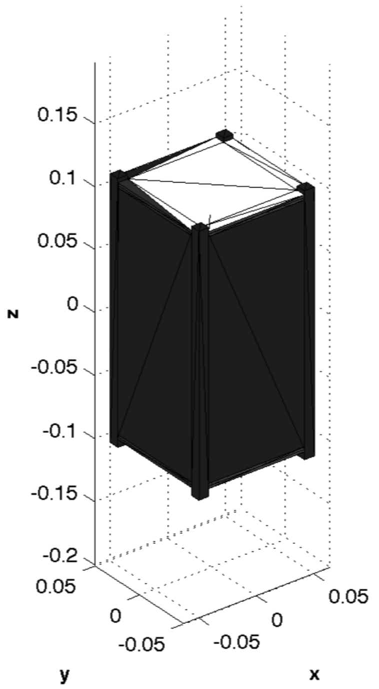
13.1 CubeSat Modeling

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 13.1 on the next 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 13.1: 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 13.1.

Table 13.1: 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.

13.2 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 13.2 on the next page compares the time output of the two examples, where we can see that the simple simulation had mostly a constant step size. Figure 13.3 shows the typical orbit results.

Figure 13.2: Orbit simulation timestep results, simple on the left with with disturbances on the right.

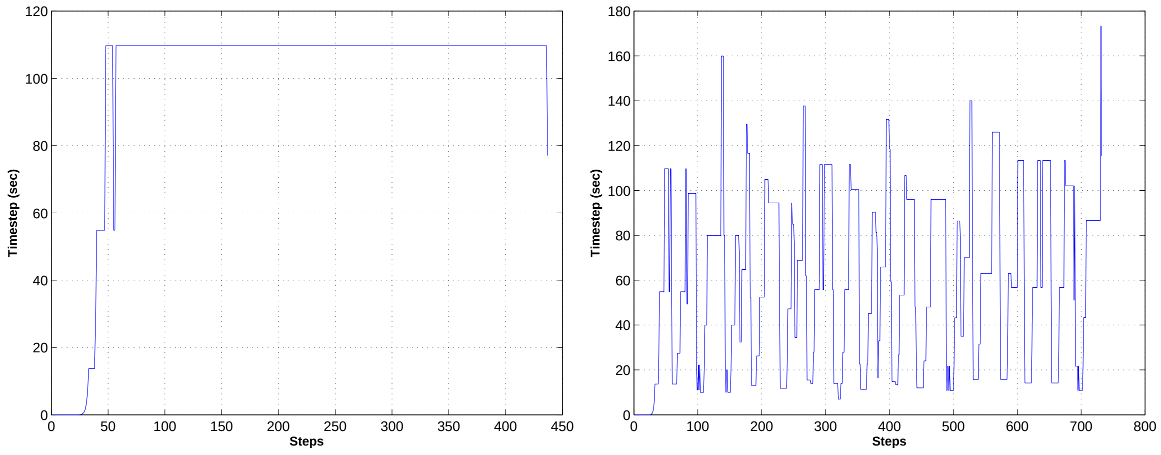
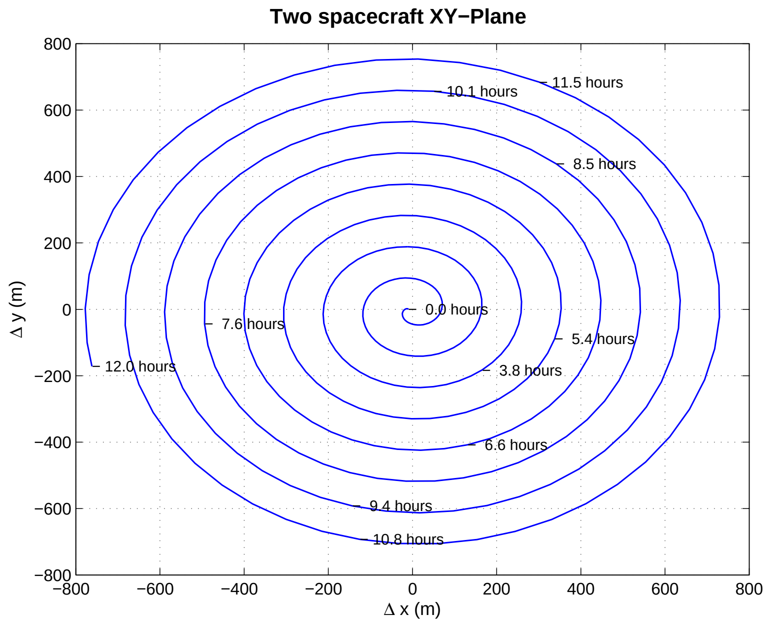


Figure 13.3: Orbit evolution for an initial separation of 10 meters

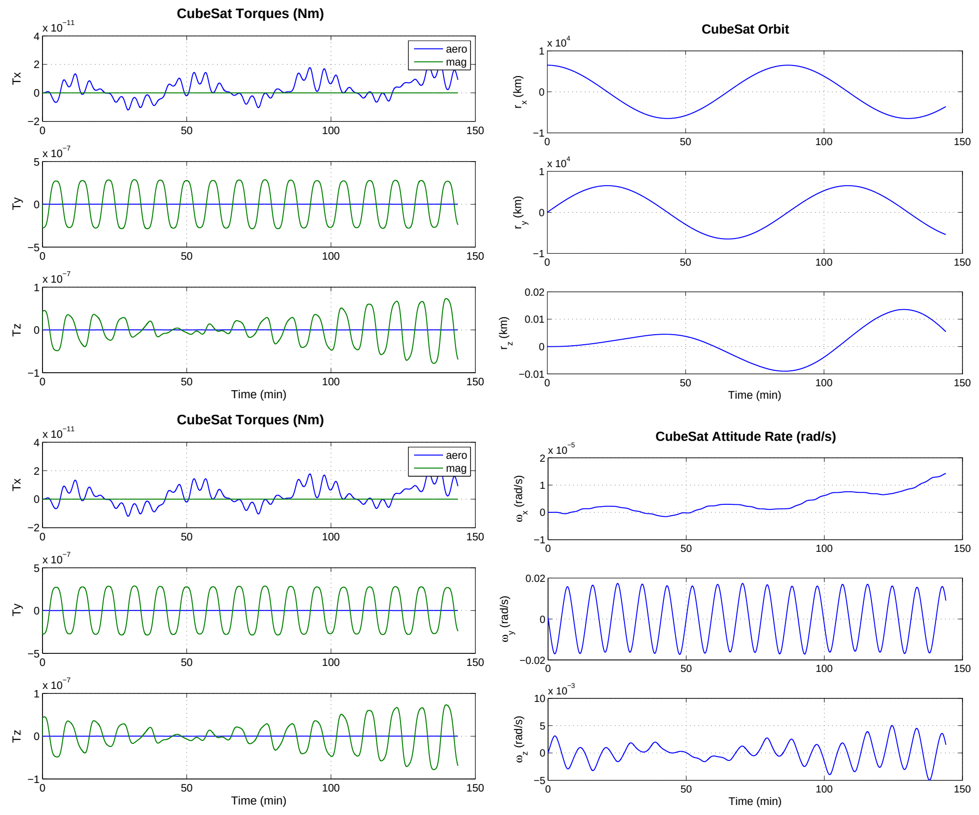


`CubeSatSimulation` simulates the attitude dynamics of the CubeSat in addition to a point-mass orbit and the power subsystem. This simulation includes forces and torques from drag, radiation pressure, and an magnetic torque, such as from a torquer control system. Since this simulation can include control, 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 13.4 for sample results. The simulation takes about one minute of computation time per low-Earth orbit.

Figure 13.4: `CubeSatSimulation` example results



The key functions used in simulations are summarized in Table 13.2. The space environment calculations from `CubeSatEnvironment` are then passed to the force models in `CubeSatAero` and `CubeSatRadiationPressure`.

Table 13.2: 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.

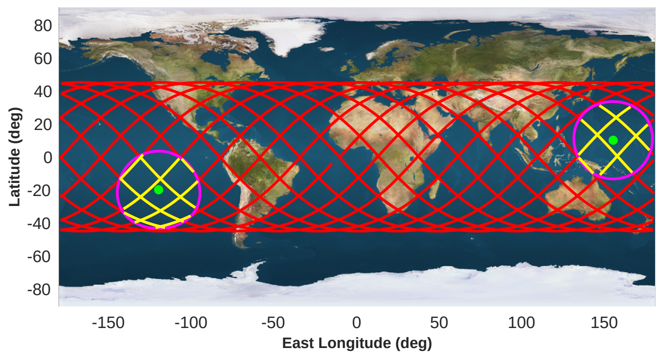
`RHSCubeSat` also models battery charging if the `batteryCapacity` field of the `power` structure has been appropriately set. Power beyond the calculated consumption will be used to charge the battery until the capacity is reached. The battery charge is always be the last element of the spacecraft state (after the states of any optional reaction wheels).

13.3 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 13.5. There are two ground targets, one in South America and one in France (large green dots).

Figure 13.5: Observation time windows

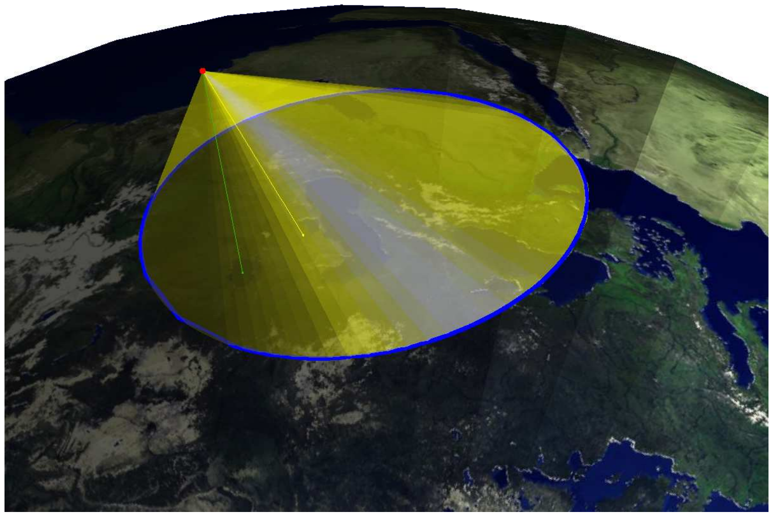


The satellite is placed in a low Earth orbit, given a field of view of 180 degrees, and the windows are generated over a 7 hour horizon. The figure shows the field of view in magenta and the satellite trajectory segments when the target is in the field of view are highlighted in yellow. This function 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 13.6 on the following page we have used the camera controls to zoom in on the sensor cone. The nadir axis is drawn in green and the boresight axis in yellow.

The `AttitudeProfileDemo` shows how to assemble several profile segments together and get the resulting ob-

Figure 13.6: `RapidSwath` built-in demo results



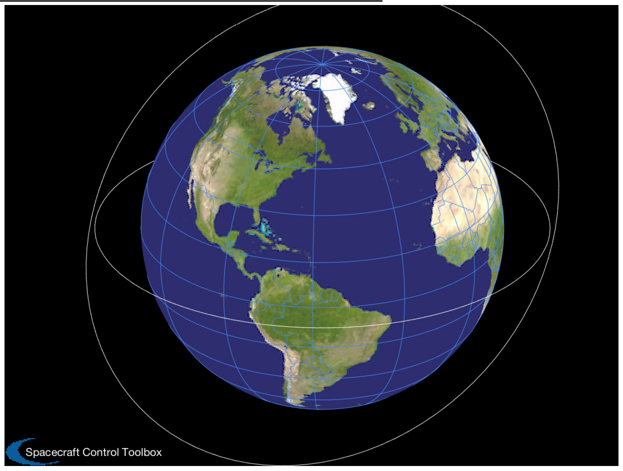
servation windows. The segments can be any of a number of modes, such as latitude/longitude pointing, nadir or sun pointing, etc.

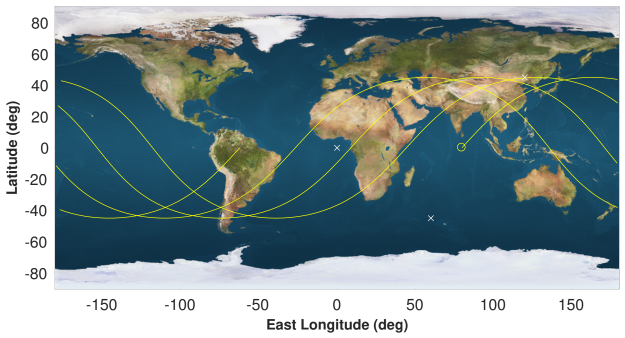
13.4 Visualization

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.

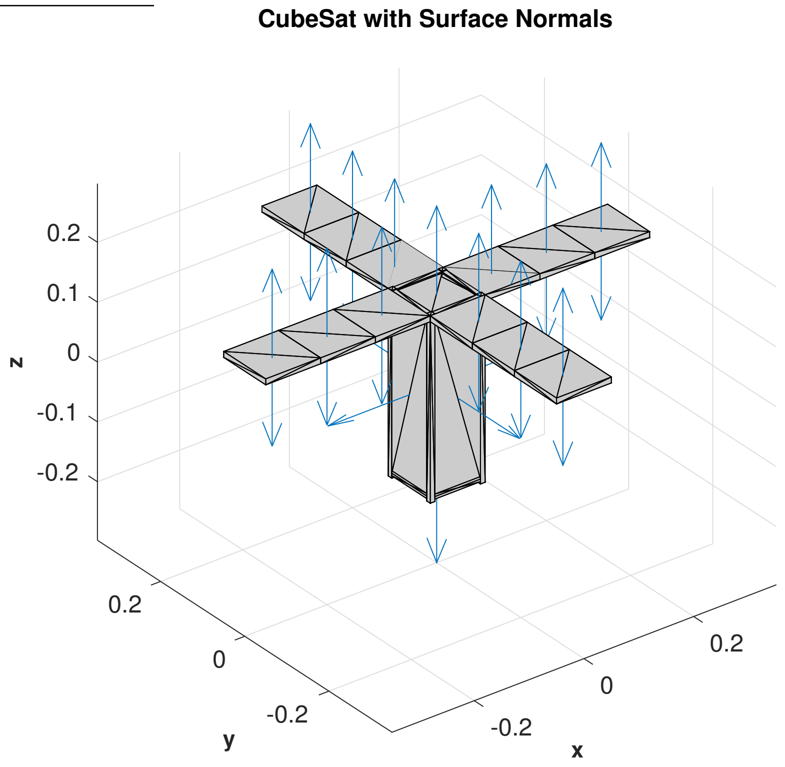
Figure 13.7: Orbit visualization with `PlotOrbit` and `GroundTrack`





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.

Figure 13.8: CubeSat model with deployable solar panels viewed with `DrawCubeSat`



A representative model of the spacecraft may also be viewed in its orbit, along with a sensor cone and lines of sight to all of the visible GPS satellites. Use `DrawSpacecraftInOrbit.m` to generate this view. An example is shown in Figure 13.9. The image on the left shows the spacecraft orbit, its sensor cone projected on the Earth, the surrounding GPS satellites, and lines of sight to the visible GPS satellites. The image on the right is a zoomed-in view, where the spacecraft CAD model may be clearly seen.

Figure 13.9: Spacecraft visualization with sight lines using `DrawSpacecraftInOrbit`

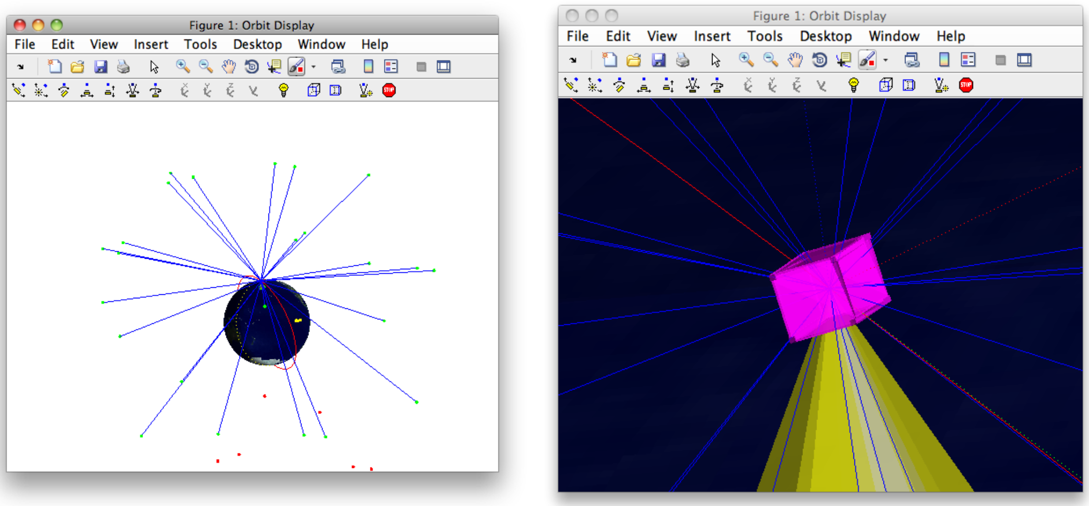
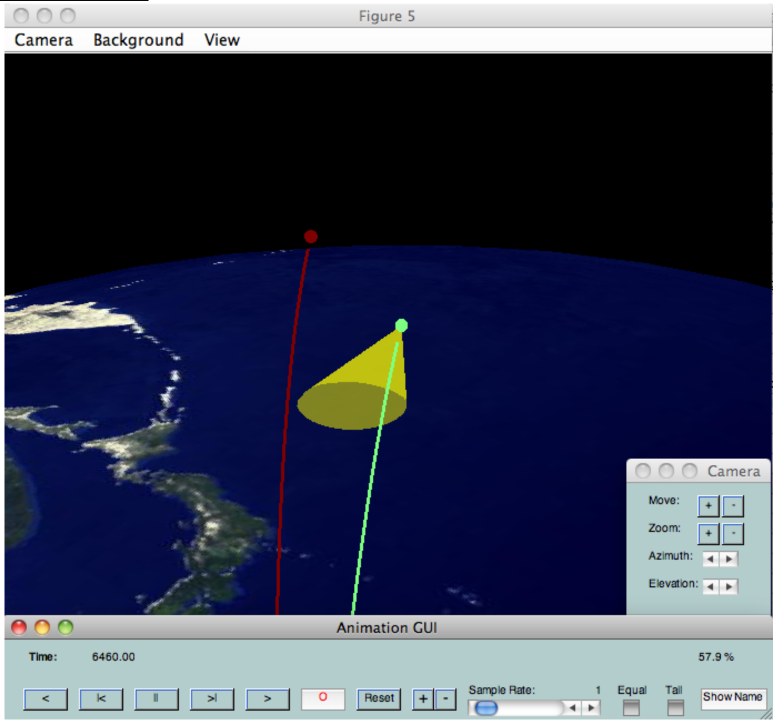


Figure 1: Orbit Display
File Edit View Insert Tools Desktop Window Help
Figure 1: Orbit Display
File Edit View Insert Tools Desktop Window Help

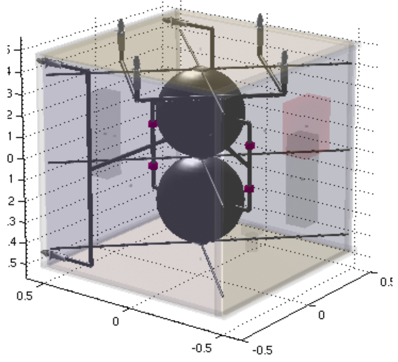
Run the `OrbitAndSensorConeAnimation.m` mission planning demo to see how to generate simulated orbits, compute sensor cone geometry, and package the data for playback using `PackageOrbitDataForPlayback` and `PlaybackOrbitSim`. The playback function loads two orbits into the `AnimationGUI`, which provides 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 13.10 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 13.10: Playback demo using `PlaybackOrbitSim`



PART II

ATTITUDE CONTROL DESIGN EXAMPLES



CHAPTER 14

ATTITUDE CONTROL

14.1 ACS Overview

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

14.2 Design Examples

14.2.1 PID Design

`Attitude3D` in SC/Demos/Attitude provides a simple example of designing a three-axis PID control system using `PIDMIMO`. A full CAD model is used with the `Disturbances` function in the simulation for calculating disturbances. `REAControl` (REA means Reaction Engine Assembly - assembly is sometimes used for hardware composed of several major parts) extends this example to a satellite with thrusters, where equivalent pulse widths are calculated.

14.2.2 Magnetic Control

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

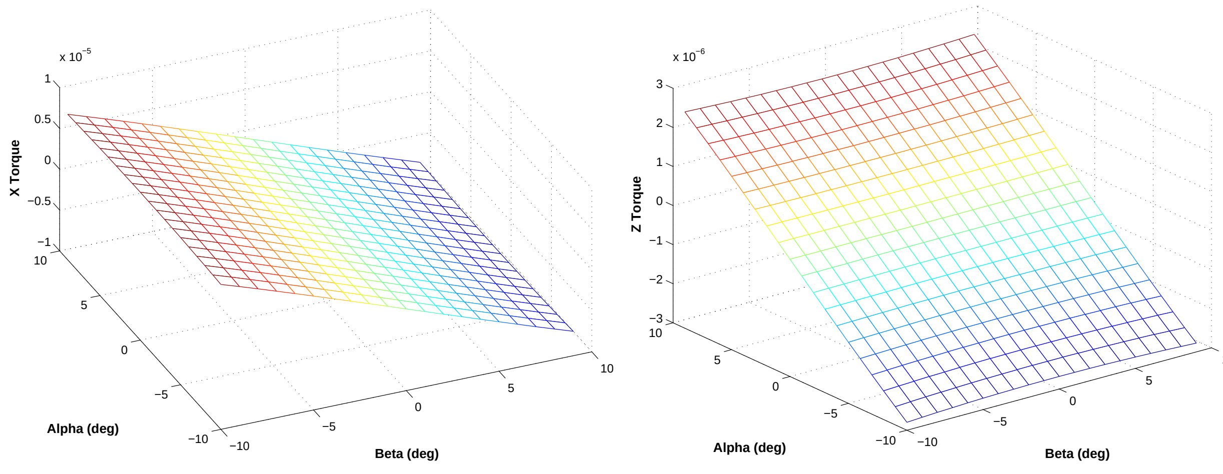
14.2.3 Microwave Anisotropy Satellite

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

14.2.4 Momentum Unloading

This design uses solar pressure on a double-gimballed solar array for two-axis momentum control. The script, `SolarMomentumControl.m` in ProControl, maps out the torque produced over a range of gimbal angles before performing the simulation. See Figure 14.1.

Figure 14.1: Solar pressure torque map for double-gimballed solar array



14.2.5 Sun Nadir Pointing

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

14.2.6 Roll Yaw Control Using a Double Pivot

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

14.2.7 ComStar

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

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

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

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

CHAPTER 15

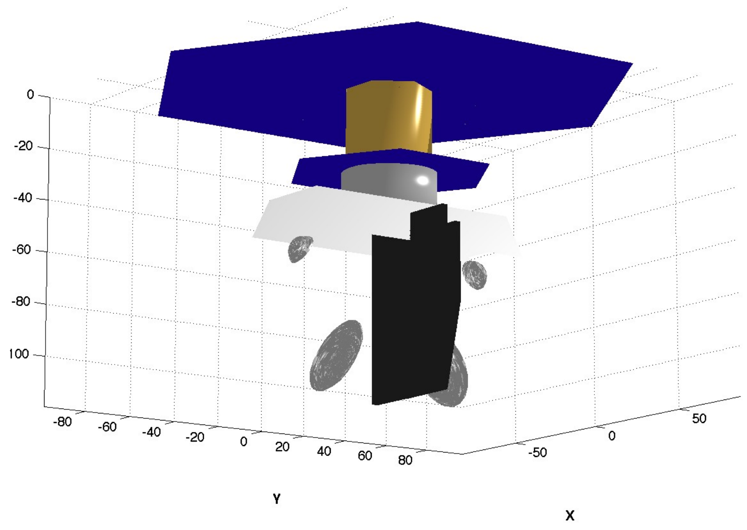
A WORKED DESIGN EXAMPLE

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

15.1 The MAP Mission

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

Figure 15.1: The Microwave Anisotropy Probe



15.2 Control Modes

The MAP spacecraft requires three control modes:

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

Optionally, a safe hold mode could be added.

15.3 Sensing and Actuation

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

15.4 Control System Design

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

15.5 Simulation Results

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

15.6 The **MAPSim** Script

In this section we will walk through the `MAPControlSim`. Once you understand how is was written you will be able to easily design and build your own control systems and simulations. Listing 15.1 shows the header which defines a few useful constants. The global variable is used to control the time GUI. We routinely close all windows.

Listing 15.1: File header *MAPControlSim.m*

```
%% Implements and simulates the MAP normal mode control.
% Demonstrate 3 axis control of a spinner using wheels.
% This is not the  NASA design but one done in house. It shows how to
% integrate control design and simulation into a single script.
% This spacecraft is a spinner and has reaction wheels. The reaction
% wheels are used to control the orientation of the spin axis.
% The simulation demonstrates precession of the spin axis.
%
% This script also generates an STK attitude file.
% -------------------------------------------------------------------------
%  See also PIDMIMO, PIDesign, FGs, QForm, QTForm, NPlot, Plot2D, PlotV,
%  TimeGUI, STKAtt, CosD, RK4, SinD, JD2Date
% -------------------------------------------------------------------------
%%
%--------------------------------------------------------------------------
%   Copyright (c) 1996-2010, 2016 Princeton Satellite Systems, Inc.
%   All rights reserved.
%--------------------------------------------------------------------------
% Since version 1.
% 2016.1 - Update RHS to use function handle. Save STK file in same directory as
% this file.
%--------------------------------------------------------------------------
```

MAPControlSim.m

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

Listing 15.2: Sampling intervals *MAPControlSim.m*

```
global simulationAction
simulationAction = '␣';

% Constants
% ---------
degToRad = pi/180;
radToDeg = 180/pi;
rPMToRPS = pi/30;

% STK Information
%----------------
sTKVersion = '3.0';

% The control sampling period and the simulation integration time step
% --------------------------------------------------------------------
tSamp       = 0.25;
```

MAPControlSim.m

Listing 15.3 defines the spacecraft mass properties. This set is not based on the actual MAP properties but is for demonstration purposes only. We are using a gyrostat (rigid body + wheels) model that allows for any number of reaction wheels so we must include the spin axis inertias of the wheels. We will use three orthogonal wheels.

Listing 15.3: Mass properties *MAPControlSim.m*

```
% -------------------------------------------------------------------------
```

```
%% Parameters for the spacecraft model
% ------------------------------------------------------------------------

% Spacecraft Inertias
% -------------------
inr          = [2000,0,0;0,2000,0;0,0,4000];
inrRWA       = 1;
inrW         = [inrRWA,inrRWA,inrRWA];
invInr       = inv(inr);
tDist        = [0;0;0];
```

MAPControlSim.m

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

Listing 15.4: Tachometer loop design *MAPControlSim.m*

```
% RWA Tach loops
% --------------
zeta         = 0.7071;    % Damping ratio
wN           = 1.0;       % Closed loop undamped natural frequency

[aTL,bTL,cTL,dTL] = PIDesign( zeta, wN, inrW(3), tSamp, 'Delta' );
```

MAPControlSim.m

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

Listing 15.5: ACS design *MAPControlSim.m*

```
% Attitude Loops
% --------------
zeta         = 0.7071;    % Damping ratio
wN           = 0.5;       % Closed loop undamped natural frequency
wR           = 4.0;       % Rate filter break frequency
tau          =  50;       % Integrator time constant

[aRoll ,bRoll, cRoll, dRoll]  = PIDMIMO( inr(1,1), zeta, wN, tau, wR, tSamp, 'Delta');
[aPitch,bPitch,cPitch,dPitch] = PIDMIMO( inr(2,2), zeta, wN, tau, wR, tSamp, 'Delta');
```

MAPControlSim.m

Listing 15.6 gives the yaw rate loop that will maintain the MAP spin rate. This uses the same PI controller as the reaction wheels but with a slower time constant.

Listing 15.6: Rate loop *MAPControlSim.m*

```
% Rate Loops
% ----------
zeta         = 1.0;  % Damping ratio
wN           = 0.5;  % Closed loop undamped natural frequency

[aYaw,  bYaw,  cYaw,  dYaw]  = PIDesign( zeta, wN, inr(3,3), tSamp, 'Delta' );
```

MAPControlSim.m

Now that the controller design is complete it is time to run a simulation to test the system. Of course we should also do frequency domain analysis of the entire controller. Even though we used SISO design techniques this is really a MIMO design. The controller has fast inner loops to control the wheels and slower outer loops to control the attitude. Several tools exist in SCT for analyzing such systems.

The first step is to initialize the control system.

Listing 15.7: Initialize the control system *MAPControlSim.m*

```
%% Initialize the control system
% ----------------------------
xTL              = zeros(3,1);
xRoll            = [0;0];
xPitch           = [0;0];
xYaw             = 0;
tC               = [0;0;0];
spinRate         = 0.464*rPMToRPS;
precRate         = 2*pi/3600;
precAngleDemand  = 22.5;
precAngle        = 0;
precAngleGain    = 0.995;

%% Momentum Management gain
% -----------------------
kMM              = 0.00500;

% The control distribution matrix converts
% torque demand to angular acceleration demand
% -------------------------------------------
aRWA        = eye(3)/inrW(3);
wRWAC       = [0;0;0];
```

MAPControlSim.m

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

Listing 15.8: Set up the simulation *MAPControlSim.m*

```
%% Plotting arrays
% ---------------
cPlot       = zeros( 3,nPlot);
hPlot       = zeros( 3,nPlot);
tPlot       = zeros( 1,nPlot);
xPlot       = zeros(10,nPlot);
zPlot       = zeros( 3,nPlot);
rPlot       = zeros( 3,nPlot);
pPlot       = zeros( 2,nPlot);
ePlot       = zeros( 2,nPlot);
wPlot       = zeros( 3,nPlot);
yPlot       = zeros( 3,nPlot);

%% Initial conditions
% ------------------
%                 q         w            wRWA
x           = [[1;0;0;0];[0;0;spinRate];[0;0;0]];

dTSim       = tSamp;
t           = 0;
nP          = 0;
kP          = 0;
tW          = zeros(3,1);
roll        = 0;
pitch       = 0;
yaw         = 0;

%% Initialize the time display
%----------------------------
tToGoMem.lastJD         = 0;
tToGoMem.lastStepsDone  = 0;
tToGoMem.kAve           = 0;
ratioRealTime           = 0;
```

MAPControlSim.m

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

Listing 15.9: Start of the simulation loop *MAPControlSim.m*

```

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

MAPControlSim.m

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

Listing 15.10: Sensor models *MAPControlSim.m*

```

% -------------------------------------------------------------------------------
% Sensors
% -------------------------------------------------------------------------------

% RWA Tachometer
% --------------
wTach  = x(8:10);

% Gyros
% -----
wCore  = x(5:7);

% Attitude
%---------
qIToB  = x(1:4);
```

MAPControlSim.m

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

Listing 15.11: Control system *MAPControlSim.m*

```

% ------------------------------------------------------------------------
% The Attitude Control System
% ------------------------------------------------------------------------

% Momentum Management
% -------------------

% Neglect the rate errors in the body component-assume exact tracking
% --------------------------------------------------------------------
hTotal = QTForm( qIToB, inr*[wCore(1);wCore(2);wCore(3)] + inrRWA*uW*(wTach+wCore) );

% Proportional controller for momentum
% We could feedforward this to the controller
% -------------------------------------------
```

MAPControlSim.m

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

Listing 15.12: Error computation *MAPControlSim.m*

```

% Compute the errors
%-------------------
precAngle   = precAngleGain*precAngle + (1-precAngleGain)*precAngleDemand;
cP          = CosD(precAngle);
sP          = SinD(precAngle);
xT          = [sP*cos(precRate*t);sP*sin(precRate*t);cP];
xTB         = QForm( qIToB, xT );
rollError   =  asin(xTB(2));
pitchError  = -asin(xTB(1));
```

MAPControlSim.m

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

Listing 15.13: Attitude loop *MAPControlSim.m*

```

% Convert torque demand to RWA angular acceleration demand
% -------------------------------------------------------
```

MAPControlSim.m

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

Listing 15.14: Compute wheel speed demand *MAPControlSim.m*

```

% Integrate to get wheel speed demand
% -----------------------------------
```

MAPControlSim.m

The wheels are commanded through the tach loops.

Listing 15.15: RWA tach loops *MAPControlSim.m*

```

% The RWA Tach Loops
% ------------------
wError =  wTach - wRWAC;
tRWA   = -dTL*wError - cTL*xTL;
```

MAPControlSim.m

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

Listing 15.16: Simulation *MAPControlSim.m*

```

% -----------------------------------------------------------------------
% Update the equations of motion
% -----------------------------------------------------------------------
x = RK4(@FGs,x,dTSim,t,inr,invInr,tDist+tMM,inrW,uW,tRWA');
```

MAPControlSim.m

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

Listing 15.17: Saving plot information *MAPControlSim.m*

```
% -----------------------------------
wRWAC  = wRWAC + tSamp*wDRWA;

% The RWA Tach Loops
% ------------------
wError =  wTach - wRWAC;
tRWA   = -dTL*wError - cTL*xTL;
xTL    =  xTL + aTL*xTL + bTL*wError;

% -----------------------------------------------------------------------
% Update the equations of motion
% -----------------------------------------------------------------------
x = RK4(@FGs,x,dTSim,t,inr,invInr,tDist+tMM,inrW,uW,tRWA');
```

```
t = t + dTSim;

% Plotting
% --------
if nP == 0,
  kP          = kP + 1;
  xPlot(:,kP) = x;
  tPlot(1,kP) = t;
  cPlot(:,kP) = tC;
  hPlot(:,kP) = hTotal;
  zPlot(:,kP) = tMM;
  xI          = QTForm(qIToB,[0;0;1]);
  rPlot(:,kP) = xI;
  pPlot(:,kP) = acos([xI(3);xT(3)]);
  ePlot(:,kP) = [rollError;pitchError];
  wPlot(:,kP) = xT;
  yPlot(:,kP) = xTB;
  nP          = nPMax - 1;
else
  nP          = nP - 1;
```

MAPControlSim.m

After the simulation is done we plot the results.

Listing 15.18: Plotting *MAPControlSim.m*

```

j     = 1:kP;
tPlot = tPlot(j);
epoch = JD2Date;

filePath = fileparts(mfilename('fullpath'));
[err, message] = STKAtt( fullfile(filePath,'STKAttitudeFile.txt'),sTKVersion,epoch,kP,tPlot,
    xPlot( 1: 4,j),'quaternion');

Plot2D(tPlot,xPlot( 1: 4,j),'Time␣(sec)',['Q_s';'Q_x';'Q_y';'Q_z'],'Quaternion')
Plot2D(tPlot,xPlot( 5: 7,j),'Time␣(sec)',['\omega_x';'\omega_y';'\omega_z'],'Body␣Rates')
Plot2D(tPlot,xPlot( 8:10,j),'Time␣(sec)',['W1';'W2';'W3'],'Reaction␣Wheels')
Plot2D(tPlot,cPlot(:,j),'Time␣(sec)',['X';'Y';'Z'],'Control␣Torque␣Demand')
Plot2D(tPlot,hPlot(:,j),'Time␣(sec)',['X';'Y';'Z'],'Body␣Momentum')
Plot2D(tPlot,zPlot(:,j),'Time␣(sec)',['X';'Y';'Z'],'Momentum␣Management␣Torque')
Plot2D(tPlot,pPlot(:,j)*radToDeg,'Time␣(sec)','Precession␣Angle␣(deg)','Precession␣Angle')
PlotV([rPlot(:,j);wPlot(:,j)],'X','Y','Z','Spin␣Axis␣and␣Target')
```

MAPControlSim.m

Figure 15.2 on the next page shows the results of `PlotV`. It shows the spin axis unit vector. It starts as [0;0;1] (the highest point on the plot) and as the spacecraft acquires picks up y and x components until it traces out a circle in the xy-plane. Figure 15.3 on the facing page shows the precession of the unit vector.

Figure 15.2: Motion of the unit vector

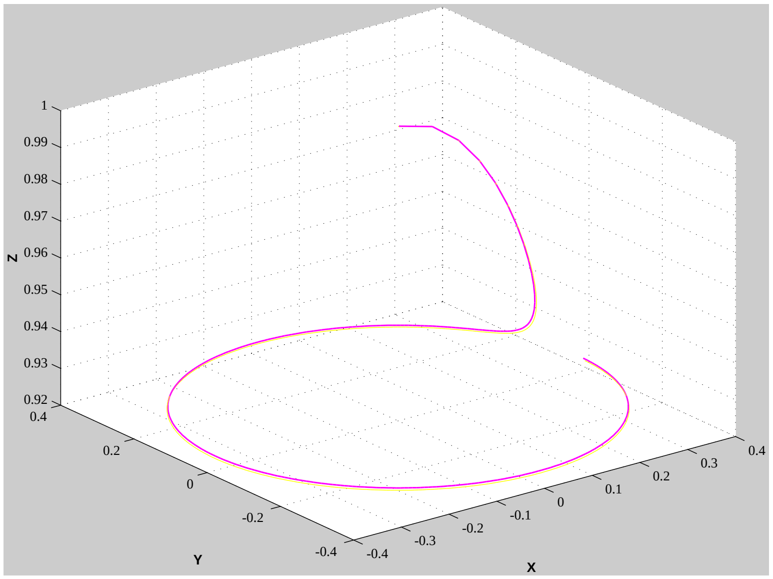
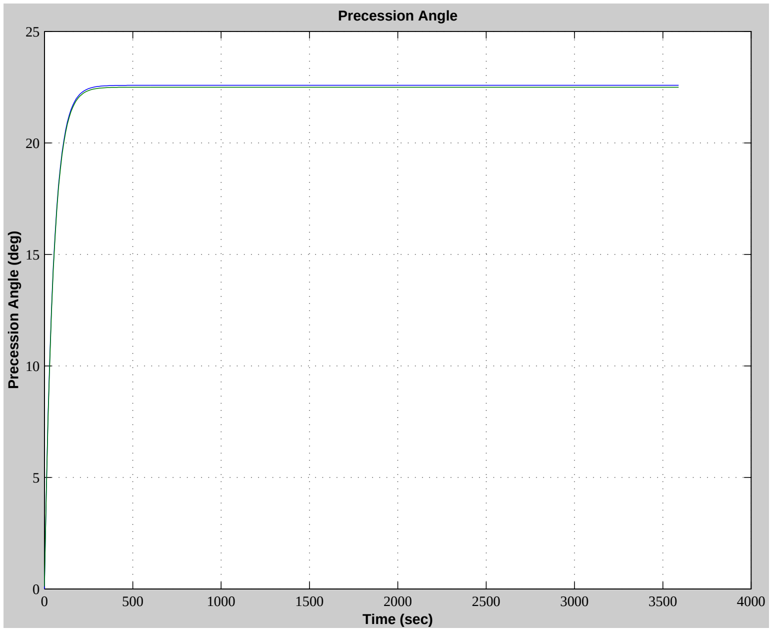


Figure 15.3: Precession of the unit vector



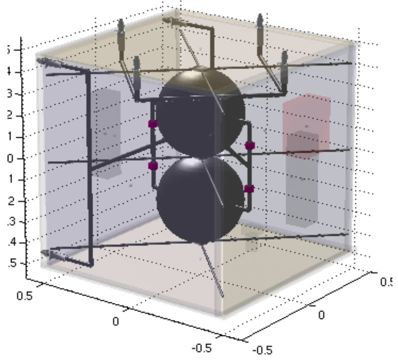
15.7 Summary

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

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

Part III

Orbit Propagation and Maneuvers



CHAPTER 16

ORBIT MODULE

16.1 Overview

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

The maneuver functions encompass a variety of types, including:

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

Additional analysis tools include a two-line element propagator, linearized orbit model, Gauss' variational equations, fuel budgets, and J2 effects. Equinoctial and Keplerian orbit mechanics are supported as well as cartesian, cylindrical, and spherical frames. If a tool or algorithm you want isn't available, please let us know!

CHAPTER 17

Orbit Propagation Examples

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

17.1 Low Thrust Orbit Raising

Reference

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

Introduction

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

$$\frac{\mathrm{d}a}{\mathrm{d}t} = \frac{2}{\sqrt{\mu}} a^(3/2)A \tag{17.1}$$

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

$$a^{-1/2} = a_0^{-1/2} - \frac{A}{\sqrt{\mu}}(t - t_0) \tag{17.2}$$

Results

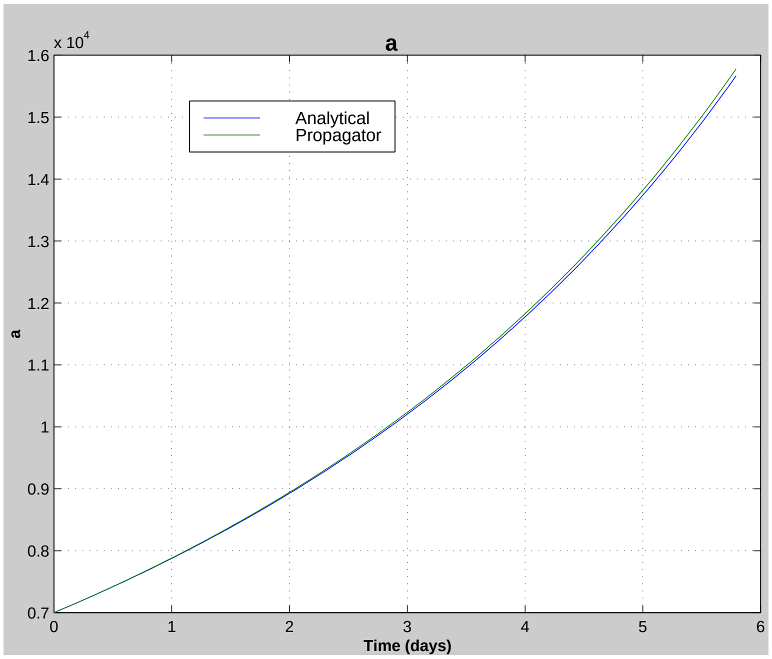
Type

```
LowThrustDemo
```

in the command window. The last plot will show the difference between the propagator and the analytical model (Figure 17.1 on the next page).

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

Figure 17.1: Comparison of an analytical low thrust model with the propagated solution.



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

17.2 Reentry Due to Drag

Reference

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

Introduction

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

$$\frac{\mathrm{d}a}{\mathrm{d}t} = -\sqrt{\mu a}\frac{C_D A}{m}\rho_0 e^{(-(a-R_E))/h} \tag{17.3}$$

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

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

$$H = h \ \ln \left(e^{H_0/h} - \frac{\rho_0 \sqrt{\mu R_E} \left(\frac{C_D A}{m}\right)}{h} (t - t_0) \right) \tag{17.4}$$

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

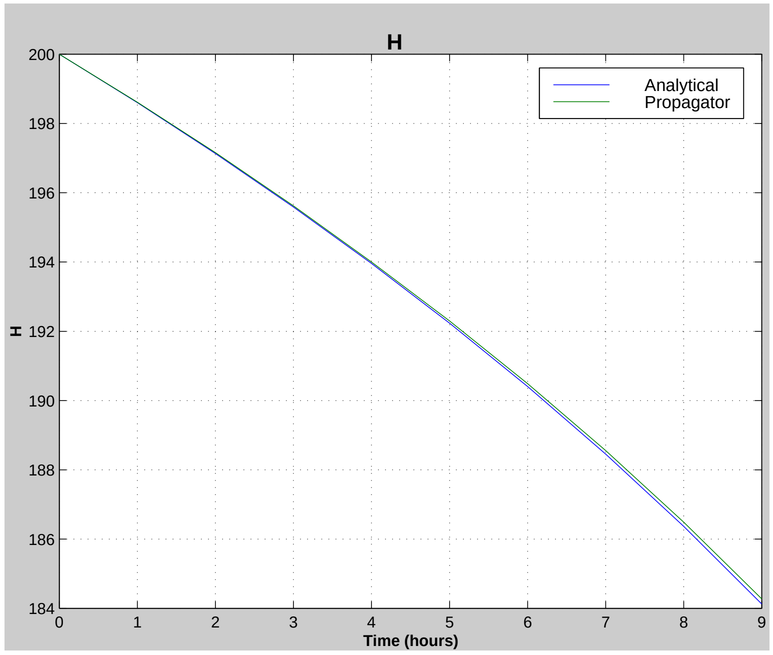
Results

Type

```
DragDemo
```

As expected, the propagator shows slightly less decay due to the rotation of the atmosphere (Figure 17.2).

Figure 17.2: Reentry Due to Drag



17.3 Topex Ephemerides

Reference

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

Introduction

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

- Gravitational harmonics
- Solar pressure
- Sun and moon perturbations

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

Results

Type

`TopexDemo`

The Topex ECI ephemerides are shown in the Figure 17.3. The range of R and V do not vary much over the period of the orbit.

Figure 17.3: Topex Ephemeris Cycle 10

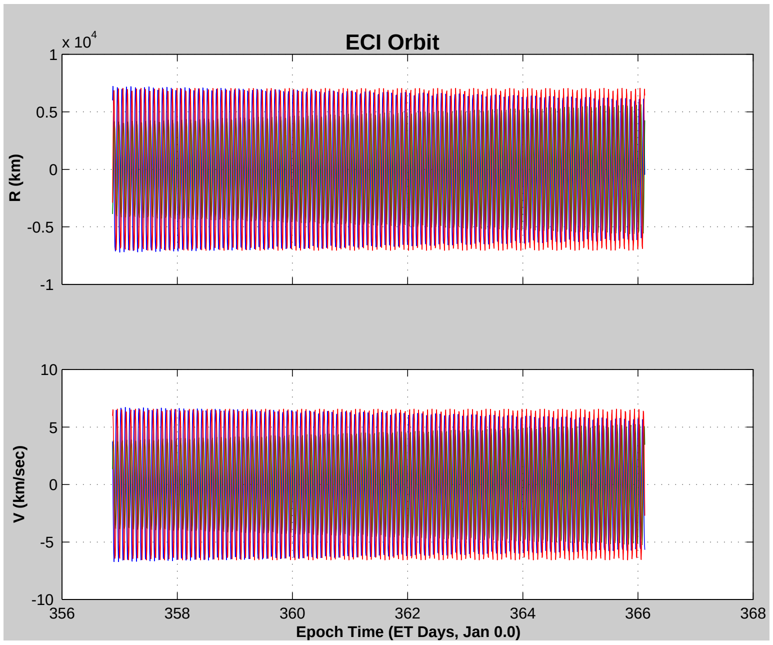


Figure 17.4 on the facing page shows the error between the Topex orbit and a perfect Keplerian orbit. The osculating elements at the start of the Topex file were used for the Kepler propagator. The osculating elements are given in Table 17.1 on the next page. The differences are enormous by the end of the simulation. Essentially, the Topex orbit is completely out of phase with the spherical model.

Figure 17.5 on the facing page shows the difference between the Topex Ephemeris and `PropagateOrbitPlugin` when just the GEM-T1 spherical earth model was used. The results (as expected) are the same.

Figure 17.6 on page 148 shows the Topex Orbit in 3D referenced to the earth-fixed frame.

Figure 17.7 on page 148 shows the difference when 4 GEM-T1 zonal harmonics are added. The results show a dramatic improvement in the tracking of the orbit. Finally, 4 tesseral harmonics are added in Figure 17.8 on page 149.

Table 17.1: Topex Osculating Elements at the Beginning of the Simulation.

Element	Value
a	7719.2 km
i	1.1528 rad
ω	-0.39392 rad
Ω	-0.94544 rad
e	0.00025586
M	0.52091 rad

Figure 17.4: Difference Between the Topex Ephemeris and a Keplerian Orbit

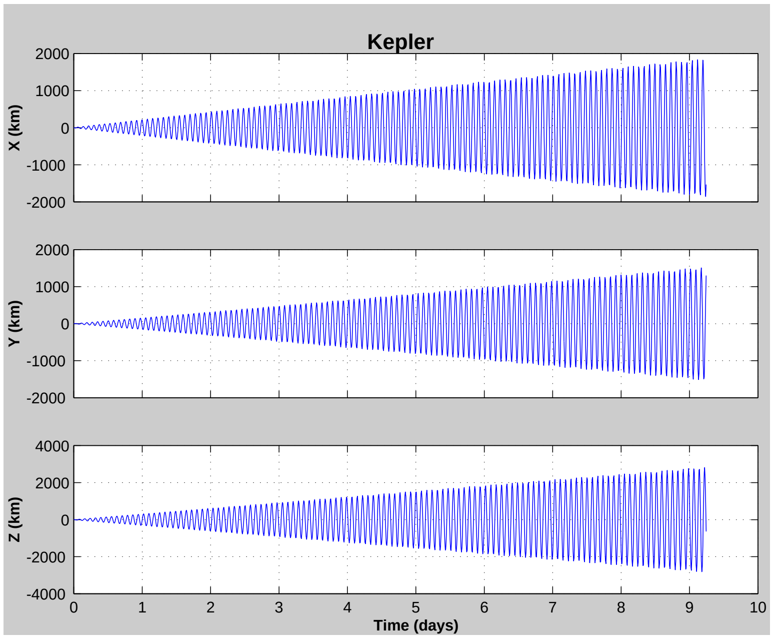


Figure 17.5: Difference Between the Topex Ephemeris and a Propagated Spherical Earth Model

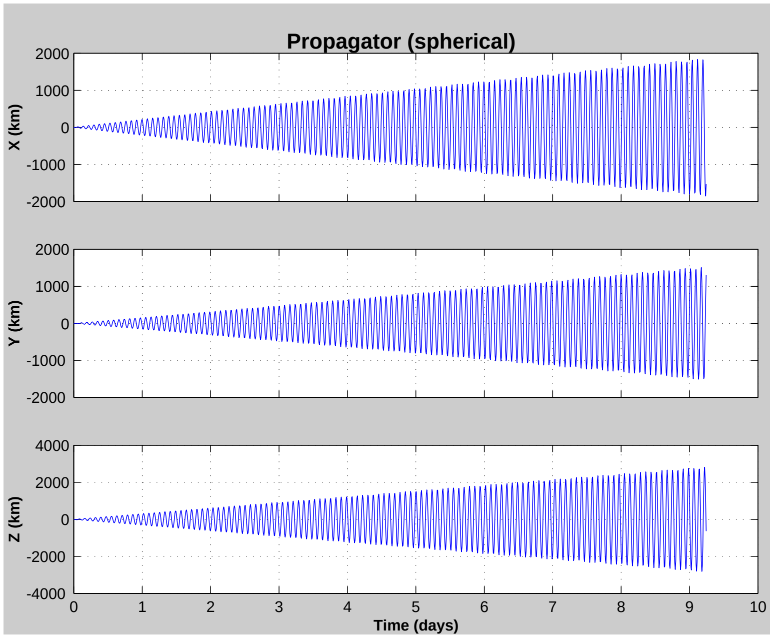


Figure 17.6: Topex Orbits in 3D

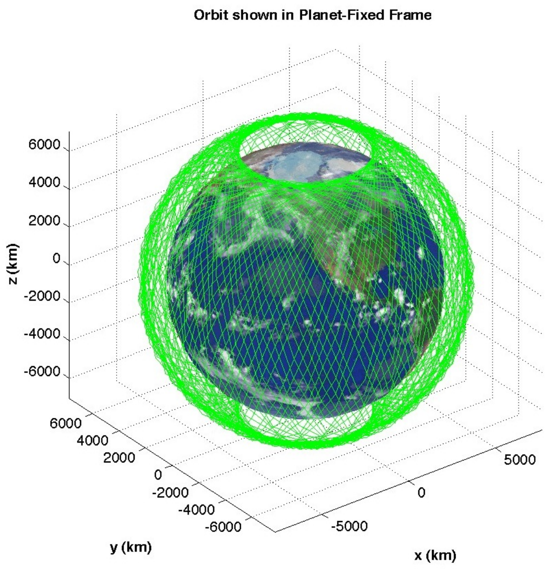
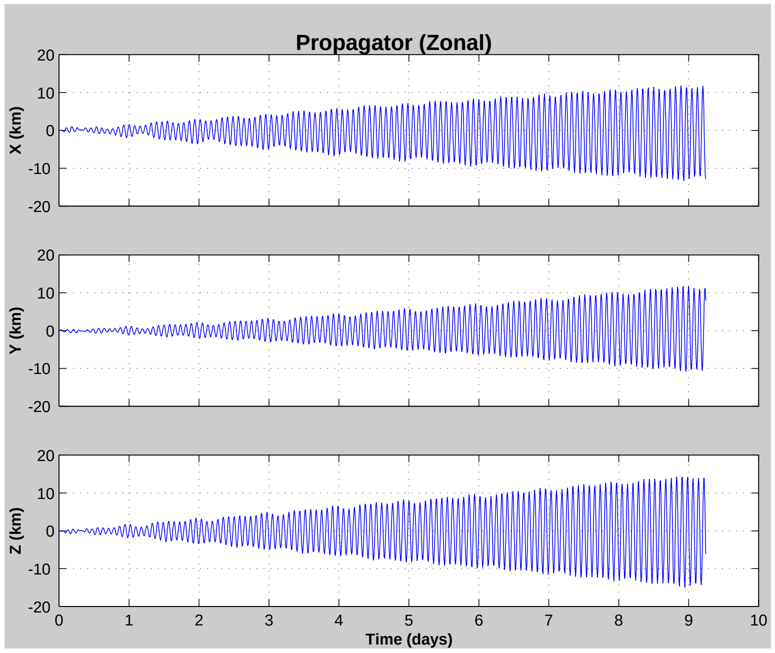
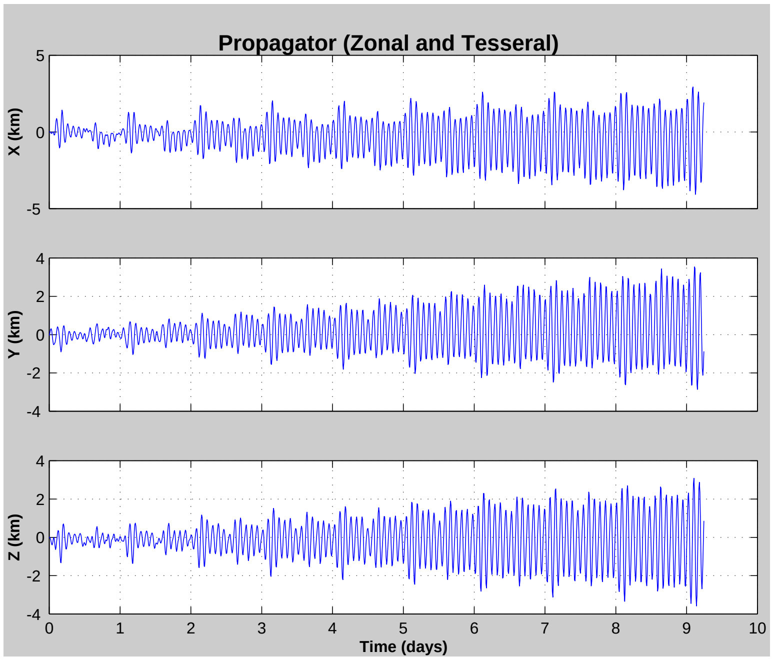


Figure 17.7: Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal Harmonics



Again, an improvement is noted.

Figure 17.8: Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal and 4 Tesseral Harmonics



17.4 Integration Tolerance

Reference

MATLAB User's Guide.

Introduction

There are two integration tolerance parameters for `ode113`; one is relative tolerance and one is absolute tolerance.

Results

Type

```
IntegrationAccuracyDemo
```

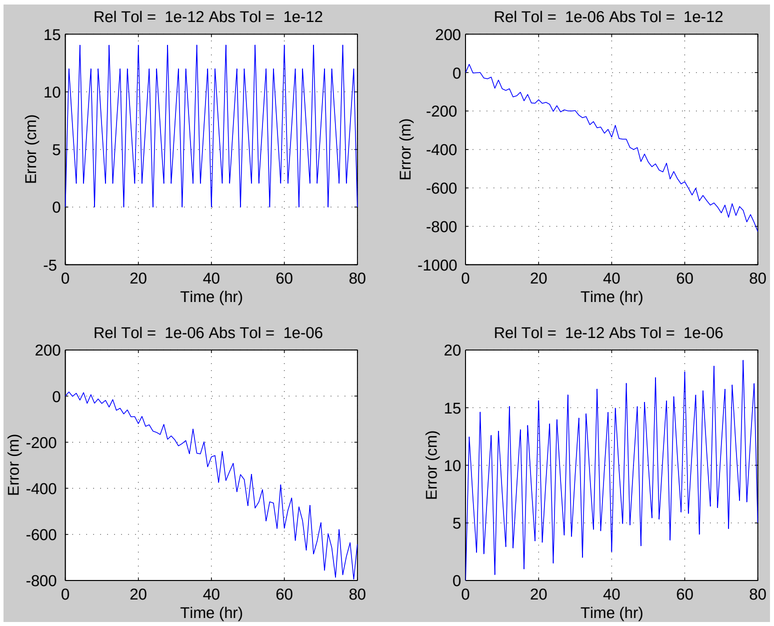
Figure 17.9 on the following page shows the effects of varying the integration tolerances for a 7000 km circular orbit.

The main diagonal plot units, with relative tolerances of 1e-12, are in centimeters while the off-diagonal plots, with relative tolerances of 1e-6, are in meters.

The differences are substantial. If you used the tolerances in the upper right hand corner, and didnt know that the orbit was unperturbed, you might think that the orbit was decaying! This also shows the danger of comparing one simulation against another. Unless you know the error tolerances for both runs, and have a good idea how the error correction algorithms work in both cases, your comparison could be meaningless.

You always want to get your simulations done as fast as possible. This is particularly true if you are doing Monte Carlo type analyses. Tighter tolerances mean longer simulation times. A balance must be made between integration

Figure 17.9: Effect of Integration Tolerances on Orbit Propagation



accuracy and time.

The number of samples output does not have an effect on simulation time.

The durations for the four simulations are given in the following table. These durations include all graphics generated at the end of the simulations.

Table 17.2: Simulation Times for a 140 hour Circular Orbit Simulation run on a 500 MHz Apple PowerBook G3

Rel Tol	Abs Tol	Duration (sec)	Accuracy
1e-12	1e-12	76.3	15 cm
1e-06	1e-12	36.2	1000 m
1e-06	1e-06	37.6	1000 m
1e-12	1e-06	62.4	20 cm

CHAPTER 18

ORBIT MANEUVERS

This chapter shows you how to plan orbit maneuvers.

18.1 Orbit Maneuver Functions

The following listing from `OrbMnvrDemo` will result in a demo of many of the orbit maneuver functions.

Listing 18.1: Orbit maneuver demonstration script

```
OrbMnvrBiElliptic
OrbMnvrHohmann
OrbMnvrInclination
OrbMnvrOneTangent
OrbMnvrLongitude
OrbMnvrLongAndIncl
OrbMnvrSemimajor
```

The following listing shows the results of running the script. Each function has a built-in demonstration to illustrate the inputs and outputs of the function. Each implements a relatively simple maneuver and they are useful for developing maneuver strategies.

Listing 18.2: Results

```
Bi-elliptic Transfer
------------------------------------------------
Initial Orbit Radius = 6569.5100
Final Orbit Radius = 382689.9000
Intermediate Orbit Radius = 510253.2000
------------------------------------------------
Delta V Total = 3.9040
Delta V at A = 3.1562
Delta V at B = 0.6774
Delta V at C = 0.0705
Time of Flight = 593.9238 hours

Hohmann Transfer
------------------------------------------------
Initial Orbit Radius = 6569.5091
Final Orbit Radius = 42159.5136
------------------------------------------------
Delta V Total = 3.9352
Delta V at A = 2.4570
Delta V at B = 1.4782
```

```
Time of Flight = 5.2567 hours

Inclination Correction
------------------------------------------------
Velocity = 3.5680
Eccentricity = 0.3000
True anomaly = 150.0000 deg
Inclination Correction = 15.0000 deg
------------------------------------------------
Delta V Total = 0.9129
Flight Path Angle = 11.4559 deg

One-Tangent Transfer
------------------------------------------------
Initial Orbit Radius = 6569.5091
Final Orbit Radius = 42159.5136
True anomaly = 160.0000 deg
Type = periapsis
------------------------------------------------
Delta V Total = 4.6993
Delta V at A = 2.5754
Delta V at B = 2.1239
Eccentric Anomaly = 127.8053 deg
Time of Flight = 3.4574 hours

Longitude Correction
------------------------------------------------
Velocity = 5.8923
Inclination = 55.0000 deg
Longitude Correction = 45.0000 deg
------------------------------------------------
Delta V Total = 3.6942
Initial latitude = 103.3647 deg
Final latitude = 76.6353 deg

Longitude Correction
------------------------------------------------
Velocity = 5.8923
Inclination = 0.0000 deg
Longitude Correction = 45.0000 deg
------------------------------------------------
Delta V Total = 3.6159
Initial latitude = 128.9041 deg
Final latitude = 97.3803 deg

Semimajor Axis Correction
------------------------------------------------
Velocity = 7.7843
Semimajor axis = 6578.0000
Semimajor axis Correction = 13.0000
------------------------------------------------
Delta V Total = 0.0077
```

18.2 Launch and Early Orbit Operations

The toolbox can be used for Launch and Early Orbit Operations (LEOP) analysis. `LEOPAnalysis.m` demonstrates these capabilities. Liquid apogee engines (LAEs) allow for precise control of insertion burns leading to less of a need to use the mission orbit system. However, because LAEs have much lower thrusts than solid motors it is necessary to perform multiple burns (compared to one for the solid). The script allows you to select a burn sequence for geosynchronous orbit transfer using a single array

```
burn = [0 0.5 0 0.3 0 0.15 0 0.05 0]
```

In this case nine revolutions (orbits) are in the plan and there are burns on rev 2, 4, 6 and 8. The number is the percentage of the total delta-v to be done on each burn. The script uses the rocket equation to compute the total amount of fuel needed for the spacecraft. All burns are done around apogee so the required delta-v vector is

$$\Delta v = v_{drift} - v_{transfer}$$

where drift refers to the orbit after the final burn because the insertion point is usually not the final station and the spacecraft "drifts" to station. Transfer orbit is the orbit upon separation from the launch vehicle.

Figure 18.1 shows the transfer orbit and the semi-major axis, inclination and eccentricity. The five distinct orbital paths correspond to the initial transfer orbit followed by the four burn sequence. The script uses MATLAB's `ode113` for orbit propagation and uses ode113's "events" capability to find apogee on each burn rev. The entire script is only 172 lines long and can be easily customized to provide more sophisticated burn planning.

Figure 18.1: Launch and early orbit operations analysis

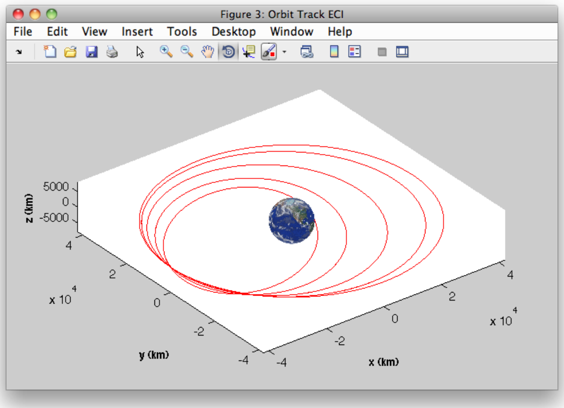


Figure 3: Orbit Track ECI
File Edit View Insert Tools Desktop Window Help
z (km)
5000
0
-5000
x 10^4
y (km)
x (km)
x 10^4
4
2
0
-2
-4

The dynamical model uses a point mass Earth model, although higher fidelity models can easily be used instead (the toolbox includes a 36th order gravity model). In this example the final burn leaves the spacecraft on station. In practice, it would be left with some drift velocity. In addition, the burn vector is held constant when small adjustments might need to be made due to orbit determination errors and orbit perturbations.

18.3 Drag Correction Example

This example shows how to compute the compensation for drag. The script is shown in Listing 18.3. The demo first compares an analytical model of drag with the orbit propagator. It then computes osculating elements for each (r, v) pair. The osculating elements are plotted. The trend shows a steady decrease in semimajor axis. This delta is then compensated for by a burn at perigee computed using one of the maneuver functions.

Listing 18.3: Drag correction script *DragCompensationDemo.m*

```
mu          = Constant('mu_earth');
rE          = Constant('equatorial_radius_earth');
area        = 1; % m^2
mass        = 100;
cD          = 2.7;
bStar       = cD*area/mass;
H0          = 200;
H1          = 200;
H2          = 180;
```

```
rho1        = AtmDens2( 200 );
rho2        = AtmDens2( 180 );
h           = (H2-H1)/log(rho1/rho2);
rho0        = rho1/exp(-H1/h);
dT          = 3600;
nSim        = 10;

t           = (0:(nSim-1))*dT;
xPlot       = zeros(2,length(t));
xPlot(1,:) = h*log( exp(H0/h) - sqrt(mu*rE)*bStar*rho0*1e3*t/h );

% Run the orbit propagator in batch
%----------------------------------
d           = load('DragCompensationDemo.mat');
tag         = PropagateOrbitPlugIn( 'initialize' );
PropagateOrbitPlugIn( 'set␣data', tag, d );
PropagateOrbitPlugIn( 'propagate', tag );

r           = PropagateOrbitPlugIn( 'get␣r', tag );
v           = PropagateOrbitPlugIn( 'get␣v', tag );
nP          = size(r,2);

if( ~isempty(r) & ~isempty(v) & nP > 0 )
  t           = t(1:nP);
  xPlot       = xPlot(:,1:nP);
  xPlot(2,:) = RV2AE( r, v, mu ) - rE;
  [t, xLabl] = TimeLabl( t );
  [h, hA]     = Plot2D( t, xPlot, xLabl, 'H' );
  legend( hA.h, 'Analytical', 'Propagator' );

  n = nP;
  for k = 1:n
      el(:,k) = RV2El( r(:,k), v(:,k) )';
  end

  yL = ['a';'i';'W';'w';'e'];
  Plot2D( t, el(1:5,:), 'Time', yL, 'Elements' );

  % Compute the change in semimajor axis
  %-------------------------------------
  deltaA = el(1,1) - el(1,end);
  deltaV = OrbMnvrSemimajor( Mag(v(:,1)),el(1,1),deltaA);
  disp(sprintf('Drag␣compensation␣delta-V␣is␣%12.4f␣km/sec',deltaV.total));
end
```

DragCompensationDemo.m

This demo also shows you how to use the orbit propagator in batch mode. When you execute the script the Orbit Propagator GUI appears as shown in Figure 18.2 on the next page. The aero model inputs are shown as well.

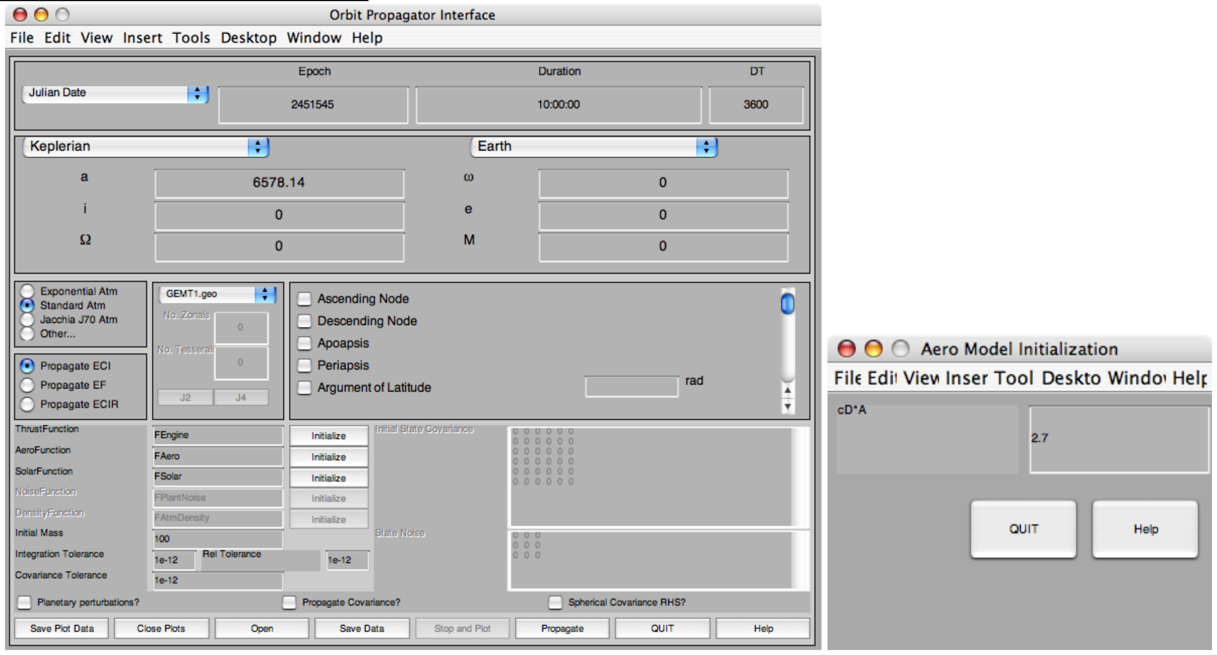
The model assumes the vehicle is a sphere with a simple drag coefficient. The first plot (Figure 18.3 on the facing page) shows the analytical and propagated results. The next plot shows the variation in the orbital elements.

Since the orbit is circular the argument of perigee is not well-defined. The variation in the semimajor axis is nearly linear. This variation is used to compute the required delta-V.

The required delta-V is displayed in the command window.

```
>> DragCompensationDemo
Drag compensation delta-V is       0.0077 km/sec
```

Figure 18.2: Orbit propagation GUI and aero model inputs


Orbit Propagator Interface
File Edit View Insert Tools Desktop Window Help
Aero Model Initialization
File Edit View Insert Tools Desktop Window Help
Initialize
Save Plot Data
Close Plots
Open
Save Data
Stop and Plot
Propagate
QUIT
Help

Julian Date

Epoch: 2451545

Duration: 10:00:00

DT: 3600

Keplerian

Earth

a: 6578.14

i: 0

Ω: 0

ω: 0

e: 0

M: 0

[ ] Exponential Atm [✓ Standard Atm] [ ] Jacchia J70 Atm [ ] Other...

[✓ Propagate ECI] [ ] Propagate EF [ ] Propagate ECIR

GEMT1.geo

No. Zonals: 0

No. Tesserals: 0

J2 J4

[ ] Ascending Node [ ] Descending Node [ ] Apoapsis [ ] Periapsis [ ] Argument of Latitude rad

ThrustFunction: FEngine

AeroFunction: FAero

SolarFunction: FSolar

NoiseFunction: FPlantNoise

DensityFunction: FAtmDensity

Initial Mass: 100

Integration Tolerance: 1e-12

Rel Tolerance: 1e-12

Covariance Tolerance: 1e-12

Initial State Covariance

State Noise

[ ] Planetary perturbations? [ ] Propagate Covariance? [ ] Spherical Covariance RHS?

cD*A: 2.7

Figure 18.3: Comparison of analytical drag with propagated

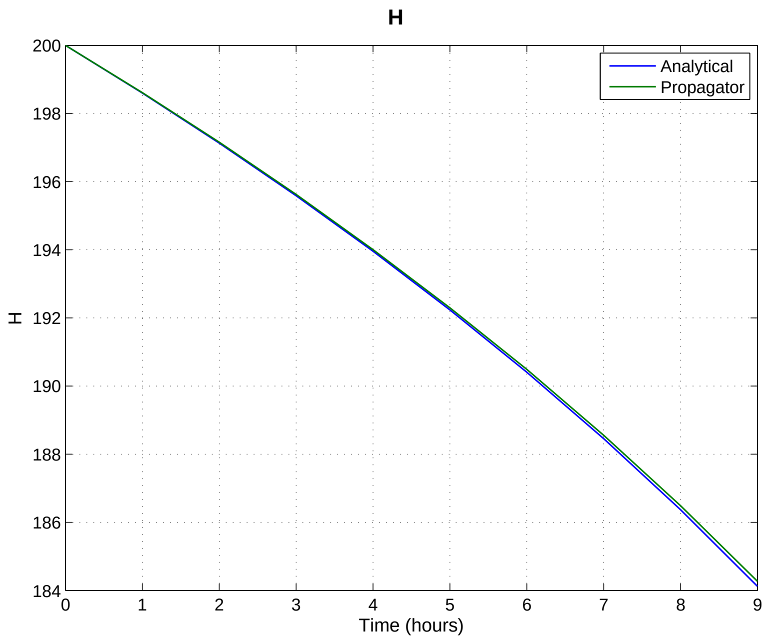
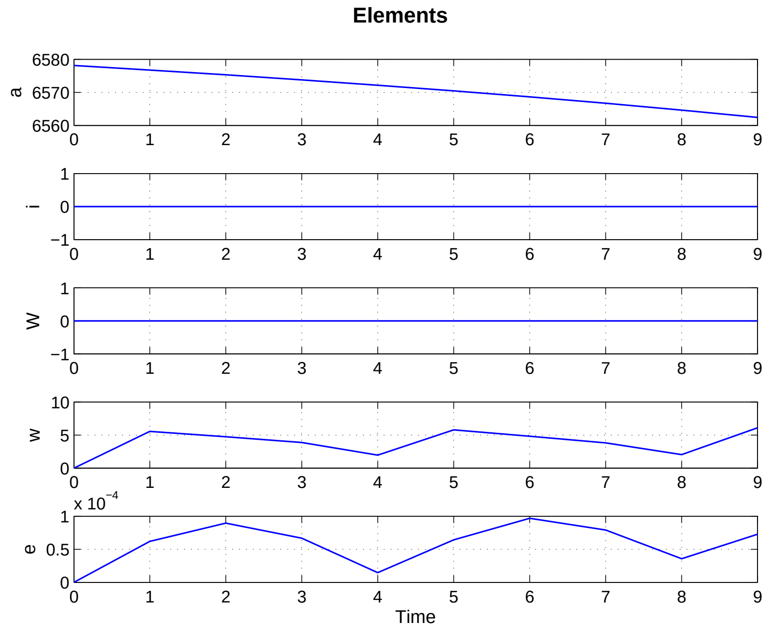


Figure 18.4: Orbital elements



18.4 Glideslopes

A glideslope is a trajectory where range rate is proportional to range. This can be used for rendezvous and proximity operations, however, these algorithms are only applicable to circular orbits. The toolbox has functions for generating maneuvers for inbound and outbound glideslopes. The same functions can also be used for circumnavigation in a set number of pulses.

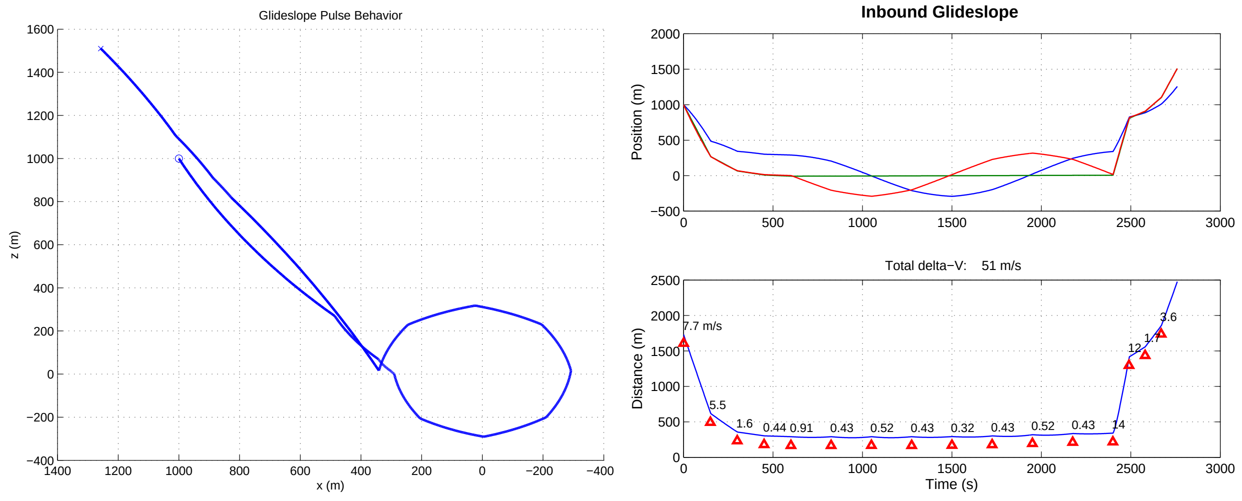
The functions in glideslope have built-in demos. The function `GlideslopePlanner` calculates sequential glideslope maneuvers. The default data contains three segments, an inbound slope, a circumnavigation, and an outbound slope. The Clohessy-Wiltshire equations are used to propagate in between segments.

Listing 18.4: Default data for the built-in demo *GlideslopePlanner.m*

```
%--------------------------------------------------------------------------------
%  Default data for a planning demo
%--------------------------------------------------------------------------------
function d = DefaultData

% Inbound slope
d(1).x0      = [1000; 1000; 1000; 0; 0; 0];
d(1).rTarget = [300; 0; 0];
d(1).V       = -[17.56 0.084];
d(1).T       = 10*60;
d(1).N       = 4;
% Circumnavigation
d(2).rTarget = 300;
d(2).V       = [];
d(2).T       = 30*60;
d(2).N       = 8;
d(2).a       = [0;1;0];
% Outbound slope
d(3).rTarget = [750; 750; 750];
d(3).T       = 6*60;
d(3).N       = 4;
d(3).V       = [1 10];
```
GlideslopePlanner.m

The results of this demo are shown in Figure 18.5. The maneuvers are indicated by red triangles.

Figure 18.5: Glideslope demo results



CHAPTER 19

ORBIT PROPAGATOR GUI PLUGIN

This chapter shows you how to propagate orbits and perform related tasks using the Propagator Plug In tool, `PropagateOrbitPlugIn`. This tool enables you to create some very elaborate propagations without writing lengthy scripts. The functions can be found in the `OrbitPropagator` folder.

19.1 The Orbit Propagator Tool

19.1.1 Introduction

The function `PropagateOrbitPlugIn` allows for 3 degree-of-freedom propagation of orbits. It can be used interactively or in batch mode. It has the following features:

- User selectable gravity models. GEM-T1, JGM-2, JGM-3 and WGS-84 are supported. Other models can be downloaded.
- Ballistic propagation of covariance matrix for state vector. A plug in function allows the user to control the plant noise covariance matrix.
- Propagate in ECI, ECR (earth-fixed) or ECIR. The last is an inertial frame that is coincident with EF at time 0.
- The user has complete control accuracy of numerical integration. The propagator uses the MathWorks' state-of-the-art `ode113` propagator.
- The user can select the time step. The time step can be variable and negative. It can be defined by any MATLAB expression or function.
- The user can specify any one of over thirty stopping conditions.
- Flight path angle, altitude, geodetic latitude and longitude are automatically computed.
- Outputs can be saved to a mat-file for further analysis.
- Sun/moon/earth perturbations can be added if you are propagating in an earth-centered or moon-centered frame.

19.1.2 Limitations

The following are some limitations that will be addressed in a future version:

- The propagator does not switch centers when the trajectory leaves one planet's sphere of influence and enters another.

- Only the moon and sun are added as perturbations when the earth is the center and the sun and earth are entered as perturbations when the moon is the center.
- Only centers for which the toolbox has pictures are included. This leaves out many moons.
- Only Mean-of-Aries 2000, Earth-Fixed and inertial coincident with earth-fixed frames are available.
- The noise function is not computed automatically.

19.2 Using the Graphical User Interface

The graphical user interface displays all options for using the propagator.

Figure 23-1 Orbit Propagation GUI

19.2.1 Overview

The orbit propagator interface, which is created by `PropagateOrbitPlugIn.m`, allows you to propagate any type of orbit using `ode113`, the MathWork's state-of-the-art propagator.

The GUI has several different "panes." These are summarized below. The following sections discuss each in more detail.

- Time Pane - set the time
- Elements Pane - set the initial orbital elements. These are always referenced to the ECI frame
- Atmosphere Pane - select one of four atmosphere models
- Propagation Pane - select the coordinate frame in which to propagate
- Gravity Model Pane - select the gravity model and the order of the model
- Stopping Conditions Pane - select the stopping condition for the simulation
- Customization Pane - select simulation options and functions you wish to plug into the simulation
- Buttons - control the simulation

The propagator can be run interactively or as part of a MATLAB script.

19.2.2 Panes

Time Pane

The time plug-in, in the upper right corner of the Propagator window, contains the values for the epoch, maximum duration, and time step of the next propagation run. This plug-in allows you to convert the epoch between Julian Date and UTC Calendar date. The duration field allows you to enter a maximum duration for the run. If the specified stopping condition is not met within the specified duration the run will stop. The default duration is one hour. The DT field allows you to specify the time step of the propagation. The default DT is 100 seconds. DT may be negative.

A set of times can be entered for the duration. For example you could enter:

```
0:25 hr
```

or

```
linspace(0,60000,1000) sec
```

or

```
[0 1 7 9 11 22 43 900] min
```

or

```
[900 43 22 11 9 7 1 0] hr
```

or

```
MyTimeSequence hr
```

or

```
3 orbits
```

Any MATLAB command or function can be entered into duration. If anything except a time is entered, DT is ignored. The output will be computed at these specific times only. The numerical integration routine will select the appropriate time step to achieve the specified numerical accuracy independent of the entered times.

Elements Pane

The elements plug-in allows you to enter and convert between Keplerian elements, equinoctial elements, and RV elements. The planet used for the orbit center is also shown. Element sets are given in the following table. The RV elements are position and velocity vector.

Table 19.1: Element sets

Keplerian		**RP/RA**		**Equinoctial**		**RV**
a	Semi-major axis	R_p	Perigee radius	a	Semi-major axis	r_1
i	Inclination	i	Inclination	P_1		r_2
Ω	Right Ascension of the Ascending Node	Ω	Right Ascension	P_2		r_3
ω	Argument of Perigee	ω	Argument of Perigee	Q_1		v_1
e	Eccentricity	R_a	Apogee radius	Q_2		v_2
M	Mean anomaly	M	Mean anomaly	l	Mean longitude	v_3

Atmosphere Pane

You can select any one of four models. The first two are altitude dependent and are based on a spherical planet model. The third will enable the default Atmospheric density plugin function, `FAtmDensity`, which implements the Jacchia J70 model. Other... allows you to enter your own atmosphere model.

Propagate Pane

This selects the frame in which to propagate. The ECIR frame is an inertial frame which coincides with the EF frame at the start of the simulation.

Gravity Model Pane

The Database model is a spherical planet model with the gravitational parameter extracted from the `Constant` function. Currently, this the only model you can use for planets or moons other than the earth.

You can download GEMT*, JGM-* and WGS84 models from various websites. Any file with the suffix .GEO will be displayed in the pull-down menu. You can enter the number of zonal and tesseral harmonics you wish to use. If you just want the J2 or J4 zonal, you can hit the button to set the appropriate model order.

Stopping Conditions Pane

You can select any combination of stopping conditions listed in the pane. For each you can enter a tolerance. Some require data. The stopping conditions are described in the following table. Some of the conditions aren't relevant to true Keplerian orbits in which the orbital elements do not change.

Table 19.2: Stopping Conditions

Stopping Condition	Explanation
Impact	Stop when orbit crosses the central body ellipsoid. This is the same as setting the geodetic altitude to zero.
Ascending Node	Stop at the ascending node.
Descending Node	Stop at the descending node.
Apoapsis	Stop at apoapsis.
Periapsis	Stop at periapsis
Argument of Latitude	Stop when $\omega + \nu$ = value
XZ Crossing	Stop when ECI y = 0
XY Crossing	Stop when ECI z = 0
YZ Crossing	Stop when ECI x = 0
True Anomaly	Stop when ν = value
Approaching Distance	Stop when r = value and is decreasing
Receding Distance	Stop when r = value and is increasing.
Increasing Velocity	Stop when v = value and is decreasing
Decreasing Velocity	Stop when v = value and is increasing.
Flight Path Angle	Stop when the flight path angle = the value
Perigee Height	Stop when perigee radius = value
Apogee Height	Stop when apogee radius = value
Geodetic Latitude	Stop when geodetic latitude = value
Geocentric latitude	Stop when geocentric latitude = value
Geodetic Altitude	Stop when geodetic altitude = value
Semi major Axis	Stop when a = value
Mean Anomaly	Stop when M = value
Orbital Period	Stop when orbital period = value
Right Ascension of Ascending Node	Stop when Ω = value and only at the descending node
Right Ascension of Descending Node	Stop when Ω = value and only at the ascending node
Change in Inertial Velocity Magnitude	Stop when $\|v\|$ = value
Vehicle Longitude	Stop with earth-fixed longitude = value
Argument of Perigee	Stop when argument of perigee = value
Eccentric Anomaly	Stop when E = value

Customization Pane

The first four buttons are the names of functions the propagator should call. The default names in the GUI are default functions. When you hit Initialize, the function will display an initialization GUI.

The Noise Function returns the plant noise matrix, q, for the covariance propagation:

pDot = f*p + p*f' + q

where f is the matrix of partials of the right-hand-side of the orbit.

The initial mass is the mass of the spacecraft.

The next two parameters relate to the integration accuracy.

Rel Tolerance gives the relative error tolerance. The estimated error in each integration step satisfies

```
e(i) <= max(RelTol*abs(y(i)),Tol(i))
```

Integration Tolerance gives the absolute error tolerance which applies to all components of the solution vector.

The next two checkboxes allow you to select planetary perturbations and/or to propagate the covariance. When you select planetary perturbations you will get the moon and the sun if the earth is selected as the center through the elements pane or the earth and the sun if the moon is selected as the center. If you select other planets or moons you will not get any perturbations.

When you select Propagate Covariance the initial covariance input will be enabled. You should enter the initial state covariance into the box. You need only enter upper or lower triangular information. The propagator automatically does

```
p = 0.5*(p + p')
```

to insure that the covariance matrix is symmetric. You have the option to select Spherical Cov. This uses a spherical earth model just to propagate the covariance. This is much faster than using the full model.

Buttons

The following is the list of buttons in the GUI.

- Save Plot Data saves the plot data in a mat file of type *.OPP. The plot is saved as a matrix with rows: [x;y;z;vX;vY;vZ;mass;pXX;....pVZVZ;pXY;pYZ...] p is saved in columns starting with the main diagonal and moving up. For example, a 3x3 matrix would be saved as [a11;a22;a33;a12;a23;a13];
- Close PlotsClose the plot windows.
- OpenOpen a gui dat file of type *.OPG.
- Save DataSave the gui data in a mat file of type *.OPG.
- PropagateRun the propagator.
- QUITQuit. It will ask you to save unsaved plot and GUI data.
- HelpGet help.

19.3 Plugin Functions

The orbit propagator has four plugin in functions that allow you to customize the simulation. When you first bring up the GUI four default functions are shown. They are described in Table 19.3.

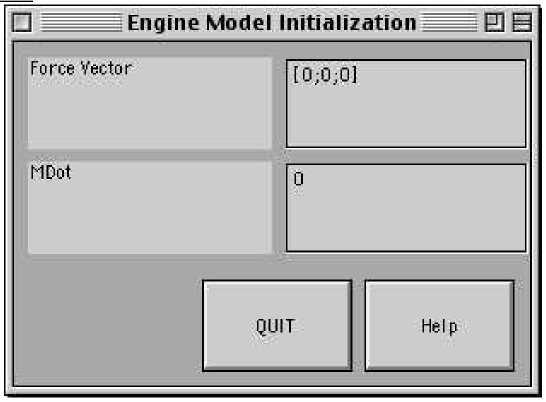
Table 19.3: Plugin Functions

Function	Description
`FEngine`	Computes engine force and spacecraft mass. The force is in the LVLH frame.
`FAero`	Computes aero forces in the ECI frame.
`FSolar`	Computes solar forces in the ECI frame.
`FPlantNoise`	The Q matrix in the covariance propagation equation.
`FAtmDensity`	Atmospheric density. The baseline model is Jacchia J70.

Each model supplied with the toolbox has its own GUI. These popup when you hit the Initialize button next to each function's name.

The `FEngine` model, Figure 19.1 on the next page, allows you to enter a constant force in the ECI frame and a constant mass change rate.

Figure 19.1: `FEngine` GUI

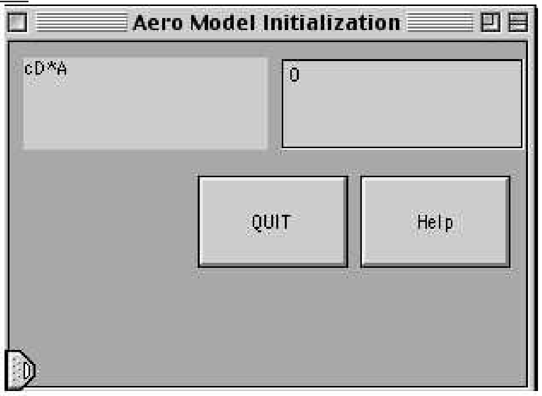

Engine Model Initialization

Force Vector [0;0;0]

MDot 0

The `FAero` model, Figure 19.2, assumes a flat plate spacecraft with the plate always normal to the velocity vector. The input is the product of the drag coefficient and plate area.

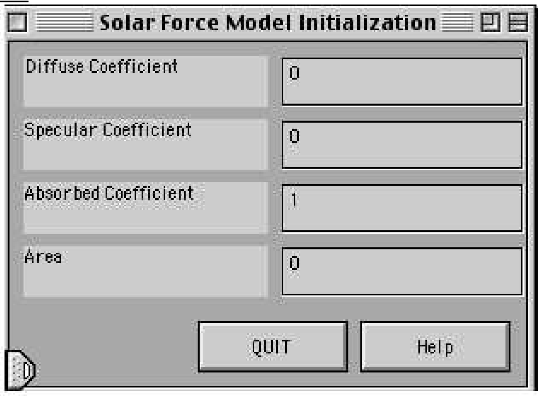
Figure 19.2: `FAero` GUI


Aero Model Initialization

cD*A 0

The `FSolar` model, Figure 19.3, is a flat plat model in which the plate is always normal to the sun vector.

Figure 19.3: `FSolar` GUI


Solar Force Model Initialization

Diffuse Coefficient 0

Specular Coefficient 0

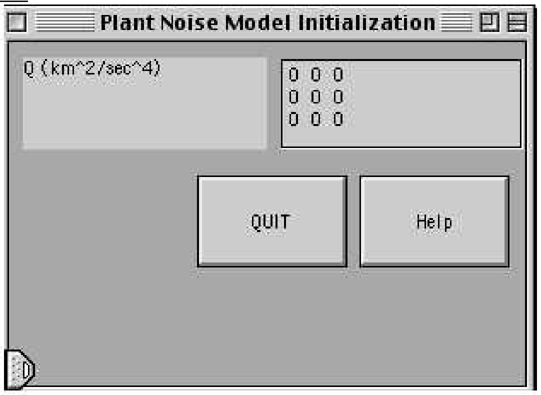
Absorbed Coefficient 1

Area 0

`FPlantNoise` function, Figure 19.4 on the facing page, assumes a constant plant noise covariance matrix. The inputs are the uncertainty in body accelerations.

The `FAtmDensity` GUI, Figure 19.5 on the next page, gives inputs for the Jacchia J70 model. The J70 inputs are

Figure 19.4: `FPlantNoise` GUI

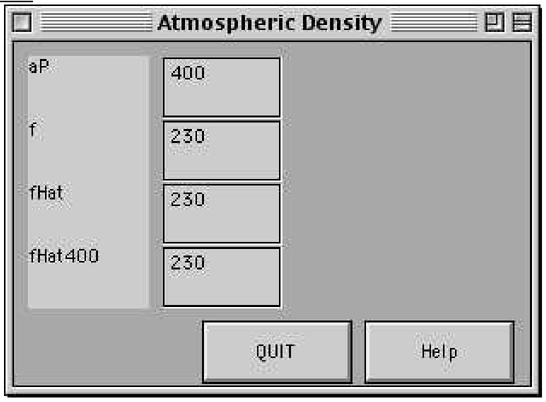


Q (km^2/sec^4)

```
0 0 0
0 0 0
0 0 0
```

given in Table 19.4 on the facing page.

Figure 19.5: `FAtmDensity` GUI



aP	400
f	230
fHat	230
fHat400	230

Table 19.4: Jacchia J70 Atm inputs

Input	Description
aP	Geomagnetic index 6.7 hours before the computation
f	Daily 10.7 cm solar flux (e-22 watts/m^2/cycle/sec)
fHat	81-day mean of f (e-22 watts/m^2/cycle/sec)
fHat400	fHat 400 days before computation date

19.4 Building a Plugin Function

All plugin in functions have the same structure. In this section, we will walk you through the structure of `FAero`, the simplest function.

Listing 19.1: Function Definition *FAero.m*

```
function x = FAero( action, modifier, u, t )
```

FAero.m

Listing 19.2 on the next page shows the switch statement that makes up the main function. `action` is the argument for the switch statement and `modifier` is the GUI tag, which identifies the GUI.

Listing 19.2: Main Function Body *FAero.m*

```
if( nargin < 1 )
  action = 'initialize';
end

switch action

  case 'initialize'
    x = Initialize( modifier );

  case 'update'
    x = Update( modifier, u, t );

  case 'get␣data'
    x = GetData( modifier );

  case 'store␣data'
    h   = GetH( modifier );
        h.d = GetData( modifier );
        PutH( h );

  case 'set␣data'
    SetData( modifier, u );

  case 'help'
    HelpSystem( 'initialize', 'OnlineHelp' );

  case 'quit'
    h = GetH( modifier );
        if( ishandle( h.propagateOrbitPluginTag ) )
          PropagateOrbitPlugIn( 'close␣function', h.propagateOrbitPluginTag, 'FAero' );
        end
    CloseFigure( h.fig )

end
```

FAero.m

The `Initialize` subfunction draws the GUI and returns the tag for the figure.

Listing 19.3: Initialize *FAero.m*

```
function tag = Initialize( propagateOrbitPluginTag )

name      = 'Aero␣Model␣Initialization';
position  = [5 5 300 200];
tag       = GetNewTag( name );
h.fig     = figure( 'position', position,'color',[0.66 0.66 0.66], 'NumberTitle', 'off',...
                    'name', name, 'resize', 'off', 'tag', tag, 'CloseRequestFcn',
                        CreateCallback( 'quit', tag ) );
fontSize  = position(4)*9/185;
v         = { 'parent', h.fig, 'fontunits', 'pixels', 'fontsize', fontSize, '
    horizontalalignment','left'};
space     = position(3)/40;
x0        = space;
dX        = position(3)/2;
xW        = dX - space;
dY        = (position(4) - 2*space)/3;
yW        = dY - space;

x         = x0;
y         = position(4) - dY;

% The orbit propagator tag
%-------------------------
h.propagateOrbitPluginTag = propagateOrbitPluginTag;

hMatlab6Bug      = uicontrol('visible','off'); % Added to fix a Matlab 6 bug
```

```

% Inputs
%-------
h.gui.cDAText    = uicontrol( v{:}, 'position', [x y xW yW], 'string', 'cD*A', 'style', 'text
    ' ); x = x + xW + space;
h.gui.cDA        = uicontrol( v{:}, 'position', [x y xW yW], 'string', '0',    'style', 'edit
    ' ); x = x0; y = y - dY;

% GUI controls
%-------------
xW               = (position(3) - 5*space)/3;
x                = position(3) -  xW - space;
y                = y - space;

h.gui.help       = uicontrol( v{:}, 'position', [x y xW yW], 'string', 'Help', 'callback',
    CreateCallback( 'help', tag ) );  x = x - xW - space;
h.gui.quit       = uicontrol( v{:}, 'position', [x y xW yW], 'string', 'QUIT', 'callback',
    CreateCallback( 'quit', tag ) );  x = x - xW - space;

PutH( h )
```

FAero.m

The Inputs and GUI controls lines show you how to write uicontrols for edit boxes seen in the GUI. uicontrols require you to pass them parameters in pairs. To simplify the code we put a lot of the parameters in the cell array `v`. The `CreateCallback` subfunction (Listing 19.4) generates a call back to the `FAero` function. For example the last line creates the call:

```
FAero(  quit', tag )
```

Listing 19.4: CreateCallback *FAero.m*

```
function s = CreateCallback( action, modifier )

s = ['FAero(␣''' action ''',''' modifier ''')'];
```

FAero.m

See the MATLAB documentation on how to build GUIs for more information. The get data function gets data stored in the GUI and set data sets the data. `SetData` is used by the orbit propagation function when you pass it the saved propagator data structure. `GetData` is used internally as shown in the next listing.

Listing 19.5: Getting and setting data *FAero.m*

```
%---------------------------------------------------------------------------
%   Get the data from the gui
%---------------------------------------------------------------------------
function d = GetData( modifier )

h = GetH( modifier );

d.cDA = str2num( get( h.gui.cDA, 'string' ) );

%---------------------------------------------------------------------------
%   Set the data in the gui
%---------------------------------------------------------------------------
function SetData( modifier, d )

h = GetH( modifier );

set( h.gui.cDA,  'string', num2str( d.cDA ) );
```

FAero.m

The `Update` subfunction computes the aerodynamic drag. It gets the figure handle, gets the data from the figure handle, and combines it with the velocity stored in data structure `u`, to compute the force.

The next listing completes our review of `FAero`.

Listing 19.6: The Update subfunction *FAero.m*

```
function x = Update( modifier, u, t )

h   = GetH( modifier );
d   = GetData( modifier );

x   = -0.5*d.cDA*u.rho*Mag(u.v)*u.v*1e3; % kg-km/sec^2: m^2*(kg/m^3)*km^2/sec^2, kg-km^2/m
    sec^2
```

FAero.m

19.5 Reference Frames

19.5.1 Coordinate Transformations

`CoordinateTransform`, in the Common/Coord portion of the Core module, transforms between:

- ECI
- ECR
- Geodetic latitude, longitude and altitude

For example

```
r = [6524.834;6862.875;6448.296];
x = CoordinateTransform( 'eci', 'ef', r, 2449773 )
x =
   1.0e+03 *
   2.85101441661017
   9.03243563995358
   6.44514208556912

x = CoordinateTransform( 'eci', 'llr', r, 2449773 )
x =
   1.0e+03 *
   0.00059923067421
   0.00126505172630
   5.08520910899187
```

19.5.2 Finding ECI/ECR coincidence

`EarthRotationZero`, in the SC/Ephem portion of the Core module, finds a Julian date for which the Greenwich Apparent Sidereal Time (GAST) is zero.

For example

```
x = EarthRotationZero( JD2000 )
x =
   2.4515e+06
```

gives a Julian Date within a day of JD2000. For this date

```
GASTime( x )

ans =
7.2978e-09
```

You can also find the transformation matrix from ECI to ECR at any time by typing

```
m = ECIToECIR( jD )
```

m transforms from ECI to the frame aligned with EF at time jD. This constant offset frame can be used to propagate equations in the ECIR frame.

19.6 Sun

`SunV1`, also in SC/Ephem, provides a moderate precision (0.01 deg) sun angle.

```
uSun = SunV1( jD );
[rA, dec] = U2RADec( uSun )
```

The second function gives the right ascension and declination for the sun.

19.7 Interfacing to STK

You can interface to STK using the function `STKOrbit`. `STKOrbit` allows you to exchange data with STK by creating an input file for STK.

```
[err, message] = STKOrbit( fileName, ver, epoch, nPoints, time, position,
    velocity, type )
```

Table 19.5: I/O for STKOrbit

Variable	Description
fileName	Filename for output. Will be overwritten if it already exists.
ver	STK version number (string!)
epoch	Beginning date and time: [Y M D H M S]
nPoints	Number of orbit data points
time	Time in seconds for each data point
position	Position data in km (if type = ECF or ECI) or in [degrees,degrees,km] (if type = LLR)
velocity	Velocity data in km/sec (if type = ECF or ECI) or [degrees,degrees,km]/sec (if type = LLR)
type	'ECI','ECF',or 'LLR'
err	err = 0 means an error occurred
message	Error message

19.8 Ground Coverage

Ground coverage is found using this function from OrbitMechanics:

```
x = GroundCoverage( xECI, jD, d )
```

where `xECI` is the ECI state vector `[r;v]`, `jD` is the Julian date and `d` is the data structure in which `d.fOV.x` is the x axis field-of-view, `d.fOV.y` is the y-axis field-of-view and `d.m` transforms from the body frame to the sensor frame. The sensor is assumed to have its boresight along +z in the sensor frame. If `d.m` were the identity matrix the sensor would point along $+z$ in the body frame. For this function, the satellite body frame is assumed to be aligned with the LVLH frame.

`d.e` is a 3-by-1 vector with the [x dimension;y dimension;z dimension] of the ellipsoid. If the planet is spherical the three dimensions are the same.

19.9 Orbit Propagation API

19.9.1 Batch Runs

You can run `PropagateOrbitPlugIn` from a script. This allows you to set up large numbers of cases and run them automatically. `OPBatchDemo` shows how to set up a batch run.

```
d = load('OPDemo.mat');
d.epoch.dT = 3600;
tag = PropagateOrbitPlugIn( 'initialize' );
      PropagateOrbitPlugIn( 'set␣data', tag, d );
      PropagateOrbitPlugIn( 'propagate', tag );
      pause
      PropagateOrbitPlugIn( 'close␣plots', tag );
      PropagateOrbitPlugIn( 'quit', tag );
```

In the first line a saved data file is loaded. This file was created and saved using the GUI interactively. In the second the time step is customized. Any field in the data structure may be customized. In the third step the GUI is initialized. In the fourth the datastructure `d` is loaded. The orbit is then propagated. Plots are closed after the pause and the final step is to quit. tag identifies the GUI. Possible commands are listed in Table 19.6.

Table 19.6: PropagateOrbitPlugIn API

Prototype	Purpose
tag = PropagateOrbitPlugIn('initialize');	Initialize the plugin.
PropagateOrbitPlugIn('set data', tag, d);	Set the data in the GUI
PropagateOrbitPlugIn('propagate', tag);	Propagate the orbit.
PropagateOrbitPlugIn('close plots', tag);	Close the plots
PropagateOrbitPlugIn('save, tag, fileName);	Save the GUI data.
PropagateOrbitPlugIn('save plot data', tag, fileName);	Save the plot data
PropagateOrbitPlugIn('quit', tag);	Close the GUI.

19.9.2 Data Structure

The data structure `d` is defined in the following table.

Table 19.7: Input data structure for `PropagateOrbitPlugIn`

Field	SubField	Type	Description
f		Struct	Plugin function names and data
	thrustFunction	String (1,:)	Name of the thrust function
	aeroFunction	String (1,:)	Name of the aerodynamics function
	solarFunction	String (1,:)	Name of the solar pressure function
	noiseFunction	String (1,:)	Name of the covariance noise function
	densFunction	String (1,:)	Name of the atmospheric density function
	thrustData	User defined	Data for the users thrust function
	aeroData	User defined	Data for the users aerodynamic function
	solarData	User defined	Data for the users solar pressure function
	noiseData	User defined	Data for the users covariance noise function
	densData	User defined	Data for the users density function
	planetaryDisturbancesOn	(1,1)	1 turns on planetary disturbances
	atmosphere	String (1,:)	String for atmosphere model name
	nZonal	(1,1)	Number of zonal harmonics
	nTesseral	(1,1)	Number of tesseral harmonics
	stopping.condition	String (1,:)	Stopping condition string

covariancePropagationOn		(1,1)	Turns on covariance propagation
q		Struct	
	simpleCovarianceRHSOn	(1,1)	Simple covariance right-hand-side enabled
propagate		String (1,:)	Propagation method
mass		(1,1)	Spacecraft mass
epoch		Struct	Time epoch
	jDEpoch	(1,1)	Julian date (days)
	dT	(1,1)	Time step (sec)
	duration	(1,1)	Duration (sec)
tol		(1,1)	Absolute tolerance for ode113
reltol		(1,1)	Relative tolerance for ode113

19.9.3 Function Inputs

The following table lists all inputs to `PropagateOrbitPlugIn`.

Table 19.8: Inputs to `PropagateOrbitPlugIn`

Action	Inputs	Output	Description
initialize	None	None	Tells the GUI to initialize itself.
propagate	None	None	Propagate the orbit
propagate covariance	None	None	Propagate the covariance
atmosphere	None	None	Open the atmosphere GUI
gravity	None	None	Open the gravity GUI
set data	Data structure (see above)	None	Provide a data structure to the GUI
get data	None	GUI data structure	Returns the GUI data structure
get results	None	Plot data	Returns the plot data
get r	None	r(3,:)	Returns the position vector array
get v	None	v(3,:)	Returns the velocity vector array
get mass	None	(1,1)	Returns the spacecraft mass
save	String (1,:)	None	Save the data in the GUI
open	None	None	Open a saved file
save plot data	String (1,:)	None	Plot data file name
initialize function	None	None	Initialize the plugin functions
close function	String (1,:)	None	Close the input plugin
j2	None	None	Set the number of zonal harmonics to 2 and tesseral to 0
j4	None	None	Set the number of zonal harmonics to 4 and tesseral harmonics to 0
help	None	None	Open the help GUI
quit	None	None	Close the GUI

19.9.4 Function Outputs

The plot outputs for `PropagateOrbitPlugIn` are given in the following table.

Table 19.9: Plot outputs for `PropagateOrbitPlugIn`

Plot Data	Description	Units
x(1:3,:)	Position	km
x(4:6,:)	Velocity	km/sec
x(7,:)	Mass	kg
x(8,:)	Geodetic Latitude	deg
x(9,:)	Longitude	deg

x(10,:)	Altitude	km
x(11,:)	Flight path angle	rad
x(12:17,:)	Covariance diagonal	

CHAPTER 20

FUEL BUDGETS

This chapter discusses how to generate fuel budgets. These budgets are some of the most important elements of spacecraft system design.

20.1 Fuel Budget

Computation of fuel budgets is a straightforward bookkeeping exercise. The most important point to remember is that the fuel consumption of the thrusters is a function of system pressure. Consequently, fuel budgets must be done chronologically.

`FBudget` generates fuel budgets. There are four types of maneuvers supported. They are spin precession maneuvers, spin changes, AKM firings, and stationkeeping maneuvers. You set up each maneuver using the `FBudget` data structures as shown in the following example.

Listing 20.1: Fuel Budget: Moment Arms

```
% You need position vectors and moment arms.
r = [ -0.8 -0.8 -0.8 -0.8 0.8 0.8 0.8 0.8 0.8 -0.8 -0.8 0.8 0.2 -0.2 -0.2 0.2;...
      -0.9 -0.8 0.7 0.8 -0.8 -0.8 0.8 0.8 -0.8 -0.9 -0.8 -0.7 -0.8 -0.8 -0.8 -0.8;
       0.8 -0.8 -0.8 0.8 0.9 -0.8 -0.8 0.7 0.8 0.8 -0.8 -0.8 0.15 -0.2 -0.2 0.2];
u = [ -1 -1 -1 -1 1 1 1 1 0 0 0 0 0 0 0 0;...
       0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1;...
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

In the next listing the thruster performance data is collected into data structures.

Listing 20.2: Fuel Budget: Thruster performance data

```
for k = 1:12
  f(k).iSP = [100 100 10]; % See REA.m for use of this array
  f(k).thrust = 2.07e-6; % See REA.m for use of this array
  f(k).riseTime = 0.016; % In seconds
  f(k).fallTime = 0.016;
  f(k).u = u(:,k);
  f(k).r = r(:,k);
  f(k).type = 'liquid';
  f(k).systemID = Odd(k) + 1; % You can have as many fuel systems as you want
end
for k = 13:16
  f(k).iSP = [100 250 10];
  f(k).thrust = 0.828e-6; % In this case just multiplying this by pressure
  f(k).riseTime = 0.016;
  f(k).fallTime = 0.016;
  f(k).u = u(:,k);
  f(k).r = r(:,k);
  f(k).type = 'liquid';
  f(k).systemID = Odd(k) + 1;
end
```

The next listing has general spacecraft data. This includes details about the thruster system. In this case, there are two systems (often called half systems) each using liquid propellant thrusters with equal volumes, pressures and temperatures.

Listing 20.3: Fuel Budget: General spacecraft data

```
% Assemble general spacecraft data.
x.regulatedPressure = [250 250]*6895;
x.systemTemp = [298 298];
x.systemType = ['liquid';'liquid'];
x.systemVol = [1.6505e-01 1.6505e-01];
x.systemFuel = [1.1789e+02 1.1789e+02];
x.pressurantMass = [1.8372e-01 1.8372e-01];
x.pressurantR = MolWt2R( 0.004 )*[1 1];
x.dryMass = 9.5514e+02;
x.fuelDensity = [1000 1000];
x.thrusterData = f;
x.controlDT = 1;
k = 0;
```

The next set of listings define the fuel burning events for the budget. The event structure fields are defined in the header of `FBudget`, along with an indication of which events require which data. The definitions are copied below for reference.

Listing 20.4: Fuel Budget: Event structure fields

```
%   event (:)    Event data structure
%                event.description   (1,:) String describing the event
%                event.type          (1,:) 'initial', 'delta-v', 'delta-h', 'acs','pulses',
%                                          'circular'
%                event.aCSThruster   (1,:) List of ACS thrusters to use    <-  all
%                event.dVThruster    (1,:) List of deltaV thrusters to use <- 'delta-V'
%                event.dVTotal       (1,1) Total delta-V                   <- 'delta-V'
%                event.dTControl     (1,1) Control period                  <- 'delta-V', 'acs'
%                event.burnEff       (1,1) Burn efficiency                 <- 'delta-V'
%                event.dHTotal       (1,1) Total delta-H                   <- 'delta-H'
%                event.disturbance   (3,1) Attitude disturbance            <- 'acs'
%                event.pulsewidth    (1,1) Pulsewidth to be used           <-  all
%                event.nPulses       (1,1) Number of pulses                <-  pulse
%                event.duration      (1,1) Used for attitude control       <- 'acs'
%                event.regulated     (1,1) 1 if regulated at following level
%                event.cM            (3,1) Center of mass
%                event.circular      (1,1) struct('period',p,'radius',r,'duration',t)
%                                                                    <- 'circular'
```

The first event shows the initialization of the arrays.

Listing 20.5: Fuel Budget: Array initialization

```
% Event 1: Initialization
%-------------------------
k = k + 1;
event(k).type = 'initial';
event(k).description = 'Initial␣State';
event(k).aCSThruster = [];
event(k).dVThruster = [];
event(k).dVTotal = [];
event(k).dHTotal = [];
event(k).disturbance = [];
event(k).pulsewidthACS = 0;
event(k).pulsewidthDV = 0;
event(k).nPulses = [];
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];
```

The next shows an entry for a continuous Delta-V burn.

Listing 20.6: Fuel Budget: Continuous delta-v burn

```
% Event 2: A continuous delta-V burn using different ACS and DV thrusters
%-------------------------------------------------------------------------
k = k + 1;
event(k).type = 'delta-v';
event(k).description = 'Delta␣V␣Continuous';
event(k).aCSThruster = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster = [13 14];
event(k).dVTotal = 100; % Total delta V in m/sec
event(k).dHTotal = [];
event(k).dTControl = 1;
event(k).disturbance = [];
event(k).nPulses = [];
event(k).duration = [];
event(k).regulated = 0; % If not regulated use blowdown curves
event(k).cM = [0;0;0];
```

The next listing shows the entry for a delta-v burn with off-pulsing.

Listing 20.7: Fuel Budget: Delta-v with off-pulsing

```
% Event 3: Delta-V off-pulsing some thrusters
%--------------------------------------------
k = k + 1;
event(k).type = 'delta-v';
event(k).description = 'Delta␣V␣Off␣Pulse';
event(k).aCSThruster = [1 2 3 4 9 10 11 12];
event(k).dVThruster = [1 2 3 4];
event(k).dVTotal = 100;
event(k).dHTotal = [];
event(k).dTControl = 1;
event(k).nPulses = [];
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];
```

The next listing shows an unloading momentum entry.

Listing 20.8: Fuel Budget: Unloading momentum

```
% Event 4: Unloading momentum
%-----------------------------
k = k + 1;
event(k).type = 'delta-h';
event(k).description = 'Delta␣H';
event(k).aCSThruster = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster = [];
event(k).dVTotal = [];
event(k).dHTotal = [1;0;0];
event(k).dTControl = 1;
event(k).disturbance = [];
event(k).nPulses = [];
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];
```

This listing shows firing of the ACS thrusters in response to a disturbance.

Listing 20.9: Fuel Budget: Using ACS thrusters

```
% Event 5: Using thrusters for ACS
%---------------------------------
k = k + 1;
event(k).type = 'acs';
event(k).description = 'ACS␣Disturbance';
event(k).aCSThruster = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster = [];
event(k).dVTotal = [];
event(k).dHTotal = [];
event(k).dTControl = 1;
event(k).disturbance = [0;1;0];
event(k).nPulses = [];
event(k).duration = 10000;
event(k).regulated = 0;
event(k).cM = [0;0;0];
```

The next listing shows an example of firing a few pulses.

Listing 20.10: Fuel Budget: Firing a few pulses

```
% Event 6: Firing a few pulses
%-----------------------------
k = k + 1;
event(k).type = 'pulsed';
event(k).description = 'Short␣Pulse';
event(k).aCSThruster = [1 2];
event(k).dVThruster = [];
event(k).dVTotal = [];
event(k).dHTotal = [];
event(k).disturbance = [];
event(k).nPulses = []; % Won't use any fuel this way
event(k).pulsewidth = 0.05;
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];
```

The results for these events are shown in Listing 20.11.

```
FBudget( event, x ) % Generates the following results
```

Listing 20.11: Fuel Budget Example, Six Events

```
TestBudget Propellant budget 19-Sep-1997
Item Description Total Fuel Remaining Fuel Used Pressure ACS Isp DV Isp
1 Initial State 1190.92 117.89 117.89 0.000 0.000 2.4e+06 2.4e+06 0.00 0.00
2 Delta V Continuous 1153.87 99.36 99.36 18.525 8.525 2.4e+06 2.4e+06 100.02 360.00
3 Delta V Off Pulse 1099.18 72.02 72.02 27.346 27.346 1.7e+06 1.7e+06 100.00 210.00
4 Delta H 1099.18 72.02 72.02 0.001 0.001 1.2e+06 1.2e+06 100.41 0.00
5 ACS Disturbance 1087.45 66.16 66.16 5.862 5.862 1.2e+06 1.2e+06 100.39 0.00
6 Short Pulse 1087.45 66.16 66.16 0.000 0.000 1.2e+06 1.2e+06 100.50 0.00
```

For east/west stationkeeping the delta V required because of longitude drift, due to the earths tesseral gravitational harmonics is computed by

```
[dVEW,dTEW] = DVLDrift( box, scLon );
```

where the first number is the size of the stationkeeping box in degrees and the second is the spacecraft longitude. `DVLDrift` outputs the delta V required per maneuver and the time between maneuvers. For north/south stationkeeping the corresponding function is

```
[dVNS,dTNS] = DVIDrift( box, year );
```

where the second argument is the year for which the calculation is done. The inclination drift is a function of the earth/moon geometry which is included in `DVIDrift`.

CHAPTER 21

ORBIT REFERENCES

This chapter gives selected orbit resources.

21.1 Orbits

Excellent applied navigation and interesting mathematical approaches to orbit dynamics. Prof. Battin gives extensive information on the Apollo navigation system.

> Battin, R. H. "An Introduction to the Mathematics and Methods of Astrodynamics." AIAA Education Series.

The most comprehensive and up-to-date book on applied orbit dynamics.

> Vallado, David. A. and W. D. McClain. *Fundamentals of Astrodynamics and Applications, Second Edition*. Microcosm Press, El Segundo, CA, 2001.

A good introduction with many physical insights.

> Wiesel, W. E. *Spaceflight Dynamics*. McGraw-Hill, 1989.

21.2 Web Sites

Gravity models have been available at

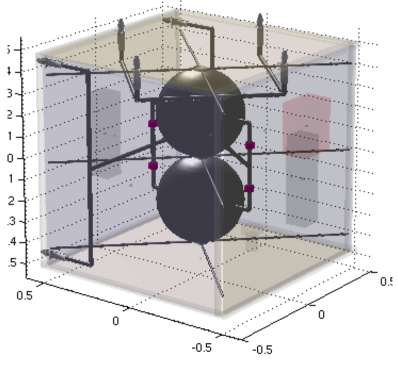
> http://cddis.nasa.gov/926/egm96/

> http://www.iges.polimi.it/pagine/services/repo/repo_model.asp

JPL solar systems dynamics information, including the Horizons ephemeris tool and astrodynamics constants, is available at http://ssd.jpl.nasa.gov/

Part IV

Subsystems Analysis



CHAPTER 22

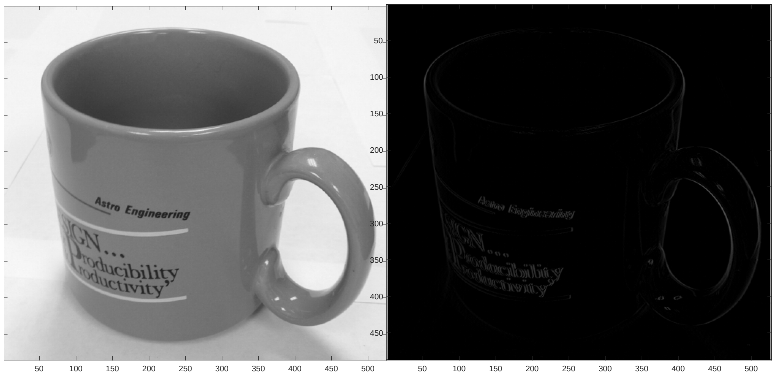
IMAGING

This chapter gives some examples of functions in the Imaging module.

22.1 ImageMatching

ImageMatching has the Canny Edge Detection algorithm, image derivative and a function for fitting a circle to points in an image. Figure 22.1shows how `CannyEnhancer` enhances edges.

Figure 22.1: CannyEnhancer demo



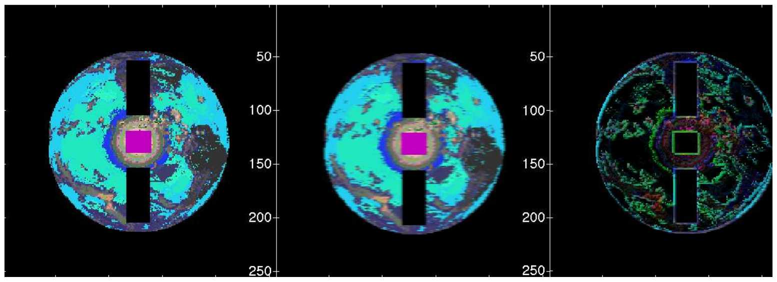
22.2 ImageProcessing

This folder includes utilities for image processing

1. `GaussianImageFilter`
2. `IndexedImageToRGB`
3. `MedianImageFilter`

For example `GaussianImageFilter`.

Figure 22.2: GaussianImageFilter demo



The first image shows the original, the second the filtered and the third the difference. This function handles images of the form `a(n,m,3)`. The matrices must be doubles.

CHAPTER 23

LINK MODULE

The link module provides basic functions for computing link margins. Link margins are the signal to noise ratios for communication links between spacecraft and between spacecraft and the ground. The module is divided into utilities, design functions, and RF and optical link analysis functions. The list of folders is shown below.

```
>> help Link
  PSS Toolbox Folder Link
  Version 8.0      03-Dec-2009

  Directories:
  Coverage
  Demos
  Demos/Optical
  Demos/RF
  Help
  Ladar
  LinkDesign
  LinkUtilities
  Optical
  RF
  Radar
```

23.1 Link Utilities

Link utilities are for conversions and other basic link calculations. For the complete listing type

```
>> help LinkUtilities
```

and you will get the list of functions with a help comment.

```
>> help LinkUtilities

  Link/LinkUtilities

  B
     BEP                             - Computes the bit error probability given C/N
         and the bit rate.

  C
     CNRComp                         - Compute the carrier to noise ratio.
```

```
    ConstellationGroundContact   - Compute the number of contacts for a
        constellation with ground users.
    ConstellationMaxDistance     - Compute maximum distances between satellites
        in a constellation.

  D
    DBPower                      - Power conversion to decibels. dB = 20 log10(s
        ).
    DBSignal                     - Signal conversion to decibels. dB = 10 log10(
        s).
    DBSignalToPower              - Convert decibels to power.
    DBWToNW                      - Convert dBW (decibels-Watts) to nanowatts.
    DBWToPW                      - Convert dBW to picowatts.
    DataRate                     - Computes communications sytems data rates
    DopplerBlind                 - Finds Doppler blind speeds.
    DopplerShift                 - Doppler shift

  F
    FPD                          - Function called by fzero to compute the
        carrier to noise ratio.
    FPFA                         - Right hand side for probability of false
        alarm computation.
    Frequency                    - Outputs the frequencies for specified bands
        of electromagnetic radiation.
    FrequencyToWavelength        - Converts frequency to wavelength.

  M
    ModulationSpectralEfficiency - Spectral efficiency of different modulation
        schemes
    MultiBEPPlot                 - Computes and plots multiple bit error rates
        on the same plot.

  P
    PeriodToSemimajorAxis        - Computes semi-major axis from orbital period
    ProbDetection                - Probability of detection for a diffuse target
        .

  S
    Shannon                      - The Shannon information theorem relating S/N
        and C/B.

  T
    TToPower                     - Convert temperature to power.

  W
    Wavelength                   - Outputs the wavelengths for bands of
        electromagnetic radiation.
    WavelengthToFrequency        - Converts wavelength to frequency.

  Y
    YBComp                       - yB computation used as part of the
        probability of detection algorithm.

   Folders named LinkUtilities
```

For example to get nanowatts from dB,

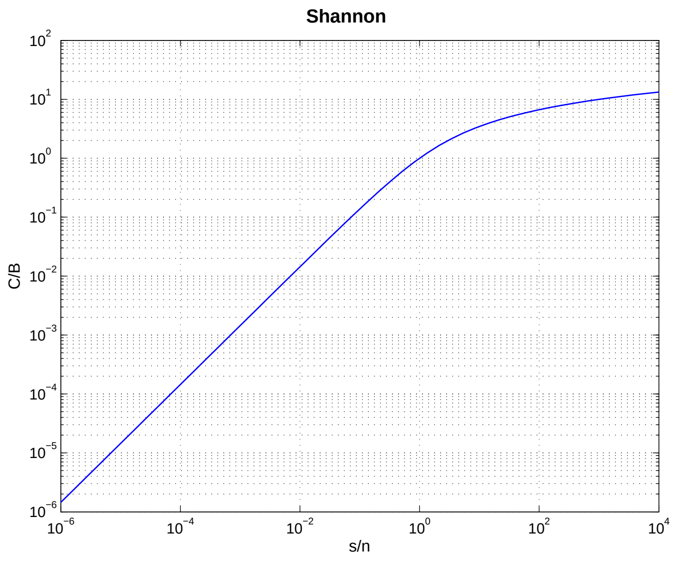
```
>> power = DBWToNW(-106)
power =
   2.5119e-02
```

or picowatts

```
>> power = DBWToPW(-106)
power =
   2.5119e+01
```

If you type `Shannon` you get Figure 23.1 (a demo of the `Shannon` function). This relates the information content to the signal to noise ratio.

Figure 23.1: Shannon demo



23.2 Optical

Two functions are available for optical calculation. The first, `OpticalBackgroundNoise`, computes the optical background noise for

- Solar
- Lunar
- Mercury
- Venus
- Mars
- Jupiter
- Saturn
- Sunlit clouds
- Sunlit snow and ice
- Starfield

The second, `OpticalReceiverNoiseDensity`, computes optical receiver noise density for coherent and direct receivers.

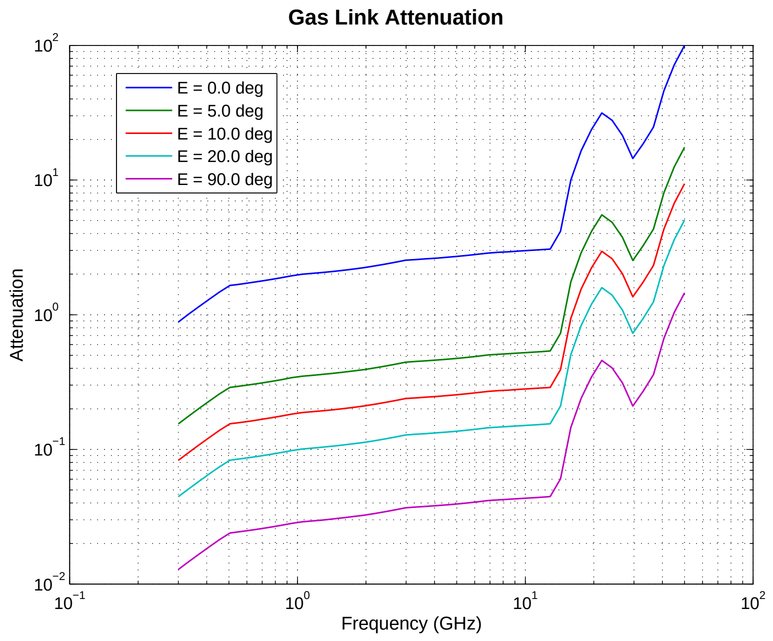
23.3 RF

The RF functions compute various losses and noise temperatures. For example typing

```
LossAtmosphericGas
```

gives you path losses due to atmospheric gas, see Figure 23.2. Other loss functions include `LossFreeSpace` and `LossPrecipitation`. Typing

Figure 23.2: Atmospheric Gas Loss



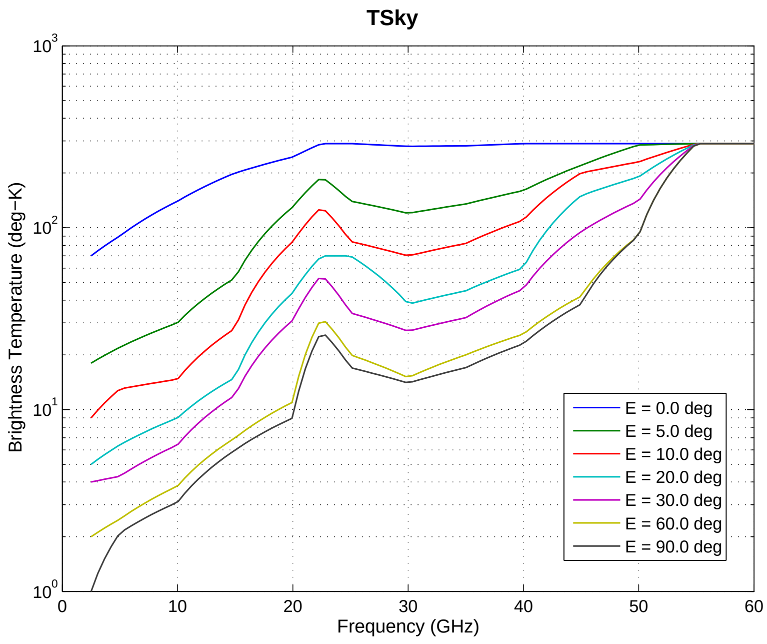
```
TSky
```

gives you a sky temperature demo, see Figure 23.3 on the facing page. You can also get temperatures for the moon, sun, receiver, attenuator, etc.

You can compute losses and noise inputs using a variety of functions in the toolbox. All of these tools are integrated into the `LinkAnalysisTool`. The built-in demo of this function is shown below and the results in Figure 23.4 on page 190.

```
% Space link
%-----------
spaceLink.theta3DB             = 2; % (deg)
spaceLink.fIllumination        = 70; % (deg)
spaceLink.effTransmit          = 0.55;
spaceLink.powerTransmit        = 10; % dbW
spaceLink.lossEdge             = 3; % dB
spaceLink.lossTransmitterFeed  = 0.5;
spaceLink.frequency            = 12;
spaceLink.diameterAperture     = 1;
spaceLink.thetaDepointing      = 0.1; % (deg)

% Ground link
%------------
groundLink.antenna.type                 = 'circularAperture';
groundLink.antenna.area                 = pi*2^2/4;
```

Figure 23.3: Sky temperature



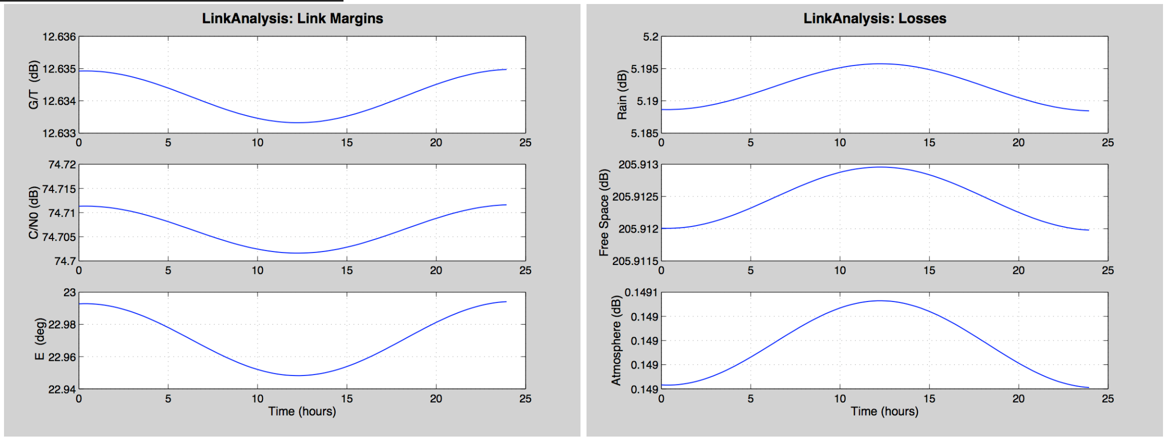
```
  groundLink.latitude                      = 45;
  groundLink.longitude                     = 0;
  groundLink.altitude                      = 0;
  groundLink.horizonAngle                  = 0;
  groundLink.tGround                       = 45; % Ambient ground temperature
  groundLink.tAttenuatorAmbient            = 290; % Ambient attenuator temperature
  groundLink.lossAttenuator                = 1; % Attenuator loss (dB)
  groundLink.lossFeeder                    = 0.5;
  groundLink.climateZone                   = 'e';
  groundLink.precipitationFractionOfTime = 0.01;
  groundLink.polarizationTiltAngle         = 0;

  % Orbit
  %-------
  orbit.el = [42164 0 0 0 0 0];
  orbit.jD = DateStringToJD( '11/04/2004' );
  orbit.t  = linspace(0,Period(orbit.el(1)));

  LinkAnalysisTool( spaceLink, groundLink, orbit );
```

23.4 Downlink Analysis Example

This example shows how to do a downlink analysis. The reference is Maral, G. and M. Bousquet. *Satellite Communications Systems*, Third Edition. John Wiley (1998). See this and more examples in `LinkExamples.m`.

```
%------------------------------------------------------------------------------
% Example 2 pp. 24-25
%------------------------------------------------------------------------------
fD              = 12; % GHz
pT              = DBSignal(10); % dBW
r               = 40000; % km
```

Figure 23.4: Results of `LinkAnalysisTool`



```
theta3DB      = 2;
effT          = 0.55;
effR          = 0.6;
dR            = 4;
fIllumination = 70;

gT            = Gain3dB( theta3DB, fIllumination, effT );
eIRPSL        = pT + gT;
gR            = AntennaGain(struct('type','circular␣aperture','area',pi*dR^2/4,'
   eff',effR),fD);
lFS           = LossFreeSpace( fD, r );
pR            = eIRPSL - lFS + gR;

fprintf('\n\n2.3.2␣Example␣2:␣The␣Downlink\n-----------------------------')
fprintf('Gain␣Transmit␣Antenna␣=␣%10.2f␣(dBW)\n',gT);
fprintf('EIRP␣SL␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣%10.2f␣(dBW)\n',eIRPSL);
fprintf('Free␣Space␣Loss␣␣␣␣␣␣␣=␣%10.2f␣(dB\n)',lFS);
fprintf('Gain␣Receive␣Antenna␣␣=␣%10.2f␣(dB)\n',gR);
fprintf('Power␣Received␣␣␣␣␣␣␣␣=␣%10.2f␣(dBW)\n',pR);
fprintf('Power␣Received␣␣␣␣␣␣␣␣=␣%10.2f␣(pW)\n',DBWToPW(pR));
```

The only loss is the free space loss which is due to the $1/r^2$ dissipation of the radio waves with distance in free space. The result of this example is

```
2.3.2 Example 2: The Downlink
-----------------------------
Gain Transmit Antenna = 38.23 (dBW)
EIRP SL = 48.23 (dBW)
Free Space Loss = 206.07 (dB)
Gain Receive Antenna = 51.81 (dB)
Power Received = -106.03 (dBW)
Power Received = 24.94 (pW)
```

With a distance of 40,000 km the free space loss dominates the losses. Only 24.94 picowatts is delivered to the receiver. This analysis ignores other losses such as receiver loss, loss due to atmosphere and loss due to rain. In addition it is not a complete example since we really need to know the signal to noise ratio at the receiver to determine how much information we can transmit.

23.5 LADAR

Two main functions are available for analyzing LADAR systems. `OpticalLinkBudget` is used to compute link budgets for optical systems. The following example shows how to assign values to the data structure. Note that losses are from 0 to 1 not dB. 1 is lossless and 0 is infinite losses.

```
disp('Pluto␣Mission');
d.wavelength                = 532e-9; % nd:YLF (Neodymium doped: Ytterbium Lithium
    Floride)
d.transmitApertureDiameter = 0.2; % M
d.thetaOffset               = 0; % No offset
d.transmitPower             = 100; % W
d.wavefrontError            = 0; % Loss is exp(-2*pi*error/wavelength)
d.range                     = 41*149.6e9; % 41 AU
d.receiverLoss              = 1; % 0 to 1, 1 means no loss
d.receiverPointingLoss      = 1; % 0 to 1, 1 means no loss
d.receiveApertureDiameter  = 1; % Telescope diameter (m)
OpticalLinkBudget( d );
```

This example results in for a total link loss of -292.7 dB. Compare this with the results given in the previous section!

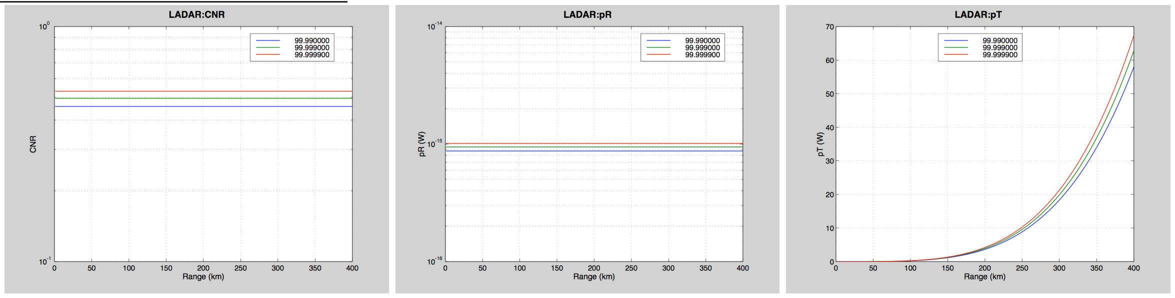
```
Pluto Mission
Transmit Power          =     100.0000 W
Wavelength              =     532.0000 nm
Theta Divergence        =   3.3868e-06 rad
Beamwidth at Range      =   2.0773e+07 m
Pointing Loss           =   1.0000e+00
Transmit Antenna Gain   =   1.3949e+12
Range Loss              =   4.7640e-41
Wavefront Loss          =   1.0000e+00
Receive Antenna Gain    =   3.4872e+13
Receiver Pointing Loss =   1.0000e+00
Receiver Loss           =   1.0000e+00
Receive Power           =   2.3173e-13 W
Losses are 0 to 1. 1 means no loss
```

The transmit power can be computed using `LADARTransmitPower`.

```
d.b      = 30e3;
d.n      = 500;
d.etaM   = 0.5;
d.d      = 0.2;
d.etaR   = 0.7;
d.theta  = 90e-6;
d.k      = 2;
d.etaT   = 0.7;
d.etaQ   = 0.8;
d.etaE   = 0.7;
d.pFA    = 1e-6;
d.pD     = 1 - [1e-4 1e-5 1e-6];
d.xC     = 1;
d.lambda = 11.149e-6;
d.r      = linspace(1,400)*1000;
d.tA     = 1;
d.thetaT = 0;
d.x      = d.r/1000;
d.xLabel = 'Range␣(km)';
LADARTransmitPower( d );
```

This function generates 3 figures shown in Figure 23.5 on the following page.

Figure 23.5: Results of `LADARTransmitPower`



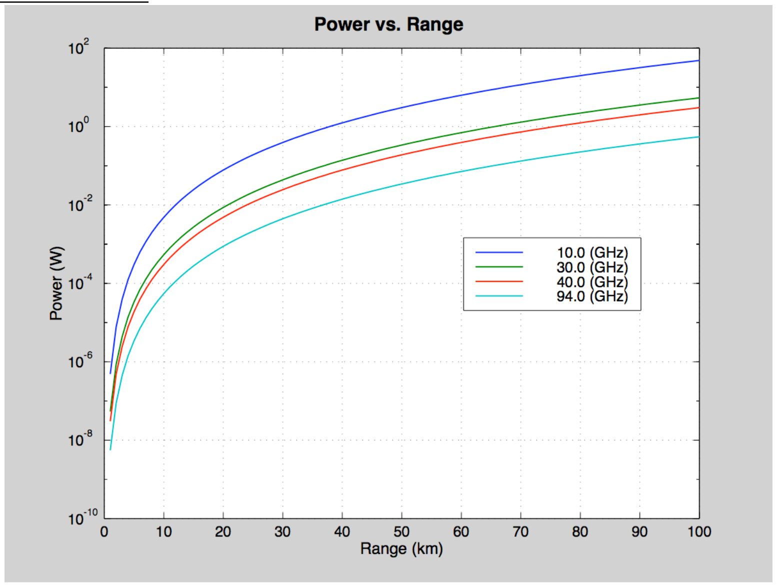
23.6 RADAR

There are many tools available for radar analysis. The radar equation may be computed in various forms by:

```
RadarEquationPulse.m
RadarEquationSN.m
RadarEquation.m
```

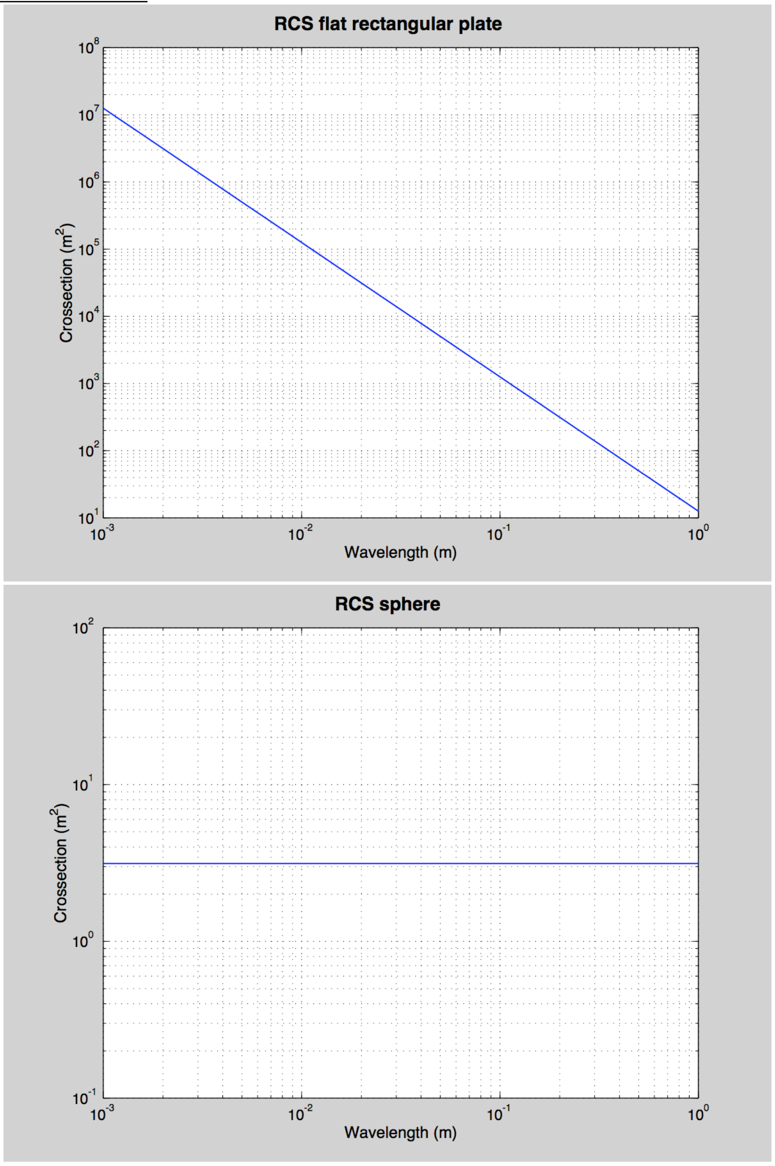
Most have demos. For example if you type `RadarEquation` you will see Figure 23.6.

Figure 23.6: Results of `RadarEquation`



Type `RadarCrosssection` and you will get the radar crosssections of a plate and sphere.

Figure 23.7: Results of `RadarCrosssection`



CHAPTER 24

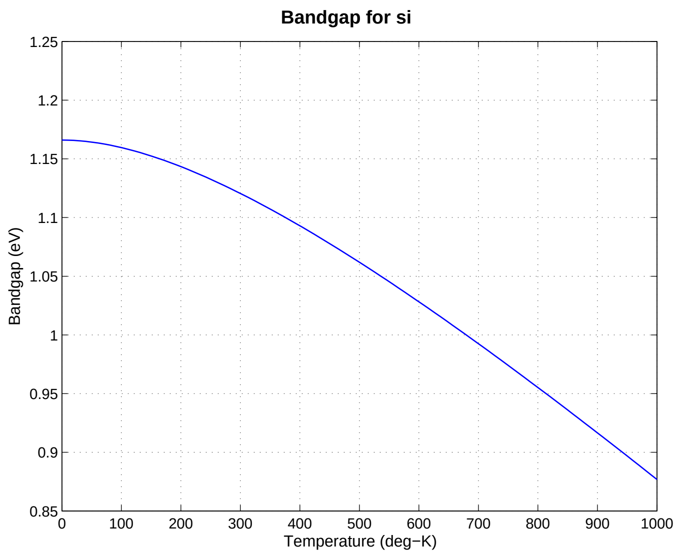
POWER

This chapter discusses how to generate use the power module.

The Source folder provides functions for computing parameters for solar cells.

`BandGap` computes the band gap for solar cells as a function of temperature, as shown in Figure 24.1.

Figure 24.1: Band gap



`SolarCell` computes the I-V curves for solar cells as in Figure 24.2 on the next page.

`SolarCellEff` computes the efficiency for solar cells as in Figure 24.3 on the following page.

Figure 24.2: Solar cell I-V curves

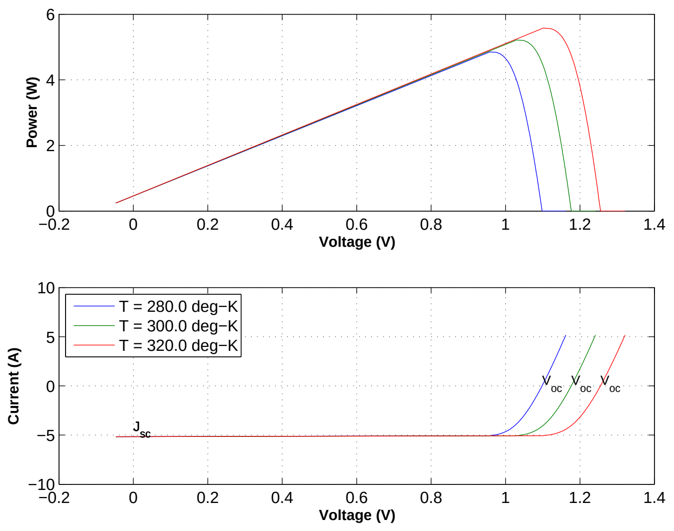
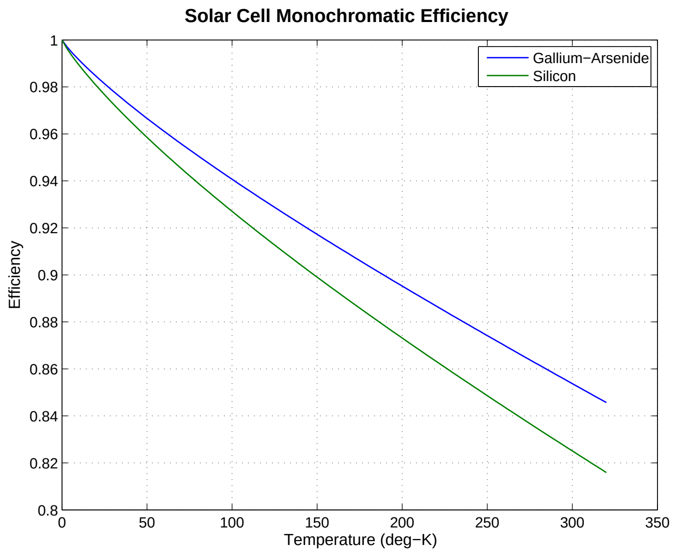


Figure 24.3: Solar cell efficiency



CHAPTER 25

PROPULSION MODULE

This chapter presents the propulsion module.

25.1 Module Overview

The propulsion module has several broad categories

- Chemical propulsion
- Electric propulsion
- Multistage rocket analysis
- Rocket

The chemical category includes functions for modeling and designing blowdown systems and functions for analyzing the chemistry or chemical rockets. The electric propulsion category includes functions for analyzing and optimizing electric propulsion systems. The multistage rocket analysis functions are for launch vehicle design and the rocket functions are functions the apply to all types of rockets.

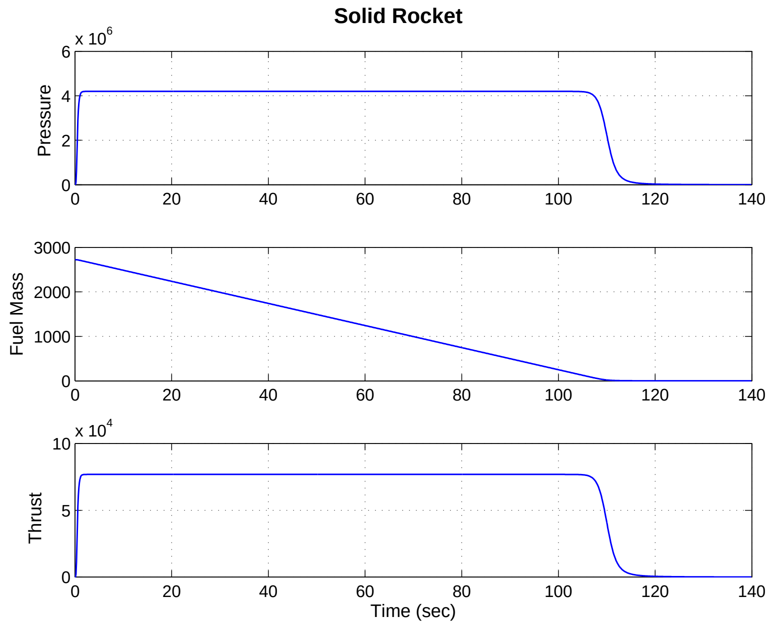
The complete list of folders is shown below.

```
>> help Propulsion
  PSS Toolbox Folder Propulsion
  Version 2019.1      23-Dec-2019

  Directories:
  Chemical
  Demos
  Demos/Chemical
  Demos/Electric
  Demos/Utilities
  Electric
  Gas
  MultiStage
  Nuclear
  Rocket
  Utilities
```

25.2 Solid Rocket Motor

`TSolid` is an example of a solid motor simulation. The function `SolidR`, which models the dynamics of the burning motor, is numerically integrated in a loop. See Figure 25.1 for the demo results.

Figure 25.1: Solid motor firing



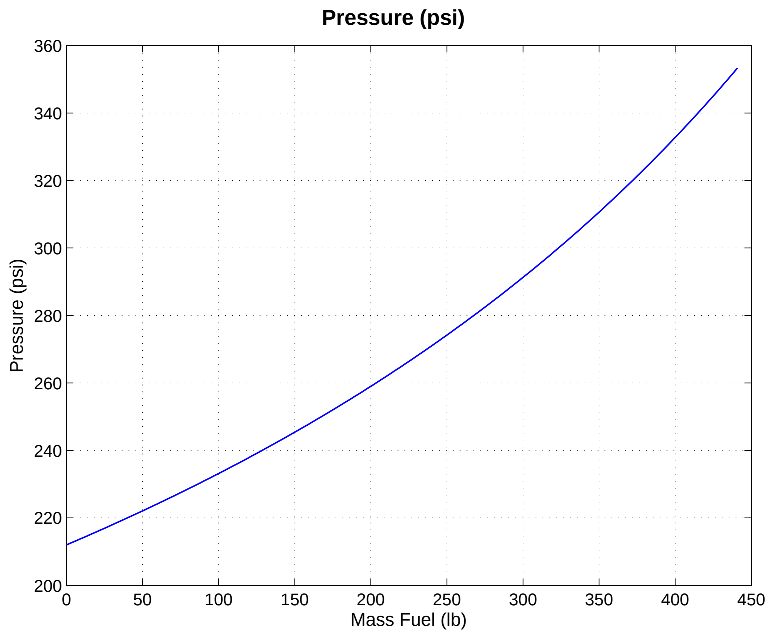
25.3 Blowdown Curve

Designing a blowdown system is of critical importance for most satellites. You need to specify the mass of fuel, initial pressure and final pressure. The pressure are set by the thrusters. The following code designs a blowdown system.

```
Nm2ToPSI = Constant('N/m^2 to PSI');
KgToLbF = Constant('Kg to Lb Force');
molWt = 0.004;
rPress = MolWt2R(molWt);
mFuel = 0:200;
rhoFuel = 1000;
T = 293;
vTank = 0.5;
mPress = 1.2;
p = BloDown(mPress,rhoFuel,vTank,rPress,T,mFuel);
Plot2D(mFuel*KgToLbF,p*Nm2ToPSI,'Mass Fuel (lb)','Pressure (psi)')
```

See Figure 25.2 on the facing page. With no fuel, the system pressure is 210 psi. With the full fuel load the pressure is 350 psi.

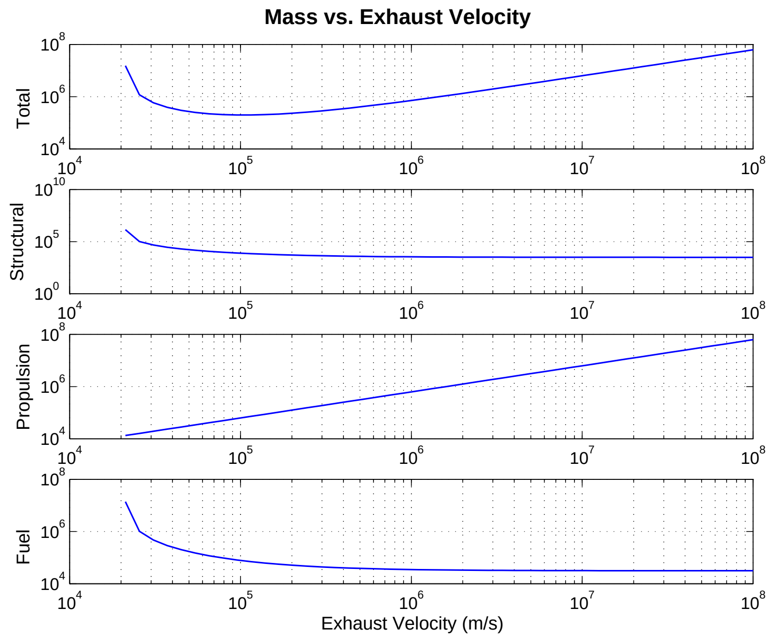
Figure 25.2: Blowdown curve



25.4 Electric propulsion

Type `ElectricPropulsion` and you will get the plot in Figure 25.3. This shows the mass of an electrically propelled vehicle as a function of exhaust velocity. The optimal exhaust velocity is surprisingly low.

Figure 25.3: Optimal electric vehicle



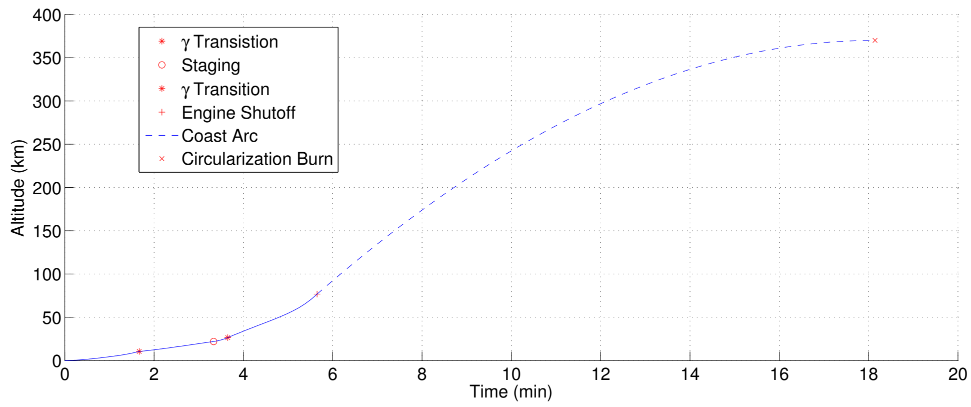
The vehicle characteristics are printed in the MATLAB command window:

```
       mP: 50000
   thrust: 15000
       dV: 50000
       pR: 20000
       sR: 1.0000e-01
      eff: 6.0000e-01
       uE: 1.0381e+05
    power: 1.2976e+09
       mR: 6.4880e+04
       mF: 7.5771e+04
       mS: 7.5771e+03
       mD: 1.2246e+05
       mT: 1.9823e+05
     mDot: 1.4450e-01
```

25.5 Launch vehicle analysis

Type `TSTODemo` for a simulation of a two-stage-to-orbit horizontal takeoff/horizontal landing launch vehicle. The first stage is an air-breathing vehicle powered by two turbofan and two hydrogen-fueled ramjet engines. The transition from turbofan to ramjet power occurs at Mach 1.5. The second stage is powered by a hydrogen rocket. Type `help TSTODemo` for a description of the mission. The flight path of both stages and also the staging conditions have been optimized to maximize the mass to orbit. The optimization has been previously performed with `OptimizeTSTO`, which requires MATLAB's Optimization Toolbox. Figure 25.4 shows the results of the demo, in which a payload of 593 kg is delivered to orbit.

Figure 25.4: Two-State-to-Orbit Simulation



CHAPTER 26

THERMAL MODULE

This chapter introduces the thermal analysis module.

26.1 Module Overview

The thermal module provides basic functions for performing thermal analysis of spacecraft. It is not meant to be a substitute for packages such as TRASYS or SINDA. Rather, it is useful to help GN&C analysts uncover potential thermal problems caused by GN&C activity.

There are two classes of functions. One class is the properties functions. The other is the thermal analysis functions. The complete list of folders is shown below.

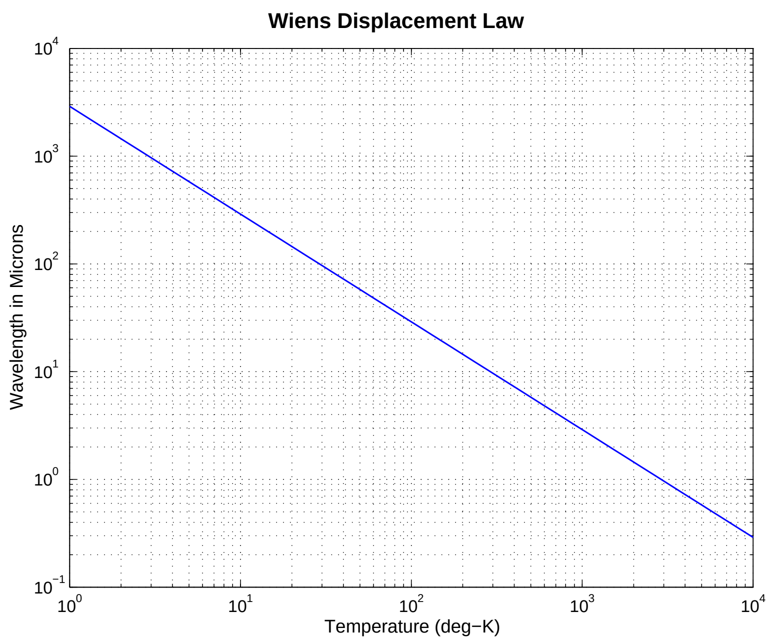
```
>> help Thermal

  CubeSat/Demos/Thermal

  C
     CubeSatFaceTemperaturesDemo - Demonstrate temperatures of faces of a CubeSat

  I
     IsothermalCubeSatDemo       - Isothermal satellite demo

  CubeSat/Thermal

  A
     AddThermalConductivity - Add a thermally conductive path between cubesat
         faces

  I
     IsothermalCubeSatSim   - An isothermal CubeSat simulation using Euler
         integration.

  R
     RHSIsothermalCubeSat   - An isothermal CubeSat amodel right-hand-side.

  PSS Toolbox Folder Thermal
  Version 2019.1      23-Dec-2019

  Directories:
```

```
Brayton
Demos
Demos/ThermalAnalysis
Demos/ThermalControl
Demos/ThermalGraphics
HeatExchanger
ThermalAnalysis
ThermalData
ThermalGraphics
ThermalProperties

  Folders named Thermal  ThermalProperties
```

26.2 Thermal Properties

These functions generate information about the thermal properties of objects. For example typing `WiensDsp` computes the peak black body wavelength as a function of temperature, as shown in Figure 26.1.

Figure 26.1: Black body wavelength as a function of temperatures



`AbsDa` computes the absorbed heat as a function of absorptivity and incoming flux. See Figure 26.2 on the facing page. Other functions compute the temperate of simple geometric objects under incoming radiation.

Another function is `PlanckL` which gives Planck's Law. See Figure 26.3 on the next page.

26.3 Thermal Analysis

There are two thermal analysis functions. One is `Thermnet` which generates a state space system for a general thermal system including radiation heat transfer. This is done by analytically linearizing the equations about a nominal temperature. You can specify any connections between nodes and declare external nodes for heat flux into the system.

Figure 26.2: AbsDa demo

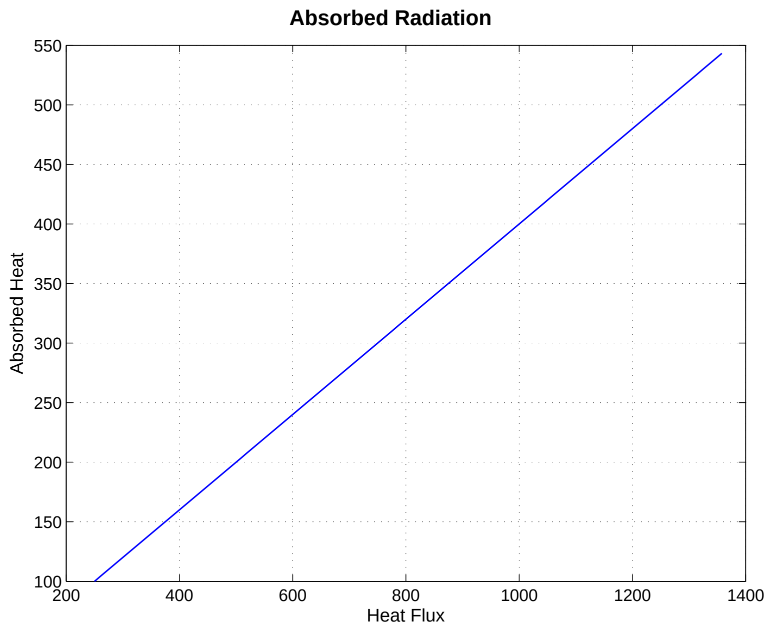
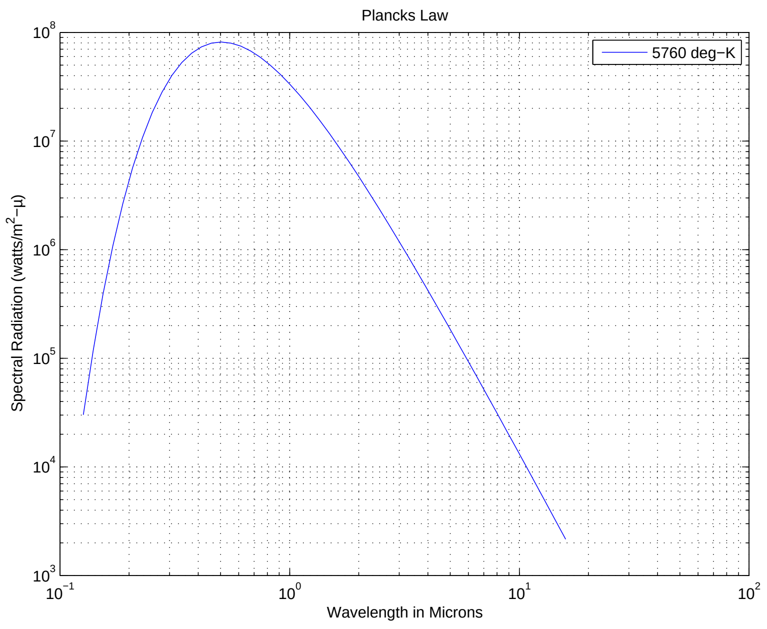


Figure 26.3: Plancks Law



The second function is `ThermalEquilibrium`. This function computes the thermal equilibrium for a system governed by the static equation

$$lT^4 + kT = bq \tag{26.1}$$

using MATLAB's sparse matrix functions.

26.3.1 `Thermnet` Example

`Thermnet` creates a set of first order differential equations to model a thermal circuit. Inputs are the nodal masses, a conductive node map, a radiative node map, an input node map and an output node map. The output is a set of first order differential equations in the form

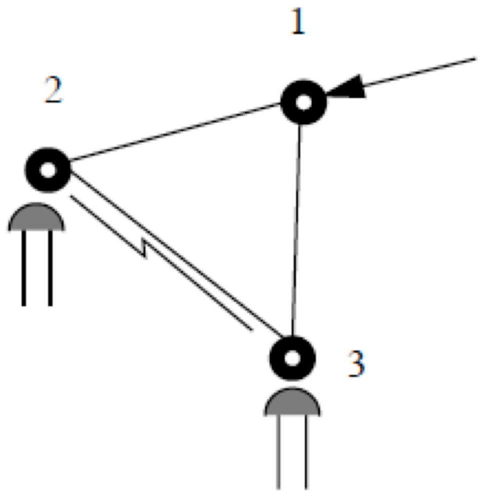
$$\begin{aligned} \dot{T} &= aT + bq \\ y &= cT \end{aligned} \tag{26.2}$$

where T is the nodal temperature, q is the thermal input and y is the vector of measurements.

A node map has one row for each node and as many columns as are necessary to specify the connections. Each node map is accompanied by a coefficient map that gives either the conductive or radiative heat transfer coefficients for the node pair. `Thermnet` will automatically make the a matrix symmetric for conduction. `Thermnet` will warn you if you entered a coefficient twice and the two were different. You only have to specify the connection in one direction.

Assume that we want to model the system shown in Figure 26.4.

Figure 26.4: Thermal network



External flux is coming in on node 1 and measurements are taken on node 3 and 2. There is conduction heat transfer between all nodes, but radiation transfer only between nodes 3 and 2, as indicated by the zigzag line.

Assume that the thermal mass of each node is one and that the nominal temperature is 300 deg-K. Then

```
massNode = [1 1 1];
tNominal = [300 300 300];
```

The input node is 1 so

```
inputNodeMap = 1;
```

and the output nodes are 2 and 3

```
outputNodeMap = [2 3];
```

There is one row per node in the conduction node map. All nodes are connected to all other nodes so the conduction node map is

```
conductionNodeMap = [2 3;1 3;1 2];
```

and the radiation node map is

```
radiationNodeMap = [inf;3;2];
```

The first node radiates to infinity. If there are different numbers of connections per node, then fill in zeros to make the matrix full. The coefficients are

```
conductionCoeff = [1 2;1 2;2 2];
radiationCoeff = [1;1;1]*0.25e-6
```

Again, each row is for a node and the column corresponds to the index in the node map. For example, the conduction coefficient between node 1 and 2 is 1. Type

```
[a, b, c] = Thermnet( massNode,...
                      conductionNodeMap, conductionCoeff,...
                      inputNodeMap, outputNodeMap,...
                      radiationNodeMap, radiationCoeff,...
                      tNominal )
```

and the result is

```
a =
   -30     1     2
     1   -30    29
     2    29   -31
b =
     1
     0
     0
c =
     0     1     0
     0     0     1
```

26.3.2 Thermal Equilibrium Example

The simplest case is two nodes connected by a conductor. Heat is coming through node 1. Both nodes are radiating to free space. `k` is the linear term coefficient and `l` is the quartic coefficient. `b` is the input matrix. The input flux `q` is set to the typical value of the solar flux at the Earth. The input node temperatures `t` are guesses or prior values. The default number of iterations is 40 and the default tolerance is MATLAB's eps parameter (2.2204e-16).

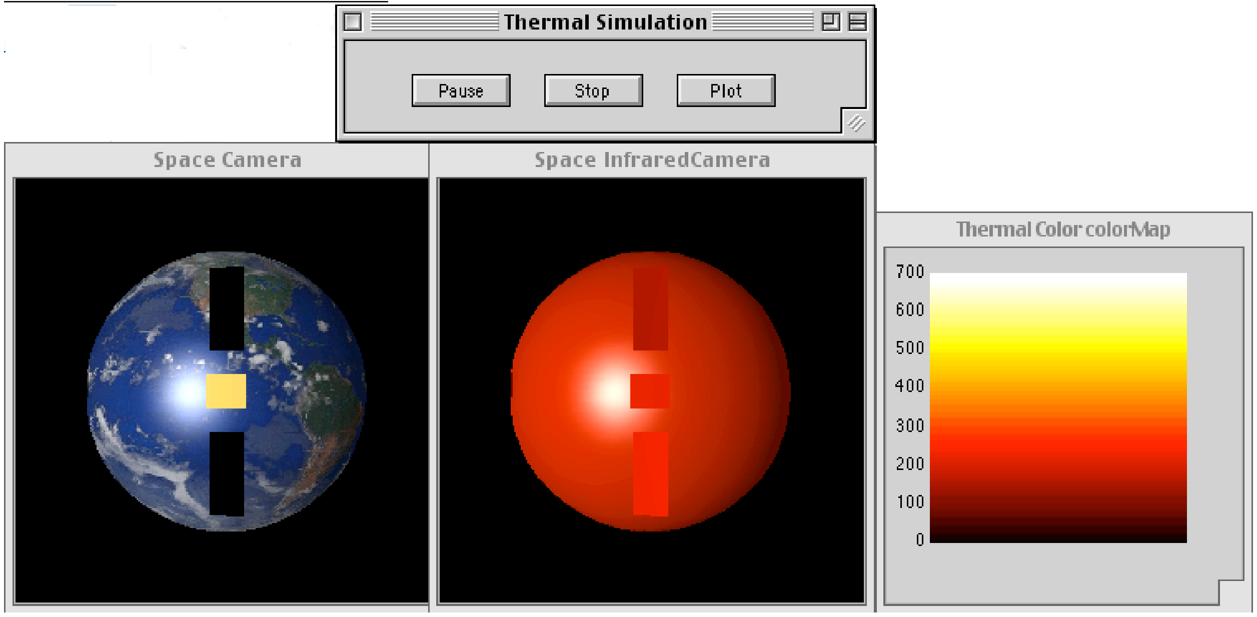
```
d.k = 10*[1 -1;-1 1];
d.l = 5.67e-8*[1 0;0 1];
q   = 1358;
t   = [300;300];
d.b = [1;0];

[t, delta]   = ThermalEquilibrium( d, q, t )

t =
   3.5236e+02
   3.0396e+02
delta =
   8.5508e-16
```

As expected, node 1 is hotter than node 2, by about 50 degrees. This is a good model for a solar array.

26.3.3 Thermal Imaging

`SCThermalDemo` demonstrates thermal imaging code. The script draws a view of the spacecraft as seen in "natural" lighting and then draws one which shows an image as function of temperature. Figure 26.5 shows the four windows that are generated.

Figure 26.5: Thermal imaging demo



This simulation employs the model generated in `BuildSCModelForImaging` which is a simple spacecraft model.

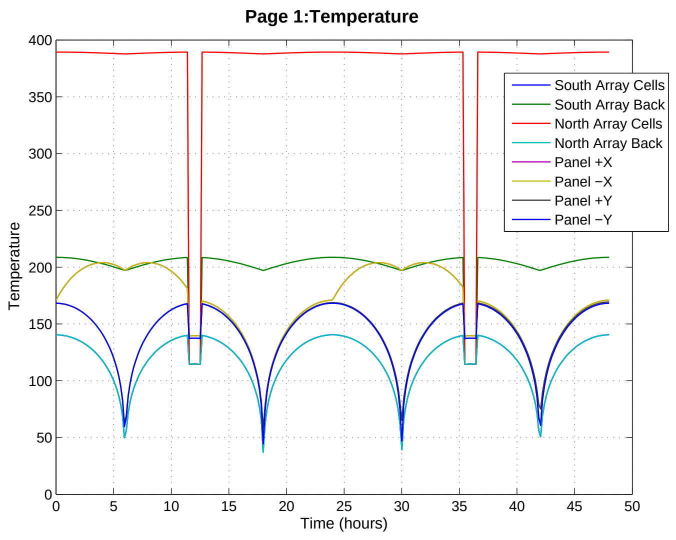
```
g = load('SCForImaging');
```

Two different cameras are used. `SpaceCamera` shows the spacecraft in solar lighting. In figure ambient lighting is set high so that it is clearly visible. `ThermalImager` generates the thermal color data. This script employs the standard PSS CAD model format with the fields for position and quaternion set in the script.

```
% CAD body structure
%--------------------
g.body(1).bHinge.q      = QPose(qLVLH);
g.body(2).bHinge.angle = theta + pi;
g.body(3).bHinge.angle = theta + pi;
g.rECI                  = rECI(:,k);
g.qLVLH                 = qLVLH;
```

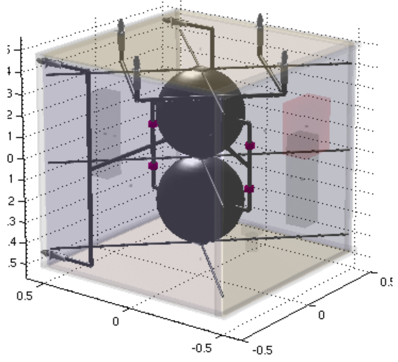
While the script is running you can end it and plot all temperatures by pushing the plot button.

Figure 26.6: Thermal imaging demo temperature plot



PART V

ATTITUDE AND ORBIT ESTIMATION



CHAPTER 27

INTRODUCTION

27.1 Overview

The Estimation module contains software for developing estimation algorithms. The algorithms can be used to solve any estimation problem. Particular technologies included are

1. Fixed Gain Kalman Filters
2. Variable Gain Kalman Filters
3. Extended Kalman Filters
4. Unscented Kalman Filters
5. Fault Detection

Topic areas discussed in this part of the guide are

1. Recursive attitude determination
2. Recursive orbit determination
3. Stellar attitude determination

This guide covers the basics of operating the software. Estimation theory is covered in the accompanying book, *Spacecraft Attitude and Orbit Control*.

CHAPTER 28

GENERAL ESTIMATION

28.1 Introduction

This module focuses on recursive estimation. Recursive estimation is simply taking a measurement and using it to update an existing estimate of the state of the system. The measurement need not be a state, but for all states to be observed the measurement must be related to the states or derivatives of the states.

In this module, recursive estimators can be divided into six classes:

1. Fixed gain estimators
2. Variable gain estimators for linear systems
3. Variable gain estimators for nonlinear systems in which the plant (the state dynamics) are linearized about the current state and propagated as a linear system. This type of system is used for attitude estimation. This is also called the Extended Kalman Filter.
4. Variable gain estimators for nonlinear systems in which the plant is numerically integrated. This type of system is used for orbit estimation. This is also called the Continuous Discrete Extended Kalman Filter.
5. Unscented Kalman Filters. These filters compute multiple states and covariances for each in effect computing the statistics online.

This chapter will discuss estimation in general with examples from attitude and orbit estimation.

28.2 Fixed Gain Estimators

Fixed gain estimators are written in the form

$$x_{k+1} = ax_k + bu_k + k(y - cx_k) \tag{28.1}$$

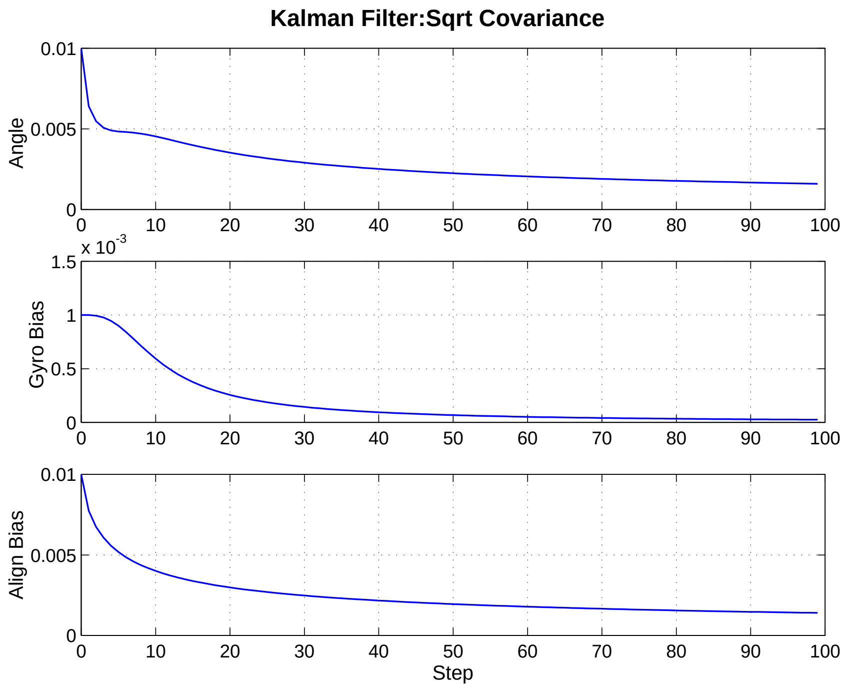
where x_k is the state vector at step k, a is the state transition matrix, b is the input matrix, u_k is the input vector, y is the measurement vector and c is the measurement matrix that relates the states to the measurements.

The gain matrix, k may be computed in many different ways. Note that this, in contrast to a conventional noise filter, requires an estimate of the inputs. However, the filter returns the vector of all observable states which is needed for linear quadratic controllers.

This kind of estimator can be designed using the toolbox function `QCE`, for continuous systems, and `DQCE` for discrete systems. If the former is used, the filter must be converted to discrete time in a second step. If the latter is used you must be careful that the filter state matrix is balanced and well-conditioned.

An interesting example is given in `KalmanFilterDemo` in which two angle measurements are available for single degree of freedom case. One of the states is the unknown misalignment between the two sensors. Figure 28.1 shows the covariance for this problem.

Figure 28.1: Fixed gain Kalman filter



28.3 Variable Gain Estimators

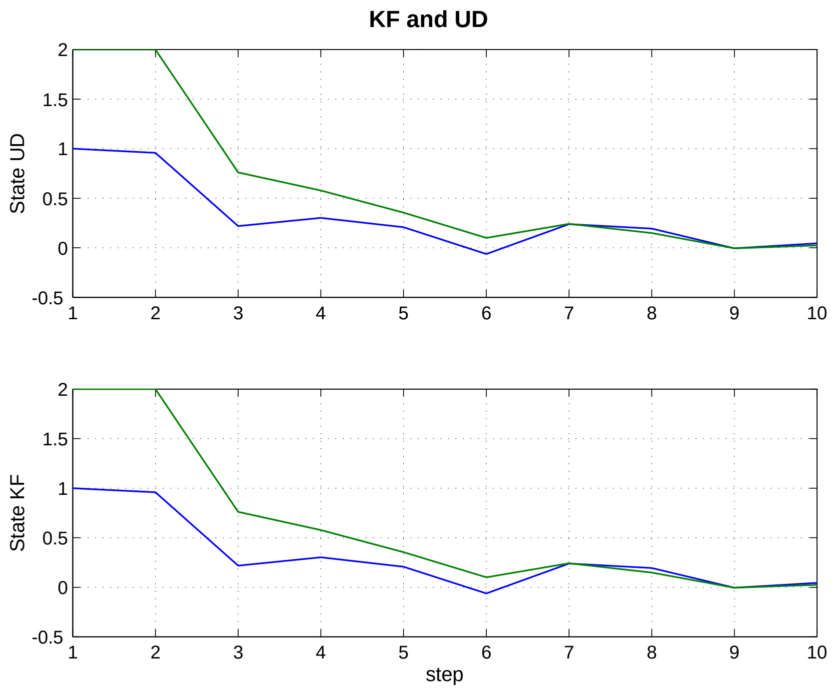
Two routines are available for variable gain estimators. These are `KFilter` for conventional Kalman filters and `UDKalmanFilter`, for filters in which the plant matrix is formulated in Upper Diagonal (UD) form. The latter has better numerical properties. Both filters are demonstrated for double integrator plant in the demo `UDKFDemo`. The results are compared in Figure 28.2 on the next page.

As expected with double precision arithmetic both filters produce the same result. With single precision arithmetic the UD form should be better results. Note that `KFilter` eliminates the problem of instabilities due to asymmetric covariance matrices by forcing it to be symmetric each step.

28.4 Extended Kalman Filter

The extended Kalman Filter is used for attitude determination and many other applications. In an extended Kalman filter the state transition matrix and measurement matrix is linearized about the current estimated state. This is in contrast to the linearized Kalman Filter in which the measurement and state transition matrices are linearized about a predetermined state or state trajectory. In the context of attitude determination, attitude measurements are unit

Figure 28.2: UD filter



vector measurements. Since a unit vectors magnitude is 1 there are only 2 independent quantities. Thus a unit vector provides only a two-axis measurement. Since complete attitude knowledge requires at least 3-measurements a multiple unit vector measurements need to be used. Sources for measurements are

- Sun
- Planets
- Moon
- Magnetic field
- Stars
- Features on a planetary surface

In each case a reference catalog for the object is maintained in the software. The first four provide unambiguous references. However, stars and planetary features need to be identified. The next chapter goes into stellar attitude determination in great detail. The abstracting of all attitude measurements into unit vectors in the toolbox makes it possible to have a unified approach to attitude determination. There are two approaches to the plant model. One is to run gyros all the time as model only the gyro noise. This approach is used on high precision pointing spacecraft. The other approach is to model the dynamics of the spacecraft. This approach requires a reasonable plant model. If the spacecraft is under closed loop control, a low order model can be extracted from the closed loop dynamics and used when control is active. Otherwise it is necessary to accurately model the spacecraft dynamics, including all control inputs.

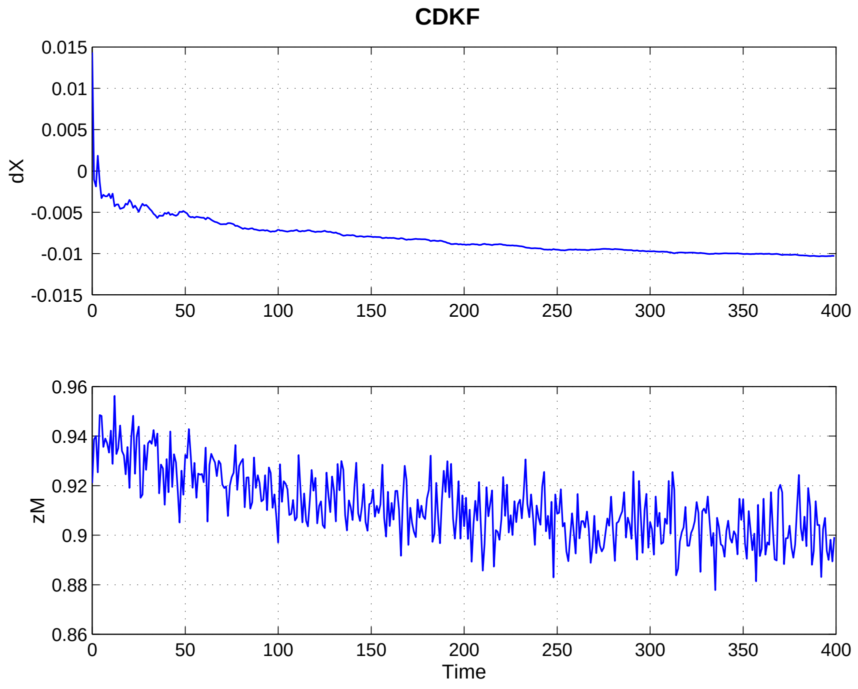
Two examples are given in this module, `SunMagAttDet` used in `MagSimWithEstimator`, and `SunEarthADSim`. The first combines sun and magnetic field measurements and the latter, sun and earth measurements. Another demo, `KFSunAndEarthTest2` compares different types of sensors.

28.5 Continuous Discrete Extended Kalman Filter

28.5.1 Introduction

The script `CDKFDemo` demonstrates the use of a Continuous Discrete Kalman Filter with a nonlinear spring model, and a nonlinear position measurement. The first plot in Figure 28.3 shows that the estimate converges to the true value of the spring position less a bias. The second shows the noisy measurement.

Figure 28.3: Continuous discrete Kalman filter



28.5.2 Orbit Determination

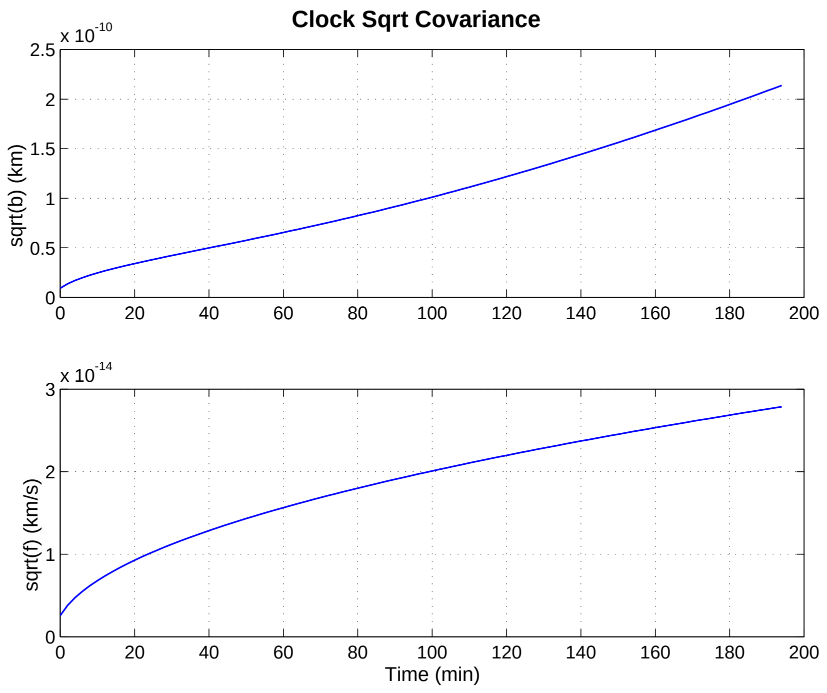
The orbit determination system used in this module assumes that all measurements used for orbit determination are range measurements. These can be from GPS satellites or ground stations. Both range and range rate are assumed to be available. One measurement gives enough information to estimate position and velocity along a single axis. At least three measurements are needed for a position fix. The filter incorporates each measurement as it arrives and propagates the orbit between measurements by numerically integrating the orbital equations of motion.

An orbit estimator is demonstrated in the script `OrbitKalmanFilter`. The clock error covariance is shown in Figure 28.4 on the next page.

28.6 Unscented Kalman Filter

The Unscented Kalman Filter (UKF) is able to achieve greater estimation performance than the Extended Kalman Filter (EKF) through the use of the unscented transformation (UT). The UT allows the UKF to capture first and second order terms of the nonlinear system. Instead of just propagating the state, the filter propagates the state and additional sigma points which are the states plus the square roots of rows or columns of the covariance matrix. Thus the state

Figure 28.4: Clock error covariance for an orbit Kalman Filter



and the state plus a standard deviation are propagated. This captures the uncertainty in the state. It is not necessary to numerically integrate the covariance matrix.

Unscented Kalman filters are discussed in more detail in Chapter 30.

CHAPTER 29

STELLAR ATTITUDE DETERMINATION

29.1 Introduction

This chapter describes how to use the stellar attitude determination routines in the Estimation module to develop attitude determination systems for your applications. The process of stellar attitude determination is shown in Figure 29.1 on the following page.

A simulation generates an ECI to body attitude quaternion and body rates. These are used for attitude determination. The body rates are only needed if the attitude determination system is gyro based. That means the model used by the filter is the model of the gyro, not of the spacecraft.

The simulated quaternion determines the pointing direction. If you want to test star centroiding you need a catalog of star images that cover the entire sky. For many purposes you can bypass that step and pass the measured unit vectors from the pinhole camera directly to the star identification software. At that point the pyramid algorithm, that uses four stars at a time, is used to determine which stars are in the field-of-view. This information is passed to the Kalman filter or least squares algorithm to determine the attitude.

29.2 Star Catalogs

The toolbox includes the catalogs given in Table 29.1.

Table 29.1: Star catalog

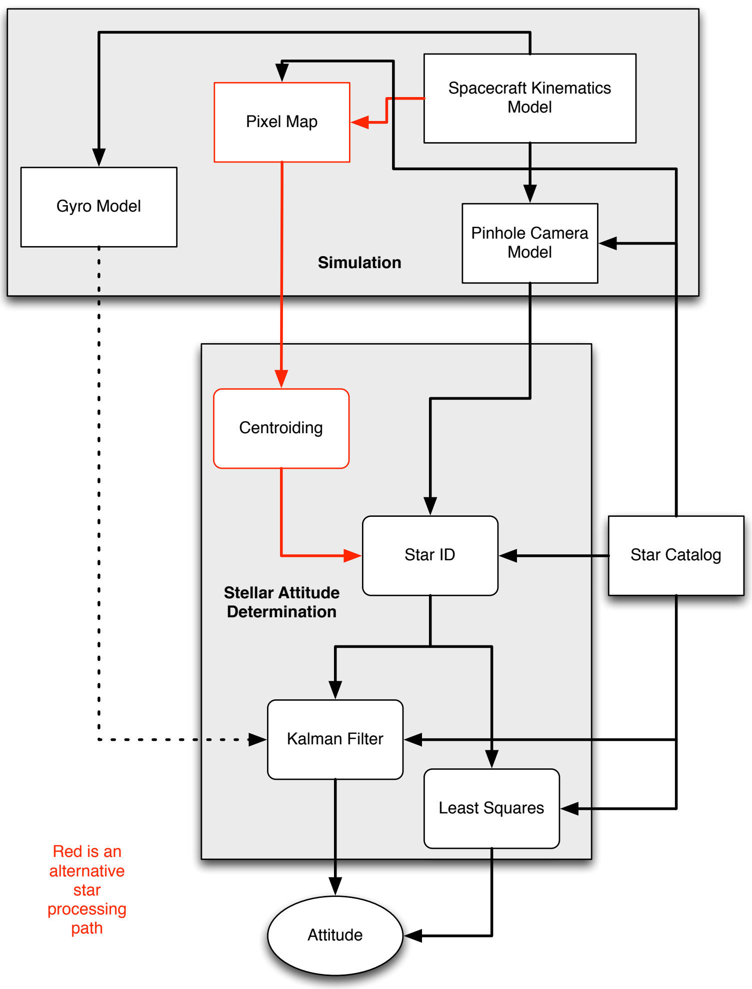
Catalog	Number of Stars	Minimal Visual Magnitude	Description
Hipparcos	118218	14	Stars measured by the Hipparcos
FK5	1536	8.35	Standard star catalog
FK5 Polar	14	7.17	Polar stars from FK5
FK5 Short	768	7.75	Subset of FK5

You load a catalog by typing:

Listing: Catalog

```
starCatalog = LoadCatalog( 'FK5_Short', 'all' )

starCatalog =
name1: [768x8 char ]
name2: [768x8 char ]
```

Figure 29.1: Stellar attitude determination process.



```
name3: [768x8 char ]
rA: [ 1x768 double]
dec: [ 1x768 double]
pMRA: [ 1x768 double]
pMDec: [ 1x768 double]
parallax: [ 1x768 double]
rV: [ 1x768 double]
vM: [ 1x768 double]
```

`'all'` may be replaced by a maximum visual magnitude which is determined by the sensitivity of the star camera that you plan to use in your application. If you type

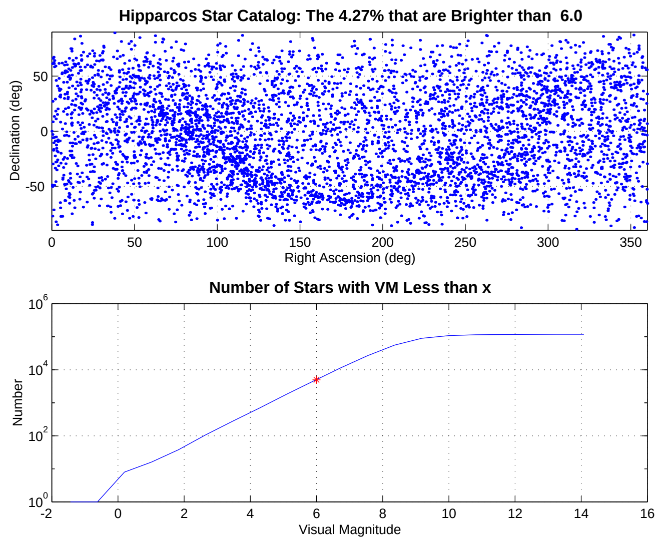
`LoadCatalog`

you will get the default Hipparcos catalog with stars brighter than visual magnitude 6. With no outputs, Figure 29.2 is generated.

Table 29.2: Star catalogs

Catalog	Number of Stars	Minimum visual magnitude	Description
Hipparcos	118218	14	Stars measured by the Hipparcos satellite.
FK5	1536	8.35	Standard star catalog.
FK5 Polar	14	7.17	Polar stars from FK5.
FK5 Short	768	7.7500	Subset of FK5.

Figure 29.2: LoadCatalog default



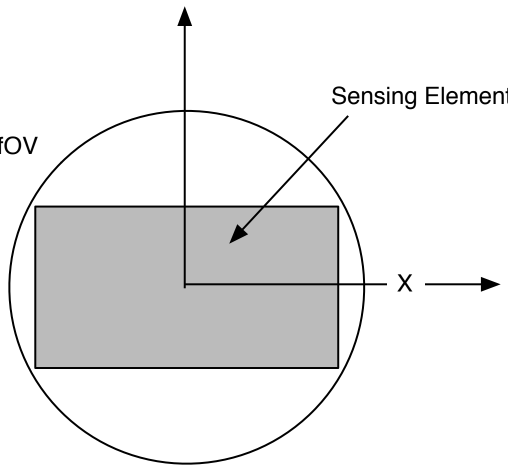
The red mark shows the selected visual magnitude. The upper picture shows the star map. After generating a catalog you need to generate a dot product catalog. The dot products are used in the star identification procedures. This is done using `StarDataGeneration`,

```
d = StarDataGeneration( starCatalog, fieldOfView, angularResolution )
```

where `starCatalog` is the data structure generated above, `fieldOfView` is the field-of-view in radians and `angularResolution` is the minimum resolvable angle.

The data structure d has several fields. They are given in the following table. `nearMatrix` is a 3-by-m matrix with dot products for all stars and their neighbors within the `fieldOfView` of the camera. That is, if a star is in the center of the field-of-view, it gives dot products for all stars that it can see. `fieldOfView` should be chosen as in illustration in Figure 29.3. The `fieldOfView` should circumscribe the sensing element. `StarDataGeneration` does not

Figure 29.3: FOV determination



store duplicate pairs. Once you have run `StarDataGeneration` you can save the results in a mat file and use that. Unless you change the star catalog, `dPTol` or `fOV` there is no reason to rerun the function.

Table 29.3: StarDataGeneration output fields

Field	Size	Units or Type	Description
starMatrix	(4,:)	[intensity;rad;rad;rad]	An array with intensity and unit vectors for all stars in star catalog.
nearMatrix	(3,m)	[dot product;star 1;star 2]	The first row is the dot product and the next two rows are the two stars corresponding to the dot product pair.
starCatalog	(1,1)	Data structure	The reduced star catalog.
angularResolution	(1,1)	rad	The input angular resolution.
fieldOfView	(1,1)	rad	The input field of view fOV Sensing element .

29.3 Generating Star Data

There are two functions that you can use to generate star data. One, `PixelMapCentroid`, generates star data from input images. The second, `PinholeStarCamera`, generates star data from a star catalog. The latter is useful for simulation. Both output the same data structure shown in Table 29.4.

Table 29.4: `StarMeasurement` data structure

Name	Size	Units or Type	Description
pixelLocation	(2,m)	pixels	The star centroid in pixels
intensity	(2,m)	intensity	The intensity of the star

29.3.1 `PixelMapCentroid`

PixelMapCentroid is invoked with the call

```
starMeasurement = PixelMapCentroid( pixelMap, threshold, show )
```

`pixelMap` may be a pixel map, a string with the name of an image file, or an empty matrix in which case the function brings up a file selection dialog. threshold is from 0 to 1 and tells the function to only include stars whose intensity is that fraction of the maximum intensity measured. Any argument for show will generate a set of plots.

You can generate a pixel map by typing

```
pixelMap = double(imread( fileName ) );
```

where `fileName` is the name of an image file. See `imread` for more information. Doing this outside of `PixelMapCentroid` can save time if you want to experiment with different thresholds.

29.3.2 `PinholeStarCamera`

`PinholeStarCamera` is invoked with the call

```
[starMeasurement, camera] = PinholeStarCamera( bCatalogToCamera, camera,
    starCatalog, drawImage );
```

`starCatalog` is the star catalog data structure generated by `LoadCatalog`. `camera` is the camera data structure. The fields in the camera data structure are, `wX`, the pixel width in x, `wY` the pixel width in y and `fScale` which can be found from

```
f = ComputePixelMapScale( wX, xFieldOfView )
```

Any value for `drawImage` cause an image to be drawn.

29.4 Centroiding

The first step in star identification is to take the pixel map from the CCD camera and to identify stars and compute centroids. This can be done with the function

```
PixelMapCentroid
```

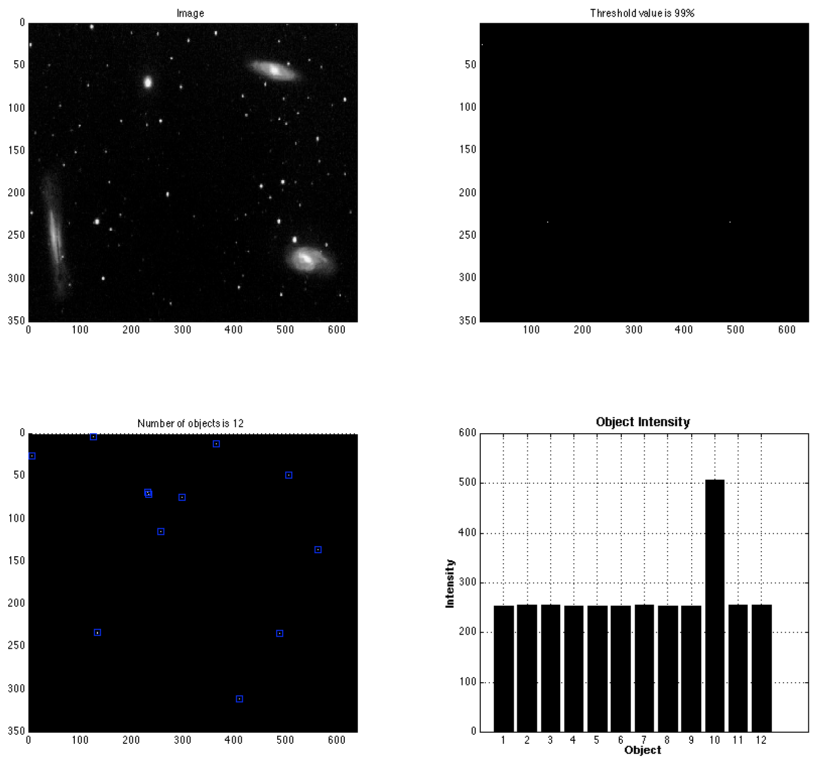
If you just type in the above name in a MATLAB command window and then select the file "StarFieldDemo.jpg" you will get the display shown in Figure 29.4 on the next page.

12 objects are identified. It is important to set the minimum tolerance appropriately. Every pixel will have some finite value due to dark current and noise. If the minimum tolerance is set too low these pixels will be identified as stars. If this is combined with relatively low resolutions, it may result in misidentified stars.

Another approach is `StarCentroidCOM`. This finds the centroid given intensities on n pixels and their coordinates using the center of mass method. The center of mass will have larger errors if the width of the region of interest goes outside the pixel mask. Generally, star measurements on the edge will not be reliable due to optical errors so should be discarded if you have enough stars that for than n pixels from the edges. Figure 29.5 on page 225 the result from the built-in demo.

```
>> StarCentroidCOM

Measured Centroids (microns)
     70.0000      90.0000   12665.3031
    167.8630     386.0009   14374.9016
    250.2159     230.0000   19686.0882
    359.9821    1240.1813   37047.1646
    383.9552     321.9790   19828.4600
    530.2069     881.9847   29506.0417
    879.9857    1041.9738   37011.9382
```

Figure 29.4: Centroiding demo



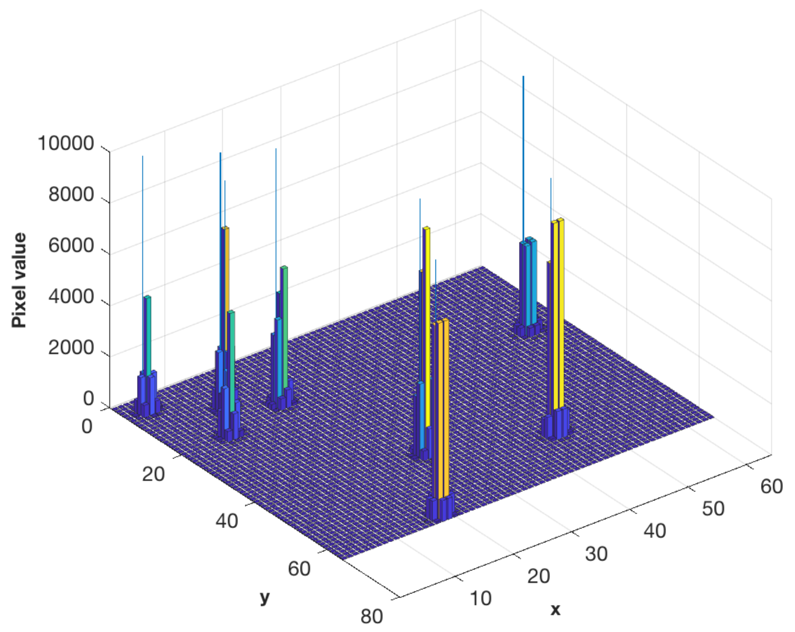
```
   1160.1834      439.9843  17161.5555

True Centroids (The order may be different)
     70.0000       90.0000   12666.9016       14.0000
    168.0000      386.0000   14476.4591       16.0000
    250.0000      230.0000   19905.1312       22.0000
    360.0000     1240.2000   37079.0291       41.0000
    384.0000      322.0000   19905.1310       22.0000
    530.0000      882.0000   29857.6965       33.0000
    880.0000     1042.0000   37095.9263       41.0000
   1160.2000      440.0000   17190.7948       19.0000
```

29.5 Star Identification

The Spacecraft Control Toolbox employs the Pyramid algorithm for star identification. The star ID algorithm begins with proper processing of the star catalog. We have implemented the k-vector search less technique for looking up star pairs based on measured angles. This is combined with a pyramid star algorithm that find star triads and then matches them with an additional fourth star, nearly eliminating incorrect star identification.

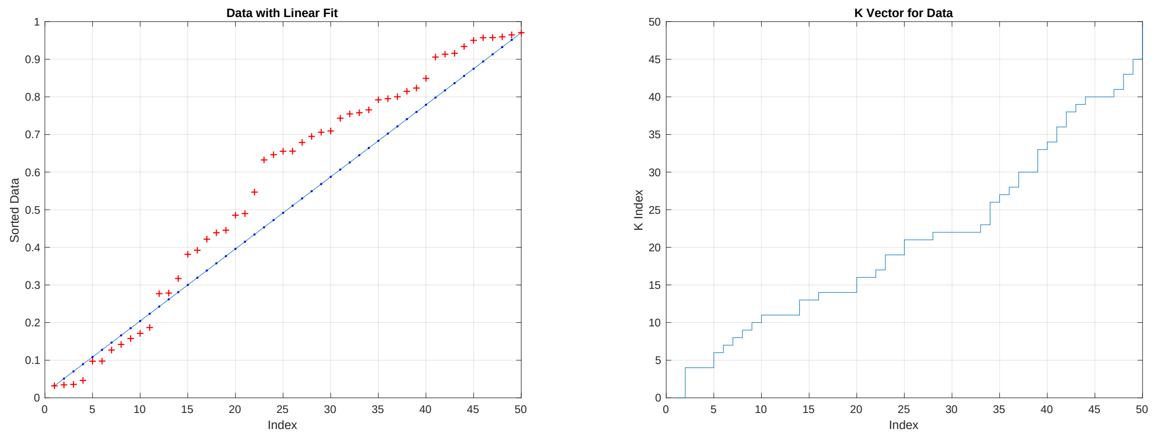
The k-vector range search technique allows a range of values from the near matrix to be extracted without searching, by using indexing. The star angles of the star pairs in the near matrix are sorted in ascending order, producing an array of star angles s, and an additional integer index value is assigned to each pair. A slope and intercept is calculated for an equivalent straight line through the data. The indices of s can then be calculated for a given star angle range and all values within the range easily retrieved. Practically, the index k gives the number of elements in the angle array below the value dictated by the equivalent line.

Figure 29.5: `StarCentroidCOM` demo



The script `CreateKVector` produces a demo in MATLAB with the plots in Figure 29.6. This is for a simple set of random numbers on the interval (0,1). The plot on the left shows the steps in k, in a stair plot, as compared against a linear progression. For example, at index 4, k jumps from 2 to 4, indicating that there are 4 data points with values below the line value at that index. On the right, the data is printed against the straight line fit to the endpoints, with a range calculated of 0.5 to 0.75. The range is marked with the magenta solid lines. The data points that are circled are the values returned as in the desired range. We can see in this particular example that one data point below 0.5 was returned, because it is also above the value of the closest line point, 0.4642. The line points bracketing the range are indicated by the black dashed lines.

Figure 29.6: K-Vector Example in MATLAB



When using k-vector with measured star pairs, the angle range will be the measured angle θ plus and minus some δ. This should be a high confidence interval such as 4σ. This could be a constant value for the entire pixel array or calculated based on the star locations or angle as is explained in the next section.

The pyramid algorithm[1] begins with star triads but adds a fourth star to the pattern for more precise identification. It has been tested on orbit on the HETE spacecraft.

The demo `StarIDPyramidDemo`.

```
%% Parameters
% Constant
degToRad        = pi/180;

% Camera parameters
fOV             = 30*degToRad;
fScale          = 1;
nPixels   = 1024;

qBToS           = [1;0;0;0];
uS              = [0 0 1]';

%% Create a random star catalog
nStars          = 1000;
angRes    = 5*fOV/nPixels;
[~,starCatalog]   = RandSC(nStars);
d                 = StarDataGeneration( starCatalog, fOV*sqrt(2), angRes );

%% Identify stars form random initial conditions
disp('Demo␣of␣StarIDPyramid␣--')
options   = StarIDPyramid;
options.pixelMapScale = fScale;
for k = 1:100
  q                        = QRand;
  s                        = StarSensor( q, qBToS, uS, d.starMatrix, fOV, fScale )
      ;
  starMeas.pixelIntensity = s(1,:);
  starMeas.pixelLocation  = s(2:3,:);
  starID                   = StarIDPyramid( starMeas, d, options );
  fprintf('Quaternion:␣[%f␣%f␣%f␣%f]\n',q)
  fprintf('Star␣IDs:␣␣␣␣%s\n',num2str(starID));
end
```

shows how to use the pyramid algorithm. Only the first few quaternions are shown below.

```
>> StarIDPyramidDemo

StarDataGeneration: Eliminating stars that cannot be separated based on the input
     angular resolution.

StarDataGeneration:     2 stars will be eliminated that are too close to other
    stars.
                      998 stars will be in the reduced catalog.
Demo of StarIDPyramid --
Quaternion: [0.578263 -0.410593 0.341393 -0.616828]
Star IDs:    36  156  208  249  256  322  350  363  368  389  397  482
Quaternion: [0.708439 0.147505 -0.272617 -0.634064]
Star IDs:    9   22  176  210  238  353  391  413  427  432  465  518
Quaternion: [0.549248 -0.491669 -0.015056 -0.675545]
Star IDs:    4   25   29   93  146  162  165  193  237  271  340  343
```

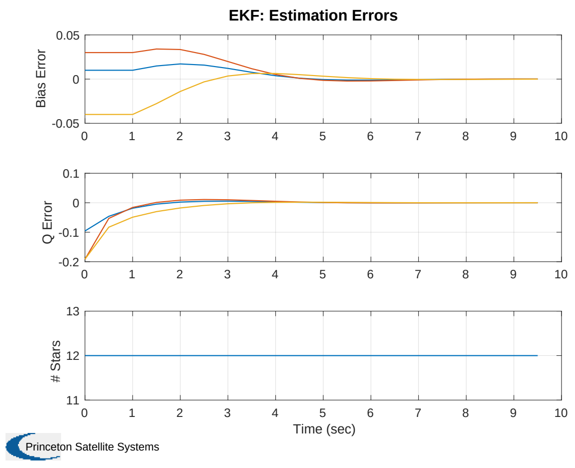
The pre-processing of the catalog removes closely spaced stars.

[1]Mortari, Samaan, Bruccoleri, and Junkins. *The Pyramid Star Identification Technique*, Navigation, Vol. 51, No. 3, Sept. 2004

29.6 Kalman Filtering

The final step is to integrate these functions with filtering. Both gyro based and gyro-less algorithms are available. You can use the Extended Kalman Filter, Least Squares or the Unscented Kalman Filter. UKF is useful with nonlinear sensor models. `AttDetEKFSim` is a simulation using gyros and the EKF. `StellarAttDetEKF` implements stellar attitude determination with the EKF. Figure 29.7 shows the convergence of the attitude errors.

Figure 29.7: Use of the EKF for stellar attitude determination.



CHAPTER 30

UNSCENTED KALMAN FILTER

30.1 Introduction

The unscented Kalman Filter is an alternate approach to estimating states and parameters in a nonlinear system. In fact, it is not necessary to distinguish between states and parameters at all. A major advantage, albeit at the cost of additional computation, is that it can employ any nonlinear measurement model or plant model.

This chapter gives an example of an unscented Kalman Filter.

30.2 Pressure Kalman Filter

This demonstration in `PressureKF` for a fuel mass Kalman filter for a blowdown system. In a blowdown system, the fuel is pressurized by a gas, such as helium. As fuel is consumed the gas expands into the empty volume and the pressure decreases. The estimator uses a pulsewidth model and a pressure measurement. The pressure measurements on a spacecraft are generally low resolution, typically 8 bits over the entire pressure range (of 350 psi to 100 psi.) Every time a thruster fires, the model computes the fuel used based on a model of fuel consumption versus pulsewidth. This is the dynamical model. The measurement of pressure is then incorporated into the estimate using an Unscented Kalman Filter (UKF) implemented in `UKF`. The UKF can use a nonlinear measurement and plant (dynamics) model directly without any linearization.

The script computes a random pulsewidth from 0 to 8 seconds in length when a second random number between 0 and 1, is greater than 0.95. The pressure measurement has 12 bits resolution and random noise.

The script is fairly short so we show it all here. A problem that may occur is if you have defined `d` prior to running this script. You may get an error

```
>> PressureKF
Unable to perform assignment because dot indexing is not supported for variables
    of this type.
Error in PressureKF (line 56)
d.x            = 0.9997*mFuel;
```

Just `clear all`.

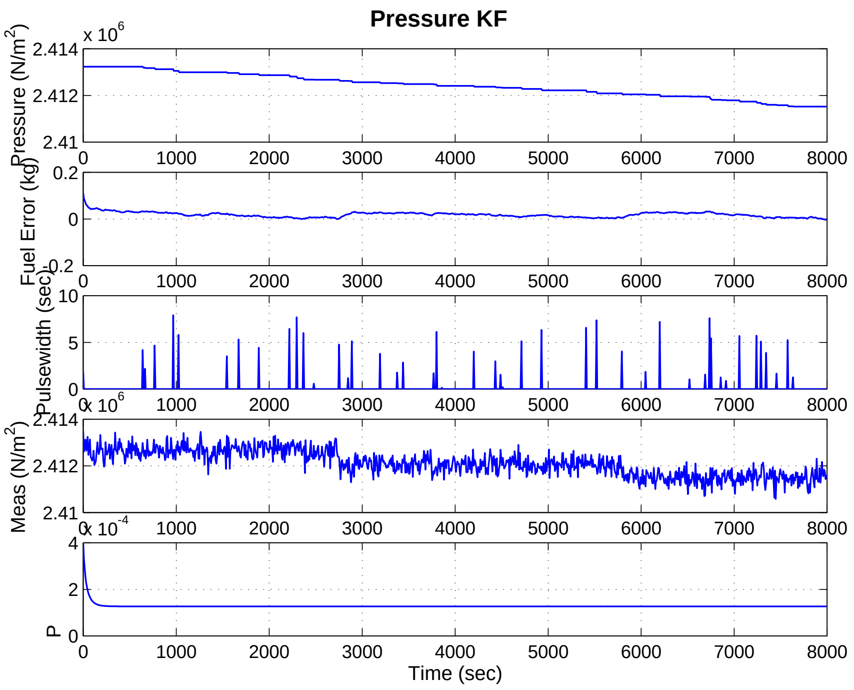
Listing 30.1: Fuel mass Kalman filter example *PressureKF.m*

```
%% Demonstrate a fuel mass Kalman filter for a blowdown system.
% In a blowdown system, the fuel is pressurized by a gas, such as
% helium. As fuel is consumed the gas expands into the empty volume
% and the pressure decreases. The estimator
% uses a pulsewidth model and a pressure measurement. The pressure
% measurements on a spacecraft are generally low resolution, typically
% 8 bits over the entire pressure range (of 350 psi to 100 psi.)
% Every time a thruster fires, the model computes the fuel used
% based on a model of fuel consumption versus pulsewidth. This is
% the dynamical model. The measurement of pressure is then
% incorporated into the estimate using an Unscented Kalman Filter
% (UKF). The UKF can use a nonlinear measurement and plant (dynamics)
% model directly without any linearization.
%
% The script computes a random pulsewidth from 0 to 8 seconds in
% length when a second random number between 0 and 1, is greater than
% 0.95.
%
% The pressure measurement has 12 bits resolution and random noise.
%
% The script plots true pressure, fuel mass estimated errors, pulsewidth
% and the pressure measurement.
%
% ------------------------------------------------------------------------
%  See also NQuant, Plot2D, UKF, BloDownMass, UE, BloDown, MolWt2R
% ------------------------------------------------------------------------
%%
%--------------------------------------------------------------------------
%   Copyright (c) 2006 Princeton Satellite Systems, Inc.
%   All rights reserved.
%--------------------------------------------------------------------------

uE      = 200*9.806;
thrust  = 1;
dT      = 8;
mDot    = thrust/uE;

T       = 300;
vTank   = 0.5;
rhoFuel = 1000;
mPress  = 0.5;
rPress  = MolWt2R( 0.004 ); % He

nSim    = 1000;

p       = 350*6895;
mFuel   = BloDownMass( mPress, rhoFuel, vTank, rPress, T, p );
xPlot   = zeros(6,nSim);
lSB     = p/2^12;
sigQ    = sqrt(NQuant( lSB, 'truncate' ));
sigN    = lSB/2;

%% Estimation parameters
%-----------------------
d.x           = 0.9997*mFuel;
d.rHSFun      = 'RHSPressure';
d.measFun     = 'GPressure';
d.measFunData = struct('vTank',vTank,'rhoFuel',rhoFuel,'mPress',mPress,'rPress',rPress,'T',T)
    ;
d.alpha       = 0.5;
d.kappa       = 0;
d.beta        = 2;
d.dY          = 1;
d.dT          = dT;
d.rM          = sigQ^2 + sigN^2;
d.rP          = 0.00001;
d.p           = 40*d.rP;
d             = UKF('initialize', d );

for k = 1:nSim
```

```
  if( rand > 0.95 )
    pw = rand*8;
  else
    pw = 0;
  end
  p             = BloDown( mPress, rhoFuel, vTank, rPress, T, mFuel );
  pMeas         = lSB*floor(p/lSB) + sigN*randn;
  xPlot(:,k)    = [p;mFuel;pw;pMeas;d.x;d.p];
  mFuel         = mFuel - mDot*pw;
  d.t           = 0;
  d.rHSFunData = struct('mDot',mDot,'pw',pw,'dT',dT);
  d             = UKF( 'update', d, pMeas );
end

yL = {'Pressure␣(N/m^2)' 'Fuel␣Error␣(kg)' 'Pulsewidth␣(sec)' 'Meas␣(N/m^2)' , 'P'};
Plot2D( (0:(nSim-1))*dT,[xPlot(1,:);xPlot(2,:) - xPlot(5,:);xPlot([3 4 6],:)] , 'Time␣(sec)',
     yL, 'Pressure␣KF')
```

PressureKF.m

Figure 30.1: Pressure Kalman Filter



The top plot in Figure 30.1 shows the pressure. The second shows the fuel mass estimation error. The next plot shows the pulsewidth of thruster firings which are consuming the fuel. The noisy pressure estimate is shown below that. The covariance, shown in the last plot, decreases rapidly and then is constant. This is confirmed by the mass error which reaches steady state quickly.

APPENDICES

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 237.

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

These examples require the GOES CD. `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 238 shows the resulting plot.

A.4.3 Weather

`Weather` gives a summary plot of all GOES data. For example, type

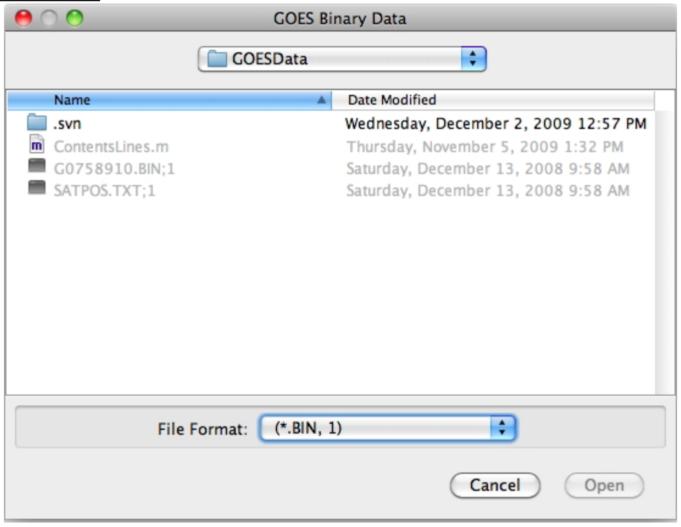
```
>> Weather
```

and select the file in the GOESData folder. You will then see the plot in Figure A-4 on page 238.

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

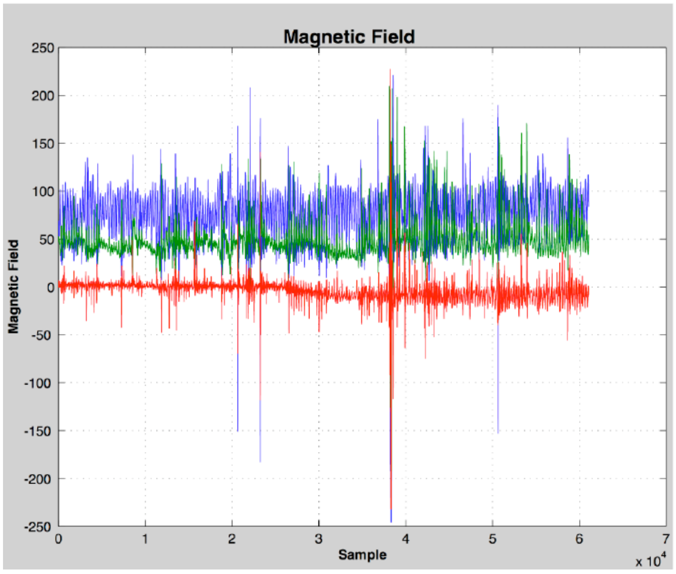


Figure A-3: GOES Electrons

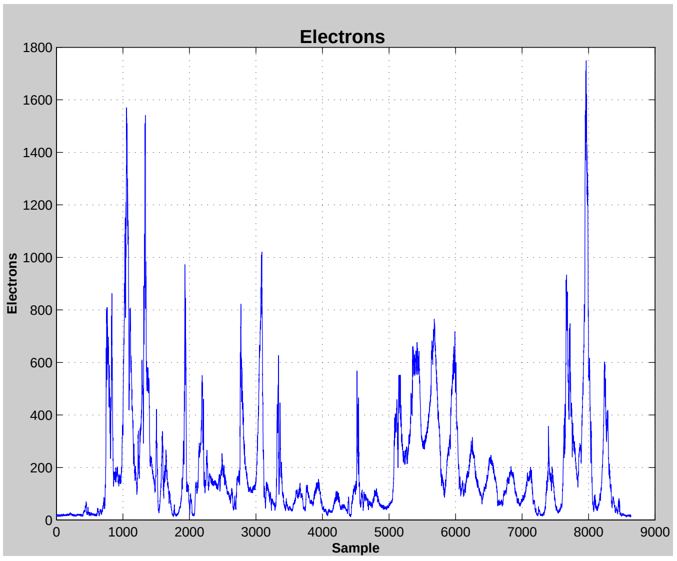
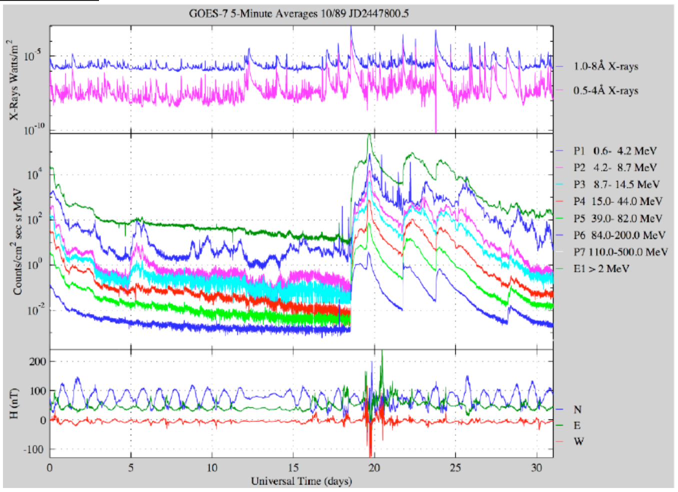


Figure A-4: GOES Weather



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.

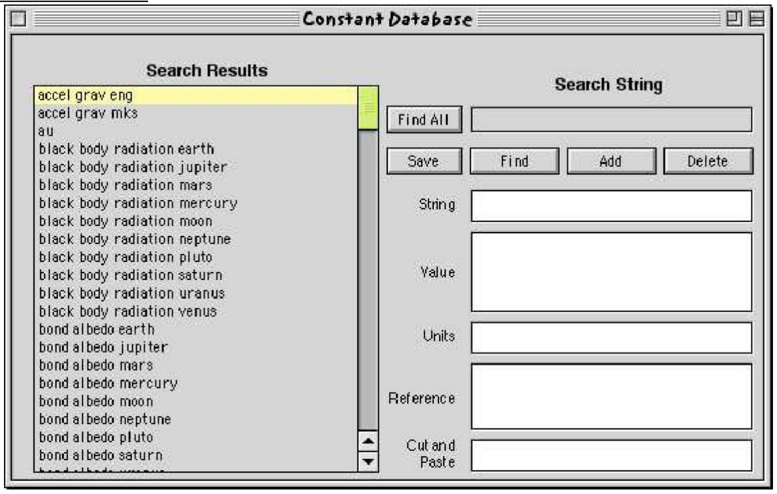
B.1 The Constant Database

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

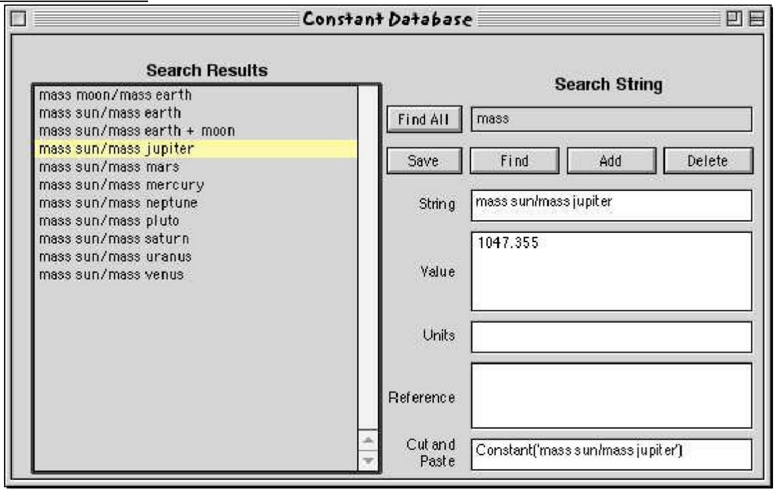


```
accel grav eng
accel grav mks
au
black body radiation earth
black body radiation jupiter
black body radiation mars
black body radiation mercury
black body radiation moon
black body radiation neptune
black body radiation pluto
black body radiation saturn
black body radiation uranus
black body radiation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

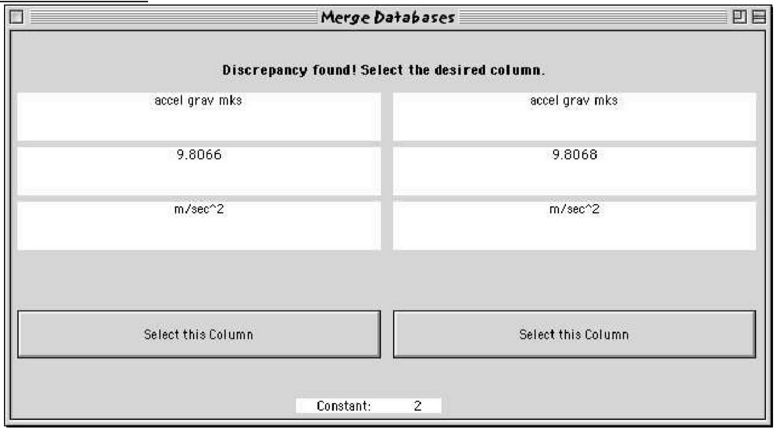
```
MergeConstantDB( 'initialize', a, b )
```

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


Merge Databases

Discrepancy found! Select the desired column.

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

APPENDIX C

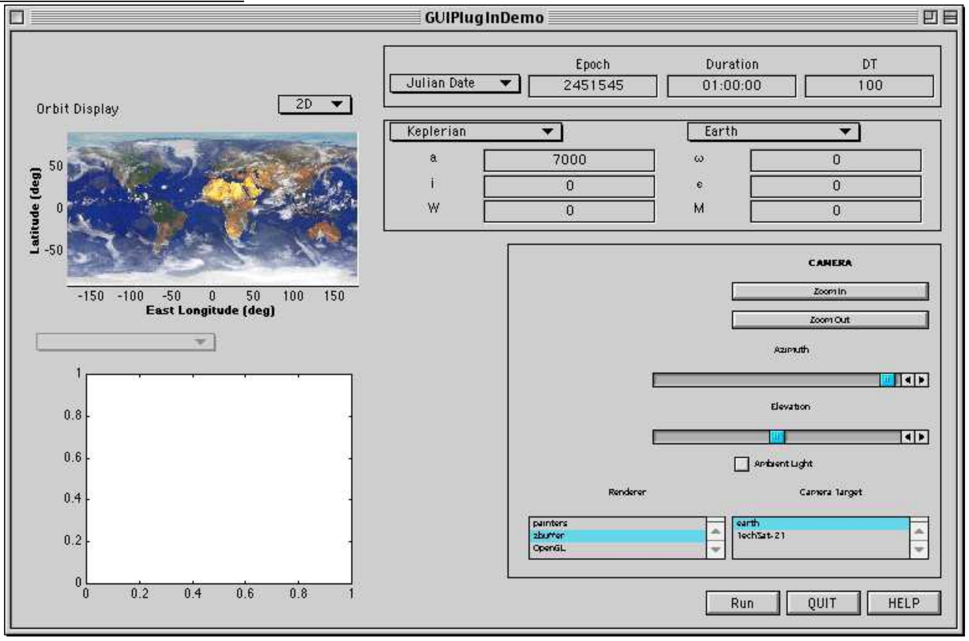
GUI PLUG INS

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



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

Keplerian — Earth

a	7000	ω	0
i	0	e	0
W	0	M	0

Renderer: painters, [✓ zbuffer], OpenGL

Camera Target: [✓ earth], TechSat-21

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

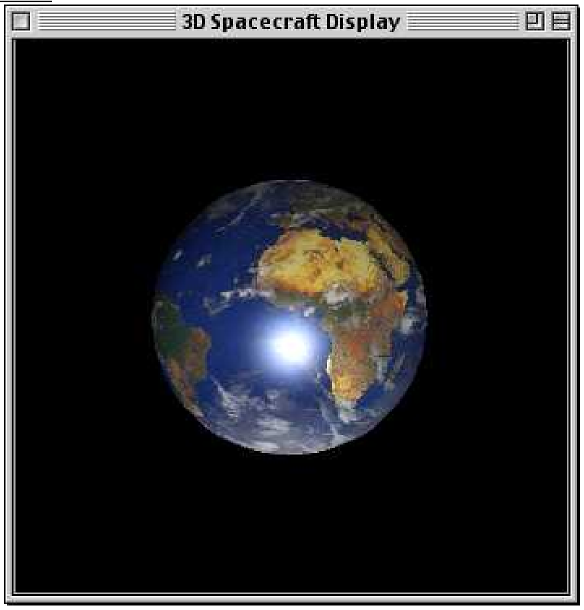
- OrbitDisplayPlugin

- 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 248.

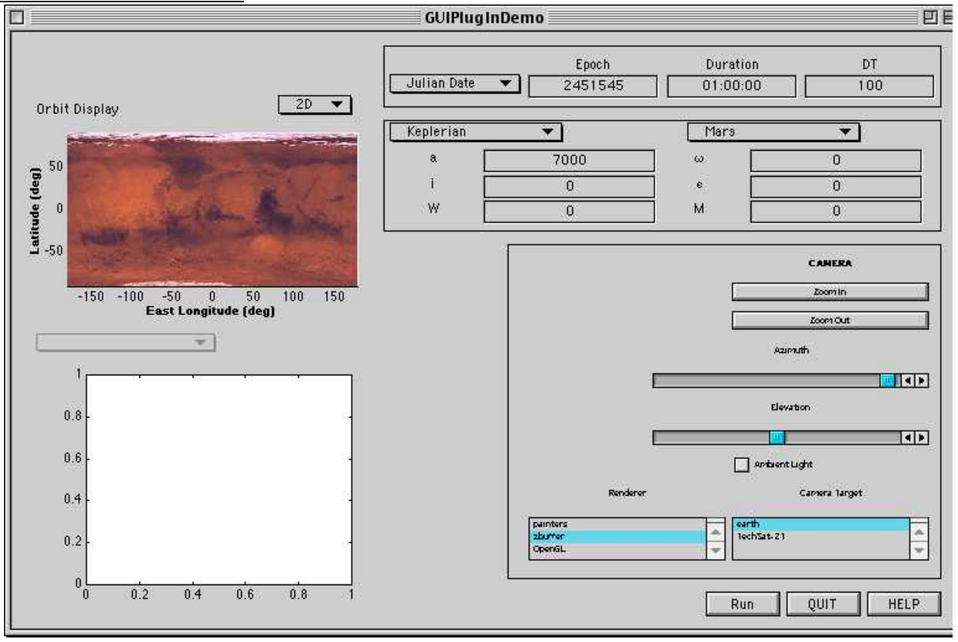
C.2 Writing Your Own GUI Function

`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



Julian Date

Epoch	Duration	DT
2451545	01:00:00	100

Keplerian — Mars

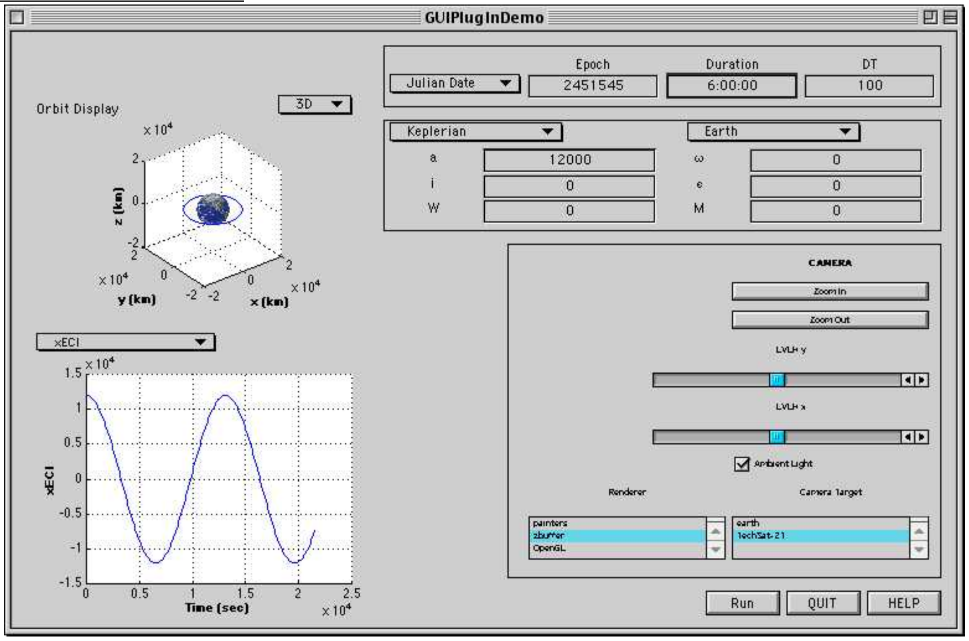
a	7000	ω	0
i	0	e	0
W	0	M	0

☐ Ambient Light

Renderer: painters, zbuffer, OpenGL (zbuffer selected)

Camera Target: earth, TechSat-21 (earth selected)

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



Julian Date

Epoch	Duration	DT
2451545	6:00:00	100

Keplerian — Earth

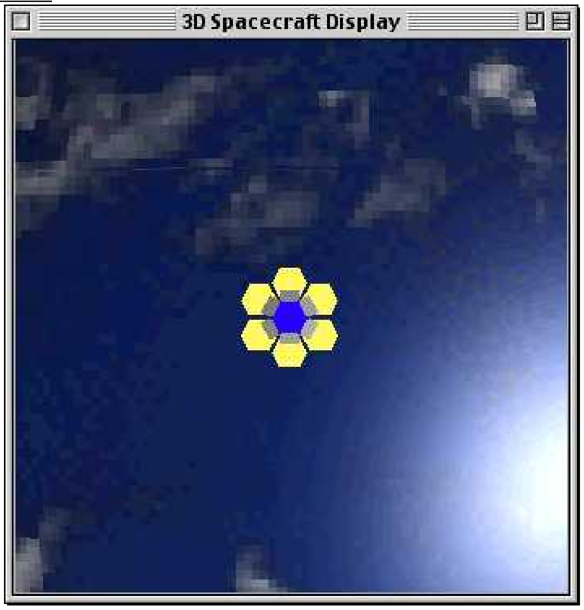
a	12000	ω	0
i	0	e	0
W	0	M	0

☑ Ambient Light

Renderer: painters, zbuffer, OpenGL (zbuffer selected)

Camera Target: earth, TechSat-21 (TechSat-21 selected)

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