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

Spacecraft Control Toolbox

May 31, 2012

©Copyright 1996-2011 by Princeton Satellite Systems, Inc. All rights reserved.

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

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

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

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

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

List of Examples viii

I Toolbox Basics 1

1 Introduction 3
  1.1 Organization .............................................. 3
  1.2 Requirements ........................................... 4
  1.3 Installation ............................................. 4
  1.4 Getting Started .......................................... 4

2 Getting Help 7
  2.1 MATLAB’s Built-in Help System .......................... 7
  2.2 Command Line Help ....................................... 8
  2.3 FileHelp ................................................ 11
  2.4 Searching in File Help ................................... 13
  2.5 DemosPSS ............................................... 14
  2.6 Graphical User Interface Help ............................ 15
  2.7 Technical Support ...................................... 15

3 Basic Functions 17
  3.1 Introduction ............................................. 17
  3.2 Function Features ....................................... 17
  3.3 Example Functions ...................................... 20
  3.4 GUI Tools in the Toolbox ............................... 27

4 SCT Tutorial 29

5 Coordinates 33
  5.1 Transformation Matrices ................................. 33
  5.2 Quaternions ............................................ 34
  5.3 Coordinate Frames ..................................... 35

6 CAD 37
  6.1 Introduction ............................................ 37
  6.2 Building a Spacecraft Model Using BuildCADModel 38
  6.3 Visualizing your CAD Model ........................... 58

7 Attitude Simulations 63
  7.1 Dynamics Models ....................................... 63
  7.2 Wire Model ............................................. 64
  7.3 Tree Model ............................................ 68
<table>
<thead>
<tr>
<th>Content</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Ephemeris</td>
<td>73</td>
</tr>
<tr>
<td>8.1 Overview</td>
<td>73</td>
</tr>
<tr>
<td>8.2 Almanac functions</td>
<td>73</td>
</tr>
<tr>
<td>8.3 JPL Ephemeris</td>
<td>74</td>
</tr>
<tr>
<td>9 Plotting</td>
<td>77</td>
</tr>
<tr>
<td>9.1 Introduction</td>
<td>77</td>
</tr>
<tr>
<td>9.2 Getting Started</td>
<td>78</td>
</tr>
<tr>
<td>9.3 GUI Layout and Features</td>
<td>79</td>
</tr>
<tr>
<td>9.4 Creating Figures</td>
<td>81</td>
</tr>
<tr>
<td>9.5 Grouping Variables</td>
<td>82</td>
</tr>
<tr>
<td>9.6 Excluding Variables</td>
<td>83</td>
</tr>
<tr>
<td>9.7 Deriving New Data</td>
<td>83</td>
</tr>
<tr>
<td>9.8 Animation</td>
<td>84</td>
</tr>
<tr>
<td>9.9 Templates</td>
<td>84</td>
</tr>
<tr>
<td>9.10 Plotting Output from Custom Simulation Software</td>
<td>85</td>
</tr>
<tr>
<td>9.11 Summary</td>
<td>89</td>
</tr>
<tr>
<td>10 Pointing Budgets</td>
<td>91</td>
</tr>
<tr>
<td>10.1 Pointing Budget</td>
<td>91</td>
</tr>
<tr>
<td>10.2 Pointing Budget GUI</td>
<td>92</td>
</tr>
<tr>
<td>11 Designing Controllers</td>
<td>95</td>
</tr>
<tr>
<td>11.1 Using the block diagram</td>
<td>95</td>
</tr>
<tr>
<td>11.2 Linear Quadratic Control</td>
<td>96</td>
</tr>
<tr>
<td>11.3 Single-Input-Single-Output</td>
<td>96</td>
</tr>
<tr>
<td>11.4 Eigenstructure Assignment</td>
<td>100</td>
</tr>
<tr>
<td>12 CubeSat</td>
<td>103</td>
</tr>
<tr>
<td>12.1 Modeling</td>
<td>103</td>
</tr>
<tr>
<td>12.2 Simulation</td>
<td>104</td>
</tr>
<tr>
<td>12.3 Mission Planning</td>
<td>107</td>
</tr>
<tr>
<td>12.4 Visualization</td>
<td>108</td>
</tr>
<tr>
<td>II Attitude Control Design Examples</td>
<td>111</td>
</tr>
<tr>
<td>13 Attitude Control</td>
<td>113</td>
</tr>
<tr>
<td>13.1 Module Overview</td>
<td>113</td>
</tr>
<tr>
<td>13.2 Design Examples</td>
<td>113</td>
</tr>
<tr>
<td>14 ACS Example</td>
<td>115</td>
</tr>
<tr>
<td>14.1 The MAP Mission</td>
<td>115</td>
</tr>
<tr>
<td>14.2 Control Modes</td>
<td>116</td>
</tr>
<tr>
<td>14.3 Sensing and Actuation</td>
<td>116</td>
</tr>
<tr>
<td>14.4 Control System Design</td>
<td>116</td>
</tr>
<tr>
<td>14.5 Simulation Results</td>
<td>116</td>
</tr>
<tr>
<td>14.6 The TMapSim Script</td>
<td>117</td>
</tr>
<tr>
<td>14.7 Summary</td>
<td>124</td>
</tr>
</tbody>
</table>

Spacecraft Control Toolbox
### III  Attitude and Orbit Estimation  

#### 15  Introduction  
15.1 Overview  
15.2 Demos  

#### 16  General Estimation  
16.1 Introduction  
16.2 Fixed Gain Estimators  
16.3 Variable Gain Estimators  
16.4 Extended Kalman Filter  
16.5 Continuous Discrete Extended Kalman Filter  
16.6 Unscented Kalman Filter  

#### 17  Stellar  
17.1 Introduction  
17.2 Star Catalogs  
17.3 StarIdentification Behavior  
17.4 Centroiding  
17.5 Generating Star Data  
17.6 StarIDGUI  

#### 18  Unscented Kalman Filter  
18.1 Introduction  
18.2 Pressure Kalman Filter  

### IV  Orbit Propagation and Maneuvers  

#### 19  Orbit Module  
19.1 Overview  
19.2 Orbit Demos  

#### 20  Orbit Propagation  
20.1 Low Thrust Orbit Raising  
20.2 Reentry Due to Drag  
20.3 Topex Ephemerides  
20.4 Integration Tolerance  

#### 21  Orbit Maneuvers  
21.1 Orbit Maneuver Functions  
21.2 Launch and Early Orbit Operations  
21.3 Drag Correction Example  
21.4 Glideslopes  

#### 22  Orbit Propagator GUI  
22.1 The Orbit Propagator Tool  
22.2 Using the Graphical User Interface  
22.3 Plugin Functions  
22.4 Building a Plugin Function  
22.5 Reference Frames  
22.6 Sun  

---

*Spacecraft Control Toolbox*
# CONTENTS

22.7 Interfacing to STK ................................................................. 181  
22.8 Ground Coverage ................................................................. 181  
22.9 Orbit Propagation API ............................................................ 182  

23 Fuel Budgets ............................................................................. 185  
23.1 Fuel Budget ........................................................................... 185  

24 Orbit References ........................................................................ 191  
24.1 Orbits .................................................................................... 191  
24.2 Web Sites .............................................................................. 191  

V Subsystems Analysis .................................................................... 193  

25 Imaging .................................................................................... 195  
25.1 ImageMatching ...................................................................... 195  
25.2 ImageProcessing .................................................................... 195  

26 Link Module ................................................................................ 197  
26.1 Link Utilities .......................................................................... 197  
26.2 Optical .................................................................................. 198  
26.3 RF ......................................................................................... 199  
26.4 Downlink Analysis Example .................................................... 201  
26.5 LADAR .................................................................................. 202  
26.6 RADAR .................................................................................. 204  

27 Power ....................................................................................... 207  

28 Propulsion Module ..................................................................... 209  
28.1 Module Overview ................................................................... 209  
28.2 Solid Rocket Motor ................................................................. 211  
28.3 Blowdown Curve ................................................................... 211  
28.4 Electric propulsion ................................................................. 212  
28.5 Launch vehicle analysis .......................................................... 213  

29 Thermal Module ......................................................................... 215  
29.1 Module Overview ................................................................... 215  
29.2 Thermal Properties ................................................................. 215  
29.3 Thermal Analysis ................................................................... 217  

Appendices .................................................................................... 221  

A GOES Interface .......................................................................... 223  
A.1 Introduction ............................................................................ 223  
A.2 Loading GOES Data ............................................................... 223  
A.3 Getting GOES Longitude ........................................................ 224  
A.4 GOES functions ..................................................................... 224  
A.5 For More Information .............................................................. 224  
A.6 References .............................................................................. 227  

B Using Databases ........................................................................... 229
B.1 The Constant Database ................................................................. 229
B.2 Merging Constant Databases ...................................................... 230

C GUI Plug Ins ................................................................................. 233
  C.1 GUI PlugIn Demo ................................................................. 233
  C.2 Writing Your Own GUI Function ............................................ 234
CODE EXAMPLES

3.1 Earth radiation ................................................................. 20
3.2 Torquer and thruster comparison results ................................... 21
3.3 Antenna patch ................................................................ 23
3.4 Hall thruster ................................................................... 23
3.5 Solar flux from the sun ......................................................... 24
6.1 Simple patch with one triangle ................................................. 39
7.1 Rigid Body Attitude Simulation ............................................... 64
LIST OF FIGURES

2.1 MATLAB Help .................................................. 7
2.2 Toolbox Demos ................................................. 8
2.3 The file help GUI ............................................. 12
2.4 The demo GUI .................................................. 14
2.5 On-line Help ................................................... 15

3.1 Atmospheric density from AtmDens2 .......................... 18
3.2 Dual Spin Stability ............................................ 21
3.3 Elliptical orbit from RVFromKepler .......................... 22
3.4 A sine wave using Plot2D ................................... 25
3.5 Pivot mechanism .............................................. 26
3.6 Finder window ............................................... 28

4.1 ECI and LVLH Coordinate Frames ............................ 29
4.2 Control torques ............................................... 31
4.3 Control torque, ideal and with a 0.25 sec delay ............ 32

5.1 Frames A and B ............................................... 33
5.2 Selenographic to ECI frame .................................. 35
5.3 Areocentric frame ............................................ 36

6.1 Component geometry and frames ............................. 39
6.2 Body geometry and frames ................................... 40
6.3 Cylinder demo of geometry and frames. Hinge vectors are in black, rA is cyan and rB is magenta. The axes are coded X (blue), Y (green), and Z (red). .............. 41
6.4 Simple Sat .................................................. 43
6.5 CreateComponent color display ............................ 51
6.6 BuildCADModel Component View .......................... 52
6.7 BuildCADModel Body view ................................ 54
6.8 Vehicle tab with Show Vehicle pushed ..................... 54
6.9 3D View obtained by pushing Show Vehicle, left, and another sail model shown using Show Spacecraft in Orbit, right ........................................ 55
6.10 Subsystem Transparency and Exploded views ............. 55
6.11 2D View from clicking Show 2D Plan ..................... 56
6.12 Tie fighter as loaded from a dxf file ....................... 57
6.13 DrawSCPlanPlugIn showing the XYZSat model .......... 59
6.14 DrawSCPlanPlugIn showing the XYZSat model after a change to g ........................................ 60

7.1 Wire model .................................................. 65
7.2 Wire simulation: Results .................................... 68
7.3 Topological Tree ............................................. 69
7.4 Tree Simulation: One of several plots. ...................... 71
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Planet trajectories as generated by the built-in demo of SolarSysJPL</td>
<td>76</td>
</tr>
<tr>
<td>9.1</td>
<td>Plotting Tool GUI with Newly Loaded Data File</td>
<td>78</td>
</tr>
<tr>
<td>9.2</td>
<td>Diagram of the Plotting Tool GUI</td>
<td>79</td>
</tr>
<tr>
<td>9.3</td>
<td>Plotting Tool Generated Figure</td>
<td>82</td>
</tr>
<tr>
<td>9.4</td>
<td>Animation GUI Snap-Shot</td>
<td>85</td>
</tr>
<tr>
<td>10.1</td>
<td>Open the GUI</td>
<td>92</td>
</tr>
<tr>
<td>10.2</td>
<td>Entering the first error</td>
<td>93</td>
</tr>
<tr>
<td>10.3</td>
<td>Entering a second error</td>
<td>93</td>
</tr>
<tr>
<td>10.4</td>
<td>Save window</td>
<td>93</td>
</tr>
<tr>
<td>11.1</td>
<td>Block diagram</td>
<td>95</td>
</tr>
<tr>
<td>11.2</td>
<td>Step response</td>
<td>97</td>
</tr>
<tr>
<td>11.3</td>
<td>LQ GUI</td>
<td>97</td>
</tr>
<tr>
<td>11.4</td>
<td>Step response from the GUI</td>
<td>98</td>
</tr>
<tr>
<td>11.5</td>
<td>SISO inputs</td>
<td>98</td>
</tr>
<tr>
<td>11.6</td>
<td>MapIO</td>
<td>99</td>
</tr>
<tr>
<td>11.7</td>
<td>SISO step response</td>
<td>99</td>
</tr>
<tr>
<td>11.8</td>
<td>Eigenstructure design GUI</td>
<td>101</td>
</tr>
<tr>
<td>11.9</td>
<td>Step response with eigenstructure assignment</td>
<td>101</td>
</tr>
<tr>
<td>12.1</td>
<td>Model of 2U CubeSat</td>
<td>104</td>
</tr>
<tr>
<td>12.2</td>
<td>Orbit simulation timestep results, simple on the left with with disturbances on the right.</td>
<td>105</td>
</tr>
<tr>
<td>12.3</td>
<td>Orbit evolution for an initial separation of 10 meters</td>
<td>105</td>
</tr>
<tr>
<td>12.4</td>
<td>CubeSatSimulation example results</td>
<td>106</td>
</tr>
<tr>
<td>12.5</td>
<td>Observation time windows</td>
<td>107</td>
</tr>
<tr>
<td>12.6</td>
<td>RapidSwath built-in demo results</td>
<td>108</td>
</tr>
<tr>
<td>12.7</td>
<td>Playback demo</td>
<td>109</td>
</tr>
<tr>
<td>12.8</td>
<td>CAD model visualization</td>
<td>109</td>
</tr>
<tr>
<td>13.1</td>
<td>Solar pressure torque map for double-gimballed solar array</td>
<td>114</td>
</tr>
<tr>
<td>14.1</td>
<td>The Microwave Anisotropy Probe</td>
<td>115</td>
</tr>
<tr>
<td>14.2</td>
<td>Motion of the unit vector</td>
<td>123</td>
</tr>
<tr>
<td>14.3</td>
<td>Precession of the unit vector</td>
<td>123</td>
</tr>
<tr>
<td>16.1</td>
<td>Fixed gain Kalman filter</td>
<td>130</td>
</tr>
<tr>
<td>16.2</td>
<td>UD filter</td>
<td>131</td>
</tr>
<tr>
<td>16.3</td>
<td>Continuous discrete Kalman filter</td>
<td>132</td>
</tr>
<tr>
<td>16.4</td>
<td>Clock error covariance for an orbit Kalman Filter</td>
<td>133</td>
</tr>
<tr>
<td>17.1</td>
<td>LoadCatalog default</td>
<td>136</td>
</tr>
<tr>
<td>17.2</td>
<td>FOV determination</td>
<td>138</td>
</tr>
<tr>
<td>17.3</td>
<td>Pivot method</td>
<td>139</td>
</tr>
<tr>
<td>17.4</td>
<td>Centroiding demo</td>
<td>141</td>
</tr>
<tr>
<td>17.5</td>
<td>GUI with identified stars</td>
<td>142</td>
</tr>
<tr>
<td>17.6</td>
<td>Camera view and FK5 catalog</td>
<td>144</td>
</tr>
<tr>
<td>17.7</td>
<td>Hits and star snapshot</td>
<td>145</td>
</tr>
<tr>
<td>18.1</td>
<td>Pressure Kalman Filter</td>
<td>149</td>
</tr>
<tr>
<td>Figure Number</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>20.1</td>
<td>Comparison of an analytical low thrust model with the propagated solution.</td>
<td>156</td>
</tr>
<tr>
<td>20.2</td>
<td>Reentry Due to Drag</td>
<td>157</td>
</tr>
<tr>
<td>20.3</td>
<td>Topex Ephemeris Cycle 10</td>
<td>158</td>
</tr>
<tr>
<td>20.4</td>
<td>Difference Between the Topex Ephemeris and a Keplerian Orbit</td>
<td>159</td>
</tr>
<tr>
<td>20.5</td>
<td>Difference Between the Topex Ephemeris and a Propagated Spherical Earth Model</td>
<td>159</td>
</tr>
<tr>
<td>20.6</td>
<td>Topex Orbits in 3D</td>
<td>160</td>
</tr>
<tr>
<td>20.7</td>
<td>Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal Harmonics</td>
<td>160</td>
</tr>
<tr>
<td>20.8</td>
<td>Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal and 4 Tesseral Harmonics</td>
<td>161</td>
</tr>
<tr>
<td>20.9</td>
<td>Effect of Integration Tolerances on Orbit Propagation</td>
<td>162</td>
</tr>
<tr>
<td>21.1</td>
<td>Launch and early orbit operations analysis</td>
<td>165</td>
</tr>
<tr>
<td>21.2</td>
<td>Orbit propagation GUI and aero model inputs</td>
<td>167</td>
</tr>
<tr>
<td>21.3</td>
<td>Comparison of analytical drag with propagated</td>
<td>167</td>
</tr>
<tr>
<td>21.4</td>
<td>Orbital elements</td>
<td>168</td>
</tr>
<tr>
<td>21.5</td>
<td>Glideslope demo results</td>
<td>169</td>
</tr>
<tr>
<td>22.1</td>
<td>FEngine GUI</td>
<td>176</td>
</tr>
<tr>
<td>22.2</td>
<td>FAero GUI</td>
<td>176</td>
</tr>
<tr>
<td>22.3</td>
<td>FSolar GUI</td>
<td>177</td>
</tr>
<tr>
<td>22.4</td>
<td>FPlantNoise GUI</td>
<td>177</td>
</tr>
<tr>
<td>22.5</td>
<td>FAtmDensity GUI</td>
<td>177</td>
</tr>
<tr>
<td>25.1</td>
<td>CannyEdgeDetector demo</td>
<td>195</td>
</tr>
<tr>
<td>25.2</td>
<td>GaussianImageFilter demo</td>
<td>196</td>
</tr>
<tr>
<td>26.1</td>
<td>Shannon demo</td>
<td>199</td>
</tr>
<tr>
<td>26.2</td>
<td>Atmospheric Gas Loss</td>
<td>200</td>
</tr>
<tr>
<td>26.3</td>
<td>Sky temperature</td>
<td>200</td>
</tr>
<tr>
<td>26.4</td>
<td>Results of LinkAnalysisTool</td>
<td>201</td>
</tr>
<tr>
<td>26.5</td>
<td>Results of LADARTransmitPower</td>
<td>203</td>
</tr>
<tr>
<td>26.6</td>
<td>Results of RadarEquation</td>
<td>204</td>
</tr>
<tr>
<td>26.7</td>
<td>Results of RadarCrosssection</td>
<td>205</td>
</tr>
<tr>
<td>27.1</td>
<td>Band gap</td>
<td>207</td>
</tr>
<tr>
<td>27.2</td>
<td>Solar cell I-V curves</td>
<td>208</td>
</tr>
<tr>
<td>27.3</td>
<td>Solar cell efficiency</td>
<td>208</td>
</tr>
<tr>
<td>28.1</td>
<td>Solid motor firing</td>
<td>211</td>
</tr>
<tr>
<td>28.2</td>
<td>Blowdown curve</td>
<td>212</td>
</tr>
<tr>
<td>28.3</td>
<td>Optimal electric vehicle</td>
<td>212</td>
</tr>
<tr>
<td>28.4</td>
<td>Two-State-to-Orbit Simulation</td>
<td>213</td>
</tr>
<tr>
<td>29.1</td>
<td>Black body wavelength as a function of temperatures</td>
<td>216</td>
</tr>
<tr>
<td>29.2</td>
<td>AbsDa demo</td>
<td>216</td>
</tr>
<tr>
<td>29.3</td>
<td>Plancks Law</td>
<td>217</td>
</tr>
<tr>
<td>29.4</td>
<td>Thermal network</td>
<td>218</td>
</tr>
<tr>
<td>29.5</td>
<td>Thermal imaging demo</td>
<td>220</td>
</tr>
<tr>
<td>29.6</td>
<td>Thermal imaging demo temperature plot</td>
<td>220</td>
</tr>
<tr>
<td>A-1</td>
<td>Loading GOES data</td>
<td>225</td>
</tr>
<tr>
<td>A-2</td>
<td>GOES Magnetic Field Data</td>
<td>225</td>
</tr>
<tr>
<td>A-3</td>
<td>GOES Electrons</td>
<td>226</td>
</tr>
</tbody>
</table>
A-4  GOES Weather ............................................................. 226
B-1  Constant database ...................................................... 229
B-2  Searching for mass .................................................... 230
B-3  MergeConstantDB function ......................................... 231
C-1  PlugInDemo on starting .............................................. 233
C-2  PlugInDemo with the earth as center ............................ 234
C-3  Selecting planets in the elements plugin ......................... 235
C-4  PlugInDemo at the end of the simulation ......................... 235
C-5  PlugInDemo 3D Spacecraft Display ............................... 236
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>
<thead>
<tr>
<th>Module</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Coordinate transformations, math, control, CAD tools, time conversions, graphis and general utilities</td>
</tr>
<tr>
<td>CubeSat</td>
<td>CubeSat and nanosatellite modeling</td>
</tr>
<tr>
<td>Plotting</td>
<td>Plotting tools for complex simulations including animation</td>
</tr>
<tr>
<td>SC</td>
<td>Attitude dynamics, pointing budgets, basic orbit dynamics, environment, sample CAD models, ephemeris, sensor and actuator modeling.</td>
</tr>
<tr>
<td>SCPro</td>
<td>Additional high-fidelity models for environment, sensors, actuators.</td>
</tr>
<tr>
<td>AttitudeControl</td>
<td>In-depth attitude control system design examples, including the hypothetical geosynchronous satellite ComStar</td>
</tr>
<tr>
<td>Imaging</td>
<td>Image processing functions</td>
</tr>
<tr>
<td>Link</td>
<td>Basic RF and optical link analysis.</td>
</tr>
<tr>
<td>Orbit</td>
<td>Orbit mechanics, maneuver planning, fuel budgets, and high-fidelity simulation.</td>
</tr>
<tr>
<td>Power</td>
<td>Basic power modeling</td>
</tr>
<tr>
<td>Propulsion</td>
<td>Electric and chemical propulsion. Launch vehicle analysis.</td>
</tr>
<tr>
<td>Thermal</td>
<td>Basic thermal modeling</td>
</tr>
<tr>
<td>Formation Flying</td>
<td>Add-On  Formation flying control</td>
</tr>
<tr>
<td>SAAD</td>
<td>Add-On  Spin axis attitude determination</td>
</tr>
<tr>
<td>Solar Sail</td>
<td>Add-On  Solar sail control and mission analysis</td>
</tr>
</tbody>
</table>
The Spacecraft Control Toolbox core, *Attitude Control, CubeSat, Estimation, 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 Classroom Learning Edition consists of only the *Common, SC,* and *Plotting* directories. 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** 7.0 at a minimum is required to run all of the functions. Most of the functions will run on previous versions but we are no longer supporting them.

### 1.3 Installation

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

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

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

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

### 1.4 Getting Started

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

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

Another useful utility function you should try is **Finder**. This GUI allows you to search your entire path, or selected folders, for instances of any keyword. While **FileHelp** allows you to search headers for keywords, **Finder** allows you to search for instances of a function or variable in the code itself. This GUI is described in Section 3.4.1. 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 through our website and use our web forums to join discussions about the toolboxes.

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. Its location is depicted below.

Figure 2.1. MATLAB Help

<table>
<thead>
<tr>
<th>Search</th>
<th>Search Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>Search Results</td>
</tr>
<tr>
<td>Release Notes</td>
<td></td>
</tr>
<tr>
<td>Installation</td>
<td></td>
</tr>
<tr>
<td>MATLAB</td>
<td></td>
</tr>
<tr>
<td>Spacecraft Control Toolbox</td>
<td></td>
</tr>
<tr>
<td>What’s Included</td>
<td></td>
</tr>
<tr>
<td>Installation</td>
<td></td>
</tr>
<tr>
<td>New Features</td>
<td></td>
</tr>
<tr>
<td>Getting Started</td>
<td></td>
</tr>
<tr>
<td>Demos</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; Functions</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; AttitudeControl</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; AttitudeManeuver</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; FRSCGen</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; FRSMPlan</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; FRSPProp</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; FRSTarg</td>
<td></td>
</tr>
</tbody>
</table>

FRSCGen:

Executes the fast reorientation system slew maneuver. This is a combination of both the FRSProp and FRSTorque functions i.e. it computes the maneuver torque and updates the model at the same rate.

Form:

[qref00, modelrate, tfrs, xmodel, umnvr1, umnvr1, xf] = FRSCGen( axis0, af, bf, cf, df, xmodel, tbscale, qref00, umnvr1, xf, dt, nhalf, rate, iner, maxaccel, i )

Inputs

axis0 (3,1) The maneuver axis unit vector
af (n,n) The shaping filter plant matrix
bf (n,1) The shaping filter input matrix
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 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 them to view the output, or also open them for editing. 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 hierarchical structure used for browsing the demos.

**Figure 2.2.** Toolbox Demos

![Toolbox Demos](image)

### 2.2 Command Line Help

You can get help for any function by typing

```plaintext
>> help functionName
```

*Spacecraft Control Toolbox*
For example, if you type

```
>> help C2DZOH
```

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

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

Given
  \[ x = ax + bu \]

Find \( f \) and \( g \) where
  \[ x(k+1) = fx(k) + gu(k) \]

Form:
  \[ \{f, g\} = C2DZOH(a, b, T) \]
```

---

**Inputs**

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

---

**Outputs**

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

---

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

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

When you want more information about a folder of interest, remember that 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 Hardware

SC/Hardware

F
  FloatToWord - Convert floating point to word.

H
  HMagEl - Computes the magnetic field for a filament or loop magnetic element.
```

*Spacecraft Control Toolbox*
PivotMch - Models a pivot mechanism and converts stepping motor steps to pivot angle.

WordToFloat - Convert an integer word to a floating point number.

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

```plaintext
>> 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.
```

This type of 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.

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

Directories:
AStar
CAD
Classes
CommonData
ComponentModels
Control
Control1GUI
Coord
Database
Demos
Demos/AStar
Demos/CAD
Demos/Control
Demos/Database
Demos/General
Demos/Graphics
Demos/Help
Demos/Math
Demos/Plugins
Demos/Probability
Failure
General
Graphics
Help
Interface
Math
Plugins
```
The function `ver` lists the current version of all your installed toolboxes. Each SCT module that you have installed will be listed separately. For instance,

```matlab
>> ver
```

------------------------------------------
MATLAB Version 7.8.0.347 (R2009a)
MATLAB License Number: 61520
Operating System: Mac OS X Version: 10.6.2 Build: 10C540
Java VM Version: Java 1.6.0_15-b03-219 with Apple Inc. Java HotSpot(TM) Client VM mixed mode
------------------------------------------
MATLAB Version 7.8 (R2009a)
PSS Toolbox Folder AttitudeControl Version 8.0
PSS Toolbox Folder Common Version 8.0
PSS Toolbox Folder CubeSat Version 8.0
PSS Toolbox Folder Estimation Version 8.0
PSS Toolbox Folder FormationFlying Version 8.0
PSS Toolbox Folder Imaging Version 8.0
PSS Toolbox Folder Link Version 8.0
PSS Toolbox Folder Orbit Version 8.0
PSS Toolbox Folder Plotting Version 8.0
PSS Toolbox Folder Power Version 8.0
PSS Toolbox Folder Propulsion Version 8.0
PSS Toolbox Folder SC Version 8.0
PSS Toolbox Folder SCPro Version 8.0
PSS Toolbox Folder Sail Version 8.0
PSS Toolbox Folder Thermal Version 8.0

### 2.3 FileHelp

#### 2.3.1 Introduction

When you type

```
FileHelp
```

the FileHelp GUI appears, Figure 2.3 on the following 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.
**Figure 2.3.** The file help GUI
2.3.2 The List Pane

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

2.3.3 Edit Button

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

2.3.4 The Example Pane

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

2.3.5 Run Example Button

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

2.3.6 Save Example Button

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

2.3.7 Help Button

Opens the on-line help system.

2.3.8 Quit

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

2.4 Searching in File Help

2.4.1 Search File Names Button

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

2.4.2 Find All Button

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

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

2.4.4 Search String Edit Box

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

2.5 DemosPSS

If you type DemoPSS you will see the GUI in Figure 2.4. 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 2.4 shows the Common/Control folder in the core toolbox. The hierarchical menu shows the highest level folders.

**Figure 2.4.** The demo GUI

Your own demos will appear if they are put in any of the Demos folders. If you would like to look at, or edit, the script, push Show the Script.
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 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 2.5 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 and hitting the appropriate button. Restore List restores the list window to its previous configuration.

2.7 Technical Support

Contact support@psatellite.com for free email technical support. We are happy to add functions and demos for our customers when asked.
CHAPTER 3

BASIC FUNCTIONS

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

3.1 Introduction

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

This chapter will review some basic features built into the SCT functions and highlight some examples from the folder 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 233.

3.2 Function Features

3.2.1 Introduction

Functions have several features that are helpful to understand. Features that are available in the functions are listed in Table 3.1.

Table 3.1. Features in Spacecraft Control Toolbox functions

<table>
<thead>
<tr>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in demos</td>
</tr>
<tr>
<td>Default parameters</td>
</tr>
<tr>
<td>Built-in plotting</td>
</tr>
<tr>
<td>Error checking</td>
</tr>
<tr>
<td>Variable inputs</td>
</tr>
</tbody>
</table>

These are illustrated in the examples given below.
3.2.2 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 \texttt{AtmDens2} which generates the plot in Figure 3.1. This plot shows the atmospheric density as a function of altitude over the full range of the model. The inputs for the built-in demo are generally specified near the top of the function so it is easy to check for one by looking at the code. Plots are produced at the bottom of a function.

3.2.3 Default parameters

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

\[
\texttt{[]}\n\]

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, \texttt{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.

\[
\begin{bmatrix}
-0.9815 & -0.1914 & 0 \\
0.1914 & -0.9815 & 0 \\
0 & 0 & 1.0000
\end{bmatrix}
\]

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

\[
\begin{bmatrix}
\end{bmatrix}
\]
CHAPTER 3. BASIC FUNCTIONS

3.2. FUNCTION FEATURES

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

```
>> type EarthRot

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

%--
% Computes the earth greenwich matrix that transforms from ECI to earth fixed.
%--
% Form:
% [g, gMST, gAST] = EarthRot( T, eOfECalc )
%--
...

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

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

```

where we have excerpted the code creating the default input.

3.2.4 Built-in plotting

Many of the functions in the toolbox will plot the results if there are no output arguments. In many cases, you do not need any input arguments to get useful plots due to the built in defaults, but you can also generate plots with your own inputs. Calling for example `EarthRad` by itself will generate a plot of the absorbed flux per unit area as shown in Example 3.1 on the 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.

3.2.5 Error checking

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

3.2.6 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 ]
```

Spacecraft Control Toolbox

19
3.3. Example Functions

Example 3.1 Earth radiation

EarthRad

or the data structure

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

The options are listed in the header.

3.3 Example Functions

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

3.3.1 Attitude

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

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

```
>> DSpnStab( 1, 100, 1000, 1000 )
```

The result is \( dSS = 0 \) which means the system is unstable! A stability plot is shown in Figure 3.2 on the facing page. One means stable. \( \text{DSpnStab} \) is a typical function in SCT. If no outputs are specified it generates a plot.

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

3.3.2 Basic Orbit

\( \text{RVFromKepler} \) uses Kepler’s equation to propagate the position and velocity vectors. The output of the demo is shown in Figure 3.3 on page 22.
Figure 3.2. Dual Spin Stability

Example 3.2 Torquer and thruster comparison results

MagTComp( 120, 0.8, 75e-9 )

Spacecraft Control Toolbox
3.3. EXAMPLE FUNCTIONS

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

**Figure 3.3.** Elliptical orbit from RVFromKepler

![Figure 3.3](image)

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

r =

3200
0
0

v =

0
13.83596536017765
2.80468902945837

3.3.3 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

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

q =

0.5
0.5
0.5
-0.5

ans =

0
-1
0

3.3.4 CAD

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

A Hall thruster can be drawn with HallThrusterModel as shown in Example 3.4 on the facing page.
Example 3.3 Antenna patch

Example 3.4 Hall thruster
3.3.5 Control

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

\begin{verbatim}
>> C2DZOH([0 1;0 0], [0;1], 0.5 )
ans =
    1  0.5
    0  1
\end{verbatim}

3.3.6 Dynamics

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

\begin{verbatim}
>> [a, b, c, d] = MBModel( diag([10000 2000 10000]),[0;400;0],[0;7.291e-5;0]);
>> eig(a)
\end{verbatim}

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.

\begin{verbatim}
ans =
     0 + 0.00007291000000i
     0 - 0.00007291000000i
     0 + 0.03994167200000i
     0 - 0.03994167200000i
     0
     0
\end{verbatim}

3.3.7 Environs

Models are available to compute solar flux, planetary radiation, magnetic fields and atmospheric properties. \texttt{SolarFlx} computes flux as a function of astronomical units from the sun. The function’s output is shown in Example 3.5.

Example 3.5 Solar flux from the sun

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{solar_flux.png}
\caption{Solar flux from the sun}
\end{figure}
3.3.8 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,

```matlab
>> [u, r] = SunV1( JD2000, [7000;0;0] )
```

```
u =
  0.18006
 -0.90249
 -0.39127
r =
  1.4729e+08
```

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

3.3.9 Graphics

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

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

**Figure 3.4.** A sine wave using `Plot2D`

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

3.3.10 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
3.3.11 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.

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

```
jD =
    2.451251504154273e+06
ans =
[1999 3 14 0 0.005 0.0589292138814]
```
3.4 GUI Tools in the Toolbox

Table 3.2 summarizes the GUIs available in the toolbox.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnimationGUI</td>
<td>Animate 3D variables from the PlottingTool</td>
<td>Section 9.8 on page 84</td>
</tr>
<tr>
<td>Build3DSC</td>
<td>Old CAD viewer function</td>
<td>Chapter 6 on page 37</td>
</tr>
<tr>
<td>BuildCADModel</td>
<td>View results of a CAD building script</td>
<td>Section B.1 on page 229</td>
</tr>
<tr>
<td>Constant</td>
<td>Search for and output the value of a constant</td>
<td>Section 11 on page 95</td>
</tr>
<tr>
<td>ControlDesignGUI</td>
<td>Design control systems interactively</td>
<td>Section 2.5 on page 14</td>
</tr>
<tr>
<td>DemoPSS</td>
<td>Browse all the demos in the toolbox</td>
<td></td>
</tr>
<tr>
<td>DetectionFilterBuilder</td>
<td>Build detection filters interactively</td>
<td></td>
</tr>
<tr>
<td>Figui</td>
<td>Display a compact list of figure windows</td>
<td></td>
</tr>
<tr>
<td>FileHelp</td>
<td>View and search the file headers</td>
<td>Section 2.3 on page 11</td>
</tr>
<tr>
<td>GUIColors</td>
<td>Set and save interface colors</td>
<td></td>
</tr>
<tr>
<td>HelpSystem</td>
<td>Provides online help for other GUIs</td>
<td>Section 2.6 on page 15</td>
</tr>
<tr>
<td>PointingBudgetGUI</td>
<td>Create pointing budgets</td>
<td>Section 10.2 on page 92</td>
</tr>
<tr>
<td>PlottingTool</td>
<td>Plot and animate simulation data</td>
<td>Chapter 9 on page 77</td>
</tr>
<tr>
<td>PropagateOrbitPlugIn</td>
<td>Main GUI for propagating an orbit using ode113 and user-selected models</td>
<td>Chapter 22 on page 171</td>
</tr>
<tr>
<td>MergeConstantDB</td>
<td>Merge two databases into a new mat-file</td>
<td>Section B.2 on page 230</td>
</tr>
<tr>
<td>MessageQueue</td>
<td>Accumulates messages from any function that sends messages to the queue</td>
<td></td>
</tr>
<tr>
<td>RWASpec</td>
<td>Calculates RWA parameters from spec sheet parameters</td>
<td></td>
</tr>
<tr>
<td>StopGUI</td>
<td>GUI for stopping scripts. Has Pause, Stop, and Plot buttons</td>
<td>Section 17.6 on page 142</td>
</tr>
<tr>
<td>StarIDGUI</td>
<td>Perform star identification</td>
<td></td>
</tr>
<tr>
<td>ScDB</td>
<td>Output values of spacecraft constants</td>
<td></td>
</tr>
<tr>
<td>Telemetry</td>
<td>Generates a set of telemetry pages</td>
<td></td>
</tr>
<tr>
<td>TelemetryOffline</td>
<td>Plots telemetry files previously saved by Telemetry</td>
<td></td>
</tr>
<tr>
<td>TimeGUI</td>
<td>Displays the time remaining and estimated completion of a simulation</td>
<td></td>
</tr>
</tbody>
</table>

3.4.1 Finder GUI

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

The Finder function has options for searching the entire path or a selected directory. The subfolders of a higher-level directory can be included or not. The Pick button brings up a file selection dialog where you can navigate to your desired directory. The search can be case sensitive and you can select whole word matching. You can search on just file help comments, or include or exclude them. For example, you can find all functions and demos that actually use the function PIDMIMO by searching with comments excluded. Once your search results are displayed in the Results list, you can open any file by clicking the Edit button.
**Figure 3.6.** Finder window
CHAPTER 4

SPACECRAFT CONTROL TUTORIAL

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

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

\[
\dot{q} = f(q, \omega) \quad (4.1)
\]
\[
T = I \dot{\omega} + \omega \times I \omega \quad (4.2)
\]

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

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

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

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

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

\[
T = I \dot{\omega} \quad (4.4)
\]
This is a simple double integrator \((1/s^2)\).

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

\[
\begin{align*}
x_{k+1} &= Ax_k + Bu_k \\ y_k &= Cx_k + Du_k
\end{align*} \tag{4.5}
\]

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

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

The function takes the following inputs:

**Table 4.1. PIDMIMO input parameters**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Size</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>inr</td>
<td>(n,n)</td>
<td>Inertia matrix</td>
<td></td>
</tr>
<tr>
<td>zeta</td>
<td>(n,1)</td>
<td>Vector of damping ratios</td>
<td>A large zeta will cause the system to be sluggish. General practice is to set zeta for critical damping.</td>
</tr>
<tr>
<td>omega</td>
<td>(n,1)</td>
<td>Vector of undamped natural frequencies</td>
<td>The bandwidth can’t 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.</td>
</tr>
<tr>
<td>tauInt</td>
<td>(n,1)</td>
<td>Vector of integrator time constants</td>
<td>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, tauInt is too fast, then the system is hard to stabilize with delays.</td>
</tr>
<tr>
<td>omegaR</td>
<td>(n,1)</td>
<td>Vector of derivative term roll-off frequencies</td>
<td>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.</td>
</tr>
<tr>
<td>tSamp</td>
<td>1</td>
<td>Sampling period</td>
<td>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. Delta ((x = x + Ax + Bu)) is better from a numeric point of view.</td>
</tr>
<tr>
<td>sType</td>
<td>:</td>
<td>State equation type (’Delta’ or ’Z’)</td>
<td></td>
</tr>
</tbody>
</table>

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

```matlab
>> [a, b, c, d, k] = PIDMIMO( 1, 0.7071, 0.1, 100, 1, 0.25, delta )
a =
 0 0
 0 -0.259968823501459
b =
 0.25
 0.25996882350146
0.000521750104942956
0.223137533733209
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

\[
\text{inertia} = \text{diag}([24.5\ 10\ 25])
\]

\[
\begin{bmatrix}
24.5 & 0 & 0 \\
0 & 10 & 0 \\
0 & 0 & 25
\end{bmatrix}
\]

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

\[
x = \text{RK4}(\ \text{FRB},\ x,\ t\text{Samp},\ t(k),\ \text{inertia},\ \text{invInertia},\ t\text{Ext}+t\text{Dist})
\]

where \text{RK4} and \text{FRB} are both SCT functions. \text{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

\[
\text{accel}(1) = c \cdot x\text{Roll} + d \cdot \text{angleError}(1);
\]

\[
x\text{Roll} = x\text{Roll} + a \cdot x\text{Roll} + b \cdot \text{angleError}(1);
\]

\[
\text{accel}(2) = c \cdot x\text{Pitch} + d \cdot \text{angleError}(2);
\]

\[
x\text{Pitch} = x\text{Pitch} + a \cdot x\text{Pitch} + b \cdot \text{angleError}(2);
\]

\[
\text{accel}(3) = c \cdot x\text{Yaw} + d \cdot \text{angleError}(3);
\]

\[
x\text{Yaw} = x\text{Yaw} + a \cdot x\text{Yaw} + b \cdot \text{angleError}(3);
\]

\[
t\text{Ext} = -\text{inertia} \cdot \text{accel};
\]

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.

\[
\text{qECIToLVLH} = \text{QLVLH}( \text{rECI}, \text{vECI} );
\]

\[
\text{qBodyToLVLH} = \text{QPose}( \text{QMult}( \text{QPose( qECIToBody) }, \text{qECIToLVLH} ) );
\]

\[
\text{angleError} = -2 \cdot \text{qBodyToLVLH}(2:4);
\]

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

**Figure 4.2.** Control torques
Next, we simulate the controller again, this time with a delay equal to one sampling time. We see in Figure 4.3 that our controller does fine with this delay.

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

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

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

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

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

Delays may come from sensors, other hardware, or software. Once identified, they can be accounted for in the controller gains. For example, PSS once had to upload new gains to a geosynchronous spacecraft when a 2 second delay was discovered in the earth sensor after launch!
This chapter shows you how to use Spacecraft Control Toolbox functions for coordinate transformations. There is a very extensive set of functions in the Common/Coord folder covering quaternions, Euler angles, transformation matrices, right ascension/declination, spherical coordinates, geodetic coordinates, and more. A few are discussed here. For more information on coordinate frames and representations, please see the Coordinate Systems and Kinematics chapters in the accompanying book *Spacecraft Attitude and Orbit Control*.

### 5.1 Transformation Matrices

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

**Figure 5.1.** Frames A and B

1. \( u_A = [1;0;0] \);
2. \( \theta = \pi/6 \);
3. \( m = \begin{bmatrix} \cos(\theta), \sin(\theta), 0; \\ -\sin(\theta), \cos(\theta), 0; \\ 0, 0, 1 \end{bmatrix} \);
4. \( u_B = m \times u_A \);

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

\[
\begin{align*}
\mathbf{u}_B &= \text{Eul2Mat}([0,0,\theta]) \times \mathbf{u}_A;
\end{align*}
\]

Use \text{Mat2Eul} to switch back to an Euler angle representation.
5.2 Quaternions

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

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

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

We could also get \(q\) by typing

```plaintext
def q = Eul2Q([0;0;\theta])
```

Much as you can concatenate coordinate transformation matrices, you can also multiply quaternions. If \(q_{A\rightarrow B}\) transforms from A to B and \(q_{B\rightarrow C}\) transforms from B to C then

```plaintext
def q_{A\rightarrow C} = QMult(q_{A\rightarrow B},q_{B\rightarrow C})
```

The transpose of a quaternion is just

```plaintext
def q_{C\rightarrow A} = QPose(q_{A\rightarrow C})
```

You can extract Euler angles by

```plaintext
def e_{A\rightarrow C} = Q2Eul(q_{A\rightarrow C})
```

or matrices by

```plaintext
def m_{A\rightarrow C} = Q2Mat(q_{A\rightarrow C})
```

If we convert the three Euler angles to a quaternion

```plaintext
def q_{I\rightarrow B} = Eul2Q(e);
```

\(q_{I\rightarrow B}\) 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

```plaintext
def q_{I\rightarrow B} = QPose(q_{BInitialToBFinal})
```

Given a vector of small angles \(e_{Small}\) that rotate from vectors from frame A to B, the transformation from A to B is

```plaintext
def u_B = (eye(3)-SkewSymm(eSmall))*u_A;
```

where

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

Note that \(\text{SkewSymm}(x) \ast y\) is the same as \(\text{Cross}(x,y)\).
5.3 Coordinate Frames

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

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

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

The relationship between the Selenographic and the ECI frame is computed by MoonRot and shown in Figure 5.2.

Figure 5.2. Selenographic to ECI frame

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

The areocentric frame computed by MarsRot is shown in Figure 5.3 on the next page. $\Omega$, $\omega$, and $i$ are the standard Euler rotations of the Mars vernal equinox, $\Upsilon$ with respect to the Earth’s vernal equinox, $\Upsilon$.
Figure 5.3. Areocentric frame
CHAPTER 6

USING CAD FUNCTIONS

6.1 Introduction

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

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

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

Table 6.1. CAD demos

<table>
<thead>
<tr>
<th>Demo</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuildAgileSat</td>
<td>Design the Small Agile Satellite.</td>
</tr>
<tr>
<td>BuildCADModelDemo</td>
<td>Build a solar sail model using BuildCADModel.</td>
</tr>
<tr>
<td>BuildCassini</td>
<td>Demonstrate how to create a CAD model from a .dxf file.</td>
</tr>
<tr>
<td>BuildComsat</td>
<td>Build up a picture of a geosynchronous communications satellite.</td>
</tr>
<tr>
<td>BuildCubeSat</td>
<td>A very simple spacecraft with thrusters and one sensor.</td>
</tr>
<tr>
<td>BuildCylinders</td>
<td>Demonstrate CAD geometry (vectors and rotations) using cylinders.</td>
</tr>
<tr>
<td>BuildDXF</td>
<td>Loads in .dxf data for Cassini and Huygens, and saves data to mat-files.</td>
</tr>
<tr>
<td>BuildGOES</td>
<td>Make a model of the GOES spacecraft</td>
</tr>
<tr>
<td>BuildMAP</td>
<td>Build a model of the Microwave Anisotropy Probe.</td>
</tr>
<tr>
<td>BuildNEP</td>
<td>Build a nuclear electric vehicle using BuildCADModel</td>
</tr>
<tr>
<td>BuildPlutoProbe</td>
<td>A very simple spacecraft with thrusters and one sensor.</td>
</tr>
<tr>
<td>BuildRecon</td>
<td>Draw a picture of the recon satellite.</td>
</tr>
<tr>
<td>BuildSC</td>
<td>Build up a picture of a geosynchronous communications satellite.</td>
</tr>
<tr>
<td>BuildSCModelForImaging</td>
<td>Creates a model for the thermal demo.</td>
</tr>
<tr>
<td>BuildShuttle</td>
<td>Build a shuttle model</td>
</tr>
<tr>
<td>BuildSimpleSat</td>
<td>A very simple spacecraft with one sensor.</td>
</tr>
<tr>
<td>BuildSolar Sail</td>
<td>Build a solar sail model using BuildCADModel.</td>
</tr>
<tr>
<td>BuildSubassembly</td>
<td>Inputs a .dxf file and saves as a subassembly.</td>
</tr>
<tr>
<td>BuildTSTOUpperStage</td>
<td>Upper stage</td>
</tr>
</tbody>
</table>
6.2 Building a Spacecraft Model Using BuildCADModel

6.2.1 Introduction

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuildTwoArraySC</td>
<td>Make a detailed demonstration spacecraft with two solar arrays.</td>
</tr>
<tr>
<td>BuildXYZSat</td>
<td>A very simple 'XYZ' spacecraft with six panels and axes.</td>
</tr>
<tr>
<td>CADColorDemo</td>
<td>Demonstrate CADColor</td>
</tr>
<tr>
<td>CassiniColorDemo</td>
<td>Demonstrate CADColor using the Cassini orbiter.</td>
</tr>
<tr>
<td>CassiniSim</td>
<td>Draws a picture of the Cassini orbiter, Huygens probe, Saturn, and Titan</td>
</tr>
<tr>
<td>DisturbanceFlatPlate</td>
<td>Disturbance analysis with two thin plates.</td>
</tr>
<tr>
<td>DisturbanceModel</td>
<td>A Disturbance model. All units are in meters.</td>
</tr>
<tr>
<td>DisturbancesDefaults</td>
<td>Calling the disturbance model using defaults and a simple CAD model.</td>
</tr>
<tr>
<td>SCDisturb</td>
<td>Demonstrate how to use the disturbance modelling function with CAD models.</td>
</tr>
<tr>
<td>ShuttleSim</td>
<td>Simulate the space shuttle model.</td>
</tr>
</tbody>
</table>

```matlab
>> BuildCADModel
User Actions
    'initialize'
    'add_units'
    'add_name'
    'set_aerodynamic_model'
    'set_aerodynamic_model_file'
    'add_reci'
    'add_veci'
    'add_recitobody'
    'add_qlvlh'
    'add_omega'
    'add_mass'
    'set_mass'
    'add_body'
    'add_component'
    'lock_mass_properties'
    'unlock_mass_properties'
    'update_mass_properties'
    'spacecraftplugin'
    'create_body_arrays'
    'add_subsystem'
    'add_subassembly'
    'add_panel'
```
6.2.2 Geometry

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

**Example 6.1 Simple patch with one triangle**

```matlab
>> p = patch([0 1 0],[0 0 1],'');
>> v = get(p,'vertices')
V =
0 0
1 0
0 1
>> f = get(p,'faces')
f =
1 2 3
```

We mentioned above that each model consists of bodies and components. Each component is defined in a local frame, for example the cylinder primitive in the SCT has height along the Z axis. The component center of mass is defined in this frame. In the case of a cylinder, this is measured from the base, \( cM = [0;0;h/2] \). The inertia matrix is defined about the center of mass. Each component can then be placed in a body using two vectors and a rotation in between them. The vectors are depicted in Figure 6.1 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

\[ r_{CM|body} = r_A + b \times (r_B + r_{CM|comp}) \]

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

\[ r = B_{core} \times (r_H + B_H \times (r_A + b \times r_B)) \]

where we see again that the hinge vector is defined in the previous body’s frame, rA is in the body frame and rB and cM are in the component frame. The body’s center of mass in the core frame is

\[ r_{CM|core} = r_H + B_H \times (r_{CM|body}) \]

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

**Figure 6.2.** Body geometry and frames

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

The vectors and rotations described here are explained further in the sections on bodies and components later in this chapter, Section 6.2.5 and Section 6.2.6.
Figure 6.3. Cylinder demo of geometry and frames. Hinge vectors are in black, rA is cyan and rB is magenta. The axes are coded X (blue), Y (green), and Z (red).
6.2.3 A Simple Example

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

Listing 6.1. Simplest spacecraft CAD script

```matlab
%% 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.
%----------------------------------------------------------------------
% Set mass properties
%-------------------
mass = struct('mass',20,'cM',[0;0;0],'inertia',eye(3));

% Initialize
BuildCADModel( 'initialize' );

% Add spacecraft properties
%---------------------------
BuildCADModel( 'setName', 'SimpleSpacecraft' );
BuildCADModel( 'setUnits', 'mks' );
BuildCADModel( 'setReciBody', [1;0;0;0] );
BuildCADModel( 'setReci', [7000;0;0] );

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

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

% 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', mass, 'body', 1 );
BuildCADModel( 'addComponent', m );

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

% Save the model
%----------------
g = BuildCADModel( 'getCadModel' );
d = FindDirectory('SCMat');
SaveStructure( g, fullfile(d,'SimpleSat') )
```

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

Figure 6.4. Simple Sat

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

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

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

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

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

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

```matlab
>> 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.

```matlab
>> g.component
ans =
1x2 struct array with fields:
  faceColor
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:

```matlab
>> g.component(2).mass
ans =
  mass: 1
cM: [3x1 double]
inertia: [3x3 double]
>> g.component(2).mass.cM
ans =
  4.000000000000000e-01
  4.000000000000000e-01
  4.000000000000000e-01
>> g.component(2).mass.inertia
ans =
  0 0 0
  0 0 0
  0 0 0
```
The default mass is 1 kg and the inertia is zero. 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. You will also find that the body properties and the vehicle properties are, in this case, the same.

```
>> g.mass
ans =
  mass: 21
  inertia: [3x3 double]
  cM: [3x1 double]
>> g.mass.inertia
ans =
   3.6381  -0.1524  -0.1524
  -0.1524   3.6381  -0.1524
  -0.1524  -0.1524   3.6381
>> g.mass.cM
ans =
   0.0190
   0.0190
   0.0190
```

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'
   'absorptivity', '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.

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

### 6.2.4 Setting Spacecraft Properties

The following code sets selected spacecraft properties:

```matlab
1. BuildCADModel ('initialize');
2. BuildCADModel ('setName', 'MySpacecraft');
3. BuildCADModel ('setUnits', 'mks');
4. BuildCADModel ('setRECI', rECI);
5. BuildCADModel ('setVECI', vECI);
6. BuildCADModel ('setqLVLH', qLVLH);
7. BuildCADModel ('setqECIToBody', q);
8. BuildCADModel ('setOmega', omega);
9. BuildCADModel ('setMass', mass);
```

An orbit can be set for use by the spacecraft visualization tools, for instance by using variables called \( r_{ECI} \) and \( v_{ECI} \) 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,

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

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

### 6.2.5 Creating Bodies

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

```matlab
% The core
%--------
m = CreateBody( 'make', 'name', 'Core' );
BuildCADModel('addBody', m );
% Rotating bodies
%----------------
axis = [1 2 3];
for k = 1:3
    m = CreateBody( 'make', 'name', ['Boom', num2str(k)], 'bHinge', struct( 'b', eye(3), 'axis', axis(k) ), 'previousBody', 1, 'rHinge', [0;0;0] );
end
BuildCADModel('addBody', m );
```
CHAPTER 6. CAD 6.2. BUILDING A SPACECRAFT MODEL USING BUILDCADMODEL

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

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

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

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

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

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

6.2.6 Creating Components

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

Recall the camera component from BuildSimpleSat,

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

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

>> CreateComponent('type')
----------------------------------------
Components
----------------------------------------
    'sphere'
    'box'
    'cylinder'
    'hollowCylinder'
    'hollowBox'
    'hollowSphere'
    'antenna'
    'ellipsoid'
    'surfaceOfRevolution'
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

```matlab
>> CreateComponent('inputs')
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of component</td>
</tr>
<tr>
<td>rB</td>
<td>Displacement of component before rotation</td>
</tr>
</tbody>
</table>
These are the critical parameters where you place your component in particular location, rotate it as needed, and assign it to a body. The matrix is the rotation required to transform from the component frame to the body frame. Note that \( r_A \) is the displacement of the rotated component, i.e. after rotation to the body frame, and \( r_B \) is a displacement before the rotation, so \( r_A \) is defined in the body frame and \( r_B \) is defined in the component frame. The center of mass of the component is also in the component frame. Recall from the section on geometry (6.2.2) that

\[
r_{CM|body} = r_A + b \ast (r_B + r_{CM|comp})
\]

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

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

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

```
>> CreateComponent('parameters','imu')
```

----------

Parameters for imu
----------

'rUpper'
'rLower'
'h'
'n'
'thermal.absorptivity'
'thermal.emissivity'
'thermal.heatGeneratedOn'
'thermal.heatGeneratedStandby'
'thermal.temperature'
'thermal.surfaceArea'
'thermal.uNormal'
'thermal.thermalMass'
'mass.mass'
'mass.cM'
'mass.inertia'
'power.powerStandby'
'power.powerOn'
'power.electricalConversionEfficiency'
'power.powerHeater'
'power.powerIsOn'
'optical.sigmaT'
'optical.sigmaA'
'optical.sigmaD'
'optical.sigmaS'
'infrared.sigmaRT'
'infrared.sigmaRA'
'infrared.sigmaRD'
'infrared.sigmaRS'
'aero.cD'
'rf.flux'
'rf.u'
The top properties listed are specific to the geometry of the IMU. The structures following are sets of properties needed for disturbance, power, and thermal analysis, for which defaults are obtained from GenericProperties as alluded to in Section 6.2.3. The graphics properties are used only for displaying the component in MATLAB. The names are mostly self-explanatory except for the optical and infrared coefficients; the T, A, D, and S stand for transmissivity, absorptivity, diffuse and specular reflectivity, respectively. These coefficients must sum to one. The aerodynamics parameter \( c_D \) is the coefficient of drag. To set any of these properties, you need only enter the field name, i.e. `absorptivity`, not `thermal.absorptivity`. If you do not enter properties, generic properties are used instead.

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

```matlab
    case 'imu'
        p = struct('scale',1,'bias',0,'randomWalk',0,'angleRandomWalk',0,...
                   'outputNoise1Sigma',0,'beta',0,'rateLimit',0,'scaleFactor',1,...
                   'lSB',1e-7,'countLimit',16777215);
        g = {'rUpper', 'rLower', 'h', 'n'};
        if geom
            if ~exist('n', 'var')
                n = 12;
            end
            m = CylinderModel( rUpper, rLower, h, n, m, computeInertia );
        end
```

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

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

Perhaps the shortest component function call outside of the demo is this:
>> m = CreateComponent('make','sphere','radius',5)  

```
  m =  
      class: 'sphere'
    thermal: [1x1 struct]
    mass: [1x1 struct]
    power: [1x1 struct]
    optical: [1x1 struct]
    infrared: [1x1 struct]
    aero: [1x1 struct]
     rF: [1x1 struct]
    magnetic: [1x1 struct]
    propulsion: [1x1 struct]
    graphics: [1x1 struct]
      inside: 1
    model: 'generic'
  manufacturer: 'none'
deviceinfo: {}  
v: [5x3 double]
f: [6x3 double]
```

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

You can determine what predefined colors are available by typing  
CreateComponent('colors'). You will see the window in Figure 6.5.

**Figure 6.5.** CreateComponent color display

![Color Popup](image)

6.2.7 Subsystems

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

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

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

6.2.8 Panels

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

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

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

6.2.9 Viewing the CAD Model

When your CAD script is done, execute the script. The GUI shown in Figure 6.6 will appear as in BuildSimpleSat in the beginning of this chapter. For this section we will view the results of BuildCADModelDemo.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.

**Figure 6.6. BuildCADModel Component View**

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

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

---

*Spacecraft Control Toolbox*  
52
### CHAPTER 6. CAD

#### 6.2. BUILDING A SPACECRAFT MODEL USING BUILDCADMODEL

Sail.mat.

<table>
<thead>
<tr>
<th>Subsystem Component</th>
<th>Mass (kg)</th>
<th>Power (W)</th>
<th>Heater Power (W)</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sail Subsystem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Panel +X/+Y</td>
<td>2.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Panel +X/-Y</td>
<td>2.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Panel -X/+Y</td>
<td>2.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Panel -X/-Y</td>
<td>2.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Drive +X/+Y</td>
<td>0.50</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Drive +X/-Y</td>
<td>0.50</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Drive -X/+Y</td>
<td>0.50</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Drive -X/-Y</td>
<td>0.50</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Sail +X</td>
<td>20.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Sail -X</td>
<td>20.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Sail +Y</td>
<td>20.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Sail -Y</td>
<td>20.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Bus Subsystem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RWA X</td>
<td>14.30</td>
<td>0.00</td>
<td>0.00</td>
<td>none hr60</td>
</tr>
<tr>
<td>RWA Y</td>
<td>14.30</td>
<td>0.00</td>
<td>0.00</td>
<td>none hr60</td>
</tr>
<tr>
<td>RWA Z</td>
<td>14.30</td>
<td>0.00</td>
<td>0.00</td>
<td>none hr60</td>
</tr>
<tr>
<td>South Array</td>
<td>0.03</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>North Array</td>
<td>0.03</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Core</td>
<td>10.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Radiator +Y</td>
<td>2.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Radiator -Y</td>
<td>2.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Chassis</td>
<td>3.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Battery</td>
<td>12.00</td>
<td>0.00</td>
<td>0.00</td>
<td>none generic</td>
</tr>
<tr>
<td>Total</td>
<td>161.95</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

If you push the Body tab you get the body view with body level data, shown in Figure 6.7 on the following page. The list of components associated with the body is given in a popup menu on the left. In the Vehicle view, if you push the Show Vehicle button you get a 3D window with the spacecraft and a list of components organized by body, see Figure 6.8 on the next page and Figure 6.9 on page 55.

The radio buttons in the Vehicle tab select the mode. If All is selected all components are shown. If Add is selected, no components are shown and you add them by clicking on the component in the list. If Subtract is selected all components are shown and you remove components by clicking on the component in the list. Within the 3D view, you can highlight certain subsystems. The buttons on the left of the window toggle each subsystem’s, or the whole spacecraft’s, transparency. See Figure 6.10 on page 55, 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:

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

An example view of another sail model in orbit is shown on the right in Figure 6.9 on page 55. The quaternions set here are used only for display purposes.

The Show 2D Plan button shows a 2D picture with 2D views of all of the components. The components are scaled to fit inside the same size rectangle.
6.2 BUILDING A SPACECRAFT MODEL USING BUILDCADMODEL

**Figure 6.7.** BuildCADModel Body view

![BuildCADModel Body view](image1)

**Figure 6.8.** Vehicle tab with Show Vehicle pushed

![Vehicle tab with Show Vehicle pushed](image2)

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

Figure 6.10. Subsystem Transparency and Exploded views
6.2. BUILDING A SPACECRAFT MODEL USING BUILDCADMODEL

6.2.10 Importing Models

Another model can be imported if it has previously been saved as a .mat file. This uses the ‘add subassembly’ action of BuildCADModel.

```matlab
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,

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

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

```matlab
>> 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]
```

Figure 6.11. 2D View from clicking Show 2D Plan
CHAPTER 6. CAD

6.2. BUILDING A SPACECRAFT MODEL USING BUILDCADMODEL

```plaintext
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 and are not used in disturbance modeling.

Figure 6.12. Tie fighter as loaded from a dxf file

![Tie fighter as loaded from a dxf file](image)

6.2.11 Exporting

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

```plaintext
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
sigmam 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

\[
g = \text{BuildCADModel}('get\_cad\_model');
\]

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

\[
\text{SaveStructure}(g, 'MyCADFileName');
\]

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

\[
g = \text{load}('MyCADFileName.mat');
\]

### 6.3 Visualizing your CAD Model

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

- \text{DrawSC}
- \text{DrawSCPlanPlugIn}
- \text{DrawSCPlugIn}

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

\[
g = \text{load}('XYZSat.mat');
\text{tag} = \text{DrawSCPlanPlugIn}('initialize', g);
\]
Figure 6.13. DrawSCPlanPlugIn showing the XYZSat model

See the figure in Figure 6.13 on the facing page. This is the same view that was obtained using the Show Vehicle button in the CAD model GUI. You can create a figure like this inside a script, then update it iteratively in a loop using the call

```matlab
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.

```matlab
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

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

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

```matlab
>> g.component(1)
ans =
```
We will rewrite the `faceColor` field. Set the field to any RGB triple, the update the plug in. For example,

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

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

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 also that the transpose of the inertial-to-body quaternion is passed to the first body (`g(1).body(1).bHinge.q`), as the `bHinge` field specifies the rotation to the previous frame.

**Listing 6.2.** Shuttle simulation visualizing the opening of the bay doors during an orbit simulation

```matlab
%% Simulate the space shuttle model.
% See also DrawSCPlugIn and BuildShuttle., QLVLH, QPose, StopGUI, RK4,
% JD2000, E12RV
% Copyright 1999 Princeton Satellite Systems, Inc. All rights reserved.
% Constants
---
degToRad = pi/180;

%% Global for the time GUI
Figure 6.14. DrawSCPlanPlugIn showing the XYZSat model after a change to g

```matlab
%------------------------
g = load('ShuttleModel');
%------------------------
global simulationAction
simulationAction = ' ';

%% 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(1).body(1).bHinge.q = QPose(QLVLH(x1(1:3), x1(4:6)));
g(1).rECI = x1(1:3);
g(1).qLVLH = QLVLH(x1(1:3), x1(4:6));
g(1).name = 'Space_Shuttle_Orbiter';
tag3DWindow = DrawSCPlugIn('initialize', g, [], [], 'Earth', jD);

%% Generate the two orbits using numerical integration
%----------------------------------------------------
doorAngle = 0;
rMSAngle = 0;
StopGUI('Space_Shuttle');
rMSAxis = [3 2 2 2];
while t<120
    % Transformation matrices
    %------------------------
g(1).body(1).bHinge.q = QPose(QLVLH(x1(1:3), x1(4:6)));
g(1).rECI = x1(1:3);
```
6.3. VISUALIZING YOUR CAD MODEL

```matlab
function disp(around = 0)
g(1).QLVLH = QLVLH(x1(1:3), x1(4:6));
if( doorAngle > -139 )
doorAngle = doorAngle - 1;
elseif( rMSAngle < 45 )
rMSAngle = rMSAngle + 1;
end
g(1).body(2).bHinge = struct( 'angle', doorAngle*pi/180, 'axis', 1 );
g(1).body(3).bHinge = struct( 'angle', -doorAngle*pi/180, 'axis', 1 );
for j = 1:5
g(1).body(j+3).bHinge = struct( 'angle', -rMSAngle*pi/180, 'axis', rMSAxis(j) );
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;
end
```

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

### 7.1 Dynamics Models

Table 7.1 lists the attitude dynamics models and their capabilities. The functions are located in the Dynamics folder of SC.

**Table 7.1. Attitude Simulation Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Support Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASim</td>
<td>Attitude simulation of a rigid body.</td>
<td></td>
</tr>
<tr>
<td>DIModel</td>
<td>Double integrator model.</td>
<td></td>
</tr>
<tr>
<td>FAKM</td>
<td>Solid rocket simulation.</td>
<td></td>
</tr>
<tr>
<td>FDC</td>
<td>Constrained equations in the form $M\dot{x} = F$.</td>
<td></td>
</tr>
<tr>
<td>FGs</td>
<td>Gyrostat dynamics with quaternion kinematics.</td>
<td></td>
</tr>
<tr>
<td>GSModel</td>
<td>Gyrostat dynamics.</td>
<td></td>
</tr>
<tr>
<td>FPivot</td>
<td>Spacecraft with pivoted momentum wheel and quaternion kinematics.</td>
<td></td>
</tr>
<tr>
<td>SCwPivot</td>
<td>Spacecraft with pivoted momentum wheel.</td>
<td></td>
</tr>
<tr>
<td>FRB</td>
<td>Rigid body with quaternion kinematics.</td>
<td></td>
</tr>
<tr>
<td>RBModel</td>
<td>Rigid body.</td>
<td></td>
</tr>
<tr>
<td>FRBWithMag</td>
<td>Rigid body RHS with magnetic torque computed internally.</td>
<td></td>
</tr>
<tr>
<td>FTB</td>
<td>Two body with quaternion kinematics.</td>
<td></td>
</tr>
<tr>
<td>TBModel</td>
<td>Two body.</td>
<td></td>
</tr>
<tr>
<td>FXModel</td>
<td>Flexible spacecraft model.</td>
<td></td>
</tr>
<tr>
<td>MBModel</td>
<td>Momentum bias spacecraft (dynamics only).</td>
<td></td>
</tr>
<tr>
<td>Tree</td>
<td>Topological tree.</td>
<td>TreeAdd TreeH TreeInit TreePlot TreePrnt</td>
</tr>
<tr>
<td>VIModel</td>
<td>Body with time varying inertia.</td>
<td></td>
</tr>
<tr>
<td>WireFRB</td>
<td>Deployable wire model</td>
<td>WireDMch WireH WireInit</td>
</tr>
</tbody>
</table>
Some models are represented with two functions, one that has only the dynamics (such as RBModel) and the other which appends quaternion kinematics to the equations of motion, making a complete attitude right-hand-side for integration. You can get linearized models of those that don’t directly output state space equations by using the Jacobian function.

A simple rigid body attitude simulation can be embodied in the function,

\[
\text{xDot} = \text{ASim}(\text{x}, \text{t}, \text{inertia}, \text{torque});
\]

which contains the following code,

\[
\text{xDot} = [\text{QIToBDot}(\text{x}(1:4), \text{x}(5:7)); \text{RBModel}(\text{x}(5:7), \text{inertia}, \text{torque})];
\]

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 \(\text{ASim}\). This function is used in Example 7.1. Initializing \(x\text{Plot}\) in the script speeds up the script considerably. Notice the use of brackets in \(x\) to make the script easier to read.

**Example 7.1 Rigid Body Attitude Simulation**

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

**7.2 Wire Model**

The wire model simulates a spacecraft with deployable wires as is illustrated in Figure 7.1 on the facing page. Two models are included. WireFRB includes the extensional dynamics, modeling the stretching of the wire. The second, WireC, applies kinematic constraints to control the length between wire nodes. The former can introduce very high frequency dynamics into the simulation but is useful if you expect the wire to stretch. The latter is useful for most...
scientific satellites with copper or aluminum wires. Either model can simulate deployment and both account for system center-of-mass motion. A spacecraft can have any number of wires.

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

### Listing 7.1. Wire Simulation: Header

```matlab
% 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;
```

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

### Listing 7.2. Wire Simulation: Initialization

```matlab
% Simulation parameters
 %
 tSim    = 120.0;
 dT      = 0.125;
 nSim    = tSim/dT;
 nPlot   = min([tSim/dT 200]);
 nPMax   = floor(nSim/nPlot);
 gGOn    = false;
 kConst  = true;
 % Print the time to go message every nTTGo steps
 %
 nTTGo   = 1000;
```

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

### Listing 7.3. Wire Simulation: Spacecraft properties

```matlab
% Spacecraft properties
```
% ----------------------
mass = 800; % kg
r0 = [0;0;0];
inertia = [104 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

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

Listing 7.4. Wire Simulation: Initialization of the wire model

% The wire model. Each column is one wire
%----------------------------------------

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

Listing 7.5. Wire Simulation: Kinematic constraints

% If using the kinematic constraints
%-------------------------------

Initialize the plotting arrays.

Listing 7.6. Wire Simulation: Initialization of plotting

% Plotting arrays
%-----------------

Spacecraft Control Toolbox 66
The simulation loop is next. The first part shows the plotting functions.

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

```matlab
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
end
```

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

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

```matlab
% 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
```

The following listing shows the plotting code.

**Listing 7.9. Wire Simulation: Plotting**

```matlab
% Output
```
7.3 Tree Model

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

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.
CHAPTER 7. ATTITUDE SIMULATIONS

7.3. TREE MODEL

Figure 7.3. Topological Tree

Listing 7.10. Tree Simulation: Initialization

```
%------------------------------------------------------------------------
% 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]);
```

Listing 7.11. Tree Simulation: Adding bodies.

```
% 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' );
```

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

Tree Simulation: A hinge acceleration.

Spacecraft Control Toolbox

69
% 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, tToGoMem ] = TimeGUI( nSim, 0, tToGoMem, 0, dTSim, 'Tree_Sim' );

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


% 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 = 'p';
    case 'stop'
        return;
    case 'plot'
        break;
end
end

Plot the results.

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

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

An example plot is shown in Figure 7.4.

**Figure 7.4.** Tree Simulation: One of several plots.
CHAPTER 8

PLANETARY AND SUN EPHEMERIS

8.1 Overview

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

8.2 Almanac functions

The toolbox has a variety of almanac functions for obtaining the ephemeris of the planets\(^1\), moon, and sun. For example, the following functions are available in Ephem:

- MoonV1\(^2\)
- MoonV2\(^3\)
- PlanetPosition
- SolarSys
- SunV1\(^4\)
- SunV2\(^5\)

\(^1\)Explanatory Supplement to the Astronomical Almanac (1992.) Table 5.8.1. p. 316.
\(^5\)Montenbruck and Pfleger, p. 36.
8.3 JPL Ephemeris

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

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

8.3.1 Creating and managing binary files of the JPL ephemerides

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

```bash
$ ./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`.

```bash
$ ./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
```

Spacecraft Control Toolbox 74
CHAPTER 8. EPHEMERIS

8.3. JPL EPHEMERIS

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.

8.3.2 The InterpolateState function

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

\[ [X, GM] = \text{InterpolateState}( \text{Target}, \text{Time}, \text{fileName} ) \]

The numbering convention for the Target bodies is given in table Table 8.1 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.

8.3.3 JPL Ephemeris Demos

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

State for Mercury on Jan 1, 2001
2.454786013744712e+07
-5.399651407895338e+07
8.3. JPL EPHEMERIS

Table 8.1. Target Numbering Convention

<table>
<thead>
<tr>
<th>Number</th>
<th>Bodies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mercury</td>
</tr>
<tr>
<td>2</td>
<td>Venus</td>
</tr>
<tr>
<td>3</td>
<td>Earth-Moon Barycenter</td>
</tr>
<tr>
<td>4</td>
<td>Mars</td>
</tr>
<tr>
<td>5</td>
<td>Jupiter</td>
</tr>
<tr>
<td>6</td>
<td>Saturn</td>
</tr>
<tr>
<td>7</td>
<td>Uranus</td>
</tr>
<tr>
<td>8</td>
<td>Neptune</td>
</tr>
<tr>
<td>9</td>
<td>Pluto</td>
</tr>
<tr>
<td>10</td>
<td>Geocentric Moon</td>
</tr>
<tr>
<td>11</td>
<td>Sun</td>
</tr>
</tbody>
</table>

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

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

Figure 8.1. Planet trajectories as generated by the built-in demo of SolarSysJPL

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

9.1 Introduction

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

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

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

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

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

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

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

Directories:
Demos
Demos/GUI
Derivation Functions
GUI
Help
Parametric
Sim Results
```
The remainder of this chapter describes how to use the Plotting Tool, and discusses the various features that you may find useful.

### 9.2 Getting Started

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

```matlab
>> PlottingTool
```

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

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

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

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

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

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

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

## 9.3 GUI Layout and Features

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

**Figure 9.2** Diagram of the Plotting Tool GUI
9.3 GUI LAYOUT AND FEATURES

9.3.1 Data File Pull-Down Menu

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

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

9.3.2 Object Pull-Down Menu

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

9.3.3 Variable List

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

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

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

The grouping, exclusion, and derivation of variables may be conducted using separate GUI’s, as described in Sections 9.7, 9.5, and 9.6.

9.3.4 Variable Information & Time-History

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

9.3.5 Figure Editing

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

9.3.6 3-D Variable Animation

A center portion of the GUI is used for setting up the animation of 3-dimensional variables. Animation is discussed in detail in Section 9.8 on page 84.
5.7 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.

5.8 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.

5.9 Shortcuts

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

<table>
<thead>
<tr>
<th>Action</th>
<th>Shortcut (PC)</th>
<th>Shortcut (Mac)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Data File</td>
<td>CTRL + L</td>
<td>CMD + L</td>
</tr>
<tr>
<td>Close Data File</td>
<td>CTRL + X</td>
<td>CMD + X</td>
</tr>
<tr>
<td>Apply Template</td>
<td>CTRL + A</td>
<td>CMD + A</td>
</tr>
<tr>
<td>Save Template</td>
<td>CTRL + T</td>
<td>CMD + T</td>
</tr>
<tr>
<td>Remove Template</td>
<td>CTRL + R</td>
<td>CMD + R</td>
</tr>
<tr>
<td>Export Data to Workspace</td>
<td>CTRL + E</td>
<td>CMD + E</td>
</tr>
<tr>
<td>Derive Data</td>
<td>CTRL + D</td>
<td>CMD + D</td>
</tr>
<tr>
<td>Quit</td>
<td>CTRL + Q</td>
<td>CMD + Q</td>
</tr>
</tbody>
</table>

5.4 Creating Figures

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

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

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

Figure 9.3 provides an example of the type of figures that can be quickly generated with the Plotting Tool. It shows the relative position and velocity of three spacecraft. The \( x \), \( y \), and \( z \) elements are separated by row, the \( r_{\text{Hills}} \) and \( v_{\text{Hills}} \) variables separated by column, and the satellites distinguished by linestyle. The figure could be easily generated again in several different formats. For each figure created, the title, variables, satellites and all plotting characteristics are stored in the template.

**Figure 9.3. Plotting Tool Generated Figure**

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

### 9.5 Grouping Variables

It is often useful to work with a vector of data rather than the individual elements. The plotting tool allows you to group individual variables together.
By pressing the Grouping button at the bottom of the Plotting Tool, a new Group Variables GUI will open up. This GUI may also be opened from the Actions menu. Within the Group Variables GUI, you may press the New Group button to create a new variable group. Once a group has been created, you may scroll through the checkbox list of simulation elements to assign variables to the group. Groups may always be edited, renamed, or deleted.

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

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

### 9.6 Excluding Variables

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

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

### 9.7 Deriving New Data

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

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

The steps for deriving new data are summarized as follows:

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

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

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

If changes have been made to your derivation settings, all of your derived outputs will automatically be recalculated upon exiting the derivation window. In addition, you may re-derive data at any time by pressing the Derive Now button on the bottom of the main Plotting Tool GUI.
When you exit from the Data Derivation GUI, the derivation functions are called for all outputs that you have created, and the derived data is stored in the PlottingTool GUI. If you have assigned the wrong number of inputs to a function, or have supplied inputs of an improper size, a dialogue box will be displayed indicating the error. If there is an error inside the function itself, the GUI can be closed once the error has been fixed.

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

### 9.8 Animation

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

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

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

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

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

### 9.9 Templates

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

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

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

You will now be prompted to select a template file. The first “Templates” folder found in your MATLAB path is used as the default folder in the browser window. Choose the FF_Matlab.mat template file that was included with the build. The template is now applied to the data file in the Plotting Tool.
9.10 Plotting Output from Custom Simulation Software

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

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

9.10.1 Reading a Custom Output File

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

Creating a Custom Output File

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

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

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

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

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

$$d = \text{ReadCustomFile}(\text{fId}, \text{any\_other\_necessary\_parameters});$$

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

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

Table 9.2. Fields of Output Data Structure $d$

<table>
<thead>
<tr>
<th>Field</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>{1 x nObjects} cell array with each cell containing a (nD x nT) matrix of raw data</td>
<td>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.</td>
</tr>
<tr>
<td>dataNames</td>
<td>{1 x nD} cell array of strings</td>
<td>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.</td>
</tr>
<tr>
<td>dataUnits</td>
<td>{1 x nD} cell array of strings</td>
<td>Each string describes the units of the corresponding data element. Optional.</td>
</tr>
<tr>
<td>excluded</td>
<td>{1 x n} cell array of strings</td>
<td>Each string is the dataName of an element that is not to be displayed in the variable list. Optional.</td>
</tr>
<tr>
<td>groups</td>
<td>() array of data structures with fields 'name' and 'elements'</td>
<td>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.</td>
</tr>
<tr>
<td>jDEpoch</td>
<td>scalar</td>
<td>Simulation start time as Julian date. Optional.</td>
</tr>
<tr>
<td>objNames</td>
<td>{1 x nObjects} cell array of strings</td>
<td>Each string is the name of the corresponding object. If this field doesn’t exist, the objects are named ‘Obj_1’, ‘Obj_2’, etc.</td>
</tr>
<tr>
<td>time</td>
<td>(1 x nT)</td>
<td>Row vector of elapsed simulation time in seconds. If this field doesn’t exist, it is assumed the time step is one second.</td>
</tr>
</tbody>
</table>

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:

```matlab
% 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:

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

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

### 9.10.2 Using the PSS Simulation Output Format

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

```matlab
>> 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.

<table>
<thead>
<tr>
<th>Listing 9.1. Example Format for Simulation Output Text File</th>
<th>SampleSimOutput.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Agile Satellite Simulation Output</td>
<td></td>
</tr>
<tr>
<td>Epoch 2.452994833e+06</td>
<td></td>
</tr>
<tr>
<td>nObjects 1</td>
<td></td>
</tr>
<tr>
<td>Target</td>
<td></td>
</tr>
<tr>
<td>// time (secs) [1] // rECI1 (km) [3] {x, y, z} // vECI1 (km/s) [3] {xDot, yDot, zDot} . . .</td>
<td></td>
</tr>
</tbody>
</table>

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

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

would cause the individual elements to be named ‘rECI_x’, ‘rECI_y’, ‘rECI_z’. The common ‘rECI_’ will not appear in any plots.

The preceding example would cause the variable list in the Plotting Tool to display ‘time’, ‘rECI’, and ‘vECI’ and exclude ‘rECI_x’, ‘rECI_y’, etc.

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

### 9.11 Summary

The Plotting Tool is a MATLAB GUI designed to display the results of raw simulation data in a useful way. The purpose of this tool is to provide users with the ability to quickly, conveniently, and consistently analyze the results of their simulation. If you find that you repeatedly view the same types of plots or derive the same types of data, then the Plotting Tool can help to automate these tasks by storing your settings in template files. The output files of both MATLAB and external simulations may be loaded into the GUI, while different templates are applied to change how the data is displayed. The result is a flexible, user-configurable method for post simulation analysis.
CHAPTER 10

POINTING BUDGETS

This chapter discusses how to generate a pointing budget.

10.1 Pointing Budget

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

Listing 10.1. PBudget Example

\[
\begin{align*}
\text{errors} & = [0.01, 0.02, 0.03; 0.02, 0.04, 0.01; 0.04, 0.03, 0.05; 0.05, 0.00, 0.06]; \text{ % degrees} \\
\text{categ} & = \text{str2mat('Bias')}; \\
\text{categ} & = \text{str2mat('Diurnal')}; \\
\text{desc} & = \text{str2mat('Thermal')}; \\
\text{desc} & = \text{str2mat('Misalignments')}; \\
aZ & = 0.0; \\
eL & = 0.0; \\
[\text{cep}, \alpha, s, t] & = \text{PBudget(errors, categ, desc, aZ, eL, 'MyBudget.txt')};
\end{align*}
\]

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

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

Listing 10.2. Pointing budget example output

\[
\begin{align*}
91
\end{align*}
\]
### 10.2 Pointing Budget GUI

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

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

**Figure 10.1. Open the GUI**

![GUI Image](image)

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

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

Figure 10.4 on the next page shows save file dialog box.
CHAPTER 10. POINTING BUDGETS

10.2. POINTING BUDGET GUI

**Figure 10.2.** Entering the first error

![Figure 10.2. Entering the first error](image)

**Figure 10.3.** Entering a second error

![Figure 10.3. Entering a second error](image)

**Figure 10.4.** Save window

![Figure 10.4. Save window](image)
This chapter shows how to design controllers using the ControlDesignGUI. The three major methodologies discussed are: Linear Quadratic, Eigenstructure Assignment, and Single-Input-Single-Output. This section focuses on how to use the Control Designer GUI.

11.1 Using the block diagram

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

**Figure 11.1. Block diagram**

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

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

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

This example is automated using LQFullState, shown below.

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

```matlab
function LQFullState.m

a = [0 1;0 0];
b = [0;1];
c = eye(2);
d = [0;0];
g = statespace( a, b, c, d, 'DoubleIntegrator',...
    {'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 );
```

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

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

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

Next click the control block to unhighlight it so that you can work with the whole system. You can now do a step response by pushing Step, Figure 11.4 on page 98.

11.3 Single-Input-Single-Output

Close and reopen the GUI and load in the double integrator plant. Next select the control block and the SISO tab. Add the input position and output force, by typing in the boxes and then clicking Add. The results will show up in the list on the left. Then add a transfer function TF at the bottom of the tab. Push the button to make position the
Figure 11.2. Step response

![Figure 11.2. Step response](image)

Figure 11.3. LQ GUI

![Figure 11.3. LQ GUI](image)
11.3. SINGLE-INPUT-SINGLE-OUTPUT  

CHAPTER 11. DESIGNING CONTROLLERS

Figure 11.4. Step response from the GUI

![States for LQ * Double Integrator](image)

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

Figure 11.5. SISO inputs

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

**Figure 11.6. MapIO**

![MapIO Diagram]

**Figure 11.7. SISO step response**

![SISO Step Response Graph]
11.4 Eigenstructure Assignment

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

Listing 11.2. Control configured vehicle example

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

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

% We really want to decouple gamma
%---------------------------------
w = [ 1 1 1 1 1; 1 1 1 1 1; 100 100 1 1 1];

% The design matrix.
%----------------------------------------
d = [eye(3),zeros(3,2); eye(3),zeros(3,2); 0 1 0 0 0; 0 0 1 0 0; 0 0 0 1];

% Rows in d per 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);
```

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

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

Push Create. Next push Step. You will see the plot in Figure 11.9 on the facing page.
Figure 11.8. Eigenstructure design GUI

Figure 11.9. Step response with eigenstructure assignment
CHAPTER 12

CUBE SAT

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 is below, as returned from MATLAB’s help function.

```
>> help CubeSat
PSS Toolbox Folder CubeSat
Version 7.2 11-Nov-2009

Directories:
Actuators
AttitudeControl
Communications
Demos
Demos/MissionPlanning
Demos/RelativeOrbit
Demos/Simulation
Help
MissionPlanning
Power
RelativeOrbit
Simulation
Thermal
Utilities
```

The Actuators folder includes a cold gas propulsion model. The AttitudeControl folder has functions for supporting reaction wheel design and simulations. Communications provides a function for link analysis over a satellite trajectory. The MissionPlanning folder contains a set of functions for attitude profiles, ground tracks and sensor observation planning. Power provides a function for modeling solar cells. RelativeOrbit provides functions for simulating two or more satellites in proximity. The Simulation folder provides aerodynamic and radiation pressure models and a simple attitude simulation right-hand-side. Thermal provides an isothermal model.

12.1 Modeling

The CubeSat model is generated by CubeSatModule, 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 for disturbance purposes. The function header is below and the resulting model is in Figure 12.1 on the next page. This model has 152 faces.

```
>> help CubeSatModel
```
Generate vertices and faces for a CubeSat model. You can draw the cubesat model using `patch`:

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

Type `CubeSatModel` for a demo of a 2U CubeSat.

---

Form:

```matlab
[v, f] = CubeSatModel( type, t )
```

---

**Inputs**

- `type` (1,:) 'nU' where n may be any number
- `t` (1,1) Panel thickness (mm)

---

**Outputs**

- `v` (:,3) Vertices
- `f` (:,3) Faces

---

Reference: CubeSat Design Specification (CDS) Revision 9

---

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

---

12.2 Simulation

Example simulations are in the `Demos/RelativeOrbit` and `Demos/Simulation` folders. The first has a formation flying demo, `FFSimDemo`. The second has a pair of two CubeSat orbit simulations and an attitude control simulation.
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

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

The default simulation length is 12 hours, and the simple sim results in 438 timesteps while the one with disturbances computes 733 steps; Figure 12.2 compares the time output of the two examples, where we can see that the simple simulation had mostly a constant step size. Figure 12.3 shows the typical orbit results.

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

![Orbit simulation timestep results](image1)

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

![Orbit evolution for an initial separation of 10 meters](image2)

`CubeSatSimulation` simulates the attitude dynamics of the CubeSat in addition to a point-mass orbit and the `Spacecraft Control Toolbox`.
12.2. SIMULATION

CHAPTER 12. CUBESAT

power subsystem. This simulation includes forces and torques from drag 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

\begin{verbatim}
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
\end{verbatim}

where the control, d.dipole, would be computed before the integration at every step for a fixed timestep (1 second).

Figure 12.4. CubeSatSimulation example results

![CubeSat Simulation Graphs]

Spacecraft Control Toolbox 106
12.3 Mission Planning

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

```
>> help MissionPlanning
CubeSat/MissionPlanning

A
AttitudeProfile - Build an attitude profile consisting of multiple overlapping modes.

D
DrawSpacecraftInOrbit - Draw the Earth, CAD model, orbit, sensor cone(s), and GPS constellation

O
ObservationTimeWindows - Generate a set of observation time windows for each lat/lon target
ObservationTimeWindowsPlot - Plot ground track and observation windows for a satellite and targets

P
PackageOrbitDataForPlayback - Package orbit data into a structure for use in PlaybackOrbitSim.
PlaybackOrbitSim - Play back an orbit simulation of multiple objects with sensor cones.

R
RapidSwath - Compute the intersection curve between a sensor cone and spherical planet
RepeatGroundTrack - Design a repeat ground-track orbit.
```

ObservationTimeWindows has a built-in demo which also demonstrates ObservationTimeWindowsPlot, as shown in Figure 12.5. There are two ground targets, one in South America and one in France (large green dots).

**Figure 12.5.** Observation time windows

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

RapidSwath also has a built-in demo. The demos uses an altitude of 2000 km and a field of view half-angle of 31 degrees. The function allows you to specify a pitch angle between the sensor boresight axis and the nadir axis, in this case 15 degrees. When called with no outputs, the function generates a 3D plot. In Figure 12.6 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.

Figure 12.6. RapidSwath built-in demo results

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

12.4 Visualization

The CubeSat Toolbox provides some useful tools to visualize orbits, field of view, lines of sight, and spacecraft orientations. Run the OrbitAndSensorConeAnimation.m mission planning demo to see how to generate simulated orbits, compute sensor cone geometry, and package the data for playback using PackageOrbitDataForPlayback and PlaybackOrbitSim. The playback function loads two orbits into the AnimationGUI, which provides VCR like controls for playing the simulation forward and backward at different speeds. Set the background color to black and point the camera at a spacecraft, then use the camera controls to move in/out, zoom in/out, and rotate the camera around the spacecraft within a local coordinate frame. The screenshot in Figure 12.7 on the facing page 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.

The CAD model of the spacecraft may also be viewed in its orbit, along with a sensor cone and lines of sight to all of the visible GPS satellites. Use DrawSpacecraftInOrbit.m to generate this view. An example is shown in Figure 12.8 on the next page. The image on the left shows the spacecraft orbit, its sensor cone projected on the Earth, the surrounding GPS satellites, and lines of sight to the visible GPS satellites. The image on the right is a zoomed-in view, where the spacecraft CAD model may be clearly seen.
Figure 12.7. Playback demo

Figure 12.8. CAD model visualization
PART II

ATTITUDE CONTROL DESIGN EXAMPLES
ATTITUDE CONTROL MODULE

13.1 Module Overview

The attitude control module (AttitudeControl) provides a set of attitude control system design examples. These design examples are implemented as demonstration scripts. One, for the Microwave Anisotropy Satellite, MAP, is explained in detail in the next chapter.

13.2 Design Examples

13.2.1 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 TMAPSim.

13.2.2 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 TSNPSim.m.

13.2.3 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 TRYDP.m.

13.2.4 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 Demos/Comstar 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

### 13.2.5 Momentum Unloading

This design uses solar pressure on a double-gimballed solar array for two-axis momentum control. The script, `TMMC.m`, maps out the torque produced over a range of gimbal angles before performing the simulation. See Figure 13.1.

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

![Solar pressure torque map](image)

### 13.2.6 Magnetic Control

The module has both two-axis and three-axis magnetic control examples, `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.
This section walks you through a complete design example, that of the Microwave Anisotropy Satellite normal and acquisition modes.

14.1 The MAP Mission

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

**Figure 14.1.** The Microwave Anisotropy Probe
14.2 Control Modes

The MAP spacecraft requires three control modes:

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

Optionally, a safe hold mode could be added.

14.3 Sensing and Actuation

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

14.4 Control System Design

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

14.5 Simulation Results

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

In this section we will walk through the TMapSim script. Once you understand how is was written you will be able to easily design and build your own control systems and simulations. Listing 14.1 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 14.1. File header**

```matlab
% Implements and simulates the MAP normal mode control. 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 script also generates an STK attitude file.

% Copyright (c) 1996-2001 Princeton Satellite Systems, Inc. All rights reserved.

% Clean up the workspace
close all

% Global for the time GUI
global simulationAction
simulationAction = 'J';

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

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 14.2. Sampling intervals**

```matlab
% The control sampling period and the simulation integration time step

% Number of sim steps

% Plot every nPMax steps

% Print the time to go message every nTTGo steps
```

Listing 14.3 on the following page 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.
14.6. THE TMAPSIM SCRIPT

Listing 14.3. Mass properties

```matlab
14.6. THE TMAPSIM SCRIPT

```

Listing 14.4. Tachometer loop design

```matlab
14.6. THE TMAPSIM SCRIPT

```

Listing 14.5. ACS design

```matlab
14.6. THE TMAPSIM SCRIPT

```

Listing 14.6. Rate loop

```matlab
14.6. THE TMAPSIM SCRIPT

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

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

**Listing 14.7. Initialize the control system**

```matlab
% 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 = 0.000;

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

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

**Listing 14.8. Set up the simulation**

```matlab
% 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
% ------------------
x = [Q;0;0];
spinRate;

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

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

**Spacecraft Control Toolbox**

119
This is the start of the simulation loop. The first function displays the status message in the command window.

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

```matlab
% Run the simulation
for k = 1:nSim
  % Display the status message
  [ ratioRealTime, tToGoMem ] = TimeGUI( nSim, k, tToGoMem, ratioRealTime, tSamp );
```

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

**Listing 14.10. Sensor models**

```matlab
% Sensors
% ---------------------------------------------
% RWA Tachometer
wTach = x(8:10);
% Gyros
wCore = x(5:7);
% Attitude
qIToB = x(1:4);
```

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

**Listing 14.11. Control system**

```matlab
% 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
tMM = QForm( qIToB, -kMM*hTotal );
```

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

**Listing 14.12.** Error computation

```matlab
% 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));
yawError = wCore(3) - spinRate;
```

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

**Listing 14.13.** Attitude loop

```matlab
% The attitude control loops
% --------------------------
tC(1) = -cRoll *xRoll - dRoll*rollError;
xRoll = xRoll + aRoll*xRoll + bRoll*rollError;

tC(2) = -cPitch *xPitch - dPitch*pitchError;
xPitch = xPitch + aPitch*xPitch + bPitch*pitchError;

tC(3) = -cYaw *xYaw - dYaw*yawError;
xYaw = xYaw + aYaw*xYaw + bYaw*yawError;
```

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

**Listing 14.14.** Compute wheel speed demand

```matlab
% Convert torque demand to RWA angular acceleration demand
% --------------------------------------------------------
wDRWA = -aRWA*tC;

% Integrate to get wheel speed demand
% -----------------------------------
wRWAC = wRWAC + tSamp*wDRWA;
```

The wheels are commanded through the tach loops.

**Listing 14.15.** RWA tach loops

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

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

**Listing 14.16. Simulation**

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

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

**Listing 14.17. Saving plot information**

```
% 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;
end

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

After the simulation is done we plot the results.

**Listing 14.18. Plotting**

```
Plot2D(tPlot,xPlot( 1: 4,j),’Time’(sec),’[’Qs’;’Qx’;’Qy’;’Qz’]’,’Quaternion’)
Plot2D(tPlot,xPlot( 5: 7,j),’Time’(sec),’[’Wx’;’Wy’;’Wz’]’,’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’)
```
Figure 14.2 shows the results of \texttt{PlotV}. It shows the spin axis unit vector. It starts as [0;0;1] (the highest point on the plot) and as the spacecraft acquires picks up $y$ and $x$ components until it traces out a circle in the $xy$-plane. Figure 14.3 shows the precession of the unit vector.

\textbf{Figure 14.2.} Motion of the unit vector

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig14_2.png}
\caption{Motion of the unit vector}
\end{figure}

\textbf{Figure 14.3.} Precession of the unit vector

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig14_3.png}
\caption{Precession of the unit vector}
\end{figure}
14.7 Summary

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

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

ATTITUDE AND ORBIT ESTIMATION
CHAPTER 15

INTRODUCTION

15.1 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.

15.2 Demos

Table 15.1 lists all demos in the module and categorizes them according to topic.

<table>
<thead>
<tr>
<th>Category</th>
<th>Demo</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Estimation</td>
<td>CDKFDemo.m</td>
<td>Demonstrate the continuous discrete iterated Kalman Filter.</td>
</tr>
<tr>
<td>Orbit Estimation</td>
<td>UDKFDemo.m</td>
<td>Demonstrate the Kalman Filter using UD factorization.</td>
</tr>
<tr>
<td></td>
<td>BatchLSQDemo.m</td>
<td>Demonstrate batch orbit determination.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>OrbitDisturbanceEstimator.m</td>
<td>Demonstrates orbit disturbance estimation using linearized equations.</td>
<td></td>
</tr>
<tr>
<td>OrbitKalmanFilter.m</td>
<td>Test the orbit Kalman Filter with omni and GPS measurements.</td>
<td></td>
</tr>
<tr>
<td>SeqBatchLSQDemo.m</td>
<td>Demonstrate Sequential Batch Least-Squares Orbit Determination</td>
<td></td>
</tr>
<tr>
<td>StellarNavigationDemo.m</td>
<td>Test stellar navigation. Uses a random star catalog.</td>
<td></td>
</tr>
<tr>
<td>CovarianceAnalysis.m</td>
<td>Performs a covariance analysis of different gyros.</td>
<td></td>
</tr>
<tr>
<td>KFSunAndEarthTest2.m</td>
<td>Test a Kalman Filter using a sun vector and the earth vector.</td>
<td></td>
</tr>
<tr>
<td>MagSimWithEstimator.m</td>
<td>Simulate attitude determination for a momentum bias spacecraft.</td>
<td></td>
</tr>
<tr>
<td>SunEarthADSim.m</td>
<td>Test a sun sensor attitude determination system.</td>
<td></td>
</tr>
<tr>
<td>YRGCDemo.m</td>
<td>Demonstrate the yaw/roll gyrocompassing system.</td>
<td></td>
</tr>
<tr>
<td>ADSim.m</td>
<td>Test a star sensor attitude determination system with gyros.</td>
<td></td>
</tr>
<tr>
<td>ADSimNoGyros.m</td>
<td>Test a star sensor attitude determination system.</td>
<td></td>
</tr>
<tr>
<td>ADSimStatic.m</td>
<td>Demonstrates static attitude determination.</td>
<td></td>
</tr>
<tr>
<td>HipparcosDemo.m</td>
<td>Demonstrates star identification using the Hipparcos catalog.</td>
<td></td>
</tr>
<tr>
<td>HipparcosGroupDemo.m</td>
<td>Demonstrates star identification using the Group algorithm.</td>
<td></td>
</tr>
<tr>
<td>KalmanFilterDemo.m</td>
<td>Demonstrate a Kalman Filter for a gyro based attitude determination system.</td>
<td></td>
</tr>
<tr>
<td>PGIcatLoad.m</td>
<td>Brings up the plugin with desired catalog and dPTol</td>
<td></td>
</tr>
<tr>
<td>PinholeCameraDemo.m</td>
<td>Demonstrate the Pinhole camera function</td>
<td></td>
</tr>
<tr>
<td>QFromUDemo.m</td>
<td>Demonstrate QFromU which performs static attitude determination.</td>
<td></td>
</tr>
<tr>
<td>StarDataGenerationDemo.m</td>
<td>Demonstrate StarDataGeneration.</td>
<td></td>
</tr>
<tr>
<td>StarIDBatch.m</td>
<td>The star identification plugin batch test.</td>
<td></td>
</tr>
<tr>
<td>StellarADDemos.m</td>
<td>Run a variety of examples using the group attitude determination</td>
<td></td>
</tr>
<tr>
<td>IsothermalUKF</td>
<td>A UKF that estimates thermal parameters for an isothermal model.</td>
<td></td>
</tr>
<tr>
<td>IsothermalUKFState</td>
<td>Estimate temperature and external flux using a UKF.</td>
<td></td>
</tr>
<tr>
<td>PressureKF.m</td>
<td>Demonstrate a fuel mass Kalman filter for a blowdown system.</td>
<td></td>
</tr>
<tr>
<td>UKFDemoNLSpring.m</td>
<td>Simulate a UKF with a nonlinear spring example.</td>
<td></td>
</tr>
<tr>
<td>UKFDemoOrbit.m</td>
<td>Demonstrate orbit estimation using a UKF.</td>
<td></td>
</tr>
<tr>
<td>UKFDemoOrbitOpticalNav</td>
<td>Demonstrate orbit estimation using a UKF with optical measurements.</td>
<td></td>
</tr>
<tr>
<td>UKFPDemoNLSpring.m</td>
<td>Demonstrate UKF parameter estimation with a nonlinear spring example.</td>
<td></td>
</tr>
<tr>
<td>UKFPDemoRigidBody.m</td>
<td>Demonstrate UKF parameter estimation with a rigid body.</td>
<td></td>
</tr>
<tr>
<td>OrbitFDIRDemo.m</td>
<td>Demonstrate an orbit detection filter for a spacecraft with one thruster.</td>
<td></td>
</tr>
<tr>
<td>SpinnerFDIRDemo.m</td>
<td>Demonstrate a detection filter for a spinning spacecraft.</td>
<td></td>
</tr>
<tr>
<td>FDIROLASim.m</td>
<td>Test Fault Diagnosis using online approximators.</td>
<td></td>
</tr>
<tr>
<td>DCMotorSim.m</td>
<td>Simulates 2 DC Motors with failure detection logic.</td>
<td></td>
</tr>
</tbody>
</table>

There are additional examples in the accompanying book, *Spacecraft Attitude and Orbit Control*. 
CHAPTER 16

GENERAL ESTIMATION

16.1 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.

16.2 Fixed Gain Estimators

Fixed gain estimators are written in the form

\[
 x_{k+1} = ax_k + bu_k + k(y - cx_k)
\]  

(16.1)

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 16.1 shows the covariance for this problem.

Figure 16.1. Fixed gain Kalman filter

16.3 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 16.2 on the facing page.

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

16.4 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 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.
16.5 Continuous Discrete Extended Kalman Filter

16.5.1 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 16.3 shows that the estimate converges to the true value of the spring position less a bias. The second shows the noisy measurement.

**Figure 16.3.** Continuous discrete Kalman filter

16.5.2 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 16.4 on the facing page.

16.6 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...
Figure 16.4. Clock error covariance for an orbit Kalman Filter

sigma points which are the states plus the square roots of rows or columns of the covariance matrix. Thus the state 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 18.
CHAPTER 17

**STELLAR ATTITUDE DETERMINATION**

**17.1 Introduction**

This chapter describes how to use the stellar attitude determination routines in the Estimation module to develop attitude determination systems for your applications.

**17.2 Star Catalogs**

The toolbox includes the catalogs given in Table 17.1.

**Table 17.1. Star catalog**

<table>
<thead>
<tr>
<th>Catalog</th>
<th>Number of Stars</th>
<th>Minimal Visual Magnitude</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hipparcos</td>
<td>118218</td>
<td>14</td>
<td>Stars measured by the Hipparcos</td>
</tr>
<tr>
<td>FK5</td>
<td>1536</td>
<td>8.35</td>
<td>Standard star catalog</td>
</tr>
<tr>
<td>FK5 Polar</td>
<td>14</td>
<td>7.17</td>
<td>Polar stars from FK5</td>
</tr>
<tr>
<td>FK5 Short</td>
<td>768</td>
<td>7.75</td>
<td>Subset of FK5</td>
</tr>
</tbody>
</table>

You load a catalog by typing:

**Listing. Catalog**

```matlab
starCatalog = LoadCatalog( 'FK5 Short', 'all' )
```

'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 17.1 is generated.

Table 17.2. Star catalogs

<table>
<thead>
<tr>
<th>Catalog</th>
<th>Number of Stars</th>
<th>Minimum visual magnitude</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hipparcos</td>
<td>118218</td>
<td>14</td>
<td>Stars measured by the Hipparcos satellite.</td>
</tr>
<tr>
<td>FK5</td>
<td>1536</td>
<td>8.35</td>
<td>Standard star catalog.</td>
</tr>
<tr>
<td>FK5 Polar</td>
<td>14</td>
<td>7.17</td>
<td>Polar stars from FK5.</td>
</tr>
<tr>
<td>FK5 Short</td>
<td>768</td>
<td>7.7500</td>
<td>Subset of FK5.</td>
</tr>
</tbody>
</table>

Figure 17.1. 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,

\[
d = \text{StarDataGeneration}( \text{starCatalog}, \text{fieldOfView}, \text{angularResolution} )
\]

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 17.2 on page 138. The fieldOfView should circumscribe the sensing element. StarDataGeneration does not 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 17.3. StarDataGeneration output fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Units or Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>starMatrix</td>
<td>(4,:;4,:)</td>
<td>[intensity;rad;rad;rad]</td>
<td>An array with intensity and unit vectors for all stars in star catalog.</td>
</tr>
</tbody>
</table>
17.3 StarIdentification Behavior

17.3.1 Introduction

StarIdentification has four options for star identification. They are...
The algorithms are explained below. The first three can be accessed through the StarIdentification function. The last can be accessed through StarIdentificationGroup. In addition, the toolbox provides functions for using intensity to select pairs of stars or single stars. These functions would be used if one or two stars appeared in the field-of-view.

17.3.2 Triad

The triad method works by looking at all combinations of three stars in the measured set. Stars pairs that match the three measured dot products are reconciled to find three stars that are consistent with the measurements. This is repeated with all triads. The resulting sets of star identifications are combined and the star that appears most often for each measurement is selected.

If a triad does not produce any identifications it is ignored. This method is insensitive to false stars but may fail if there are too many catalog dot products that match the measured dot products. This can happen with low resolution imagers.

17.3.3 PivotStar

PivotStar works as follows.

1. The pixel map is taken from the camera and the centroids of all bright objects are computed.
2. The pixel locations for each centroid are converted into measured unit vectors
3. The dot product of each unit vector is taken with all the rest resulting in a scalar array of dot products
4. The dot products are compared against all dot products in the catalog. The dot products of all stars and their neighbors are precomputed.
5. The catalog has three entries, primary star, neighboring star and dot product. A list of hits for each primary star is generated.

6. Each measured dot product may correspond to several catalog dot products. All of these dot products and their catalog stars are recorded. The stars that appear in all dot product sets are considered star candidates for that measurement. If there is no common star, the measurement is discarded.

7. Using only the remaining measurements after 6, the sets are combined to see if any consistent star identifications are possible. For example, assume that there are three measurements. There are two unique measured dot products using measurement 1 as the pivot and 1 using measurement 2. With pivot 1 particular stars selected for measurement 1 implies a star (or stars) for measurements 2 and 3. However, using measurement 2 for the pivot will also generate potential star candidates for stars 2 and 3. A star is identified if combining the sets results in unique identifications for each star. If more than one star is possible, additional sets (if they are available) are incorporated. With only three stars this method is identical to the triad method.

Figure 17.3. Pivot method

Figure 17.3 illustrates the concept. When a dot product agrees with a cataloged dot product of one star it means that the star is somewhere on a circle around the primary star. The second star is necessary to reduce that to a point. When only two measurements are available, relative intensity is used to identify the stars after the dot product search. If only one star is available, an absolute intensity match is attempted.

17.3.4 PathSearch

PathSearch searches paths through the measured star field. It optionally uses intensity to narrow the searches. The intensity comparison is strictly pass-fail. A measured star is considered to have the same intensity as all catalog stars which are within the specified tolerance (input variable intTol). If there are no catalog stars within this tolerance, then no attempt at identification of that measurement (although other measurements in the same set may be identified).

The first cut at pattern matching is a pass-fail criteria. Only stars which passed the intensity test are included. Once star pairs which fail to meet the dot product tolerance (input variable dPTol) are eliminated, then the difference between the measured dot products and those of the catalog star pairs is minimized.

No identification will be possible if

- there are no measurements which match any of the catalog intensities.
- there is only one measurement, or only one measurement which has an intensity match, and that measurement has more than one intensity match, resulting in an ambiguity.
• there is no set of catalog stars with suitable intensities that match the pattern of the measured stars within the specified tolerance.
• there are some measurements for which there are no intensity matches, those measurements cannot be identified.
• there is one (or more) measurement that has no matches for any of its dot products, then that measurement will not be identified.

Without significant culling based on intensity, this algorithm can be very slow!

### 17.3.5 Group

Group is a recursive algorithm that looks at all stars in the field-of-view to find a consistent subset of the stars in the field-of-view that match the catalog. It is the most robust of all the algorithms to false stars in the field-of-view. When called via `StarIdentificationGroup` it automatically uses absolute intensity if only one star is in the field-of-view and relative intensity if two stars are in the field-of-view and identification cannot be made by angular separation alone.

### 17.4 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

\[
\text{PixelMapCentroid}
\]

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 17.4 on the facing 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.

### 17.5 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 17.4. This data structure is used by

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Units or Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pixelLocation</td>
<td>(2,m)</td>
<td>pixels</td>
<td>The star centroid in pixels</td>
</tr>
<tr>
<td>intensity</td>
<td>(2,m)</td>
<td>intensity</td>
<td>The intensity of the star</td>
</tr>
</tbody>
</table>

### 17.5.1 PixelMapCentroid

PixelMapCentroid is invoked with the call

\[
\text{Spacecraft Control Toolbox}\n\]
17.5. GENERATING STAR DATA

Figure 17.4. Centroiding demo

![Centroiding demo](image)

```matlab
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

```matlab
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.

17.5.2 PinholeStarCamera

PinholeStarCamera is invoked with the call

```matlab
[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

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

Any value for drawImage cause an image to be drawn.
17.6 StarIDGUI

This GUI allows you to test star identification. You can read in sets of star data, enter parameters and identify them using intensity and/or angular separation. The GUI will automatically display the star catalog and the star data. The GUI is shown after the StarIDBatch demo in Figure 17.5.

Figure 17.5. GUI with identified stars

![Star Identification GUI](image)

17.6.1 Display boxes

**Star Data.** This is a list of star measurement data. To set the directory, click Select Directory in the bottom row of buttons. Use the file dialog the find the directory of interest. Enter the directory and select any file. All files that are not .m or .mat files will be listed. Files are expected to be formatted with one centroid per row with each row containing the x pixel location, y pixel location and intensity.

**Listing.** Typical star file

```
1  72.48  36.02  271
2  30.36  46.09  581
3  235.13  49.91  191
4  68.17  56.64  635
5  219.5  57.5  31
6  97.37 110.43  949
7   74.5 149.5  29
8  143.55 173.06  305
9  179.5  204.5  30
10  194.38  227.33  1163
11  170.5  225.5  29
```
Selected Data. This is a list of star data to be analyzed. Each file will be analyzed separately. If you hit Run Movie each file will be displayed as a frame in a movie. If you hit Snapshots, all of the frames will be displayed in a single figure.

Stars. This displays the identified stars.

17.6.2 Inputs

Intensity Scale. This converts the intensity data input to units of watt/m$^2$.

Pixels. The number of x and number of y pixels.

X Axis FOV (deg). The field of view in the x pixel direction.

Load Catalog. Load the catalog selected in the popup menu.

Run Movie. Run a movie with each file in the Selected Data list as a frame in the movie.

Snapshots. Display all files in the Selected Data list in a single figure.

Tolerance. Tolerance on intensity. This is not used in the current version since the identification method only uses dot products.

Pixel Interpolation. The accuracy of the centroiding in fractions of a pixel. Most centroiding schemes are accurate to about 0.1. If you don’t get any identifications you can make this number as large as you like.

Method. The three star identification methods are selectable in the pull down menu.

17.6.3 Message Window

This window displays messages from the GUI.

17.6.4 Buttons

Close Plots. Close all plots generated by PSS toolbox functions.

Clear Messages. Clear the message list.

Identify. Identify the stars selected in the Star Data list. To select multiple files, hold down the shift key.

Select Directory. Select the directory with the star data.

Help. Get online help.

QUIT. Close the GUI.

17.6.5 Batch operation

You can run the software in batch. Type

StarIDBatch

Spacecraft Control Toolbox 143
The MATLAB command window will show the output in Listing 17.6.5. The camera is pointing at the north pole and the FK5 catalog is used. StarDataGeneration is called twice, once in the script and once in the StarIDGUI. Five figures are generated, see Figures 17.6 and 17.7 on the facing page. Only one dot product match is found per measured dot product. This is because the resolution is sufficiently high and we are using the catalog as the input.

**Listing.** StarIDBatch

```matlab
>> StarIDBatch
StarDataGeneration: Eliminating stars that cannot be separated based on the input angular resolution.
tol = 1
StarDataGeneration: 0 stars will be eliminated that are too close to other stars.
1345 stars will be in the reduced catalog.

Table of Star Names
-------------------
Number Name
63 145 ALP UMI
447 977 DEL UMI
1242

StarDataGeneration: Eliminating stars that cannot be separated based on the input angular resolution.
tol = 1.0000
StarDataGeneration: 0 stars will be eliminated that are too close to other stars.
1345 stars will be in the reduced catalog.
```

**Figure 17.6.** Camera view and FK5 catalog
Figure 17.7. Hits and star snapshot
18.1 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.

18.2 Pressure Kalman Filter

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

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

The script is fairly short so we show it all here.

Listing 18.1. Fuel mass Kalman filter example

```matlab
% 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) 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.
```
18.2. PRESSURE KALMAN FILTER

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.

% (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.
%------------------------------------------------------------------------
% Copyright (c) 2006 Princeton Satellite Systems, Inc.
% All rights reserved.
%------------------------------------------------------------------------

\begin{verbatim}

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/2ˆ12;
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.dP = 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', pMeas, d );
end
\end{verbatim}

Spacecraft Control Toolbox 148
The top plot in Figure 18.1 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.
PART IV

ORBIT PROPAGATION AND MANEUVERS
CHAPTER 19

ORBIT MODULE

19.1 Overview

This module (Orbit) contains software for orbit mechanics, orbit coordinate transformations, and maneuver analysis and planning. 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!

19.2 Orbit Demos

Table 19.1 lists all demos in the module and categorizes them according to topic. In addition many functions have built-in demos, such as those in Glideslope.

Table 19.1. Estimation demos

<table>
<thead>
<tr>
<th>Category</th>
<th>Demo</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrbitManeuver</td>
<td>ApophisTransfer</td>
<td>Compute a transfer from earth orbit to the asteroid Apophis.</td>
</tr>
<tr>
<td></td>
<td>ComStarAKMDV</td>
<td>Plan the AKM burn by looking at the AKM thrust variations</td>
</tr>
<tr>
<td></td>
<td>DragCompensationDemo</td>
<td>Demonstrates drag compensation maneuvers.</td>
</tr>
<tr>
<td></td>
<td>GEOStationKeeping</td>
<td>GEO Stationkeeping Example</td>
</tr>
<tr>
<td>InsertionDeltaV</td>
<td>Computes the insertion delta-V for a Hohmann transfer.</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>LEOPAnalysis</td>
<td>Perform LEOP (Launch and Early Analysis Phase) analysis.</td>
<td></td>
</tr>
<tr>
<td>LowThrustEscapeSim</td>
<td>Simulates a low thrust escape from an earth orbit.</td>
<td></td>
</tr>
<tr>
<td>LowThrustHeliocentricSim</td>
<td>Perform a heliocentric simulation that goes from one circular orbit to another.</td>
<td></td>
</tr>
<tr>
<td>LowThrustMarsMission</td>
<td>Compute the delta-V for a low thrust mission to Mars synchronous orbit.</td>
<td></td>
</tr>
<tr>
<td>LowThrustMarsOptimalTransfer</td>
<td>Perform an optimal transfer from earth to mars orbits.</td>
<td></td>
</tr>
<tr>
<td>MarsOptimalTransfer</td>
<td>Perform an optimal transfer from earth to mars orbits.</td>
<td></td>
</tr>
<tr>
<td>OrbMnvrDemo</td>
<td>Runs demonstrations of selected orbit maneuver functions.</td>
<td></td>
</tr>
<tr>
<td>SolarSailSim</td>
<td>Orbit simulation of a solar sail.</td>
<td></td>
</tr>
<tr>
<td>TargetDemo</td>
<td>Demonstrates the Lambert targeting function DVTarget.</td>
<td></td>
</tr>
<tr>
<td>TwoBurnGeoTransfer</td>
<td>Delta-V for a Hohmann transfer between LEO and GEO.</td>
<td></td>
</tr>
<tr>
<td>TwoStageDesign</td>
<td>Computes a preliminary design of a two stage to orbit vehicle.</td>
<td></td>
</tr>
</tbody>
</table>

| ComStarFBudget | Demonstrate a fuel budget for the ComStar satellite. |
| EarthMoonSunSim | Simulate a spacecraft in orbit near the earth and moon. |
| EarthOrbitSim | Simulates a spacecraft in earth orbit with spherical harmonics. |
| InterstellarSim | Interstellar mission simulation with a sun gravity assist. |
| LinOrbitDemo | Linear orbit simulation. This compares nonlinear with linear. |
| LunarOrbitSim | Simulate an orbit about the moon. Show the visibility sun, earth, moon. |
| RelativeOrbitControl | Simulates two orbits and applies a relative controller. |
| RelativeOrbitSim | Simulates two orbits and plots their relative positions. |
| TDV | Demonstrate the various orbit change functions. |
| TDVDisp | Compute delta-v dispersions given declination and velocity errors. |
| TNORAD | Demo two of the five NORAD element propagators. |
| TOrbit | Investigate orbit coordinate systems. |
| TVarEq | Simulate an orbit using the variational equations. |
| TwoOrbSim | Simulates two orbits and plots their relative positions |

| DragDemo | Demonstrate drag reentry. |
| IntegrationAccuracyDemo | Integration Accuracy for PropagateOrbitPlugin. |
| LowThrustDemo | Demonstrate low thrust transfer using PropagateOrbitPlugIn. |
| OPBatchDemo | Demonstrate using PropgateOrbitPlugin in batch. |
| PropagatorComparison | Integration Accuracy study comparing RK4, RK45, and ode113. |
| TopexDemo | Compare various orbit propagators with a Topex ephemeris. |

| MarsOrb | Generates a Mars orbit plot. |
| OrbViewDemo | Demonstrate the OrbView function. |
| RelMotionVis | Visualize relative orbital motion in the inertial frame. |

There are additional examples in the accompanying book, *Spacecraft Attitude and Orbit Control.*
CHAPTER 20

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.

20.1 Low Thrust Orbit Raising

Reference


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)

\[
\frac{da}{dt} = \frac{2}{\sqrt{\mu}} a^{3/2} A
\]  

(20.1)

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

\[
a^{-1/2} = a_0^{-1/2} - \frac{A}{\sqrt{\mu}} (t - t_0)
\]  

(20.2)

Results

Type

LowThrustDemo

in the command window. The last plot will show the difference between the propagator and the analytical model (Figure 20.1 on the next page).

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

### 20.2 Reentry Due to Drag

**Reference**


**Introduction**

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

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

(20.3)

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
\[ H = h \ln \left( e^{H_0/h} - \frac{\rho_0 \sqrt{\mu R_E}}{h} \frac{C_D A}{m} (t - t_0) \right) \] (20.4)

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

Results

Type

DragDemo

As expected, the propagator shows slightly less decay due to the rotation of the atmosphere (Figure 20.2).

Figure 20.2. Reentry Due to Drag

---

20.3 Topex Ephemerides

Reference

**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 20.3. The range of $R$ and $V$ do not vary much over the period of the orbit.

**Figure 20.3.** Topex Ephemeris Cycle 10

![Figure 20.3. Topex Ephemeris Cycle 10](image)

Figure 20.4 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 20.1 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 20.5 on the facing page shows the difference between the Topex Ephemeris and `PropagateOrbitPlugin` when just the GEM-T1 spherical earth model was used. The results (as expected) are the same.
Table 20.1. Topex Osculating Elements at the Beginning of the Simulation.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>7719.2 km</td>
</tr>
<tr>
<td>$i$</td>
<td>1.1528 rad</td>
</tr>
<tr>
<td>$\omega$</td>
<td>-0.39392 rad</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>-0.94544 rad</td>
</tr>
<tr>
<td>$e$</td>
<td>0.00025586</td>
</tr>
<tr>
<td>$M$</td>
<td>0.52091 rad</td>
</tr>
</tbody>
</table>

Figure 20.4. Difference Between the Topex Ephemeris and a Keplerian Orbit

Figure 20.5. Difference Between the Topex Ephemeris and a Propagated Spherical Earth Model
Figure 20.6 shows the Topex Orbit in 3D referenced to the earth-fixed frame.

**Figure 20.6.** Topex Orbits in 3D

Figure 20.7 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 20.8 on the next page.

**Figure 20.7.** Difference Between the Topex Ephemeris and a propagated orbit with 4 Zonal Harmonics

Again, an improvement is noted.
20.4 Integration Tolerance

Reference

MATLAB Users Guide.

Introduction

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

Results

Type

IntegrationAccuracyDemo

Figure 20.9 on the following page shows the effects of varying the integration tolerances for a 7000 km circular orbit.

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

The differences are substantial. If you used the tolerances in the upper right hand corner, and didn’t know that the orbit was unperturbed, you might think that the orbit was decaying! This also shows the danger of comparing one simulation against another. Unless you know the error tolerances for both runs, and have a good idea how the error correction algorithms work in both cases, your comparison could be meaningless.

You always want to get your simulations done as fast as possible. This is particularly true if you are doing Monte Carlo type analyses. Tighter tolerances mean longer simulation times. A balance must be made between integration
Figure 20.9. 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 20.2. Simulation Times for a 140 hour Circular Orbit Simulation run on a 500 MHz Apple PowerBook G3

<table>
<thead>
<tr>
<th>Rel Tol</th>
<th>Abs Tol</th>
<th>Duration (sec)</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e-12</td>
<td>1e-12</td>
<td>76.3</td>
<td>15 cm</td>
</tr>
<tr>
<td>1e-06</td>
<td>1e-12</td>
<td>36.2</td>
<td>1000 m</td>
</tr>
<tr>
<td>1e-06</td>
<td>1e-06</td>
<td>37.6</td>
<td>1000 m</td>
</tr>
<tr>
<td>1e-12</td>
<td>1e-06</td>
<td>62.4</td>
<td>20 cm</td>
</tr>
</tbody>
</table>
CHAPTER 21

ORBIT MANEUVERS

This chapter shows you how to plan orbit maneuvers.

21.1 Orbit Maneuver Functions

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

Listing 21.1. Orbit maneuver demonstration script

```plaintext
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 21.2. 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
```
21.2 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 21.1 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 21.1. Launch and early orbit operations analysis](image)

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.

### 21.3 Drag Correction Example

This example shows how to compute the compensation for drag. The script is shown in Listing 21.3. 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 21.3. Drag correction script**

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

**DragCompensationDemo.m**
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( 'setData', tag, d );
PropagateOrbitPlugIn( 'propagate', tag );

r = PropagateOrbitPlugIn( 'get r', tag );
v = PropagateOrbitPlugIn( 'get v', tag );

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(:,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 %.12f km/sec',deltaV.total));

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 21.2 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 21.3 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 21.2. Orbit propagation GUI and aero model inputs

Figure 21.3. Comparison of analytical drag with propagated
21.4 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 21.4. Default data for the built-in demo

```matlab
%-------------------------------------------------------------------------------
% 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];
```

Figure 21.4. Orbital elements

![Orbital elements graph]

Spacecraft Control Toolbox 168
The results of this demo are shown in Figure 21.5. The maneuvers are indicated by red triangles.

**Figure 21.5.** Glideslope demo results
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.

### 22.1 The Orbit Propagator Tool

#### 22.1.1 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.

#### 22.1.2 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.
22.2. USING THE GRAPHICAL USER INTERFACE

- 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.

22.2 Using the Graphical User Interface

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

Figure 23-1 Orbit Propagation GUI

22.2.1 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.

22.2.2 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
CHAPTER 22. ORBIT PROPAGATOR GUI

22.2. USING THE GRAPHICAL USER INTERFACE

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 22.1. Element sets

<table>
<thead>
<tr>
<th>Keplerian</th>
<th>RP/RA</th>
<th>Equinoctial</th>
<th>RV</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Semi-major axis</td>
<td>R_p</td>
<td>Perigee radius</td>
</tr>
<tr>
<td>i</td>
<td>Inclination</td>
<td>i</td>
<td>Inclination</td>
</tr>
<tr>
<td>( \Omega )</td>
<td>Right Ascension of the Ascending Node</td>
<td>( \Omega )</td>
<td>Right Ascension</td>
</tr>
<tr>
<td>( \omega )</td>
<td>Argument of Perigee</td>
<td>( \omega )</td>
<td>Argument of Perigee</td>
</tr>
<tr>
<td>e</td>
<td>Eccentricity</td>
<td>R_a</td>
<td>Apogee radius</td>
</tr>
<tr>
<td>M</td>
<td>Mean anomaly</td>
<td>M</td>
<td>Mean anomaly</td>
</tr>
</tbody>
</table>

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.

Spacecraft Control Toolbox 173
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 22.2. Stopping Conditions

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

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:

$$pDot = f^t p + p f^t + q$$

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

\[ e(i) \leq \max(\text{RelTol} \cdot |y(i)|, \text{Tol}(i)) \]

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

\[ p = 0.5 \cdot (p + p') \]

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: 
  \([x;y;z;vx;vy;vz;mass;pXX;\ldots;pVZVZ;pXYPY;\ldots]\) 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 Plots** Close the plot windows.
- **Open** Open a gui dat file of type *.OPG.
- **Save Data** Save the gui data in a mat file of type *.OPG.
- **Propagate** Run the propagator.
- **QUIT** Quit. It will ask you to save unsaved plot and GUI data.
- **Help** Get help.

### 22.3 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 22.3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEngine</td>
<td>Computes engine force and spacecraft mass. The force is in the LVLH frame.</td>
</tr>
<tr>
<td>FAero</td>
<td>Computes aero forces in the ECI frame.</td>
</tr>
<tr>
<td>FSolar</td>
<td>Computes solar forces in the ECI frame.</td>
</tr>
<tr>
<td>FPlantNoise</td>
<td>The Q matrix in the covariance propagation equation.</td>
</tr>
<tr>
<td>FAtmDensity</td>
<td>Atmospheric density. The baseline model is Jacchia J70.</td>
</tr>
</tbody>
</table>

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 22.1, allows you to enter a constant force in the ECI frame and a constant mass change rate.

**Figure 22.1. FEngine GUI**

![FEngine GUI](image)

The **FAero** model, Figure 22.2, 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 22.2. FAero GUI**

![FAero GUI](image)

The **FSolar** model, Figure 22.3 on the next page, is a flat plate model in which the plate is always normal to the sun vector.

The **FPlantNoise** function, Figure 22.4 on the facing page, assumes a constant plant noise covariance matrix. The inputs are the uncertainty in body accelerations.

The **FAtmDensity** GUI, Figure 22.5 on the next page, gives inputs for the Jacchia J70 model. The J70 inputs are given in Table 22.4.

**Table 22.4. Jacchia J70 Atm inputs**

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aP</td>
<td>Geomagnetic index 6.7 hours before the computation</td>
</tr>
<tr>
<td>f</td>
<td>Daily 10.7 cm solar flux (e-22 watts/m²/cycle/sec)</td>
</tr>
<tr>
<td>fHat</td>
<td>81-day mean of f (e-22 watts/m²/cycle/sec)</td>
</tr>
<tr>
<td>fHat400</td>
<td>fHat 400 days before computation date</td>
</tr>
</tbody>
</table>
Figure 22.3. FSolar GUI

Figure 22.4. FPlantNoise GUI

Figure 22.5. FAtmDensity GUI
22.4 Building a Plugin Function

All plugin functions have the same structure. In this section, we will walk you through the structure of \texttt{FAero}, the simplest function.

\textbf{Listing 22.1. Function Definition}

\begin{verbatim}
function x = FAero( action, modifier, u, t )
\end{verbatim}

Listing 22.2 shows the switch statement that makes up the main function. \texttt{action} is the argument for the switch statement and \texttt{modifier} is the GUI tag, which identifies the GUI.

\textbf{Listing 22.2. Main Function Body}

\begin{verbatim}
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
\end{verbatim}

The \texttt{Initialize} subfunction draws the GUI and returns the tag for the figure.

\textbf{Listing 22.3. Initialize}

\begin{verbatim}
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;
\end{verbatim}
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 22.4) generates a call back to the FAero function. For example the last line creates the call:

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

Listing 22.4. CreateCallback

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 22.5. Getting and setting data

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

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

```
FAero.m
```

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

FAero.m
```

```
function d = GetData( modifier )

h = GetH( modifier );
```

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 22.6. The Update subfunction**

```matlab
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
```

## 22.5 Reference Frames

### 22.5.1 Coordinate Transformations

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

- ECI
- ECR
- Geodetic latitude, longitude and altitude

For example

```matlab
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
```
22.5.2 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

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

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

```plaintext
GASTime( x )
ans =
7.2978e-09
```

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

```plaintext
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.

22.6 Sun

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

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

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

22.7 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.

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

22.8 Ground Coverage

Ground coverage is found using this function from OrbitMechanics:

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

where $x_{ECI}$ 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.
Table 22.5. I/O for STKOrbit

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

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.

22.9 Orbit Propagation API

22.9.1 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.

```matlab
   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 22.6.

Table 22.6. PropagateOrbitPlugIn API

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag = PropagateOrbitPlugIn(‘initialize’);</td>
<td>Initialize the plugin.</td>
</tr>
<tr>
<td>PropagateOrbitPlugIn(‘set data’, tag, d);</td>
<td>Set the data in the GUI</td>
</tr>
<tr>
<td>PropagateOrbitPlugIn(‘propagate’, tag);</td>
<td>Propagate the orbit.</td>
</tr>
<tr>
<td>PropagateOrbitPlugIn(‘close plots’, tag);</td>
<td>Close the plots</td>
</tr>
<tr>
<td>PropagateOrbitPlugIn(‘save’, tag, fileName );</td>
<td>Save the GUI data.</td>
</tr>
<tr>
<td>PropagateOrbitPlugIn(‘save plot data’, tag, fileName );</td>
<td>Save the plot data</td>
</tr>
<tr>
<td>PropagateOrbitPlugIn(‘quit’, tag );</td>
<td>Close the GUI.</td>
</tr>
</tbody>
</table>
22.9.2 Data Structure

The data structure \( d \) is defined in the following table.

**Table 22.7. Input data structure for PropagateOrbitPlugIn**

<table>
<thead>
<tr>
<th>Field</th>
<th>SubField</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>thrustFunction</td>
<td>Struct</td>
<td>Plugin function names and data</td>
</tr>
<tr>
<td></td>
<td>aeroFunction</td>
<td>String (1,:)</td>
<td>Name of the aerodynamics function</td>
</tr>
<tr>
<td></td>
<td>solarFunction</td>
<td>String (1,:)</td>
<td>Name of the solar pressure function</td>
</tr>
<tr>
<td></td>
<td>noiseFunction</td>
<td>String (1,:)</td>
<td>Name of the covariance noise function</td>
</tr>
<tr>
<td></td>
<td>densFunction</td>
<td>String (1,:)</td>
<td>Name of the atmospheric density function</td>
</tr>
<tr>
<td></td>
<td>thrustData</td>
<td>User defined</td>
<td>Data for the users thrust function</td>
</tr>
<tr>
<td></td>
<td>aeroData</td>
<td>User defined</td>
<td>Data for the users aerodynamic function</td>
</tr>
<tr>
<td></td>
<td>solarData</td>
<td>User defined</td>
<td>Data for the users solar pressure function</td>
</tr>
<tr>
<td></td>
<td>noiseData</td>
<td>User defined</td>
<td>Data for the users covariance noise function</td>
</tr>
<tr>
<td></td>
<td>densData</td>
<td>User defined</td>
<td>Data for the users density function</td>
</tr>
<tr>
<td></td>
<td>planetaryDisturbancesOn</td>
<td>(1,1)</td>
<td>1 turns on planetary disturbances</td>
</tr>
<tr>
<td></td>
<td>atmosphere</td>
<td>String (1,:)</td>
<td>String for atmosphere model name</td>
</tr>
<tr>
<td></td>
<td>nZonal</td>
<td>(1,1)</td>
<td>Number of zonal harmonics</td>
</tr>
<tr>
<td></td>
<td>nTesseral</td>
<td>(1,1)</td>
<td>Number of tesseral harmonics</td>
</tr>
<tr>
<td></td>
<td>stopping.condition</td>
<td>String (1,:)</td>
<td>Stopping condition string</td>
</tr>
<tr>
<td></td>
<td>covariancePropagationOn</td>
<td>(1,1)</td>
<td>Turns on covariance propagation</td>
</tr>
<tr>
<td>( q )</td>
<td>simpleCovarianceRHSOn</td>
<td>(1,1)</td>
<td>Simple covariance right-hand-side enabled</td>
</tr>
<tr>
<td></td>
<td>propagate</td>
<td>String (1,:)</td>
<td>Propagation method</td>
</tr>
<tr>
<td>mass</td>
<td>epoch</td>
<td>(1,1)</td>
<td>Spacecraft mass</td>
</tr>
<tr>
<td></td>
<td>jDEpoch</td>
<td>(1,1)</td>
<td>Julian date (days)</td>
</tr>
<tr>
<td></td>
<td>dT</td>
<td>(1,1)</td>
<td>Time step (sec)</td>
</tr>
<tr>
<td></td>
<td>duration</td>
<td>(1,1)</td>
<td>Duration (sec)</td>
</tr>
<tr>
<td>tol</td>
<td>reltol</td>
<td>(1,1)</td>
<td>Absolute tolerance for ode113</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1,1)</td>
<td>Relative tolerance for ode113</td>
</tr>
</tbody>
</table>

22.9.3 Function Inputs

The following table lists all inputs to PropagateOrbitPlugIn.

**Table 22.8. Inputs to PropagateOrbitPlugIn**

<table>
<thead>
<tr>
<th>Action</th>
<th>Inputs</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize</td>
<td>None</td>
<td>None</td>
<td>Tells the GUI to initialize itself.</td>
</tr>
<tr>
<td>propagate</td>
<td>None</td>
<td>None</td>
<td>Propagate the orbit</td>
</tr>
<tr>
<td>propagate covariance</td>
<td>None</td>
<td>None</td>
<td>Propagate the covariance</td>
</tr>
<tr>
<td>atmosphere</td>
<td>None</td>
<td>None</td>
<td>Open the atmosphere GUI</td>
</tr>
<tr>
<td>gravity</td>
<td>None</td>
<td>None</td>
<td>Open the gravity GUI</td>
</tr>
<tr>
<td>set data</td>
<td>None</td>
<td>None</td>
<td>Provide a data structure to the GUI</td>
</tr>
<tr>
<td>get data</td>
<td>Data structure (see above)</td>
<td>None</td>
<td>Returns the GUI data structure</td>
</tr>
<tr>
<td>get results</td>
<td>GUI data structure</td>
<td>Plot data</td>
<td>Returns the plot data</td>
</tr>
<tr>
<td>get r</td>
<td>None</td>
<td>r(3,:)</td>
<td>Returns the position vector array</td>
</tr>
<tr>
<td>get v</td>
<td>None</td>
<td>v(3,:)</td>
<td>Returns the velocity vector array</td>
</tr>
<tr>
<td>get mass</td>
<td>None</td>
<td>(1,1)</td>
<td>Returns the spacecraft mass</td>
</tr>
</tbody>
</table>
### 22.9. Function Outputs

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

**Table 22.9. Plot outputs for PropagateOrbitPlugIn**

<table>
<thead>
<tr>
<th>Plot Data</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>x(1:3,:)</td>
<td>Position</td>
<td>km</td>
</tr>
<tr>
<td>x(4:6,:)</td>
<td>Velocity</td>
<td>km/sec</td>
</tr>
<tr>
<td>x(7,:)</td>
<td>Mass</td>
<td>kg</td>
</tr>
<tr>
<td>x(8,:)</td>
<td>Geodetic Latitude</td>
<td>deg</td>
</tr>
<tr>
<td>x(9,:)</td>
<td>Longitude</td>
<td>deg</td>
</tr>
<tr>
<td>x(10,:)</td>
<td>Altitude</td>
<td>km</td>
</tr>
<tr>
<td>x(11,:)</td>
<td>Flight path angle</td>
<td>rad</td>
</tr>
<tr>
<td>x(12:17,:)</td>
<td>Covariance diagonal</td>
<td></td>
</tr>
</tbody>
</table>
This chapter discusses how to generate fuel budgets. These budgets are some of the most important elements of spacecraft system design.

### 23.1 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 23.1. Fuel Budget: Moment Arms**

```matlab
% 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 -0.9 -0.8 -0.7 -0.8 -0.8 -0.8 -0.8 0.8 -0.8 -0.8 -0.7 -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 23.2. Fuel Budget: Thruster performance data**

```matlab
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
end
```

185
23.1. FUEL BUDGET

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 23.3. 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 23.4. 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 23.5. 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 23.6.** Fuel Budget: Continuous delta-v burn

```matlab
% 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 23.7.** Fuel Budget: Delta-v with off-pulsing

```matlab
% 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).disturbance = [];  
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 23.8.** Fuel Budget: Unloading momentum

```matlab
% 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 = [];  
event(k).dTControl = [1 2 3 4];  
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 23.9.** Fuel Budget: Using ACS thrusters

```matlab
% 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 23.10.** Fuel Budget: Firing a few pulses

```matlab
% 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).dTControl = 1;
event(k).disturbance = [];
event(k).nPulses = []; % Won't use any fuel this way
event(k).pulsethickness = 0.05;
event(k).duration = [];
event(k).regulated = 0;
event(k).cM = [0;0;0];
```

The results for these events are shown in **Listing 23.11**.

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

**Listing 23.11.** Fuel Budget Example, Six Events

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Total Fuel Remaining</th>
<th>Fuel Used</th>
<th>Pressure</th>
<th>ACS Isp</th>
<th>DV Isp</th>
<th>ACS Isp</th>
<th>DV Isp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial State</td>
<td>1190.92</td>
<td>117.89</td>
<td>0.00</td>
<td>0.00</td>
<td>2.4e+06</td>
<td>2.4e+06</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>Delta V Continuous</td>
<td>1153.87</td>
<td>99.36</td>
<td>99.36</td>
<td>18.525</td>
<td>8.525</td>
<td>2.4e+06</td>
<td>2.4e+06</td>
</tr>
<tr>
<td>3</td>
<td>Delta V Off Pulse</td>
<td>1099.18</td>
<td>72.02</td>
<td>72.02</td>
<td>27.346</td>
<td>27.346</td>
<td>1.7e+06</td>
<td>1.7e+06</td>
</tr>
<tr>
<td>4</td>
<td>Delta H 1099.18</td>
<td>72.02</td>
<td>0.001</td>
<td>0.001</td>
<td>0.00</td>
<td>1.2e+06</td>
<td>1.2e+06</td>
<td>100.41</td>
</tr>
<tr>
<td>5</td>
<td>ACS Disturbance</td>
<td>1087.45</td>
<td>66.16</td>
<td>66.16</td>
<td>5.862</td>
<td>5.862</td>
<td>1.2e+06</td>
<td>1.2e+06</td>
</tr>
<tr>
<td>6</td>
<td>Short Pulse</td>
<td>1087.45</td>
<td>66.16</td>
<td>66.16</td>
<td>0.000</td>
<td>0.000</td>
<td>1.2e+06</td>
<td>1.2e+06</td>
</tr>
</tbody>
</table>
For east/west stationkeeping the delta V required because of longitude drift, due to the earth's tesseral gravitational harmonics is computed by

\[
[dV_{EW}, dT_{EW}] = DVLDrift(\ box, \ scLon );
\]

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

\[
[dV_{NS}, dT_{NS}] = DVIDrift(\ box, \ year );
\]

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.
This chapter gives selected orbit resources.

24.1 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.


A good introduction with many physical insights.


24.2 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 V

SUBSYSTEMS ANALYSIS
CHAPTER 25

IMAGING

This chapter discusses how to use the Imaging module.

25.1 ImageMatching

ImageMatching has the Canny Edge Detection algorithm, image derivative and a function for fitting a circle to points in an image.

Figure 25.1. CannyEdgeDetector demo

25.2 ImageProcessing

This folder includes utilities for image processing

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

For example `GaussianImageFilter`.

**Figure 25.2**. 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.
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
```

### 26.1 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>BEP - Computes the bit error probability given C/N and the bit rate.</td>
</tr>
<tr>
<td>C</td>
<td>CNRComp - Compute the carrier to noise ratio.</td>
</tr>
<tr>
<td>D</td>
<td>DBPower - Power conversion to decibels. dB = 20 ( \log_{10}(s) ).&lt;br&gt;DBSignal - Signal conversion to decibels. dB = 10 ( \log_{10}(s) ).</td>
</tr>
</tbody>
</table>
26.2 OPTICAL

DBSignalToPower - Convert decibels to power.
DBWToNW - Convert dBW (decibels-Watts) to nanowatts.
DBWToPW - Convert dBW to picowatts.
DopplerBlind - Finds Doppler blind speeds.
DopplerShift - Doppler shift

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

P
ProbDetection - Probability of detection for a diffuse target.

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

T
TToPower - Convert temperature to power.

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

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

For example to get nanowatts from dB,

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

or picowatts

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

If you type Shannon you get Figure 26.1 on the next page (a demo of the Shannon function). This relates the information content to the signal to noise ratio.

26.2 Optical

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

- Solar
- Lunar
- Mercury
- Venus
- Mars
Figure 26.1. Shannon demo

• Jupiter
• Saturn
• Sunlit clouds
• Sunlit snow and ice
• Starfield

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

26.3 RF

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

\[
\text{LossAtmosphericGas}
\]

gives you path losses due to atmospheric gas, see Figure 26.2 on the following page. Other loss functions include LossFreeSpace and LossPrecipitation. Typing

\[
\text{TSky}
\]

gives you a sky temperature demo, see Figure 26.3 on the next page. You can also get temperatures for the moon, sun, receiver, attenuator, etc.

You can compute losses and noise inputs using a variety of functions in the toolbox. All of these tools are integrated into the LinkAnalysisTool. The built-in demo of this function is shown below and the results in Figure 26.4 on page 201.

\[
\text{% Space link}
\\\n\text{%---------}
\text{spaceLink.theta3DB = 2; % (deg)}
\text{spaceLink.fIllumination = 70; % (deg)}
\text{spaceLink.effTransmit = 0.55;}
\]

Spacecraft Control Toolbox 199
**Figure 26.2.** Atmospheric Gas Loss

**Figure 26.3.** Sky temperature
26.4 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.
\[
\begin{align*}
\text{FD} & = 12; \ \text{% GHz} \\
pT & = \text{DBSignal}(10); \ \text{% dBW} \\
r & = 40000; \ \text{% km} \\
\theta_3DB & = 2; \\
effT & = 0.55; \\
effR & = 0.6; \\
dR & = 4; \\
fIllumination & = 70;
\end{align*}
\]

\[
\begin{align*}
gT & = \text{Gain3dB}(\theta_3DB, fIllumination, effT) \\
eIRPSL & = pT + gT \\
gR & = \text{AntennaGain}(\text{struct('type','circularAperture','area',\pi*dR^2/4, 'eff', effR)}, fD) \\
lFS & = \text{LossFreeSpace}(fD, r) \\
pR & = eIRPSL - lFS + gR;
\end{align*}
\]

\begin{verbatim}
disp(sprintf('
\n2.3.2
Example 2:
The Downlink
-----------------------------'))
disp(sprintf('Gain Transmit Antenna = %10.2f (dBW)', gT));
disp(sprintf('EIRP SL = %10.2f (dBW)', eIRPSL));
disp(sprintf('Free Space Loss = %10.2f (dB)', lFS));
disp(sprintf('Gain Receive Antenna = %10.2f (dB)', gR));
disp(sprintf('Power Received = %10.2f (dBW)', pR));
disp(sprintf('Power Received = %10.2f (pW)', DBWToPW(pR)));
\end{verbatim}

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.

26.5 LADAR

Two main functions are available for analyzing LADAR systems. \texttt{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.

\begin{verbatim}
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
\end{verbatim}
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 26.5.

**Figure 26.5.** Results of LADARTransmitPower
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 26.6.

Figure 26.6. Results of RadarEquation

Type RadarCrosssection and you will get the radar crosssections of a plate and sphere.
Figure 26.7. Results of RadarCrosssection
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 27.1.

**Figure 27.1. Band gap**

![Bandgap for si](image)

SolarCell computes the I-V curves for solar cells as in Figure 27.2 on the following page.

SolarCellEff computes the efficiency for solar cells as in Figure 27.3 on the next page.
Figure 27.2. Solar cell I-V curves

![Solar cell I-V curves diagram]

Figure 27.3. Solar cell efficiency

![Solar cell efficiency diagram]
CHAPTER 28

PROPULSION MODULE

This chapter presents the propulsion module.

28.1 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.

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

Directories:
  Airbreathing
  Chemical
  Demos
  Demos/Airbreathing
  Demos/Chemical
  Demos/LVModels
  Demos/MultiStage
  Demos/Rocket
  Electric
  Help
  LVModels
  LaunchSim
  MultiStage
```
Nuclear
PropulsionData
Rocket
28.2 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 28.1 for the demo results.

**Figure 28.1.** Solid motor firing

![Solid Rocket Motor Simulation](image)

28.3 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.

```matlab
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 28.2 on the next page. With no fuel, the system pressure is 210 psi. With the full fuel load the pressure is 350 psi.
28.4 Electric propulsion

Type `ElectricPropulsion` and you will get the plot in Figure 28.3. This shows the mass of an electrically propelled vehicle as a function of exhaust velocity. The optimal exhaust velocity is surprisingly low.

The vehicle characteristics are printed in the MATLAB command window:
28.5 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 28.4 shows the results of the demo, in which a payload of 593 kg is delivered to orbit.

Figure 28.4. Two-State-to-Orbit Simulation
This chapter introduces the thermal analysis module.

29.1 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.

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

Directories:
Demos
Demos/ThermalAnalysis
Demos/ThermalGraphics
ThermalAnalysis
ThermalData
ThermalGraphics
ThermalProperties
```

29.2 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 29.1 on the following page.

`AbsDa` computes the absorbed heat as a function of absorptivity and incoming flux. See Figure 29.2 on the next page. Other functions compute the temperate of simple geometric objects under incoming radiation.

Another function is `PlanckL` which gives Planck's Law. See Figure 29.3 on page 217.
Figure 29.1. Black body wavelength as a function of temperatures

![Wiens Displacement Law](image1)

Figure 29.2. AbsDa demo

![Absorbed Radiation](image2)
29.3 Thermal Analysis

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

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

\[ lT^4 + kT = bq \]  

using MATLAB’s sparse matrix functions.

29.3.1 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

\[
\dot{T} = aT + bq \\
y = cT
\]  

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 29.4 on the next page.

External flux is coming in on node 1 and measurements are taken on node 3 and 2. There is conduction heat transfer between all nodes, but radiation transfer only between nodes 3 and 2, as indicated by the zigzag line.
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
```
### 29.3.2 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 ($2.2204e-16$).

```matlab
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)
```

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

### 29.3.3 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 29.5 on the following page shows the four windows that are generated.

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

```matlab
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.

```matlab
% 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 29.5. Thermal imaging demo

Figure 29.6. Thermal imaging demo temperature plot
APPENDICES
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 225.

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

<table>
<thead>
<tr>
<th>Digits</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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                                                                        |
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

```matlab
>> LoadSatP
```

A dialog box will appear listing all files with the suffix .TXT. A plot of the longitude for all GOES satellites from 1986 through the present will be displayed.

### A.4 GOES functions

#### A.4.1 LoadMag

`LoadMag` loads multiple magnetic field files into one array. For example put the GOES CD in your CD drive and type:

```matlab
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 facing page.

#### A.4.2 LoadSers

`LoadSers` loads GOES data. For example, with the CD in the drive, type

```matlab
[b,e,x,p,file] = LoadSers(7,'A','GOES_94',11,88,11,88);
Plot2D([],e,Sample,Electrons)
```

Figure A-3 on page 226 shows the resulting plot.

#### A.4.3 Weather

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

```matlab
>> Weather
```

and select the file in the GOESData folder. You will then see the plot in Figure A-4 on page 226.

### 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

![GOES Binary Data](image1)

Figure A-2. GOES Magnetic Field Data

![Magnetic Field](image2)
Figure A-3. GOES Electrons

![GOES Electrons Graph]

Figure A-4. GOES Weather

![GOES Weather Graph]
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

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**

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
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:

```plaintext
1  Constant('mass sun/mass jupiter')
2  ans =
3  1.0474e+03
```

The database loads its constants from a database the first time it is launched. Once it is launched, it will not load a new database. However there is a fair amount of overhead involved in searching for a constant so we recommend that whenever possible you get the constant once from the database and store it in a local variable.

### B.2 Merging Constant Databases

Periodically, PSS will release new constant databases. If you have customized your own database you can merge it with the PSS database using the `MergeConstantDB` function. Just type

```plaintext
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

```plaintext
MergeConstantDB('initialize','SCTConstants.mat','SCTConstantsOld.mat')
```

You will see the display in Figure B-3 on the facing page. Just click the button for the column you wish to include in `SCTConstants.mat`. 

---

**Figure B-2. Searching for mass**

![Constant Database Search](image)

**Figure B-3. Merging Constant Databases**

![Merging Database](image)
Figure B-3. MergeConstantDB function

![Diagram showing the Merge Constant DB function with two columns: one labeled \( g_0 \) and the other labeled \( g_0 \), both containing values 9.8066 and \( \text{m/s}^2 \). There are buttons labeled "Select this Column" for each column.]
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 MATLAB version 5.2 but are still supported in version 7. See Common/Plugins and SC/GUIPlugIn for available plug-ins.

C.1 GUI PlugIn Demo

PlugInDemo is a m-file function that implements an orbit simulation using several GUI PlugIn elements. This can be used as a template for your simulations.

Figure C-1. PlugInDemo on starting

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

- OrbitDisplayPlugin
C.2 Writing Your Own GUI Function

PlugInDemo is implemented as shown below.

Listing C.1. PlugInDemo top-level switch statement

```
function PlugInDemo( action )
% Process the input arguments
if( nargin < 1 )
    action = 'create_gui';
end
% Perform actions
```

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 facing page.

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

The 3D window (which is animated) looks like Figure C-5 on page 236.
**Figure C-3.** Selecting planets in the elements plugin

**Figure C-4.** PlugInDemo at the end of the simulation
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**

```matlab
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_Pli
```
'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 );
ext
    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;
ext
    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 );
and pass them to a uicontrol as

\[
\text{uicontrol( \{\}, \ldots)}
\]

the \{\} expands the values in the cell array so that this is the equivalent of

\[
\text{uicontrol('parent', h.fig, 'units', 'pixels', 'fontunits', 'pixels', \ldots)}
\]

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

```matlab
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 );
```

The simulation is run with the following code. FOrbCart returns the state derivatives for the orbit model.

**Listing C.4.** The simulation loop

```matlab
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;
```

The simulation is run with the following code. FOrbCart returns the state derivatives for the orbit model.
% Plotting
%---------
if(˜isempty(y))
    PlotPlugIn(‘clear plots’, h.plotPlugInTag, k);
    p.xLabel = {‘Time (sec)’};
    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);

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.

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 = sim.orbit.r;
OrbitDisplayPlugIn(‘update spacecraft’, h.scWindowTag, h.g, sim.time.jDEpoch);
DrawSCPlugin(‘update planet’, h.scWindowTag, sim.orbit.planet);
DrawSCPlugin(‘update planet’, h.scWindowTag, sim.orbit.planet);
PutH( h );

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 )
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**

```matlab
% Get the data from the plug ins
sim.orbit = ElementsPlugIn( 'get', h.elementsPlugInTag );
sim.time = TimePlugIn ( 'get', h.timePlugInTag );

%------------------------------------------------------------------------
% 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, ')'];
```