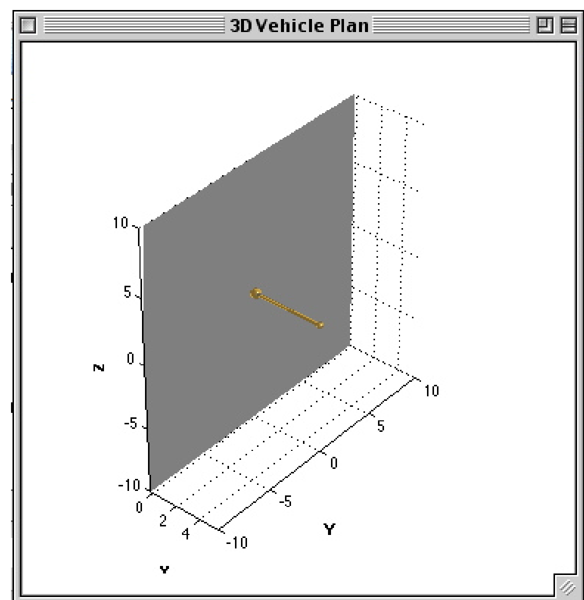
Princeton SATELLITE
SYSTEMS

Solar Sail Module

for the
Spacecraft Control Toolbox
Professional Edition



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.

Solar Sail Module

Printed April 5, 2020

©Copyright 2004-2007, 2009, 2011-2012 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 Street 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.

Printing History:

December 15, 2005 First Printing v1.0
July 15, 2006 Second Printing v1.1
April 30, 2007 Third Printing v1.1
October 19, 2009 Fourth Printing v1.2
May 31, 2012 Fifth Printing v1.2

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

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

Contents

1 Introduction **1**
- 1.1 Solar Sails . . . 1
- 1.2 Organization . . . 2
- 1.3 Requirements . . . 4
- 1.4 Installation . . . 4
- 1.5 Getting Started . . . 5

2 Sail Coordinates **7**
- 2.1 Function Overview . . . 7
- 2.2 Cone and Clock Angles . . . 8
- 2.3 Visualization . . . 14
- 2.4 Gimballed Boom Coordinates . . . 17

3 Building a Sail Model **19**
- 3.1 Function overview . . . 19
- 3.2 Creating a sail mesh . . . 21
- 3.3 Defining the sail components . . . 25
- 3.4 Storing and retrieving sail models . . . 27
- 3.5 Sail configurations . . . 30
  - 3.5.1 Flat Plate . . . 31
  - 3.5.2 Sails with flat components . . . 31
  - 3.5.3 Striped Sail . . . 32

4 Disturbances **35**
- 4.1 Function Overview . . . 35
- 4.2 Solar pressure force function . . . 36
- 4.3 Environment Function . . . 37
- 4.4 Disturbance Function . . . 39
- 4.5 Profile Data Structure . . . 39
- 4.6 `SailDisturbance` Demo . . . 40

5 Attitude Dynamics **45**
- 5.1 Function Overview . . . 45
- 5.2 Rigid Body Dynamics . . . 46
- 5.3 General Two-Body Dynamics . . . 46

5.4 Fixed Rate Rotating and Translating Bodies 46
5.5 Time Varying Inertia 48
5.6 Special Two-Gimbal Model for a Boom 48
5.6.1 Dynamical Equations 48
5.6.2 Two Body Functions 54
5.6.3 Example 56

6 Sail Attitude Actuators **57**
6.1 Sliding Masses 57
6.2 Vanes 59
6.3 Gimballed Boom 61

7 Orbit Dynamics and Ephemeris **65**
7.1 Function Overview 65
7.2 Orbit Dynamics 66
7.2.1 Combined right-hand-side 67
7.2.2 Specialized Coordinate Systems 69
7.3 Ephemeris 69

8 Analysis **73**
8.1 Creating a CAD Model 73
8.2 Performing a Disturbance Analysis 76
8.3 Simulating the Attitude Dynamics 78
8.4 Boom Control Demo 80
8.5 Heliopause Guidance Mission Demo 82
8.6 Integrated Guidance and Attitude Control 84

9 Trajectory Optimization **87**
9.1 Introduction 87
9.2 Local Control 87
9.3 Global Control 90
9.3.1 Global Methods 90
9.3.2 Function Overview 92
9.3.3 Formulation of the Problem 94
9.3.4 Zermelo's Problem 95
9.3.5 The Three Dimensional Equations of Motion 97
9.3.6 Low-thrust Mars Rendezvous 99
9.3.7 Sail 2D Optimization Examples 101
9.3.8 Heliopause Mission 104

CHAPTER 1

INTRODUCTION

This chapter describes the Solar Sail Module , shows you how to install it, and explains how it is organized.

1.1 Solar Sails

Solar sails are a class of spacecraft characterized by large, reflective surfaces that are used as a means of propulsion. This module contains specialized functions to model sails, including example sail CAD models, propulsive modeling via optical and thermal forces, sail coordinate transformations and guidance laws, and simulations. A major premise in this module is that solar sails can be modeled the same way as other spacecraft are in the Spacecraft Control Toolbox, namely, as a mesh with disturbances computed on each triangle during simulations. In this case, the disturbances are also the propulsion system, but the methods of computing them are the same. In the simplest case, a sail might be a single area element, but in a complex case, a sail can be a combination of draped and billowed membranes in any configuration. Orbit and attitude dynamics are also the same as any other spacecraft, but this module contains specific examples of proposed systems, such as rotating vanes, sliding masses, and gimballed booms.

The plots in Figure 1.2 on the following page provide an orientation to the sail accelerations produced by different sail sizes and total spacecraft mass. A near-term sail is considered to be about 60 m square, producing an acceleration on the order of 0.1-0.3 mm/s^2. A mid-range sail size is 100-150 m square allowing for accelerations on the order of 0.5-1 mm/s^2 for spacecraft of about 100-300 kg. Sailcraft with accelerations over 1 mm/s^2 are far-term, requiring additional advances in sail materials and construction. There are functions in the Utilities folder which convert between some common representations of the propulsive capability of a sail, such as *characteristic acceleration*, the acceleration produced by an ideal sail of a particular size and mass at 1 AU, *lightness*, a dimensionless parameter, and *loading*, the mass per area. The built-in demo of `AccelToSailProps` produces the plot in Figure 1.2 on the next page.

The source material for this module includes recent textbooks and conference and journal articles

Figure 1.1: Examples of square and bladed sail designs

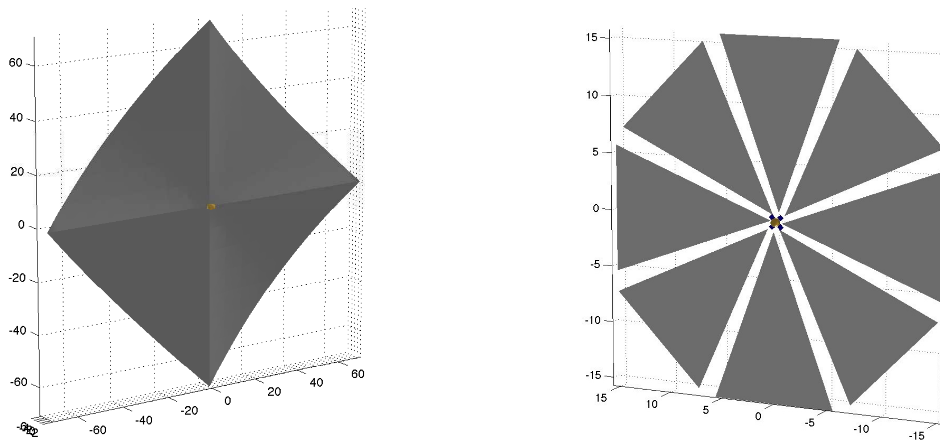
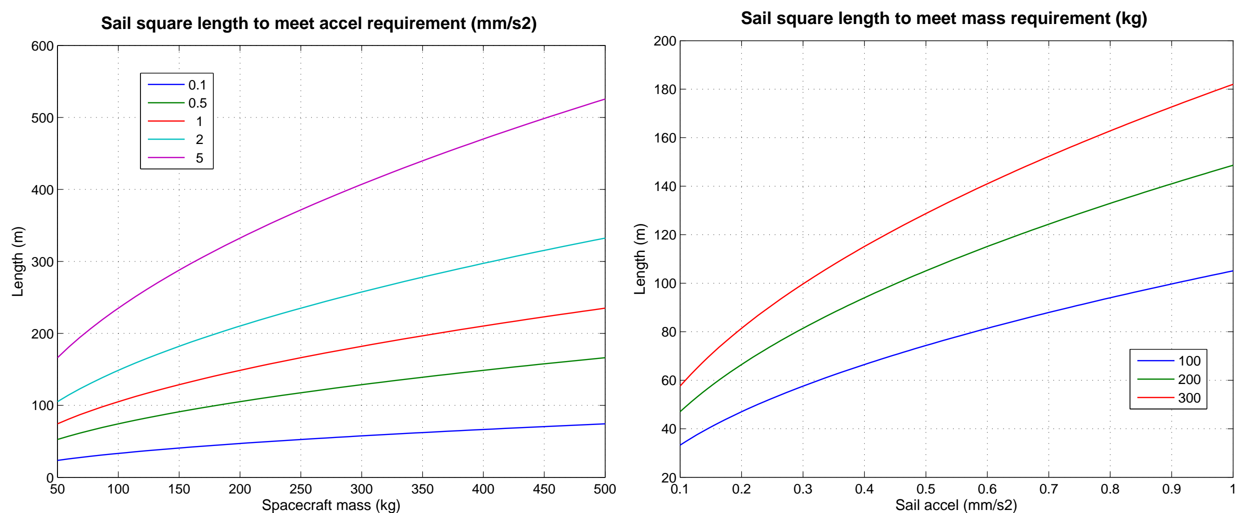


Figure 1.2: Sail size for acceleration



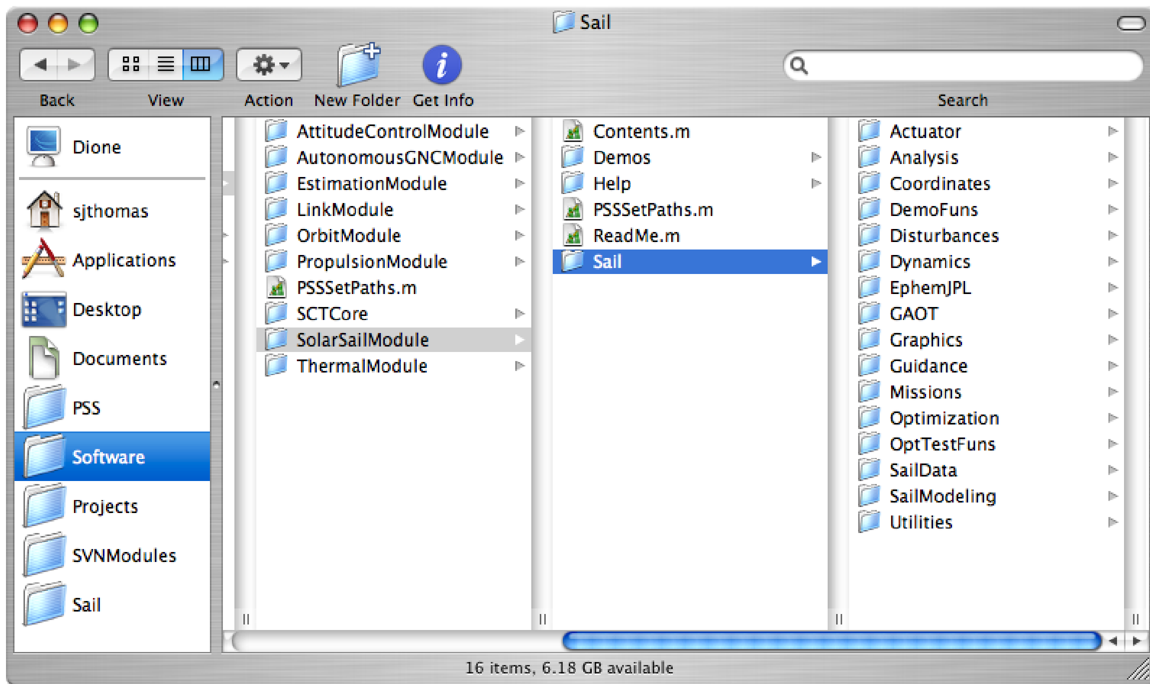
studying various aspects of sail design and potential missions enabled by this technology. This body of literature is constantly growing and if there is a particular example or mission you would like to see in the Solar Sail Module, please let us know!

1.2 Organization

The Solar Sail Module is organized into a number of folders as shown in Figure 1.3 on the facing page. Each of these folders contains function files. Most of these folders also have corresponding folders in the Demos folder which contain scripts that demonstrate how to use the functions to perform different types of analyses.

If you type `help Sail` at the command line, you will get a list of folders in your version of the Solar Sail Module.

Figure 1.3: Solar Sail Module on Mac OS X




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

  Directories:
  Actuator
  Analysis
  AttitudeDynamics
  Control
  Coordinates
  DemoFuns
  Demos
  Demos/Actuator
  Demos/Control
  Demos/Disturbances
  Demos/Dynamics
  Demos/EphemJPL
  Demos/Guidance
  Demos/Integrated
  Demos/Missions
  Demos/Optimization
  Demos/SailDesigns
  Demos/SailModeling
  Disturbances
  Dynamics
```

```
GAOT
Graphics
Guidance
Help
Missions
OptTestFuns
Optimization
OrbitDynamics
SailData
SailEphem
SailModeling
Utilities
```

1.3 Requirements

The Professional Edition of the Spacecraft Control Toolbox is required to use this module. The toolbox requires MATLAB 7.x for full functionality. Most functions will also run in earlier versions of MATLAB.

1.4 Installation

The Solar Sail Module is designed to be used with the Spacecraft Control Toolbox (SCT). You should already have the SCT installed on your computer, or this Solar Sail Module should have been installed with your complete SCT package. If you are adding this module to your PSS toolboxes or updating it, then please follow these instructions.

If you have a CD, copy the Solar Sail Module folder for your operating system from the CD into your PSS Toolboxes software folder. The Solar Sail Module should be at the same level as your other modules such as the Common and SC module folders, as shown in Figure 1.3 on the previous page. You can copy the PDF documentation anywhere you wish. If you downloaded your product from the Princeton Satellite Systems website, put the folder extracted from the archive in your PSS Toolboxes software folder. There is no "installer" application to do the copying for you. All you need to do now is to set the MATLAB path to include the folders in the Solar Sail Module.

We recommend using the supplied function `PSSSetPaths.m` instead of MATLAB's path utility. From the MATLAB prompt, cd to your PSS Toolboxes 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. You can set the path to include PSSToolboxes permanently by opening MATLAB's path dialog and saving the current path or by

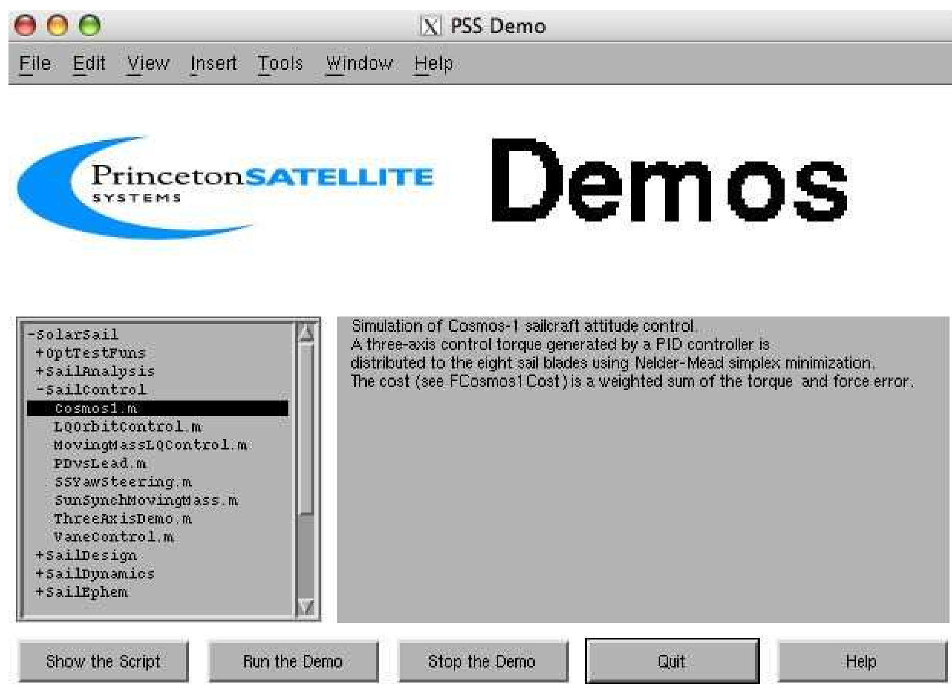
using the function path2rc. You can also add Modules to your path one at a time by using the copy of PSSSetPaths inside the Module folders.

1.5 Getting Started

The first two functions that you should try are DemoPSS and FileHelp. These are generic to all PSS toolboxes and modules and they provide the best way to get an overview of your new software's capabilities.

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, illustrated by Figure 1.4.

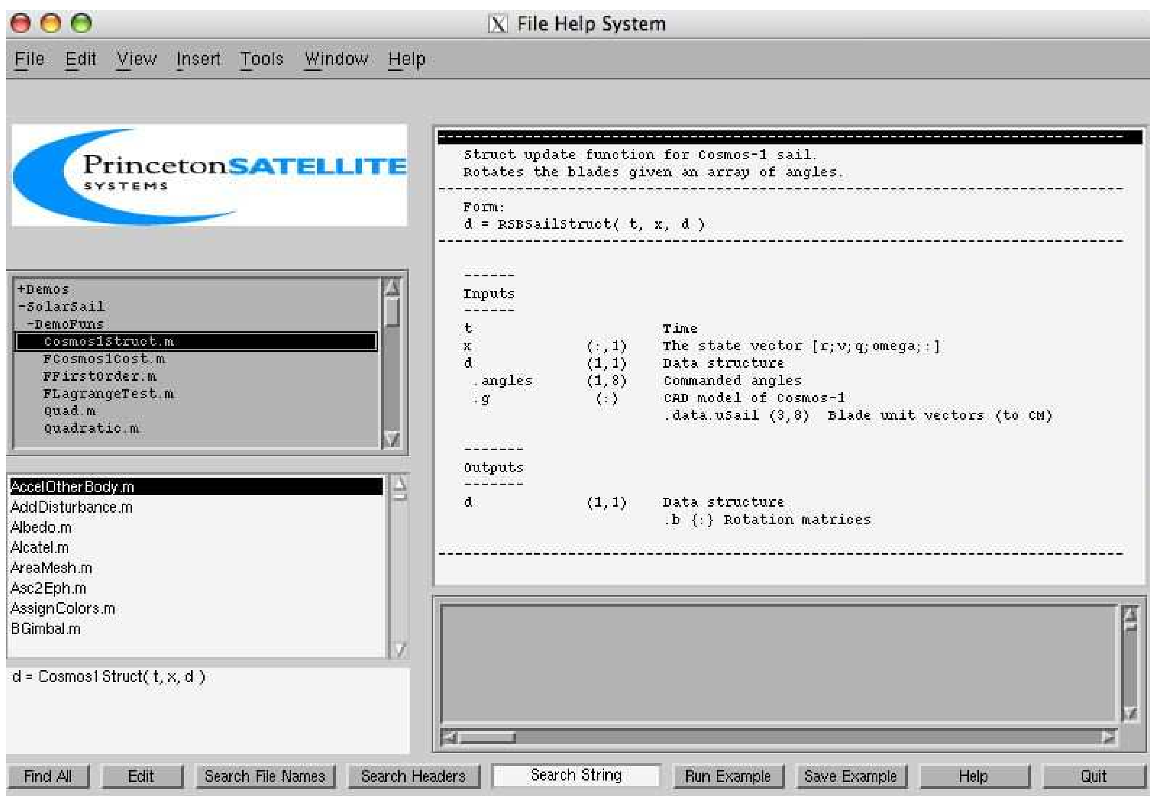
Figure 1.4: The DemoPSS GUI



The FileHelp function provides a graphical interface to the MATLAB function headers, as shown in Figure 1.5 on the next page. 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 is discussed further in the main SCT User's Guide.

Alternatively, you may now browse the module's functions and demos using MATLAB's help browser.

Figure 1.5: The File Help GUI



CHAPTER 2

SAIL COORDINATES

This chapter shows you how to use Solar Sail Module functions for commonly needed coordinate transformations.

2.1 Function Overview

The Coordinates folder contains functions for describing sail attitudes as well as orbit coordinates useful for trajectory optimization routines. Use the `help` function with the folder name to get the list of available functions with a brief description.

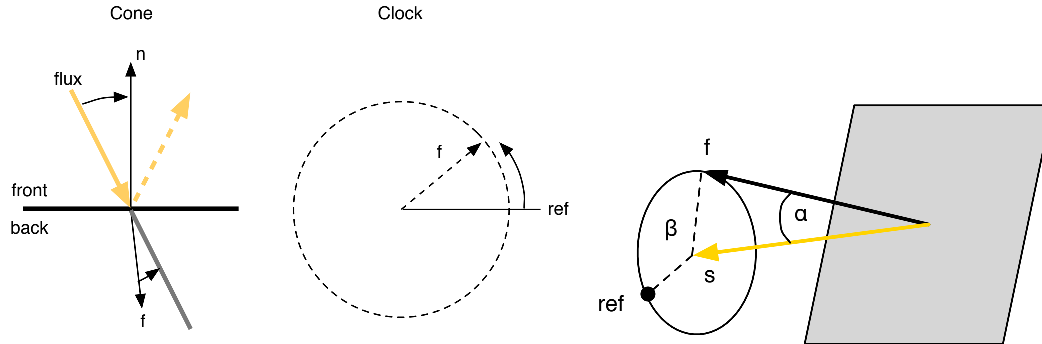
```
>> help Coordinates
  Sail/Coordinates

  A
     AttitudeProfileToTorque     - Compute an equivalent torque sequence
                                 from an attitude profile.

  B
     BConeClock                  - Compute a rotation matrix for cone and
                                 clock from the reference frame

  C
     CartToI                     - Computes inclination from cartesian
                                 elements
     ClockConversion             - Apply a clock angle convention between
                                 McInnes, PSS, and JPL.
     ConeClockConvention         - Apply a selected cone/clock sign
                                 convention for angle ranges.
     ConeClockToQConstrained     - Compute a quaternion from cone and clock
                                  angles.
     ConeClockToU                - Compute a unit vector from cone and
                                 clock angles.

  E
```

```
   ElToMEq                    - Transforms modified Keplerian elements
                              to equinoctial elements.

M
   MEQToECI                   - Transform to ECI frame from tangential
                              coordinates.
   MEQToI                     - Compute inclination from modified
                              equinoctial coordinates.
   MEqToEl                    - Transforms modified equinoctial elements
                               to Keplerian elements.
   MEqToRV                    - Transforms modified equinoctial elements
                               to r and v.

P
   PlanetRot                  - Transform from ECI to planet-fixed axes.

Q
   QSail                      - Quaternion from inertial to the local
                              rotating sun-sail frame.
   QToConeClock               - Computes cone and clock angles from an
                              inertial to body quaternion.
   QXToAlphaDelta             - Cone and clock angles (alpha,delta) to
                              inertial quaternion.

R
   RVToMEq                    - Transforms elements r and v to  modified
                               equinoctial.

S
   SteeringAnglesToQ          - Convert sail steering angles to a
                              quaternion.

U
   UToConeClock               - Computes cone and clock angles from a
                              unit vector.
```

2.2 Cone and Clock Angles

Solar sails generate thrust due to optical reflection. Therefore, solar sails must be pointed within a fairly narrow angular range of the sun vector to produce thrust. The thrust direction is opposite the surface normal facing the incoming flux. The angle between the thrust vector and the sun vector is termed the cone angle; due to the cosine rule of optical forces, this angle determine the force produced by the sail. The cone angle must be 90 degrees or less. In 3D, the cone angle literally describes a cone about the sun vector where the sail might be pointed. The angle locating the thrust around this cone is called the clock angle, and it ranges from 0 to 360 degrees. The clock angle has to be defined from some reference point on the cone. The definition of the reference direction varies widely in the sail literature but it may be in the osculating orbit or inertially defined. Figure 2.1 on the facing page shows the two angles separately and then together with a notional square sail.

Figure 2.1: Sail Cone and Clock Angle Diagram



For a perfectly specular sail, the thrust vector is always aligned with the sail normal out the back of the sail. For a realistic sail, there might be a substantial angle between these vectors due to nonideal optical properties and a nonflat sail shape. Therefore, for a real sail, we need two sets of cone and clock angles to describe the situation: the steering set which describe the sail attitude, and the actual set which describe the resulting force direction. Note that the cone angle of the attitude may be more properly termed the incidence angle. Nonideal optical properties result in a diffuse component which cases the thrust to have a smaller cone angle than the sail incidence angle. The angle between the force vector and the surface normal is called the centerline angle.

The most relevant functions for cone and clock angles in the Solar Sail Module are:

- `ClockConversion`
- `ConeClockConvention`
- `ConeClockToU`
- `QSail`
- `SteeringAnglesToQ`
- `UToConeClock`

The cone angle can only be computed one way, through a simple dot product. The symbol α is widely used for this angle. PSS has identified several major conventions for the clock angle to date, namely:

McInnes McInnes[**?**] measures clock (δ) from the osculating orbit normal. The sail normal n is defined to be pointing out the back of the sail, i.e. away from the sun.

$$
\begin{aligned}
\hat{p} &= \hat{r} \times \hat{v} \\
n_b &= \cos\alpha\,\hat{r} + \sin\alpha\cos\delta\,\hat{p} + \sin\alpha\sin\delta\,\hat{p} \times \hat{r}
\end{aligned}
$$

PSS In a planet-centric orbit, it makes sense to continue to define the angles relative to the sun vector, which is no longer coincident with the position vector. The orbit normal will also no longer be perpendicular to the sun line. Therefore we define a sun-pointing frame where the

clock angle (β) is measured from the cross product of the orbit normal and the vector to the sun. If the orbit is heliocentric and the vector s is taken to be from the sun towards the sail, then this coincides with McInnes' description.

$$\begin{aligned}
\hat{x} &= \hat{s} \\
\hat{y} &= (\hat{r} \times \hat{v}) \times \hat{x} \\
\hat{z} &= \hat{x} \times \hat{y} \\
n &= \cos\alpha\ \hat{s} + \sin\alpha\cos\beta\ \hat{y} + \sin\alpha\sin\beta\ \hat{z}
\end{aligned}$$

Alternatively, the velocity vector alone may be used to define the clock reference direction. This ensures that the y vector is always in the same hemisphere as the velocity vector. This frame is convenient for analyzing the force produced by the sail and its relationship to the trajectory.

$$\begin{aligned}
\hat{x} &= \hat{s} \\
\hat{z} &= \hat{x} \times \hat{v} \\
\hat{y} &= \hat{z} \times \hat{x}
\end{aligned}$$

In both cases above, the clock angle is measured from the reference y axis.

JPL JPL[**?**] uses a convention measuring clock from ecliptic north. This is intended only for heliocentric orbits.

Each of these conventions will result in different clock angle profiles for certain common guidance situations such as orbit raising, where the sail thrust vector has a component along the velocity vector. PSS prefers our own description since it works equally well in a heliocentric or planet-centric orbit. Hence, PSS' coordinate transformation functions use this convention, although a function `ClockConversion` also exists to convert between the three.

In Figure 2.2 on the next page, the cone angle is α and the clock angle is β, and the angles are used to define the sail forward normal vector $\hat{n}_f$. In a heliocentric frame z will always be coincident with the orbit normal.

Clock angles can be converted between formats using `ClockConversion`. The formats are numbered in the order they are given above: 1 for McInnes, 2 for PSS, 3 for JPL. Orbital and sun vectors are needed for the conversion, and they are provided in the data structure `d`. The vectors can be in either the ECI or the ecliptic frame as specified with a flag. The syntax is

```
clockNew = ClockConversion( cone, clock, fromConv, toConv, d )
```

and a built-in demo comparing the conventions is shown in Figure 2.3 on the facing page. The orbit is heliocentric with an inclination of 1 radian.

Quaternions are used for attitude representation in most PSS simulation functions. PSS defines a rotating sail frame using `QSail`, which returns the inertial-to-reference quaternion. The x axis can be either along or opposite the vector to the sun, with a flag used to set the sign convention.

Figure 2.2: Sail Cone and Clock Angle Diagram

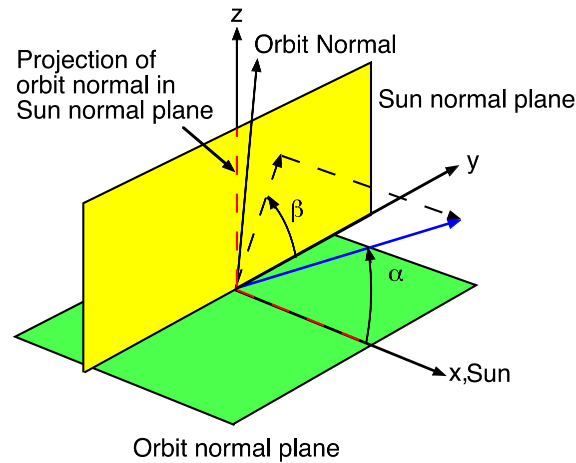
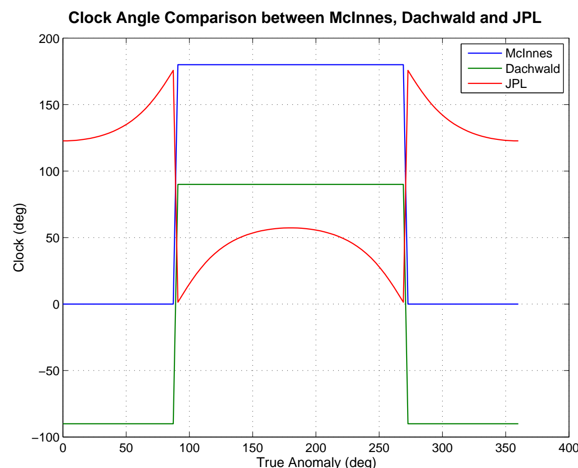


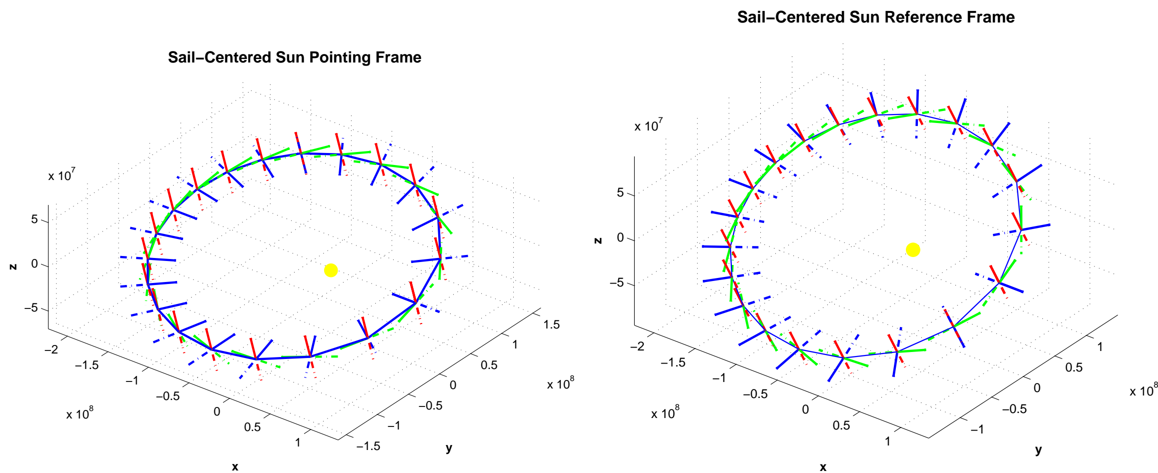
Figure 2.3: ClockConversion built-in demo



The y and z axes are defined using either the orbit normal or the velocity vector as a reference as described above. This function is comparable to `QLVLH` from the Spacecraft Control Toolbox, which defines a local vertical/local horizontal frame. `QSail` has a built-in demo which computes the quaternion for a 1 AU heliocentric orbit inclined to 0.5 radians, producing the plot shown in Figure 2.4 on the next page. As an example,

```
>> s = SunV1(2451545)

s =

    0.1801
   -0.9025
   -0.3913

>> r = [7000;0;0];
>> v = [0;7.5461;0];
```

Figure 2.4: QSail and QSunSail built-in demos. On the left, the x-axis points towards the sun and the z-axis towards the orbit normal. On the right, the x-axis points away from the sun, and the y-axis is towards the velocity vector.




```
>>  q = QSail( s, r, v )

q =

    0.7576
   -0.1266
   -0.1544
    0.6214
```

A complete attitude description will also require a quaternion which rotates from the sail reference frame to the given cone and clock angles and a quaternion which rotates from these axes to the body frame. We define the following frames for clarity:

Body frame Frame attached to the sail, for example with x as the front normal

Sail-Sun frame Also called simply the sail frame or the reference frame, this is the frame defined relative to the orbit and the sun vector from which cone and clock angles are measured.

Cone-Clock frame This is the frame rotated from the sail frame by the cone and clock angles

Cone and clock angles can be computed from unit vectors using `UToConeClock`, and `ConeClockToU` transforms in the opposite direction. `SteeringAnglesToQ` computes an inertial-to-body quaternion assuming that the sail front is along $+x$ in the body frame and that the body y axis is aligned with the clock angle. Note that without this second assumption or something similar, there are infinite ways the sail could be rotated to achieve the desired pointing, and we would need yet another quaternion to fully define the sail attitude with respect to an inertial frame. A specific control scheme will require a specific pointing function.

For example, the following lines show an example using the sun-pointing (positive) sign convention.

```
[r,v] = El2RV( [Constant('au') 0.5 0 0 0 0], [], Constant('mu_sun
    ') );
s     = -Unit(r);
cone  = 0.5;
clock = pi/2;

[u,qItoCC] = ConeClockToU( cone, clock, r, v, s )
[cone, clock] = UToConeClock( u, r, v, s )
[cone, clock] = QToConeClock( qItoCC, r, v, s )
```

This confirms that the cone and clock angles are in fact recovered, and the same is true for any additional rotation of the sail body about its x axis,

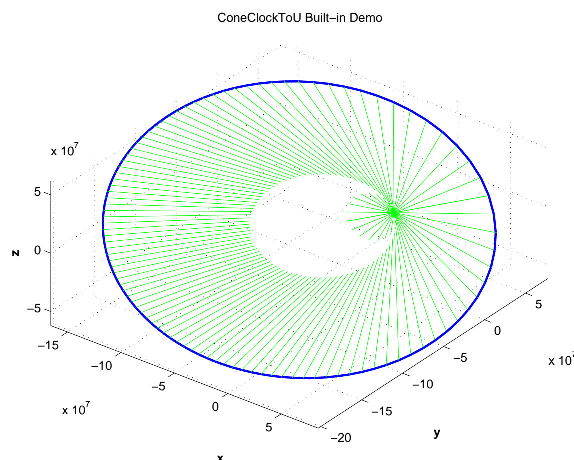
```
qSB = Eul2Q([1.2;0;0]);
qIB = QMult( qItoCC, qSB );
[cone, clock] = QToConeClock( qIB, r, v, s )
```

for which the output remains

```
cone =
          0.5
clock =
      1.5708
```

`ConeClockToU` also has a built-in demo which draws the sail vector u for zero cone and clock angle for an eccentric heliocentric orbit.

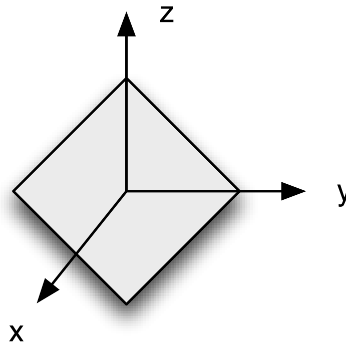
Figure 2.5: ConeClockToU built-in demo



2.3 Visualization

The CAD models present in the toolbox generally use the following frame for a square sail: $+x$ is the forward normal of the sail (towards the sun), $+y$ is along a diagonal of the square, and $+z$ completes the set, as in Figure 2.6.

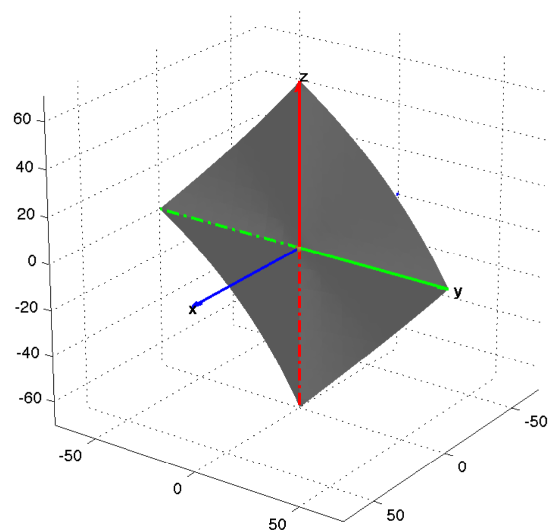
Figure 2.6: Square sail body frame



However, a CAD model may have another orientation, depending on how it was created. To view a stored CAD model, load it into a data structure and pass it into `DrawSCPlanPlugIn`. Example 2.1 shows the simple code required to do this.

Example 2.1 Visualize a CAD model with body axes

```
g = load('QuadSail_100.mat');
DrawSCPlanPlugIn( 'initialize',
g );
AddAxes(g.radius,[],[],gcf);
```



The Solar Sail Module has a number of functions that are helpful in visualizing sail attitudes, as shown in Figure 2.7 on the facing page, Figure 2.8 on the next page. Some include a no-

tional square sail, some include an actual CAD model, and other just show angles. The demo `SailForcePlots` uses several of these functions with a consistent set of attitude data.

Figure 2.7: `DrawSailAttitude`: Draw the attitude of a sail CAD model relative to the sun vector

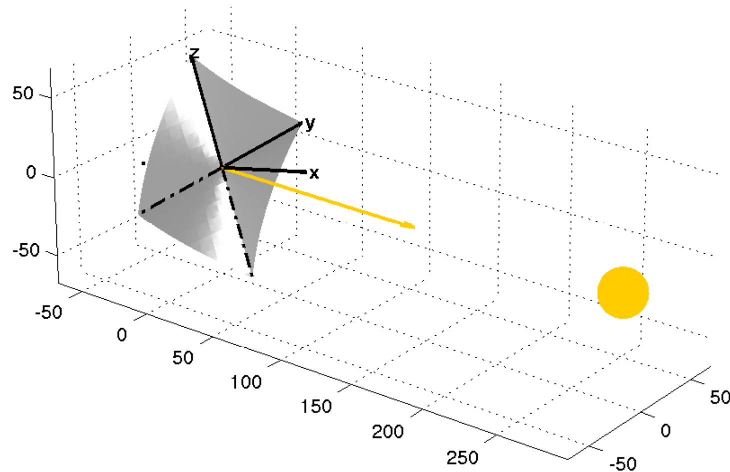


Figure 2.8: `PlotSailForce2D` and `PlotSailClock2D`: 2D plots of the sail force and normal vectors

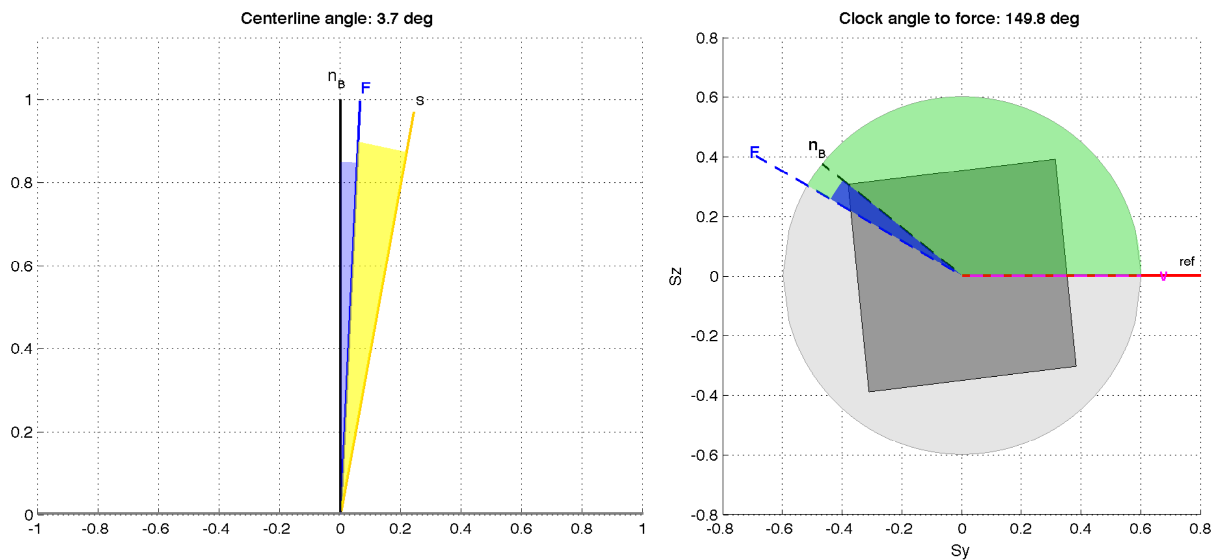


Figure 2.9: `VisualizeSailAttitude`: 3D plots of a notional sail and steering angles. The front or the back of the sail can be distinguished by shade when rotating.

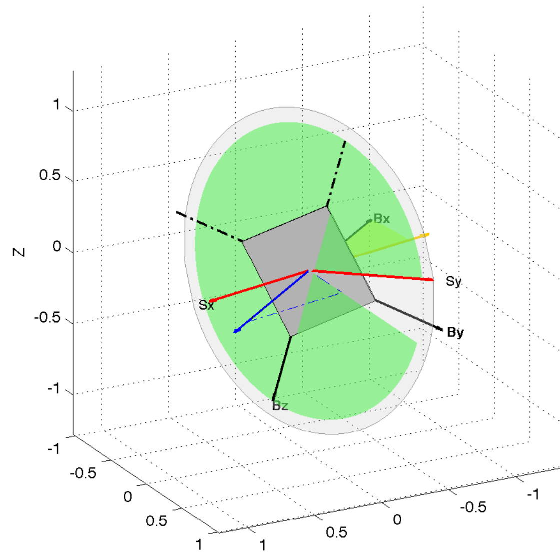
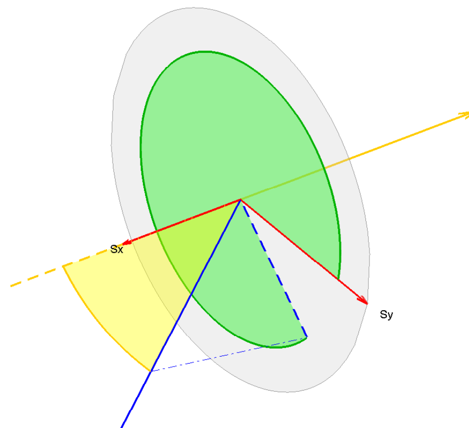


Figure 2.10: `DrawSailAngles`: Simplified 3D view without the sail.



2.4 Gimballed Boom Coordinates

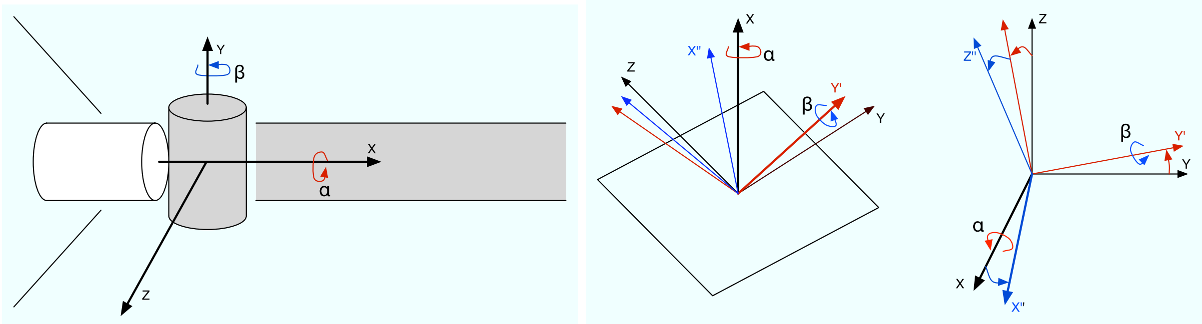
A common configuration in sails is the presence of a gimballed boom. The boom will have two gimbals which correspond to some sequence of rotations. PSS uses a convention of 1-2 angles for this configuration, as shown in Figure 2.11. See for example `GimbalRates`, which explicitly assumes this configuration. The function `HingeRotationMatrix` can compute transformation matrices for any combination of single axis rotations. In the case of the 1-2 boom gimbals, the axis vectors are

```
>> axis      = [1 0;0 1;0 0]
axis =
     1     0
     0     1
     0     0
```

The rotations around these axes will transform from the unrotated to the rotated frame. For example,

```
 >> angle = [pi/2 0.1];
>> bT    = HingeRotationMatrix( angle, axis )
bT =
        0.995      0.099833   -2.2167e-17
            0    2.2204e-16             1
     0.099833        -0.995    2.2094e-16
```

Figure 2.11: Gimballed boom frame



CHAPTER 3

BUILDING A SAIL MODEL

This chapter discusses the functions available to help create sail models and describes the examples included in the Solar Sail Module.

3.1 Function overview

The SailModeling folder contains functions for modeling sail materials (CP1), sail shape, and sail deployment.

```
>> help SailModeling
  Sail/SailModeling

  C
     CP1Props                         - Front and back optical and thermal
                                   properties of CP1 (Lambertian)
     ComputeSailNormal        - Find all sail components and calculate their
                                    composite normal.

  H
     HCircularBillow                  - Circular billow height model
     HQuadrantBillow          - Models a single sail quadrant as a section
                                   of an oblique cone.
     HRakoczy                         - Height function for a square sail
                                   with billow.

  I
     IDotS4                           - Inertia derivative of S4 sail.

  M
     MakeSquareSail           - Generate a flat, square, nonideal sail in
                                   the Y/Z plane, using CP1 properties

  S
     S4DeployTorque                   - Disturbances function for modeling
                                   deployment of scalable square sail.
```

```
   SailMesh                          - Create a mesh with the height
                                  determined by a specified function.
   StripedQuadrant                   - Create a striped sail quadrant,
                                  with billow, in the x, y plane.

Sail/Demos/SailModeling

S
   S4Deployment                      - S4 (ATK's scalable sail) deployment
                                   demo.
   SailMassAndArea                   - Sail dimensions as a function of
                                  payload mass
```

The completed sail designs are in SailDesigns.

```
>> help SailDesigns
  Demos/SailDesigns

  B
     BilloweDSquareSail    - A billowed, square, nonideal sail in the Y/Z
                             plane.

  C
     CircularSail          - Design a circular nonideal sail with billow
                             using SailMesh.
      Cosmos1               - CAD model of the Cosmos-1 solar sail.

  E
     ECHOModel             - A specular spherical sail, i.e. ECHO-2

  F
     FlatCP1Sail           - A flat, square, nonideal sail in the Y/Z
                             plane, using CP1 properties
     FlatPlate             - A flat, square, specular sail in the Y/Z
                             plane.

  P
     PlateWithBoom         - Design a gimbaled boom specular sail model
                             with two bodies.
     PlateWithBoomAndVanes - Design a specular sail model with a control
                             boom and vanes.
     PlateWithMasses       - Design a specular (plate) sail model with two
                              transverse control masses.
     PlateWithVanes        - Design a specular (plate) sail model with two
                              control vanes.

  Q
     QuadBillowedSail      - A billowed quadrant sail demonstrating
                             SailMesh. Uses CP1 properties.

  S
     S4Deploy              - 40 m Scalable Sail, for deployment analysis.
     SailWithBoom          - Design a gimbaled boom sail model with two
                             bodies.
     SailWithFourVanes     - Design a nonideal sail model with four
```

```
                              control vanes.
     SquareGEOSail          - A flat, specular sail for GEO simulations.
     StripedSail            - Design a square sail with four striped
                              quadrants. Uses CP1.
```

3.2 Creating a sail mesh

As mentioned in other chapters, a main premise of this toolbox is that a sail of complex shape can be described as a mesh, similarly to any other spacecraft, and disturbances computed on each triangle of the mesh. This is our approach to modeling sails. The Solar Sail Module provides several function to assist users in creating meshes for different sail designs. Common problems are describing a billow, similar to the sail of a boat when in the wind, and stripes of draped material. Main issues when defining a sail this way include accurately computing the resulting inertia, for attitude control purposes, and the optical, if they may be distributed.

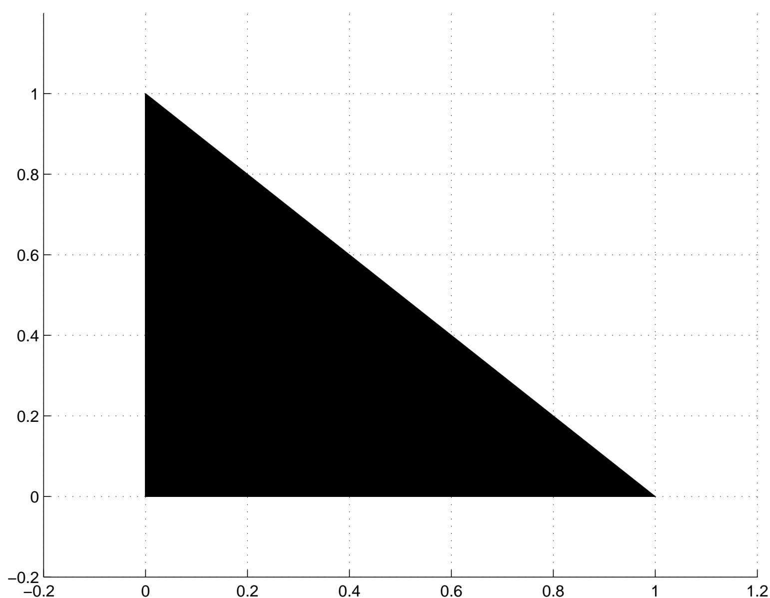
The CAD functions provide a sail component class that is recognized by the disturbances function so that the special combined thermal/optical model for membranes is used. This class requires that the sail shape be defined using vertices and faces. This is the way MATLAB defines patch objects in figures, see the help for `patch` for more information. Each vertex is a location in 3D space, and each face is defined by a set of vertices, which when connected in order make a polygon. For instance, to define a triangle, we will have three vertices and one face. The vertices must be in consecutive rows. For a triangle in the X/Y plane simply type:

```
v = [ [0,0,0]; [1,0,0]; [0,1,0] ];
f = [1 2 3];
patch('vertices',v,'faces',f)
axis([-.2 1.2 -.2 1.2]); grid on
```

and you will see the triangle in Figure 3.1 on the next page. This is really the basis of all the sail designs in Solar Sail Module. A square or bladed sail can be made from triangles in this way. Related functions available in the Spacecraft Control Toolbox are `Panel`, which defines a flat circumferential shape (like a square or hexagon), `PanelWithCenterHole`, and `PanelWithCutout`. These can create a flat triangular mesh but without further subdivision, so they are good for modeling flat sails or sail components of different shapes. For example, to make a flat panel with a cutout, we can copy the functions built-in demo but pass in a thickness of zero.

```
xW      = 10;
yW      = 20;
t       = 0;
c.x     = -1;
c.y     = 1;
c.r     = 3;
c.n     = 6;
[v, f, area] = PanelWithCutout( xW, yW, t, c )

v =
      1.1213       3.1213             0
     -1.7765       3.8978             0
```

Figure 3.1: Simple triangle patch

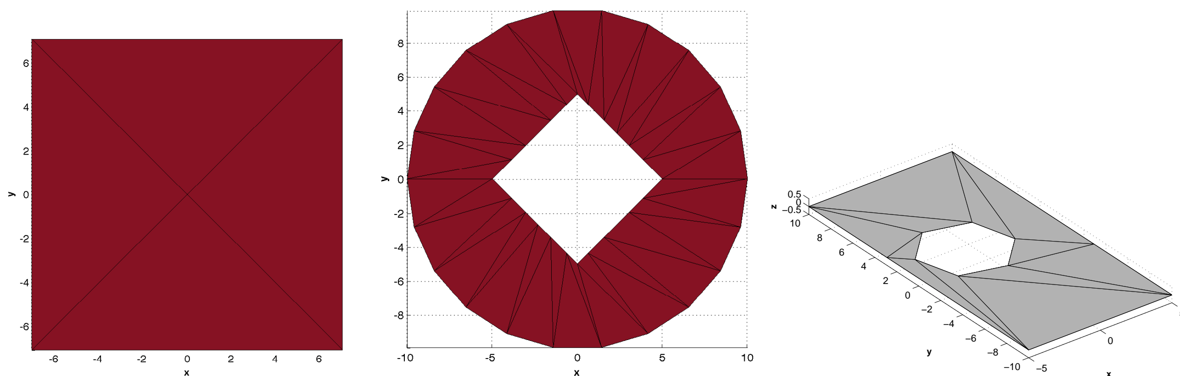



```
      -3.8978       1.7765            0
      -3.1213      -1.1213            0
     -0.22354      -1.8978            0
       1.8978      0.22354            0
            5           10            0
           -5           10            0
           -5       2.8937            0
           -5          -10            0
            5          -10            0
            5      -2.8937            0
f =
     1     7     8
     2     8     9
     3     9    10
     4    10    11
     5    11    12
     6    12     7
     1     8     2
     2     9     3
     3    10     4
     4    11     5
     5    12     6
     6     7     1
area =
       176.62
```

A more complex shape with displacement out of the plane requires a more refined mesh. This is where `SailMesh` comes in.

```
>> help SailMesh
 ---------------------------------------------------------------------------
```

Figure 3.2: Simple panel meshes: `Panel`, `PanelWithCenterHole`, and `PanelWithCutout`



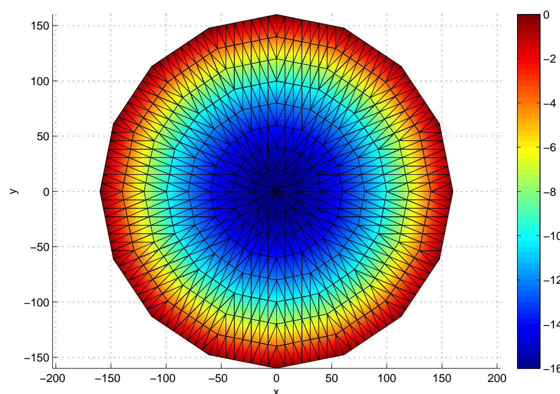
```
   Create a mesh with the height determined by a specified
      function.
   This can be used to model a sail shape such as billow. The x
      and y vectors
   should decribe the circumference of the sail. The origin is
      added so the
   mesh resembles a refined pinwheel.

   The height function is of the form:   HMesh( x, y, d )
   where x and y can be arrays. d is a data structure containing
      any
   user-defined fields. A function 'DefaultH' is provided for
      testing.

   The built-in demo creates a circular sail and a quadrant.
--------------------------------------------------------------------------

   Form:
   [v, f] = SailMesh( x, y, hFunc, d, nRefine )
--------------------------------------------------------------------------



   ------
   Inputs
   ------
   x          (1,m)     x locations of mesh
   y          (1,m)     y locations of mesh
   hFunc       (:)      Function for h positions (string or handle)
   d           (:)      Data structure to pass to z function
   nRefine    (1,1)     Number of times to refine mesh. Default is
      once.
```

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

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

As you can see, `SailMesh` returns the vertices and faces for a sail. This function allows you to specify a function describing a vertical displacement for a nonflat sail. You can also use a function to define x and y as other than a flat line. The two main examples of this function are `CircularSail` and `QuadBillowedSail`.

For the circular sail example, x and y are computed for a circle of the desired radius using 16 points along the circumference. The function `HCircularBillow` provides an out-of-plane displacement based on a slope b at the outer edge and the sail radius r. We specify that the mesh should be refined, that is each triangle refined into four more, three times. The code to compute the mesh is only a few lines. To get a figure of the resulting mesh, call `SailMesh` again without any outputs. The sail mesh will be colored based on the magnitude of the out-of-plane displacement. Notice that the units of the model is meters.

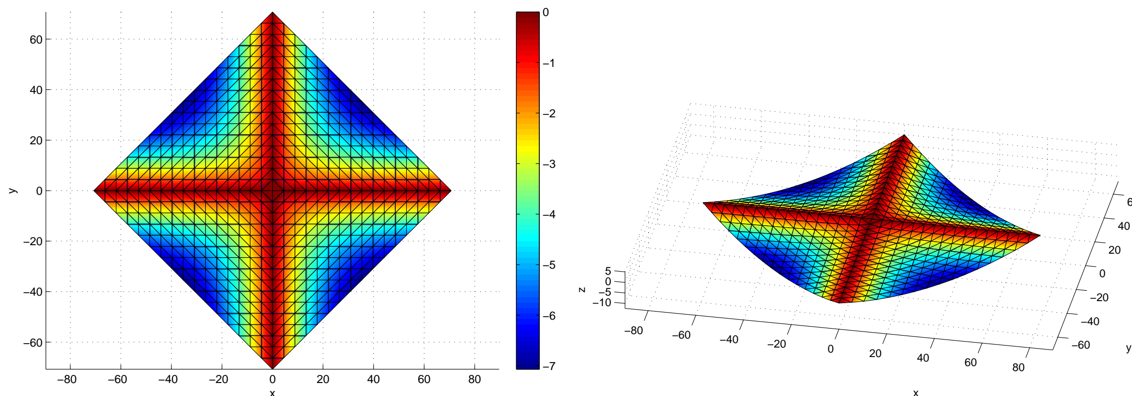
```
% Get sail shape
theta = [0:15]*pi/8;
rSail = 160;  % m
x = rSail*cos(theta);
y = rSail*sin(theta);

dBillow = struct('b',0.2,'r',rSail);
[v,f] = SailMesh( x, y, 'HCircularBillow', dBillow, 3 );
```



The second example is a square sail supported by booms such that each of four quadrants has a smooth billow, with zero displacement along the booms. In this case the x and y inputs to

`SailMesh` define the outer corners of the square. We rotate the square by 45 degrees so that each boom is along an axis in the cody frame. The function `HQuadrantBillow` defines the billow for a quadrant based on a maximum billow depth d at the outer edge of the sail, specified as a fraction of the boom length, and the length of the boom L. Again, to get a figure of the resulting mesh, call the function again with no outputs.

```
% Sail
%-----
sailWidth      = 100;
x = sailWidth/2*[1 -1 -1 1];
y = sailWidth/2*[1 1 -1 -1];
B = sin(pi/4)*[1 1; -1 1];
p = B*[x;y];
lQuadrant = sailWidth/sqrt(2);
billow = struct('L',lQuadrant,'b',0.1);
[v,f] = SailMesh( p(1,:), p(2,:), 'HQuadrantBillow', billow, 4 );
```



3.3 Defining the sail components

Once you have the vertices and faces for the sail, you are ready to make a component. The CAD component requires optical and mass properties as well as the geometric shape data, and you have to rotate it to the right place in the body frame. Components are usually defined in a default frame, such as in the X/Y plane. In addition, for a *sail* class component, you have to provide both front and back optical properties, as needed for the membrane disturbance function. If you want to model an ideal specular surface, you can do so provided the emissivity is nonzero, even if it is the same for both the front and back.

The `CreateComponent` function is described in detail in the CAD chapter of the Spacecraft Control Toolbox user's guide, and we will review it here briefly. You pass in a variety of parameters, and the function fills in all the default fields and creates a standard format for the component. You select a type of component using the 'make' action, and in this case we are making a *sail* component. You give the component a name and specify which body it attached to. Recall that

bodies may be independently rotated once the model is complete. You provide the mass properties in a data structure, as well as the optical properties sigma* and in the case of a *sail* component, emissivity as well, to complete the membrane description. You should specify this component as inside equal to 0; any small components that will have a negligible effect on the disturbances, or components that are actually inside a bus and not external to the spacecraft, would have a 1 here. You can specify a face color for the component, which affects how it displays within Matlab figures, but it not used for actual disturbances computation. Lastly, you specify the location of the component using three parameters: rA, rB, and b, which specify a displacement before and after a rotation by matrix `b`. For a component without additional rotation you can just specify rA. So here is a brief example from `SailWithBoom`:

```
v = [0 0 0 0;0.5 -0.5 -0.5 0.5;0.5 0.5 -0.5 -0.5]'*sailWidth;
m = CreateComponent( 'make', 'sail','name','Sail','body',1,...
                     'mass', massSail, 'faceColor', 'mirror','rA'
                        ,[-coreWidth/2;0;0],...
                     'sigmaS', [0.9 0.85], 'sigmaD', [0.02 0.05],
                        'sigmaA', [0.08 0.1], 'emissivity', [0.03,
                         0.3],...
                     'vertex',v ,'face', [1 2 3; 1 3 4], 'inside',
                         0 );
```

This sail is specified as two triangles using the `vertex` and `face` fields of MATLAB graphics objects. Recall that all PSS CAD models are stored as sets of triangular patches. The order of the vertices in the face field will determine the direction of the outward normal of the patch area (using the right-hand-rule). These faces have been carefully defined so that the normals face in the same direction. The optical coefficients for specular reflection, diffuse reflection, and absorption are stored in the `sigma` fields, with `S` for specular, `D` for diffuse, and `A` for absorptive; these coefficients should sum to 1 for each side of the sail. `CP1Props` gives these properties and the emissivity, approximately, for the material CP1.

Note that the sail mass structure is passed in using the variable `massSail`, which has the fields `inertia`, `mass`, and `cM`, or center of mass. Since this is a very simple square component, the `Inertias` function from the Spacecraft Control Toolbox can be used to compute the inertia of a plate with the desired sail mass and dimensions. `Inertias` computes the inertia with the axis of symmetry about z, so this must be rotated to the component frame; in this case the sail is in the Y/Z plane. An offset in the center of mass would be entered via the mass structure. The inertia is then computed using the code

```
inertiaSail   = Inertias( sailMass, [sailWidth sailWidth], 'plate'
   , 1 );
bXToZ         = [0 0 -1;0 1 0;1 0 0];
massSail      = struct('inertia', bXToZ*inertiaSail*bXToZ', 'mass'
   , sailMass, 'cM', [0;0;0] );
```

For a more complex mesh, the function `VFToMassStructure` can compute the mass properties assuming a unitary areal mass. The function `PolygonProps` from the Spacecraft Control Toolbox computes the area, outward normal, and geometric center of each triangle. Each triangle area is summed to the inertia using its distance from the mesh centroid. The `CircularSail` model

provides an example of doing this. The inertia and mass from `VFToMassStructure` are based only on area, so they need to be multiplied by the sail material areal mass.

```
% Mass properties
arealMass    = 0.005;  % kg/m2
mass         = VFToMassStructure( v, f );
mass.mass    = mass.mass*arealMass;
mass.inertia = mass.inertia*arealMass;
```

Note that accurate inertia is only necessary when you want to simulation attitude dynamics or control. If you only want to use a model for trajectory analysis, you can skip the inertia and just enter the correct mass.

3.4 Storing and retrieving sail models

After you create all the components, the resulting CAD model is stored as a data structure. You can view the fields by displaying the final structure `g` returned by `BuildCADModel( 'get cad model' )`. The following listings are from `SailWithBoom`; this model has two bodies, one for the sail and one for the boom, which rotates to provide two-axis attitude control. This CAD model is stored in Solar Sail Module as a mat file so you can retrieve the data without rerunning the script. The code to create a mat-file with the model is

```
%----------------------------------------------------------------
% Export
%----------------------------------------------------------------
if( createFiles )
  g = BuildCADModel( 'get␣cad␣model' );
  c = cd;
  cd(FindDirectory('SailData'));
  SaveStructure( g, 'SailWithBoom' );
  cd(c);
end
```

and the code to load the stored model and view the fields of the structure is

```
>> g = load('SailWithBoom.mat')

g =
         name: 'Solar␣Sail'
        units: 'mks'
         body: [1x2 struct]
    component: [1x5 struct]
       radius: 28.285
         mass: [1x1 struct]
```

We want to check the normals of the sail component to make sure that they are pointing forwards in the coordinate frame. These are computed and stored when the component is added to the CAD

model, as well as the area and location of the centroid of each triangle. We can view the names of the model's components to find the one we want,

```
>> {g.component.name}
ans =
    'CoreBox'    'Gimbal'    'Sail'    'BoomBox'    'Mast'
```

and we see that the sail is the third component, so we can examine all of its properties by displaying that substructure.

```
>> g.component(3)
ans =
                   faceColor: [0.7 0.7 0.7]
                   edgeColor: [1 1 1]
             diffuseStrength: 0.3
            specularStrength: 1
            specularExponent: 20
    specularColorReflectance: 0.5
                           b: [3x3 double]
                          rA: [3x1 double]
                           v: [4x3 double]
                           f: [2x3 double]
                           a: [2x1 double]
                           n: [2x3 double]
                           r: [2x3 double]
                      radius: [2x1 double]
                  deviceInfo: []
                       class: 'sail'
                        name: 'Sail'
                     optical: [1x1 struct]
                    infrared: [1x1 struct]
                     thermal: [1x1 struct]
                       power: [1x1 struct]
                        aero: [1x1 struct]
                    magnetic: [1x1 struct]
                        mass: [1x1 struct]
                      inside: 0
                          rF: [1x1 struct]
                        body: 1
                manufacturer: 'none'
                       model: 'generic'
```

In this listing we can see that the sail component is of the *sail* class. The normals are stored in `n` and the areas in `a`. `r` gives the vector to the centroid of each patch from the origin of the vehicle coordinate system.

```
>> g.component(3).n
ans =
     1     0     0
     1     0     0
```

```
>> g.component(3).r
ans =
        -0.25      -6.6667       6.6667
        -0.25       6.6667      -6.6667
>> g.component(3).a
ans =
   800
   800
```

For a complex sail, you might have thousands of faces with their own normals, and this simple check of the data in the structure may not suffice. The function `ComputeSailNormal` will compute the area-weighted normal of all the sail components in a model. The `CircularSail` is a good example of a mesh sail, as described above.

```
g = load('CircularSail');
n = ComputeSailNormal(g)

n =
     0
     0
     1
```

Over time, you may accumulate old stored models. To quickly get a synopsis of the properties of a sail model, use `DisplaySailProperties`. This will generate a 3D figure of the CAD model plus print out data like the sail area and spacecraft mass to the command line. You can use this function as a template to print out other data stored in the CAD models, or alternative parameters for the sail, whichever is most useful to your analysis.

```
g = load('CircularSail');
DisplaySailProperties(g)

-----------------------
Circular Sail
Sail normal: [0 0 1]
Sail area: 78373.567 m2
Sail mass: 395.63522 kg
Sail inertia (kg/m2):
   2478585.1 1.4479156e-10 1.1941665e-11
1.4479156e-10    2478585.1 1.961098e-12
1.1941665e-11 1.961098e-12      4941454

Sail characteristic accel: 1.7934 mm/s2
Number of bodies in model: 1
Number of components in model: 1
Sail class components: 1

Sail optical properties
Component Sail:
  Specular Front:      0.95 Back:   0.7
```

```
 Diffuse Front:        0.03 Back:   0.1
 Absorptivity Front:   0.02 Back:   0.2
 Emissivity Front:      0.1 Back:   0.3
```

The code that `DisplaySailProperties` uses to display the CAD model in a 3D figure is very simple and something you may want to embed in your scripts. The main function is `DrawSCPlanPlugIn`. A set of body axes can be added to the figure with `AddAxes`.

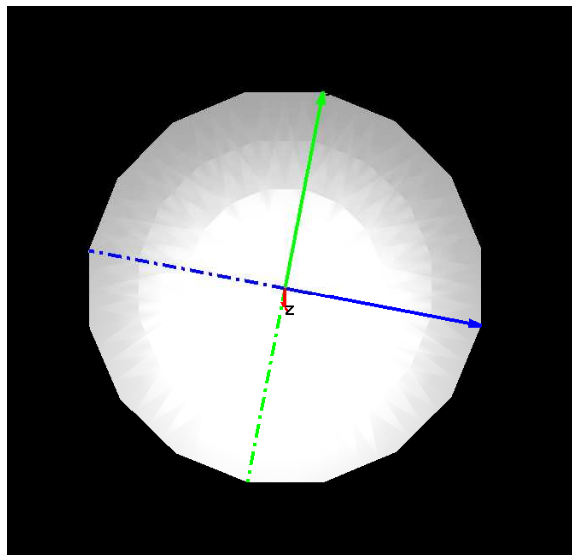
```
tag = DrawSCPlanPlugIn( 'initialize', g );
figH = findobj( 'tag', tag );
AddAxes(g.radius,[],[],figH);
```

You may always want to add a light source to the figure. This will cause the sail to be shiny on the front and darker on the back, when you rotate the figure around. This is a simple use of the `light` function. The light defaults to the infinite style, which is appropriate for the sun, in which case the position parameter is the direction from which the light shines. So, for our model, the position should be the same unit vector as the sail front normal.

```
light('position',[0;0;1])
```

When you view the sail from straight on with the light object in the figure, the sail will be white. This gets hard to see on a white background, so you can change the figure and axes background to block for a good visualization.

```
set(gcf,'color',[0 0 0])
set(gca,'color',[0 0 0])
```



3.5 Sail configurations

The Solar Sail Module provides a number of examples to help you get started modeling your own sail.

3.5.1 Flat Plate

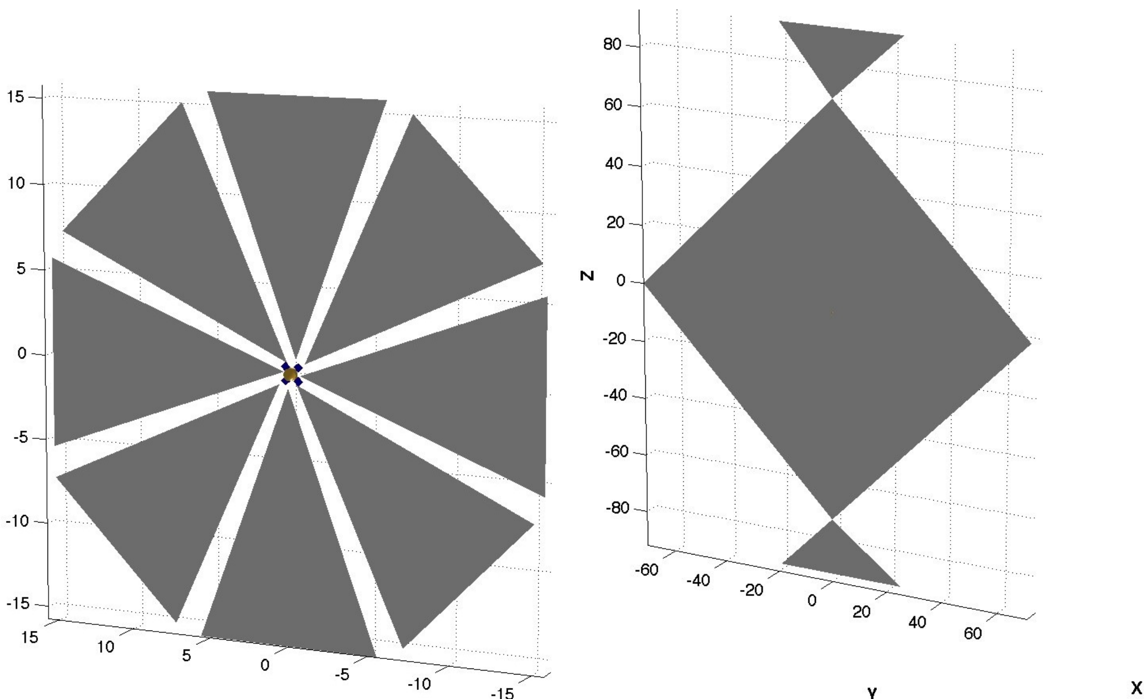
The simplest way to model a sail is as a perfectly specular plate. The membrane model requires a nonzero emissivity but if the emissivity of the front and back are equal, and the optical coefficients are specular, the model reduces to ideal specular reflection. Otherwise the flat plate sail component is created the same way as the sail from `SailWithBoom`. Just note the values of the `sigma*` and `emissivity` parameters.

```
% Sail
%-----
v = [0 0 0 0;0.5 -0.5 -0.5 0.5;0.5 0.5 -0.5 -0.5]'*sailWidth;
m = CreateComponent( 'make', 'sail','name','Sail','body',1,...
                 'mass', massSail, 'faceColor', 'mirror','rA'
                    ,[0;0;0],...
                 'sigmaS', [1 1], 'sigmaD', [0.0 0.0], 'sigmaA',
                    [0.0 0.0],...
                 'sigmaRS', [0.0 0.0], 'sigmaRD', [0.0 0.0], '
                    sigmaRA', [1 1],...
                 'emissivity', [0.03, 0.03],...
                 'vertex',v ,'face', [1 2 3; 1 3 4], 'inside', 0 )
                    ;
```

This model is used in several demos, including `SPICombinedDemo`, `SailCombinedDemo`, and `HeliopauseSimulation`. The mat-file storing this model is `FlatSail.mat`..

3.5.2 Sails with flat components

A variety of sail configurations can modeled as a set of flat components. Cosmos-1, for instance, consisted of a set of independently rotating, rigid sails. Other sail concepts involve rotating vanes at the tips of an otherwise static sail. Both `Cosmos1` and `PlateWithVanes` create multiple bodies for these rotating sail parts.

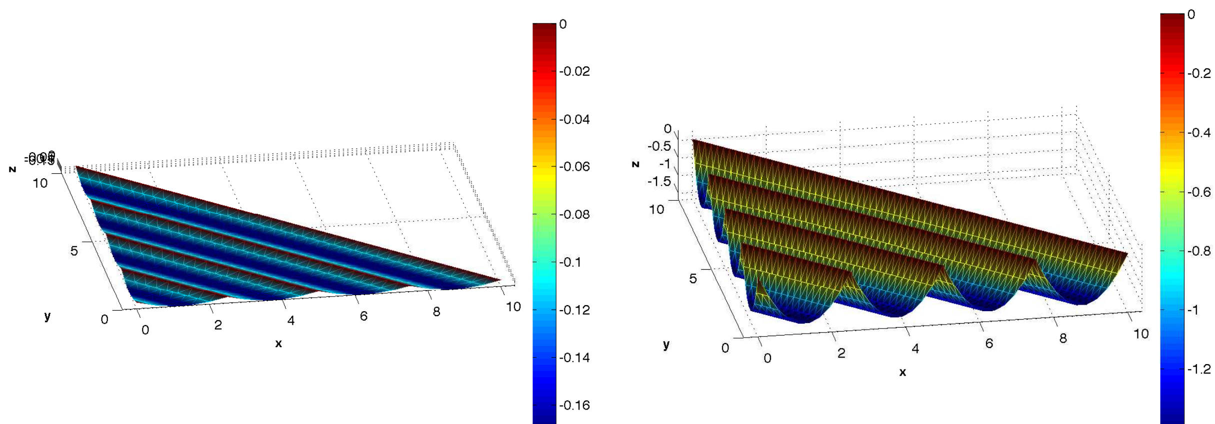


3.5.3 Striped Sail

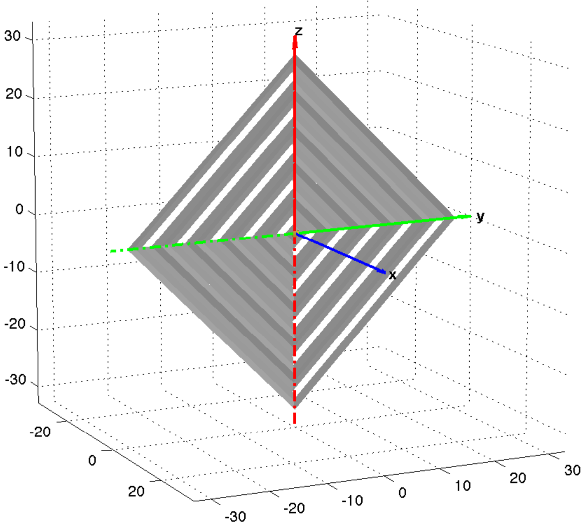
The striped sail model is a special case of a sail mesh. A square sail is divided into stripes from the inner corners of each quadrant to the outside; each of these stripes is formed by draping sail membrane over taut cords. The function `StripedQuadrant` will produce a quadrant mesh based on the quadrant length, number of stripes, subdivisions, and a fraction indicating how deep each stripe is draped compared to its width. The script `StripedSail` creates a sail model from these quadrants. This sail has a single body.

```
>> help StripedQuadrant
 ----------------------------------------------------------------------------

   Create a striped sail quadrant, with billow, in the x, y plane
      .
   This function has a built-in demo which draws a quadrant
      colored by
   vertical displacement. The billow is assumed quadratic, with a
       billow
   fraction used to define the billow as a percent of the stripe
      width.

   If a quadrant edge is 10 m, and there are 4 stripes, each will
       be about
   1.76 m wide. The billow fraction operates on this dimension.
      If nSub
```

```
   is not entered a value between 2 and 6 will be selected based
      on the
   amount of billow. The minimum number of stripe subdivisions is
       two.
--------------------------------------------------------------------------

   Form:
   [v, f] = StripedQuadrant( l, nStripes, billow, nSub )
--------------------------------------------------------------------------



   ------
   Inputs
   ------
   l           (1,1) Length of quadrant edge along x/y axes (not
      hypotenuse)
   nStripes    (1,1) Number of stripes
   billow      (1,1) Billow fraction, between 0 and 1
   nSub        (1,1) Subdivisions (rows of faces) per stripe,
      optional

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

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



Note that when put together into a sail, this model does not take into account the edges of each quadrant, and that the quadrants should meet at zero displacement along that edge. This region is relatively small compared to the breadth of the stripes across the quadrant.


z
y
x
30
20
10
0
-10
-20
-30
-20
0
20
-30
-20
-10
0
10
20
30

CHAPTER 4

DISTURBANCES

This chapter discusses how to use the disturbance functions, along with several related functions for defining the appropriate data structures.

4.1 Function Overview

```
>> help Disturbances
  Sail/Disturbances

  D
     DisturbanceStruct              - Return a default data structure for
                                    SailDisturbance.

  E
     EnvironmentStruct              - Return a default data structure for
                                    SailEnvironment.

  H
     HingeRotationMatrix            - Transformation matrix for an
                                    arbitrary number of single axis rotations.

  O
     OpticalMcInnesToPSS            - Convert McInnes optical coefficients
                                    to PSS format.

  P
     ProfileStruct                  - Return a default profile structure
                                    for SailDisturbance.

  S
     SailDisturbance                - Compute the forces and torques on a
                                    solar sail vehicle.
     SailEnvironment                - Space environment models. Designed to
                                     work with SailDisturbance.
```

```
   SolarPressureForce           - Combined thermal and optical solar
                              pressure force model.

Sail/Demos/Disturbances

E
   EarthOrbitDisturbances       - Demonstrate the solar sail
                              disturbance model in Earth orbit.

H
   HelioDisturbances            - Demonstrate the solar sail
                              disturbance model in heliocentric orbit.

S
   SolarForceDemo               - Demonstrate the solar pressure force
                              function using a striped sail
```

4.2 Solar pressure force function

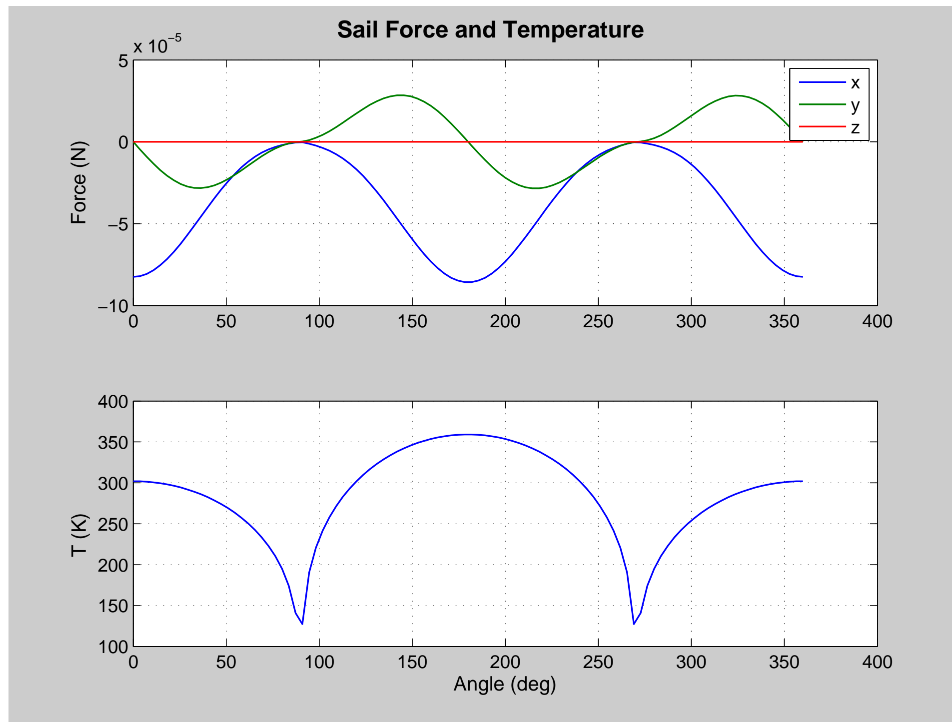
`SolarPressureForce` is a special function for sail membranes which combines the thermal and optical force models. This assumes that the membrane is at a constant temperature, which is appropriate since it is of negligible thickness. The front and back properties must be specified separately as shown in the header of the function. The function is vectorized to handle a set of element areas such as for a sail mesh. It is designed to be called for a single sun vector.

The function has a built-in demo. A single area element of 10 square meters is specified. The normal is rotated in a circle in the X-Y plane and the sun vector is along the X axis. The front of the sail has these optical coefficients: 0.8 specular fraction, 0.1 diffuse and 0.1 absorptive. The back of the sail has these coefficients: 0.7 specular, 0.1 diffuse and 0.2 absorptive.

The force and element temperature are shown in Figure 4.1 on the facing page. The temperature is seen to drop as the sail is edge on to the flux (angles of 90, and 270 degrees) and a higher temperature peak is reached when the more absorptive back side of the sail is towards the sun (angle of 180 degrees).

The force direction is best seen on a quiver plot. In Figure 4.2 on page 38, the normal of the front of the sail is shown in blue. The resulting scaled force vector is shown in red. The sun vector is shown in yellow, along the x-axis. We can see visually that the force vector is always pointing away from the sun and that the magnitude scales down as the element becomes edge-on, i.e. when the normal is aligned with the y-axis. When the element normal is facing away from the sun vector, the function automatically uses the back properties as “front” instead. You can see a slight misalignment between the force and normal at some angles due to the non-ideal optical properties.

Figure 4.1: Solar pressure force model demo



4.3 Environment Function

The environment function must be called as a precursor to the disturbance function. This will gather information on the environment of the central body. The function is

```
env = SailEnvironment( planet, p, d )
```

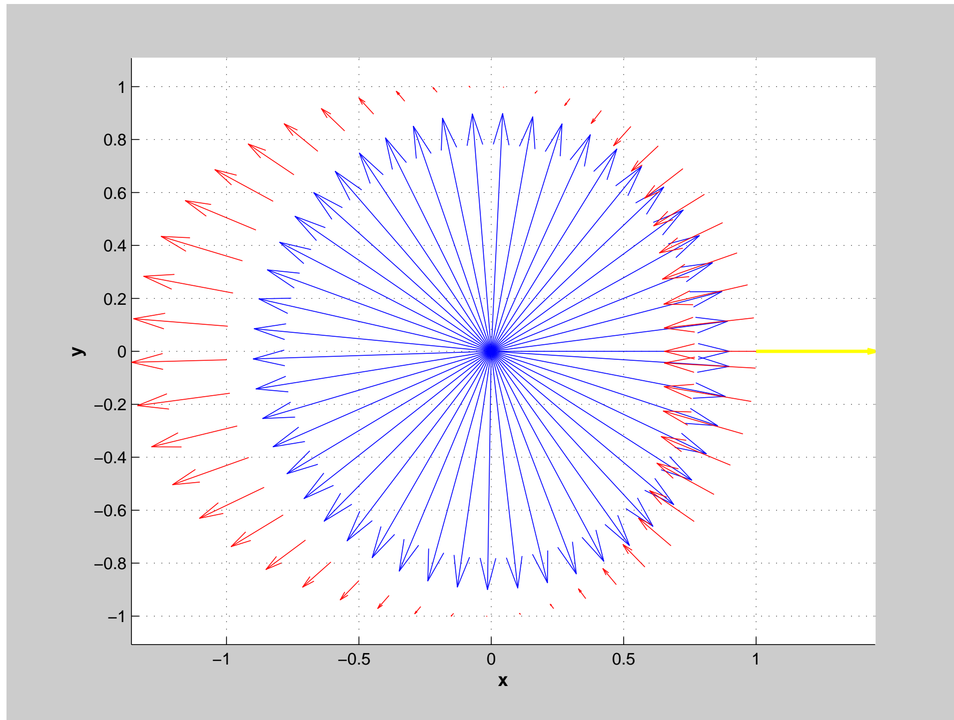
`SailEnvironment` maintains persistent memory of the central planet for efficiency and will reset automatically when called with a different planet name as the first input. The profile struct `p` requiresthe following fields:

- `jD`, Julian date of epoch
- `r`, position vector(s) of spacecraft relative to central planet
- `rPlanetH`, the heliocentric position of the planet. This field can be empty if the central body is the sun.

The data structure `d` requires the following fields describing the environment:

- `magModel`, name of magnetic field model, for example `BDipole`
- `atmModel`, name of atmospheric density model, for example `AtmDens1` or `AtmDens2`
- `j70`, data for the J70 atmospheric model if it has been selected in the previous field.

Figure 4.2: Force and normal directions using a quiver plot



The function `EnvironmentStruct` returns a default data structure with these fields. It can also be called with an existing data structure and the fields will be added.

The planet choices include the major planets and the sun. The planets are referenced by name, for example `'Earth'` or `'sun'`. The function returns a structure with the environmental data, including:

- `planet`, Planet name
- `radiation`, Black body radiation
- `albedo`, Planet albedo fraction
- `radius`, Planet equatorial radius (km)
- `mu`, Gravitational parameter
- `uSun`, Unit vector to sun, ECI frame
- `solarFlux`, Solar radiation flux (W/m^2)
- `altitude`, Altitude above the planet (km)
- `rho`, Atmospheric density (kg/m^3)*
- `bField`, Magnetic field strength*
- `radiationFlux`, Planetary radiation flux (W/m^2)*
- `albedoFlux`, Planetary albedo flux (W/m^2)*

The marked fields do not apply to the sun.

Planetary data is obtained from the `Constant` function. Planetary eclipses (when in planetary orbit) are modeled but lunar (or any moon) eclipses are not modeled. Eclipses are also not computed for heliocentric orbits.

4.4 Disturbance Function

The disturbance computation function is `SailDisturbance( g, p, e, d );` which takes as inputs a CAD model, `g`, an attitude and orbit profile structure, `p`, the environment data `e`, and a parameter structure, `d`. The scripts `EarthOrbitDisturbances` and `HelioDisturbances` demonstrate the function. The data structures are defined in the header. The function `DisturbanceStruct` returns a default data structure with the fields needed in `d`. This function can also be called with an existing data structure and the fields will be added.

The model assumes that the solar sail is composed of a core with multiple bodies attached to the core. Any component can be a sail membrane, which uses a combined thermal and optical property force model and requires both front and back properties. The function automatically adds an additional component for the back of each solar sail membrane using the specified properties. This function works on the face and vertex lists for the components. You can model a deformable sail by changing the sail vertices on each call. Shadowing is not modeled. The function can compute the following disturbances, which can each be turned on and off using the `d` structure:

- Aerodynamic force and torque
- Solar radiation pressure force and torque
- Albedo radiation force and torque
- Planetary radiation force and torque
- Gravity gradient torque
- Magnetic torque

The entered optical properties are applied to albedo and solar force and torque calculations. Radiation forces and torques use the properties specified for the infrared band.

This function can be called for a single data point or with an orbit and attitude profile. The demos listed above demonstrate the use of a profile and the built-in plots. The results of these demos are shown in the Examples chapter.

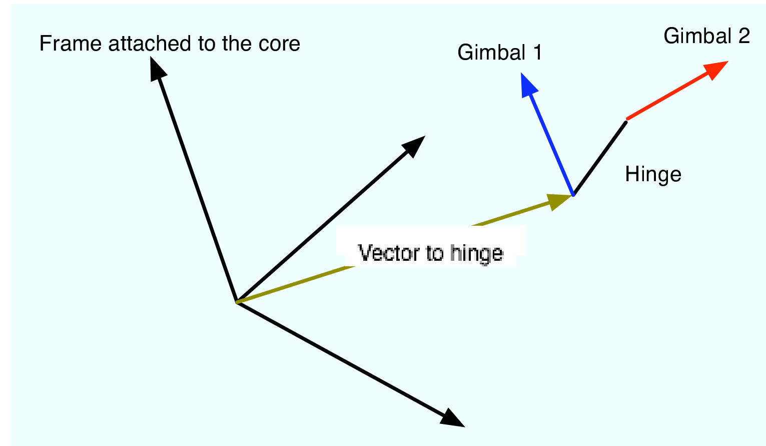
4.5 Profile Data Structure

The profile data structure gives the attitude, orbit, and Julian date of the solar sail for each point at which you want a disturbance calculation, in the fields `q`, `r`, `v`, and `jD`. The profile also contains

the heliocentric position of a central body other than the sun in the field `rPlanetH`. The function `ProfileStruct` returns a default data structure with the needed fields.

In addition to the core attitude quaternion `q`, if your sail has gimbaled bodies attached to the core, you must also input the gimbal angles. The gimbal arrangement is shown in Figure 4.3. You can append gimbals by having multiple axes and angles for each body hinge. The three elements of

Figure 4.3: Gimbal configuration



the data structure concerned with gimbals are

- `angle`, Angles
- `axis`, Axes for angles
- `body`, Body hinge for angle

Each column of `angle` is a time step. Each row is a gimbal angle. If you had 3 double-gimbaled bodies `angle` would have 6 rows. For each gimbal the angles are always ordered from the gimbal nearest to the core to the one furthest from the core. For this example the corresponding `body` array would be `[1 1 2 2 3 3]`. `axis` gives the axis of rotation for the angles. The axis of rotation for the gimbal closest to the core is in the core frame. The next axis of rotation is in the rotated frame of the first gimbal axis.

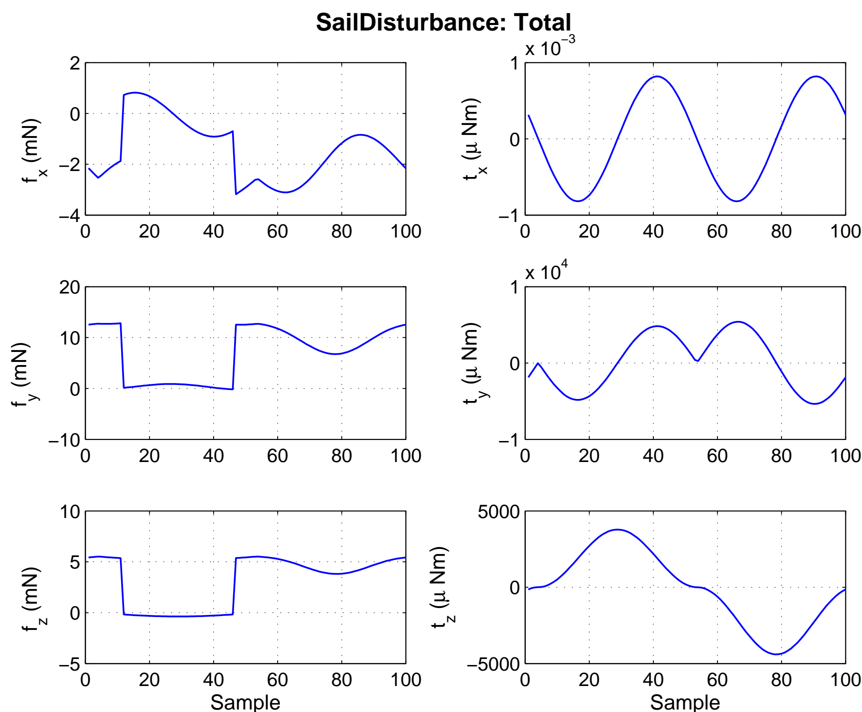
4.6 SailDisturbance Demo

`SailDisturbance` has a built-in demo of a batch analysis which demonstrates the use of these functions. The sail is analyzed in a simple low Earth orbit with sun-pointing attitude ($+x$ pointing towards the sun). The model, `SailWithBoom`, is discussed in Chapter 8 on page 73. The sail is 40 m square with non-ideal optical properties. Note that the forces are output in in the ECI frame and the torques are in the body frame.

SailDisturbance

The function will plot all outputs. Five are shown here in Figure 4.4, Figure 4.5 on the next page, Figure 4.6 on the following page, Figure 4.7 on page 43, Figure 4.8 on page 43. In the plot of solar force we can clearly see the fixed force when the sail is illuminated, resulting from the sun-pointing attitude, and the zero output when the sail is in eclipse. On the total force and torque plot we also see large periodic variations. By looking at the remaining plots of gravity gradient, albedo, and (planetary) radiation, we can trace the sources. Radiation is contributing a huge torque in y and z, and throughout the orbit as radiation is independent of eclipses. Earth albedo contributes a few mN of force and substantial torque when the sail is on the sun-side of the Earth. There is a smaller gravity gradient torque.

Figure 4.4: `SailDisturbance` demo total force and torque



We can quickly verify the rough solar force magnitude by computing the force on an ideal sail of this area at the Earth,

$$F = 2\left(\frac{1367}{3\times 10^8}\right)\cdot 1600 = 0.0146$$

or 14.6 mN. Given this the solar forces make sense, confirming that the sail is pointed squarely at the sun.

Additional disturbance examples are reviewed in 8.2 on page 76.

Figure 4.5: `SailDisturbance` demo solar force and torque

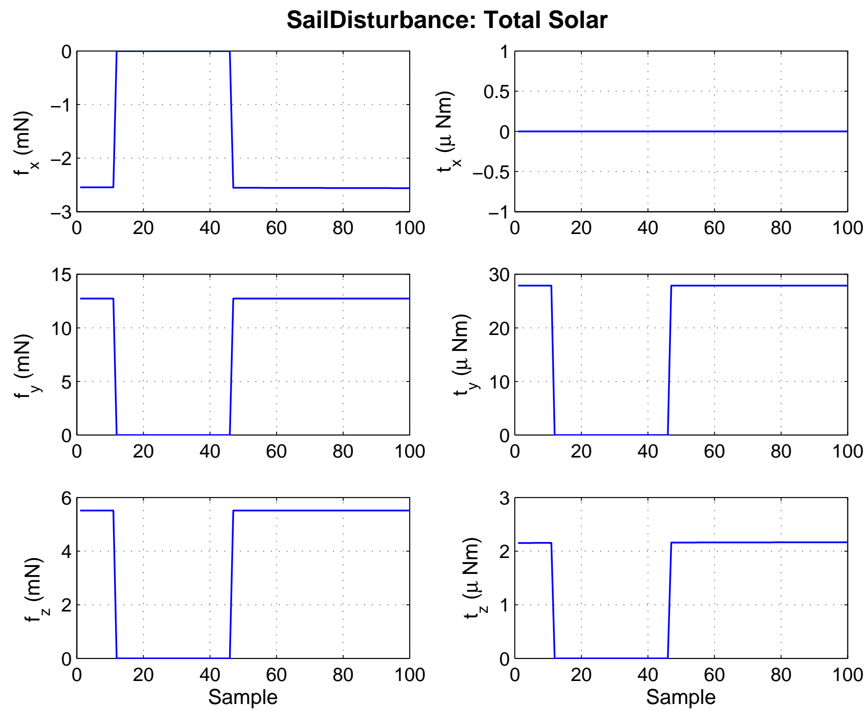


Figure 4.6: `SailDisturbance` demo gravity gradient torque

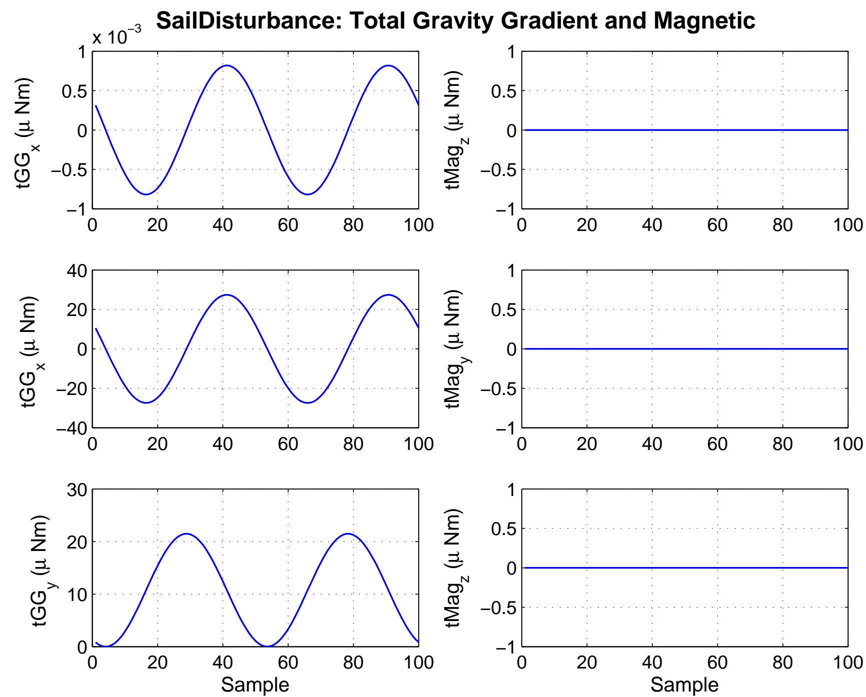


Figure 4.7: `SailDisturbance` demo albedo

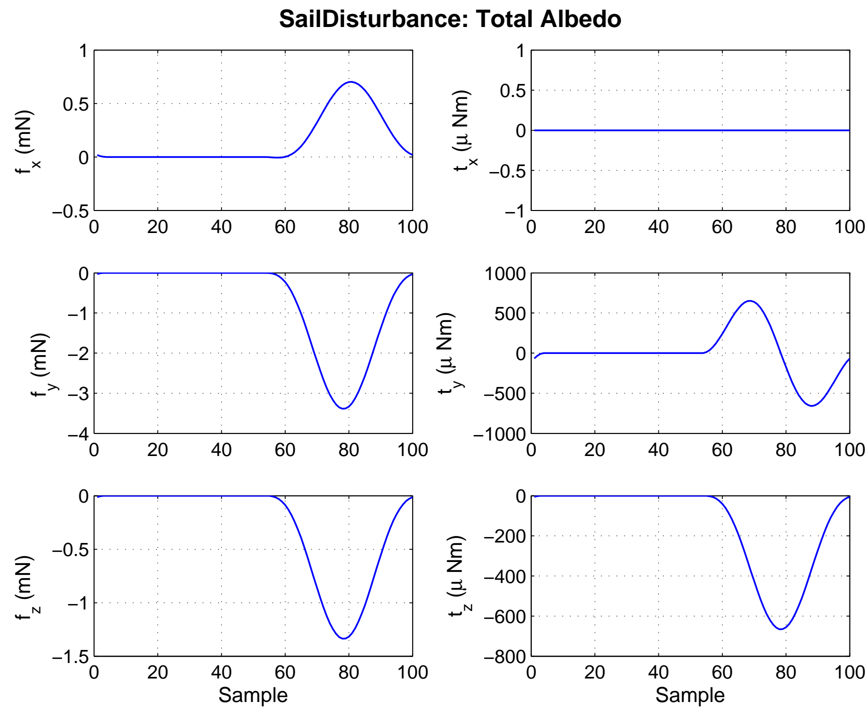
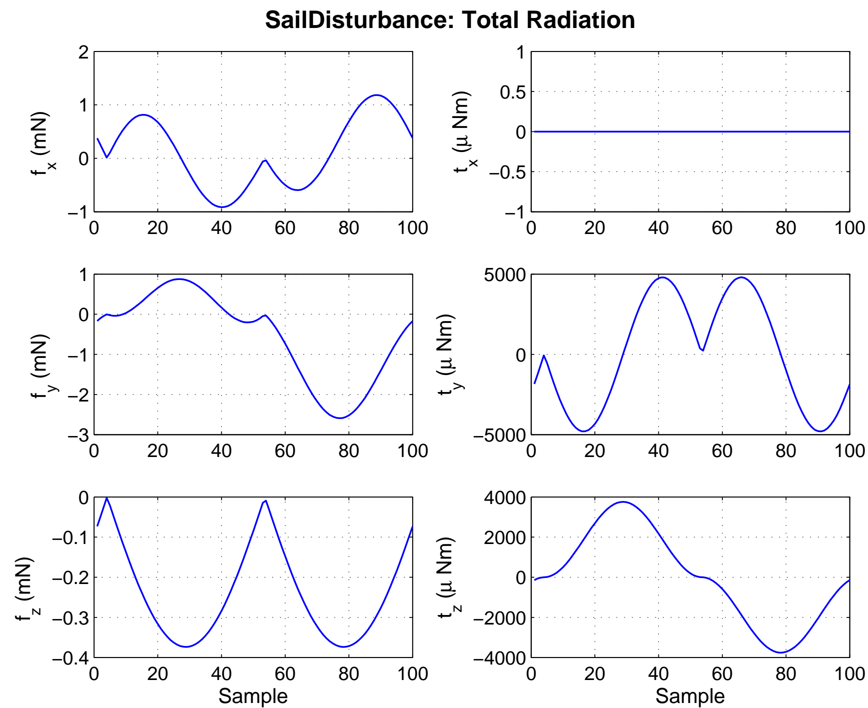


Figure 4.8: `SailDisturbance` demo radiation



CHAPTER 5

ATTITUDE DYNAMICS

This chapter shows you how to use the special attitude dynamics models included in the Solar Sail Module.

5.1 Function Overview

```
>> help AttitudeDynamics
  Sail/AttitudeDynamics

  F
     FCoreAndMoving                        - Example system RHS which
                                 incorporates FMovingBody.
     FMovingBody                           - Moving body dynamics model with
                                 instantaneous velocities.
     FSailRB                               - Rigid body right-hand-side for Sail
                                  module.
     FSailTB                               - Rewrite of FTB in preferred sail
                                 module format
     FTimeVaryingI                         - Attitude RHS with time varying
                                 inertia of a single body.

  H
     HGimballedBoom                        - Calculate angular momentum of boom
                                 system and update the body rates.

  M
     MassVehicle                           - Compute the mass data structure.

  T
     TwoBodyRateModel                      - Gimballed boom dynamics model for
                                 fixed gimbal rates.
```

5.2 Rigid Body Dynamics

The simplest implementation, rigid body dynamics, can be implemented using either FRB or FSailRB. Both models include quaternion kinematics.

The Core Toolbox function FRB can be integrated using PSS' Runge-Kutta integrators, for example

```
x = RK4( 'FRB', x, dT, t, inr, invInr, tS.total );
```

where x is the state consisting of a stacked quaternion and body rate vector.

5.3 General Two-Body Dynamics

The functions FTB and TBModel in the Core Toolbox implement a general two-body dynamical model.

The function FTB incorporates quaternion kinematics and calls the two-body dynamics model function TBModel, which is described below. The inputs to FTB includes the 14-component state vector for the two rigid bodies containing 2 sets of four quaternion elements and three angular velocity components, the time stamp, relative positions of the centers of mass of the two bodies, mass and inertia properties, force and torque inputs, and a specification of the unconstrained axes of the second rigid body in iAxis. The force input contains all the external force components acting at centers of mass. The torque input contains the total external torque acting on the body, and the internal control hinge torque. The output of FTB is the vector of state derivatives xDot.

The function TBModel models any two rigid bodies attached by a hinge, whose number of degrees of freedom can range from 1 to 3. This function essentially takes the necessary state, force and torque specifications, and the mass and inertia properties of the rigid body, which may be specified by through the function FTB for example, and outputs the angular accelerations, the total angular momentum of the system and the generalized inertia matrix. The numbers of the axes that are unconstrained must be specified in the structure element d.iAxis. For example, if the 1st and 3rd axes are unconstrained d.iAxis must be set to [1 3]. For a three degree of freedom hinge the iAxis must be set to [1 2 3].

5.4 Fixed Rate Rotating and Translating Bodies

The function FMovingBody.m incorporates a general translating and rotating body dynamics model with instantaneous velocities, which is appropriate for a system with stepping motors. The core body rates must be explicitly updated when any attached body attains new rates.

The demo MovingBodyDemo.m illustrates the application of FMovingBody.m to translating

bodies and verifies angular momentum conservation for zero external torque. The 13 states for each body, position, velocity, quaternion, and body rates, are stacked. In this case, the core is given random body rates and the masses have non-zero initial positions.

```
wCore   = randn(3,1)*0.1;
xCore   = [zeros(6,1);QZero;wCore];
xMass1  = [[0;2;0];zeros(10,1)];
xMass2  = [[0;0;-2];zeros(10,1)];
x       = [xCore;xMass1;xMass2];
```

The velocities are updated several times during the demo. The code which updates the core rates is

```
[x, h]  = FMovingBody( 'init', x, xNew, [], d );
```

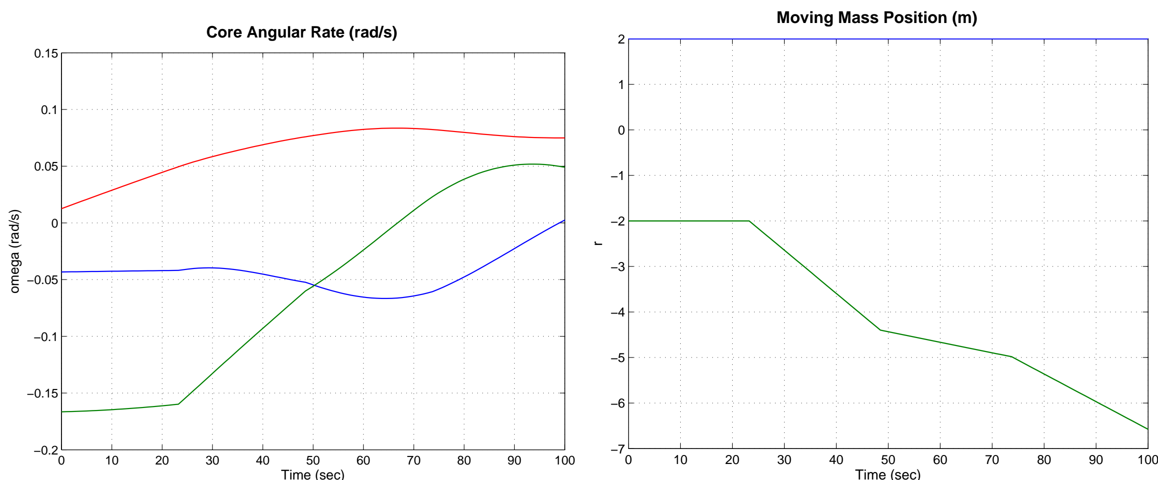
This function call also returns the angular momentum of the state `x`.

Since `FMovingBody` only returns the attitude states, it must be combined with the translational dynamics in another function. For this demo the function is `FCoreAndMoving`. This is the function which is actually integrated in the line

```
[z, x] = ode113( 'FCoreAndMoving', [t(k-1) t(k)], x, xODEOptions,
   d );
```

The results for this demo are shown in Figure 5.1.

Figure 5.1: `MovingBodyDemo` sample results



Additional examples of this dynamics function include `SMAGuidanceWithBoom`, which models a rotating boom on a general hinge, and `BallastMass2Axis`, which demonstrates sail control using two moving masses.

The function `TwoBodyRateModel.m` incorporates a gimballed boom dynamics model using fixed gimbal rates. This function is described in Section 5.6.2.

5.5 Time Varying Inertia

The function `FTimeVaryingI.m` models the orbit and attitude dynamics of a single rigid body with a time-varying inertia. The inputs to this function are time, state vector (four quaternion elements and angular velocity), force, torque and the name of the function that provides the inertia derivative. The output is the state derivative vector.

The dynamics are

$$I\dot{\omega} + \dot{I}\omega + \omega \times h = T \tag{5.1}$$

where ω are the body rates, T is the total external torque, and h is the angular momentum or $I\omega$. The state consists of the quaternion and the body rates plus the nine inertia elements.

The inertia derivative function must be of the form

```
Idot = FInertia( t, d )
```

where `Idot` is returned as a 3x3 matrix. The `FTimeVaryingI` function includes a default zero derivative function `IDotDefault` which can be used for debugging during script setup.

The demo `S4Deployment.m`, which simulates the dynamics of a solar sail while deploying, uses `FTimeVaryingI.m` with the inertia derivative function `IDotS4`. Figure 5.2 on the next page shows the variation of body rates, inertia and inertia derivatives for an implementation of this demo. The CAD model is created in `S4Deploy`; it can create both pre- and post-deployment versions of the model. A special torque function `S4DeployTorque` models a changing center-of-pressure offset. This demo was created from a student paper which analyzed ATK's scalable square sail. In summary, the following files all relate to this demo:

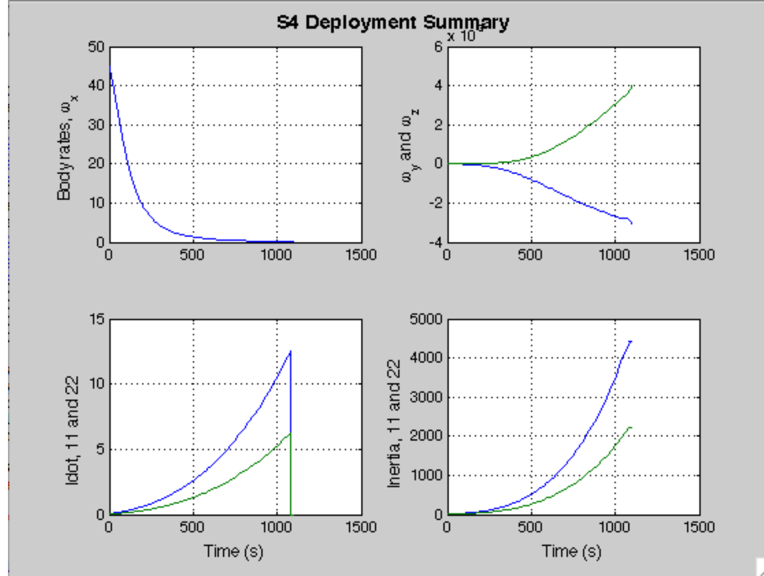
- `S4Deployment.m`
- `IDotS4.m`
- `S4Deploy.m`
- `S4DeployTorque.m`
- `S4Deployed.mat`
- `S4PreDeploy.mat`

5.6 Special Two-Gimbal Model for a Boom

5.6.1 Dynamical Equations

To simplify the model, we will assume that the gimbals achieve their nominal rate instantaneously. This means that we do not have to model the gimbal torque explicitly. We do however have to model a momentum sink on the spacecraft, such as reaction wheels, to absorb the momentum

Figure 5.2: Solar sail deployment demo



changes caused by dynamically moving the center of mass. This will keep the core body rates fixed as the boom moves to a specified orientation.

We use Hooker's derivation for multibody dynamics but will neglect the derivatives of the gimbal rates. Hooker [?] begins with the equations of motion for each body, including the constraint torques at each joint. A solvable system of equations is obtained by first summing all the body equations to form one equation, which eliminates these constraint torques. Second, the body equations are summed for each hinge outward, resulting in $n-1$ equations each with the constraint torque of the innermost hinge.

The equations of motion for a single body λ with connective joints J_λ in the set of connected bodies S are

$$\sum_{\mu \in S} \Phi_{\lambda\mu}\dot{\omega}_\mu = E_\lambda + \sum_{j \in J_\lambda} T^C_{\lambda j} \tag{5.2}$$

in which

$$\begin{aligned} E_\lambda = \tfrac{3G}{\rho^3} \times \Phi_{\lambda\lambda}\tilde{\rho} + \omega_\lambda \times \Phi_{\lambda\lambda}\omega_\lambda + T^{ext}_\lambda + \textstyle\sum_{j\in J_\lambda} T^H_{\lambda j} + D_\lambda \times F^{ext}_\lambda + \\ \textstyle\sum_{\mu \in J_\lambda} D_{\lambda\mu} \times \left[\sum_{\mu\in S_{\lambda j}} F^{ext}_\mu + m\omega_\mu \times [\omega_\mu \times D_{\mu\lambda}] + m\frac{G}{\rho^3}(I - 3\tilde{\rho}\tilde{\rho}^T)D_{\mu\lambda}\right] \end{aligned} \tag{5.3}$$

where D are augmented hinge vectors or barycenters, Φ are augmented inertia matrices for the tree, m is the total vehicle mass, and G is the gravitational constant. The torques transmitted through motors or gimbals are in T^H, while the non-gravitational external forces and torques are in F^{ext}_λ and T^{ext}_λ. The constraint torques T^C are eliminated by summing these equations over all the bodies and, for fixed λ, summing all bodies beyond a joint j. This provides a system of equations for the rates of the core body and the gimbal rates, or one vector rate equation and n scalar rate equations. The barycenters D can be understood more easily when diagrammed. For a body λ, D_λ is the vector from the body center of mass to the new center of mass obtained by lumping all directly connected bodies at their respective joints. $D_{\lambda j}$, or $D_{\lambda\mu}$ if the hinge index j is replaced by the index of the connected body.

The inertia matrices Φ are defined as

$$\Phi_{\lambda\lambda} = \Phi_\lambda - m_\lambda D_\lambda^\times D_\lambda^\times - \sum_{\mu \neq \lambda} m_\mu D_{\lambda\mu}^\times D_{\lambda\mu}^\times \tag{5.4}$$

$$\Phi_{\lambda\mu} = m D_{\mu\lambda}^\times D_{\lambda\mu}^\times \tag{5.5}$$

The rate of any body except the core is written as

$$\omega_\mu = \omega_0 + \sum_{k=1}^{n-3} \epsilon_{k\mu} \dot{\gamma}_k g_k \tag{5.6}$$

where γ is the angle of rotation about unit vector g, and $\epsilon_{k\mu}$ indicates if that g_k belongs to a joint on the chain connecting body μ and 0. This definition of ω_μ is substituted into the left-hand sides of the summed equations of motion, resulting in the combined equation

$$\sum_\lambda \sum_\mu \Phi_{\lambda\mu} \cdot (\dot{\omega}_0 + \sum_k (\ddot{\gamma}_k g_k + \dot{\gamma}_k \dot{g}_k)) = \sum_\lambda E_\lambda \tag{5.7}$$

At this point we diverge further from Hooker's tensor formulation as we require true matrix notation, including all transformations. Each body equation is written in its own frame, requiring tranformations of ω_0 and E when the equations are summed. The inertial rate of any body in its own frame is written explicitly as

$$\omega_\mu = B_{\mu 0}^T \omega_0 + \sum_{k=1}^{n-3} \epsilon_{k\mu} \dot{\gamma}_k B_{\mu k}^T g_k \tag{5.8}$$

where B_{ij} transforms a vector in the i frame to the j frame. Equation (5.7) is written in the core frame, so each equation and right-hand-side must also be transformed. The result is

$$\sum_\lambda B_{\lambda 0} \sum_\mu \Phi_{\lambda\mu} (B_{\mu 0}^T \dot{\omega}_0 + \dot{B}_{\mu 0}^T \omega_0 + \sum_k \epsilon_{k\mu} (\ddot{\gamma}_k B_{\mu k}^T g_k + \dot{\gamma}_k \dot{B}_{\mu k}^T g_k)) = \sum_\lambda B_{\lambda 0} E_\lambda \tag{5.9}$$

This is the first system equation. The remainder are obtained by summing equations from each joint j outward. Then, a dot product is taken with each resulting equation and the axis of rotation g_j. This leaves a scalar equation in which the constraint torque at j is eliminated since it is orthogonal to the axis of rotation.

The dynamical equations can be written as a single matrix equation in the following way (for two joints):

$$\begin{bmatrix} A_{00} & \vec{a_{01}} & \vec{a_{02}} \\ \vec{a}_{10} & a_{11} & a_{12} \\ \vec{a}_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} \dot{\omega}_0 \\ \ddot{\gamma}_1 \\ \ddot{\gamma}_2 \end{bmatrix} = \begin{bmatrix} \sum_\lambda B_{\lambda 0} E'_\lambda \\ g_1^T \sum_\lambda \epsilon_{1\lambda} E'_\lambda \\ g_2^T \sum_\lambda \epsilon_{2\lambda} E'_\lambda \end{bmatrix} \tag{5.10}$$

where

$$E'_\lambda = E_\lambda - \sum_\mu \phi_{\lambda\mu} \dot{B}_{\mu 0}^T \omega_0 - \sum_\mu \Phi_{\lambda\mu} (\sum_k \epsilon_{k\mu} \dot{\gamma}_k \dot{B}_\mu^T g_k) \tag{5.11}$$

Note that this generalized inertia matrix A is symmetric.

Now we can formulate the equations for our specific case two bodies with two hinges, including vector notation and all transformation matrices. The sailcraft is grouped into two bodies. The sail is considered the core body, and the gimballed boom the attached body. The vectors from the center of mass of each body to its joints are denoted by L.

First we write the angular velocity of the attached body in its own frame as a combination of the two gimbal rates. The rates γ_k at the hinges will take the values of α and β for clarity. The rotation axis vectors g are expressed in the previous frame.

$$\Omega = \alpha B_\beta g_\alpha + \beta g_\beta \tag{5.12}$$

We move next to the definition of the generalized inertia matrix, A, which is a combination of matrix A_{00}, vectors $\vec{a}$, and scalars a. The indices μ and λ take the values 0 and 1. However, it is important to note that there is only a single joint in this case, which consists of the two gimbals.

$$A_{00} = \sum_\lambda \sum_\mu \Phi_{\lambda\mu} = \Phi_{00} + \Phi_{01} + \Phi_{11} + \Phi_{10} \tag{5.13}$$

Including the necessary transformations, A becomes

$$A_{00} = \Phi_{00} + \Phi_{01} B^T + B(\Phi_{10} + \Phi_{11} B^T) \tag{5.14}$$

Since there are only two bodies, we simplify the matrix B_{10}, which transforms vectors in the 1 frame to the 0 frame, to B. B^T is therefore the same as B_{01}. We will now neglect the a terms and write the resulting equation for $\dot{\omega}_0$,

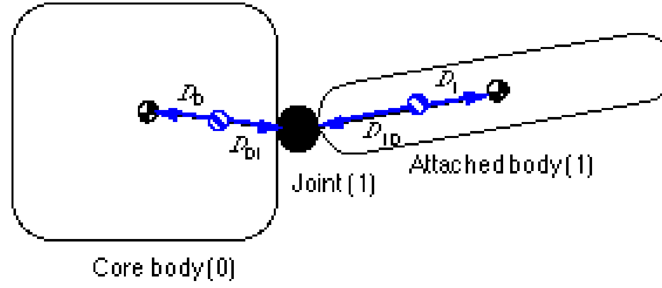
$$\left[\Phi_{00} + \Phi_{01} B^T + B(\Phi_{10} + \Phi_{11} B^T)\right] \dot{\omega}_0 = E'_0 + B E'_1 \tag{5.15}$$

The next step is deriving the generalized inertia matrices Φ. These depend on the barycenters D and, in fact, the physical interpretation of Φ_{11} is the inertia matrix of the augmented body λ about its barycenter D_1. First we write out the barycentric vectors. Note that there is only one joint. The resulting vectors are shown in Figure 5.3 on the following page.

$$\begin{aligned} D_\lambda &= -\frac{1}{m} \sum_{\mu \neq \lambda} m_\mu L_{\lambda\mu} \\ D_0 &= -\frac{m_1}{m} L_0 \\ D_1 &= -\frac{m_0}{m} L_1 \end{aligned} \tag{5.16}$$

$$\begin{aligned} D_{\lambda\mu} &= D_\lambda + L_{\lambda\mu} \\ D_{01} &= D_0 + L_0 = \frac{-m_1 + m}{m} L_0 = \frac{m_0}{m} L_0 \\ D_{10} &= \frac{m_1}{m} L_1 \end{aligned} \tag{5.17}$$

Figure 5.3: Solar sail barycenters



Now we can write out the augmented inertia matrices. We will substitute the D's defined above without showing the intermediate steps. $\bar{m}$ indicates the reduced mass of the system, $\frac{m_1 m_2}{m}$.

$$\begin{aligned} \Phi_{ii} &= \Phi_i - \bar{m} L_i^\times L_i^\times \qquad (5.18) \\ \Phi_{ii} &= \bar{m} L_i^\times L_j^\times \end{aligned}$$

The L_i must also be expressed in the correct frame. l_i is used to denote the vectors expressed in their respective body frames. Each $\Phi_{ij}$is expresssed in the i frame with a transformation matrix used for the vectors in the j frame.

$$\Phi_{00} = \Phi_0 - \bar{m} l_0^\times l_0^\times \qquad (5.19)$$

$$\Phi_{01} = \bar{m} l_0^\times B \, l_1^\times \qquad (5.20)$$

$$\Phi_{11} = \Phi_1 - \bar{m} l_1^\times l_1^\times \qquad (5.21)$$

$$\Phi_{10} = \bar{m} l_1^\times B \, l_0^\times \qquad (5.22)$$

The final step to calculating the right-hand-side of the equations is writing out the E vector. This includes external disturbances, gimbal/hinge torques (T^H), and the effect of each body on the other.

$$\begin{aligned} E_0 &= \frac{3G}{\rho^3} \hat{\rho} \times \Phi_{00} \hat{\rho} - \omega_0 \times \Phi_{00} \omega_0 + T_0^{ext} + T_{01}^H + D_0 \times F_0^{ext} \qquad (5.23) \\ &\quad + D_{01} \times \left\{ F_1^{ext} + m\omega_1 \times [\omega_1 \times D_{10}] + m \frac{G}{\rho^3} (I - 3\hat{\rho}\hat{\rho}^T) D_{10} \right\} \\ E_0' &= E_0 - \sum_\mu \Phi_{0\mu} \dot{B}_{\mu 0}^T \omega_0 - \sum_\mu \Phi_{0\mu} \cdot (\epsilon_{\alpha\mu} \dot{\alpha} \dot{g}_\alpha + \epsilon_{\beta\mu} \dot{\beta} \dot{g}_{beta}) \qquad (5.24) \\ &= E_0 - \Phi_{01} (\dot{B}^T \omega_0 + \dot{\alpha} \dot{g}_\alpha + \dot{\beta} \dot{g}_\beta) \qquad (5.25) \end{aligned}$$

The derivative of the transformation matrix is obtained by multiplying by the skew of the angular rate, in this case the rate of the attached body Ω from Equation 5.12. The second gimbal axis is fixed in the body frame leaving only the first axis with a derivative. The axis derivative is in the transformation matrix as the vector has unit length.

$$E_0' = E_0 - \Phi_{01} (-B^T \Omega^\times \omega_0 + al\dot{p}ha \dot{B}_\alpha g_\alpha) \qquad (5.26)$$

The E vector for the attached body is very much the same.

$$E_1 = \frac{3G}{\rho^3} \hat{\rho} \times \Phi_{11} \hat{\rho} - \omega_1 \times \Phi_{11} \omega_1 + T_1^{ext} + T_{11}^H + D_1 \times F_1^{ext} \qquad (5.27)$$

$$
\begin{aligned}
& +D_{10} \times \left\{ F_0^{ext} + m\omega_0 \times [\omega_0 \times D_{01}] + m\frac{G}{\rho^3}(I - 3\hat{\rho}\hat{\rho}^T)D_{01} \right\} \\
E_1' &= E_1 - \sum_\mu \Phi_{1\mu}\dot{B}_{\mu 0}^T\omega_0 - \sum_\mu \Phi_{1\mu} \cdot (\epsilon_{\alpha\mu}\dot{\alpha}\dot{g}_\alpha + \epsilon_{\beta\mu}\dot{\beta}\dot{g}_{beta}) \quad (5.28) \\
&= E_1 - \Phi_{11}(-\dot{B}^T\Omega^\times\omega_0 + \dot{\alpha}\dot{g}_\alpha + \dot{\beta}\dot{g}_\beta) \quad (5.29)
\end{aligned}
$$

The vector ρ, from the Earth to the spacecraft, is taken to the spacecraft composite center of mass and is the same for both E_0 and E_1.

Each rotation is defined by an axis and an angle. Each rotation is denoted as B_θ. The transformation from the core body frame to the attached body is the combination of the transformations,

$$B_{01} = B_\beta B_\alpha \quad (5.30)$$

In this case the second axis is fixed in the body frame and has zero derivative. The first axis is expressed in the body frame as

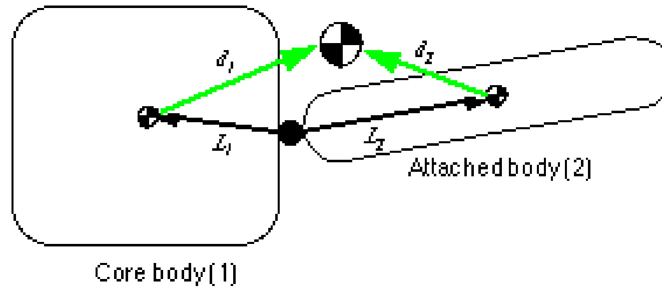
$$g_\alpha^B = B_\beta g_\alpha^C \quad (5.31)$$

The derivative of the first gimbal axis, $\dot{g}_\alpha$, in the body frame is then easy to express in terms of its outbound angular velocity.

$$
\begin{aligned}
\dot{g}_\alpha &= \dot{\alpha}B_\beta\Omega_\alpha^\times g_\alpha \quad (5.32) \\
\Omega_\alpha &= \dot{\beta}g_\beta \quad (5.33)
\end{aligned}
$$

Angular momentum conservation (in the absence of external forces and torques) is used to verify the model. The momentum is taken about the aggregate center of mass of the two bodies. In this case we take the indices of the bodies to be 1 and 2.

Figure 5.4: Aggregate center of mass



The total inertial angular momentum can be written as

$$H = AI_1\omega_1 + ABI_2(\omega_2 + B^T\omega_1) + m_1 D_1^\times \dot{D}_1 + m_2 D_2^\times \dot{D}_2 + Ah_W \quad (5.34)$$

where the inertias and angular rates are taken in the respective body frames, A transforms from the first body frame into the inertial frame, B transforms from the second body frame to the first, and h_W is the stored momentum which keeps the body rates from coupling to the gimbal rates. The

D vectors are d from Figure 5.4 on the previous page expressed in the inertial frame. They are expressed using the body frame vectors l as

$$\begin{aligned} D_1 &= \frac{m_2}{m} A(l_1 - Bl_2) \\ D_2 &= \frac{m_1}{m} A(l_1 - Bl_2) \end{aligned} \tag{5.35}$$

$$\begin{aligned} \dot{D}_1 &= \frac{m_2}{m} A(\omega_1^\times (l_1 - Bl_2) - B\omega_2^\times l_2) \\ \dot{D}_2 &= -\frac{m_1}{m} A(\omega_1^\times (l_1 - Bl_2) - B\omega_2^\times l_2) \end{aligned} \tag{5.36}$$

For this formulation we need to express ω_2 in the body 2 frame.

$$\omega_2 = \dot{\alpha} B_\beta g_\alpha + \dot{\beta} g_\beta \tag{5.37}$$

When the gimbals are assigned a new velocity, the angular momentum change is computed and the adjustment added to h_W so that ω_1 remains constant.

5.6.2 Two Body Functions

This derivation is implemented in the function `TwoBodyRateModel.m`. It provides the kinematics of the quaternion derivative in addition to the dynamics of the rate derivative. This function is used in the demo `BoomControl.m`, discussed in the next section. In addition the function `HGimballedBoom.m` determines the change in momentum when the gimbal rates change, which must be absorbed in a sink such as a set of reaction wheels.

Listing:

```
1  %
   ----------------------------------------------------------------------------
2  %   Gimballed boom dynamics model for fixed gimbal rates.
3  %   The core body should be body 1 and the boom body 2 in the CAD model.
4  %   The gimbal rates are termed aDot.
5  %
   ----------------------------------------------------------------------------
6  %   Form:
7  %   [xDot, h, gIner] = TwoBodyRateModel( x, t, force, torque, g, p, hW )
8  %
   ----------------------------------------------------------------------------
9  %
10 %   ------
11 %   Inputs
12 %   ------
13 %   t              (1,1)     Time
14 %   x              (15,1)    The state vector [q;omega;a;aDot]
15 %   force          (:)       Force structure
16 %                            .totalBody    (3,1,2) Total torque on the
   vehicle
17 %   torque         (:)       Torque structure
```

```
18 %                         .totalBody    (3,1,2) Total torque on the
   vehicle
19 %  g              (1,1)     CAD model
20 %  p              (1,1)     Profile
21 %  hW             (3,1)     Stored angular momentum, body frame
22 %
23 %  -------
24 %  Outputs
25 %  -------
26 %  xDot           (7+2n,1)  [qDot;omegaDot;aDot;aDDot]
27 %  h              (3,1)     Inertial angular momentum in the body frame
28 %  gIner          (3+n,3+n) Generalized inertia matrix
29 %
30 %
   --------------------------------------------------------------------------------
```

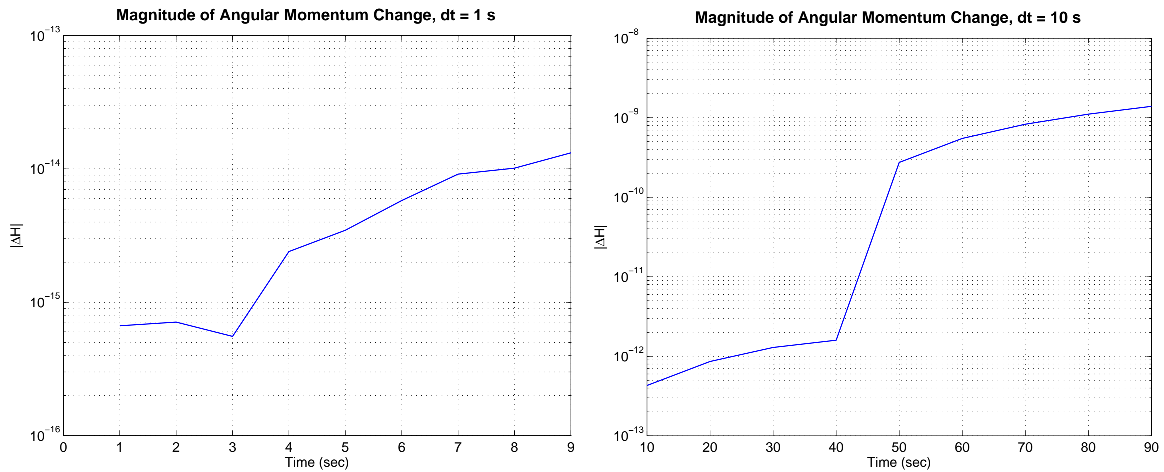
Listing:

```
1 %
   --------------------------------------------------------------------------------
2 %  Calculate angular momentum of boom system and update the body rates.
3 %
   --------------------------------------------------------------------------------
4 %  Form:
5 %  [hI, hW] = HGimballedBoom( x, g, axis, aDotNew )
6 %
   --------------------------------------------------------------------------------
7 %
8 %  ------
9 %  Inputs
10 %  ------
11 %  x              (15,1)    The state vector [r;v;q;omega;a;aDot]
12 %  g              (1,1)     CAD model
13 %  axis           (3,2)     Rotation axes of gimbals
14 %  aDotNew        (2,1)     New gimbal rates
15 %  hW             (3,1)     Stored angular momentum
16 %
17 %  -------
18 %  Outputs
19 %  -------
20 %  hI             (3,1)     Inertial angular momentum
21 %  hW             (3,1)     Updated stored momentum
22 %
23 %
   -------------------------------------------------------------------------------
```

5.6.3 Example

The demo `BoomMomentumDemo` verifies that the model conserves momentum for any set of body and gimbal rates given no external forces and torques. The user can change the simulation time step to verify that smaller steps result in better conservation. The simulation uses a fourth order Runge-Kutta integrator.

Figure 5.5: Momentum Conservation Verification Demo Results



CHAPTER 6

SAIL ATTITUDE ACTUATORS

This chapter describes how to model certain common sail attitude control schemes, including

- Sliding masses
- Vanes
- Gimballed boom

6.1 Sliding Masses

Sliding masses involves moving mass around the sail to create an offset between the center of mass and center of pressure of the sail, resulting in a torque. The torque model is

$$\vec{T} = (\vec{r}_{cp} - \vec{r}_{cm}) \times \vec{F}$$

Control of both transverse axes can be achieved with a single mass on a two-axis track system or with two separate masses on tracks at right angles. The dynamics of this mechanism can be treated in multiple ways depending on how the masses move along the tracks. For example, stepping motors can be modeled as achieving fixed rates instantaneously. Otherwise the masses can be treated using multibody dynamics.

The CAD model `PlateWithMasses` builds a 100 m specular sail model with two masses. The sail normal is along the body x axis and the masses are assigned to the y and z axes. Each mass is given its own body in the CAD structure to facilitate describing its position and recomputing the resulting mass properties. Each sliding mass is 10 kg, the core mass is 100 kg, and the sail mass is 30 kg.

The +Y mass body is created with the lines

The mass components are created with

The masses are assigned to be inside the sail so that their surface properties will be ignored during the disturbance calculations.

The script `BallastMass1Axis` demonstrates this CAD model with the actuation functions `TorqueToCM` and `CMToMassPositions`. The script demonstrates commanding a new orientation of the sail normal. A simple torque model is used instead of the full disturbance model; the force is assumed to be along the x-axis. The sail normal is determined using pitch and yaw angles from a 3-2-1 Euler angle set.

$$\hat{n} = \begin{bmatrix} \cos\theta_y \cos\theta_z \\ \cos\theta_y \sin\theta_z \\ -\sin\theta_y \end{bmatrix}$$

For small angles, this reduces to

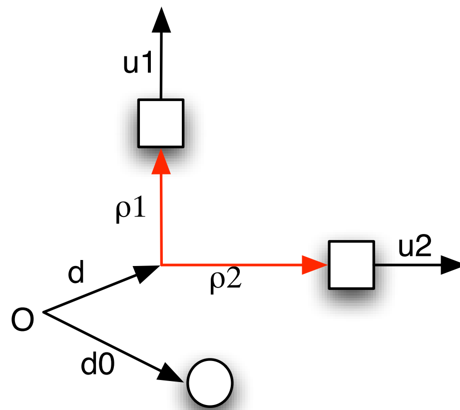
$$\hat{n} = \begin{bmatrix} 1 \\ \theta_z \\ -\theta_y \end{bmatrix}$$

These angles can be used as errors for determining y and z torques for the ballast mass system. Euler angles cannot be used directly since rotation about the y and z axes when these Euler angles are nonzero results in changes of the x Euler angle.

The actuator demand is computed in the lines

The masses of each body (including the core) are passed in `mControl`, the initial offset vectors `dOffset` are assumed to be zero, and the unit vector for control of each mass is given in `uControl`. The center of pressure `Cp` is assumed to be at the origin. The positions of the vectors along the track are designated ρ. Figure 6.1 shows the geometry.

Figure 6.1: Sliding Mass Geometry



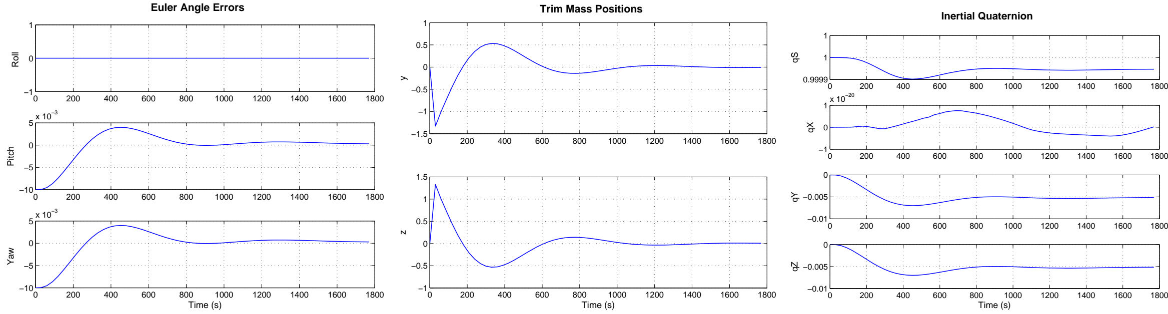
The torque model is implemented in the lines

The `rHinge` fields are used to indicate how eventual integration with the full disturbance model would be performed.

The dynamics assume fixed rates for the masses, which are computed in this discrete simulation so that the masses will achieve the commanded position on the next step (30 seconds). The dynamics are implemented in the function `FMovingBody`. The rate computation is performed in the lines

This simple setup will work for one or two-axis control if the angles are small. For example, in Figure 6.2 the step angle command is 0.01 in both transverse axes.

Figure 6.2: Two axis control using moving masses on a specular sail



6.2 Vanes

Vanes refer to smaller sections of sail membrane mounted on a rotating mechanism. Two vanes set at equal and opposite angles can be used to control one axis using a windmill effect and more vanes can be used to control multiple axes. The torque produced is nonlinear since the force on the vanes follows the same solar pressure force laws as the regular sail membrane (cosine squared), making three-axis control with vanes challenging. We will show an example where a pair of vanes is used for roll control (control about the sail normal axis).

Each vane applies a force on the sail at its center-of-mass, which are summed to produce a torque.

$$\vec{T} = \sum_i \vec{r}_{cm_i} \times \vec{F}_{s_i}$$

Computing the desired angle for a control vane requires a model of the force produced. Assuming specular reflection, the force is

$$F_i = 2P_s A \cos\gamma \, \hat{n}$$

where γ is the total angle between the vane normal and the sun vector, so that $A\cos\gamma$ is the projected area of the vane.

Assuming also that the vane is aligned with the sail and rotates around a single axis, the vane normal vector in the sail body frame is

$$\hat{n} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ 0 \end{bmatrix}$$

The vane may first be canted backwards away from the plane of the sail. This is done in some sail designs for static stability. If the cant angle is ϕ, then the sail normal with the cant angle included

is

$$\hat{n} = \begin{bmatrix} \cos\theta\cos\phi \\ \sin\theta \\ -\sin\phi\cos\theta \end{bmatrix}$$

The torques around the transverse axis cancel while the torques around the x axis sum, resulting in a control torque model of

$$T_x = 2r_{cm}\sin\theta \cdot (2P_s A\cos\phi)$$

The CAD model `PlateWithVanes` demonstrates a pair of vanes on a 100 m square specularly reflective sail (plate). The vanes are canted back 25 degrees and each have an area equal to 5% of the main sail area. The model has three bodies, with the plate as the core body and each vane having its own body. This does not imply any multi-body dynamics but serves to facilitate specifying the orientation of the vanes.

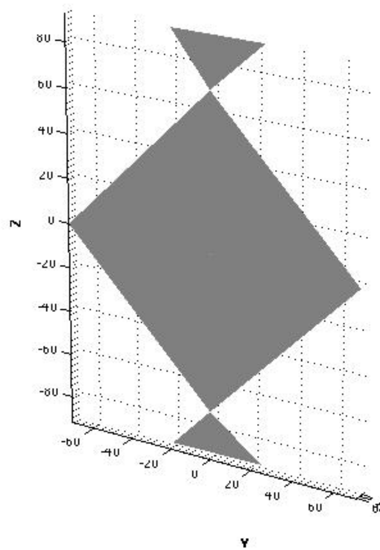
The vane bodies are created with using `CreateBody`. ______________ *PlateWithVanes*

The vane components are created using `CreateComponent`. ________ *PlateWithVanes*

As indicated in the comment, in this case we are not computing the inertia of the vanes but are treating them as masses at the end of the sail booms. The inertia could be computed from the vertices `v` using `VFToMassStructure`, which would result in the vehicle inertia changing when the vanes are rotated. For initial verification of the CAD and disturbance set up it is easier to neglect this.

The `sigmaS` property being set to 1 indicates that the vane is treated as 100% specularly reflective. The resulting CAD model is shown in Figure 6.3.

Figure 6.3: 3D view of PlateWithVanes



This CAD model is demonstrated with the full disturbance model in `VaneControl1Axis`. The disturbance model profile is updated each step with a new angle for each vane which is about the

specific vector. This example does not include any dynamics for the vanes, they are assumed to achieve the commanded angle instantaneously compared to the time scale of the attitude controller.

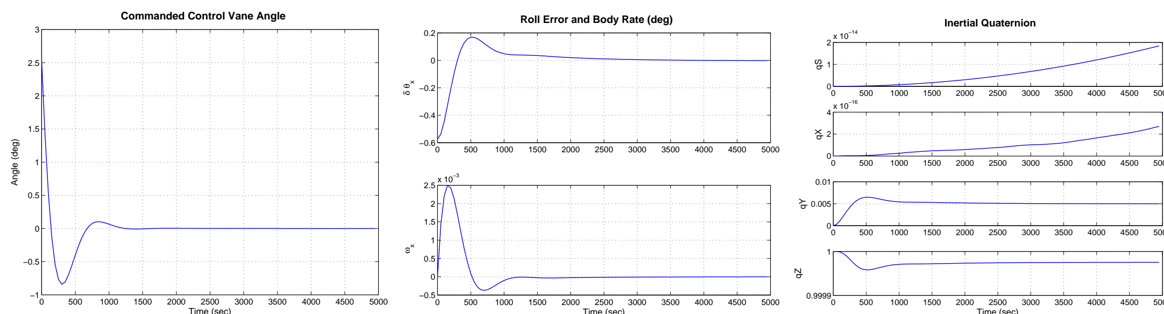
The profile for the vanes is initialized by specifying the body ID for each vane and the body rotation axes (z). The angles are initialized to zero. ______ *VaneControl1Axis*

The commanded angle is computed from the torque demand and applied to the profile inside the simulation loop. ______ *VaneControl1Axis*

Determining the angle requires knowledge of the vane area and distance to its center of mass (`lBoom`).

The script demonstrates a 0.01 radian (0.57 degree) step command angle in the sail roll angle. The sail is initially pointed directly at the sun using the quaternion produced by `QSail`. The maximum vane angle commanded is 2.5 degrees. Now that the geometry and profile setup has been verified, the model could be extended with nonideal properties, true inertias, and non-zero sail sun angle.

Figure 6.4: Roll control using specular vanes on a specular sail



6.3 Gimballed Boom

A gimballed boom is one form of center of mass actuation; the sliding masses discussed in the next section are another. A boom with two perpendicular gimbals can be oriented anywhere in the hemisphere. Practically, boom motion would probably be restricted to some cone.

The torque model is

$$\vec{T} = (\vec{r}_{cp} - \vec{r}_{cm}) \times \vec{F}$$

Modeling a particular set of gimbals means selecting which axis each gimbal controls and developing the kinematics for determining which gimbals angles should be selected for a given boom orientation. The function `TorqueToCM` can be used to determine where the center of mass should be in the transverse plane to produce the commanded torque given a center of pressure vector. The boom geometry (length and mass) is then used to determine the boom unit vector which will result in this center of mass. The quaternion or gimbal angles, if used, are then computed.

The Core Toolbox has a dynamics model, `TBModel`, which can be used to model the full two-body dynamics. This model requires the hinge torque at the gimbal interface to be an input to

the dynamics. Alternatively, the model `TwoBodyTwoRate` can be used to model two gimbals at specific fixed rates. This model assumes that the change in core momentum produced when the gimbals achieve new rates is stored in a momentum sink, such as a set of reaction wheels. Otherwise, when the boom is moved, the sail will also move to conserve angular momentum.

The CAD model `PlateWithBoom` builds a 40 m specular sail model with a 10 m gimballed boom. The sail normal is along the body x axis. The boom is a separate body from the core.

The script `BoomActuation` demonstrates this CAD model with a double gimbal model and the full disturbance model. The first (inner) gimbal is along the boom x axis and the second along the y axis. The angles α and β are assigned to these gimbals. The gimbal dynamics are modeled as fixed-rate using `TwoBodyRateModel`. Each gimbal angle and rate is a state. In this script, each gimbal is given a target angle sequence and the boom slews as the fixed rate allows. The resulting torques from the full disturbance model can then be verified.

The gimbal angles are initialized in the CAD profile with the given rotation axes. Both angles are referenced to the second body (the boom).

```
p.angle     = [0;0];
p.axis      = [1 0;0 1;0 0];
p.body      = [2 2];
```

The boom unit vector - nominally along the x axis - can be determined from the gimbals angles as

```
uB  = Eul2Mat([alpha;0;0])'*Eul2Mat([0;beta;0])'*[1;0;0];
```

The gimbal rates for the fixed rate model are computed using

```
 [aDot,angleCommand] = GimbalRates( x(8:9), [alpha;beta], aNom,
    dT );
```

where `aNom` is the nominal gimbal rate, `x(8:9)` are the current gimbal angles, and `[alpha;beta]` are the commanded gimbal angles. `GimbalRates` will look for the closest gimbal angle set which places the boom unit vector in the correct location, since there are two sets of angles defining every vector, (α, β) and $(\alpha + \pi, -\beta)$, so the actual commanded angles are returned as well as the gimbal rates. The function uses a discrete time step so that if the angles can be reached within `dT` at the nominal rate, an average rate is computed so the target angles are reached exactly.

Once the new rates are computed, the body rates must be adjusted before the integration step. This is accomplished with

```
 [hPlot(:,k), hW] = HGimballedBoom( [zeros(6,1);x], g, p.axis,
    aDot, hW );
 x(10:11) = aDot;
```

where the angular momentum is stored in a plotting array. The stored momentum `hW` is updated so that the sail core rates do not change as the boom moves. The sail rates will change only as a result of the torque applied via the new sail center-of-mass.

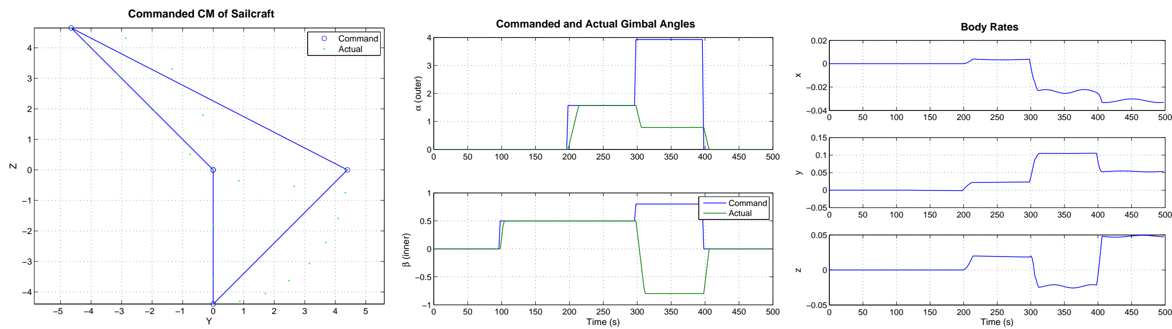
The integration is performed with

```
x = RK4( 'TwoBodyRateModel', x, dT, t, f, tq, g, p, hW );
```

After integration, the CAD profile is updated for the next step with the achieved gimbal angles.

```
p.angle = x(8:9);
```

Figure 6.5: Double-gimballed boom actuation demo showing a sequence of commanded gimbal angles



CHAPTER 7

ORBIT DYNAMICS AND EPHEMERIS

7.1 Function Overview

In this chapter we will discuss functions in the OrbitDynamics and SailEphem folders. The orbit dynamics functions are listed below using the `help` command.

```
>> help OrbitDynamics
  Sail/OrbitDynamics

  F
     FOrbitGeneral           - General gravity model allowing point mass
                               and harmonic models.
     FOrbitSingle            - Gravity model allowing a point mass or
                               harmonic models for a single body.
     FRotatingFrame          - Three-body dynamics in the rotating frame
                               with a solar sail.
     FRotatingLagrange       - Sail orbit RHS in a 3 body dimensionless
                               rotating frame.
     FRotatingPlanet         - Sail orbit RHS in a planet-centered
                               cylindrical rotating frame.
     FRotatingSun            - Sail orbit equations in a sun-centered
                               cylindrical rotating frame.
     FSailCombined           - Simulation right hand side. Calls all
                               needed portions in order.
     FSailEarthMoon          - Sail dynamics with spherical harmonic
                               models of the Earth and Moon gravity.
     FSailEarthSun           - Sail dynamics in the Earth-Sun three-body
                               system.
     FSailGuidance           - Right-hand-side for sail with fixed cone
                               and clock angles (McInnes).

  I
     InitializeSailGravity   - Initialize the sail simulation gravity
                               model.
     InitializeSailSim       - Initialize the sail simulation data for
                               common cases.
```

```
R
   RHS2DOrbit                 - This function is for the planar orbit
                                problem.
   RHS3DOrbit                 - This function is the right-hand-side for
                                the 3D orbit optimization problem.
   RHSCartesian               - Right-hand-side for point-mass cartesian
                                orbit with external acceleration.
   RHSCartesianRadialAccel    - Right-hand-side for point-mass cartesian
                                orbit with radial acceleration.
   RHSEquinoctial             - Right-hand-side for equinoctial elements.
   RHSOpt2DOrbit              - Right-hand-side for the planar orbit
                                optimization problem.
   RHSOpt3DOrbit              - This function is the right-hand-side for
                                the 3D orbit optimization problem.

V
   VarEqSailPlanet            - Sail variational equations in state space
                                form for a planet-centered orbit.
```

The special sail ephemeris functions are listed below.

```
>> help SailEphem
  Sail/SailEphem

  S
     SailEphemAlmanac - Location of planets relative to Sun in ecliptic
                         frame using the almanac.
     SailEphemEarth   - Location of Earth relative to Sun in the ecliptic
                         frame.
     SailEphemJPL     - Location of planets relative to Sun in ecliptic
                         frame using JPL ephemerides.

  Sail/Demos/SailEphem

  P
     PlanetDemo        - Demonstrate JPL ephemeris for the solar system.
```

7.2 Orbit Dynamics

The Solar Sail Module enables you to perform simulations with Keplerian orbit kinematics, simple point-mass orbit dynamics, spherical harmonic gravity models, and multibody models combining spherical harmonic and point-mass perturbations. The functions include

- FOrbitSingle
- FOrbitGeneral
- FSailGuidance

`FOrbitSingle` and `FOrbitGeneral` have the same form, and the general version allows multiple bodies, some of which may have spherical harmonic models. `FSailGuidance` is an

example of orbit dynamics combined with sail guidance into a single right-hand-side; it computes a sail acceleration from McInnes cone and clock angles and adds it to the point mass orbit acceleration. This function is demonstrated in `LocalOptimalSim`.

The Core Toolbox also has gravity models which can be used in custom right-hand-sides, for example `FOrbCart` provides a Cartesian point-mass orbit for any gravitational parameter.

The functions `RHSOpt2DOrbit` and `RHSOpt3DOrbit` are for use with the `TrajectoryOptimization` function.

7.2.1 Combined right-hand-side

The function `FSailCombined` allows you to combine the ephemeris, attitude dynamics, and orbit dynamics into a single right-hand-side call. It also allows you to specify an environment and disturbance function.

The coupled right-hand-side has the form

```
xDot = FSailCombined( t, x, jD, p, d )
```

The format is designed for use with `ode113`. `t` is the mission elapsed time in seconds `x` is the state vector. The state vector is always

```
[r;v;q;omega;additional attitude states]
```

Thus the first 4 terms, 13 states in all, define the basic rigid body states: position of the center-of-mass, velocity of the center-of-mass, quaternion from the ECI from to the body frame and angular rate defined in the body frame. `p` contains the profile data. `d` contains any additional information needed by the separate dynamics functions. At a minimum `d` includes the fields

- `ephemeris` - the name of the ephemeris function.
- `environment` - the function providing the environment data.
- `disturbance` - the name of the external disturbance function.
- `attitude` - the name of the attitude dynamics function.
- `orbit` - the name of the orbit dynamics function.
- `guidance` - the name of a guidance function in place of attitude dynamics.
- `g` - data structure generated by the CAD script.
- `jD0`

This list of fields can be obtained by calling `FSailCombined` with no inputs.

```
>> d = FSailCombined()
d =
      ephemeris: []
```

```
    environment: []
    disturbance: []
       attitude: []
          orbit: []
       guidance: []
            jD0: 2451545
         center: 1
              g: [1x1 struct]
```

Any non-empty value for the `guidance` field will override the attitude dynamics function. The attitude will be fixed to the output of the guidance functions before the disturbance function is called. The `center` field is an index into the list of planets stored in the orbit dynamics model. The dynamics functions are of the form:

Ephemeris

```
rB = FEphemeris( jD, d )
```

Attitude

```
xADot = FAttitude( t, x, f, tq, d )
```

or

```
q = FGuidance( t, x, d, env )
```

For examples, see `FSailRB` or `SPIGuidance`.

Orbit

```
xODot = FOrbit( t, x, rB, center, jD, accel )
```

The orbit function must also be able to return a list of names of the stored planets using the call

```
planets = FOrbit( 'planets' )
```

See for example `FOrbitGeneral` and `FOrbitSingle`.

Environment and Disturbances

The environment data, for example the solar flux at the spacecraft location, is obtained first, then passed to the disturbance function. The profile is updated with the position r, velocity v, quaternion q and Julian date jD using the mission elapsed time and state input to `FSailCombined`. The `planet` parameter is updated internally using the `center` field of `d`.

```
env = FEnvironment( planet, p, d )
```

```
[f, tq] = FDisturbance( g, p, env, d )
```

The `force` and `torque` from the disturbance routine are passed to the attitude and orbit routines.

7.2.2 Specialized Coordinate Systems

The Solar Sail Module contains several orbit dynamics functions that are in specialized coordinate systems, for instance for studying trajectories near Lagrange points.

7.3 Ephemeris

The Core Toolbox has 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]

The toolbox also 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. A special format is available for use with FSailCombined, see SailEphem.

```
>> help SailEphem

  Sail/SailEphem

  S
     SailEphemAlmanac - Location of planets relative to Sun in
        ecliptic frame using the almanac.
     SailEphemEarth   - Location of Earth relative to Sun in the
        ecliptic frame.
     SailEphemJPL     - Location of planets relative to Sun in
        ecliptic frame using JPL ephemerides.
```

Each of these wrapper functions has the same format to facilitate switching between them and reuse of code. A data structure is used to pass in the parameters, such as planet IDs. SailEphemEarth is used in the demos SailCombinedDemo and ST9CombinedDemo. SailEphemAlmanac is the default ephemeris function used in FSailCombined if no other ephemeris function is

[1] Explanatory Supplement to the Astronomical Almanac (1992.) Table 5.8.1. p. 316.
[2] The 1993 Astronomical Almanac, p. D46.
[3] Montenbruck, O., Pfleger, T., Astronomy on the Personal Computer, Springer-Verlag, Berlin, 1991, pp. 103-111.
[4] The 1993 Astronomical Almanac, p. C24.
[5] Montenbruck and Pfleger, p. 36.

specified. The header for the almanac function is shown below. It explains that the function is essentially a wrapper for `PlanetPosition`.

```
>> help SailEphemAlmanac
 -----------------------------------------------------------------------------

    Location of planets relative to Sun in ecliptic frame using
       the almanac.
    This function obtains the moon position using MoonV1. The
       function
    PlanetPosition must be initialized with the desired planet IDs
        before
   calling this function. This format is for use with
      FSailCombined.
   Note that the planet IDs are 1-9, the Moon ID is 10, and the
      sun ID is 0.

   If called with a vector of dates will compute the location of
      the
   center only.

    See also PlanetPosition and PlanetPosJPL.
 -----------------------------------------------------------------------------

    Form:
    rB = SailEphemAlmanac( jD, d )
 -----------------------------------------------------------------------------



    ------
    Inputs
    ------
    jD        (1,1)     Julian date(s).
    d          (:)      Data structure.
                        .kP      Array of planet IDs
                       .center   Index of gravity center, used in
                          batch mode

    -------
    Outputs
    -------
    rB        (3,:)     Position of planets OR positions of center
        planet

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

It's easy to create a quick test of the ephemeris. For instance, to get the positions of Venus, Earth,

Moon, and Mars, note that the planet IDs are 2, 3, 10 and 4, respectively, and just do:

```
jD = Date2JD;                               % use today's date
PlanetPosition('initialize',[2 3 4]); % only pass major planets to
    PlanetPosition
d.kP = [2 3 4 10];                          % add the moon at the end
rB = SailEphemAlmanac( jD, d )

rB =
   -9.213e+07   1.3764e+08   4.8653e+07   1.3729e+08
   5.5701e+07   5.6175e+07   2.0781e+08   5.6266e+07
     6.08e+06      -1777.2   3.1587e+06        10035
```

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 MATLAB functions require binary versions of the ephemeris which are system-dependent and therefore must be created by users. See the Spacecraft Control Toolbox User's Guide for more information. Note that the `AsteroidTrajectory` demo uses the `SolarSysJPL` and `PlanetPosJPL` ephemeris functions directly.

CHAPTER 8

ANALYSIS

This chapter goes through several examples of different kinds of analysis which can be performed with the toolbox. Detailed analysis with specific geometry for disturbances or actuation generally requires the creation of a CAD model, so that is discussed first.

8.1 Creating a CAD Model

A two body model is created using a script which employs the SCT function `BuildCADModel`. For the purpose of this demonstration the sail is modeled as a core box with an attached 40 m square sail. The core box has a two degree of freedom hinge to which is mounted a 10 m mast with another box at the end. The mast is modeled as another box. Thus the sail consists of three boxes, a two degree of freedom hinge actuator, and a flat plate sail. All components are assumed to be rigid. The core box and sail are assumed to be the *core* and the mast and mast tip box are the second body which will be called the *boom*.

The example script name is `SailWithBoom`. When you run the script, the CAD Builder GUI will appear to let you view the data in the resulting model. If you select the menu on the left hand side you will get a list of components. If you push the Vehicle tab you will bring up a pane that will allow you to view the spacecraft as a whole. These views are shown in Figure 8.1 on the next page. The two figures in Figure 8.2 on the following page show an exploded view of the core box, gimbal, mast and boom box on the left and the entire spacecraft on the right.

The script is organized with all properties at the top. The GUI is initialized and then the bodies are defined. Next, the components are defined using `CreateComponent`. This script has five components: the core box, gimbal cylinder, sail, mast, and mast box. The sail uses the special *sail* component type which will invoke the combined thermal and optical disturbance model. Each component is assigned to a specific body, so that the properties for each body can then be computed by the CAD builder. The finished model is extracted and stored as a mat-file in the SailData directory of the module.

The sail component type bears elaboration. All components have optical properties which are

Figure 8.1: Solar sail CAD model viewer, Component and Vehicle tabs

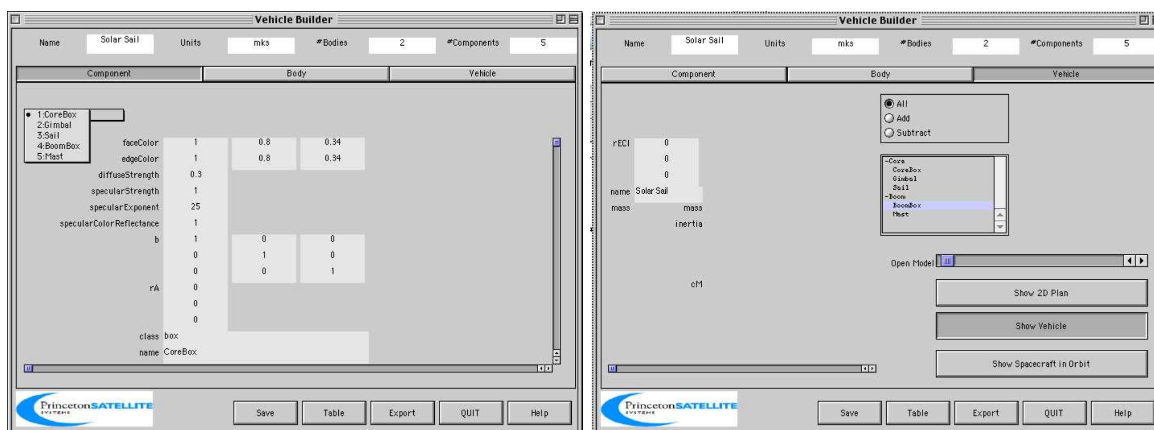
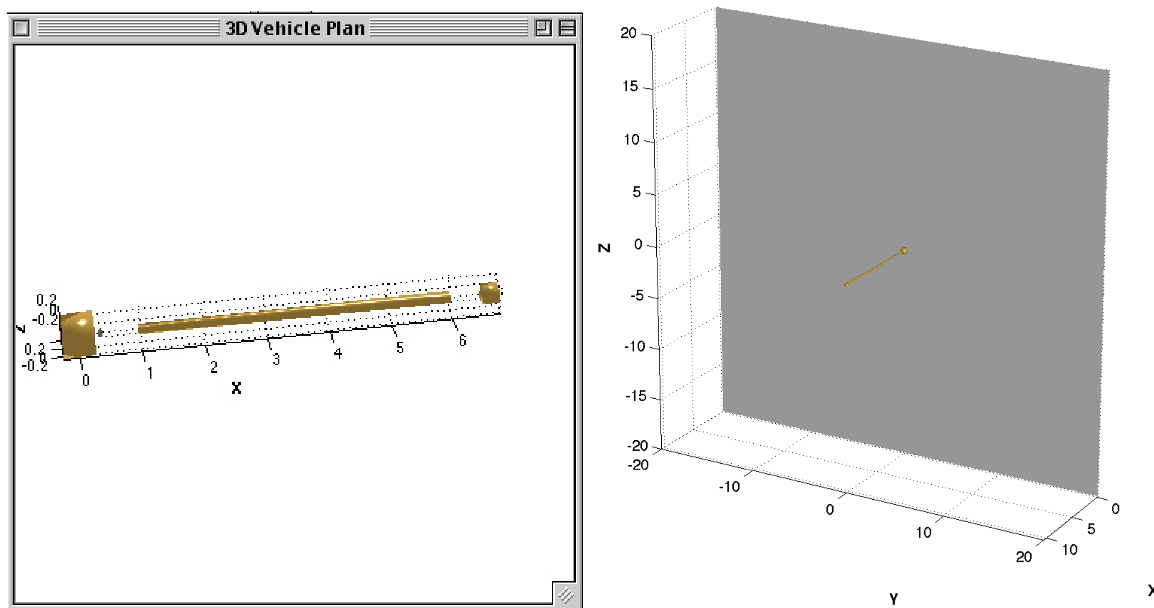


Figure 8.2: 3D views of the spacecraft



assumed to be uniform over the component, and such components are generally assumed to be three-dimensional solids. The sail type of component is assumed to be a membrane, and receives special treatment in the `SailDisturbance` function, where the combined thermal and optical force function `SolarPressureForce` is used. The membrane must have both front and back properties specified separately. Each of these could be an array so that each face of the component has different properties. This sail has only two faces which are specified explicitly with the vertices, and the front and back properties are each uniform. The mirror color is used to make the sail appear shiny in MATLAB 3D plots, but has no bearing on the optical properties used in the disturbance computation.

```
m = CreateComponent( 'make', 'sail','name','Sail','body',1,...
                     'mass', massSail, 'faceColor', 'mirror','rA'
                        ,[-coreWidth/2;0;0],...
```

```
                    'sigmaS', [0.9 0.85], 'sigmaD', [0.02 0.05],
                       'sigmaA', [0.08 0.1], 'emissivity', [0.03,
                        0.3],...
                    'vertex',v ,'face', [1 2 3; 1 3 4], 'inside',
                        0 );
```

The sail is specified as two triangles using the `vertex` and `face` fields of MATLAB graphics objects. Recall that all PSS CAD models are stored as sets of triangular patches. The order of the vertices in the face field will determine the direction of the normal of the patch area. The optical coefficients for specular reflection, diffuse reflection, and absorption are stored in the `sigma` fields.

Note that the sail mass structure is passed in using the variable `massSail`. The `Inertias` function is used to compute the inertia of a plate with the desired sail mass and dimensions. An offset in the center of mass would be entered via the mass structure.

The resulting CAD model is stored as a data structure. You can view the fields by displaying the final structure `g`.

```
g =
         name: 'Solar␣Sail'
        units: 'mks'
         body: [1x2 struct]
    component: [1x5 struct]
       radius: 1.012654309228969e+01
         mass: [1x1 struct]
```

We want to check the normals of the sail to make sure that they are pointing forwards in the coordinate frame. We can view the names of the model's components,

```
>> {g.component.name}
ans =
    'CoreBox'    'Gimbal'    'Sail'    'BoomBox'    'Mast'
```

and we see that the sail is the third component, so we can examine all of its properties by displaying that substructure.

```
>> g.component(3)
ans =
                   faceColor: [1 1 1]
                   edgeColor: [1 1 1]
              diffuseStrength: 0
             specularStrength: 1
             specularExponent: 100
    specularColorReflectance: 1
                           b: [3x3 double]
                          rA: [3x1 double]
                           v: [4x3 double]
                           f: [2x3 double]
                           a: [2x1 double]
                           n: [2x3 double]
```

```
                   r: [2x3 double]
              radius: [2x1 double]
          deviceInfo: []
               class: 'sail'
                name: 'Sail'
             optical: [1x1 struct]
            infrared: [1x1 struct]
             thermal: [1x1 struct]
               power: [1x1 struct]
                aero: [1x1 struct]
            magnetic: [1x1 struct]
                mass: [1x1 struct]
              inside: 0
                  rF: [1x1 struct]
                body: 1
        manufacturer: 'none'
               model: 'generic'
```

In this listing we can see that the sail component is of the *sail* class. The normals are stored in `n` and the areas in `a`. `r` gives the vector to the centroid of each patch from the origin of the vehicle coordinate system.

```
>> g.component(3).n
ans =
     1     0     0
     1     0     0
>> g.component(3).r
ans =
    -2.500000000000000e-01    -6.666666666666668e+00
       6.666666666666668e+00
    -2.500000000000000e-01     6.666666666666668e+00
      -6.666666666666668e+00
>> g.component(3).a
ans =
   800
   800
```

8.2 Performing a Disturbance Analysis

There are two examples of disturbance analysis which use the `SailWithBoom` model. Each demo uses the `SailEnvironment` and `SailDisturbance` functions in sequence.

- `EarthOrbitDisturbances.m`
- `HelioDisturbances.m`

In the case of the Earth orbit example, the location of the Earth relative to the sun must be computed and passed to the disturbance function. This is required to allow the user to use any ephemeris without embedding fixed options into the disturbance function itself. The entire orbit is computed a priori using `SolarSys` and passed into the functions in a single call.

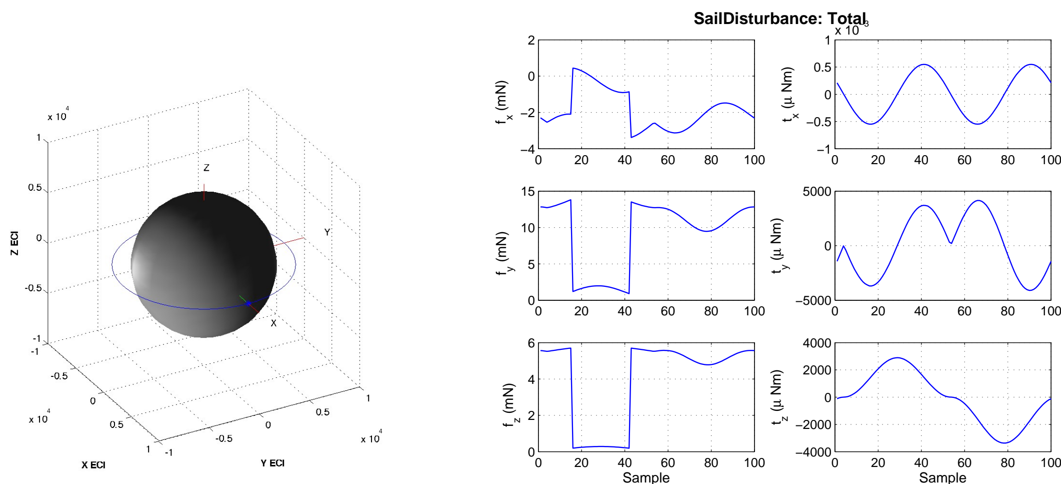
```
% Earth orbit around the sun
%---------------------------
[planet, aP, eP, iP, WP, wP, LP, jDRefP] = Planets( 'rad', 'Earth'
    );
[rX0, rY0, rZ0] = SolarSys( iP, WP, wP, aP, eP, LP, planet, jDRefP
   , JD2T( p.jD  ) );
p.rPlanetH = Constant('au')*[rX0;rY0;rZ0];
```

The profile, including the orbit, is stored in `p` while the disturbance model flags are stored in `d`. The environment data including solar flux is passed to the disturbance function in `e`. You can view each of these structures from the command line after running the demo.

```
e = SailEnvironment( d.planet, p, d );
SailDisturbance( g, p, e, d );
```

`SailDisturbance` automatically generates a set of plots when it is called without any outputs. The reference orbit and total disturbances are shown for reference in Figure 8.3. The sail follows a sun-pointing profile throughout the orbit via the function `QSunPointing`.

Figure 8.3: Sample of Earth orbit disturbances demo output

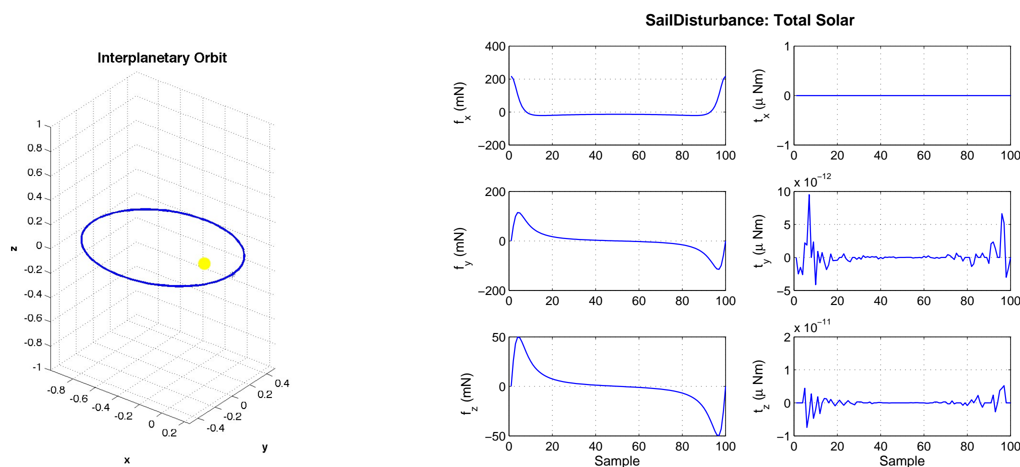


The heliocentric demo uses the same sun-pointing profile with an orbit that has an apogee of 1 AU and a perigee of 0.25 AU. Many of the disturbances, such as magnetic field and planetary radiation, are turned off as they are not relevant in this type of orbit. The only disturbance remaining is solar radiation pressure. The sample output is shown in Figure 8.4 on the following page.

In this case the environment structure `e` is

```
>> e
e =
```

Figure 8.4: Sample of heliocentric orbit disturbances demo output




```
            planet: 'sun'
              uSun: [3x100 double]
         radiation: 0
            albedo: 0
            radius: 695990
                mu: 132712438000
          altitude: [1x100 double]
         solarFlux: [1x100 double]
   eclipseFraction: [1x100 double]
     radiationFlux: [1x100 double]
        albedoFlux: [1x100 double]
               rho: [1x100 double]
            bField: [3x100 double]
```

8.3 Simulating the Attitude Dynamics

The most basic example of attitude dynamics is the evolution of a rigid body's attitude with disturbances. The very simple script `AttitudeDemo` shows this for a flat plate model at a single point in a heliocentric orbit. The sail starts with an initial attitude and body rate, and the attitude is then evolved in a simulation loop. The rigid body dynamics are in the function `FRB` which is designed for use with PSS' fourth order Runge-Kutta integrator `RK4`. This example provides a basis for adding a discrete controller; the resulting torque would be added to the disturbance torque at each step.

The initial states are defined in the lines

```
% Quaternion (sun pointing with an offset)
%-----------------------------------------
q0   = QMult(QSunPointing( -Unit(p.r) ), AU2Q(0.05,[0;1;0]) );
```

```
% Initial body rate
%------------------
w0 = [0.001;0.0005;-0.001]/100;
```

and the integration is performed in the line

```
  % Integrate the rigid body dynamics
  %-----------------------------------
  x = RK4( 'FRB', x, dT, t, inr, invInr, tS.total );
```

where the inverse inertia, which is fixed, has ben previously computed. `tS` is the torque structure returned by the disturbance function.

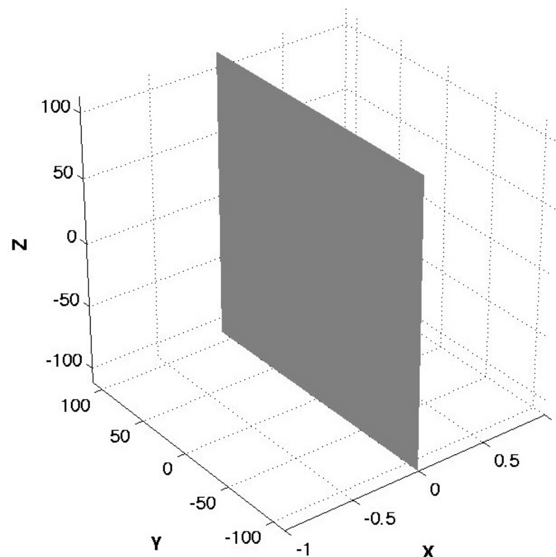
The simulation runs for 100 steps with a time step of 100 seconds. The resulting quaternion and body rates are plotted along with the force and torque computed by the disturbance function. In this case, the sail is symmetric so that the solar torque is zero. This could be easily changed for demonstration purposes by adding a small offset to the center-of-mass.

This demo also shows you how to quickly visualize a stored CAD model. The same function that is used by the CAD builder to view a model after you have run a CAD script can be used independently at any time with the CAD data structure to get a 3D view.

```
DrawSCPlanPlugIn('initialize',g)
view(3); axis equal; axis square;
```

The resulting view of the plate is shown in Figure 8.5.

Figure 8.5: Flat plate model used in `AttitudeDemo`



8.4 Boom Control Demo

This demo, `BoomControl.m`, involves a boom with 1-2 gimbals; that is, the first gimbal is along the body X axis and the second gimbal is along the y axis. This is shown in Figure 8.6 and Figure 8.7.

Figure 8.6: Boom frames of gimbals from two viewpoints

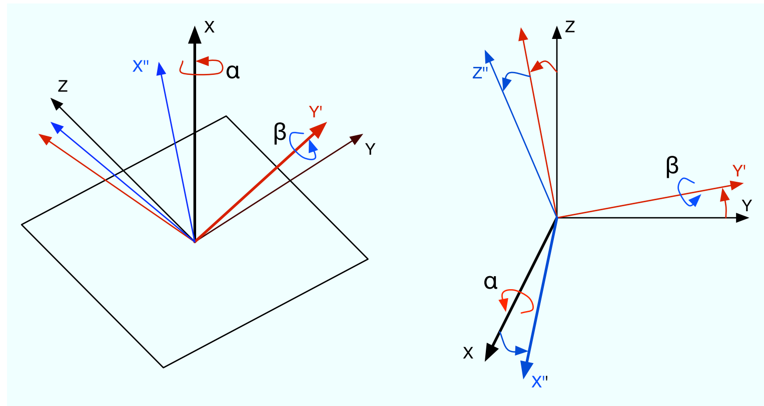
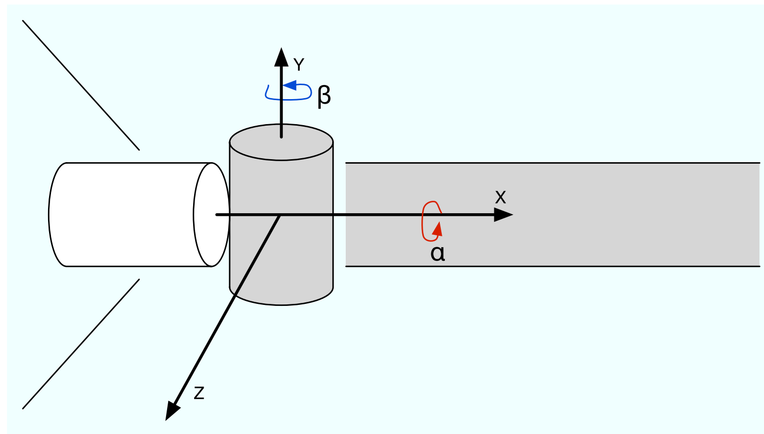


Figure 8.7: Boom gimbals



The simulation update step includes the disturbance calculation, control calculation, momentum adjustment step, and finally the integration of the RHS.

Listing: Disturbance calculation, from CAD model g and profile p

```
%---------------------------
% Compute the disturbances.
%---------------------------
[f,tq] = SailDisturbance( g, p, d );
tqPlot(4:6,k) = tq.total;
```

The control calculation has several parts, including the calculation of the control acceleration, resulting gimbal angles, and finally applying the rate velocity limits. It is desirable to have more intelligent selection of alpha and beta angular error, since one set of rotations may result in less total angle change than another. The simplest logic is shown here.

Listing: Control update

```
%-----------------
% Boom control
%-----------------

% Sign conventions
%-----------------
if( qC(1) < 0 )
  qC = -qC;
end
qIToB = x(1:4);
if( qIToB(1) < 0 )
  qIToB = -qIToB;
end

%
   ------------------------------------------------------------------------
% Euler angle error in body frame. We are only considering Y and Z
   errors.
% Determine from commanded normal in body frame.
%
   ------------------------------------------------------------------------
uSailB = QForm( qIToB, uI );
errY   = uSailB(3);
errZ   = -uSailB(2);
eulErr = [errY;errZ];

% Control law and torque
%-----------------------
angleError = [0;eulErr];
accel  = cC*xN + dC*angleError;
xN     = aC*xN + bC*angleError;
tExt   = -g.mass.inertia*accel;

% Cp/Cm offset
%-------------
cM = pinv(aBoom)/cos(cone0)^2*-tExt/(mB/(mC+mB));
mCM = Mag(cM);
if (mCM >= rBoomCM)
  % requesting CM beyond reach of boom
  hB = 0;
else
  % Boom unit vector component along normal
  hB = sqrt(rBoomCM^2 - mCM^2);
end
uB = Unit([hB;cM]);
```

```
43
44  % Gimbal angles (12 sequence)
45  %----------------------------
46  beta  = acos(uB(1));
47  alpha = atan2(uB(2),-uB(3));
48
49  % Gimbal rate limiting
50  %---------------------
51  [aDot,angleCommand] = GimbalRates( x(8:9), [alpha;beta], aNom, dTi );
```

The last part of the simulation step is the angular momentum adjustment for the new gimbals rates and the integration of the attitude right-hand-side. The profile is then updated for the disturbance calculation in the next step.

Listing:

```
1   %-------------
2   % RHS
3   %-------------
4   [hPlot(:,k), hW] = HGimballedBoom( [zeros(6,1);x], g, p.axis, aDot, hW )
       ;
5   x(10:11) = aDot;
6
7   % Integrate
8   %----------
9   x = RK4( 'TwoBodyRateModel', x, dTi, t, f, tq, g, p, hW );
10
11  % Update profiles
12  %----------------
13  t    = t + dTi;
14  p.jD = p.jD + dT/86400;
15  p.q  = x(1:4,:);
16  p.angle = x(8:9);
```

The sample maneuver involves 0.1 rad clock angle (for a cone angle of 0.5) from a heliocentric orbit. That is, `clockCommand` is 0.1 while `clock0` is 0. In Figure 8.8 on the facing page we can clearly see the boom ramp up to the commanded position at the maximum gimbal rate.

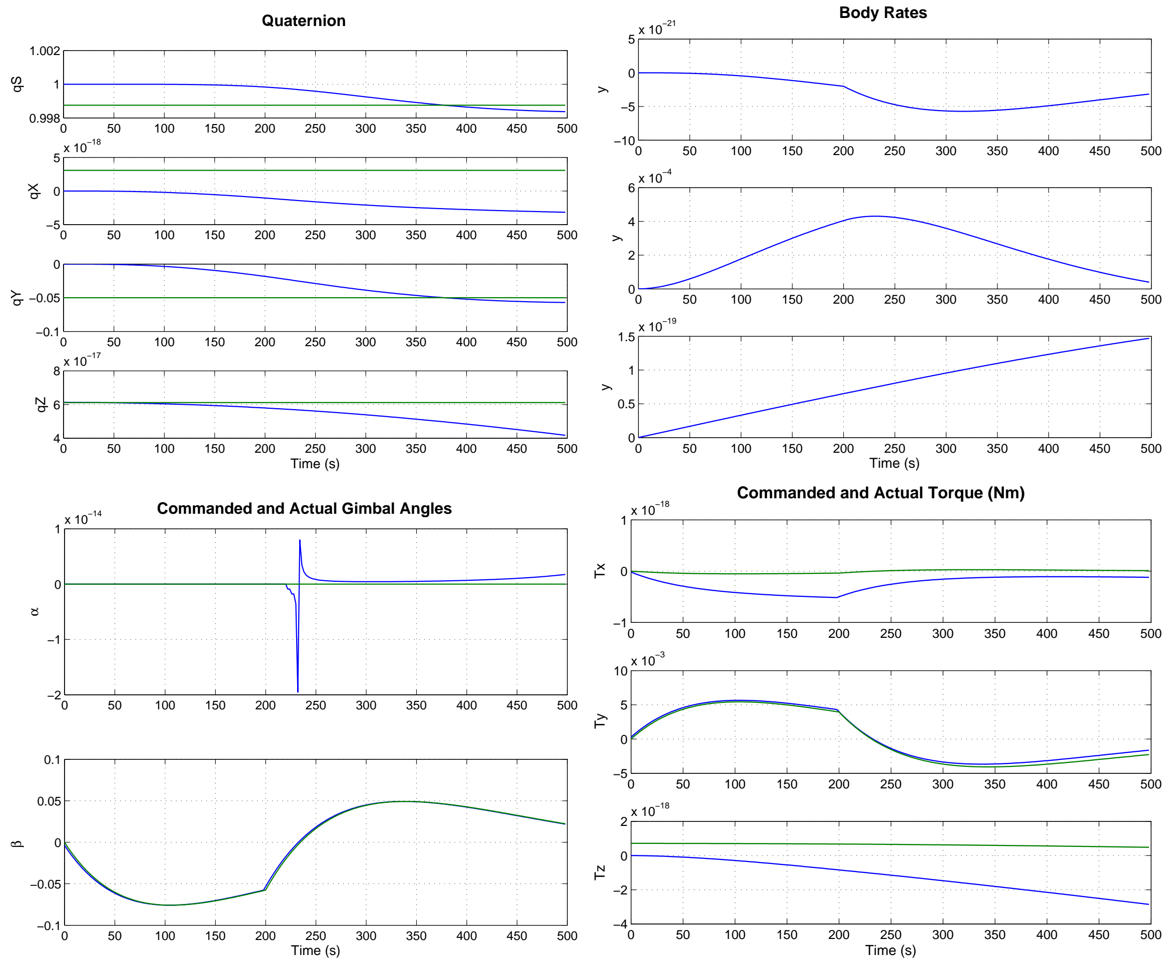
8.5 Heliopause Guidance Mission Demo

This example simulates a heliopause mission. The first step is to build a CAD model with a single flat plate. This is done in `FlatPlate`. Only one component and one body is in this model, which is the sail.

FlatPlate.m

The single body is created first and then the single sail. In this case we make the front and back properties identical and make the sail purely specular. The geometry is specified as a vertex array

Figure 8.8: Boom demo results for 0.1 rad cone angle maneuver



of two triangles. All vertex arrays must be for triangles, i.e. only 3 faces per polygon. You do not have to specify a back surface for a sail. This is done by `SailDisturbance` automatically.

The inertia for the sail is computed by modeling the entire vehicle as a flat plate. The particular mass and area are chosen to get an areal mass of 2 g/m^2 for this example.

The simulation is implemented in the `HeliopauseSimulation`. This script can use either the specular model built into `RHSHelio2DOrbit` or the full disturbance model. The full model is initialized using the code

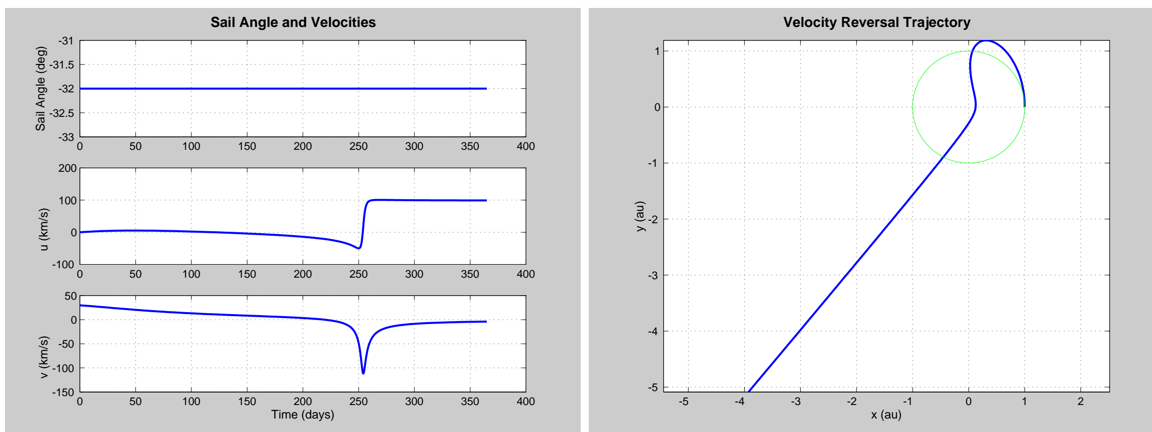
HeliopauseSimulation

You don't actually have to set the various disturbances, except solar, off when the center is the sun but we do so for completeness. The disturbance model uses the CAD model but the built-in specular model does not.

The sail angle is controlled using the function `HeliopauseSailAngle`. This fixes the angle at -32 deg from the radial direction. This leads to a tangential velocity reversal and a very high radial velocity as shown in Figure 8.9 on the next page. The entire simulation covers one Earth year.

This simple sail control brings the sail to within 0.123 astronomical units of the sun which is

Figure 8.9: Heliopause trajectory



probably too close for near term sail materials.

When the disturbance model option is used, it is necessary to convert the planar sail force angle with respect to the radial direction to an ECI to body quaternion. This is done with the following code snippet in `RHSHelio2DOrbit` *RHSHelio2DOrbit*

`x(4)` is the orbit angle. `AU2Q` converts the total angle, the sum of the orbit and sail angle `alpha`, and the rotation vector (along z) to the ECI to Body quaternion. The ECI force is converted into radial/tangential coordinates just after the call to `SailDisturbance`. The inputs to `SailDisturbance` are the CAD model, profile, and control structure.

8.6 Integrated Guidance and Attitude Control

Generally it is easiest to example attitude control maneuvers and orbit guidance in isolation. However, since the attitude and orbit are more tightly coupled in solar sails than in average vehicles, it can be desirable to combine the dynamics once the attitude control and guidance have each been validated. Here we will consider ways to integrate guidance with a boom control scheme. The general steps at a particular point in time are

1. Compute the environment based on the current state
2. Compute the disturbances based on the current state
3. Determine the guidance demand
4. Determine the attitude demand
5. Determine the actuator demand
6. Integrate the dynamics using the force and torque

Typical time scales for these steps might be a guidance update once or several times a day and an attitude controller with a frequency of 0.0001 rad/sec (about 6000 sec). The actuator demand will need to be met about five to ten times faster than the attitude controller update for stability.

The first step in combing guidance and attitude can be integrating an attitude controller of the desired frequency with a perfect actuation model into a guidance simulation. In this case, the order of the steps would change so that the disturbances are computed after the actuator demand, and the dynamics of the actuators are excluded. This will be the best possible match to the guidance profile that the attitude controller can do, before the actuation dynamics are added, and provides information on the expected actuator profile.

The simulation can be done discretely, using a fixed time step loop at a time increment small enough for the attitude controller. Depending on how many days (or years) of the guidance trajectory you want to simulate, it may be preferable to place the attitude controller inside the integration in a continuous format. Using PSS' `PIDMIMO` control function as an example, the two options are:

```
% Discretize the controller at tSamp
[a, b, c, d] = PIDMIMO( inr, zeta, omega, tauInt, omegaR, tSamp )

for k = 1:n
  % Compute control
  xN = a*xN + b*angleError;
  u = c*xN + d*angleError;

  % Integrate for dT n times
  x = RK4( @RHS, x, h, t, u );
end
```

and

```
% Continuous controller
[a, b, c, d] = PIDMIMO( inr, zeta, omega, tauInt, omegaR )

% Integrate for entire time duration
[t,x] = ode113( @RHS, x, h, [0 tF], a, b, c, d );
```

where in the second case u is computed as part of the `RHS` function. The first case has the advantage that all intermediate variables are available for storage and plotting. In the second case, `ode113` computes a time vector and a state vector which must then be post-processed to obtain all data of interest, such as the controller outputs and disturbances. The second option may or may not be faster depending on the details of the orbit and attitude dynamics; for example, if the sail passes very close the sun, then the orbit dynamics may need a very small time step during only that portion of the orbit. In this case, `ode113` will likely be faster than `RK4` with a sufficiently small time step.

The second step is to add the actuator dynamics for a more realistic attitude profile. In the case of fixed-rate gimbals, this requires another, smaller time scale to be added to the discrete formulation. If 600 seconds was sufficient for the attitude controller loop (control frequency of about 0.0001 rad/sec), then a time step of 60 seconds or less may be needed for the gimbal model. In order to use a continuous model and place the actuator dynamics and control inside the right-hand side, a continuous two-body model is needed.

The module contains the demo `SMAGuidanceWithBoom` which combines locally optimal semi-major axis guidance (in a heliocentric orbit) with gimballed boom control using the `PlateWithBoom`

model. Fixed rate dynamics are modeled for a general hinge using `FMovingBody`. The Cartesian orbit equations and attitude states are integrated together using the `FBoomHinge` right-hand-side. This demo works well with an attitude control time step of 600 seconds and an inner timestep for the boom of 120 seconds. The guidance is set on an outermost loop of two hours, since calling it more frequently slows the simulation and the commanded cone angle changes very slowly.

The guidance is implemented in a function call with

In this particular case, we know that the clock angle is fixed and equal to π in the PSS sail coordinate system.

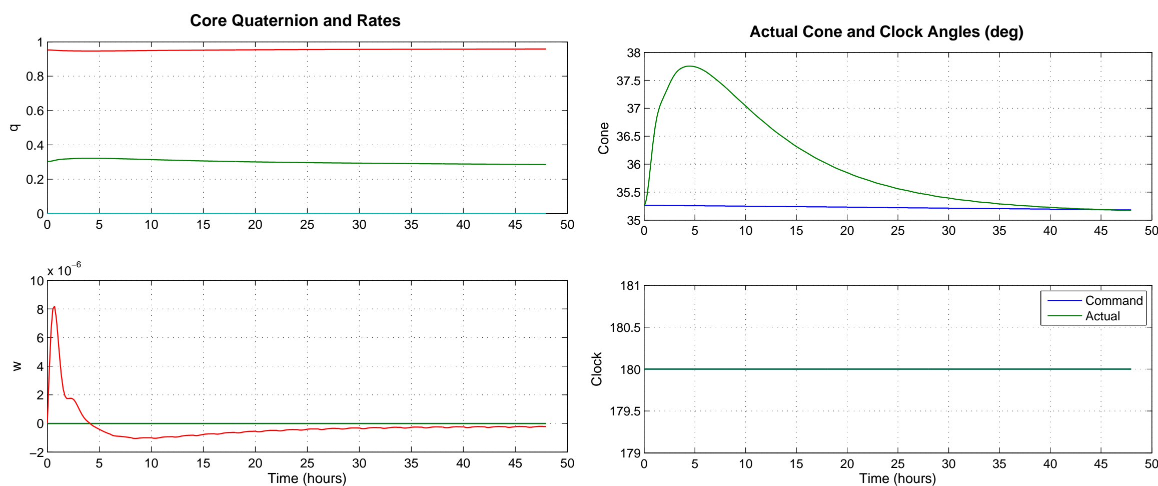
The commanded cone and clock angles must be translated to attitude angle errors. This can be done, assuming small angles, by finding the commanded normal in the current body frame. Since the normal is along the x body axis the Euler angle errors for the commanded normal can be found directly:

The actuation assumes a general hinge, so once the control torque is computed and the desired boom center-of-mass found, a corresponding axis and angle are used to limit the total boom rate.

`FMovingBody` requires that the core rates be updated whenever the other bodies are assigned new rates. This and the integration for one step are performed with the lines

The result is a simulation that can be run for two days in less than a minute (MATLAB 7 on Mac PowerBook).

Figure 8.10: Integrated guidance and attitude control for semi-major axis change



CHAPTER 9

TRAJECTORY OPTIMIZATION

This chapter discusses how to use the optimization tools.

9.1 Introduction

The Solar Sail Module provides locally optimal trajectory algorithms based on McInnes[?] and global optimization algorithms for performing trajectory optimization in 3 dimensions. The functions are in the Guidance and Optimization folders.

9.2 Locally Optimal Control Laws

Locally optimal heliocentric control laws from McInnes are implemented by the Guidance functions `LocallyOptimalTraj` and `SpecularAccelFromConeClock` as shown in the following code snippet. The first function computes the cone and clock angles. The second computes the acceleration unit vector. The unit vector needs to be multiplied by the magnitude of the acceleration to get the control acceleration.

```
[alpha, delta] = LocallyOptimalTraj( controlType, r, v, d.mu );
accel          = a*(x0/Mag(r))^2;
d.a            = accel*SpecularAccelFromConeClock( alpha, delta, r, v );
```

The demo `LocalOptimalSim` shows an implementation of these control laws. `controlType` is one of four possible locally optimal laws for changing the following orbital elements:

```
controlType   = 'ascending␣node';
controlType   = 'eccentricity';
controlType   = 'inclination';
controlType   = 'semi-major␣axis';
```

`r,v` are the position and velocity vectors. `d.mu` is the gravitational constant in units compatible with the position and velocity vectors. All four laws can be tested using this simulation. Results

for inclination control and semi-major axis control are shown in Figure 9.1 and Figure 9.2. The 3D plot is generated using `Plot3DOrbit`. This function plots the trajectory and the initial and final orbits based on the first and last pairs of `r`, `v` respectively.

Figure 9.1: Locally optimal sail control results for inclination change

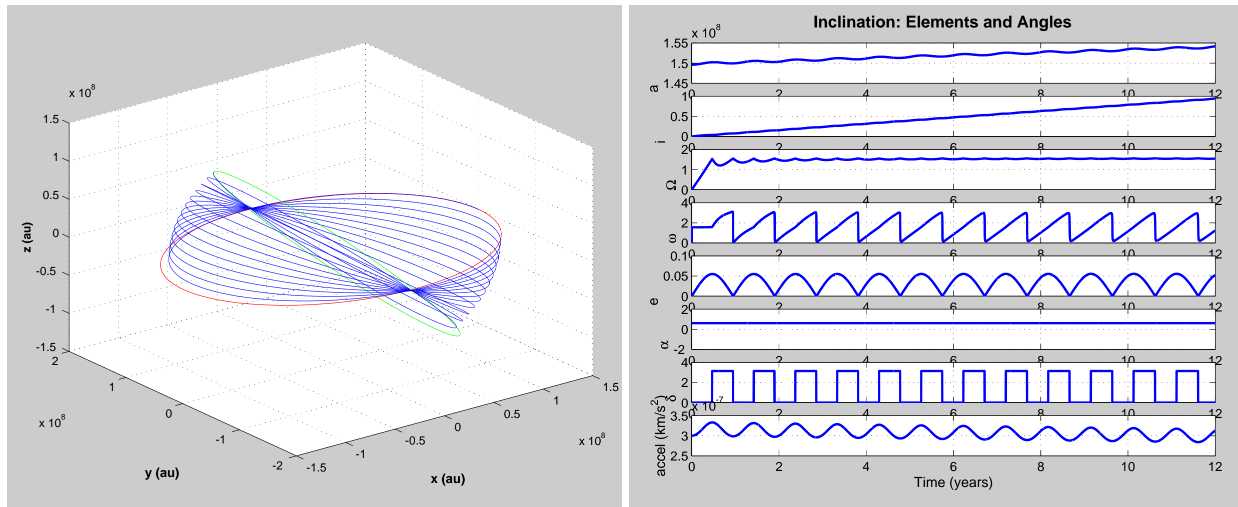
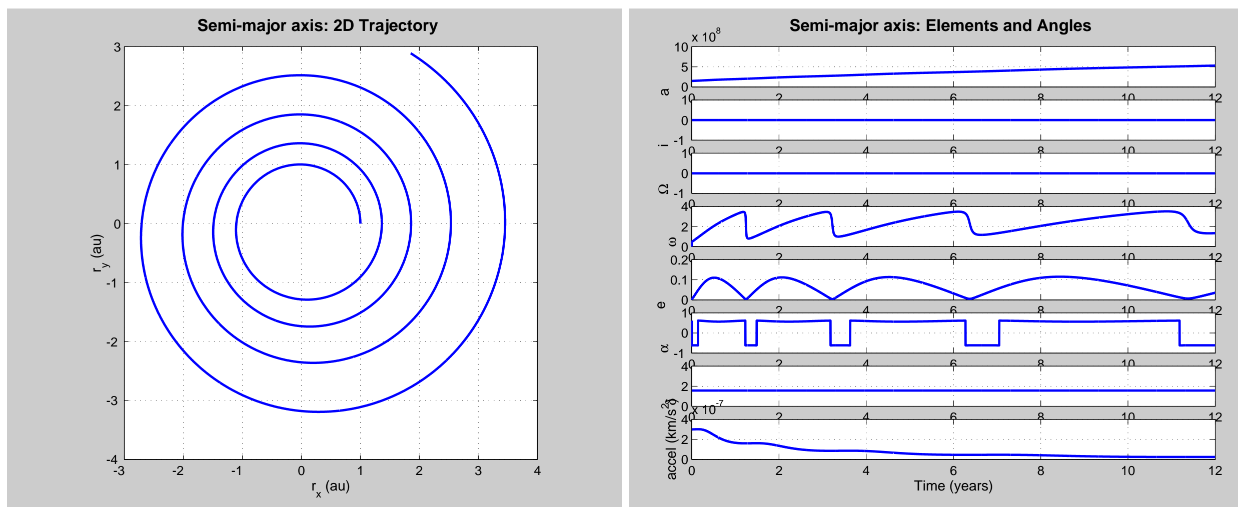


Figure 9.2: Locally optimal sail control of semi-major axis



The results for eccentricity and ascending node are shown in Figure 9.3 on the facing page and Figure 9.4 on the next page. These plots have been organized to exactly replicate those in the text.

Figure 9.3: Eccentricity simulation

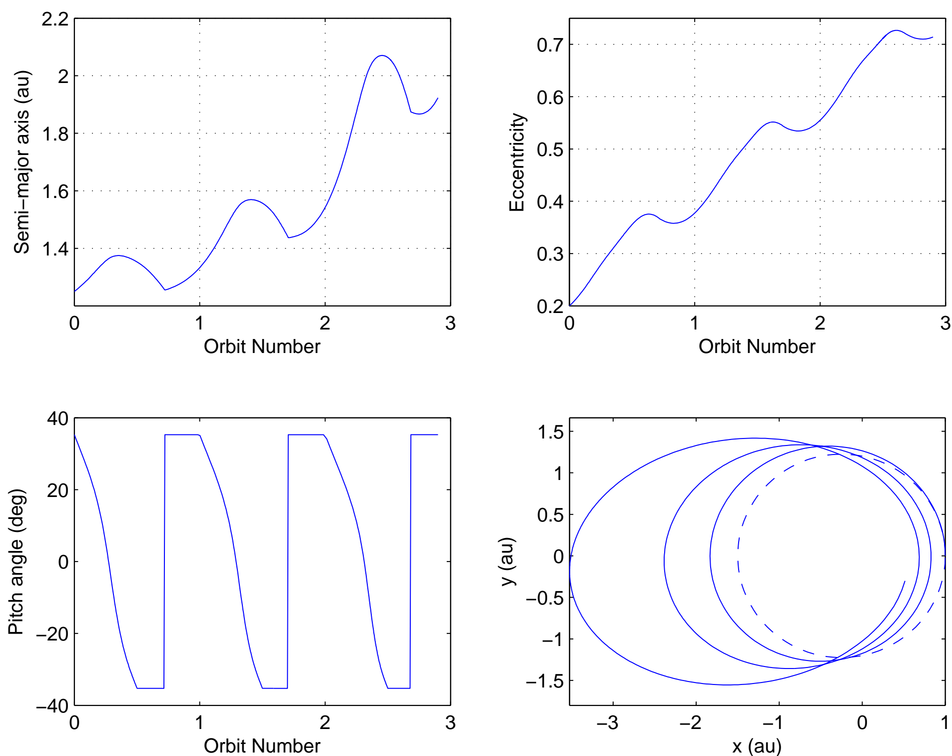
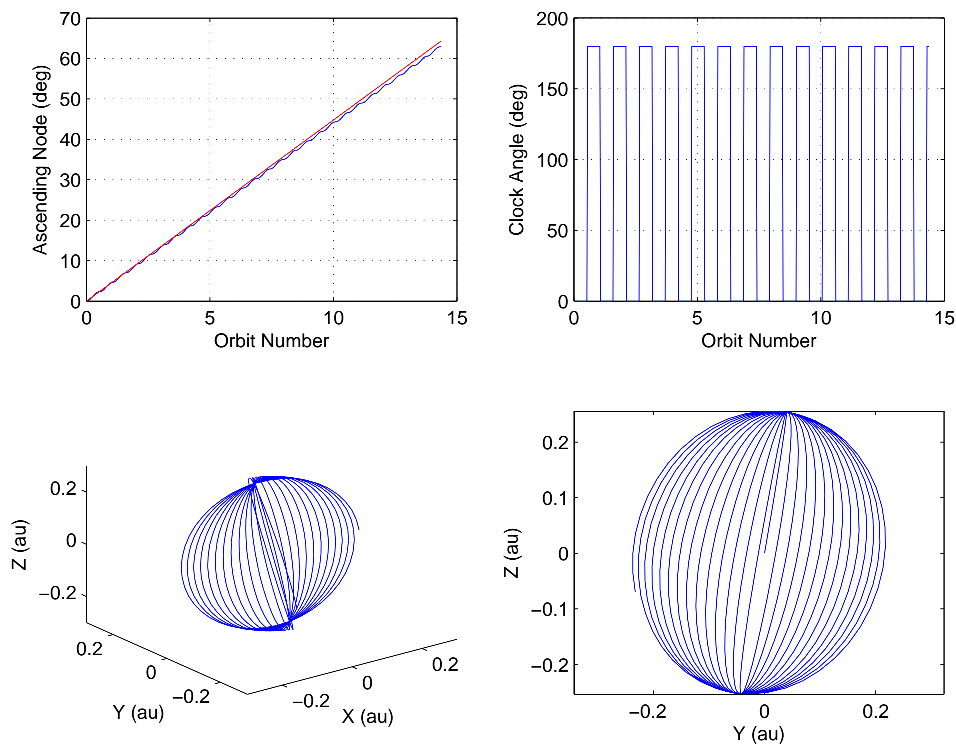


Figure 9.4: Ascending node simulation

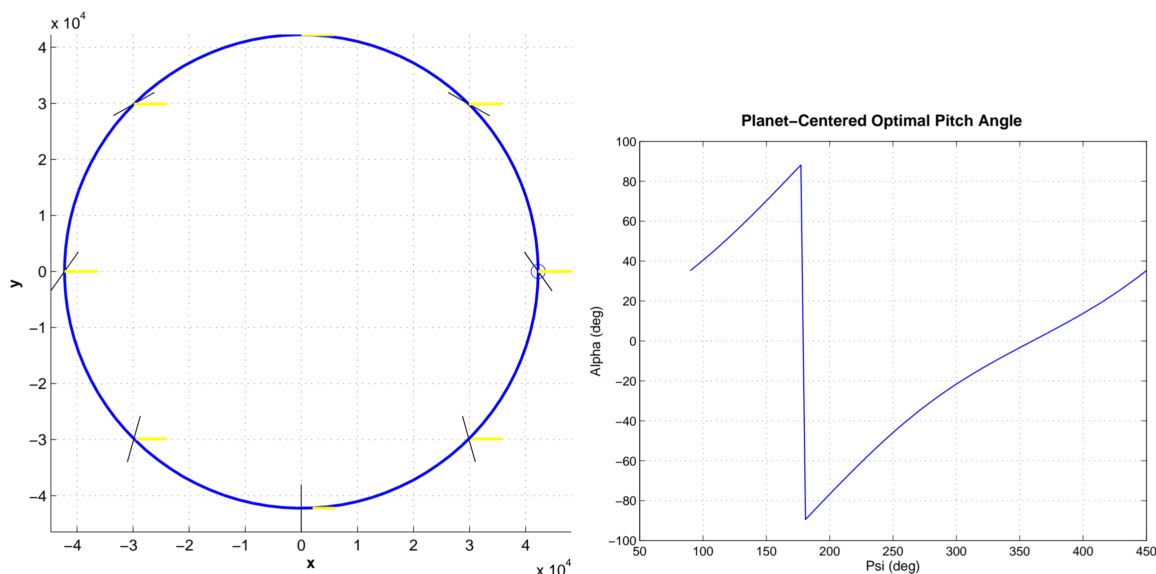


McInnes also gives a planet-centric law for maximizing the instantaneous rate of change of the orbital energy. This is implemented in `PlanetOptimalTraj` and demonstrated in `GeoOptimalSim`. The optimal pitch angle is given by

$$\alpha^* = \frac{1}{2}\left[\psi - \sin^{-1}\left(\frac{\sin\psi}{3}\right)\right] \tag{9.1}$$

where ψ is the angle between the spacecraft velocity vector and the unit vector to the sun. The sail orientation throughout one orbit is shown in Figure 9.5. The results of a simulation of a 30 day spiral from geostationary orbit are shown in Figure 9.6 on the next page.

Figure 9.5: Planet-centric locally optimal steering



9.3 Globally Optimal Control Laws

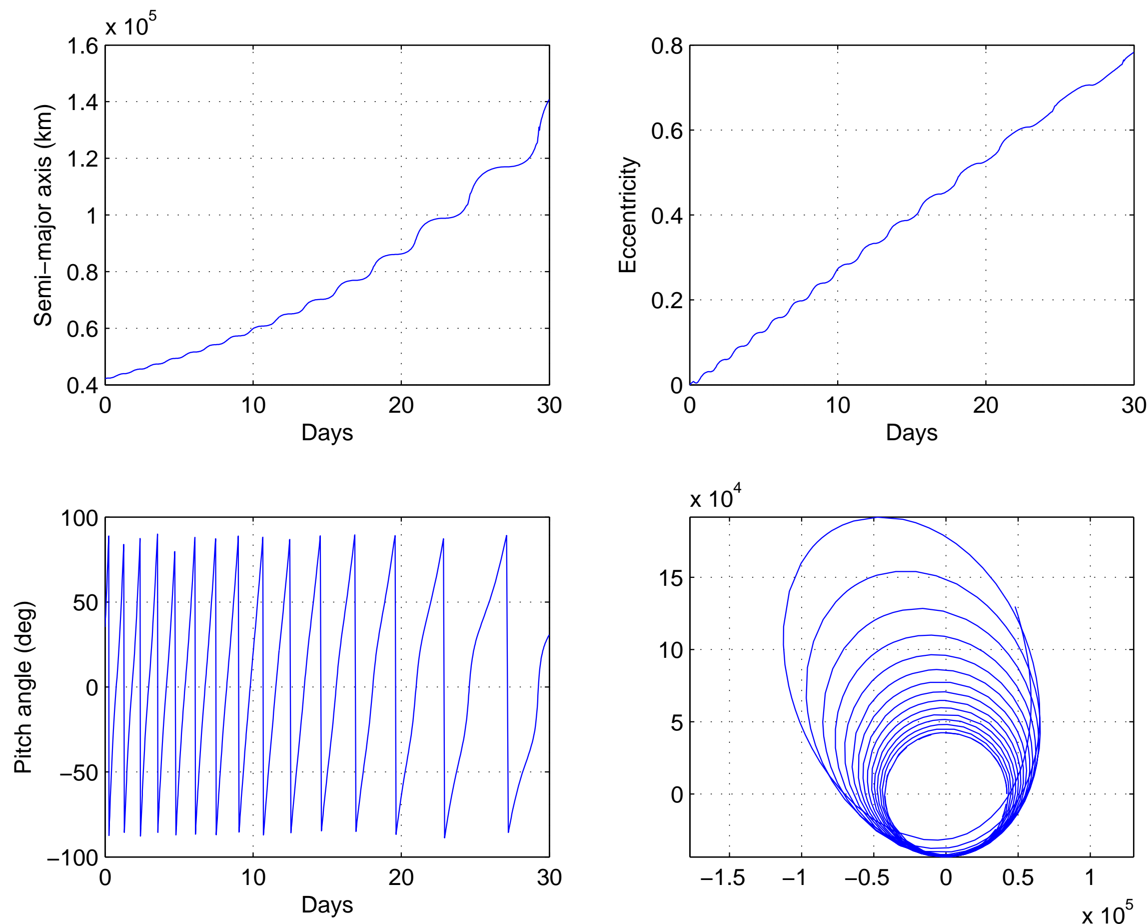
9.3.1 Global Methods

The `TrajectoryOptimization` function for global optimization supports three methods: downhill simplex, genetic algorithms, and simulated annealing, which are described in the next sections. The particular methods used in the Solar Sail Module are `fmins` and `fminsearch` (MathWorks) for downhill simplex, Genetic Algorithm Optimization Toolbox (GAOT) [**?**] for genetic algorithms and Goffe's algorithm[**?**] for simulated annealing. They are summarized in the following sections.

Downhill Simplex

The downhill simplex method is due to Nelder and Meade [**?**]. A simplex is defined as a figure of $N+1$ vertices in an N-dimensional space that do not lie in a hyperplane, i.e. have a finite volume.

Figure 9.6: 30 day spiral from geocentric orbit



This would be a tetrahedron in 3 dimensional space. Each simplex is a solution in the search space. The simplex can be expanded

$$x = x_0 + \lambda \tag{9.2}$$

contracted

$$x = x_0 - \lambda \tag{9.3}$$

or reflected

$$x = -x_0 \tag{9.4}$$

Downhill simplex first finds the points where the objective function is the highest moving the opposite face of the simplex to a lower point. These steps are called reflections. Steps are taken to maintain the volume of the simplex. The method tries to take larger steps whenever possible. When it reaches a ”valley floor” it contracts itself in a transverse direction and tries to move down the valley.

This method requires only function evaluations, derivatives of the functions are not required. This is particularly useful when derivatives are not available or are difficult to compute. The Solar Sail Module uses the MathWorks `fmins` and `fminsearch` functions for this method.

Genetic Algorithms

Genetic algorithms are a subset of evolutionary algorithms [?]. A genetic algorithm is an iterative procedure that consists of a constant size population of individuals, each one representing a solution in a given problem space. The algorithm is applied to spaces too large to be solved by exhaustive searching.

During each generation, individuals are chosen to reproduce. There are many methods for creating children but generally they involve crossovers, requiring two parents, and mutations, requiring a single parent. A selection function then determines which individuals survive to the next generation. The functions may have some dependence on the fitness of the individuals or may be completely random. Elitist implementations always keep the fittest individual (or lowest cost solution, in this case) during selection. The size of the population and number of generations are important parameters in the algorithm.

Genetic algorithms are stochastic iterative processes that are not guaranteed to converge. The termination criteria must be set at some fitness level or a fixed number of generations. The implementation used in Solar Sail Module is GAOT [?]; the supporting functions are included in the GAOT folder. `GAMin` is a wrapper for GAOT that is specific to function minimization. `GeneticAlgorithm` is another implementation based on an article by Sipper [1].

Simulated Annealing

Simulated annealing [?, ?] is a global optimization method based on the an analogy with thermodynamics. Solar Sail Module implements Goffe's algorithm in `SimulatedAnnealing`. At high temperatures the molecules of a liquid move freely with respect to each other. If the liquid is cooled slowly the thermal mobility is diminished and the atoms often can line themselves up in ordered lattices that are billions of atoms in 3 dimensions. This is a minimum energy state and if the cooling process is slow enough the crystal avoids falling into local minimums.

When minimizing a function, any downhill step is accepted and the process repeats from this new point. An uphill step may be accepted using the Metropolis criteria as the basis for decision. Thus simulated annealing process can escape from a local minimum. At each temperature the process is allowed to come to a new equilibrium.

The Metropolis criteria is an exponential function. It is computed for a given temperature. A random number is generated and compared against the value of the exponential.

As the optimization process proceeds, and temperature decreases, the length of the steps decline and the algorithm closes in on the global optimum.

9.3.2 Function Overview

The optimization functions and demos as listed in the Contents files are shown below.

```
>> help Optimization
```

[1] Sipper, M., "A Brief Introduction to Genetic Algorithms," http://www.cs.bgu.ac.il/ sipper/ga.html

```
Sail/Optimization

B
   BoundedMutation         - GA mutation function where change is bounded
                               within a fraction of the range.

C
   ControlAngles3D         - Computes the control angles for a 3D problem
                               given the costates of the
   Cost3D                  - Function to be used for optimization. It
                             computes an err magnitude
   CostLowThrust2D         - Cost function for 2D low thrust trajectory
                             optimization.
   CostLowThrust3D         - Cost function for 3D low thrust trajectory
                             optimization.
   CostSail2D              - Cost function for 2D solar sail trajectory
                             optimization.
   CostSail3D              - Cost function for 3D solar sail trajectory
                             optimization.

F
   FLambdaToConeClock      - Generates the optimal cone and clock angles
                             for 3D problems.
   FLowThrust2D            - This function is for the planar orbit
                             problem which includes
   FLowThrust3D            - This function is 3D orbit problem in
                             equinoctial coordinates.
   FSolarSailAngles        - RHS for a Newton solver for sail angles.
   FractionalSelection     - GA selection function where a fixed fraction
                               of the population survives.

G
   GAMin                   - Applies a genetic algorithm to minimizing a
                             function.
   GeneticAlgorithm        - Applies a genetic algorithm to minimizing
                             fun.

N
   NewtonSolver            - Solves for the zeros of a set of n equations
                             .

S
   SimulatedAnnealing      - Implements simulated annealing.
   SingleCrossover         - GA crossover function which switches a
                             randomly selected parameter between

T
   TrajectoryOptimization - Performs trajectory optimization.

Demos/Optimization

L
   LocallyOptimalTrajectories - Demonstrate locally optimal trajectories
                             . Uses equinoctial elements.
```

```
N
   NLEqSADemo                  - Test a nonlinear equation solver for
                               computing cone and clock.

O
   Optimization                - Demonstrate the trajectory optimization
                               function for low thrust.
   OptimizationTestGrid        - Do 10 tests on GAOT, simulated annealing
                                and downhill simplex.

P
   PlotDeJong                  - Plot the 5 De Jong functions.
   PropagationDemo             - Orbit propagation test.

S
   Sail2DOptimization          - Demonstrate the Trajectory optimization
                               function for sails in 2D orbits.
   Sail3DOptimization          - Demonstrate 3D optimization.

Z
   ZermeloCostDemo             - Compute the Zermelo cost function as a
                               function of lambda.
   ZermeloOptimization         - Demonstrate the Trajectory optimization
                               using the Zermelo problem
```

9.3.3 Formulation of the Problem

`TrajectoryOptimization` uses the indirect method of optimization [?]. The general problem for a single stage optimization problem is to minimize the cost function

$$J = \phi(x(t_f), t_f) + \int_{t_0}^{t_f} L(x, u, t)dt \tag{9.5}$$

subject to the dynamical equations

$$\dot{x} = f(x, u, t) \tag{9.6}$$

where $\phi(x(t_f), t_f)$ is a cost associated with the final state, x and time, t. $L(x, u, t)$ is the cost associated with the state, time and control, u as the trajectory progresses. For a minimum time problem, L is constant and equal to 1.

The dynamical equations are then appended to the cost functional by the Lagrange Multipliers, λ, which are functions of the states and time and are also known as the costates. The actual cost does not change since by definition the function multiplied by the costates is always zero. Additional constraints can be added to the cost functional as needed by including one additional costate per constraint. The cost function becomes

$$J = \phi(x(t_f), t_f) + \int_{t_0}^{t_f} L(x, u, t)dt + \lambda(t)^T (f(x, u, t) - \dot{x}) \tag{9.7}$$

The control Hamiltonian is defined as

$$H(x, \lambda, u, t) \quad = \quad L(x, u, t) + \lambda(t)^T f(x, u, t) \tag{9.8}$$

$$= \quad 1 + \lambda(t)^T f(x,u,t)$$

where we have substituted the minimum time value for L. The solution of the optimal control problem requires the solution of the following equations

$$\dot{x} = f(x,u,t) \tag{9.9}$$

$$\dot{\lambda}(t)^T = -\frac{\partial H}{\partial x} \tag{9.10}$$

$$\frac{\partial H}{\partial u} = 0 \tag{9.11}$$

If the final time is not constrained and the Hamiltonian is not an explicit function of time, then

$$H(t) = 0 \tag{9.12}$$

We seek to solve the following boundary value problem. Boundary conditions are set for the state equations in Eq. 9.6 on the facing page. Taking the partial of the Hamiltonian with respect to x in Eq. 9.10, the costate equations become

$$\dot{\lambda} = -\frac{\partial f}{\partial x}^T \lambda \tag{9.13}$$

where the boundary conditions of the partials with respect to the vector x are unknown. The optimality condition from Eq. 9.11 becomes

$$0 = \frac{\partial f}{\partial u}^T \lambda \tag{9.14}$$

where the subscript denotes the partial with respect to the control vector u which provides a relationship between the controls and the costates.

The boundary conditions may be on the states or the costates. In the problems discussed in this toolbox the initial conditions are always known. However, not all end conditions may be specified. For those that are not specified the boundary condition on the costate is set to zero. If S is the set of specified terminal boundary conditions then

$$x_k(t_f) = x_k, k \in S \tag{9.15}$$

$$\lambda_k(t_f) = 0, k \ni S \tag{9.16}$$

9.3.4 Zermelo's Problem

Insight into issues of global optimization can be obtained by examining Zermelo's problem. [**?**, pages 77-79] Zermelo's problem is a 2D trajectory problem of a vehicle at constant speed in a velocity field in which the velocity is a function of position, for example a ship with a given maximum speed moving through strong currents. The magnitude and direction of the currents in each axis (u,v) are functions of the position: $u(x,y)$ and $v(x,y)$. The goal is to steer the ship to find a minimum time trajectory between two points. An analytical solution is possible for the case

where only the current in the x-direction is non-zero and it is a linear function of the y position of the vehicle. The equations of motion f are

$$\begin{aligned} \dot{x} &= V\cos\theta + u(x,y) = V\cos\theta - V\frac{y}{h} \\ \dot{y} &= V\sin\theta + v(x,y) = V\sin\theta \end{aligned} \tag{9.17}$$

V is the velocity of the vehicle relative to the current which is constant and θ is the angle of the vehicle relative to the x-axis and is the control in the problem. The problem has a characteristic dimension of h. The Hamiltonian of the system is

$$H = \lambda_x(V\cos\theta + u) + \lambda_y(V\sin\theta + v) + 1 \tag{9.18}$$

Applying the costate equations (Eq. 9.13 on the previous page), we first compute the partials of the state equations,

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 0 & -V/h \\ 0 & 0 \end{bmatrix} \tag{9.19}$$

so applying the transpose to the partials matrix and substituting in we then have the costate derivatives.

$$\dot{\lambda} = -\begin{bmatrix} 0 & 0 \\ -V/h & 0 \end{bmatrix}\begin{bmatrix} \lambda_x \\ \lambda_y \end{bmatrix} = \begin{bmatrix} 0 \\ \lambda_x \frac{V}{h} \end{bmatrix} \tag{9.20}$$

We then compute the partials of the state equations with respect to the control,

$$\frac{\partial f}{\partial u} = \begin{bmatrix} -V\sin\theta \\ V\cos\theta \end{bmatrix} \tag{9.21}$$

so that the control angle θ can then be computed from the optimality condition in Eq. 9.14 on the preceding page.

$$\begin{bmatrix} -V\sin\theta & V\cos\theta \end{bmatrix}\begin{bmatrix} \lambda_x \\ \lambda_y \end{bmatrix} = 0 \tag{9.22}$$

$$\tan\theta = \frac{\lambda_y}{\lambda_x}$$

The above equations are used for the indirect optimization method. We can also compute the analytical solution for this problem when the final position is the origin (0,0). The optimal control angle as a function of the current position is expressed as

$$\frac{y}{h} = \sec\theta - \sec\theta_f \tag{9.23}$$

$$\frac{x}{h} = -\frac{1}{2}\left[\sec\theta_f(\tan\theta_f - \tan\theta) - \tan\theta_f(\sec\theta_f - \sec\theta) + \log\frac{\tan\theta_f + \sec\theta_f}{\tan\theta + \sec\theta}\right] \tag{9.24}$$

where θ_f is the final control angle and $\log$ is $\log_e$. These equations enable us to solve for the initial and final control angles θ_0 and θ_f given an initial position. Note that this corrects a sign error in the text.

These equations are implemented in the functions `RHSZermelo`, which computes the state and costate derivatives, and `ZermeloCost`, which integrates the equations using `ode113` and finds the minimum distance to the desired end point. `ZermeloCostDemo` demonstrates the cost function for an example set of initial and final conditions and a range of costate values. The cost function for solving Eq. 9.23 on the preceding page for the initial control angles and costates is `ZermeloAnalyticalCost`. `ZermeloOptimization` shows how to set up the `TrajectoryOptimizat` framework for solving the problem using downhill simplex, and compares the results to the analytical solution.

For each method being tested the optimization parameters have been chosen, by a certain amount of trial and error, to get the best results. The final λ vector is different in each case. However, the control is determined by the ratio so the magnitudes are not important. Table 9.1 gives the analytical and numerical solutions for the problem. The initial conditions are $[3.66; -1.86]$ and the final conditions are $[0; 0]$.

Table 9.1: Solutions

Test	λ_x	λ_y	Ratio
Analytical	-0.5	1.866025	-0.26795
Simplex	-0.65946	2.9404	-0.22428
Simulated Annealing	-0.68652	2.4593	-0.27915
Genetic Algorithm	-0.78899	2.9404	-0.26833

9.3.5 The Three Dimensional Equations of Motion

Now we will write the dynamical equations needed for optimization of a three-dimensional trajectory problem. We will examine both low thrust and solar sail examples. We can compare the low thrust results to standard results in the literature.

Cartesian

The cartesian equations are

$$\ddot{x} + \mu\frac{x}{r^3} = a \tag{9.25}$$

where

$$r = \sqrt{x^2 + y^2 + z^2} \tag{9.26}$$

Equinoctial

The three dimensional equations of motion in modified equinoctial coordinates are

$$\begin{aligned} p &= a(1-e^2) \\ f &= e\cos(\omega+\Omega) \\ g &= e\sin(\omega+\Omega) \end{aligned} \tag{9.27}$$

$$\begin{aligned} h &= \tan(i/2)\cos\Omega \\ k &= \tan(i/2)\sin\Omega \\ L &= \Omega+\omega+\nu \end{aligned}$$

where p is the semiparameter, a is the semimajor axis, e is the orbital eccentricity, ω is the argument of perigee, Ω is the right ascension of the ascending node, L is the true longitude and ν is the true anomaly.

The dimensional dynamical equations are

$$\dot{x} = Gu + b \tag{9.28}$$

where the state vector is

$$x = \begin{bmatrix} p \\ f \\ g \\ h \\ k \\ L \\ m \end{bmatrix} \tag{9.29}$$

where m is the mass

$$G = \begin{bmatrix} 0 & 2r\omega & 0 \\ \omega s & \omega(\gamma c+f)/q & -\Omega g \\ -\omega c & \omega(\gamma s+g)/q & -\Omega f \\ 0 & 0 & \zeta c \\ 0 & 0 & \zeta s \\ 0 & 0 & \Omega \\ 0 & 0 & 0 \end{bmatrix} \tag{9.30}$$

and

$$b = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \sqrt{\frac{\mu p}{r^2}} \\ -\frac{T}{u_e} \end{bmatrix} \tag{9.31}$$

where

$$r = \frac{p}{q} \tag{9.32}$$

$$c = \cos L \tag{9.33}$$

$$s = \sin L \tag{9.34}$$

$$z = hs - kc \tag{9.35}$$

$$\omega = \sqrt{\frac{p}{\mu}} \tag{9.36}$$
$$q = 1 + fc + gs \tag{9.37}$$
$$\gamma = q + 1 \tag{9.38}$$
$$\Omega = \omega\frac{z}{q} \tag{9.39}$$
$$\zeta = \frac{1}{2}\frac{\omega}{q}(1 + \sqrt{h^2 + k^2}) \tag{9.40}$$

a is the acceleration vector with components in the radial, tangential and normal directions. If the accelerations are zero, only L changes. The last equation is for the mass flow of the thruster.

If we define the control acceleration vector as

$$u = a\begin{bmatrix} \cos\alpha\cos\beta \\ \sin\alpha\cos\beta \\ \sin\beta \end{bmatrix} \tag{9.41}$$

then

$$U = \begin{bmatrix} -\sin\alpha\cos\beta & -\cos\alpha\sin\beta \\ \cos\alpha\cos\beta & -\sin\alpha\sin\beta \\ 0 & \cos\beta \end{bmatrix} \tag{9.42}$$

Then the optimality condition becomes

$$0 = U^T G^T \lambda \tag{9.43}$$

which is two equations in two unknowns, α and β.

9.3.6 Low-thrust Mars Rendezvous

The Mars rendezvous mission starts at a fixed time and attempts to rendezvous with Mars in the shortest possible time. The Mars trajectory is propagated so the final boundary conditions will depend on the time of intercept. Rendezvous problems can be formulated in cartesian coordinates since all six states must match the target states at the end. This is also the built-in demo of TrajectoryOptimization.

The demo setup is below. The demo is 2D, since Earth and Mars are very nearly in the same plane. The functions used are `FLowThrust2D`, `CostLowThrust2D`, and `Plot2DTrajectory`. Low-thrust is a simpler problem than sail optimization, since the thrust can be in any direction, not just opposed to the sun line, so it is a good test of the optimization functions.

```
% Demo
%-----
if( nargin < 1 )
  % Low-thrust Mars trajectory
  disp('Low-thrust Mars demo using Simplex')
  d.method           = 'downhill simplex';
```

```
  d.repeat           = 1;
  lbFToKg            = Constant('lb␣force␣to␣kg');
  aU                 = Constant('au');
  secInDay           = 86400;
  d.d.m0             = 10000*lbFToKg; % kg
  [name,a]           = Planets('rad',4); % Mars
  d.d.mu             = Constant('mu␣sun');
  d.d.rF             = a(1)*aU;
  d.d.mDot           = 6.7727e-5;
  d.d.thrust         = 0.002;
  d.d.rhsFun         = 'FLowThrust2D';
  d.d.plotFun        = 'Plot2DTrajectory';
  d.d.stateTol       = 1e-4;
  d.d.funTol         = 1e-4;
  d.d.nIts           = 600;
  d.d.costFun        = 'CostLowThrust2D';
  d.d.lambda0        = [0;0.1;1];
  d.d.x0             = [149597870;0;29.78469;4535.9237];
  d.d.xF             = [227936636.17;0;24.1295];
  d.d.data.absTol    = 1e-10;
  d.d.data.relTol    = 2.5e-8;
  d.d.data.maxStep   = 0.1;
  d.d.tEnd           = 300*secInDay;
  d.d.errorScale     = [1; 3e6; 1e7; 1e-2];

  [lambda,xF,tF] = TrajectoryOptimization( d );
  disp('Initial␣costate␣guess:')
  disp(d.d.lambda0)
  disp('Costate␣returned␣by␣Simplex:')
  disp(lambda{1})
  clear lambda
  return;
end
```

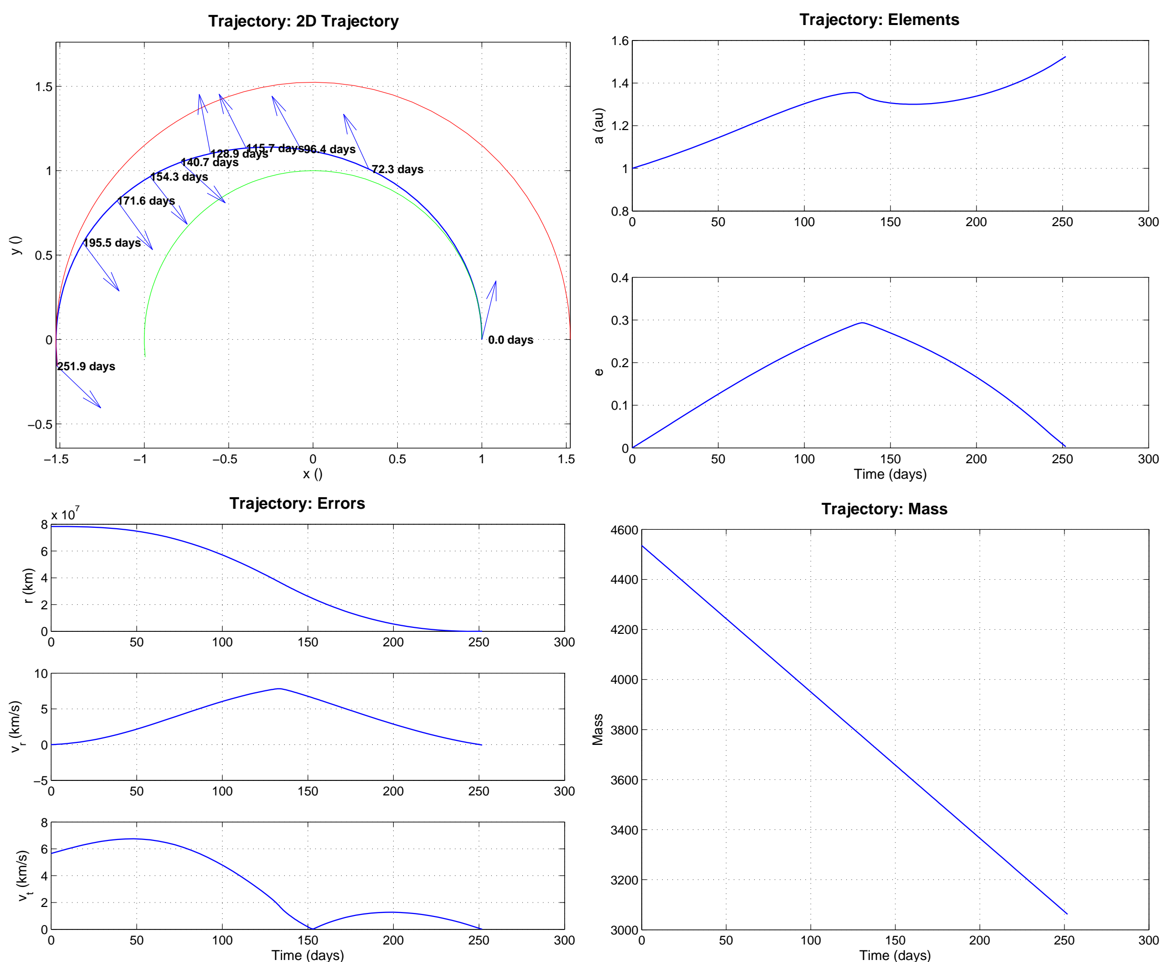
This produces the following output.

```
Low-thrust Mars demo using Simplex
Case 1: Method: downhill simplex

Exiting: Maximum number of function evaluations has been exceeded
         - increase MaxFunEvals option.
         Current function value: 212174.582193

Radial     position error =   -72772.1819 (km)
Radial     velocity error =       -0.0655 (km/s)
Tangential velocity error =       -0.0016 (km/s)
    Time of minimum error =      251.8840 days
Initial costate guess:
            0
```

```
          0.1
            1
Costate returned by Simplex:
   1.0999e-07
      0.16811
      0.70176
```

The resulting trajectory is in Figure 9.7.

Figure 9.7: Low-thrust Mars rendezvous



9.3.7 Sail 2D Optimization Examples

`Sail2DOptimization` contains two different examples, an Earth-Mars transfer and the planar portion of a Solar Polar Imager trajectory (reduction in semi-major axis from 1 AU to 0.48 AU). For each example, the demo can run either simplex or a genetic algorithm. The desired endpoint is a circular orbit at the new radius; the demo does not assure a rendezvous with Mars. The supporting functions are `TwoDOptimalSailAngle` `RHSOpt2DOrbit`, `Plot2DTrajectory`, and

`CostSail2D`. Simplex runs quite quickly for both cases. `TwoDOptimalSailAngle` gives the sail angle found by solving the combined state and costate equations for a purely specular sail in 2D radial coordinates.

The SPI results using simplex are shown below and in Figure 9.8. The transfer time to 0.48 AU is 1.74 years.

```
Case 1: Method: downhill simplex
Radial     position error =   33009.4317 (km)
Radial     velocity error =       0.0239 (km/s)
Tangential velocity error =      -0.0175 (km/s)
    Time of minimum error =       1.7414 years
Costates for Simplex
       0.1025
        28760
   5.5646e+05
```

The Mars results using simplex are shown below and in Figure 9.8. The transfer time is 2.4 years. It is much longer than the SPI transfer because the force produced by the sail is diminishing as the sail gets further from the sun.

```
Case 1: Method: downhill simplex
Radial     position error =     696.5657 (km)
Radial     velocity error =       0.0089 (km/s)
Tangential velocity error =      -0.0161 (km/s)
    Time of minimum error =       2.4188 years
Costates for Simplex
   -0.0038444
      -9640.8
       -17466
```

9.3.8 Heliopause Mission

The IHP trajectory is defined by a spacecraft delivery to the nose of the heliosphere in less than 25 years (located at 200 AU with latitude/elevation 7.5 deg, longitude/azimuth 254 deg in the ecliptic coordinate frame). The sail is jetisoned at 5 AU and there is a 0.25 AU minimal radius constraint for the solar sail assist[**?**].

Figure 9.8: 2D SPI solved using simplex

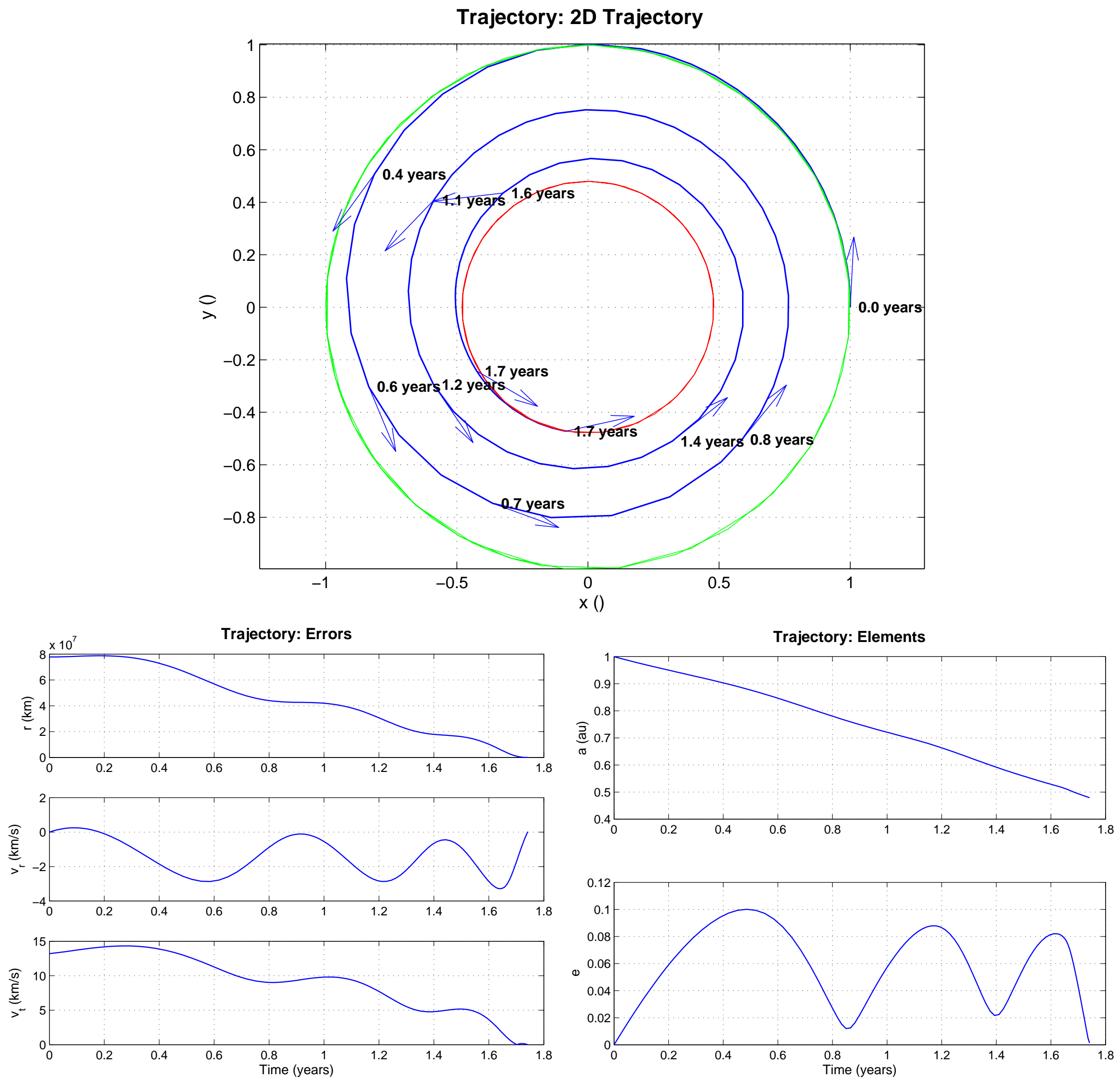


Figure 9.9: 2D Mars solved using simplex