Booklet

2

SPACECRAFT DYNAMICS SIMULATION

Russell Hawkins, Oxon Hill High School

Dynamics Simulation Coding Guide

RUSSELL HAWKINS OF OXON HILL HIGH SCHOOL

Dynamics Simulation Coding Guide

I ask that you please respect my work and keep this booklet (as well as any of my other materials) clean and return them to me or wherever my poster is located when you are finished reading.
Thank You for being interested!

Russell Hawkins
8241 Surratts Rd
Clinton, MD 20735
Phone: 301.433.3297
Email: Russell.aspirations@gmial.com

Table of Contents

Chapter 1

How This Booklet Is Organized

This chapter is meant to explain what is in this booklet and how this information is laid out through the pages.

This booklet contains all of the completed, bug-free, code written for this project. This is a collection of functions and scripts that all work together to run my final simulation. This booklet is split up by **Section Headers**, which will be written in the same style as this chapter's heading; these headers show which part of the project the following functions[1] or scripts[2] pertain to. Following the headers, my code will be written *exactly* as it was written in MatLab®, meaning it will follow the .m file coding format. I will do my best in explaining as much of the code that I can in its simplest terms.

1. A function simply takes in a set of inputs, and then uses a set of parameters and mathematical equations to create a set of outputs.
2. A script runs multiple functions in the same block of code, allowing for more complex simulations.

How to Understand Functions

This is an oversimplification of a basic function in hopes to help those who are not programmers or are not familiar with MatLab® to understand what it is that I am doing in the following pages.

All functions start off with the word function, and then go on to state the [output], which equals (=) the call_to_the_function (which cannot have spaces), immediately followed by the (input), which all in all looks like this:

Function [Product] = Multiplication(number1, number2)

Everything that follows basically tells the program how to get from the inputs to the outputs. Example: Product = number1*number2

And finally the function finishes with the word end.

Chapter 2

01_Quantize Sun Sensor Measurements:

```
%{
Section 1: Quantize Sun Angles

            For a spinning spacecraft, a slit sun sensor is commonly used to derive a
            sun angle (referenced from the spacecraft spin axis) and time the sun
            crossed the slit.  The sun data transmitted to the ground is the count
            representation of the angles. No machine is capable of infinite precision,
            so each count is rounded to a usable fraction of a radian (or degree). This
            process is known as Quantizing.
%}
```

```
function [Quantized_Sun_Angles] = Quantize_Sun_Angles (sun_angles, sun_angles_times)

       bias         = (0.625) * pi/180 ;

       Scale_factor = (0.125) * pi/180 ;

%      converting from degrees to radians

       counts               = fix((sun_angles - bias)/(Scale_factor)) ;

       unquantized_sun_data = ((sun_angles - bias)/(Scale_factor))    ;

       Quantized_Sun_Angles = (counts * Scale_factor) + bias          ;


       plot(sun_angles_times, Quantized_Sun_Angles, 'k', sun_angles_times, unquantized_sun_data, 'r')

end
```

Chapter 3

02_Time:

```
%{
        Section 2: Time

        Part 1: Calculate Julian date from input calendar format time

                This part of this section will convert calendar date and store it as Julian
                date + fractions of a day. Julian Date is the number of days since Noon
                (12:00 UT) on January 1, 4713 BC. Julian Date is found because it is the
                timeframe in which most spacecraft are monitored.

                Calendar Format = YYYYMMDD.HHMMSSmmm

                        Where YYYY=year
                              MM=month
                              DD=day
                              HH=hours
                              MM=minutes
                              SS=seconds
                              mmm=milli-seconds

                        For example: 19700101 would be 12:00 AM, January 1st, 1970

 %}



function [juliandate] = cal_to_jd(Calendar_Date)

        ymd  = floor(Calendar_Date)                                ;
               year  = floor(ymd/10000)                            ;
               month = floor((ymd-year*10000)/100)                 ;
               day   = floor(ymd-year*10000-month*100)             ;

        frac = Calendar_Date - ymd                                 ;
               hour   = floor(frac*100)                            ;
               minute = floor(frac*10000-hour*100)                 ;
               second = round(frac*1000000-minute*100-hour*10000)  ;

        fday = (hour*3600+minute*60+second)/86400                  ;


        L          = fix((month-14)/12)   ;
        juliandate = day - 32075 + fix(1461*(year + 4800 + L)/4)+ fix(367*(month - 2 -
                     L*12)/12)-fix(3*fix(((year + 4900 + L)/100)/4))   ;
        juliandate = juliandate + fday    ;
end
```

```
%{
       Section 2: Time

       Part 2: Calculate Calendar time from Julian Date

              Julian Date can be converted back to calendar format as follows:
%}
```

```
function [Calendar_Date, Y, M, D, hour, minute, second, fsec] = jd_to_cal(JD)
       p = fix(JD) + 68569             ;
       q = fix(4*p/146097)             ;
       r = p - fix((146097*q + 3)/4)   ;
       s = fix(4000*(r+1)/1461001)     ;
       t = r - fix(1461*s/4) + 31      ;
       u = fix(80*t/2447)              ;
       v = fix(u/11)                   ;
       Y = 100*(q-49)+s+v              ;
       M = u + 2 - 12*v                ;
       D = t - fix(2447*u/80)          ;

%   Where Y=year, M=month, and D=day and
%   HHMMSSmmm are calculated from fraction of julian day below

       julian_date = (Y * 10000) + (M * 100) + D;

       lowJD = floor (JD);

       decimal = JD - lowJD;

       hour = floor (decimal * 24);

       minute = floor (decimal * 1440 - hour * 60);

       second =(decimal * 86400 - (hour * 3600) - (minute * 60));

       fsec = second-floor(second);

       second = floor(second);


       partial_day = (hour / 100 ) + (minute / 10000) + (second / 1000000);

       Calendar_Date = julian_date + partial_day;

end
```

Chapter 4

03_Reference Sun Vector Calculation:

```
%{
        Section 3: Reference Sun Vector Calculation

                The position of the Sun is computed just like the position of any other
                planet, but since the Sun always is moving in the ecliptic, and since the
                eccentricity of the orbit is quite small, a few simplifications can be made.
                Therefore, a separate presentation for the Sun is given.

                Of course, this function is really computing the position of the Earth in
                its orbit around the Sun, but since we're viewing the sky from an Earth-
                centered perspective, we'll pretend that the Sun is in orbit around the
                Earth instead.

%}

function [xe,ye,ze] = sungci(firstdate);
        epic  = 2451544      ;  %epic is 19991231.0 changed to julian date
        jd_in = firstdate    ;
        d     = jd_in - epic ;  %where jd_in is the julian date for your input time.


        N = 0.0                                          ;
        i = 0.0                                          ;
        w = (282.9404 + 4.70935E-5 * d) *(pi/180)    ;  %degrees to radians
        a = 1.000000                                     ;  %(AU)
        e = 0.016709 - 1.151E-9 * d                      ;
        M = (356.0470 + 0.9856002585 * d) * (pi/180) ;  %degrees to radians

        rs = 149.6 * (10^6);
        %rs is the distance to the sun from earth

        %eccentric anomaly E from the mean anomaly M and from the eccentricity e (E and M
in radians)
        E = M + e * sin(M) * ( 1.0 + e * cos(M) ) ;

        %Then compute the Sun's distance r and its true anomaly v from:
        xv = cos(E) - e;
     yv = sqrt(1.0 - e*e) * sin(E);
        v = atan2( real(yv), xv ) ;
     r = sqrt( xv*xv + yv*yv ) ;

        %Sun's true longitude:
        lonsun = v + w ;

        %Convert lonsun,r to ecliptic rectangular geocentric coordinates xs,ys:
```

```
        xs = r * cos(lonsun);
    ys = r * sin(lonsun) ;
        %zs always equals 0

        ecl = (23.4393 - 3.563 *(10^(-7)) * d) * (pi/180); %degrees to radians
        xe = xs * rs;
    ye = ys * cos(ecl) * rs;
    ze = ys * sin(ecl) * rs;

        RA  = atan2( ye, xe ); %Sun's Right Ascension (RA) and Declination (Dec)
    Dec = atan2( ze, sqrt(xe * xe + ye * ye) );


        sun_vector=[xe,ye,ze];

        end



%{
        Section 3: Reference Sun Vector Calculation

                Check: This function is checking to see if my 'sungci' function works
                correctly. I am loading in the true sun data in the first line.

%}

load true_sun.mat
t=(cal_to_jd(true_sun(1,1))-cal_to_jd(true_sun(1,1)):1:cal_to_jd(true_sun(366,1))-
cal_to_jd(true_sun(1,1)));

time=2453827:2453827+365;
for i=1:length(time);
[x(i) y(i) z(i)]=sungci(time(i));
end

subplot (3,1,1)
plot(t,true_sun(:,2),'rs')
hold on
plot (time-time(1),x)
hold off
title('Sun Position(GCIMN2000)')
ylabel('x(GCI)')
subplot (3,1,2)
plot(t,true_sun(:,3),'rs')
hold on
plot (time-time(1),y)
hold off
ylabel('y(GCI)')
subplot (3,1,3)
plot(t,true_sun(:,4),'rs')
hold on
plot (time-time(1),z)
hold off
ylabel('z(GCI)')
xlabel('Days since 20060401')
```

Chapter 5

04_Orbital Dynamics

```
function[rdot,vdot]=deriv(r,v)
        %if the earth is a point of mass, the force due to gravity is :
        rdot=v;
        Gconstant=398600.4418;
        rmag=sqrt((r(1,1)^2)+(r(2,1)^2)+(r(3,1)^2));
        vdot=(-Gconstant*r)/(rmag^3);
end
        %Fg=(G*Mg*mx*r)/(abs(r)^3)
        %G*Mg=398600.4418(km^3s^(-2))
        %mx=spacecraft_mass(kg)

        %From newton's second law:

        %F=mass*acceleration
        %mx*accelerationx=-Fg
        %ax=(-G*Mg*r)/(abs(r^3))

        %where vector ax =  the spacecraft acceleration

        %for the ordinary differential equation (ODE) state, use the following:
        %x(1:3,1)=position(km)
        %x(4:6,1)=velocity(km/sec)

        %for this^ use 4th order Runge Kutta algorithm
```

```
function [rout,vout] = myrk4(rin,vin,dt)
        [rdot1,vdot1]=deriv(rin,vin);
        v1=vin+vdot1*(dt/2);
        r1=rin+rdot1*(dt/2);
        [rdot2,vdot2]=deriv(r1,v1);
        v2=vin+vdot2*(dt/2);
        r2=rin+rdot2*(dt/2);
        [rdot3,vdot3]=deriv(r2,v2);
        v3=vin+vdot3*(dt);
        r3=rin+rdot3*(dt);
        [rdot4,vdot4]=deriv(r3,v3);

        rout=rin+((1/6)*(rdot1+2*rdot2+2*rdot3+rdot4)*dt);
        vout=vin+((1/6)*(vdot1+2*vdot2+2*vdot3+vdot4)*dt);

end
```

```
load project4.mat;

        total_time=3480;
        dt=60;
        n=total_time/dt;
        r=zeros(3,n);
        v=zeros(3,n);
        r(:,1)= [ 1.939171267526330e+03;
         7.053946077269909e+03;
         5.216461164024867e+03];
        v(:,1)= [ 7.728790449144201e-01;
        -5.188238121076679e+00;
         3.991379730081099e+00];
        for i= 2:n
                [r(:,i),v(:,i)]= myrk4(r(:,i-1),v(:,i-1),dt);
        end
```

```
figure('name','Propagated Position Vectors(CCIMN2000)');
subplot(3,1,1)
plot(r(1,:));
ylabel('x(km)');
title('Propagated Position Vectors(CCIMN2000)');
subplot(3,1,2)
plot(r(2,:));
ylabel('y(km)');
subplot(3,1,3)
plot(r(3,:));
xlabel('Time (minutes)');
ylabel('z(km)');

figure('name','Propagated Position Magnitude(CCIMN2000)');
plot(sqrt(r(1,:).^2+r(2,:).^2+r(3,:).^2));
ylabel('km')
xlabel('Time (minutes)');
title('Propagated Position Magnitude(CCIMN2000)');

figure('name','Propagated Velocity Vectors(CCIMN2000)');
subplot(3,1,1)
plot(v(1,:));
ylabel('x(km)');
title('Propagated Velocity Vectors(CCIMN2000)');
subplot(3,1,2)
plot(v(2,:));
ylabel('y(km)');
subplot(3,1,3)
plot(v(3,:));
xlabel('Time (minutes)');
ylabel('z(km)');


figure('name','Propagated Velocity Magnitude(CCIMN2000)');
plot(sqrt(v(1,:).^2+v(2,:).^2+v(3,:).^2));
ylabel('km')
xlabel('Time (minutes)');
title('Propagated Velocity Magnitude(CCIMN2000)');
```

Chapter 6

05_Reference Magnetic Field Model

In this project, I compute the reference magnetic field vectors for the time and position vectors in Project 04.

```
function [magnetic_field_vectors]=MagVector(r,t)


dipole_movement=7.943e13;
rmag=sqrt((r(1,:)^2)+(r(2,:)^2)+(r(3,:)^2));
theta=168.6*(pi/180);
epsilon=109.3*(pi/180);
tref=2450925.4186692*86400;
omega=0.00417807462229498;
rot=(t-tref)*omega;
ur=(r./rmag);
GHA=mod(rot,360)*(pi/180);
um=[sin(theta)*cos(GHA+epsilon);
   sin(theta)*sin(GHA+epsilon);
         cos(theta)];
%greenwich hour angle
%um is a unit vector
magnetic_field_vectors=(dipole_movement./rmag^3)*((3*dot(um,ur))*ur-um);
end
```

```
total_time=3420;
dt=60;
n=total_time/dt;
r(1:3,1)= [1.939171267526330e+03;
        7.053946077269909e+03;
        5.216461164024867e+03];
r(4:6,1)=[7.728790449144201e-01;
         -5.188238121076679e+00;
         3.991379730081099e+00];

t0cal=20060401.0851000732;
t0sec=cal_to_jd(t0cal)*86400;
t=t0sec;
MagGCI=MagVector(r(1:3,1),t/86400);
for i=2:n
          [r(1:3,i),r(4:6,i)]=myrk4(r(1:3,i-1),r(4:6,i-1),dt);
          t(i)=t(i-1)+dt;
          MagGCI(1:3,i)=MagVector(r(1:3,i),t(i)/86400);
end
MagGCI
          figure (1)
          subplot(3,1,1)
          plot((t-t0sec)/60,MagGCI(1,:))
          title ('Reference Magnetic Field Vectors (GCI) Mean of Date')
          ylabel('x(mG)')
          subplot(3,1,2)
          plot((t-t0sec)/60,MagGCI(2,:))
          ylabel('y(mG)')
          subplot(3,1,3)
          plot((t-t0sec)/60,MagGCI(3,:))
          ylabel('z(mG)')
          xlabel('Time in minutes from:20060401.0851000734')

          figure (2)
          plot((t-t0sec)/60, sqrt(MagGCI(1,:).^2+MagGCI(2,:).^2+MagGCI(3,:).^2))
          title ('Reference Magnetic Field Vectors (GCI) Mean of Date')
          ylabel('mG')
          xlabel('Time in minutes from: 20060401.0851000734')
```

Chapter 7

06_Attitude Rate Dynamics

Eulers equation of motion describes the spacecraft angular motion in the presence of an external torque, M.

$$\mathbf{I} \cdot \dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (\mathbf{I} \cdot \boldsymbol{\omega}) = \mathbf{M}$$

I = Moment of inertia (kg-m^2)
w = angular rate (rad/sec)
M = external torque (N-m)

Project: Propagate rate using the 4th order runge-kutta algorithm with the following initial parameters for 1200 seconds:

```
I = [ 6.400030239e-01   -1.147147624e-01  -1.679751878e-03;
     -1.147147624e-01    9.414219149e-01   3.59946831e-04;
     -1.679751878e-03    3.59946831e-04    1.1155425364e+00];% kg-m^2

w0=[ 1.682386001340433e-02;
      1.262317498918817e-02;
     -2.832162183099951e+00]';% rad/sec

M = [0;0;0];
```

The ODE state will be w (spacecraft rate)

Chapter 8

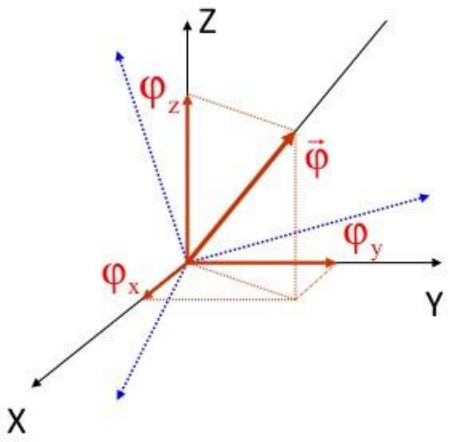
07_Attitude Quaternion Kinematics

Spacecraft orientation or attitude is defined as the parameters necessary to describe the spacecraft body axes relative to a reference coordinate system. Traditionally, the reference coordinate system is Earth centered where the reference z-axis is the Earth's spin axis and the x & y axes are in the Earth's equatorial plane. As we know, the Earth rotates. For this project (and most spacecraft), the Earth's axes are assumed frozen at January 1, 2000. All spaecraft attitudes are relative to those reference axes. Most of the time, the axes are not a problem as you are given reference data relative to this coordinate system which is called GeoCentric Inertial (GCI) Mean 2000.

As mentioned above, spacecraft attitude takes many forms. Tradionally with aircraft, attitude has been defined as roll, pitch, and yaw where roll is a rotation about the x-axis, pitch is a rotation about the y-axis, and yaw is a rotation about the z-axis. This form of attitude is easier to visualize. However, it involves trigametric functions, is dependent on the order of the rotation (roll/pitch/yaw,yaw/pitch/roll, etc) and has singularities for certain angles. Due to the these issues and the complexities of implementing trigametric functions in early computers, quaternions were selected to represent spacecraft attitude. Quaternion arithmetic is algebraic and more efficient for computation. A quaternion is defined as follows:

Quaternion

- What is quaternion-of-rotation?



$$q_1 = \sin(\frac{\varphi}{2}) \cdot \frac{\varphi_x}{\varphi}$$

$$q_2 = \sin(\frac{\varphi}{2}) \cdot \frac{\varphi_y}{\varphi}$$

$$q_3 = \sin(\frac{\varphi}{2}) \cdot \frac{\varphi_z}{\varphi}$$

$$q_4 = \cos(\frac{\varphi}{2})$$

Obviously : $\|\mathbf{q}\| = \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} = 1$

1

where $\vec{j}$ is an angular vector fixed to your reference coordinate system (i.e. GCI Mean 2000). If you rotate about that vector by angle j, the GCI axes will line up with the spacecraft body axes.

Attitude Motion is as follows:

$$\dot{\bar{q}} = \frac{1}{2} W \bar{q}$$

where

$$W = \begin{bmatrix} 0 & w_z & -w_y & w_x \\ -w_z & 0 & w_x & w_y \\ w_y & -w_x & 0 & w_z \\ -w_x & -w_y & -w_z & 0 \end{bmatrix}$$

$\bar{w}$ is the spacecraft body rate vector
$\bar{q}$ is the spacecraft quaternion (4x1 column vector)

Project: Augment (add to) the attitude rate dynamics equation (in Project 06) using the quaternion kinematics equation above and propagate using your 4th order Runge-Kutta.

Assume the following initial quaternion and utilize the rates calculated before.

```
q0 = [-6.236273220692253e-01;
      -7.496449331864782e-01;
      -1.594698559301466e-01;
       1.539181677587992e-01];
```

Chapter 9

08_Sun Sensor Model

In order to model the sun sensor, the following information is necessary:

1. Attitude (which for us will be in the form of a quaternion)
2. Sun sensor mounting angle, g, from the spacecraft x-axis (61.5 degrees)
3. Sun sensor quantization (0.125 deg/count)
4. Sun sensor noise (0.05 degrees)

For a quaternion q=[q1;q2;q3;q4], the 3x3 attitude matrix, A, is computed as follows:

```
A(1,1)=  q1q1 - q2q2 - q3q3 + q4q4;
A(2,1)=    2*(q1q2 - q3q4);
A(3,1)=    2*(q1q3 + q2q4);
A(1,2)=    2*(q1q2 + q3q4);
A(2,2)= -q1q1 + q2q2 - q3q3 + q4q4;
A(3,2)=    2*(q2q3 - q1q4);
A(1,3)=    2*(q1q3 - q2q4);
A(2,3)=    2*(q2q3 + q1q4);
A(3,3)= -q1q1 - q2q2 + q3q3 + q4q4;
```

From the definition of an attitude:

$$\overline{S}_{body} = A \times \overline{S}_{GCI}$$

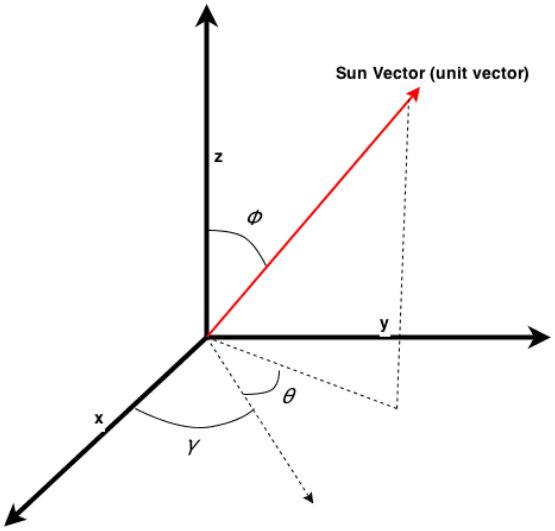


Figure 1: Sun Vector in Spacecraft Body Coordinates and Sun Sensor Angle Definitions

PROBLEM: Model a Sun Sensor as follows:

1. Using your propagated attitudes from Project 07, compute sun vectors in GCI using times in Project 07, and then compute the sun vectors in body coordinates.
2. Start Time is: `20060401.08505992969`
3. Compute the azimuth angle, q, from the mounting angle, g
4. Compute the angle from the x-y plane to the sun vector in body coordinates (sun angle), f
5. Add noise to the sun angle (HINT: use randn(1,1)*noise and make sure your units are correct)
6. Convert the sun angle to counts using the scale factor from Project 01 (ignore bias term)
7. Output the following:
    a. Time
    b. Quaternion
    c. Sun vector in gci
    d. Sun vector in body
    e. Angle in x-y plane from mounting angle
    f. Sun Angle with noise
    g. Sun Angle counts

Chapter 10

09_Magnetometer Model

In order to model the magnetometer, the following information is necessary:

1. Attitude (which for us will be in the form of a quaternion)
2. Magnetometer in GCI coordinates at attitude time
3. Magnetometer quantization (0.3 mG/count)
4. Magnetometer noise (0.1 mG)

For a quaternion $\bar{q} = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}$, the 3x3 attitude matrix is computed as follows:

$$A(\bar{q}) = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^4 & 2*(q_1q_2 + q_3q_4) & 2*(q_1q_3 - q_2q_4) \\ 2*(q_1q_2 - q_3q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^4 & 2*(q_2q_3 + q_1q_4) \\ 2*(q_1q_3 + q_2q_4) & 2*(q_2q_3 - q_1q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^4 \end{bmatrix}$$

From the definition of an attitude:

$$\bar{M}_B = A \times \bar{M}_R$$

PROJECT A:

1. `Start time is:  20060401.08505992969`
2. Use the following position and velocity:

```
r0 = [ 1.939171267526330e+03;
       7.053946077269909e+03;
       5.216461164024867e+03]; % initial position in km
v0 = [  7.728790449144201e-01;
       -5.188238121076679e+00;
        3.991379730081099e+00];% initial velocity in km/sec
```

3. Using propagated attitudes from Project 07, the orbital dynamics routines from Project 04, the magnetic dipole model from Project 05, and the corresponding magnetic field vectors in GCI, compute the magnetometer in body coordinates
4. Add noise to each component of the magnetic field(HINT: use randn(1,1)*noise and make sure your units are correct)

5. Convert the magnetometer values to counts using the equation from Project 1 using bias of 0 and .3 mG/count).
6. Output the following:
    a. Time
    b. Quaternion
    c. Magnetic Field vector in gci
    d. Magnetometer vector in body
    e. Magnetometer in Counts
    f. Magnetometer converted from counts to mG

PROJECT B:

Magnetometers have a significant number of error sources along with the reference magnetic field model. In order to confirm that your magnetometer model is actually consistent with the reference magnetic field, compute the magnitude of the magnetometer measurements and subtract from that the magnitude of the reference magnetic field vectors. The result should have a mean close to 0 and with white noise in the neighborhood of your input noise.

Chapter 11

10_Attitude Matrix Calculation

The attitude matrix, A, can simply be calculated using the Triad algorithm as follows:

$\hat{S}_b$ is the Sun unit vector in sc body coordinates
$\hat{M}_b$ is the Magnetic Field unit vector in sc body coordinates
$\hat{S}_R$ is the Sun unit vector in GCI coordinates (from model)
$\hat{M}_R$ is the Magnetic Field unit vector in GCI coordinates (from model)

For the B and R matrices as shown below (all vectors are 3x1 column vectors (i.e. 3 rows and 1 column))

$$B = \begin{bmatrix} \hat{S}_B & \hat{M}_B & \hat{S}_B \otimes \hat{M}_B \end{bmatrix}$$

$$R = \begin{bmatrix} \hat{S}_R & \hat{M}_R & \hat{S}_R \otimes \hat{M}_R \end{bmatrix}$$

$$B = A \times R$$

where A is the attitude matrix.

The attitude matrix can be calculated as follows:

$$A = B \times R^{-1}$$

PROJECT A: Perform a sanity check on sun sensor and magnetometer measurement by computing the angle between the observed sun and magnetometer vectors and comparing it to the angle between the reference sun and magnetic field vectors. The difference should be close to 0.

For large angles (>= 2.6 degrees)

$$q_{obs} = \cos^{-1}(\hat{S}_B \cdot \hat{M}_B)$$
$$q_{ref} = \cos^{-1}(\hat{S}_R \cdot \hat{M}_R)$$

For small angles (< 2.6 degrees)

$$q_{obs} = \sin^{-1}(\left|\hat{S}_B \text{ Ä } \hat{M}_B\right|)$$
$$q_{ref} = \sin^{-1}(\left|\hat{S}_R \text{ Ä } \hat{M}_R\right|)$$

PROJECT B: Develop a function to compute attitude matrices from sun sensor, magnetometer, reference sun, and reference magnetic field vectors. Check the attitude matrix as follows:

$$\text{D}q_S = \hat{S}_B - A \cdot \hat{S}_R$$
$$\text{D}q_M = \hat{M}_B - A \cdot \hat{M}_R$$

where $\text{D}q_S$ is the sun sensor measurement residual and $\text{D}q_M$ is the magnetometer measurement residual. In the presence of perfect measurements (no noise or systematic errors), the measurement residuals are 0. In the presence of white noise, the measurement residuals have a mean of 0 and a standard deviation in the ballpark of the sensor noise

Chapter

12

11_Attitude Comparison

The .mat file named "intern_project_st5_155.mat" contains the actual ST-5 (155) ground estimated attitude quaternions and rates. The formats of the two arrays are as follows:

q_truth (time(YYMMDD.HHMMSSmmm) q1 q2 q3 q4 flag(0=good))
rate_truth (time(YYMMDD.HHMMSSmmm) wx wy wz (rad/sec) flag(0=good))
sc_position_truth (time(YYMMDD.HHMMSSmmm) rx ry rz (km) vx vy vz (km/sec))

1. Start simulation with the following:

t0, q0, w0, r0, v0 from .mat file

2. Run simulation for 600 seconds from t0 with time increment of 0.05 seconds

3. Compare estimated quaternion calculated from measurements and reference vectors to truth_quaternions as follows:

interp_sim_quaternions=interp1(sim_times,sim_quaternions,truth_quat_times,'spline');

then normalize interp_sim_quaternions.

4. For propagated quaternions and the truth quaternions, do compute the spin axis vector in GCI coordinates as follows and compare the two:

   a. Compute attitude matrix from quaternion, A

b. Compute spin axis in GCI as follows: $A^{-1}\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

c. Compute the angle between the true spin axis and the simulated spin axis
    a. Plot the angle
    b. Compute the mean of the angle
    c. Compute the standard deviation of the angle

5. Compare simulated rate to rate_truth(:,2:4). Interpolate the simulated rates to the times of the true rates using the following:

interp_sim_rates=interp1(sim_times,sim_rates,truth_rate_times,'spline');

a. Plot the difference of the two rates (skipping the first 400 seconds of data)
b. Calculate the mean difference of each axis
c. Calculate the standard deviation of the difference of each axis.

6. Compare propagated position and velocity to the sc_position_truth(:,2:7)

7. Document why there may be differences.

Chapter

13

Conclusion!

I just want to say thank you very much for taking the time to read this book, I have spent a lot of time working on this project, and even though I truly enjoy it, it is still nice to see others take notice of my work!