

Earth Centered Earth Fixed Vehicle Simulation

**Chris Adan
Senior Project
March 16, 2006**

**California Polytechnic,
San Luis Obispo**

Disclaimer

This project was completed to fulfill the College of Engineering senior project requirement for attainment of the Baccalaureate degree. It has been graded and accepted as fulfillment of the degree requirement. No implication is made of its technical accuracy or reliability. Any use of this project is done so at the sole risk of the user. California Polytechnic State University at San Luis Obispo and its staff cannot be held liable for any use or misuse of the information contained herein.

Table of Contents

List of Figures	iii
List of Tables	iii
List of Symbols	iv
Math Nomenclature	iv
Subscripts	iv
Variables	iv
Abstract	1
Introduction	2
Procedure	3
Earth Centered Earth Fixed Frame	3
Body Frame	3
North-East-Down Frame	4
Longitude-Latitude-Altitude Frame	5
Euler Angles	5
Rotational Kinematics	5
Linear Kinematics	7
Results	9
Conclusion	11
APPENDIX A - matrixmath_4_0	12
Functions	12
Vector3 Example	13
Matrix3 Example	13
Code – H File	14
Code – CPP file	20
APPENDIX B - lost_physics_0015	48
ODE45	48
North East Down Frame	48
Euler Angles	49
Quaternions	49
Mass Properties	49
Code – H File	49
Code – CPP File	54
APPENDIX C – Quaternions in Depth	68
Rolfe and Staples	68
Microsoft Convention	68
Seen this before, don't remember where	69
Quaternion Properties of Interest	69
APPENDIX D – ECEF	70
Code – Cpp File	70
References	80

List of Figures

Figure 1 - Earth Centered Earth Fixed Axis	3
Figure 2 - Body Axis.....	4
Figure 3 - Latitude of Flight.....	9
Figure 4 - Longitude of Flight	9
Figure 5 - Altitude of Flight.....	10

List of Tables

Table 1 – Euler Angles	5
------------------------------	---

List of Symbols

Math Nomenclature

\bar{A}	Vector
\underline{A}	Matrix
\dot{A}	Time Rate of Change

Subscripts

<i>body</i>	Coordinate frame fixed to the body of the vehicle
<i>Inertial</i>	Coordinate frame fixed in space
<i>w</i>	First quaternion parameter
<i>x</i>	Second quaternion parameter or referring to the X-axis
<i>y</i>	Third quaternion parameter or referring to the Y-axis
<i>z</i>	Fourth quaternion parameter or referring to the Z-axis

Variables

ϕ	Roll
θ	Pitch
ψ	Yaw
\underline{DCM}	Direction Cosign Matrix
\bar{F}	Force
\underline{I}	Inertia
\bar{M}	Moment
P	Roll Rate
Q	Pitch Rate
R	Yaw Rate
\bar{X}	Position
e	Quaternion
$mass$	Mass
\bar{w}	Rotation Rate

Abstract

Flight simulation is a critical tool for testing stability and controls. It allows a more complex analysis than a simple linear model. All too often flight simulation doesn't properly simulate the environment. A common simplification is the assumption that the Earth is flat. The flat Earth model simplifies the math for basic control problems of a single plant. However it is inaccurate for simulations involving traveling long distances, such as UAV navigation or spacecraft. The solution is to use an earth centered earth fixed model (ECEF). Though not as complex as an earth centered earth rotating model (ECER), the model still allows for the basic concepts of navigation and orbits to be investigated.

Introduction

All too often flight simulation doesn't properly simulate the environment. A common simplification is the assumption that the Earth is flat. The flat Earth model simplifies the math for basic control problems of a single plant. However it is in-accurate for simulations involving traveling long distances, such as UAV navigation or spacecraft. The solution is to use an earth centered earth fixed model (ECEF). Though not as complex as an earth centered earth rotating model (ECER), the model still allows for the basic concepts of navigation and orbits to be investigated.

Many aircraft simulation books assume the flat plate world. Very few, if any involve a round earth assumption. Spacecraft simulation books rarely deal with planets except as a means of providing gravity to perform orbits. This math model is an attempt to bridge the gap between both aircraft and spacecraft math models.

The gravity model used in the simulation is independent of this senior project. As of now the gravity model assumes a round earth and has no oblateness effects.

A word of warning, the code base of the Calpoly Flight Simulator is constantly changing. This senior project report is more of a guide of how the system should operate apposed to being a strict form of documentation on the operations of the system. It's quite possible there are some errors in both this document and the source code attached. I'm sure there are a lot of algorithms that aren't optimized for speed as well.

A special thanks goes out to Mike Kosman for his work with the matrix inverse code segment. Another bigger thanks goes out to Keith Rothman for performing the last batch of code re-organization.

Procedure

It's important to quantify the system before any analysis can be performed. To make sure all audiences are caught up, a list of definitions for aerospace and math definitions is provided in this section.

Earth Centered Earth Fixed Frame

The Earth Centered Earth Fixed (ECEF) frame is considered the inertial frame. Positive X is out where the prime meridian intercepts the equator, positive Y is ninety degrees east of X along the equator and positive Z is through the geographic North Pole. All forces are transformed to this frame. Once in this frame the forces are divided by mass to get accelerations, and the accelerations are integrated to get velocity. Velocity is integrated to get position. Acceleration, velocity and position are then transformed into the body frame and feed back into the engine, landing gear and aerodynamic systems.

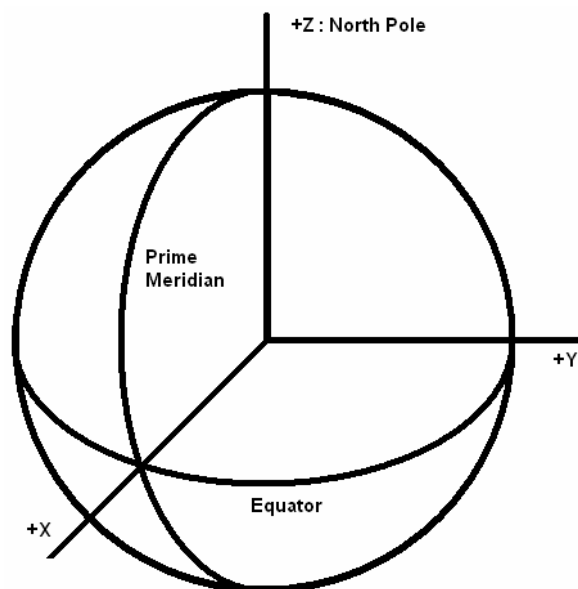


Figure 1 - Earth Centered Earth Fixed Axis

Body Frame

The body frame is the principle frame in which stability and controls are derived. The positive X is out the nose, positive Y is out the right wing, and positive Z is down. The coordinate system is right handed.

Engine, landing gear and aerodynamic forces are often calculated in this frame. Since mass is a scalar property the process of transforming the forces from this frame to another is to simply multiply by the correct direction cosign matrix.

However the process of transforming moments into other frames is more complex. Inertia is a matrix, not a scalar, and as a result the inertia matrix would have to be pre multiplied by one direction cosign matrix and post multiplied by the transpose of the direction cosign matrix. This process becomes computationally expensive quickly. A better method is to integrate the angular accelerations in the body frame and add in the Coriolis Effect.

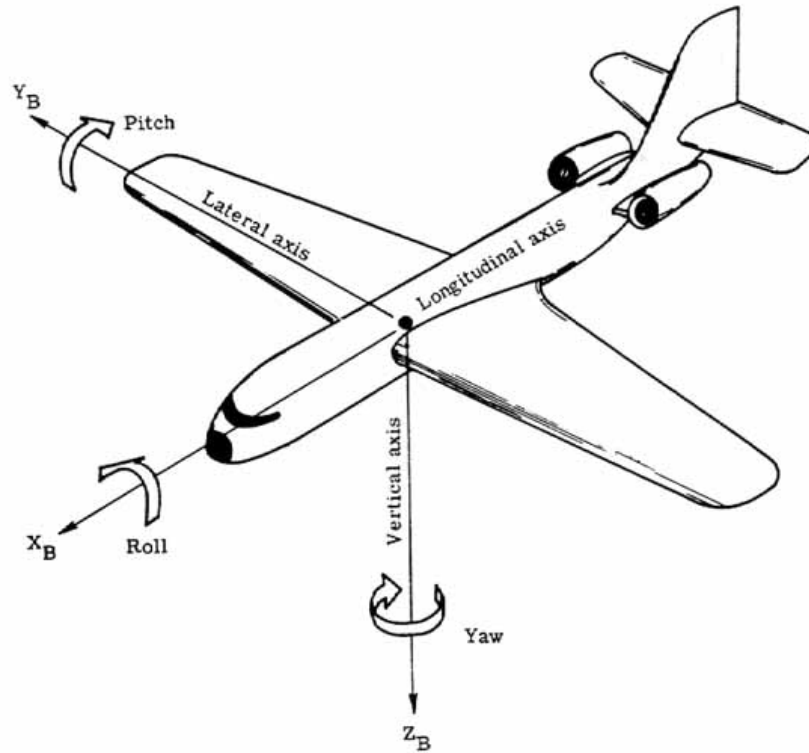


Figure 2 - Body Axis

North-East-Down Frame

North-East-Down (NED) is the most difficult frame to manage. Positive X is north, positive Y is east, and positive Z is down. The X-Plane graphics engine needed angles between the NED frame and the Body frame to properly display graphics. The process to acquire those angles is critical and will be repeated along with the derivation in Appendix B.

Longitude-Latitude-Altitude Frame

The Longitude-Latitude-Altitude frame (LLA) is the final frame of interest. The graphics system requires the LLA to do its job as a render engine. Other components, namely the atmosphere, and flight control systems, require the altitude for these computations. The LLA to ECEF conversion system used is the WS 84 Ellipsoid. The function is listed in Appendix G.

Euler Angles

Euler angles are defined as the angles between the aircraft's body frame and the north east down frame. The order of the transformation is rotate about x, rotate about y, then rotate about z. This process is called a 3-2-1 transformation. The rotation about X is called roll and given the Greek letter phi. Rotation about Y is called pitch and given the Greek letter theta. Rotation about Z is called yaw and given the Greek letter psi. It should be noted that psi is also equivalent to heading, and is undefined if the pitch is straight up or straight down. Euler angles are only used in this project as an output data for the graphics system and for the convenience of the pilot or control systems that work with the NED frame. Table 1 displays the ranges of Euler Angles.

Name	Symbol	Axis of Rotation	Min Value	Max Value
Roll	ϕ	X	-180	+180
Pitch	θ	Y	-90	+90
Yaw	ψ	Z	0	360

Table 1 - Euler Angles

Rotational Kinematics

Rotational kinematics depends heavily on the frame in which they're computed. The generalized rotation equation is

$$\bar{M} = \underline{I} \cdot \dot{\bar{w}}$$

There are two ways to handle moments. One method is to convert all the moments into the inertial frame, and handle all rotations in the inertial frame. This method has a major drawback; the inertia matrix must be transformed to the inertial frame. This is a major process, as it multiplies a three by three matrix by another three-by-three matrix. The other method is to perform the differentiation in the body frame. In that frame the inertia matrix is constant for the math model, and the rotation rates are in the body frame, which

is good for pilots and control systems. The equation below is the basis for the body frame kinematics.

$$\dot{\bar{M}}_{body} = \underline{I}_{body} \cdot \dot{\bar{w}}_{body} + \bar{w}_{body} \times \underline{I}_{body} \cdot \bar{w}_{body}$$

Solving for rotational acceleration gives the following equation.

$$\dot{\bar{w}}_{body} = \underline{I}_{body}^{-1} \cdot \left[\dot{\bar{M}}_{body} - (\bar{w}_{body} \times \underline{I}_{body} \cdot \bar{w}_{body}) \right]$$

Integrating the body acceleration gives the body rates. To get the body angles requires a bit of trickery. Euler angles have singularities in there solutions, so we should use quaternion parameters to handle orientation. The quaternion dot matrix is as follows [1].

$$\begin{bmatrix} \dot{e}_w \\ \dot{e}_x \\ \dot{e}_y \\ \dot{e}_z \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & P & Q & R \\ -P & 0 & -R & Q \\ -Q & R & 0 & -P \\ -R & -Y & P & 0 \end{bmatrix} \begin{bmatrix} e_w \\ e_x \\ e_y \\ e_z \end{bmatrix}$$

This matrix is the inverse of what is expected, but it works.

Quaternions can be initialized by the following equations. These equations convert euler angles to quaternions.

$$\begin{aligned} e_w &= \cos\left(\frac{\psi}{2}\right) \cos\left(\frac{\theta}{2}\right) \cos\left(\frac{\phi}{2}\right) + \sin\left(\frac{\psi}{2}\right) \sin\left(\frac{\theta}{2}\right) \sin\left(\frac{\phi}{2}\right) \\ e_x &= \cos\left(\frac{\psi}{2}\right) \cos\left(\frac{\theta}{2}\right) \sin\left(\frac{\phi}{2}\right) - \sin\left(\frac{\psi}{2}\right) \sin\left(\frac{\theta}{2}\right) \cos\left(\frac{\phi}{2}\right) \\ e_y &= \cos\left(\frac{\psi}{2}\right) \sin\left(\frac{\theta}{2}\right) \cos\left(\frac{\phi}{2}\right) + \sin\left(\frac{\psi}{2}\right) \cos\left(\frac{\theta}{2}\right) \sin\left(\frac{\phi}{2}\right) \\ e_z &= -\cos\left(\frac{\psi}{2}\right) \sin\left(\frac{\theta}{2}\right) \sin\left(\frac{\phi}{2}\right) + \sin\left(\frac{\psi}{2}\right) \cos\left(\frac{\theta}{2}\right) \cos\left(\frac{\phi}{2}\right) \end{aligned}$$

The process of converting quaternions to Euler angles is equally messy. Don't let the arc functions fool you. If the quaternions aren't normalized they'll result in non-real results.

Roll

$$\phi = \tan^{-1} \left(\frac{2(e_x \cdot e_y + e_w \cdot e_z)}{-e_w^2 - e_x^2 + e_y^2 + e_z^2} \right)$$

Heading, Yaw

$$\psi = \tan^{-1} \left(\frac{2(e_w \cdot e_x + e_y \cdot e_z)}{e_w^2 - e_x^2 - e_y^2 + e_z^2} \right)$$

Pitch

$$\theta = \sin^{-1} (2(e_x \cdot e_z - e_y \cdot e_w))$$

Populating direction cosine matrixes is an exciting adventures in it self.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{inertial} = \begin{bmatrix} e_w^2 - e_x^2 - e_y^2 + e_z^2 & 2(e_w \cdot e_x - e_y \cdot e_z) & 2(e_w \cdot e_y + e_x \cdot e_z) \\ 2(e_w \cdot e_x + e_y \cdot e_z) & -e_w^2 + e_x^2 - e_y^2 + e_z^2 & 2(e_x \cdot e_y - e_w \cdot e_z) \\ 2(e_w \cdot e_y - e_x \cdot e_z) & 2(e_x \cdot e_y + e_w \cdot e_z) & -e_w^2 - e_x^2 + e_y^2 + e_z^2 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{body}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{body} = \begin{bmatrix} e_w^2 - e_x^2 - e_y^2 + e_z^2 & 2(e_w \cdot e_x + e_y \cdot e_z) & 2(e_w \cdot e_y - e_x \cdot e_z) \\ 2(e_w \cdot e_x - e_y \cdot e_z) & -e_w^2 + e_x^2 - e_y^2 + e_z^2 & 2(e_x \cdot e_y + e_w \cdot e_z) \\ 2(e_w \cdot e_y + e_x \cdot e_z) & 2(e_x \cdot e_y - e_w \cdot e_z) & -e_w^2 - e_x^2 + e_y^2 + e_z^2 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{inertial}$$

Linear Kinematics

The linear dynamics are more interesting. Forces come into the physics class in the body frame. The body frame is not cool because of the Coriolis effects. All body forces are transformed to the inertial frame. Once in an inertial frame of reference, Newtonian Integration may now be performed. The equation process is detailed below.

$$\bar{F}_{inertial} = \underline{DCM}_{body}^{inertial} \cdot \bar{F}_{body}$$

$$\ddot{\bar{X}}_{inertial} = \frac{\bar{F}_{inertial}}{mass}$$

$$\dot{\bar{X}}_{inertial} = \int \ddot{\bar{X}}_{inertial} dt$$

$$\bar{X}_{inertial} = \int \dot{\bar{X}}_{inertial} dt$$

To get velocity into the body frame the following equation is used.

$$\dot{\bar{X}}_{body} = \underline{DCM}_{inertial}^{body} \cdot \dot{\bar{X}}_{inertial}$$

The same is true for the body acceleration.

Results

Once the code was compiled and debugged a series of test flights were performed. One such test flight was performed by a crude autopilot. The main control loops kept the aircraft straight, level, and at an altitude of about 40,000 ft. The autopilot took off from San Luis Obispo Airport and flew out of runway 11.

Figure 3 shows the latitude of the flight. As can be seen the craft heads in a southern direction. The southerly direction switches to a northern direction around the 16th hour of flight. This behavior is consistent with a satellite ground tracking view. If the aircraft were allowed to fly for a longer period the latitude would appear as a sin wave. The longitude in Figure 4 has a fairly constant rate of change, but the slope changes slightly relative to the latitude. This is because of the WS84 ellipsoid equations used to get ECEF to LLA.

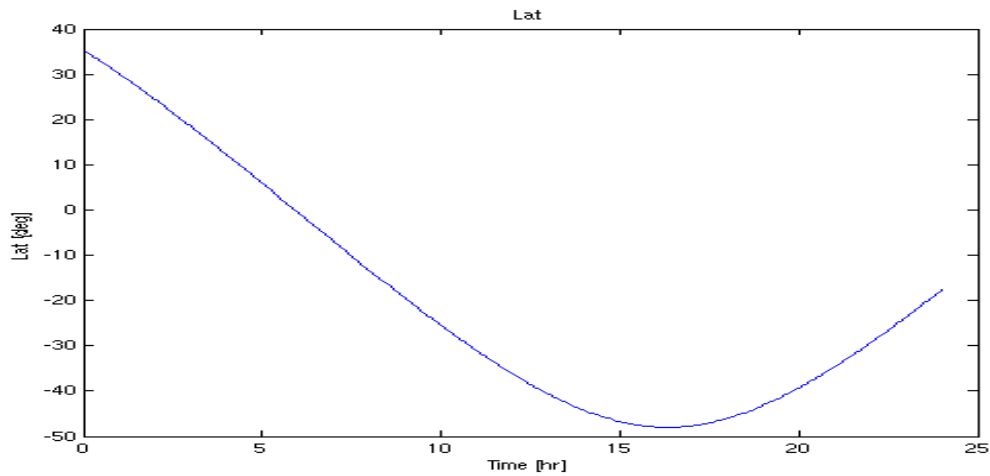


Figure 3 - Latitude of Flight

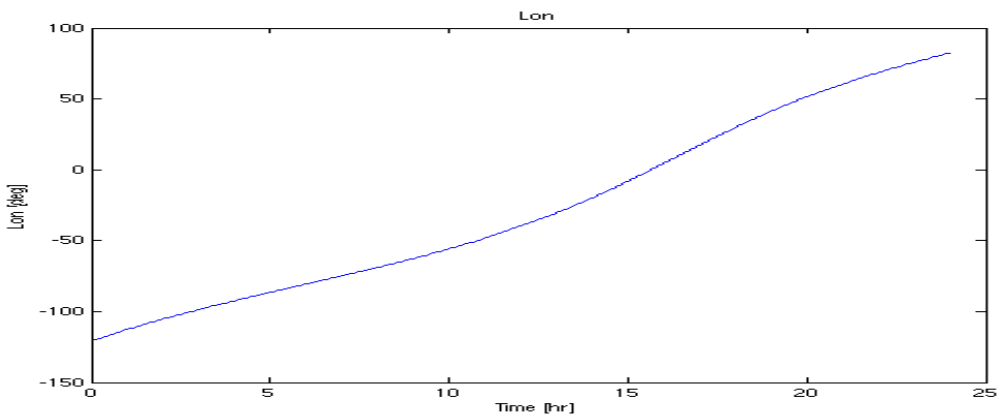


Figure 4 - Longitude of Flight

The altitude graph, Figure 5 shows the take off from San Luis Obispo and the aircraft cruising of around 42,000 feet. The altitude is not constant for a number of reasons. The first is there could be an issue with the pitch controller to provide for slight variations. The second is due to the ECEF to LLA makes the altitude a function of latitude and longitude.

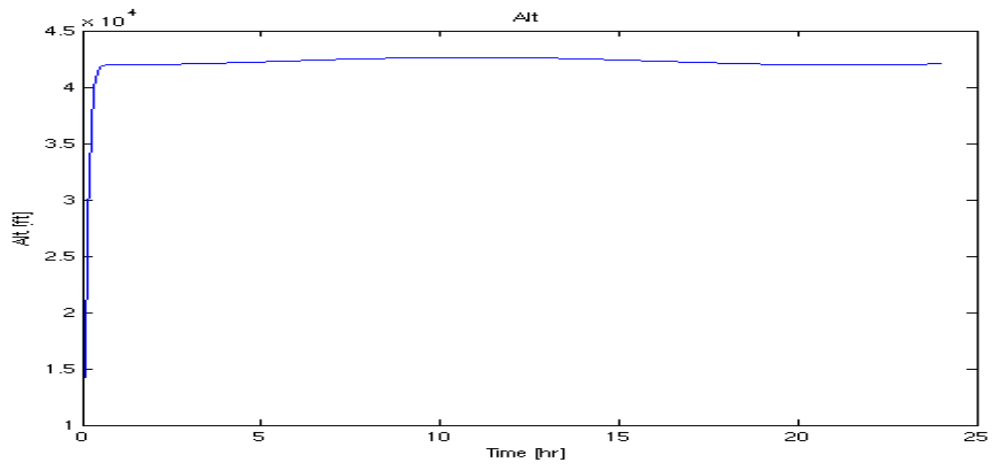


Figure 5 - Altitude of Flight

Conclusion

Currently the Calpoly's flight simulator has been operating off this ECEF model. The model has proven to be stable for the year it's been in operation. So far it has mainly served as a long distance aircraft and UAV flight tool, but it still has the potential to be used as a spacecraft simulation.

No longer is the flight simulator bound to a single plate, or an odd assembly of plates adjacent to each other. The ability to fly any where in the world, or any place with in the Earths sphere of influence has been bestowed to the flight simulator. The potential for future development on aircraft and spacecraft simulation models is now truly limitless.

APPENDIX A - matrixmath_4_0

matrixmath_4_0 is a set of functions and data structures for the purpose of linear algebra. The math set includes 2d, 3d, and 4d vectors and matrices.

Functions

Common functions exist for each set of dimensions.

Zero2(), Zero3(), Zero4(): Sets all vector values to zero.
Eye2(), Eye3(), Eye4(): Sets a matrix to the identity.
Dot(a,b): Dot product
Cross(a,b): Cross product, only for 3 dimensional vectors
MakeUnitVector(a) Makes a 3 or 4 dimensional vector a unit vector.

Special house keeping functions also exist.

Quat_To_Euler: Turns a quaternion 4d vector to an Euler 3d vector
FindQuat: Turns a 3d Euler angle vector into a 4d quaternion vector.
A_to_B: Transformation matrix handling. Will generate an Earth to Body style matrix. Takes quaternion vector or Euler angle vector.
B_to_A: Transformation matrix handling. Will generate a Body to Earth style matrix. Takes quaternion vector or Euler angle vector.
LLA_to_ECEF: Converts Latitude, Longitude Altitude coordinates to Earth Centered Earth fixed coordinates.
ECEF_to_LLA: Converts Earth Centered Earth Fixed coordinates to Latitude, longitude, altitude coordinates.
Dismantle_A_to_B: Pulls the Euler angles out of the Earth to Body transformation matrix.
Dismantle_B_to_A: Pulls the Euler angles out of the Body to Earth transformation matrix.

Vector3 Example

The Vector3 is a grouping of three double point precision numbers. The data structure is declared in C++ by the following line.

```
struct Vector3      {      double      x, y, z;  };
```

To declare a variable of the type Vector3 use the following code.

```
Vector3 Position;
```

To initialize the value of “Position” the following three lines of code are used.

```
Position.x = 1;  
Position.y = 2;  
Position.z = 3;
```

A series of operator functions were developed to provide most simple vector operations, such as addition and subtraction. In those events the vectors are treated exactly like any other type of variable. The list of operations are...

- Addition
- Subtraction
- Scaling

Matrix3 Example

The Matrix3 grouping was done in an attempt to simplify memory management for the passing of arrays through functions and other objects. The matrix 3 data structure is declared like so.

```
struct Matrix3      {      double a[3][3];      };
```

The first array coordinate is the row, the second is the column. Declaring a Matrix3 variable is done with the following code.

```
Matrix3 DCM;
```

To set “DCM” to identity requires the following operation.

```
DCM.a[0][0] = 1;  DCM.a[0][1] = 0;  DCM.a[0][2] = 0;  
DCM.a[1][0] = 0;  DCM.a[1][1] = 1;  DCM.a[1][2] = 0;  
DCM.a[2][0] = 0;  DCM.a[2][1] = 0;  DCM.a[2][2] = 1;
```

Note that C++ begins counting positions at zero, and not one.

A series of math operators were created for matrixes and vectors. To multiply a matrix by a vector simply uses an asterisk between the two. The list of operations provided is...

- Matrix multiplied by a Vector
- Matrix multiplied by a Matrix
- Matrix multiplied by a scalar
- Matrix divided by a scalar
- Matrix added with a Matrix
- Matrix subtracted from a matrix

Code – H File

```
///=====
=====//
// MODULE:      MatrixMath_3_1.c

//
// AUTHOR(S):   Chris Adan [CJA2], Mike Francis Kosman [MFK]
//
// DATE:        01/05/04
//
// Copyright (c) ALL RIGHTS RESERVED
//
// REVISION HISTORY:
//
//   REV  AUTHOR    DATE        DESCRIPTION
//   ---  -
//   0    CJA2      01/05/04     Created Function Library
//   1     CJA2     01/07/04     Added this comment section
//   2     CJA2     01/12/04     Changed functions to extern
//                                     Added multi-load safty
//   3     CJA2     01/26/04     Added support for custom structs
//   4     CJA2     01/30/04     Added Kinematic Integration
//   5     CJA2     02/23/04     Added rotational kinematic integration
//   6     MFK      03/01/04     Added Matrix Inverse
//   7     CJA2     05/03/04     Code clean up and integration of
components
//   8     CJA2     05/22/04     Added 2d matrix code
//   9     CJA2     05/22/04     Added Integrators
//  10    CJA2     07/30/04     Overhaul, Overloading of C++ Operators
//
//   S-mex: See simulink/src/sfuntmpl.doc
//
//   Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights
Reserved.
//   $Revision: 1.3
//
//   CREDITS:
//   This file is based on "Flight Simulation"
//
//   NOTES:
//
```

```

// These are the standard vector notations...
// a[m][n] = matrix
// m = row
// n = column
// there for...
// b[3] = a 3x1 matrix
//
//
// Matrix Notes:
//     Declaring a m x 1 Matrix Row = m
//         double MatrixA[row]
//         These vectors are lower case letters
//
//     Declaring a m x n Matrix.  Row = m, Col = n
//         double MatrixB[row][col]
//         These matrixes are upper case letters
//
// Quaternion Definitions
//     e0 = cos(theta/2)
//     e1 = A sin(theta/2)
//     e2 = B sin(theta/2)
//     e3 = C sin(theta/2)
//
//
// UPDATE_FLAG :      marks places that will need to be changed
//                   later to match the new sixdof
//
// INTEGRATION_FLAG :      Marks where the integration process was
// haulted.
//
//=====
//====//
#ifndef MATRIXMATH_4_0_H
#define MATRIXMATH_4_0_H

#include <stdio> //    for printing data
#include <cmath> //    for sin, cos, power, ect
#include "globals_1_0.h"

// Basic Vectors
struct Vector2    {    double    x, y;        };
struct Vector3    {    double    x, y, z;     };
struct Vector4    {    double    w, x, y, z;  };

// Basic Matrices with out the x
struct Matrix2    {    double    a[2][2];    };
struct Matrix3    {    double    a[3][3];    };
struct Matrix4    {    double    a[4][4];    };

// Chris Adan
// 05/22/04
// Contains a snap shot of physics data
struct state_struct
{

    Vector3 Pos, Vel, Acc, Force;
    Vector3 Theta, Omega, Alpha, Moment;

```

```

        Vector4 Quat, QuatDot;
};

//=====//
//
//      //
//      Function Declorations
//      //
//      //
//=====//

//=====//
//      Basic Addition Operators
//      //
//=====//
Vector2 operator + (const Vector2 lhs, const Vector2 rhs);
Vector3 operator + (const Vector3 lhs, const Vector3 rhs);
Vector4 operator + (const Vector4 lhs, const Vector4 rhs);

Matrix2 operator + (const Matrix2 lhs, const Matrix2 rhs);
Matrix3 operator + (const Matrix3 lhs, const Matrix3 rhs);
Matrix4 operator + (const Matrix4 lhs, const Matrix4 rhs);

//*****//
//      Advance Addition, +=
//      //
//*****//
Vector2 operator += (Vector2 &lhs, const Vector2 rhs);
Vector3 operator += (Vector3 &lhs, const Vector3 rhs);
Vector4 operator += (Vector4 &lhs, const Vector4 rhs);

Matrix2 operator += (Matrix2 &lhs, const Matrix2 rhs);
Matrix3 operator += (Matrix3 &lhs, const Matrix3 rhs);
Matrix4 operator += (Matrix4 &lhs, const Matrix4 rhs);

//=====//
//      Basic Subtraction Operators
//      //
//=====//
Vector2 operator - (const Vector2 rhs);
Vector3 operator - (const Vector3 rhs);
Vector4 operator - (const Vector4 rhs);

Vector2 operator - (const Vector2 lhs, const Vector2 rhs);
Vector3 operator - (const Vector3 lhs, const Vector3 rhs);
Vector4 operator - (const Vector4 lhs, const Vector4 rhs);

Matrix2 operator - (const Matrix2 lhs, const Matrix2 rhs);
Matrix3 operator - (const Matrix3 lhs, const Matrix3 rhs);
Matrix4 operator - (const Matrix4 lhs, const Matrix4 rhs);

//*****//
//      Advance Addition, -=
//      //
//*****//

```

```

Vector2 operator -= (Vector2 &lhs, const Vector2 rhs);
Vector3 operator -= (Vector3 &lhs, const Vector3 rhs);
Vector4 operator -= (Vector4 &lhs, const Vector4 rhs);

Matrix2 operator -= (Matrix2 &lhs, const Matrix2 rhs);
Matrix3 operator -= (Matrix3 &lhs, const Matrix3 rhs);
Matrix4 operator -= (Matrix4 &lhs, const Matrix4 rhs);

//*****//
// Multiplication types *
//
// There should be no *= or /= with matrix math //
//*****//
// Base type,
// dot product 1xn * nx1 = 1x1
// ???          nx1 * 1xn = nxn
// std          nxn * nx1 = nx1
// std          1xn * nxn = 1xn
// std          nxn * nxn = nxn

double Dot(const Vector2 lhs, const Vector2 rhs);
double Dot(const Vector3 lhs, const Vector3 rhs);
double Dot(const Vector4 lhs, const Vector4 rhs);

Matrix2 operator * (const Vector2 lhs, const Vector2 rhs);
Matrix3 operator * (const Vector3 lhs, const Vector3 rhs);
Matrix4 operator * (const Vector4 lhs, const Vector4 rhs);

Vector2 operator * (const Matrix2 lhs, const Vector2 rhs);
Vector3 operator * (const Matrix3 lhs, const Vector3 rhs);
Vector4 operator * (const Matrix4 lhs, const Vector4 rhs);

Vector2 operator * (const Vector2 lhs, const Matrix2 rhs);
Vector3 operator * (const Vector3 lhs, const Matrix3 rhs);
Vector4 operator * (const Vector4 lhs, const Matrix4 rhs);

Matrix2 operator * (const Matrix2 lhs, const Matrix2 rhs);
Matrix3 operator * (const Matrix3 lhs, const Matrix3 rhs);
Matrix4 operator * (const Matrix4 lhs, const Matrix4 rhs);

Vector2 operator * (double lhs, const Vector2 rhs);
Vector2 operator * (const Vector2 lhs, double rhs);
Vector3 operator * (double lhs, const Vector3 rhs);
Vector3 operator * (const Vector3 lhs, double rhs);
Vector4 operator * (double lhs, const Vector4 rhs);
Vector4 operator * (const Vector4 lhs, double rhs);

Matrix2 operator * (double lhs, const Matrix2 rhs);
Matrix2 operator * (const Matrix2 lhs, double rhs);
Matrix3 operator * (double lhs, const Matrix3 rhs);
Matrix3 operator * (const Matrix3 lhs, double rhs);
Matrix4 operator * (double lhs, const Matrix4 rhs);
Matrix4 operator * (const Matrix4 lhs, double rhs);

Vector2 operator / (const Vector2 lhs, double rhs);
Vector3 operator / (const Vector3 lhs, double rhs);
Vector4 operator / (const Vector4 lhs, double rhs);

```

```

Vector2 operator /= (const Vector2 lhs, double rhs);
Vector3 operator /= (const Vector3 lhs, double rhs);
Vector4 operator /= (const Vector4 lhs, double rhs);

Matrix2 operator / (const Matrix2 lhs, double rhs);
Matrix3 operator / (const Matrix3 lhs, double rhs);
Matrix4 operator / (const Matrix4 lhs, double rhs);

//=====//
//   State Struct supporting functions                               //
//=====//
//   Initilize a vector to zero
extern Vector2 Zero2 ();
extern Vector3 Zero3 ();
extern Vector4 Zero4 ();
//extern state_struct ZeroState ();

//=====//
//   Matrix Initiliation functions                               //
//=====//
//   Set up Identity matrix
extern Matrix2 Eye2();
extern Matrix3 Eye3();
extern Matrix4 Eye4();

//=====//
//   Inv
//           //
//   Desc:
//           //
//           Inverses a matrix.  This is an update to the //
//           Mike Kosman code to include Matrix Structures. //
//   Bugs:
//           //
//           * Doesn't handle singular matrixes //
//=====//
extern Matrix3 Inv(Matrix3 Matrix);
extern Matrix3 Cofactor(Matrix3 aMatrix);
extern double Determinate(Matrix3 aMatrix, Matrix3 ACofactor);
extern double Determinate(double a[2][2]);
extern Matrix3 Adjoint(Matrix3 Matrix);
extern double Minor(int m, int n, Matrix3 Matrix);

//=====//
//
//           //
//   Integration Methods
//           //
//           //
//=====//

//   ADAMS_BASHFORTH_MOULTON
//   The sim's method
extern double Integrate_ADAM_BASHFORTH_MOULTON(double curr, double
prev, double deltaT);

```

```

extern Vector2 Integrate_ADAM_BASHFORTH_MOULTON(Vector2 curr,
Vector2 prev, Vector2 old, double deltaT);
extern Vector3 Integrate_ADAM_BASHFORTH_MOULTON(Vector3 curr,
Vector3 prev, Vector3 old, double deltaT);
extern Vector4 Integrate_ADAM_BASHFORTH_MOULTON(Vector4 curr,
Vector4 prev, Vector4 old, double deltaT);

//=====================================================//
// Quaternion Math Soul Slaughtering Omni Attack IX //
//=====================================================//

// Chris Adan
// 08/05/04
// Checked 06/02/04
// Bugs - May be sign errors, no two books have the same matrix
// Main Quaternion Rate Matrix
// This can be life or death
// This contains "DEADLY_BUG_0", a fix with no explanation or
understanding.
extern Matrix4 FindQuatDotMatrix(Vector3 Omega);
extern Vector4 FindQuat(Vector3 Theta);

// Returns the transformation matrix
// Frame A contains Frame B
extern Matrix3 A_to_B(const Vector4 quat);
extern Matrix3 A_to_B(Vector3 Theta);

extern Matrix3 B_to_A (const Vector4 quat);
extern Matrix3 B_to_A (const Vector3 Theta);

// Finds the quaternions given phi, theta, psi
// From "Flight Simulation", Cambridge Series of Aerospace
//extern Vector4 FindQuat(Vector3 Theta);
extern double Mag(Vector4 Q);
extern double Mag(Vector3 Q);

// From rolph and staples
extern Vector3 Quat_To_Euler(Vector4 quat);

// Chris Adan
// 04 01 04
// Cross Product, for 3x1 vectors.
// Vector a x Vector b
extern Vector3 Cross(const Vector3 a, const Vector3 b);
extern Vector4 MakeUnitVector(Vector4 Quat);
extern Vector3 MakeUnitVector(Vector3 vector);

extern Vector3 LLA_to_ECEF (Vector3 LLA);
extern Vector3 ECEF_to_LLA (Vector3 Pos);

extern Matrix4 FindQuatDotMatrix(Vector3 Omega);

extern Vector3 Dismantle_A_to_B(Matrix3 LMN);
extern Vector3 Dismantle_B_to_A(Matrix3 NML);
extern double Sgn(double number);

bool IsNaN(Vector2);

```

```

bool IsNaN(Vector3);
bool IsNaN(Vector4);

#endif // #ifndef (MATRIXMATH_4_0_H)

```

Code – CPP file

```

#include "MatrixMath_4_0.h"

//=====//
//   Basic Addition Operators
//
//=====//
//   Vector2 = Vector2 + Vector2
Vector2 operator + (const Vector2 lhs, const Vector2 rhs)
{
    Vector2 ans;
    ans.x = lhs.x + rhs.x;
    ans.y = lhs.y + rhs.y;
    return ans;
};

//   Vector3 = Vector3 + Vector3
Vector3 operator + (const Vector3 lhs, const Vector3 rhs)
{
    Vector3 ans;
    ans.x = lhs.x + rhs.x;
    ans.y = lhs.y + rhs.y;
    ans.z = lhs.z + rhs.z;
    return ans;
};

//   Vector4 = Vector4 + Vector4
Vector4 operator + (const Vector4 lhs, const Vector4 rhs)
{
    Vector4 ans;
    ans.w = lhs.w + rhs.w;
    ans.x = lhs.x + rhs.x;
    ans.y = lhs.y + rhs.y;
    ans.z = lhs.z + rhs.z;
    return ans;
};

//   Matrix2 = Matrix2 + Matrix2
Matrix2 operator + (const Matrix2 lhs, const Matrix2 rhs)
{
    int row;
    int col;
    Matrix2 ans;
    for (row = 0; row < 2; row++)
        for (col = 0; col < 2; col++)
            ans.a[row][col] = lhs.a[row][col] + rhs.a[row][col];
    return ans;
};

```

```

//    Matrix3 = Matrix3 + Matrix3
Matrix3 operator + (const Matrix3 lhs, const Matrix3 rhs)
{
    int row;
    int col;
    Matrix3 ans;
    for (row = 0; row < 3; row++)
        for (col = 0; col < 3; col++)
            ans.a[row][col] = lhs.a[row][col] + rhs.a[row][col];
    return ans;
};

//    Matrix4 = Matrix4 + Matrix4
Matrix4 operator + (const Matrix4 lhs, const Matrix4 rhs)
{
    int row;
    int col;
    Matrix4 ans;
    for (row = 0; row < 4; row++)
        for (col = 0; col < 4; col++)
            ans.a[row][col] = lhs.a[row][col] + rhs.a[row][col];
    return ans;
};

//*****//
//    Advance Addition, +=
//
//*****//
//    Vector2 = Vector2 + Vector2
Vector2 operator += (Vector2 &lhs, const Vector2 rhs)
{
    lhs.x += rhs.x;
    lhs.y += rhs.y;
    return lhs;
};

//    Vector3 = Vector3 + Vector3
Vector3 operator += (Vector3 &lhs, const Vector3 rhs)
{
    lhs.x += rhs.x;
    lhs.y += rhs.y;
    lhs.z += rhs.z;
    return lhs;
};

//    Vector4 = Vector4 + Vector4
Vector4 operator += (Vector4 &lhs, const Vector4 rhs)
{
    lhs.w += rhs.w;
    lhs.x += rhs.x;
    lhs.y += rhs.y;
    lhs.z += rhs.z;
    return lhs;
};

//    Matrix2 = Matrix2 + Matrix2

```

```

Matrix2 operator += (Matrix2 &lhs, const Matrix2 rhs)
{
    int row;
    int col;
    for (row = 0; row < 2; row++)
        for (col = 0; col < 2; col++)
            lhs.a[row][col] += rhs.a[row][col];
    return lhs;
};

// Matrix3 = Matrix3 + Matrix3
Matrix3 operator += (Matrix3 &lhs, const Matrix3 rhs)
{
    int row;
    int col;
    for (row = 0; row < 3; row++)
        for (col = 0; col < 3; col++)
            lhs.a[row][col] += rhs.a[row][col];
    return lhs;
};

// Matrix4 = Matrix4 + Matrix4
Matrix4 operator +=(Matrix4 &lhs, const Matrix4 rhs)
{
    int row;
    int col;
    for (row = 0; row < 4; row++)
        for (col = 0; col < 4; col++)
            lhs.a[row][col] += rhs.a[row][col];
    return lhs;
};

//=====//
// Basic Subtraction Operators
//
//=====//

// Vector2 = - Vector2
Vector2 operator - (const Vector2 rhs)
{
    Vector2 ans;
    ans.x = - rhs.x;
    ans.y = - rhs.y;
    return ans;
};

// Vector3 = Vector3 - Vector3
Vector3 operator - (const Vector3 rhs)
{
    Vector3 ans;
    ans.x = - rhs.x;
    ans.y = - rhs.y;
    ans.z = - rhs.z;
    return ans;
};

```

```

//    Vector4 = Vector4 - Vector4
Vector4 operator - (const Vector4 rhs)
{
    Vector4 ans;
    ans.w = - rhs.w;
    ans.x = - rhs.x;
    ans.y = - rhs.y;
    ans.z = - rhs.z;
    return ans;
};

//    Vector2 = Vector2 - Vector2
Vector2 operator - (const Vector2 lhs, const Vector2 rhs)
{
    Vector2 ans;
    ans.x = lhs.x - rhs.x;
    ans.y = lhs.y - rhs.y;
    return ans;
};

//    Vector3 = Vector3 - Vector3
Vector3 operator - (const Vector3 lhs, const Vector3 rhs)
{
    Vector3 ans;
    ans.x = lhs.x - rhs.x;
    ans.y = lhs.y - rhs.y;
    ans.z = lhs.z - rhs.z;
    return ans;
};

//    Vector4 = Vector4 - Vector4
Vector4 operator - (const Vector4 lhs, const Vector4 rhs)
{
    Vector4 ans;
    ans.w = lhs.w - rhs.w;
    ans.x = lhs.x - rhs.x;
    ans.y = lhs.y - rhs.y;
    ans.z = lhs.z - rhs.z;
    return ans;
};

//    Matrix2 = Matrix2 - Matrix2
Matrix2 operator - (const Matrix2 lhs, const Matrix2 rhs)
{
    int row;
    int col;
    Matrix2 ans;
    for (row = 0; row < 2; row++)
        for (col = 0; col < 2; col++)
            ans.a[row][col] = lhs.a[row][col] - rhs.a[row][col];
    return ans;
};

//    Matrix3 = Matrix3 - Matrix3
Matrix3 operator - (const Matrix3 lhs, const Matrix3 rhs)
{
    int row;
    int col;

```

```

    Matrix3 ans;
    for (row = 0; row < 3; row++)
        for (col = 0; col < 3; col++)
            ans.a[row][col] = lhs.a[row][col] - rhs.a[row][col];
    return ans;
};

// Matrix4 = Matrix4 - Matrix4
Matrix4 operator - (const Matrix4 lhs, const Matrix4 rhs)
{
    int row;
    int col;
    Matrix4 ans;
    for (row = 0; row < 4; row++)
        for (col = 0; col < 4; col++)
            ans.a[row][col] = lhs.a[row][col] - rhs.a[row][col];
    return ans;
};

//*****//
// Advance Addition, -=
//
//*****//
// Vector2 = Vector2 - Vector2
Vector2 operator -= (Vector2 &lhs, const Vector2 rhs)
{
    lhs.x -= rhs.x;
    lhs.y -= rhs.y;
    return lhs;
};

// Vector3 = Vector3 - Vector3
Vector3 operator -= (Vector3 &lhs, const Vector3 rhs)
{
    lhs.x -= rhs.x;
    lhs.y -= rhs.y;
    lhs.z -= rhs.z;
    return lhs;
};

// Vector4 = Vector4 - Vector4
Vector4 operator -= (Vector4 &lhs, const Vector4 rhs)
{
    lhs.w -= rhs.w;
    lhs.x -= rhs.x;
    lhs.y -= rhs.y;
    lhs.z -= rhs.z;
    return lhs;
};

// Matrix2 = Matrix2 - Matrix2
Matrix2 operator -= (Matrix2 &lhs, const Matrix2 rhs)
{
    int row;
    int col;
    for (row = 0; row < 2; row++)

```

```

        for (col = 0; col < 2; col++)
            lhs.a[row][col] -= rhs.a[row][col];
    return lhs;
};

// Matrix3 = Matrix3 - Matrix3
Matrix3 operator -= (Matrix3 &lhs, const Matrix3 rhs)
{
    int row;
    int col;
    for (row = 0; row < 3; row++)
        for (col = 0; col < 3; col++)
            lhs.a[row][col] -= rhs.a[row][col];
    return lhs;
};

// Matrix4 = Matrix4 - Matrix4
Matrix4 operator -= (Matrix4 &lhs, const Matrix4 rhs)
{
    int row;
    int col;
    for (row = 0; row < 4; row++)
        for (col = 0; col < 4; col++)
            lhs.a[row][col] -= rhs.a[row][col];
    return lhs;
};

//*****//
// Multiplication types *
//
// There should be no *= or /= with matrix math //
//*****//
// Base type,
// dot product 1xn * nx1 = 1x1
// ???          nx1 * 1xn = nxn
// std          nxn * nx1 = nx1
// std          1xn * nxn = 1xn
// std          nxn * nxn = nxn

/*
// Dot Product
double Dot(const Vector2 lhs, const Vector2 rhs)
{
    return (lhs.x*rhs.x) + (lhs.y*rhs.y);
};

// Dot Product
double Dot(const Vector3 lhs, const Vector3 rhs)
{
    return (lhs.x*rhs.x) + (lhs.y*rhs.y) + (lhs.z*rhs.z);
};

```

```

//    Dot Product
double Dot(const Vector4 lhs, const Vector4 rhs)
{
    return (lhs.w*rhs.w) + (lhs.x*rhs.x) + (lhs.y*rhs.y) +
    (lhs.z*rhs.z);
};
*/

//    Chris Adan
//    04 01 04
//    The dot product for v2.0
double Dot(const Vector4 a, const Vector4 b)
{
    return a.x*b.x + a.y*b.y + a.z*b.z + a.w*b.w;
}

double Dot(const Vector3 a, const Vector3 b)
{
    return a.x*b.x + a.y*b.y + a.z*b.z;
}

double Dot(const Vector2 a, const Vector2 b)
{
    return a.x*b.x + a.y*b.y;
}

//    Outer Product
Matrix2 operator * (const Vector2 lhs, const Vector2 rhs)
{
    //    a * c d = ac ad
    //    b          bc bd
    Matrix2 ans;
    ans.a[0][0] = lhs.x * rhs.x; ans.a[0][1] = lhs.x * rhs.y;
    ans.a[1][0] = lhs.y * rhs.x; ans.a[1][1] = lhs.y * rhs.y;
    return ans;
};

//    Outer Product
Matrix3 operator * (const Vector3 lhs, const Vector3 rhs)
{
    //    a * c d = ac ad
    //    b          bc bd
    Matrix3 ans;
    ans.a[0][0] = lhs.x * rhs.x; ans.a[0][1] = lhs.x * rhs.y;
    ans.a[0][2] = lhs.x * rhs.z;
    ans.a[1][0] = lhs.y * rhs.x; ans.a[1][1] = lhs.y * rhs.y;
    ans.a[1][2] = lhs.y * rhs.z;
    ans.a[2][0] = lhs.z * rhs.x; ans.a[2][1] = lhs.z * rhs.y;
    ans.a[2][2] = lhs.z * rhs.z;
    return ans;
};

//    Outer Product
Matrix4 operator * (const Vector4 lhs, const Vector4 rhs)
{
    //    a * c d = ac ad
    //    b          bc bd

```

```

Matrix4 ans;

ans.a[0][0] = lhs.w * rhs.w;  ans.a[2][0] = lhs.w * rhs.x;
ans.a[2][1] = lhs.w * rhs.y;  ans.a[2][2] = lhs.w * rhs.z;
ans.a[0][1] = lhs.x * rhs.w;  ans.a[0][0] = lhs.x * rhs.x;
ans.a[0][1] = lhs.x * rhs.y;  ans.a[0][2] = lhs.x * rhs.z;
ans.a[0][2] = lhs.y * rhs.w;  ans.a[1][0] = lhs.y * rhs.x;
ans.a[1][1] = lhs.y * rhs.y;  ans.a[1][2] = lhs.y * rhs.z;
ans.a[0][3] = lhs.z * rhs.w;  ans.a[2][0] = lhs.z * rhs.x;
ans.a[2][1] = lhs.z * rhs.y;  ans.a[2][2] = lhs.z * rhs.z;

return ans;
};

//          std          nxn * nx1 = nx1
Vector2 operator * (const Matrix2 lhs, const Vector2 rhs)
{
    Vector2 ans;
    ans.x = lhs.a[0][0] * rhs.x + lhs.a[0][1] * rhs.y;
    ans.y = lhs.a[1][0] * rhs.x + lhs.a[1][1] * rhs.y;
    return ans;
};

//          std          nxn * nx1 = nx1
Vector3 operator * (const Matrix3 lhs, const Vector3 rhs)
{
    Vector3 ans;
    ans.x = lhs.a[0][0] * rhs.x + lhs.a[0][1] * rhs.y + lhs.a[0][2] *
rhs.z;
    ans.y = lhs.a[1][0] * rhs.x + lhs.a[1][1] * rhs.y + lhs.a[1][2] *
rhs.z;
    ans.z = lhs.a[2][0] * rhs.x + lhs.a[2][1] * rhs.y + lhs.a[2][2] *
rhs.z;
    return ans;
};

//          std          nxn * nx1 = nx1
Vector4 operator * (const Matrix4 lhs, const Vector4 rhs)
{
    Vector4 ans;
    ans.w = lhs.a[0][0] * rhs.w + lhs.a[0][1] * rhs.x + lhs.a[0][2] *
rhs.y + lhs.a[0][3] * rhs.z;
    ans.x = lhs.a[1][0] * rhs.w + lhs.a[1][1] * rhs.x + lhs.a[1][2] *
rhs.y + lhs.a[1][3] * rhs.z;
    ans.y = lhs.a[2][0] * rhs.w + lhs.a[2][1] * rhs.x + lhs.a[2][2] *
rhs.y + lhs.a[2][3] * rhs.z;
    ans.z = lhs.a[3][0] * rhs.w + lhs.a[3][1] * rhs.x + lhs.a[3][2] *
rhs.y + lhs.a[3][3] * rhs.z;
    return ans;
};

//          std          1xn * nxn = 1xn
Vector2 operator * (const Vector2 lhs, const Matrix2 rhs)
{
    Vector2 ans;
    ans.x = lhs.x * rhs.a[0][0] + lhs.y * rhs.a[1][0];
    ans.y = lhs.x * rhs.a[0][1] + lhs.y * rhs.a[1][1];

```

```

        return ans;
};

//      std          1xn * nxn = 1xn
Vector3 operator * (const Vector3 lhs, const Matrix3 rhs)
{
    Vector3 ans;
    ans.x = lhs.x * rhs.a[0][0] + lhs.y * rhs.a[1][0] + lhs.z *
rhs.a[2][0];
    ans.y = lhs.x * rhs.a[0][1] + lhs.y * rhs.a[1][1] + lhs.z *
rhs.a[2][1];
    ans.z = lhs.x * rhs.a[0][2] + lhs.y * rhs.a[1][2] + lhs.z *
rhs.a[2][2];
    return ans;
};

//      std          1xn * nxn = 1xn
Vector4 operator * (const Vector4 lhs, const Matrix4 rhs)
{
    Vector4 ans;
    ans.w = lhs.w * rhs.a[0][0] + lhs.x * rhs.a[1][0] + lhs.y *
rhs.a[2][0] + lhs.z * rhs.a[3][0];
    ans.x = lhs.w * rhs.a[0][1] + lhs.x * rhs.a[1][1] + lhs.y *
rhs.a[2][1] + lhs.z * rhs.a[3][1];
    ans.y = lhs.w * rhs.a[0][2] + lhs.x * rhs.a[1][2] + lhs.y *
rhs.a[2][2] + lhs.z * rhs.a[3][2];
    ans.z = lhs.w * rhs.a[0][3] + lhs.x * rhs.a[1][3] + lhs.y *
rhs.a[2][3] + lhs.z * rhs.a[3][3];
    return ans;
};

//      std nxn * nxn = nxn
Matrix2 operator * (const Matrix2 lhs, const Matrix2 rhs)
{
    int size = 2;
    Matrix2 ans;
    int row;
    int col;
    int i;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
        {
            ans.a[row][col] = 0;
            for (i = 0; i < size; i++)
                ans.a[row][col] += lhs.a[row][i]*rhs.a[i][col];
        }
    return ans;
};

//      std nxn * nxn = nxn
Matrix3 operator * (const Matrix3 lhs, const Matrix3 rhs)
{
    int size = 3;
    Matrix3 ans;
    int row;
    int col;

```

```

    int i;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            {
                ans.a[row][col] = 0;
                for (i = 0; i < size; i++)
                    ans.a[row][col] += lhs.a[row][i]*rhs.a[i][col];
            }
    return ans;
};

// std nxn * nxn = nxn
Matrix4 operator * (const Matrix4 lhs, const Matrix4 rhs)
{
    int size = 4;
    Matrix4 ans;
    int row;
    int col;
    int i;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            {
                ans.a[row][col] = 0;
                for (i = 0; i < size; i++)
                    ans.a[row][col] += lhs.a[row][i]*rhs.a[i][col];
            }
    return ans;
};

// Simple scaling of a vector
Vector2 operator * (double lhs, const Vector2 rhs)
{
    Vector2 ans;
    ans.x = rhs.x * lhs;
    ans.y = rhs.y * lhs;
    return ans;
};
/*
// Simple scaling of a vector
Vector2 operator * (double lhs, const Vector2 rhs)
{
    Vector2 ans;
    ans.x = rhs.x * lhs;
    ans.y = rhs.y * lhs;
    return ans;
};
*/
// Simple scaling of a vector
Vector2 operator * (const Vector2 lhs, double rhs)
{
    Vector2 ans;
    ans.x = lhs.x * rhs;
    ans.y = lhs.y * rhs;
    return ans;
};

// Simple scaling of a vector

```

```

Vector3 operator * (double lhs, const Vector3 rhs)
{
    Vector3 ans;
    ans.x = rhs.x * lhs;
    ans.y = rhs.y * lhs;
    ans.z = rhs.z * lhs;
    return ans;
};

// Simple scaling of a vector
Vector3 operator * (const Vector3 lhs, double rhs)
{
    Vector3 ans;
    ans.x = lhs.x * rhs;
    ans.y = lhs.y * rhs;
    ans.z = lhs.z * rhs;
    return ans;
};

// Simple scaling of a vector
Vector4 operator * (double lhs, const Vector4 rhs)
{
    Vector4 ans;
    ans.w = rhs.w * lhs;
    ans.x = rhs.x * lhs;
    ans.y = rhs.y * lhs;
    ans.z = rhs.z * lhs;
    return ans;
};

// Simple scaling of a vector
Vector4 operator * (const Vector4 lhs, double rhs)
{
    Vector4 ans;
    ans.w = lhs.w * rhs;
    ans.x = lhs.x * rhs;
    ans.y = lhs.y * rhs;
    ans.z = lhs.z * rhs;
    return ans;
};

// Simple scaling of matrix
Matrix2 operator * (double lhs, const Matrix2 rhs)
{
    int size = 2;
    Matrix2 ans;
    int row;
    int col;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            ans.a[row][col] = rhs.a[row][col] * lhs;
    return ans;
}

// Simple scaling of matrix

```

```

Matrix2 operator * (const Matrix2 lhs, double rhs)
{
    int size = 2;
    Matrix2 ans;
    int row;
    int col;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            ans.a[row][col] = lhs.a[row][col] * rhs;
    return ans;
}

// Simple scaling of matrix
Matrix3 operator * (double lhs, const Matrix3 rhs)
{
    int size = 3;
    Matrix3 ans;
    int row;
    int col;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            ans.a[row][col] = rhs.a[row][col] * lhs;
    return ans;
}

// Simple scaling of matrix
Matrix3 operator * (const Matrix3 lhs, double rhs)
{
    int size = 3;
    Matrix3 ans;
    int row;
    int col;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            ans.a[row][col] = lhs.a[row][col] * rhs;
    return ans;
}

// Simple scaling of matrix
Matrix4 operator * (double lhs, const Matrix4 rhs)
{
    int size = 4;
    Matrix4 ans;
    int row;
    int col;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            ans.a[row][col] = rhs.a[row][col] * lhs;
    return ans;
}

// Simple scaling of matrix
Matrix4 operator * (const Matrix4 lhs, double rhs)
{
    int size = 4;
    Matrix4 ans;
    int row;

```

```

        int col;
        for (row = 0; row < size; row++)
            for (col = 0; col < size; col++)
                ans.a[row][col] = lhs.a[row][col] * rhs;
        return ans;
};

// Vector Scaling
Vector2 operator / (const Vector2 lhs, double rhs)
{
    Vector2 ans;
    ans.x = lhs.x / rhs;
    ans.y = lhs.y / rhs;
    return ans;
};

// Simple scaling of a vector
Vector3 operator / (const Vector3 lhs, double rhs)
{
    Vector3 ans;
    ans.x = lhs.x / rhs;
    ans.y = lhs.y / rhs;
    ans.z = lhs.z / rhs;
    return ans;
};

// Simple scaling of a vector
Vector4 operator / (const Vector4 lhs, double rhs)
{
    Vector4 ans;
    ans.w = lhs.w / rhs;
    ans.x = lhs.x / rhs;
    ans.y = lhs.y / rhs;
    ans.z = lhs.z / rhs;
    return ans;
};

// Vector Scaling
Vector2 operator /= (const Vector2 lhs, double rhs)
{
    Vector2 ans;
    ans.x = lhs.x / rhs;
    ans.y = lhs.y / rhs;
    return ans;
};

// Simple scaling of a vector
Vector3 operator /= (const Vector3 lhs, double rhs)
{
    Vector3 ans;
    ans.x = lhs.x / rhs;
    ans.y = lhs.y / rhs;
    ans.z = lhs.z / rhs;
    return ans;
};

// Simple scaling of a vector
Vector4 operator /= (const Vector4 lhs, double rhs)

```

```

{
    Vector4 ans;
    ans.w = lhs.w / rhs;
    ans.x = lhs.x / rhs;
    ans.y = lhs.y / rhs;
    ans.z = lhs.z / rhs;
    return ans;
};

// Simple scaling of matrix
Matrix2 operator / (const Matrix2 lhs, double rhs)
{
    int size = 2;
    Matrix2 ans;
    int row;
    int col;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            ans.a[row][col] = lhs.a[row][col] / rhs;
    return ans;
};

// Simple scaling of matrix
Matrix3 operator / (const Matrix3 lhs, double rhs)
{
    int size = 3;
    Matrix3 ans;
    int row;
    int col;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            ans.a[row][col] = lhs.a[row][col] / rhs;
    return ans;
};

// Simple scaling of matrix
Matrix4 operator / (const Matrix4 lhs, double rhs)
{
    int size = 4;
    Matrix4 ans;
    int row;
    int col;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            ans.a[row][col] = lhs.a[row][col] / rhs;
    return ans;
};

//=====//
// State Struct supporting functions //
//=====//

// Initilize a vector to zero
Vector2 Zero2 () {Vector2 a; a.x = 0; a.y = 0; return a;};
Vector3 Zero3 () {Vector3 a; a.x = 0; a.y = 0; a.z = 0; return a;};

```

```

Vector4 Zero4 () {Vector4 a; a.x = 0; a.y = 0; a.z = 0; a.w = 0;
    return a;};
/*
state_struct ZeroState ()
{
    state_struct a;
    a.Pos = Zero3();
    a.Vel = Zero3();
    a.Acc = Zero3();
    a.Force = Zero3();

    a.Theta = Zero3();
    a.Omega = Zero3();
    a.Alpha = Zero3();
    a.Moment= Zero3();

    a.Quat = Zero4();
    a.Quat.w = 1.0;
    a.QuatDot= Zero4();
    return a;
};
*/

//=====//
// Matrix Initiation functions //
//=====//
// Set up Identity matrix
Matrix2 Eye2()
{
    Matrix2 Answer;
    int row;
    int col;
    int size = 2;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            {
                if (col == row)
                    Answer.a[row][col] = 1;
                else
                    Answer.a[row][col] = 0;
            }
    return Answer;
};

Matrix3 Eye3()
{
    Matrix3 Answer;
    int row;
    int col;
    int size = 3;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            {
                if (col == row)
                    Answer.a[row][col] = 1;
                else
                    Answer.a[row][col] = 0;
            }
};

```

```

        }
        return Answer;
};

Matrix4 Eye4()
{
    Matrix4 Answer;
    int row;
    int col;
    int size = 4;
    for (row = 0; row < size; row++)
        for (col = 0; col < size; col++)
            {
                if (col == row)
                    Answer.a[row][col] = 1;
                else
                    Answer.a[row][col] = 0;
            }
    return Answer;
};

//=====//
//    Inv
//    //
//    Desc:
//    //
//    Inverses a matrix. This is an update to the //
//    Mike Kosman code to include Matrix Structures. //
//    Bugs:
//    //
//    * Doesn't handle singular matrixes //
//=====//
Matrix3 Inv(Matrix3 Matrix)
{
    double aDeterminate;
    Matrix3 ACofactor;
    Matrix3 AAdjoint;
    Matrix3 Answer;

    ACofactor = Cofactor(Matrix);

    aDeterminate = Determinate(Matrix, ACofactor);

    AAdjoint = Adjoint(ACofactor);

    Answer = AAdjoint / aDeterminate;

    return Answer;
};

Matrix3 Cofactor(Matrix3 aMatrix)
{
    Matrix3 ans;
    int size = 3;
    int row;

```

```

    int col;

    for(col=0; col < size; col++)
    {
        for(row=0;row < size; row++)
        {
            ans.a[row][col]= pow((double)-1, row + col + 2) *
Minor(row, col, aMatrix);
        }
    }
    return ans;
};

double Determinate(Matrix3 aMatrix, Matrix3 ACofactor)
{
    int size = 3;

    double Sum;
    Sum = 0.0;

    int col;
/*
NOTE: Ask Mike about this, I need a why answer
double BCofactor[1][3];

for(col = 0; col < size; col++)
{
    BCofactor[0][col]=ACofactor[0][col];
}
*/
for(col = 0; col<size; col++)
{
    Sum += aMatrix.a[0][col]*ACofactor.a[0][col];
}

    return Sum;
};

double Determinate(double a[2][2])
{
    return a[0][0]*a[1][1]-a[0][1]*a[1][0];
};

Matrix3 Adjoint(Matrix3 Matrix)
{
    Matrix3 Answer;
    int size=3;

    for(int j=0;j<size;j++)
    {
        for(int i=0;i<size;i++)
        {
            Answer.a[j][i]=Matrix.a[i][j];
        }
    }
    return Answer;
};

```

```

double Minor(int m, int n, Matrix3 Matrix)
{
    int size=3;
    int w;
    int z;
    double b[2][2];
    double ans;

    z=0; // b column start at 0

    for(int j=0;j<size;j++)
    {
        if(j!=n) // element is not of this column
        {
            w=0; // b row start at 0

            for(int i=0;i<size;i++)
            {
                if(i!=m) // element is not of this row
                {
                    b[w][z]=Matrix.a[i][j];
                    w++; // increment b row if this element
used
                }
            }

            z++; // increment b row if this column used, after
all rows filled
        }
    }

    ans=Determinate(b); // find the 2x2 Det

    return ans;
};

// Integration Methods

// ADAMS_BASHFORTH_MOULTON
// The sim's method
double Integrate_ADAM_BASHFORTH_MOULTON(double curr, double prev,
double deltaT)
{
    return (3.0*curr - prev)*0.5 * deltaT;
};

//
Vector2 Integrate_ADAM_BASHFORTH_MOULTON(Vector2 curr, Vector2 prev,
Vector2 old, double deltaT)
{
    // Before the Children of Kerensky returned
    // return Add(Multiply(Subtract(Multiply(curr, 3.0), prev),
deltaT/2.0), old);

    // After the apocolipse the children brought
    return ((3.0 * curr) - prev)* 0.5 * deltaT + old;
};

```

```

};
Vector3 Integrate_ADAM_BASHFORTH_MOULTON(Vector3 curr, Vector3 prev,
Vector3 old, double deltaT)
{
    //    Before the Children of Kerensky returned
    //    return Add(Multiply(Subtract(Multiply(curr, 3.0), prev),
deltaT/2.0), old);

    //    After the apocolipse the children brought
    return (3.0 * curr - prev)*0.5 * deltaT + old;

};
Vector4 Integrate_ADAM_BASHFORTH_MOULTON(Vector4 curr, Vector4 prev,
Vector4 old, double deltaT)
{
    //    Before the Children of Kerensky returned
    //    return Add(Multiply(Subtract(Multiply(curr, 3.0), prev),
deltaT/2.0), old);

    //    After the apocolipse the children brought
    return (3.0 * curr - prev)*0.5 * deltaT + old;

};

//    Returns the transformation matrix
//    Frame A contains Frame B
Matrix3 A_to_B(const Vector4 quat)
{
    //    These place holders are here for clarity.
    //    double e0, e1, e2, e3;

    //    e0 = cos(0.5 * psi) * cos(0.5 * theta) * cos(0.5 * phi) +
sin(0.5 * psi) * sin(0.5 * theta) * sin(0.5 * phi);
    //    e1 = cos(0.5 * psi) * cos(0.5 * theta) * sin(0.5 * phi) -
sin(0.5 * psi) * sin(0.5 * theta) * cos(0.5 * phi);
    //    e2 = cos(0.5 * psi) * sin(0.5 * theta) * cos(0.5 * phi) +
sin(0.5 * psi) * cos(0.5 * theta) * sin(0.5 * phi);
    //    e3 = -cos(0.5 * psi) * sin(0.5 * theta) * sin(0.5 * phi) +
sin(0.5 * psi) * cos(0.5 * theta) * cos(0.5 * phi);

    double l1, l2, l3, m1, m2, m3, n1, n2, n3;

    l1 = quat.w*quat.w + quat.x*quat.x - quat.y*quat.y -
quat.z*quat.z;
    l2 = 2*(quat.x * quat.y + quat.z * quat.w);
    l3 = 2*(quat.x * quat.z - quat.y * quat.w);

    m1 = 2*(quat.x * quat.y - quat.z * quat.w);
    m2 = quat.w*quat.w - quat.x*quat.x + quat.y*quat.y -
quat.z*quat.z;
    m3 = 2*(quat.z * quat.y + quat.x * quat.w);

    n1 = 2*(quat.w * quat.y + quat.z * quat.x);
    n2 = 2*(quat.z * quat.y - quat.x * quat.w);
    n3 = quat.w*quat.w - quat.x*quat.x - quat.y*quat.y +
quat.z*quat.z;

```

```

Matrix3 ans;
// See, nice and clear =)
ans.a[0][0] = l1; ans.a[0][1] = l2; ans.a[0][2] = l3;
ans.a[1][0] = m1; ans.a[1][1] = m2; ans.a[1][2] = m3;
ans.a[2][0] = n1; ans.a[2][1] = n2; ans.a[2][2] = n3;

return ans;
};

Matrix3 A_to_B(Vector3 Theta)
{
    Vector4 Quat = FindQuat(Theta);
    // LogInfo("IHateMatlab.html","Theta %f %f %f<br>Quaternions %f
%f %f %f",
    // Theta.x, Theta.y, Theta.z,
    // Quat.w, Quat.x, Quat.y, Quat.z);
    return A_to_B(Quat);
};

Matrix3 B_to_A (const Vector4 quat)
{
    return Inv(A_to_B(quat));
};

Matrix3 B_to_A (const Vector3 Theta)
{
    return Inv(A_to_B(Theta));
};

// Finds the quaternions given phi, theta, psi
// From "Flight Simulation", Cambridge Series of Aerospace
Vector4 FindQuat(Vector3 Theta)
{
    Vector4 Quat;
    Quat.w = cos(0.5 * Theta.z) * cos(0.5 * Theta.y) * cos(0.5 *
Theta.x) + sin(0.5 * Theta.z) * sin(0.5 * Theta.y) * sin(0.5 *
Theta.x);
    Quat.x = cos(0.5 * Theta.z) * cos(0.5 * Theta.y) * sin(0.5 *
Theta.x) - sin(0.5 * Theta.z) * sin(0.5 * Theta.y) * cos(0.5 *
Theta.x);
    Quat.y = cos(0.5 * Theta.z) * sin(0.5 * Theta.y) * cos(0.5 *
Theta.x) + sin(0.5 * Theta.z) * cos(0.5 * Theta.y) * sin(0.5 *
Theta.x);
    Quat.z = -cos(0.5 * Theta.z) * sin(0.5 * Theta.y) * sin(0.5 *
Theta.x) + sin(0.5 * Theta.z) * cos(0.5 * Theta.y) * cos(0.5 *
Theta.x);
    return Quat;
};

double Mag(Vector3 Q)
{
    return sqrt(Q.x*Q.x + Q.y*Q.y + Q.z*Q.z);
};

double Mag(Vector4 Q)
{

```

```

    return sqrt(Q.w*Q.w + Q.x*Q.x + Q.y*Q.y + Q.z*Q.z);
};

// From rolph and staples
Vector3 Quat_To_Euler(Vector4 quat)
{
    double l1, l2, l3, m1, m2, m3, n1, n2, n3;

    quat = quat / Mag(quat);
    l1 = quat.w*quat.w + quat.x*quat.x - quat.y*quat.y -
quat.z*quat.z;
    l2 = 2*(quat.x * quat.y + quat.z * quat.w);
    l3 = 2*(quat.x * quat.z - quat.y * quat.w);

    m1 = 2*(quat.x * quat.y - quat.z * quat.w);
    m2 = quat.w*quat.w - quat.x*quat.x + quat.y*quat.y -
quat.z*quat.z;
    m3 = 2*(quat.z * quat.y + quat.x * quat.w);

    n1 = 2*(quat.w * quat.y + quat.z * quat.x);
    n2 = 2*(quat.z * quat.y - quat.x * quat.w);
    n3 = quat.w*quat.w - quat.x*quat.x - quat.y*quat.y +
quat.z*quat.z;

    Vector3 Theta;

    // Arcsin/Arccos check
    if (l3*l3 >= 1.0)
        Theta.y = 0.0;
    else
        Theta.y = asin(-l3);

    double sign_l2;
    if (l2<0)
        sign_l2 = -1;
    else
        sign_l2 = 1;

    // Arcsin/Arccos check
    double coef = l1/cos(Theta.y);
    if (coef*coef >= 1.0)
        Theta.z = 0;
    else
        Theta.z = acos(coef)*sign_l2;

    double sign_m3;
    if (m3<0)
        sign_m3 = -1;
    else
        sign_m3 = 1;

    // Arcsin/Arccos check
    coef = n3/cos(Theta.y);

```

```

    if (coef*coef>=1)
        Theta.x = 0;
    else
        Theta.x = acos(n3/cos(Theta.y))*sign_m3;

    return Theta;
};

// Chris Adan
// 04 01 04
// Cross Product, for 3x1 vectors.
// Vector a x Vector b
Vector3 Cross(const Vector3 a, const Vector3 b)
{
    // I J      K
    //  a0  a1  a2
    //  b0  b1  b2
    // I(a1 * b2 - a2*b1)  J(a2 * b0 - a0 * b2)  K(a0 * b1 - a1 * b0)
    Vector3 ans;

    ans.x = a.y * b.z - a.z * b.y;
    ans.y = a.z * b.x - a.x * b.z;
    ans.z = a.x * b.y - a.y * b.x;

    return ans;
};

Vector3 XYZ2LLA(Vector3 simXYZ, Vector3 InitialLLA)
{
    /*****
    * - Simulator World Coordinates -- *
    *
    *          X: + North          - South          *
    *          Y: + East           - West           *
    *          Z: + Down           - Up             *
    *
    *
    *****/
    Vector3 vLLA;

    double latitude_Initial = InitialLLA.x;
    double longitude_Initial = InitialLLA.y;
    double LAT_FT_PER_DEG, LONG_FT_PER_DEG;

    // Find the size of one degree of Latitude
    LAT_FT_PER_DEG = MERIDIAN_RADIUS * M_2_FT * PI / 180;

    // Find the current Latitude
    vLLA.x = latitude_Initial + (simXYZ.x)/LAT_FT_PER_DEG; //
    Find the current latitude of the aircraft

    // Find the size of one degree of longitude (which changes with
    latitude)
    LONG_FT_PER_DEG = EQUATORIAL_RADIUS*cos((InitialLLA.x)*PI/180)*
    M_2_FT * PI / 180;

    // Find the current Longitude of the aircraft

```

```

vLLA.y = longitude_Initial + (simXYZ.y)/LONG_FT_PER_DEG;

// Find the current altitude of the aircraft
vLLA.z = -(simXYZ.z); // Export Altitude of Aircraft}

return vLLA;
}

double Sgn(double number)
{
    // Returns the sign of a number
    if(number>0.0)
        return 1.0;
    else if(number<0.0)
        return -1.0;
    else
        return 0.0;
} // End of Sgn Function

Vector4 Quat_Product(const Vector4 q1, const Vector4 q2)
{
    Vector4 ans;

    ans.w = q1.w*q2.w - q1.x*q2.x - q1.y*q2.y - q1.z*q2.z;
    ans.x = q1.w*q2.x + q1.x*q2.w + q1.y*q2.z - q1.z*q2.y;
    ans.y = q1.w*q2.y - q1.x*q2.z + q1.y*q2.w + q1.z*q2.x;
    ans.z = q1.w*q2.z - q1.x*q2.y - q1.y*q2.x + q1.z*q2.w;

    return ans;
}

Vector4 MakeUnitVector(Vector4 Quat)
{
    return Quat / Mag(Quat);
};

Vector3 MakeUnitVector(Vector3 vector)
{
    return vector / Mag(vector);
};

// From AERO 551 and a GPS report (PDF)
// LLA.x = Latitude
// LLA.y = Longitude
// LLA.z = Altitude
Vector3 LLA_to_ECEF (Vector3 LLA)
{
    Vector3 Pos;
    double f, a, b, e, eprime, Radius_Curvature;

    Vector3 LLA_Metric;
    LLA_Metric.x = LLA.x * PI/180; // The degrees to
radians
    LLA_Metric.y = LLA.y * PI/180; // The degrees to
radians

```

```

    LLA_Metric.z = LLA.z / 3.2808399;    //    The feets to meters

    // Data Concerning the WGS84 Ellipsoid
    f = 1/(298.257223563); //    [unitless]    Curvature factor
    a = 6378137; //    [meters]    Radius of the
earth at the equator
    b = a*(1-f); //    [meters]    Radius of the
earth in from center to north/south pole
    e = sqrt((a*a-b*b)/(a*a)) ; //    Ecentricity about
one axis
    eprime = sqrt((a*a-b*b)/(b*b)); //    Ecentricity about
the other axis

    Radius_Curvature = a/sqrt(1-
e*e*sin(LLA_Metric.x)*sin(LLA_Metric.x));

    Pos.x = (Radius_Curvature +
LLA_Metric.z)*cos(LLA_Metric.x)*cos(LLA_Metric.y);
    Pos.y = (Radius_Curvature +
LLA_Metric.z)*cos(LLA_Metric.x)*sin(LLA_Metric.y);
    Pos.z = (b*b/(a*a)*Radius_Curvature +
LLA_Metric.z)*sin(LLA_Metric.x);

    //    Now convert from meters to feet
    Pos = Pos * 3.2808399;

    return Pos;
}

//    From AERO 551 and a GPS report (PDF)
Vector3 ECEF_to_LLA (Vector3 Pos)
{
    //LogInfo("Physics_Log.html","ECEF_to_LLA");
    //LogInfo("Physics_Log.html","Pos [%f %f %f]",Pos.x, Pos.y,
Pos.z);

    Vector3 LLA;
    double a, b, f, e, e_prime, p, theta, N, X, Y, Z;

    //    Go from feet to meters
    X = Pos.x/ 3.2808399;
    Y = Pos.y/ 3.2808399;
    Z = Pos.z/ 3.2808399;
    //LogInfo("Physics_Log.html","X %f" ,X);
    //LogInfo("Physics_Log.html","Y %f" ,Y);
    //LogInfo("Physics_Log.html","Z %f" ,Z);

    // Data Concerning the WGS84 Ellipsoid
    a = 6378137.000; //    Semi-Major Axis: the equatorial
earth radius
    b = 6356752.314; //    Semi-Minor Axis: the polar earth
radius % Also: b = a*(1-f)
    f = 1/298.257223563; //    Ellipsoid flattening parameter

```

```

    e = sqrt((a*a-b*b)/(a*a));          //    First Eccentricity e
for the earth
    //LogInfo("Physics_Log.html","E = %f",e);

    e_prime = sqrt((a*a-b*b)/(b*b));    //    Second Eccentricity e'
for the earth
    //LogInfo("Physics_Log.html","E prime = %f",e_prime);

    p = sqrt(X*X+Y*Y);                  //    Supplementary
value, Radius
    //LogInfo("Physics_Log.html","p = %f",p);

    theta = atan2(Z*a,p*b);             //    Supplementary value,
Angle
    //LogInfo("Physics_Log.html","theta = %f",theta);
//=====
===//
    //    Closed Form Solution For LLA: Latitude, Longitude, And
Altitude    //
    //=====
===//

    LLA.y = atan2(Y,X);                  // Calculate Longitude

    LLA.x = atan2(
        Z+e_prime*e_prime*b*pow(sin(theta),3.0),
        p-e*e*a*pow(cos(theta),3.0)); // Calculate Latitude

    N = a/sqrt(1-e*e*sin(LLA.x)*sin(LLA.x)); //    Radius of
Curvature
    LLA.z = p/cos(LLA.x)-N;              //    Calculate Altitude

    //    Go from metric to english
    LLA.x *= 180/PI; //    Go to degrees
    LLA.y *= 180/PI; //    Go to degrees
    LLA.z *= 3.2808399; //    Go to feet

    return LLA;
};

/*
Vector3 ECEF_to_LLA_Rotate_Earth(Vector3 Pos, double theta)
{
    Matrix3 DCM_I_to_Rotate;

    double c = cos(theta);
    double s = sin(theta);

    DCM_I_to_Rotate.a[0][0] = c; DCM_I_to_Rotate.a[0][1] = s;
    DCM_I_to_Rotate.a[0][2] = 0;
    DCM_I_to_Rotate.a[1][0] = -s; DCM_I_to_Rotate.a[1][1] = c;
    DCM_I_to_Rotate.a[1][2] = 0;
    DCM_I_to_Rotate.a[2][0] = 0; DCM_I_to_Rotate.a[2][1] = 0;
    DCM_I_to_Rotate.a[2][2] = 1;

    Vector3 Pos_after;

```

```

    Pos_after = DCM_I_to_Rotate * Pos;

    //printf("Pos = %f %f %f\n",Pos.x, Pos.y, Pos.z);
    //printf("Pos_After = %f %f %f\n",Pos_after.x, Pos_after.y,
Pos_after.z);

    return ECEF_to_LLA(Pos_after);
}
*/

```

```

Matrix4 FindQuatDotMatrix(Vector3 Omega)

```

```

{
    Matrix4    Temp;

    // From Book, Flight Simulation, Rolfe and Staples
    Temp.a[0][0] = 0;           Temp.a [1][0] = -Omega.x/2.0;
    Temp.a[2][0] = -Omega.y/2.0; Temp.a[3][0] = -Omega.z/2.0;
    Temp.a[0][1] = Omega.x/2.0; Temp.a [1][1] = 0;
    Temp.a[2][1] = Omega.z/2.0; Temp.a[3][1] = -Omega.y/2.0;
    Temp.a[0][2] = Omega.y/2.0; Temp.a [1][2] = -Omega.z/2.0;
    Temp.a[2][2] = 0;           Temp.a[3][2] =
Omega.x/2.0;
    Temp.a[0][3] = Omega.z/2.0; Temp.a [1][3] = Omega.y/2.0;
    Temp.a[2][3] = -Omega.x/2.0; Temp.a[3][3] = 0;

    //    Magic reverser
    //    DEADLY_BUG_0
    for (int row = 0; row < 4; row++)
        for (int col = 0; col < 4; col++)
            Temp.a[row][col] = Temp.a[row][col]*-1;

    return Temp;
};

```

```

Vector3 Dismantle_A_to_B(Matrix3 LMN)

```

```

{
    Vector3 Theta;
    double L1, L2, L3;
    double M1, M2, M3;
    double N1, N2, N3;
    double pitch, yaw, roll;
    double sign_L2, sign_M3;
    L1 = LMN.a[0][0]; L2 = LMN.a[0][1]; L3 = LMN.a[0][2];
    M1 = LMN.a[1][0]; M2 = LMN.a[1][1]; M3 = LMN.a[1][2];
    N1 = LMN.a[2][0]; N2 = LMN.a[2][1]; N3 = LMN.a[2][2];

    //    Check for L3 values that will cause
    //    imaginary arc sines
    if (L3 > 1)
        L3 = 1;
    if (L3 < -1)
        L3 = -1;
    pitch = asin(-L3);

    if (L2<0)

```

```

        sign_L2 = -1;
else
    sign_L2 = 1;

L1 /= cos(pitch);

//    Check for L1 values that will cause
//    imaginary arc cosines
if (L1 > 1)
    L1 = 1;

if (L1 < -1)
    L1 = -1;

yaw = acos(L1)*sign_L2;

if (M3<0)
    sign_M3 = -1;
else
    sign_M3 = 1;

N3 /= cos(pitch);

//    Check for N3 values that will cause
//    imaginary arc cosines
if (N3 > 1)
    N3 = 1;
if (N3 < -1)
    N3 = -1;

roll = acos(N3)*sign_M3;

Theta.x = roll;
Theta.y = pitch;
Theta.z = yaw;
return Theta;
};

Vector3 Dismantle_B_to_A(Matrix3 NML)
{
    Vector3 Theta;
    Matrix3 LMN;
    LMN = Inv(NML);
    Theta = Dismantle_A_to_B(LMN);
    return Theta;
};

bool IsNaN(Vector2 a)
{
    if(!(a.x == a.x) || !(a.y == a.y))
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

```

    }
};

bool IsNaN(Vector3 a)
{
    if(!(a.x == a.x) || !(a.y == a.y) || !(a.z == a.z))
    {
        return true;
    }
    else
    {
        return false;
    }
};

bool IsNaN(Vector4 a)
{
    if(!(a.w == a.w) || !(a.x == a.x) || !(a.y == a.y) || !(a.z ==
a.z))
    {
        return true;
    }
    else
    {
        return false;
    }
};

```

APPENDIX B - lost_physics_0015

The physics class is contained in the lost_physics_0015 files. This class keeps track of the forces, moments, velocity, position and orientation of the vehicle. There is a state associated with each frame of reference. The three frames of reference are Body, Earth Fixed, and Local vertical local tangential.

ODE45

An ODE45 integrator was implemented into the physics class [5]. Some significant modifications had to be made from the version in the reference. This enables accurate integration of the state variables. It's assumed that the forces and moments are constant over the given time step, and each time step is broken down into 10 sub steps. This enables some of the non-linearities (mass flow rate for example) of 6 dimensional flight to surface. In previous versions the ode solver could be set to iterate until a desired error tolerance was reached. This sometimes caused problems, as the solver would frequently break down time steps into the nano-second range. It would be impossible to maintain realtime flight with the error tolerance forcing, and thus the fixed step breakdown is used.

North-East-Down Frame

All forces are processed in the inertial frame, and all moments are processed in the body frame. The inertial position and velocity is then transformed to the body frame. As a last step, the Euler angles are found for the local vertical local tangential frame for the graphics engine to display the model.

Step 1: Take the position vector of the aircraft in the ECEF frame. The down vector is equal to the negative of the position vector.

Step 2: To find east a vector is needed to be slightly north. Down Cross North will give East. Add one to the Z component of down to get a slightly more north vector.

Step 3: Make a better north vector by crossing East with Down.

Step 4: Convert NED vectors to unit vectors.

Step 5: Direction cosign matrixes are essentially unit vectors dotted with each other. Using Rolfe and Staples, produce the LMN matrix components with the following pairs.

L1 = North * Body X Unit Vector
L2 = East * Body X Unit Vector
L3 = Down * Body X Unit Vector

M1 = North * Body Y Unit Vector
M2 = East * Body Y Unit Vector

M3 = Down * Body Y Unit Vector

N1 = North * Body Z Unit Vector

N2 = East * Body Z Unit Vector

N3 = Down * Body Z Unit Vector

Step 6: Use the rolfe and staple procedure to extract phi, theta and psi from the matrix components.

Euler Angles

Phi, Theta and Psi are needed by the graphics engine and standard control system loops. Phi is the roll angle about the X axis, Theta is the pitch angle about the Y axis and Yaw is the rotation angle about the Z axis. Due to the way the rotations are performed, when Pitch is +/- 90 degrees Yaw angle is undefined.

Quaternions

The orientation between the body frame and the inertial frame is handled with quaternions. Quaternions have four components. The first component, W, is the real component. The other three components are imaginary, X, Y, Z. Each of the imaginaries can be thought of as a rotation about that axis. The magnitude of the quaternion vector should always be one.

Mass Properties

The physics class needs two sets of mass properties. The first of which is mass. Since the flight simulator is in English units, mass must be in slugs. It's possible to send a mass dot term. This term is best used to represent fuel flow.

Inertia is the second set of mass properties. The physics class does not care if the matrix is diagonal, or a full 3x3. The units of inertia is slugs ft².

Code – H File

```
//=====//  
//   Physics Handling System   //  
//=====//  
#ifndef PHYSICS_H  
#define PHYSICS_H  
#include "MatrixMath_4_0.h"  
  
//   Number of equations to integrate  
//   1-3 = moments  
//   4-7 = quats  
//   8-10 = Forces  
//   11-13 = velocity  
//   14 = fuel flow  
static const int N_MAX = 14;
```

```

//static double ode[N_MAX];

//static double xn_1[N_MAX];
//static double *temp;

static bool debug = false;

//    for managing a state in time
class CState_Struct
{
public:
    Vector3 Force;
    Vector3 Acc;
    Vector3 Vel;
    Vector3 Pos;

    Vector3 Moment;
    Vector3 Alpha;
    Vector3 Omega;

    Vector4 Quat_Dot;
    Vector4 Quat;
};

//    For managing the frame of refrence
class CFrame_Struct
{
public:
    // CState_Struct Prev;
    CState_Struct Curr;
};

//=====//
//    This is inteded to be the ultimate dynamic plant.           //
//    Throw it a force or moment in any frame,                   //
//    and it should be able to transform and integrate it.      //
//=====//
class CPhysics
{
public:
    //    Default constructor
    //    Very dangerouse to use
    CPhysics();

    //    Derivative function
    void derivative(double t, double x[N_MAX], double ode[N_MAX]);
    void RK45(double xn[N_MAX], double h, double t, double
xn_1[N_MAX]);
    void ODE45(double Tinitial, double deltaT, double Tfinal, double*
IC, double* data);

    //=====//
    //    Set Commands      //
    //=====//
    //    Changes the mass of the object, returns true if it works

```

```

bool SetMass(double newMass);
bool SetMassDot(double newMassDot);

//=====//
//   Inertia Sets      //
//       Begin        //
//=====//
//   Changes the Ixx Inertia value, true if it works
bool SetIxx(double Ixx);

//   Changes the Iyy Inertia value, true if it works
bool SetIyy(double Iyy);

//   Changes the Izz Inertia value, true if it works
bool SetIzz(double Izz);

//   Set the products of inertia, true if works
bool SetIxy (double Ixy);

//   Set the products of inertia, true if works
bool SetIyz (double Iyz);

//   Set the products of inertia, true if works
bool SetIxz (double Ixz);

//   Sets the interia matrix
bool SetInertia(double * I);

bool SetInertiaInverse();
//=====//
//   Inertia Sets      //
//       End          //
//=====//

//=====//
//   Initial Condition Sets //
//       Begin          //
//=====//
//   Initlizes intial conditions for integration
//   Returns true if successful
//   Force and moment aren't initial conditions,
//   they're applied to the model during run time.
void SetInitials(Vector3 LLA, Vector3 Theta, Vector3 Vel, Vector3
Omega);

bool SetPos(Vector3 v3Pos);

bool SetRotationOffset(Vector3 v3Vel);

bool SetPos(double dx, double dy, double dz);

bool SetVel(Vector3 v3Vel);
bool SetBodyVel(Vector3 v3Vel);

bool SetAcc(Vector3 v3Acc);

bool SetTheta(Vector3 v3Theta);

```

```

bool SetOmega(Vector3 v3Omega);

bool SetAlpha(Vector3 v3Alpha);

bool SetQuat(Vector4 vQuat);
//=====//
//   Initial Condition Sets //
//           End //
//=====//

//=====//
//   Run Time Functions //
//           Begin //
//=====//
//   The external model should call on these frequently

//   Call this function at the begining of the time step!!!
//   It sets the moment and forces to zero!!!
bool InitilizeTimeStep();

//=====//
//   Force adding functions //
//=====//
//   This function adds forces in the body frame
//   All math is done in the body frame anyway.
bool Add_Body_Force (Vector3 v3BodyForce);

//   Transforms inertial frame forces too a body force.
bool Add_Inertial_Force (Vector3 v3InertialForce);

//=====//
//   Moment adding functions //
//=====//
//   This function adds Moments in the body frame
//   All math is done in the body frame anyway.
bool Add_Body_Moment (Vector3 v3BodyMoment);

//   Transforms inertial frame Moment too a body Moment.
bool Add_Inertial_Moment (Vector3 v3InertialMoment);

//=====//
//   This is the real work horse, Integration //
//   Provides a new set of pos, and vels, //
//   based on forces and moments. //
//
//
//   MicrosoPIC deltaT's work good, say 1/1000. //
//=====//
bool IntegrateAll(double deltaT);

//=====//
//   Get Functions //
//=====//
Vector3 GetPos();
Vector3 GetVel();

```

```

Vector3 GetBodyVel();
Vector3 GetBodyAcc();
Vector3 GetAcc();
Vector4 GetQuat();
Vector3 GetOmega();
Vector3 GetTheta();

Vector3 GetXUnitVector();
Vector3 GetYUnitVector();
Vector3 GetZUnitVector();

Vector3 GetLLA();

double GetMass();

private:

//    this guy is neeto
//double ode[N_MAX];
double mdot;

//=====//
//    Required variables      //
//=====//
//    States for integrating
//    All in metric

//    All data is handled in inertial
CFrame_Struct Inertial;
CFrame_Struct Body;
CFrame_Struct Lhlt;

//    in kg
double Mass;

//    Inertia matrix, far from princible axis
Matrix3 Inertia, Inertia_Inverse;

//=====//
//    Intermediate variables      //
//=====//

//    Quaternion rate matrix
Matrix4 QuatRateMatrix;

double time;

};

#endif //ifndef PHYSICS_H

```

Code – CPP File

```
#include "lost_physics_0015.h"

// This derivative is integrated by the solver
void CPhysics::derivative(double t, double x[N_MAX], double ode[N_MAX])
{
    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);
    Matrix3 DCM_B_to_I = B_to_A(Body.Curr.Quat);
    Matrix3 DCM_I_to_B = A_to_B(Body.Curr.Quat);

    // Omega dot equation
    Vector3 Alpha;
    Alpha = Inertia_Inverse*(Body.Curr.Moment -
Cross(Body.Curr.Omega, Inertia*Body.Curr.Omega));

    ode[0] = Alpha.x;
    ode[1] = Alpha.y;
    ode[2] = Alpha.z;

    // Quaternion dot equations
    QuatRateMatrix = FindQuatDotMatrix(Body.Curr.Omega);
    Body.Curr.Quat_Dot = QuatRateMatrix * Body.Curr.Quat;

    ode[3] = Body.Curr.Quat_Dot.x;
    ode[4] = Body.Curr.Quat_Dot.y;
    ode[5] = Body.Curr.Quat_Dot.z;
    ode[6] = Body.Curr.Quat_Dot.w;

    Body.Curr.Acc = Body.Curr.Force/Mass;
    Inertial.Curr.Acc = DCM_B_to_I * Body.Curr.Acc;

    // integtate acc to get vel
    ode[7] = Inertial.Curr.Acc.x;
    ode[8] = Inertial.Curr.Acc.y;
    ode[9] = Inertial.Curr.Acc.z;

    // integrate vel to get pos
    ode[10] = x[7];
    ode[11] = x[8];
    ode[12] = x[9];

    ode[13] = mdot;

    //return ode;
};

// runge-kutta solver
// THE solution for solving equations with unknown properties
// Xn = values at begining of state
// h = step size (deltaT) of this segment
// t = time, for things like sin(t)
// Xn_1 = end of state value
void CPhysics::RK45(double xn[N_MAX], double h, double t, double
xn_1[N_MAX])
{
```

```

double k1[N_MAX];
double k2[N_MAX];
double k3[N_MAX];
double k4[N_MAX];
double xn2[N_MAX];
double temp[N_MAX];

int i;

derivative(t, xn, temp);
for (i = 0; i < N_MAX; i++)
{
    k1[i] = temp[i]*h;
    xn2[i] = xn[i] + k1[i]/2.0;
}

derivative(t+h/2.0, xn2, temp);
for (i = 0; i < N_MAX; i++)
{
    k2[i] = temp[i]*h;
    xn2[i] = xn[i] + k2[i]/2.0;
}

derivative(t+h/2.0, xn2, temp);
for (i = 0; i < N_MAX; i++)
{
    k3[i] = temp[i]*h;
    xn2[i] = xn[i] + k3[i];
}

derivative(t+h, xn2, temp);
for (i = 0; i < N_MAX; i++)
{
    k4[i] = temp[i]*h;
}

for (i = 0; i < N_MAX; i++)
{
    xn_1[i] = xn[i] + (k1[i] + 2.0*k2[i] + 2.0*k3[i]
+k4[i])/6.0;
};

// Simple ode solver, solves for one time increment.
// Reduces that one time increment into 10 sub increments.
void CPhysics::ODE45(double Tinitial, double deltaT, double Tfinal,
double* IC, double* data)
{
    double dt2 = deltaT;
    double time = Tinitial;

    double data_temp_0[N_MAX];
    double data_temp_2[N_MAX];

    double i;

```

```

int k;

// Now we break down that first guess
dt2 = deltaT/10;

// Copy the initial conditions into the data array
for (k = 0; k < N_MAX; k++)
{
    data[k] = IC[k];
    data_temp_2[k] = IC[k];
}

// for the time of the simulation
for (i = 0; i < deltaT; i = i + dt2)
{
    // Solve for the ode by running through the data
    RK45(&data_temp_2[0], dt2, Tinitial+i, data_temp_0);
    for (k = 0; k < N_MAX; k++)
    {
        data_temp_2[k] = data_temp_0[k];
    }

    // update the data for the next time step
    for (k = 0; k < N_MAX; k++)
    {
        data[k] = data_temp_0[k];
    };

// return void;
};

// Default constructor
// Very dangerous to use
CPhysics::CPhysics()
{
    Vector3 v3Theta;
    v3Theta.x = 0;
    v3Theta.y = 0;
    v3Theta.z = 0;

    SetTheta(v3Theta);

    time = 0;

    // Default mass
    Mass = 1;

    // Default Inertias
    Inertia.a[0][0] = 1;Inertia.a[0][1] = 0;Inertia.a[0][2] = 0;
    Inertia.a[1][0] = 0;Inertia.a[1][1] = 1;Inertia.a[1][2] = 0;

```

```

        Inertia.a[2][0] = 0;Inertia.a[2][1] = 0;Inertia.a[2][2] = 1;

        SetInertiaInverse();
};

//=====//
//    Set Commands    //
//=====//
//    Changes the mass of the object, returns true if it works
bool CPhysics::SetMass(double newMass)
{
    if (newMass > 0)
    {
        Mass = newMass;
        return true;
    }
    else
    {
        Mass = 1;
        return false;
    }
};

bool CPhysics::SetMassDot(double newMassDot)
{
    mdot = newMassDot;
    return true;
}

//=====//
//    Inertia Sets    //
//    Begin          //
//=====//
//    Changes the Ixx Inertia value, true if it works
bool CPhysics::SetIxx(double Ixx)
{
    if (Ixx > 0)
    {
        Inertia.a[0][0] = Ixx;
        return true;
    }
    else
    {
        Inertia.a[0][0] = 1;
        return false;
    }
};

//    Changes the Iyy Inertia value, true if it works
bool CPhysics::SetIyy(double Iyy)
{
    if (Iyy > 0)
    {
        Inertia.a[1][1] = Iyy;
        return true;
    }
    else

```

```

        {
            Inertia.a[1][1] = 1;
            return false;
        };
};

// Changes the Izz Inertia value, true if it works
bool CPhysics::SetIzz(double Izz)
{
    if (Izz > 0)
    {
        Inertia.a[2][2] = Izz;
        return true;
    }
    else
    {
        Inertia.a[2][2] = 1;
        return false;
    };
};

// Set the products of inertia, true if works
bool CPhysics::SetIxy (double Ixy)
{
    Inertia.a[0][1] = Ixy;
    Inertia.a[1][0] = Ixy;
    return true;
};

// Set the products of inertia, true if works
bool CPhysics::SetIyz (double Iyz)
{
    Inertia.a[1][2] = Iyz;
    Inertia.a[2][1] = Iyz;
    return true;
};

// Set the products of inertia, true if works
bool CPhysics::SetIxz (double Ixz)
{
    Inertia.a[0][2] = Ixz;
    Inertia.a[2][0] = Ixz;
    return true;
};

bool CPhysics::SetInertia(double * I)
{
    SetIxx(I[0]);
    SetIyy(I[1]);
    SetIzz(I[2]);
    SetIxy(I[3]);
    SetIxz(I[4]);
    SetIyz(I[5]);
    SetInertiaInverse();
    return true;
}

```

```

bool CPhysics::SetInertiaInverse()
{
    Inertia_Inverse = Inv(Inertia);
    /*//LogInfo("IHateMatlab.html","Inv<br>%f, %f, %f<br>%f, %f,
%f<br>%f, %f, %f",
        Inertia.a[0][0],Inertia.a[0][1],Inertia.a[0][2],
        Inertia.a[1][0],Inertia.a[1][1],Inertia.a[1][2],
        Inertia.a[2][0],Inertia.a[2][1],Inertia.a[2][2]);
    //LogInfo("IHateMatlab.html","Inv<br>%f, %f, %f<br>%f, %f,
%f<br>%f, %f, %f",

        Inertia_Inverse.a[0][0],Inertia_Inverse.a[0][1],Inertia_Inverse.a
[0][2],

        Inertia_Inverse.a[1][0],Inertia_Inverse.a[1][1],Inertia_Inverse.a
[1][2],

        Inertia_Inverse.a[2][0],Inertia_Inverse.a[2][1],Inertia_Inverse.a
[2][2]);*/
    return true;
}
//=====//
//    Inertia Sets        //
//        End                //
//=====//

//=====//
//    Initial Condition Sets //
//        Begin                //
//=====//
//    Initlizes intial conditions for integration
//    Returns true if successful
//    Force and moment aren't initial conditions,
//    they're applied to the model during run time.
bool CPhysics::SetPos(Vector3 v3Pos)
{
    //    Body.Prev.Pos = v3Pos;
    Body.Curr.Pos = v3Pos;
    Inertial.Curr.Pos = v3Pos;
    //    Inertial.Prev.Pos = v3Pos;
    return true;
};

bool CPhysics::SetRotationOffset(Vector3 v3Vel)
{
    Inertial.Curr.Vel += v3Vel;
    //    Inertial.Prev.Vel += v3Vel;
    return true;
};

bool CPhysics::SetPos(double dx, double dy, double dz)
{
    Vector3 Temp;
    Temp.x =dx;
    Temp.y =dy;
    Temp.z =dz;
    SetPos(Temp);
}

```

```

        return true;
};

bool CPhysics::SetVel(Vector3 v3Vel)
{
    //Body.Prev.Vel = v3Vel;
    // Body.Curr.Vel = v3Vel;
    Inertial.Curr.Vel = v3Vel;
    // Inertial.Prev.Vel = v3Vel;
    return true;
};

bool CPhysics::SetBodyVel(Vector3 v3Vel)
{
    //Body.Prev.Vel = v3Vel;
    // Body.Curr.Vel = v3Vel;

    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);
    Matrix3 DCM_B_to_I = B_to_A(Body.Curr.Quat);
    Inertial.Curr.Vel = DCM_B_to_I * v3Vel;

    // Inertial.Prev.Vel = v3Vel;
    return true;
};

bool CPhysics::SetAcc(Vector3 v3Acc)
{
    Body.Curr.Acc = v3Acc;
    Inertial.Curr.Acc = v3Acc;
    return true;
};

bool CPhysics::SetTheta(Vector3 v3Theta)
{
    // Body.Prev.Theta = v3Theta;
    // Body.Curr.Theta = v3Theta;
    Body.Curr.Quat = FindQuat(v3Theta);
    // Body.Curr.Quat = Body.Prev.Quat;
    return true;
};

bool CPhysics::SetOmega(Vector3 v3Omega)
{
    // Body.Prev.Omega = v3Omega;
    Body.Curr.Omega = v3Omega;
    Inertial.Curr.Omega = v3Omega;
    // Inertial.Prev.Omega = v3Omega;
    return true;
};

bool CPhysics::SetAlpha(Vector3 v3Alpha)
{
    // Body.Prev.Alpha = v3Alpha;
    Body.Curr.Alpha = v3Alpha;
    return true;
};

bool CPhysics::SetQuat(Vector4 vQuat)
{
    // Body.Prev.Quat = vQuat;
    Body.Curr.Quat = vQuat;
    // Inertial.Prev.Quat = vQuat;

```

```

        Body.Curr.Quat = vQuat;
        return true;
    }
    //=====//
    //    Initial Condition Sets    //
    //          End                    //
    //=====//

    //=====//
    //    Run Time Functions        //
    //          Begin                //
    //=====//
    //    The external model should call on these frequently

    //    Call this function at the begining of the time step!!!
    //    It sets the moment and forces to zero!!!
bool CPhysics::InitilizeTimeStep()
{
    Body.Curr.Force = Zero3();
    Body.Curr.Moment = Zero3();
    return true;
};

    //=====//
    //    Force adding functions    //
    //=====//
    //    This function adds forces in the body frame
    //    All math is done in the body frame anyway.
bool CPhysics::Add_Body_Force (Vector3 v3BodyForce)
{
    Body.Curr.Force += v3BodyForce;
    return true;
};

    //    Transforms inertial frame forces too a body force.
bool CPhysics::Add_Inertial_Force (Vector3 v3InertialForce)
{
    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);
    Matrix3 DCM_I_to_B = A_to_B(Body.Curr.Quat);
    Add_Body_Force (DCM_I_to_B * v3InertialForce);
    return true;
};

    //=====//
    //    Moment adding functions  //
    //=====//
    //    This function adds Moments in the body frame
    //    All math is done in the body frame anyway.
bool CPhysics::Add_Body_Moment (Vector3 v3BodyMoment)
{
    Body.Curr.Moment += v3BodyMoment;

    ////LogInfo("IHateMatlab.html","Set Moment %f, %f,
%f",Body.Curr.Moment.x,Body.Curr.Moment.y,Body.Curr.Moment.z);
    return true;
};

```

```

// Transforms inertial frame Moment too a body Moment.
bool CPhysics::Add_Inertial_Moment (Vector3 v3InertialMoment)
{
    // Vector3 v3BodyMoment = Multiply (DCM_I_to_B,
v3InertialMoment);
    // Add_Body_Moment (v3BodyMoment);
    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);
    Matrix3 DCM_I_to_B = A_to_B(Body.Curr.Quat);
    Add_Body_Moment (DCM_I_to_B *v3InertialMoment);
    return true;
};

//=====//
// This is the real work horse, Integration //
// Provides a new set of pos, and vels, //
// based on forces and moments. //
//
//
// MicrosoPIC deltaT's work good, say 1/1000. //
//=====//
bool CPhysics::IntegrateAll(double deltaT)
{
    double IC[N_MAX];
    double data[N_MAX];
    IC[0] = Body.Curr.Omega.x;
    IC[1] = Body.Curr.Omega.y;
    IC[2] = Body.Curr.Omega.z;

    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);
    IC[3] = Body.Curr.Quat.x;
    IC[4] = Body.Curr.Quat.y;
    IC[5] = Body.Curr.Quat.z;
    IC[6] = Body.Curr.Quat.w;

    IC[7] = Inertial.Curr.Vel.x;
    IC[8] = Inertial.Curr.Vel.y;
    IC[9] = Inertial.Curr.Vel.z;

    IC[10] = Inertial.Curr.Pos.x;
    IC[11] = Inertial.Curr.Pos.y;
    IC[12] = Inertial.Curr.Pos.z;

    ODE45(time, deltaT, time+deltaT, &IC[0], &data[0]);

    Body.Curr.Omega.x = data[0];
    Body.Curr.Omega.y = data[1];
    Body.Curr.Omega.z = data[2];

    Body.Curr.Quat.x = data[3];
    Body.Curr.Quat.y = data[4];
    Body.Curr.Quat.z = data[5];
    Body.Curr.Quat.w = data[6];

    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);

    Inertial.Curr.Vel.x = data[7];

```

```

    Inertial.Curr.Vel.y = data[8];
    Inertial.Curr.Vel.z = data[9];

    Inertial.Curr.Pos.x = data[10];
    Inertial.Curr.Pos.y = data[11];
    Inertial.Curr.Pos.z = data[12];

    time += deltaT;

    // Reset forces and moments to zero
    InitilizeTimeStep();

    return true;
};

//=====//
// Get Functions //
//=====//
Vector3 CPhysics::GetPos() { return Inertial.Curr.Pos; };
Vector3 CPhysics::GetVel() { return Inertial.Curr.Vel; };

Vector3 CPhysics::GetBodyVel()
{
    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);
    Matrix3 DCM_I_to_B = A_to_B(Body.Curr.Quat);
    return DCM_I_to_B * Inertial.Curr.Vel;
};
Vector3 CPhysics::GetBodyAcc() { return Body.Curr.Acc; };
Vector3 CPhysics::GetAcc() { return Inertial.Curr.Acc; };
Vector4 CPhysics::GetQuat() { return Body.Curr.Quat; };
Vector3 CPhysics::GetOmega() { return Body.Curr.Omega; };
// Vector3 GetTheta() { return Body.Curr.Theta; };

Vector3 CPhysics::GetXUnitVector()
{
    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);
    Matrix3 DCM_B_to_I = B_to_A(Body.Curr.Quat);
    Matrix3 DCM_I_to_B = A_to_B(Body.Curr.Quat);
    Vector3 Initial = Zero3();
    Initial.x = 1;

    return DCM_B_to_I * Initial;
};

Vector3 CPhysics::GetYUnitVector()
{
    Body.Curr.Quat = MakeUnitVector(Body.Curr.Quat);
    Matrix3 DCM_B_to_I = B_to_A(Body.Curr.Quat);
    Matrix3 DCM_I_to_B = A_to_B(Body.Curr.Quat);
    Vector3 Initial = Zero3();
    Initial.y = 1;

    return DCM_B_to_I * Initial;
};

Vector3 CPhysics::GetZUnitVector()

```

```

{
    Body.Curr.Quat    = MakeUnitVector(Body.Curr.Quat);
    Vector3 Initial = Zero3();
    Matrix3 DCM_B_to_I = B_to_A(Body.Curr.Quat);
    Matrix3 DCM_I_to_B = A_to_B(Body.Curr.Quat);
    Initial.z = 1;

    return DCM_B_to_I * Initial;
};

Vector3 CPhysics::GetLLA()
{
    return ECEF_to_LLA(Inertial.Curr.Pos);
};
double CPhysics::GetMass()
{
    return Mass;
};

void CPhysics::SetInitials(Vector3 LLA, Vector3 Theta, Vector3 Vel,
Vector3 Omega)
{
    double Max_Count = 1000;
    Matrix3 DCM_I_to_NED, DCM_B_to_I, DCM_B_to_NED;
    double dt = 0.01;
    Vector3 Pos = LLA_to_ECEF(LLA);
    SetPos(Pos);

    SetOmega(Omega);
    SetAcc(Zero3());
    SetAlpha(Zero3());

    //=====//
    //    Find North, East and Down    //
    //=====//
    //    Make position the down vector
    Vector3 Down = Pos * (-1);
    Vector3 NorthPrime;
    NorthPrime.x = Down.x;
    NorthPrime.y = Down.y;
    NorthPrime.z = Down.z+1.0;

    Vector3 East = Cross(Down, NorthPrime);
    Vector3 North = Cross(East, Down);

    North = MakeUnitVector(North);
    East = MakeUnitVector(East);
    Down = MakeUnitVector(Down);

    //=====//
    //    End of Find North, East and Down    //
    //=====//
    double L1, L2, L3, M1, M2, M3, N1, N2, N3;

    SetTheta(Zero3());

    //    From Rolfe and Staples

```

```

L1 = Dot(North, this->GetXUnitVector());
M1 = Dot(North, this->GetYUnitVector());
N1 = Dot(North, this->GetZUnitVector());

L2 = Dot(East, this->GetXUnitVector());
M2 = Dot(East, this->GetYUnitVector());
N2 = Dot(East, this->GetZUnitVector());

L3 = Dot(Down, this->GetXUnitVector());
M3 = Dot(Down, this->GetYUnitVector());
N3 = Dot(Down, this->GetZUnitVector());

DCM_I_to_NED.a[0][0] = L1;    DCM_I_to_NED.a[0][1] = L2;
DCM_I_to_NED.a[0][2] = L3;
DCM_I_to_NED.a[1][0] = M1;    DCM_I_to_NED.a[1][1] = M2;
DCM_I_to_NED.a[1][2] = M3;
DCM_I_to_NED.a[2][0] = N1;    DCM_I_to_NED.a[2][1] = N2;
DCM_I_to_NED.a[2][2] = N3;

DCM_B_to_NED = B_to_A(Theta);
DCM_B_to_I = DCM_I_to_NED * DCM_B_to_NED;

Vector3 New_Theta = Dismantle_B_to_A(DCM_B_to_I);
SetTheta(New_Theta);

SetBodyVel(Vel);

InitilizeTimeStep();
}

Vector3 CPhysics::GetTheta()
{
    //    Yaw angle = arctan2 of y, x
    Vector3 pos = Inertial.Curr.Pos;

    //=====//
    //    Find North, East and Down    //
    //=====//
    //    Make position the down vector
    Vector3 Down = pos * (-1);
    Vector3 NorthPrime;
    NorthPrime.x = Down.x;
    NorthPrime.y = Down.y;
    NorthPrime.z = Down.z+1.0;

    Vector3 East = Cross(Down, NorthPrime);
    Vector3 North = Cross(East, Down);

    North = MakeUnitVector(North);
    East = MakeUnitVector(East);
    Down = MakeUnitVector(Down);

    //=====//
    //    End of Find North, East and Down    //
    //=====//
}

```

```

//    From Rolfe and Staples
double L1 = Dot(North, this->GetXUnitVector());
double M1 = Dot(North, this->GetYUnitVector());
double N1 = Dot(North, this->GetZUnitVector());

double L2 = Dot(East, this->GetXUnitVector());
double M2 = Dot(East, this->GetYUnitVector());
double N2 = Dot(East, this->GetZUnitVector());

double L3 = Dot(Down, this->GetXUnitVector());
double M3 = Dot(Down, this->GetYUnitVector());
double N3 = Dot(Down, this->GetZUnitVector());

double L3_temp = L3;
//    Check for L3 values that will cause
//    imaginary arc sines
if (L3_temp > 1)
    L3_temp = 1;
if (L3_temp < -1)
    L3_temp = -1;

double pitch = asin(-L3_temp);

double sign_L2;

if (L2<0)
    sign_L2 = -1;
else
    sign_L2 = 1;

double yaw;
double L1_temp;
L1_temp = L1/cos(pitch);

//    Check for L1 values that will cause
//    imaginary arc cosines
if (L1_temp>1)
    L1_temp = 1;

if (L1_temp < -1)
    L1_temp = -1;

yaw = acos(L1_temp)*sign_L2;

double sign_M3;
if (M3<0)
    sign_M3 = -1;
else
    sign_M3 = 1;

double N3_temp;
N3_temp = N3/cos(pitch);

//    Check for N3 values that will cause
//    imaginary arc cosines
if (N3_temp>1)

```

```

        N3_temp = 1;
    if (N3_temp < -1)
        N3_temp = -1;

    double roll = acos(N3_temp)*sign_M3;

    debug = true;
    // End Rolfe and staples
    if (debug)
    {
        //LogInfo("NED.html","North Prime [%f %f %f]",NorthPrime.x,
NorthPrime.y,    NorthPrime.z);
        //LogInfo("NED.html","North [%f %f %f]", North.x,
North.y,    North.z);
        //LogInfo("NED.html","East [%f %f %f]", East.x,
East.y,    East.z);
        //LogInfo("NED.html","Down [%f %f %f]", Down.x,
Down.y,    Down.z);

        //LogInfo("NED.html","GetX [%f %f %f]",
GetXUnitVector().x,    GetXUnitVector().y,
    GetXUnitVector().z);
        //LogInfo("NED.html","GetY [%f %f %f]",
GetYUnitVector().x,    GetYUnitVector().y,
    GetYUnitVector().z);
        //LogInfo("NED.html","GetZ [%f %f %f]",
GetZUnitVector().x,    GetZUnitVector().y,
    GetZUnitVector().z);

    }
    debug = false;

    Vector3 Theta;
    Theta.x = roll;
    Theta.y = pitch;
    Theta.z = yaw;
    //LogInfo("theta.html","Theta [%f %f %f]", Theta.x, Theta.y,
Theta.z);

    return Theta;
}

```

APPENDIX C – Quaternions in Depth

Quaternions are an interesting bugger. They're representative of a group called hyper complex numbers. That's a fancy way of saying theirs one real value and three imaginary. Because of their hyper complex status, quaternion numbers need special addition, subtraction and multiplication rules. Since we're not developing a control system that's driven by quaternions, those operations are irrelevant. All we need to do is integrate them to track our orientation. It should be noted that every source of quaternion equations uses different subscript notation. The notation must be checked before the equations are compared. The following is a short list of how different organizations store their quaternion data. For the most part, this paper will use the ESDU notation, unless designated as different.

Rolfe and Staples

Flight Simulation by Rolfe and Staples is a time honored classic aerospace simulation book. Although written in 1986, it's still an accurate source with regards to the mathematics. An excellent notation to use if you're computer language allows zero indexing of arrays.

$$e_0 = \cos\left(\frac{\alpha}{2}\right) \quad \text{Real Component}$$

$$e_1 = \cos(\hat{x})\sin\left(\frac{\alpha}{2}\right) \quad \text{X cosine component}$$

$$e_2 = \cos(\hat{y})\sin\left(\frac{\alpha}{2}\right) \quad \text{Y cosine component}$$

$$e_3 = \cos(\hat{z})\sin\left(\frac{\alpha}{2}\right) \quad \text{Z cosine component}$$

Microsoft Convention

Oddly enough quaternions are the core of the modern 3d computer graphics engine (and therefore gaming industry). Microsoft has published computer code libraries (DirectX) on the subject. Microsoft breaks the vectors into a class format. As a class there are no arrays to index. This notation clears up confusion as to which component is real, and which is an axis component. All of the Simulation S-Functions using quaternion are notated in this fashion.

$$w = \cos\left(\frac{\alpha}{2}\right) \quad \text{Real Component}$$

$$x = \cos(\hat{x})\sin\left(\frac{\alpha}{2}\right) \quad \text{X cosine component}$$

$$y = \cos(\hat{y})\sin\left(\frac{\alpha}{2}\right)$$

Y cosine component

$$z = \cos(\hat{z})\sin\left(\frac{\alpha}{2}\right)$$

Z cosine component

Seen this before, don't remember where

Very much like the ESDU Paper – 89024a notation, with the exception that the subscripts are all incremented one. This notation is handy for checking and writing MATLAB script files as MATLAB doesn't allow zero indexes in their matrixes or vectors.

$$e_1 = \cos(\hat{x})\sin\left(\frac{\alpha}{2}\right)$$

X cosine component

$$e_2 = \cos(\hat{y})\sin\left(\frac{\alpha}{2}\right)$$

Y cosine component

$$e_3 = \cos(\hat{z})\sin\left(\frac{\alpha}{2}\right)$$

Z cosine component

$$e_4 = \cos\left(\frac{\alpha}{2}\right)$$

Real Component

Quaternion Properties of Interest

One of the properties of quaternion orientations is that their unitary. That means the quaternions must be a unit vector at every time instance. If they're not a unit vector, it's possible to affect the direction cosine matrix. The direction cosine matrix will start causing scaling effects; IE forces in one direction could be increased, while forces in other directions could be decreased. It's perfectly safe to perform a normalize operation during every time step. It is in fact recommended to do a normalization to prevent quaternion drift. A Runge Kutta 4/5 solver with tight tolerances shouldn't need the normalization. The down side, a Runge Kutta solvers with tight tolerances will consume tons of processor power and run slow.

APPENDIX D – ECEF

ecef_sixdof_0017 is a wrapper file for the physics class. It transforms MATLAB signals into data for the physics class, then sends the output from the physics class back to MATLAB.

Code – Cpp File

```
//=====
//
// AUTHOR(S): Chris Adan [CJA2]
//
// DATE:      05/03/04
//
//
// Copyright (c) ALL RIGHTS RESERVED
//
//
// REVISION HISTORY:
//
//
//
// REV AUTHOR      DATE      DESCRIPTION
// 0   CJA2      05/03/04   File created
// 1   CJA2      05/18/04   Linear Equation of motion put in.
// 2   CJA2      05/25/04   Full integration system in place,
debugging
// 2   CJA2      06/08/04   Altered Gravity System, now works
// 3   CJA2      06/09/04   Adding my physics class in. The physics
//
//                               class of my game has
the rotational and //
//                               quaternion elements
already programmed //
//                               in.
//
// 4   CJA2      07/09/04   Though the system runs, it doesn't handle
//
//                               earth rotation
//
//
// S-mex: See simulink/src/sfuntmpl.doc
//
//
// Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights
Reserved. //
```

```

// $Revision: 1.3
//
//
//=====
=//
// Note earth is neglected, because it's lame.
// Earth frame only matters for landing orientation
// instruments, and the like.
// Earth = North East Down
// Hey, the earth is round now!! Sphere round!!

#include "simstruc.h"

#include "lost_physics_0015.h"

// Function: mdlInitializeSizes
=====
static void mdlInitializeSizes(SimStruct *S)
{
    // Number of parameters to type in, usually
    // a file name or a number of time steps, ect.
    // Second parameter is the number of inputs.
    ssSetNumSFcnParams(S, 0);

    // Check for Simulink port matching errors
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
    {
        printf("S Function Parameter Error.\n");
        return;
    };

    // File name input example.
    // mxGetString(ssGetSFcnParam(S,0),fname,N_STRING);

    //=====//
    // Initilize Input Ports //
    //=====//
    // Set the number of input ports. INPUT PORTS =
    // //Log the error.
    if (!ssSetNumInputPorts(S, 10))
    {
        printf("Set Number Input Ports Error.\n");
        return;
    };

    // Initial Location in LLA !!!
    ssSetInputPortWidth(S, 0, 3);

    // Intial Oritenation in Inertial Frame !!!
    ssSetInputPortWidth(S, 1, 3);

    // Initial Velocity in the Body
    ssSetInputPortWidth(S, 2, 3);

    // Initial Anglular Velcotiy in the Body

```

```

ssSetInputPortWidth(S, 3, 3);

// Forces and moments in the body frame
ssSetInputPortWidth(S, 4, 6);

// Forces and moments in the interial frame
ssSetInputPortWidth(S, 5, 6);

// Mass
ssSetInputPortWidth(S, 6, 1);

// MassDot
ssSetInputPortWidth(S, 7, 1);

// Interias
ssSetInputPortWidth(S, 8, 6);

// Resets
ssSetInputPortWidth(S, 9, 1);

// This command must be in here,
// The second value is the port number
// The last value is 1.
for (int i = 0; i < 10; i++)
    ssSetInputPortDirectFeedThrough(S,i,1);

//=====//
// Initilize Output Ports //
//=====//
// very much like set input ports
if (!ssSetNumOutputPorts(S,8))
{
    printf("Set Num Output Ports Error\n");
    return;
}

// Needed outputs
// LLA
ssSetOutputPortWidth(S, 0, 3);

// Euler in local frame
ssSetOutputPortWidth(S, 1, 3);

// PQR in local frame
ssSetOutputPortWidth(S, 2, 3);

// XYZ Inertial
ssSetOutputPortWidth(S, 3, 3);

// UVW Inertial
ssSetOutputPortWidth(S, 4, 3);

// XYZ acceleration inertial
ssSetOutputPortWidth(S, 5, 3);

// Quats
ssSetOutputPortWidth(S, 6, 4);

```

```

    // Reset
    ssSetOutputPortWidth(S, 7, 1);

    //=====//
    //    No feed through for outputs!!! //
    //=====//

    //    Number of sample times s has.  One is good.
    ssSetNumSampleTimes(S, 1);

    ssSetNumPWork(S, 1);

    // Exception free code is currently disabled.
    //    Needed to increase speed
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
};

// Function: mdlInitializeSampleTimes
=====
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MDL_START
#if defined(MDL_START)

// Function: mdlStart
=====
static void mdlStart(SimStruct *S)
{
    ssGetPWork(S)[0] = new CPhysics();
}
#endif // MDL_START

#define MDL_INITIALIZE_CONDITIONS
#if defined(MDL_INITIALIZE_CONDITIONS)
static void mdlInitializeConditions(SimStruct *S)
{
    CPhysics *pPhysics = (CPhysics *) ssGetPWork(S)[0];

    InputRealPtrsType uInitialLLA =
ssGetInputPortRealSignalPtrs(S,0);
    InputRealPtrsType uInitialTheta =
ssGetInputPortRealSignalPtrs(S,1);
    InputRealPtrsType uInitialUVW =
ssGetInputPortRealSignalPtrs(S,2);
    InputRealPtrsType uInitialPQR =
ssGetInputPortRealSignalPtrs(S,3);

    Vector3 LLA, Theta, Vel, Omega;
    LLA.x = *uInitialLLA[0];
    LLA.y = *uInitialLLA[1];
    LLA.z = *uInitialLLA[2];
}
#endif

```

```

    Theta.x = *uInitialTheta[0];
    Theta.y = *uInitialTheta[1];
    Theta.z = *uInitialTheta[2];

    Vel.x = *uInitialUVW[0];
    Vel.y = *uInitialUVW[1];
    Vel.z = *uInitialUVW[2];

    Omega.x = *uInitialPQR[0];
    Omega.y = *uInitialPQR[1];
    Omega.z = *uInitialPQR[2];

    pPhysics->SetInitials(LLA, Theta, Vel, Omega);
}

#endif          //      #define MDL_INITIALIZE_CONDITIONS

// Function: mdlOutputs
=====
static void mdlOutputs(SimStruct *S, int_T tid)
{
    //LogInfo("IHateMatlab.html","mdlOutputs %f\n", ssGetT(S));
    CPhysics *pPhysics = (CPhysics *) ssGetPWork(S)[0];

    Vector3 BodyForce, InertialForce;
    Vector3 BodyMoment, InertialMoment;
    Vector3 LLA;
    double dt = ssGetStepSize(S);
    int iSixdofMult = 1;

    InputRealPtrsType uFMBodyPtrs          =
ssGetInputPortRealSignalPtrs(S,4);
    InputRealPtrsType uFMInertialPtrs      =
ssGetInputPortRealSignalPtrs(S,5); // Gravity
    InputRealPtrsType uMassPtrs            =
ssGetInputPortRealSignalPtrs(S,6);
    InputRealPtrsType uMassDotPtrs         =
ssGetInputPortRealSignalPtrs(S,7);
    InputRealPtrsType uInteriaPtrs         =
ssGetInputPortRealSignalPtrs(S,8);
    InputRealPtrsType resetPtrs            =
ssGetInputPortRealSignalPtrs(S,9);

    if (*resetPtrs[0]>0.5)
    {
        InputRealPtrsType uInitialLLA =
ssGetInputPortRealSignalPtrs(S,0);
        InputRealPtrsType uInitialTheta =
ssGetInputPortRealSignalPtrs(S,1);
        InputRealPtrsType uInitialUVW =
ssGetInputPortRealSignalPtrs(S,2);
        InputRealPtrsType uInitialPQR =
ssGetInputPortRealSignalPtrs(S,3);

        Vector3 LLA, Theta, Vel, Omega;
        LLA.x = *uInitialLLA[0];

```

```

    LLA.y = *uInitialLLA[1];
    LLA.z = *uInitialLLA[2];

    Theta.x = *uInitialTheta[0];
    Theta.y = *uInitialTheta[1];
    Theta.z = *uInitialTheta[2];

    Vel.x = *uInitialUVW[0];
    Vel.y = *uInitialUVW[1];
    Vel.z = *uInitialUVW[2];

    Omega.x = *uInitialPQR[0];
    Omega.y = *uInitialPQR[1];
    Omega.z = *uInitialPQR[2];

    pPhysics->SetInitials(LLA, Theta, Vel, Omega);
}

BodyForce.x = *uFMBodyPtrs[0];
BodyForce.y = *uFMBodyPtrs[1];
BodyForce.z = *uFMBodyPtrs[2];

BodyMoment.x = *uFMBodyPtrs[3];
BodyMoment.y = *uFMBodyPtrs[4];
BodyMoment.z = *uFMBodyPtrs[5];

InertialForce.x = *uFMInertialPtrs[0];
InertialForce.y = *uFMInertialPtrs[1];
InertialForce.z = *uFMInertialPtrs[2];

InertialMoment.x = *uFMInertialPtrs[3];
InertialMoment.y = *uFMInertialPtrs[4];
InertialMoment.z = *uFMInertialPtrs[5];

double Mass = *uMassPtrs[0];
double MassDot = *uMassDotPtrs[0];
double I[6];
I[0] = *uInteriaPtrs[0];
I[1] = *uInteriaPtrs[1];
I[2] = *uInteriaPtrs[2];
I[3] = *uInteriaPtrs[3];
I[4] = *uInteriaPtrs[4];
I[5] = *uInteriaPtrs[5];

pPhysics->SetMass(Mass);
pPhysics->SetMassDot(MassDot);

pPhysics->SetInertia(I);

//    Reset the forces and moments
pPhysics->InitilizeTimeStep();

//    Add Body forces and moments
//    Examples, lift, aerodynamics, ground friction
pPhysics->Add_Body_Force(BodyForce);
pPhysics->Add_Body_Moment(BodyMoment);

```

```

//    Add Inertial Forces and moments
pPhysics->Add_Inertial_Force(InertialForce);
pPhysics->Add_Inertial_Moment(InertialMoment);

//    Try to integrate the data
pPhysics->IntegrateAll(dt);

//    Grab the pointers to the output stream.
real_T *uLLAOutPtrs      = ssGetOutputPortRealSignal(S,0);
real_T *uEulerOutPtrs   = ssGetOutputPortRealSignal(S,1);
real_T *uEulerRates     = ssGetOutputPortRealSignal(S,2);
real_T *uXYZ_Inertial   = ssGetOutputPortRealSignal(S,3);
real_T *uUVW_Inertial   = ssGetOutputPortRealSignal(S,4);
real_T *uXYZ_Acc_Inertial = ssGetOutputPortRealSignal(S,5);
real_T *uQuatOut        = ssGetOutputPortRealSignal(S,6);
real_T *ResetPtr        = ssGetOutputPortRealSignal(S,7);

//    Non-rotating earth
LLA      = ECEF_to_LLA(pPhysics->GetPos());
Vector3 Theta = pPhysics->GetTheta();
Vector3 Omega = pPhysics->GetOmega();
Vector3 Inertial_Pos = pPhysics->GetPos();
Vector3 Inertial_Vel = pPhysics->GetBodyVel();
Vector3 Inertial_Acc = pPhysics->GetBodyAcc();

if(IsNaN(LLA) || IsNaN(Theta) || IsNaN(Omega) ||
IsNaN(Inertial_Pos) || IsNaN(Inertial_Vel) || IsNaN(Inertial_Acc))
{
    /*ssSetErrorStatus(S,"SixDof Eulers are Nan\n");
    ssSetStopRequested(S, 1);
    return;*/
    InputRealPtrsType uInitialLLA =
ssGetInputPortRealSignalPtrs(S,0);
    InputRealPtrsType uInitialTheta =
ssGetInputPortRealSignalPtrs(S,1);
    InputRealPtrsType uInitialUVW =
ssGetInputPortRealSignalPtrs(S,2);
    InputRealPtrsType uInitialPQR =
ssGetInputPortRealSignalPtrs(S,3);
    Vector3 Vel;
    for(int k = 0; k < 2; k++)
    {

        LLA.x = *uInitialLLA[0];
        LLA.y = *uInitialLLA[1];
        LLA.z = *uInitialLLA[2];

        Theta.x = *uInitialTheta[0];
        Theta.y = *uInitialTheta[1];
        Theta.z = *uInitialTheta[2];

        Vel.x = *uInitialUVW[0];
        Vel.y = *uInitialUVW[1];
        Vel.z = *uInitialUVW[2];

        Omega.x = *uInitialPQR[0];

```

```

    Omega.y = *uInitialPQR[1];
    Omega.z = *uInitialPQR[2];

    pPhysics->SetInitials(LLA, Theta, Vel, Omega);

    pPhysics->SetMass(Mass);
    pPhysics->SetMassDot(MassDot);

    pPhysics->SetInertia(I);

    //    Reset the forces and moments
    pPhysics->InitilizeTimeStep();

    //    Add Body forces and moments
    //    Examples, lift, aerodynamics, ground friction
    pPhysics->Add_Body_Force(Zero3());
    pPhysics->Add_Body_Moment(Zero3());

    //    Add Interial Forces and moments
    pPhysics->Add_Inertial_Force(Zero3());
    pPhysics->Add_Inertial_Moment(Zero3());

    //    Try to integrate the data
    pPhysics->IntegrateAll(dt);
}
LLA          = ECEF_to_LLA(pPhysics->GetPos());
Theta = pPhysics->GetTheta();
Omega = pPhysics->GetOmega();
Inertial_Pos = pPhysics->GetPos();
Inertial_Vel = pPhysics->GetBodyVel();
Inertial_Acc = pPhysics->GetBodyAcc();

ResetPtr[0] = true;
}

uLLAOutPtrs[0] = LLA.x;
uLLAOutPtrs[1] = LLA.y;
uLLAOutPtrs[2] = LLA.z;

uEulerOutPtrs[0] = Theta.x;
uEulerOutPtrs[1] = Theta.y;
uEulerOutPtrs[2] = Theta.z;

uEulerRates[0] = Omega.x;
uEulerRates[1] = Omega.y;
uEulerRates[2] = Omega.z;

uXYZ_Inertial[0] = Inertial_Pos.x;
uXYZ_Inertial[1] = Inertial_Pos.y;
uXYZ_Inertial[2] = Inertial_Pos.z;

uUVW_Inertial[0] = Inertial_Vel.x;
uUVW_Inertial[1] = Inertial_Vel.y;
uUVW_Inertial[2] = Inertial_Vel.z;

uXYZ_Acc_Inertial[0] = Inertial_Acc.x;
uXYZ_Acc_Inertial[1] = Inertial_Acc.y;

```

```

uXYZ_Acc_Inertial[2] = Inertial_Acc.z;

ResetPtr[0] = false;
/*
if(IsNaN(LLA))
{
    ssSetErrorStatus(S,"SixDof LLA is Nan\n");
    ssSetStopRequested(S, 1);
    return;
}
uLLAOutPtrs[0] = LLA.x;
uLLAOutPtrs[1] = LLA.y;
uLLAOutPtrs[2] = LLA.z;
uEulerOutPtrs[0] = Theta.x;
uEulerOutPtrs[1] = Theta.y;
uEulerOutPtrs[2] = Theta.z;

Vector3 Omega;
Omega = pPhysics->GetOmega();
if(IsNaN(Omega))
{
    ssSetErrorStatus(S,"SixDof Euler Rates are Nan\n");
    ssSetStopRequested(S, 1);
    return;
}
uEulerRates[0] = Omega.x;
uEulerRates[1] = Omega.y;
uEulerRates[2] = Omega.z;

Vector3 Inertial_Pos = pPhysics->GetPos();
if(IsNaN(Inertial_Pos))
{
    ssSetErrorStatus(S,"SixDof Position is Nan\n");
    ssSetStopRequested(S, 1);
    return;
}
uXYZ_Inertial[0] = Inertial_Pos.x;
uXYZ_Inertial[1] = Inertial_Pos.y;
uXYZ_Inertial[2] = Inertial_Pos.z;

////LogText("flight_data_final.xls", "%f\t%f\t%f\t%f\t%f\n", Inertial_Pos.x, Inertial_Pos.y, Inertial_Pos.z, dmag, Mag(pPhysics->GetPos()));

Vector3 Inertial_Vel = pPhysics->GetBodyVel();
if(IsNaN(Inertial_Vel))
{
    ssSetErrorStatus(S,"SixDof Body Velocities are Nan\n");
    ssSetStopRequested(S, 1);
    return;
}
uUVW_Inertial[0] = Inertial_Vel.x;
uUVW_Inertial[1] = Inertial_Vel.y;
uUVW_Inertial[2] = Inertial_Vel.z;

Vector3 Inertial_Acc = pPhysics->GetBodyAcc();
if(IsNaN(Inertial_Acc))

```

```

    {
        ssSetErrorStatus(S,"SixDof Body Accelerations are Nan\n");
        ssSetStopRequested(S, 1);
        return;
    }
    uXYZ_Acc_Inertial[0] = Inertial_Acc.x;
    uXYZ_Acc_Inertial[1] = Inertial_Acc.y;
    uXYZ_Acc_Inertial[2] = Inertial_Acc.z;*/

    Vector4 Quats = pPhysics->GetQuat();
    double dmag = Mag(Quats);
    uQuatOut[0] = Quats.w;
    uQuatOut[1] = Quats.x;
    uQuatOut[2] = Quats.y;
    uQuatOut[3] = Quats.z;
}

/* Function: mdlTerminate
=====
* Abstract:
*/
// Memory clean up, ect goes here.
static void mdlTerminate(SimStruct *S)
{
    CPhysics *pPhysics = (CPhysics *) ssGetPWork(S)[0];
    delete(pPhysics);
    pPhysics = NULL;
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-
file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

```

References

- [1] Cambridge Aerospace Series, “Flight Simulation”, by Rolfe and Staples.
- [2] <http://en.wikipedia.org/wiki/Quaternion>
- [3] <http://mathworld.wolfram.com/Quaternion.html>
- [4] <http://www.nasa.gov> <Figure 2 - Body Axis>
- [5] “Numerical Recipes”, Press, Flannery, Teukolsky and Vetterling, Cambridge