Using Simulink and STK for Attitude Modeling

A co-simulation guide

Giuseppe Corrao – European System Engineer

Version 4 – May 2015

CONTENTS

1 - Connection Architecture

To connect STK with the external world there are basically three ways:

1) ***Using external file*** – This is a completely off-line solution. Using a parser the outcomes from both MATLAB and STK can be made available for data ingestion and evaluation from the other software;
2) ***Using Connect*** – This solution is available when the MATLAB connector is installed as STK plugin and allows MATLAB to act as a client respect to STK, which is a server in this case;
3) ***Using COM*** – COM technology is available in Windows environment and allows third-parties software to exchange data thorough COM objects. This is a flexible, object oriented approach that also allows to send connect commands using the COM interface.

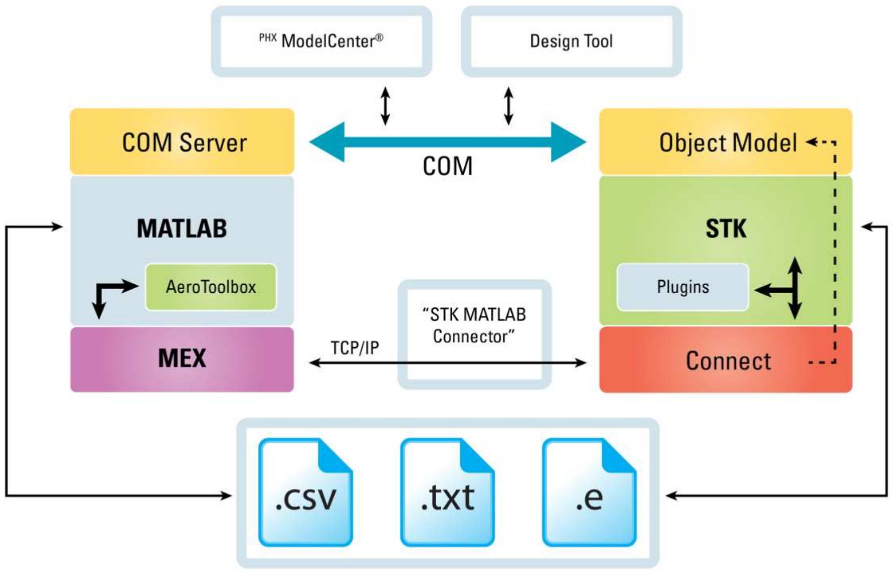


Fig 1 –Connection Architecture

COM objects are used to connect STK with Simulink; this does not require to have the MATLAB Connector installed. A set of code snippets is listed in Annex B.

Using the COM paradigm is possible to embed an STK scenario (and to get back numerical values from it at each integration step) by using a special Simulink block that is called *S-Function* (see next Section for additional information). Through the S-Function any of the data providers available in STK can be imported into the Simulink plant and used as input data for custom models.

To explain how we can take benefit from introducing STK into a Simulink model an attitude control example has been build.

2 The Simulink Plant

In the proposed Simulink model STK is embedded by using an S-Function that loads the proper scenario and then, at each integration step:

1) Get from STK (using data providers) the current attitude state (quaternion, angular speeds and Euler angles) of two satellites:
   a. The first (called *Reference*) is directly managed by STK. It is has a Multi-Segment attitude profile whose segments can be changed over time from the satellite's attitude properties. This allows the user to emulate attitude change commands sent to the satellite;
   b. The second (called *True*) has instead a Real Time attitude propagator that receives data from the Simulink model.
2) Receives the updated attitude of the *True* satellite and displays it in STK

This model allows working into two different modes:

1) Open Loop:

Using this option we force the *True* satellite to get the commanded attitude There is no feedback in this configuration: the satellite remains in the commanded attitude state until we change something.

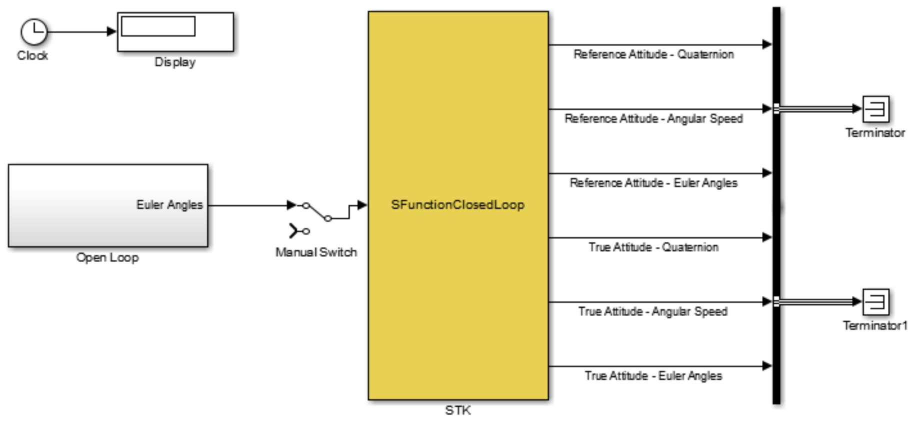

Clock
Display
Euler Angles
Open Loop
Manual Switch
SFunctionClosedLoop
STK
Reference Attitude - Quaternion
Reference Attitude - Angular Speed
Reference Attitude - Euler Angles
True Attitude - Quaternion
True Attitude - Angular Speed
True Attitude - Euler Angles
Terminator
Terminator1

Fig 2 – Equivalent Simulink plant for Open Loop case

In the Open Loop block we can change any of the Euler angle independently from the others. Virtually any attitude profile can be designed in Simulink and then inserted into STK for successive analysis and trade studies.

2) *Closed loop:*

Using this option we can directly use custom control law and integrate attitude dynamics. The true attitude state is compared with the reference one and, according with the implemented control law, a torque is applied. Attitude dynamics and kinematics equations are then integrated to push back the new attitude in STK.

In the following Figure is shown the Simulink plant that emulates the onboard attitude control system. The most important blocks (Navigation, Guidance & Control, actuators and equations of motion) are here presented in the most generic form: in such a way that the model itself can be properly modified for specific purposes.

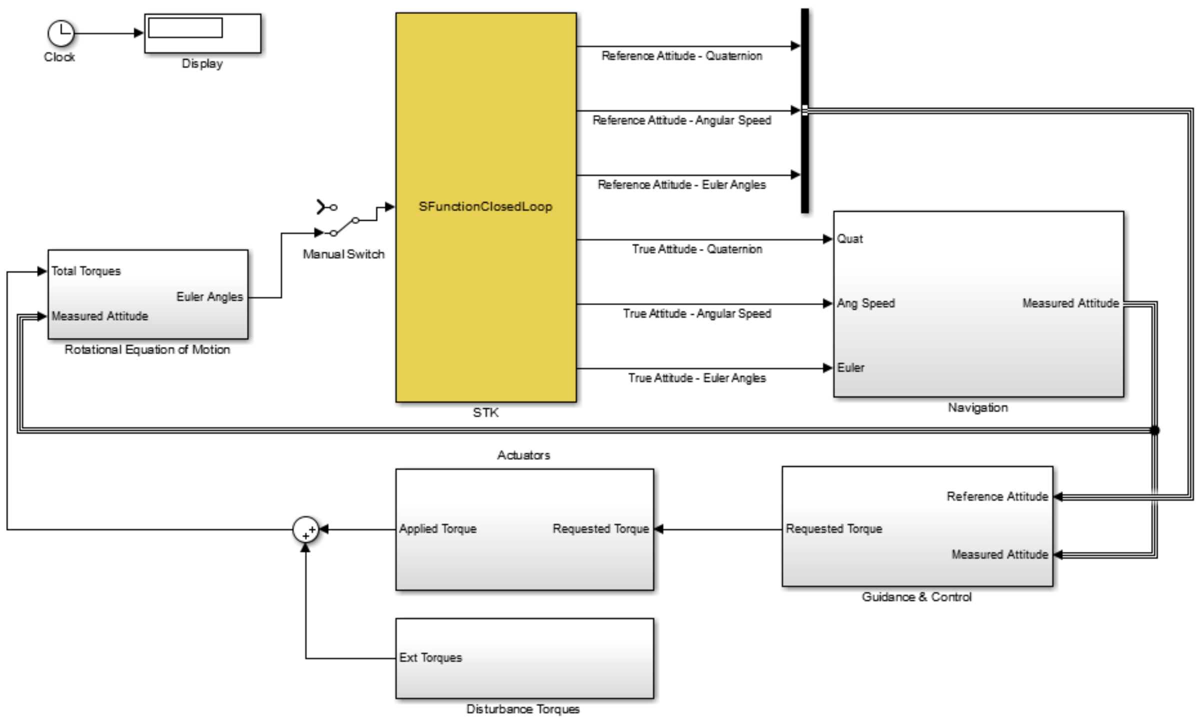


Fig 3 – Equivalent Simulink plant for Closed Loop case

2.2 – Navigation Block

The Navigation block receives the attitude data of the *True* satellite from STK at each integration step. Since the data represent "perfect" measurements, we can decide to add some noise in the loop in order to emulate the onboard attitude estimation process.

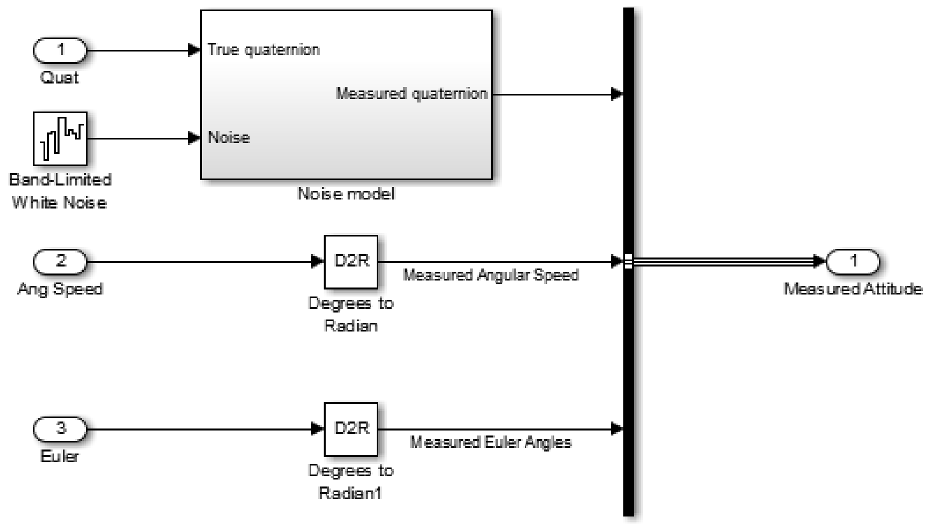


Fig 4 – Attitude Determination block

Here the user has just the ability to inject a white noise over the true quaternion values; by implementing a proper measurement bias model the attitude stability performances can be investigated in detail.

In the Figure below are shown the *True* satellite YPR error angles and the measured angular speed for a nadir pointing attitude with white noise.

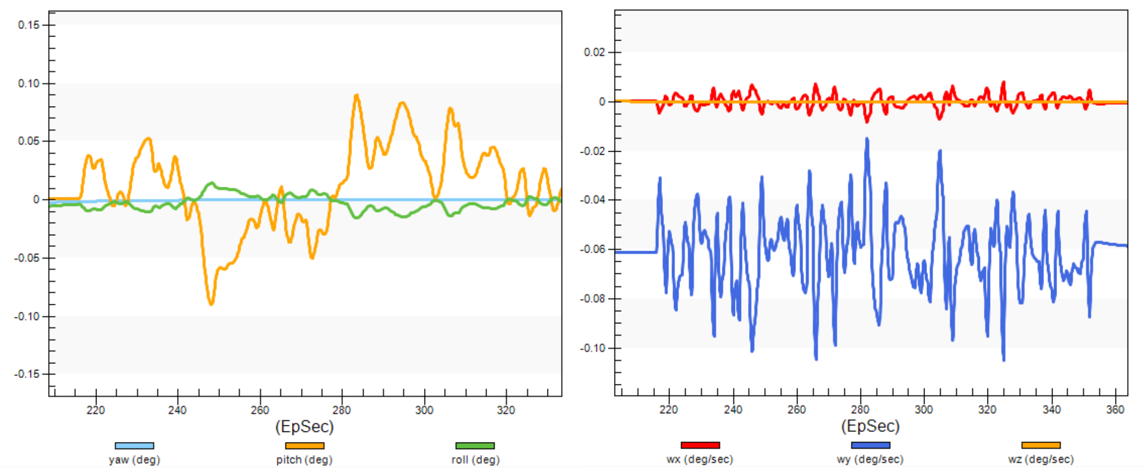


Fig 5 – Measured attitude state using white noise

2.3 – Guidance & Control

The Guidance & Control block hosts the control feedbacks law. A control law receives the attitude error data and outputs the commanded torques across the three body axes. In this example a simple Proportional Derivative controller is used; it has been coded in a MATLAB script embedded to the Simulink model.

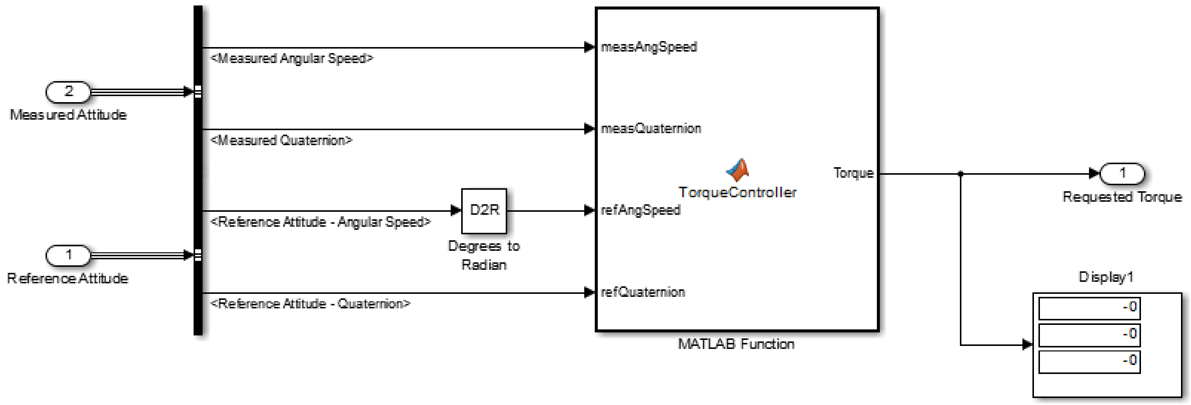


Fig 6 – Guidance & Control block

Here below is shown a screenshot from STK that reports the angular speed of the satellite after a commanded Sun pointing state from an inertially fixed state. Note how the angular velocity along the Z body axis as a stationary state of +1 deg/sec (as expected from the chosen attitude profile).

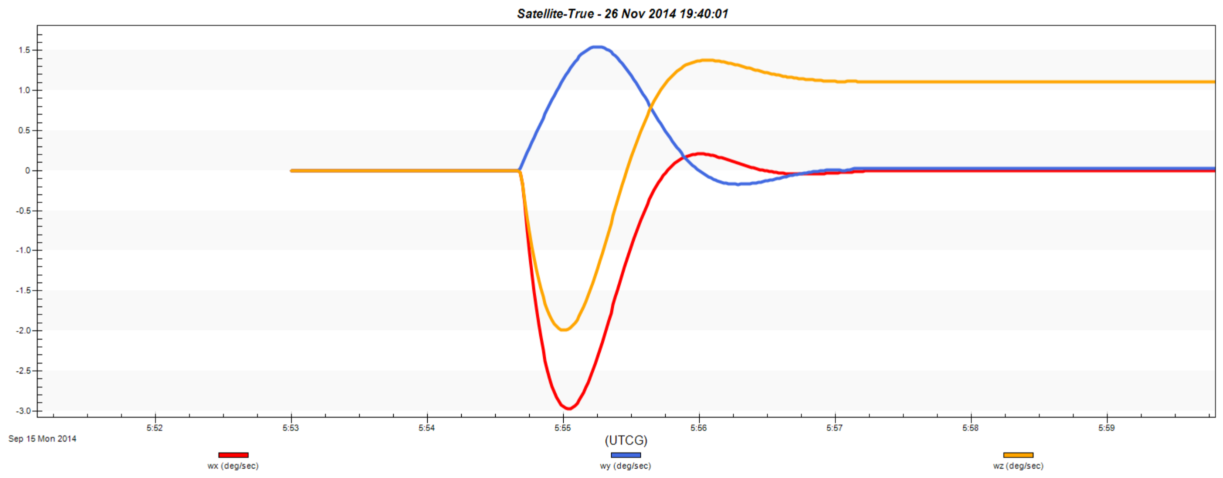


Fig 7 – Angular Rates achieved while following target angular speed (6 and 0 deg/sec)

2.4 – Actuators

The Actuators block hosts the wheel/reaction thruster model. It is configurable as well; in this example there's just a slope limiter and a saturation filter.

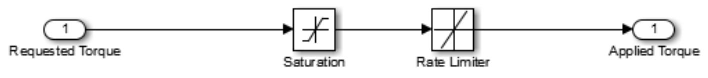


Fig 8 – Actuators block

2.5 – Disturbance Torques

The Disturbance Torques block adds environmental torque to the model. Despite it can be fit with any disturbance torque (gravity gradient, drag, etc.), in this model we have a simple sinusoidal model whose period is an half of the orbit period.

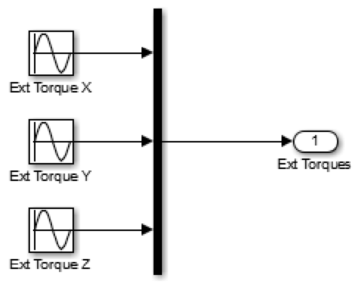


Fig 9 – Disturbance Torque block

2.6 – Equation of Motion

It is composed by two subsystems:

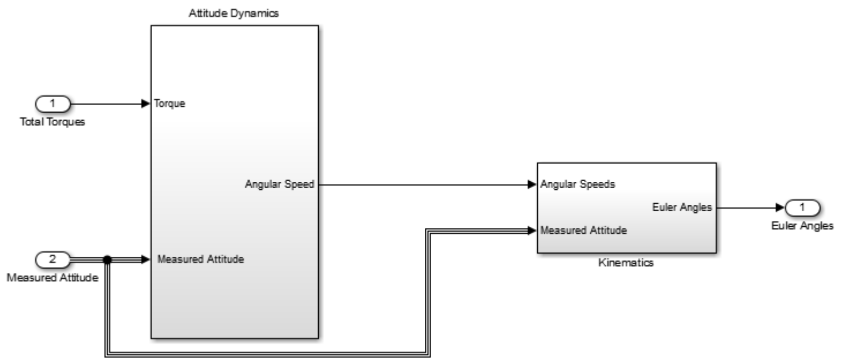


Fig 10 – Equation of Motion block

2.6.1 – Attitude Dynamics

The Attitude Dynamics block solves the Euler's equation at each time step. Here the satellite inertia matrix is defined and used.

The block outputs the angular speed respect to the body axes form the applied torque.

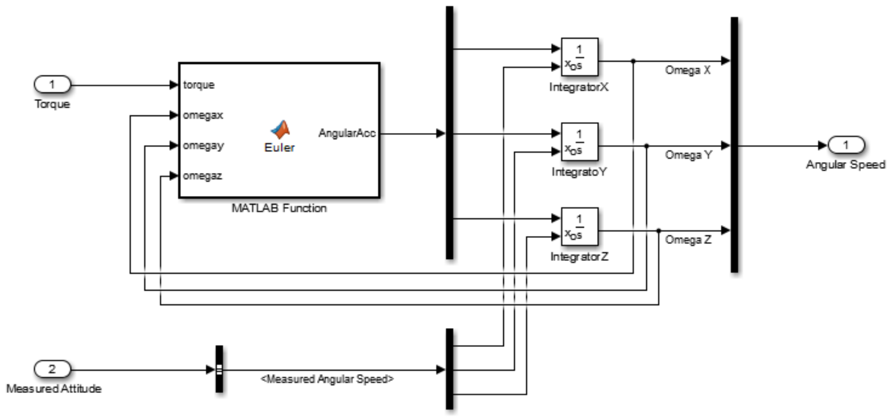


Fig 9 – Attitude Dynamics block

2.6.3 – Kinematics

Finally, the *Kinematics* block returns the commanded Euler angles to STK.

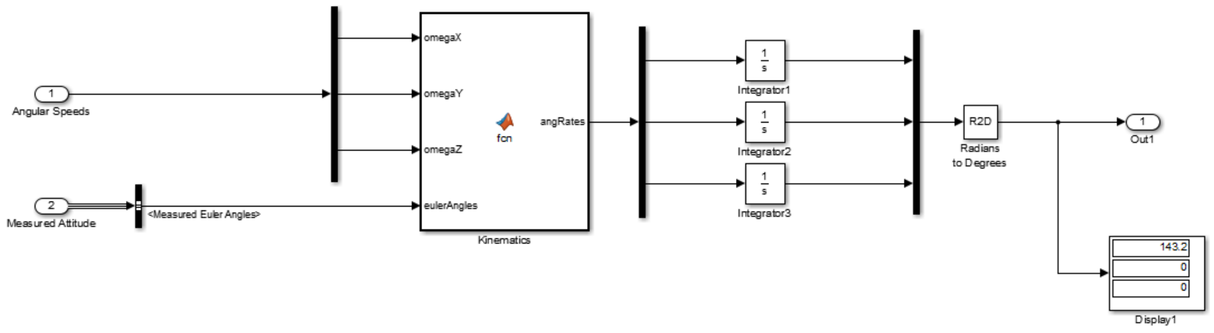
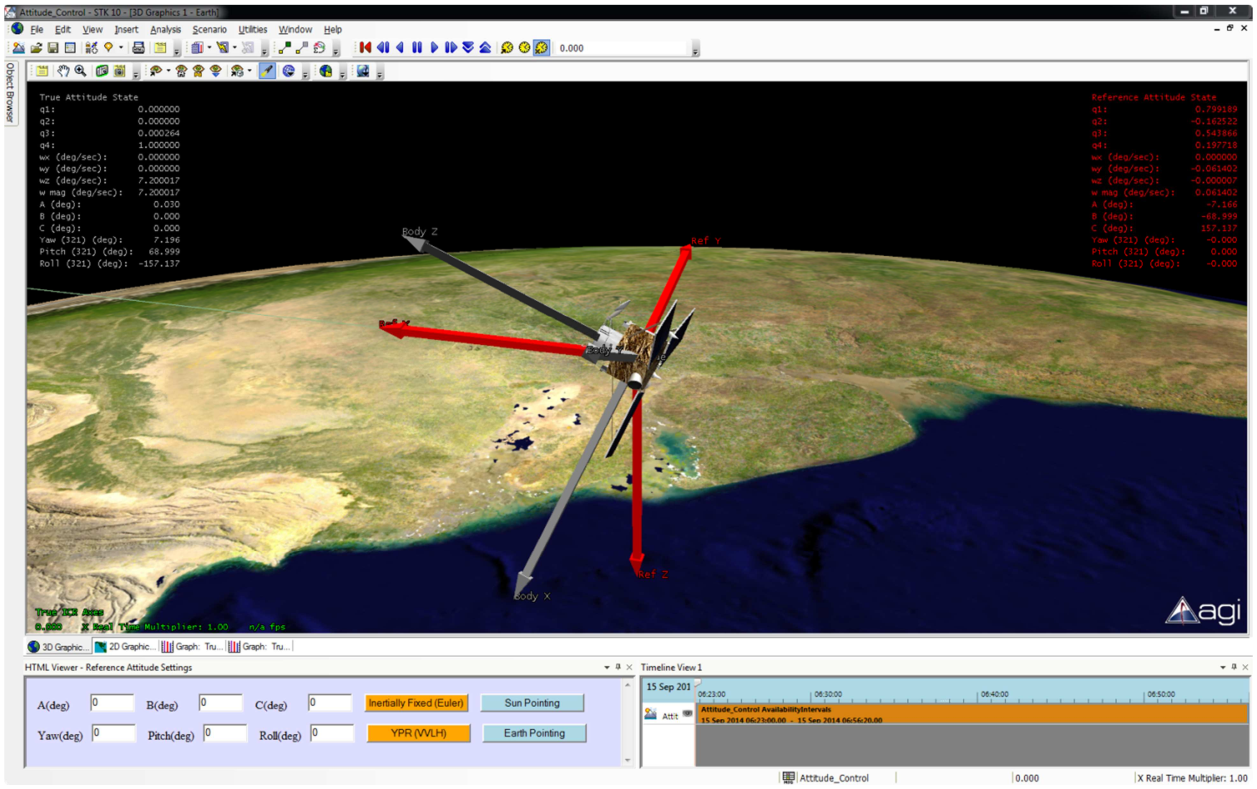


Fig 10 – Kinematics block

2.7 – STK Scenario (managed by the S-Function)

The model uses an STK scenario that contains two satellites: the *True* one (whose attitude is changed using the data coming from the Simulink model) and the *Reference* one (whose attitude is the reference for the Guidance & Control block); this has to be changed directly from STK using the HTML toolbar as target attitude generator. STK is polled each time step, and both the reference attitude data and the true attitude data are sent out from it.


Attitude_Control - STK 10 - [3D Graphics 1 - Earth]
File Edit View Insert Analysis Scenario Utilities Window Help
Object Browser
Body Z
Ref Y
Ref X
Body X
Ref Z
3D Graphic... 2D Graphic... Graph: Tru... Graph: Tru...
HTML Viewer - Reference Attitude Settings
Inertially Fixed (Euler)
Sun Pointing
YPR (VVLH)
Earth Pointing
Timeline View 1
15 Sep 201
Attitude_Control AvailabilityIntervals
Attitude_Control
X Real Time Multiplier: 1.00

```
True Attitude State
q1:                  0.000000
q2:                  0.000000
q3:                  0.000264
q4:                  1.000000
wx (deg/sec):        0.000000
wy (deg/sec):        0.000000
wz (deg/sec):        7.200017
w mag (deg/sec):     7.200017
A (deg):                0.030
B (deg):                0.000
C (deg):                0.000
Yaw (321) (deg):        7.196
Pitch (321) (deg):     68.999
Roll (321) (deg):    -157.137
```

```
Reference Attitude State
q1:                   0.799189
q2:                  -0.162522
q3:                   0.543866
q4:                   0.197718
wx (deg/sec):         0.000000
wy (deg/sec):        -0.061402
wz (deg/sec):        -0.000007
w mag (deg/sec):      0.061402
A (deg):                -7.166
B (deg):               -68.999
C (deg):               157.137
Yaw (321) (deg):        -0.000
Pitch (321) (deg):       0.000
Roll (321) (deg):       -0.000
```

A(deg)	0	B(deg)	0	C(deg)	0
Yaw(deg)	0	Pitch(deg)	0	Roll(deg)	0

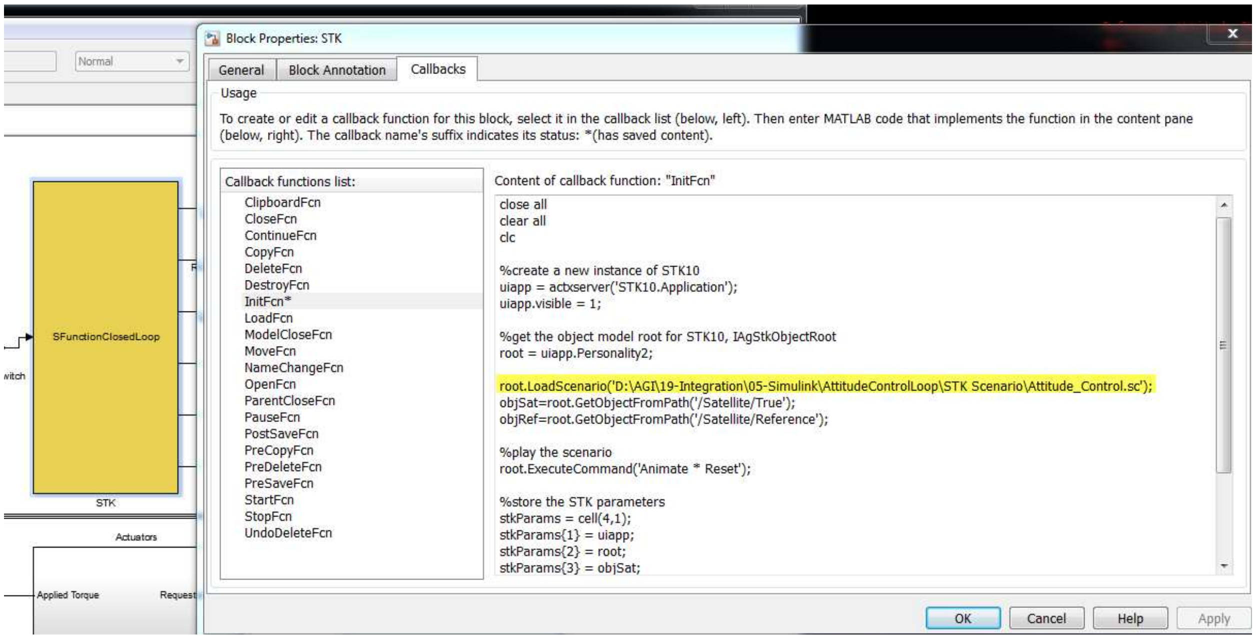
Fig 11 – STK Scenario

As example four different attitude profiles are provided:

1. An *Inertially Fixed* profile: here the user can change the A, B and C (Euler) angles respect to the J2000 reference system by manually changing their values and then pressing the Inertially Fixed button;
2. A *YPR* attitude profile (respect to the VVLH reference system): here the user can modify the Yaw, Pitch and Roll angles versus the orbital reference. This attitude profile is normally used when operating EO satellite.
3. A Sun pointing profile, where the satellite orients its –Z body axes towards the Sun. There is also a spinning rate around the Z body axis to obtain some inertial rigidity along the pointing axes;
4. An Earth pointing profile, where the satellite is pointing its +Z body axes towards the Earth's center.

Additional reference profiles can be defined directly from the STK attitude property page by editing the Multi Segment attitude profile elements.

Before running the simulation be sure that the scenario path contained into the startup callback is the correct one, otherwise you'll get an exception from the Simulink code:



To create or edit a callback function for this block, select it in the callback list (below, left). Then enter MATLAB code that implements the function in the content pane (below, right). The callback name's suffix indicates its status: *(has saved content).

Content of callback function: "InitFcn"

```
close all
clear all
clc

%create a new instance of STK10
uiapp = actxserver('STK10.Application');
uiapp.visible = 1;

%get the object model root for STK10, IAgStkObjectRoot
root = uiapp.Personality2;

root.LoadScenario('D:\AGI\19-Integration\05-Simulink\AttitudeControlLoop\STK Scenario\Attitude_Control.sc');
objSat=root.GetObjectFromPath('/Satellite/True');
objRef=root.GetObjectFromPath('/Satellite/Reference');

%play the scenario
root.ExecuteCommand('Animate * Reset');

%store the STK parameters
stkParams = cell(4,1);
stkParams{1} = uiapp;
stkParams{2} = root;
stkParams{3} = objSat;
```

Once the simulation is started from Simulink, the STK time will be slaved to the Simulink clock. By default this is a discrete time step with 0.1 sec. as integration interval; this allows to capture dynamics that are in the order of tents of second.

2.7.1 – Case Studies

Pointing Error at Nadir

Run the Simulink model and press the YPR (VVLH) button with 0, 0, 0 as Yaw, Pitch and Roll value; this means that we want a nadir pointing attitude, with the spacecraft's X body axes in the direction of the velocity.

If the initial attitude is different from the nadir pointing wait for the True satellite to reach the target attitude and run the *Angular Speed* graph that is available as quick report. Note as, after the transient phase, the angular speed values are very little and stationary over time.

At this point, go to the Navigation block and change the Noise Power value of the Band-Limited White Noise block (I've put it equal to one). By doing this, we are adding attitude estimation uncertainties since we may be interested in seeing which are the effects in terms of pointing errors.

The white noise insertion has the effect that the Guidance systems continuously outputs commanded torques to correct errors, and this has an effect on both YPR angles and angular speeds. In particular the angular errors lead to an error distance between the nominal and the effective point spotted an Earth.

Update the *Angular Speed* graph and run the *Pointing Error* one. The results should be like this:

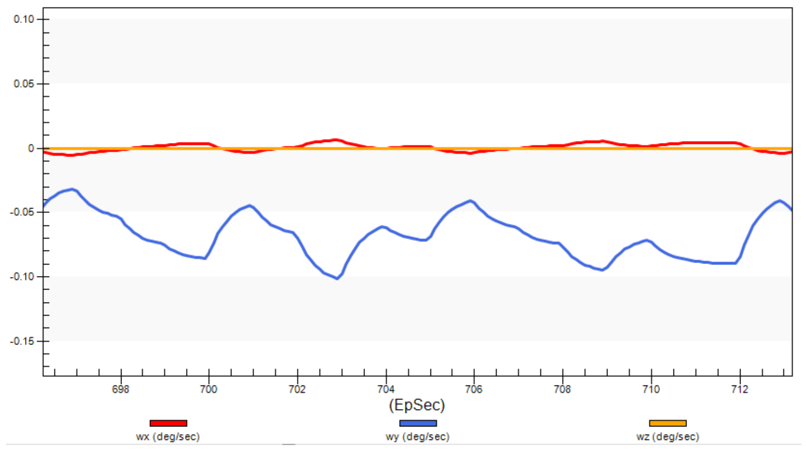


Fig 12 – Angular speeds after white noise injection

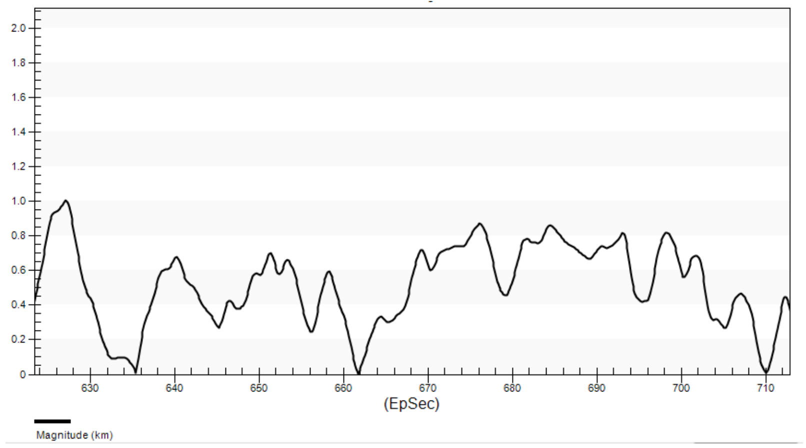


Fig 13 – Pointing errors after white noise injection

Testing the Satellite's Attitude Changes

As in the previous case, run the Simulink model and press the YPR (VVLH) button with 0, 0, 0 as Yaw, Pitch and Roll value to get a nadir pointing attitude.

Wait for the True satellite to reach the target attitude and change the YPR state to 0, 0, -30. This will set the satellite 30 deg. right looking (max roll angle). When the new attitude is reached, change it again to get the min roll angle (left looking – 0, 0, -30) and then finally back to the nadir pointing. The resulting YPR angles and angular rates should look like this:

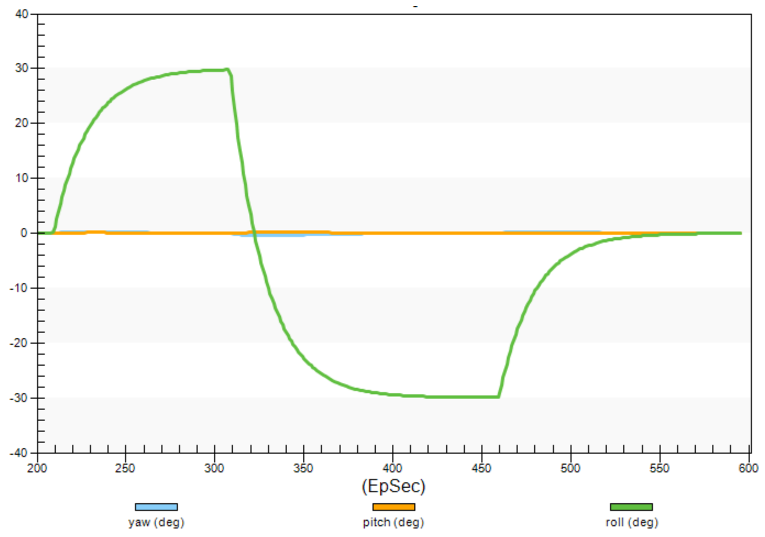


Fig 14 – YPR angles

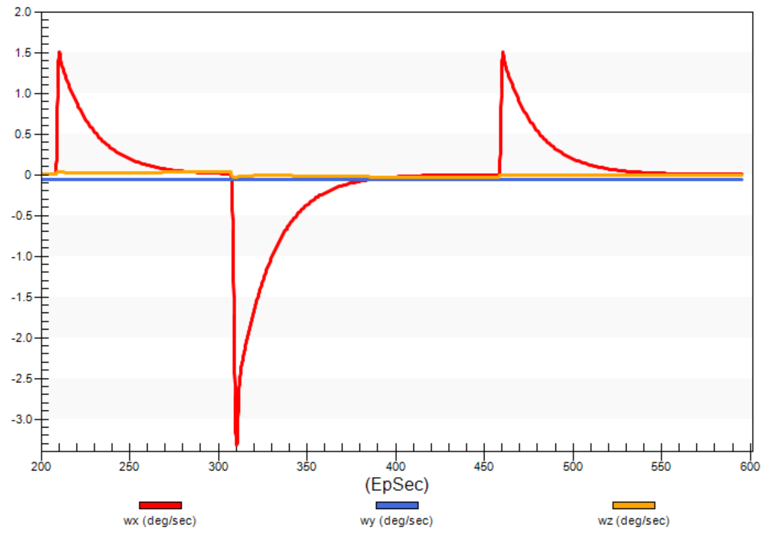


Fig 15 – Pointing errors after white noise injection

3 – Attitude Dynamics and Kinematics

3.1 - Euler Equations

For a rigid body the following equation applies:

$$\boldsymbol{H} = I\boldsymbol{\omega} \tag{1}$$

, where $\boldsymbol{H}$ is the angular momentum, $\boldsymbol{\omega}$ the angular velocity vector and I is the inertia matrix. Both $\boldsymbol{H}$ and $\boldsymbol{\omega}$ are expressed in the body frame. We also define the inertia matrix as

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{zx} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}$$

I is real, symmetric and coordinate dependent matrix (also called tensor). When the axis are aligned with the principal axis of inertial it can be expressed as

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

If we apply a torque $\boldsymbol{T}$ to a free-floating object it will change its spin rate. For a rigid body, a reference system centered in the center of mass (CM) frame, and ω resolved in the body axis frame the following is valid:

$$\boldsymbol{T} = {}^{d\boldsymbol{H}}\!/_{dt} = \dot{\boldsymbol{H}} + \omega \times \boldsymbol{H} \tag{2}$$

From this equation we can get the following scalar relations in a body fixed, principal axis center of mass frame:

$$\begin{aligned} \dot{H}_x &= I_{xx}\dot{\omega}_x = T_x + \left(I_{yy} - I_{zz}\right)\omega_y\omega_z \\ \dot{H}_y &= I_{yy}\dot{\omega}_y = T_y + (I_{zz} - I_{xx})\omega_z\omega_x \\ \dot{H}_z &= I_{zz}\dot{\omega}_z = T_z + \left(I_{xx} - I_{yy}\right)\omega_x\omega_y \end{aligned} \tag{3}$$

Those equations are called *Euler equations*. No general solution exists, but there are particular solutions for simple torques.

The simplest case is a free body, with no applied external torques. In this case the Euler equations can be written as:

$$I_{xx}\dot{\omega}_x + \left(I_{zz} - I_{yy}\right)\omega_y\omega_z = 0$$
$$I_{yy}\dot{\omega}_y + (I_{xx} - I_{zz})\omega_z\omega_x = 0 \qquad (4)$$
$$I_{zz}\dot{\omega}_z + \left(I_{yy} - I_{xx}\right)\omega_x\omega_y = 0$$

With no external torques the kinetic energy is constant and equals to:

$$E = \frac{1}{2}\left[I_x\omega_x^2 + I_y\omega_y^2 + I_z\omega_z^2\right]$$

The magnitude of angular momentum is also constant:

$$M^2 = I_x^2\omega_x^2 + I_y^2\omega_y^2 + I_z^2\omega_z^2$$

3.2 – Attitude Geometry

There are several ways to describe the orientation of a rigid body respect to a reference system: (direction cosine, Euler angles, quaternion) whit a minimum of three parameters required. Euler angles and quaternion are here defined.

3.2.1 – Euler Angles

Any orientation can be achieved by composing three elemental rotations. The elemental rotations can either occur about the axes of the fixed coordinate system (in this case they are also known as YPR angles) or about the axes of a rotating coordinate system, which is initially aligned with the fixed one, and modifies its orientation after each elemental rotation (intrinsic rotations). The latter case we are dealing with Euler angles.

The Euler angles are called as φ, θ, ψ and in this case we consider the 321 sequence, so we start rotating the reference frame of an angle ψ around the Z axis, then we rotate the rotated frame of an angle θ around the Y' axis and then we rotate of an angle φ around the X" axis, where XYZ, X'Y'Z' and X"Y"Z" are three consecutive frames obtained by rotating three times the reference frame.

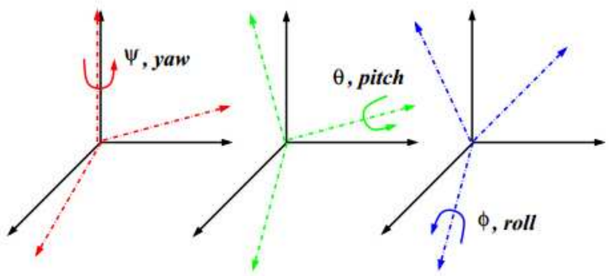


Fig 2 –Euler Angles

The rotation matrix between a starting (reference) frame and final one is written as:

$$\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} \cos(\theta)\cos(\psi) & \cos(\theta)\sin(\psi) & -\sin(\theta) \\ \sin(\varphi)\sin(\theta)\cos(\psi) - \cos(\varphi)\sin(\psi) & \sin(\varphi)\sin(\theta)\sin(\psi) + \cos(\varphi)\cos(\psi) & \sin(\varphi)\cos(\theta) \\ \cos(\varphi)\sin(\theta)\cos(\psi) + \sin(\varphi)\sin(\psi) & \cos(\varphi)\sin(\theta)\sin(\psi) - \sin(\varphi)\cos(\psi) & \cos(\varphi)\cos(\theta) \end{bmatrix} \begin{bmatrix} E_1 \\ E_2 \\ E_3 \end{bmatrix} \tag{5}$$

Where $\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}$ is the final frame after a 321 sequence and $\begin{bmatrix} E_1 \\ E_2 \\ E_3 \end{bmatrix}$ is the starting frame.

2.2.2 - Quaternions

The Euler's theorem says:

The orientation of a body is uniquely specified by a vector giving the direction of a body axis and a scalar specifying a rotation angle about the axis.

So, to univocally specify a certain rotation about a reference frame we need to state 4 parameters. In mathematics, the quaternions are a number system that extends the complex numbers. They are defined in a four-dimensional vector space over the real numbers: $q \in R^4$:

$$q = q_1 i + q_2 j + q_3 k + q_4 \tag{6}$$

$q_1 i + q_2 j + q_3 k$ is the vector part of the quaternion and is purely imaginary ($i^2 = j^2 = z^2 = -1$), whereas q_4 is the real (scalar part). The vector part defines a vector in the imaginary space that have components u_x, u_y, u_z.

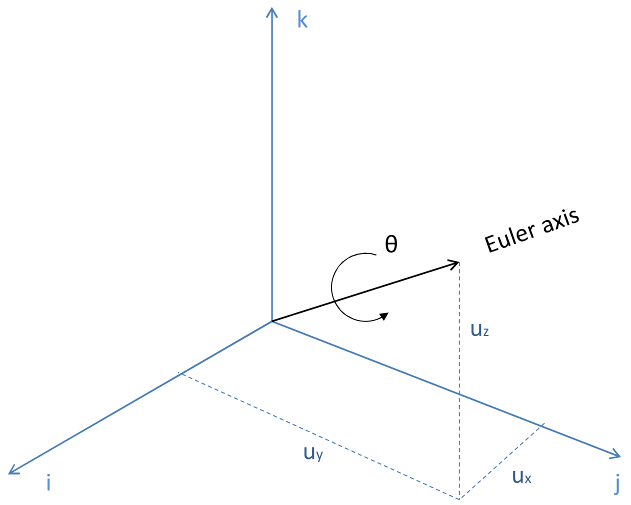


Fig 3 - Quaternion elements in 3D space

If we assume that the rotation is θ we can write (Euler's formula):

$$q = (u_x i + u_y j + u_z k) * sin(^{\vartheta}/_2) + \ cos(^{\vartheta}/_2)$$

Or, in a different notation:

$$\begin{aligned} q_1 &= u_x * sin(^{\vartheta}/_2) \\ q_2 &= u_y * sin(^{\vartheta}/_2) \\ q_3 &= u_z * sin(^{\vartheta}/_2) \\ q_4 &= cos(^{\vartheta}/_2) \end{aligned} \tag{7}$$

The magnitude of a quaternion is defined as $|q| = \sqrt{{q_1}^2 + {q_2}^2 + {q_3}^2 + {q_4}^2}$. Unit quaternions ($|q| = 1$), also known as versors, provide a convenient mathematical notation for representing orientations and rotations of objects in three dimensions.

The identity quaternion (no rotation – initial orientation) is written as:

$$[q_1 \quad q_2 \quad q_3 \quad q_4] = [0 \ \ 0 \ \ 0 \ \ 1]$$

Other important (normal) quaternions are $[1 \ \ 0 \ \ 0 \ \ 0]$, $[0 \ \ 1 \ \ 0 \ \ 0]$ and $[0 \ \ 0 \ \ 1 \ \ 0]$ that represent Euler axis along principal axis with no rotation.

The conjugate of the quaternion q is written as:

$$q^* = -q_1 i - q_2 j - q_3 k + q_4$$

So it has an inverted vectorial part. The inverse of a quaternion is its conjugate normalized:

$$q^{-1} = \frac{q^*}{|q|}$$

The conjugate and the inverse quaternion are the same for a unit quaternion.

3.3 - Kinematics

The kinematic equations relate the angular velocity along the body axes and the orientation of the satellite respect to the inertial frame. They can be written as:

$$\frac{d}{dt}\begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0 & \omega_z & -\omega_y \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix}\begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \tag{8}$$

The rotational kinematics can also be expressed in terms of angular velocity vector and Euler angles:

$$\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \begin{bmatrix} -\dot{\psi}\sin(\theta) + \dot{\varphi} \\ \dot{\psi}\cos(\theta)\sin(\varphi) + \dot{\theta}\cos(\varphi) \\ \dot{\psi}\cos(\theta)\cos(\varphi) - \dot{\theta}\sin(\varphi) \end{bmatrix} \tag{9}$$

Conversely, the rate of change of Euler angles can be expressed in terms of the components of the angular velocity vector as:

$$\begin{bmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} \omega_y \sec(\theta)\sin(\varphi) + \omega_z \sec(\theta)\cos(\varphi) \\ \omega_y \cos(\varphi) - \omega_z \sin(\varphi) \\ \omega_x + \omega_y \tan(\theta)\sin(\varphi) + \omega_z \tan(\theta)\cos(\varphi) \end{bmatrix} \tag{10}$$

Note as, for this sequence, there is a singularity for $\theta=\frac{\pi}{2}$. In this case the $\dot{\psi}$ value is not defined.

3.4 – Feedback Laws

Feedback laws implementation is needed when we'd like to automatically control the attitude state of the satellite over time.

By using closed loop control laws we are defining commanded torques whose aim is to maintain the satellite attitude regime as close as possible respect to the reference one.

Many different control laws can be used, according with the dynamic state of the satellite and the intended goals. Here we define a couple of simple control laws that can be implemented when, for example, we build a Simulink model with a closed control loop on it.

It is assumed that the dynamics is described by the equations (3).

3.4.1 - Spacecraft Angular Rate only

If we want to control the spacecraft angular rate only, the following asymptotic condition shall be satisfied:

$$(\omega_x, \omega_y, \omega_z) \to (\omega_{xr}, \omega_{yr}, \omega_{zr})\ as\ t \to \infty$$

The control laws that accomplish these objectives can be written as:

$$\begin{aligned} T_x &= -(I_{yy} - I_{zz})\omega_y\omega_z - K_{p1}(\omega_x - \omega_{xr}) + I_{xx}\dot{\omega}_{xr} \\ T_y &= -(I_{zz} - I_{xx})\omega_z\omega_x - K_{p2}(\omega_y - \omega_{yr}) + I_{xx}\dot{\omega}_{yr} \\ T_z &= -(I_{xx} - I_{yy})\omega_x\omega_y - K_{p3}(\omega_z - \omega_{zr}) + I_{zz}\dot{\omega}_{zr} \end{aligned} \tag{11}$$

,where K_{pi} are the proportional gains.

3.4.2 - Spacecraft Angular Rate and Attitude

In this case we want to control both the angular rates and the attitude (Euler angles or quaternions) over time The conditions to be satisfied are the following:

$$(\omega_x, \omega_y, \omega_z) \to (\omega_{xr}, \omega_{yr}, \omega_{zr})\ as\ t \to \infty$$
$$([q_1 \quad q_2 \quad q_3 \quad q_4]) \to ([q_{1r} \quad q_{2r} \quad q_{3r} \quad q_4r])\ as\ t \to \infty$$

The control law used in this case is:

$$\begin{bmatrix} T_x \\ T_y \\ z \end{bmatrix} = 2 * K_p * \begin{bmatrix} q_{1E}q_{4E} \\ q_{2E}q_{4E} \\ q_{3E}q_{4E} \end{bmatrix} + K_d * \begin{bmatrix} \omega_{1E} \\ \omega_{1E} \\ \omega_{1E} \end{bmatrix} \tag{12}$$

, where the quaternion error is defined as (q_s= current quaternion, q_{ref}= reference quaternion)

$$q_E = q_s^{-1}q_{ref}$$

And the angular velocity error is

$$\omega_e = \omega - \omega_{ref}$$

We have proportional and derivative gains in this case to satisfy both the conditions.

Annex A – S-Function

The S-Function block allows to run MATLAB code while the Simulink plant is running (the code is actually executed at each time step). In this case we're going to use the Level-2 MATLAB S-Function.

S-Function Initialization

Each S-Function comes with a set of available callbacks (*right click->Properties…->Callbacks*). Here we focus on the *InitFcn* callback, which is executed just once when the Simulink plant is started.

A typical InitFcn could be:

```
close all
clear all
clc

%create a new instance of STK10
uiapp = actxserver('STK10.Application');
uiapp.visible = 1;

%get the object model root for STK10, IAgStkObjectRoot
root = uiapp.Personality2;

%load an already existing STK scenario and get control of a satellite
root.LoadScenario('C:\Test\TestScenario.sc');
objSat=root.GetObjectFromPath('/Satellite/Satellite1);

%reset the scenario
root.ExecuteCommand('Animate * Reset');

%store the STK parameters
stkParams = cell(4,1);
stkParams{1} = uiapp;
stkParams{2} = root;
stkParams{3} = objSat;
stkParams{4} = objRef;

set_param(gcb, 'UserData', stkParams);

%change the unitsettings in STK
root.ExecuteCommand('SetUnits / EpSec');
root.UnitPreferences.SetCurrentUnit('DateFormat', 'EpSec');
```

Once defined some variables, we can put them directly into the S-Function code by setting a vector called *stkParams* and filling it with the variables we need. The *set_param* routine assures that the data are correctly forwarded into the S-Function code.

S-Function Code

The S-Function code is split into many different pieces, each of them being responsible for a small amount of activities. As first, a function named as the file name shall be declared:

```
function SFunctionNoSensors(block)
  setup(block);
```

then the actual *setup* routine is executed:

```
function setup(block)

  %% Register number of input and output ports
  block.NumInputPorts  = 1;
  block.NumOutputPorts = 6;

  % Override input port properties
  block.InputPort(1).Dimensions  = 3;
  block.InputPort(1).DatatypeID  = 0;  % double
  block.InputPort(1).Complexity  = 'Real';
  block.InputPort(1).DirectFeedthrough = true;

  % Override output port properties
  block.OutputPort(1).Dimensions  = 4;
  block.OutputPort(1).DatatypeID  = 0;  % double
  block.OutputPort(1).Complexity  = 'Real';
  block.OutputPort(2).Dimensions  = 3;
  block.OutputPort(2).DatatypeID  = 0;  % double
  block.OutputPort(2).Complexity  = 'Real';
  block.OutputPort(3).Dimensions  = 3;
  block.OutputPort(3).DatatypeID  = 0;  % double
  block.OutputPort(3).Complexity  = 'Real';
  block.OutputPort(4).Dimensions  = 4;
  block.OutputPort(4).DatatypeID  = 0;  % double
  block.OutputPort(4).Complexity  = 'Real';
  block.OutputPort(5).Dimensions  = 3;
  block.OutputPort(5).DatatypeID  = 0;  % double
  block.OutputPort(5).Complexity  = 'Real';
  block.OutputPort(6).Dimensions  = 3;
  block.OutputPort(6).DatatypeID  = 0;  % double
  block.OutputPort(6).Complexity  = 'Real';

  %% Set block sample time
  block.SampleTimes = [0.1 0];

  %% Register methods
  block.RegBlockMethod('Start', @Start);
  block.RegBlockMethod('Outputs', @Output);
  block.RegBlockMethod('SetInputPortSamplingMode', @SetInpPortFrameData);
  block.RegBlockMethod('PostPropagationSetup',    @DoPostPropSetup);
```

The *setup* routine does the following:

1) Registers how many input and output we have in the block;
2) Overrides the default properties by the definition of explicit properties for each input/output signal;
3) Sets the block sample time to a particular rate; in this example we have a 10Hz process;
4) Registers the methods that are being used in the S-Function.

The *PostPropagationSetup* method initializes one DWork vector with the name Euler321:

```
function DoPostPropSetup(block)

block.NumDworks = 1;

block.Dwork(1).Name            = 'Euler321';
block.Dwork(1).Dimensions      = 3;
block.Dwork(1).DatatypeID      = 0;      % double
block.Dwork(1).Complexity      = 'Real'; % real
block.Dwork(1).UsedAsDiscState = true;
```

The *SetInpPortFrameData* method configures the sample time of the input and output ports:

```
function SetInpPortFrameData(block, idx, fd)

  block.InputPort(idx).SamplingMode = fd;
  block.OutputPort(1).SamplingMode  = fd;
  block.OutputPort(2).SamplingMode  = fd;
  block.OutputPort(3).SamplingMode  = fd;
  block.OutputPort(4).SamplingMode  = fd;
  block.OutputPort(5).SamplingMode  = fd;
  block.OutputPort(6).SamplingMode  = fd;
```

The Start method defines the initial state of the input vector:

```
function Start(block)
%define a null initial state
block.Dwork(1).Data = [0, 0, 0];
```

Finally, the *Output* method is used to get information from STK and to send them out of the S-Function block:

```
function Output(block)

%retreive the stkParameters array from the UserData
stkParameters = get_param(block.BlockHandle, 'UserData');

%get the root and the access objects from the array
root = stkParameters{2};
sat = stkParameters{3};
ref = stkParameters{4};

%update the feedback - get the current Euler angles from the input
block.Dwork(1).Data = block.InputPort(1).Data;
yaw = block.Dwork(1).Data(1);
pitch = block.Dwork(1).Data(2);
roll = block.Dwork(1).Data(3);

%change the animation to the next step
%%%% with this line of code we can actually take control of STK time and slave it to the Simulink clock
root.CurrentTime = block.CurrentTime;

%set the satellite attitude according with the received inputs
sat.Attitude.AddEuler(root.CurrentTime, '321', yaw, pitch, roll);

%get the Euler attitude of the body frames respect to the ICRF frame
satBodyAxes = sat.Vgt.Axes.Item('Body');
satAttitudeEuler = satBodyAxes.FindInAxes(root.CurrentTime,
sat.Vgt.Axes.Item('ICRF')).Orientation.QueryEulerAnglesArray('e321');
refBodyAxes = ref.Vgt.Axes.Item('Body');
refAttitudeEuler = refBodyAxes.FindInAxes(root.CurrentTime,
ref.Vgt.Axes.Item('ICRF')).Orientation.QueryEulerAnglesArray('e321');

%%%% get the reference attitude state %%%%%%%%
% define a data provider and the elements to get from it
referenceState = ref.DataProviders.Item('Attitude Quaternions');
Elems = {'q1';'q2';'q3';'q4';'wx';'wy';'wz'};
% execute the query and get back the results
Results1 = referenceState.ExecElements(root.CurrentTime, root.CurrentTime, 10, Elems);
q1 = cell2mat(Results1.DataSets.GetDataSetByName('q1').GetValues);
q2 = cell2mat(Results1.DataSets.GetDataSetByName('q2').GetValues);
q3 = cell2mat(Results1.DataSets.GetDataSetByName('q3').GetValues);
q4 = cell2mat(Results1.DataSets.GetDataSetByName('q4').GetValues);
wx = cell2mat(Results1.DataSets.GetDataSetByName('wx').GetValues);
wy = cell2mat(Results1.DataSets.GetDataSetByName('wy').GetValues);
wz = cell2mat(Results1.DataSets.GetDataSetByName('wz').GetValues);
%refAttitudeState = [q1(1,1), q2(1,1), q3(1,1), q4(1,1), wx(1,1), wy(1,1), wz(1,1)];
refQuaternion = [q1(1,1), q2(1,1), q3(1,1), q4(1,1)];
refAngularSpeed = [wx(1,1), wy(1,1), wz(1,1)];

%%%% get the current attitude state %%%%%%%%
% define a data provider and the elements to get from it
currentState = sat.DataProviders.Item('Attitude Quaternions');
Elems = {'q1';'q2';'q3';'q4';'wx';'wy';'wz'};
% execute the query and get back the results
Results2 = currentState.ExecElements(root.CurrentTime, root.CurrentTime, 10, Elems);
q1 = cell2mat(Results2.DataSets.GetDataSetByName('q1').GetValues);
q2 = cell2mat(Results2.DataSets.GetDataSetByName('q2').GetValues);
q3 = cell2mat(Results2.DataSets.GetDataSetByName('q3').GetValues);
```

```
q4 = cell2mat(Results2.DataSets.GetDataSetByName('q4').GetValues);
wx = cell2mat(Results2.DataSets.GetDataSetByName('wx').GetValues);
wy = cell2mat(Results2.DataSets.GetDataSetByName('wy').GetValues);
wz = cell2mat(Results2.DataSets.GetDataSetByName('wz').GetValues);
%satAttitudeState = [q1(1,1), q2(1,1), q3(1,1), q4(1,1), wx(1,1), wy(1,1), wz(1,1)];
satQuaternion = [q1(1,1), q2(1,1), q3(1,1), q4(1,1)];
satAngularSpeed = [wx(1,1), wy(1,1), wz(1,1)];

%%%%%%%%%%%%%%%% set the outputs %%%%%%%%%%%%%%%%%%%%%%%
block.OutputPort(1).Data = refQuaternion;
block.OutputPort(2).Data = refAngularSpeed;
block.OutputPort(3).Data = cell2mat(refAttitudeEuler);
block.OutputPort(4).Data = satQuaternion;
block.OutputPort(5).Data = satAngularSpeed;
block.OutputPort(6).Data = cell2mat(satAttitudeEuler);
```

Annex B – Code Snippets

Start STK and get Object References

```
%create an instance of STK10
uiapp = actxserver('STK10.Application');
uiapp.visible = 1;
%get the object model root for STK10
root = uiapp.Personality2;
```

Create a New Scenario

```
%Check if a scenario exists, close and start a new scenario
if isempty(root.CurrentScenario)
    root.NewScenario('ScenarioName')
else
    root.CloseScenario;
    root.NewScenario('ScenarioName')
end
```

Load an Existing Scenario

```
root.LoadScenario('D:\...your path…\Scenario1.sc');
```

Set The Scenario Analysis Time

```
%set units before setting scenario time period
root.UnitPreferences.Item('DateFormat').SetCurrentUnit('UTCG'); %or
%set scenario time period and animation period
root.CurrentScenario.SetTimePeriod('1 Sep 2013 04:00:00.000', '2 Sep 2013 04:00:00.000');
root.CurrentScenario.Epoch = '1 Sep 2013 04:00:00.000';
```

Change the Measurement Units

```
%using connect
root.ExecuteCommand('SetUnits / EpSec');
%using COM
root.UnitPreferences.SetCurrentUnit('DateFormat', 'EpSec');
```

Get Control of STK Objects

```
objSat1=root.GetObjectFromPath('/Satellite/Satellite1');
objSensor1=root.GetObjectFromPath('/Satellite/Satellite1/Sensor/Sensor1');
```

Create and Propagate a Satellite

```
% create the satellite
satObj = root.CurrentScenario.Children.New('eSatellite', 'Sat');
%set the propagator
satObj.Propagator.InitialState.Representation.AssignClassical(...
    'eCoordinateSystemJ2000', 7000, 0.1, 53.4, 0, 0, 0);
% define propagation times and propagate
satObj.Propagator.StartTime = '25 May 2013 12:00:00.000';
satObj.Propagator.StopTime  = '25 May 2013 15:00:00.000';
satObj.Propagator.Propagate;
```

Get the Euler Angles from a Satellite

```
satObj=root.GetObjectFromPath('/Satellite/Satellite1);
satBodyAxes = satObj.Vgt.Axes.Item('Body');
satAttitudeEuler =
satBodyAxes.FindInAxes(root.CurrentTime,satObj.Vgt.Axes.Item('ICRF')).Orientation.QueryEulerAnglesArray('e321')
;
```

DATA PROVIDERS

GET LAT/LON FROM A SATELLITE

```
% define a data provider and the elements to get from it
LLAState = satObj.DataProviders.Item('LLA State').Group.Item('Fixed');
Elems = {'Time';'Lat';'Lon'};
% define the time span for a variable data provider
satStartTime = satObj.Propagator.EphemerisInterval.FindStartTime;
satStopTime = satObj.Propagator.EphemerisInterval.FindStopTime;
% execute the query and get back the results
Results = LLAState.ExecElements(satStartTime, satStopTime, 10, Elems);
time = cell2mat(Results.DataSets.GetDataSetByName('Time').GetValues);
Lat  = cell2mat(Results.DataSets.GetDataSetByName('Lat').GetValues);
Long = cell2mat(Results.DataSets.GetDataSetByName('Lon').GetValues);
```

GET THE ATTITUDE QUATERNIONS

```
satObj=root.GetObjectFromPath('/Satellite/Satellite1');
% define a data provider and the elements to get from it
state = satObj.DataProviders.Item('Attitude Quaternions');
Elems = {'q1';'q2';'q3';'q4';'wx';'wy';'wz'};
% execute the query for the current time and get back the results
Results = state.ExecElements(root.CurrentTime, root.CurrentTime, 10, Elems);
q1 = cell2mat(Results.DataSets.GetDataSetByName('q1').GetValues);
q2 = cell2mat(Results.DataSets.GetDataSetByName('q2').GetValues);
q3 = cell2mat(Results.DataSets.GetDataSetByName('q3').GetValues);
q4 = cell2mat(Results.DataSets.GetDataSetByName('q4').GetValues);
wx = cell2mat(Results.DataSets.GetDataSetByName('wx').GetValues);
wy = cell2mat(Results.DataSets.GetDataSetByName('wy').GetValues);
wz = cell2mat(Results.DataSets.GetDataSetByName('wz').GetValues);
```

EXTRACTING ELEMENTS WITH PRE-DATA

```
%get a sensor object from the root
objSensor1=root.GetObjectFromPath('/Satellite/Satellite1/Sensor/Sensor1');
%define the data provider to report about Sun vector
sensorVect = objSensor1.DataProviders.Item('Vector Choose Axes');
dataProvSun = sensorVect.Group.Item('Sun');
%Choose the referense system you want to report the Sun data
dataProvSun.PreData = '/Satellite/Satellite1/Sensor/Sensor1 Body';
rptElems = {'Time';'RightAscension';'Declination'};
```

ACCESS

GET AER DATA

```
%get the objects for access calculation
objSun = root.GetObjectFromPath('/Planet/Sun');
objSensor1=root.GetObjectFromPath('/Satellite/Satellite1/Sensor/Sensor1');
%calculate the access
sunAccess = objSensor1.GetAccessToObject(objSun);
%get the access data
AER = sunAccess.DataProviders.Item('AER Data').Group.Item('BodyFixed').Exec(currentTime,currentTime,60);
azimuth = cell2mat(AER.DataSets.Item(2).GetValues);
elevation = cell2mat(AER.DataSets.Item(3).GetValues);
%%%%%%%% NOTE THAT IF THE ACCESS IS FROM A MOVING OBJECT BY DEFAULT THE Az/El DATA ARE RELEVANT TO THE OBJECT
VELOCITY VECTOR- BODY FIXED HAS TO BE SPECIFIED %%%%%%%%%%%%%%%%%%%%%%%%%%%
```

VGT

CREATE A FIXED VECTOR

```
% get the satellite object
objSat=root.GetObjectFromPath('/Satellite/Probe1');

% get the Vector Geometry Tool Interface
vgtSat = objSat.vgt;

%Returns a provider associated with the specified object
provider = root.CentralBodies.Earth.Vgt.Axes.Item('MeanEclpJ2000');
```

```
%define a new vector (type = Fixed in Axes)
VectFactory = vgtSat.Vectors.Factory;
fixedAxesVector = VectFactory.Create('VectorTest','','eCrdnVectorTypeFixedInAxes');
fixedAxesVector.ReferenceAxes.SetAxes(provider);
fixedAxesVector.Direction.AssignXYZ(0,0,1);
```