



How to create a custom Thruster Set using Engine Plugin

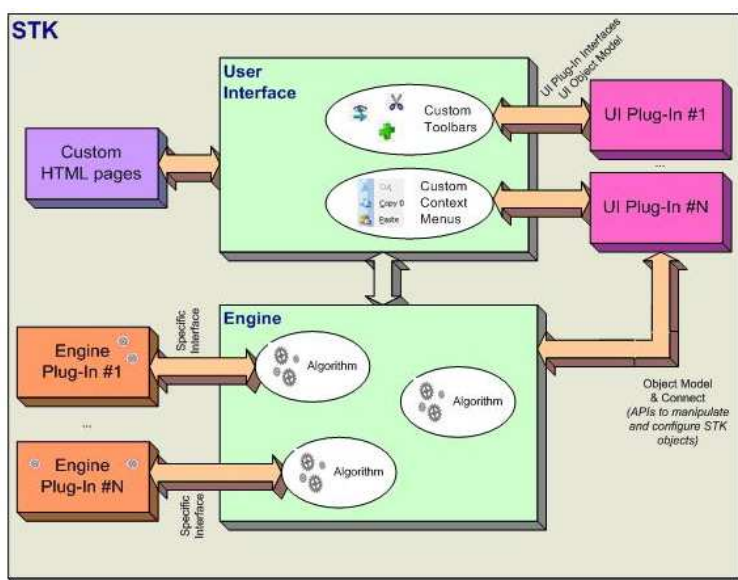
Version	1.0
Author	Giuseppe Corrao
Date	20 January 2016
STK ver.	11.0.0

Contents

Introduction	3
1 – Addressing the problem.....	4
1.1 – Engines and Satellite Specifications	4
1.2 – Orbit Specification.....	5
1.2 – Maneuver Strategy Specification	6
2 – Creating a Valid Plugin	7
2.1 - Create a new Solution as Class Library	7
2.2 – Configure the Class for COM.....	8
2.3 – Add COM References	9
2.4 – Add GUID and ProgID	10
2.5 - Implement the IAgGatorPluginEngineModel Interface	11
3 - Customizing the interface.....	12
3.1 – Add Global Variables.....	12
3.2 – Setup <i>Init</i> method	13
3.3 – Setup <i>Evaluate</i> method.....	13
3.4 – Setup <i>Free</i> method.....	14
3.5 – Setup <i>PreNextStep</i> and <i>PrePropagate</i> methods	14
4 – Create the PlusY Class	14
5 – Create the MinusY Class.....	17
6 – Register the plugin	20
6.1 – STK Registration	20
6.2 – Windows Registration	21
7 – Create a Suitable STK scenario	23

In computing, a plugin is a software component that adds a specific feature to an existing software application: when an application supports plug-ins, it enables customization. AGI provides a variety of ways to extend STK. These extensibility mechanisms can be divided into two distinct areas:

- 1) **User Interface extensibility:** allow users to create custom graphical user interfaces (GUI) and controls for STK to provide user-defined workflows which can combine STK and in-house functionality
- 2) **Engine extensibility:** allow users to customize just those aspects of the modeling that are really non-generic while leveraging all the other generic capabilities of the COTS software.



Engine plugins provide the capability for users to incorporate customer-specific non-generic modeling into computations. A plugin component is a user-supplied software component called by the application at certain pre-defined event times within the computation cycle. The plugin is allowed to modify the computation by adding additional considerations or modifying parameters. STK offers specific plugin points for most computations. The interfaces and information available to plugins depend on the particular plugin point.

Engine plugins can be either *COM based* (using .Net, C/C++, Perl, VBScript etc.) or *script based* (Perl, MATLAB or VBScript). COM based engine plugin components provide a capability for users to incorporate customer-specific non-generic modeling into computations. An engine plugin component is a software component built using Microsoft's COM technology that is called by the application at certain pre-defined event times within the computation cycle. The plugin is allowed to modify the computation by adding additional considerations or modifying parameters. Here we focus on COM based plugin, using VS2012 as developing environment and C# as programming language.

1 – Addressing the problem

The tutorial goal is to define and implement a custom thruster set in STK to model a continuous, low thrust transfer orbit from an inclined LEO parking orbit up to the GEO orbit. Since we want to apply a custom strategy for the transfer orbit, we need a plugin implementing both the engines specifications and the strategy itself (i.e. a thrust profile along time). To get the result several steps are required, but basically we have those two main activities to carry out

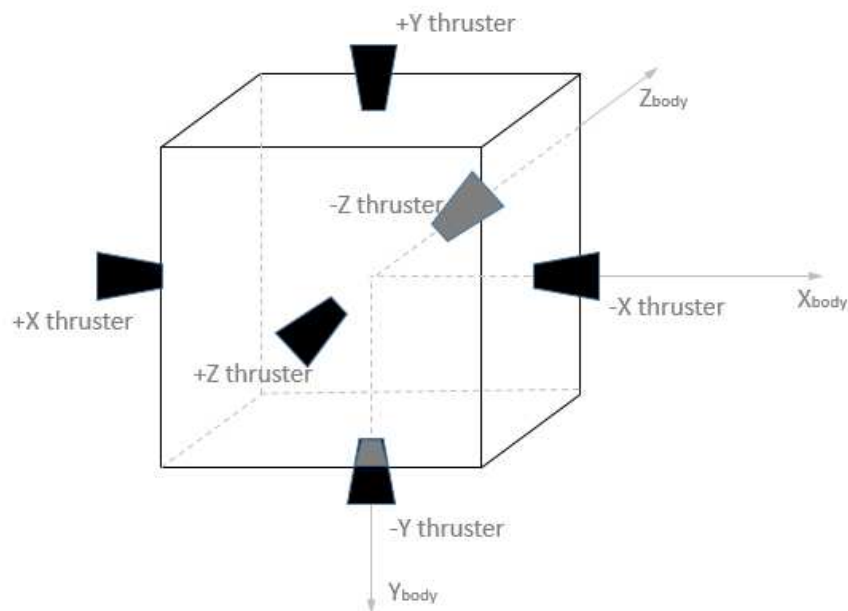
- 1) To write and register the COM plugin.
- 2) To properly configure the Component Browser and the Astrogator MCS in STK.

A step by step tutorial is here provided to have the plugin up and running in STK 11, ready for further enhancements and customizations.

1.1 – Engines and Satellite Specifications

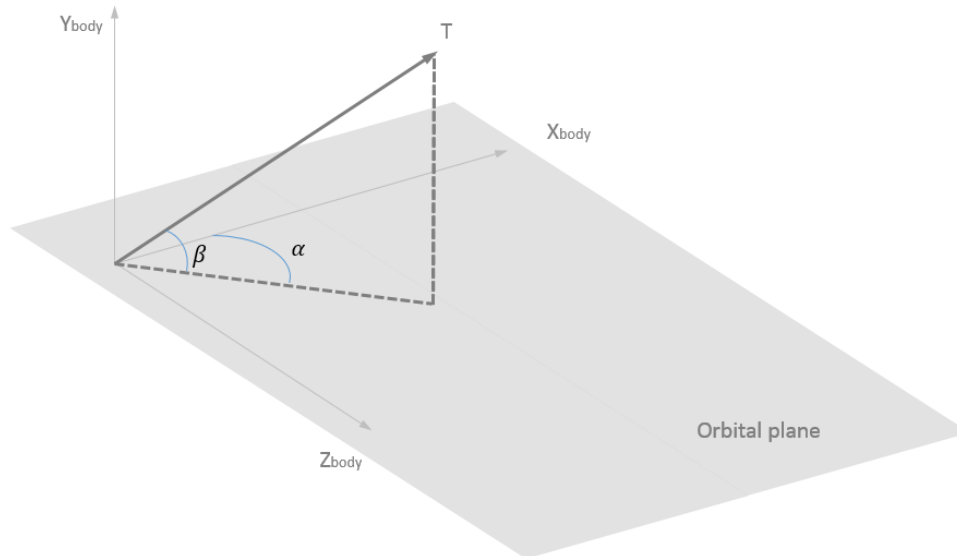
Our thruster set is potentially composed by 6 different engines, firing along the three main directions in body axis. The reason why we need 6 engines instead of three (just one in any body direction) is that we assume a constant attitude profile along the entire maneuver, so there will be no attitude changes to re-orient the spacecraft.

Engine name is related with the direction of thrust. Engines layout is the following:



Engines have 0.95 N as maximum thrust and 3250 sec as Isp.

We assume the satellite attitude profile to be Nadir alignment with ECI velocity constraint (the default in STK), so we have the Z_{body} axis pointed towards the Earth's center, the X_{body} axis constrained along the velocity vector and the Y_{body} axis orthogonal to the orbital plane. The thrust profile we'd like to implement is relevant to this reference system.



By combining the instantaneous thrust for each engine we get a resulting thrust that is a vector pointing in a direction identified by two angles:

- α , the angle between the X_{body} axis and the thrust component along the orbital plane and
- β , the angle between the orbital plane and the thrust vector.

α and β can be computed by STK, and the resulting thrust can be projected to any reference frame, allowing the user to define the custom thrust profile along different frames (e.g. inertial).

1.2 – Orbit Specification

Our starting orbit is the following:

- Semimajor Axis = 7000 km
- Eccentricity = 0
- Inclination = 28.5
- RAAN = 0
- Argument of Periapsis = 0
- True Anomaly = 0

The target orbit is GEO.

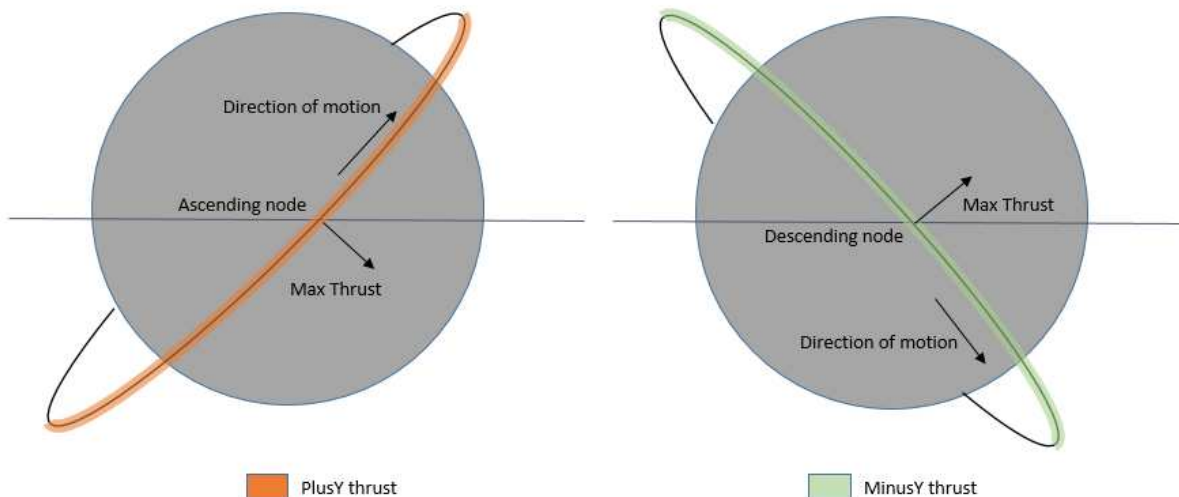
1.2 – Maneuver Strategy Specification

When we configure each engine with a custom thrust profile we can perform complex maneuvers that involve the modification of more than one orbital parameter at time. This tutorial is focused on the technical aspects about the plugin deployment/implementation, and not on the maneuver strategy optimization: we are going to use a pretty basic and easy strategy to get our goal, but the user can implement its own to model a very efficient transfer orbit.

Since both starting and target orbits are circular, we need to change just semimajor axis and inclination, not the eccentricity. To accomplish this, we actually don't need to thrust in the $\pm Z$ direction (if you decide to model those engines for completeness, their thrust should be zero all time) and in $-X$ direction. Under this assumptions the α angle shall always be zero (no thrust component in the radial direction); β varies with a sinusoidal law in the XY plane according with the following logic:

- The $+X$ thruster is constantly firing along the velocity direction with full thrust.
- The $+Y$ thruster is only firing during the ascending portion of the orbit with a sinusoidal profile having maximum at the ascending node.
- The $-Y$ thruster is only firing during the descending portion of the orbit with a sinusoidal profile having maximum at the descending node.
- The $+X$ thruster stopping condition is Semimajor Axis \geq SMA target.
- The $\pm Y$ thrusters stopping condition is inclination = 0 within a certain threshold.

Conditions b) and c) are represented in the Figure below:



$\pm Y$ thrusts vary according with the cosine of the Argument of Latitude.

2 – Creating a Valid Plugin

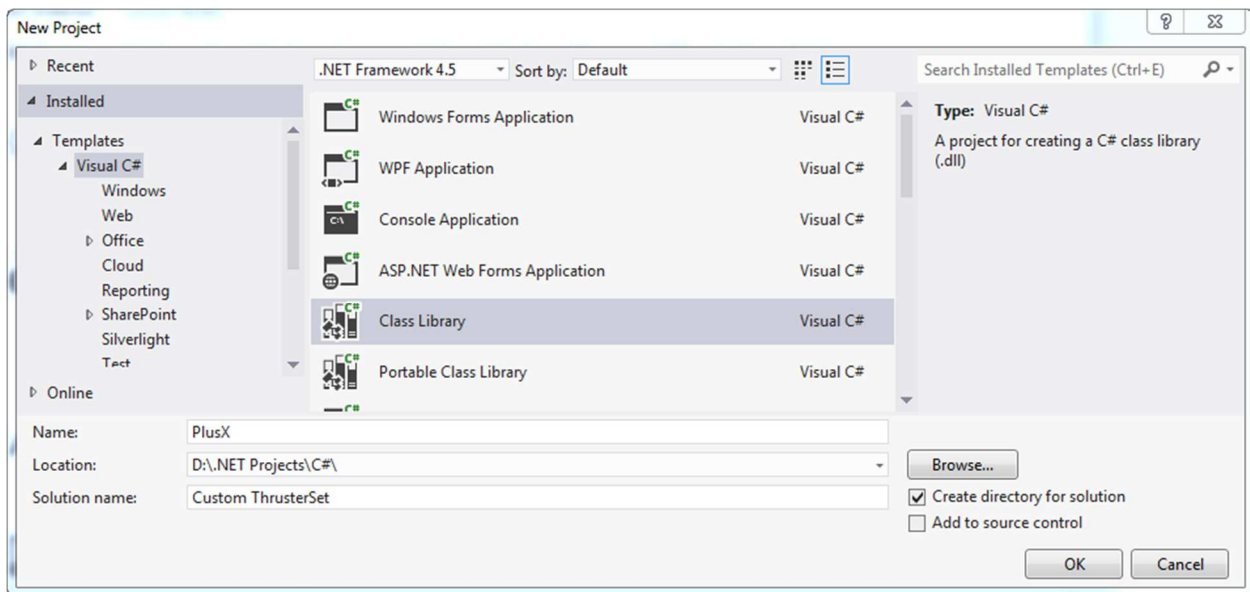
In this section we focus on the creation of a valid Astrogator Engine Model Plugin. This basically requires the following steps:

- 1) Create a new collection of class libraries using Visual Studio.
- 2) Connect the plugin to STK using registration file.

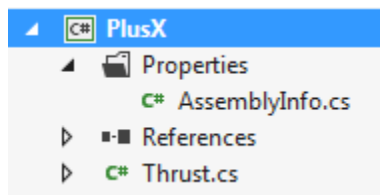
Here we'll use VS 2012 as developing environment. We'd like to create a single Solution containing different projects (defined as class libraries), one for each of the engine. This way we can operate on any individual engine independently from the others.

2.1 - Create a new Solution as Class Library

- 1) Run VS 2012 as Administrator and create a new Class Library. Set *CustomThrusterSet* as Solution Name and *PlusX* as class Name:



- 2) Press OK, then right click the *Class1.cs* item in the Solution Explorer and rename it as *Thrust*.



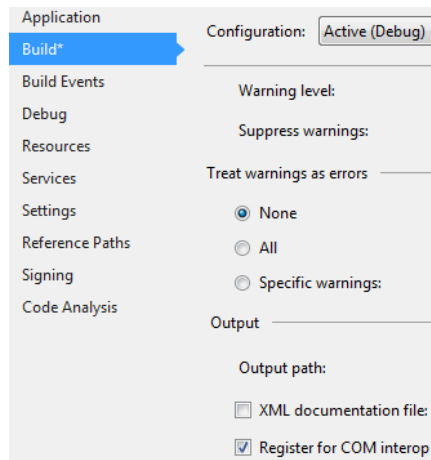
- 3) Click the *Save All* button to save all the files in the new project.

This will give us the opportunity to start working on our first engine model, as explained in the following Sections.

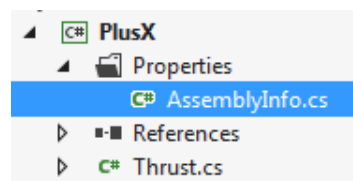
2.2 – Configure the Class for COM

We need to ensure that the types in the assembly are visible and accessible to COM components. This is done by registering the class for COM interoperability and setting the *ComVisible* attribute to true for the entire assembly:

- 1) Right click the *PlusX* item in the Solution Explorer and choose Properties. Go to the Build tab and check *Register for Com Interop* flag.



- 2) Expand the Properties folder in the Solution Explorer and open *AssemblyInfo.cs*:



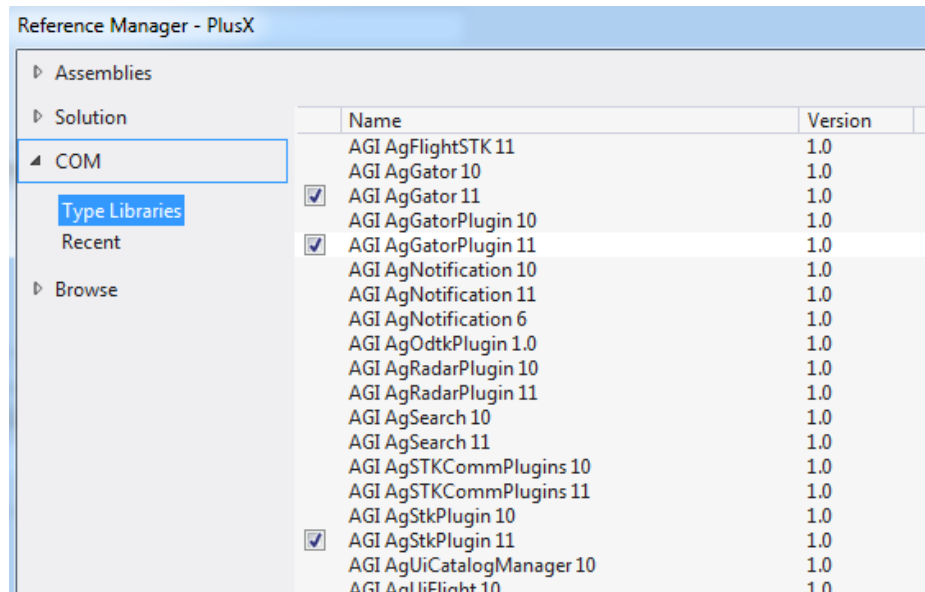
- 3) Locate the *ComVisible* attribute and set it to true:

```
// Setting ComVisible to false makes the types in this assembly not visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(true)]
```

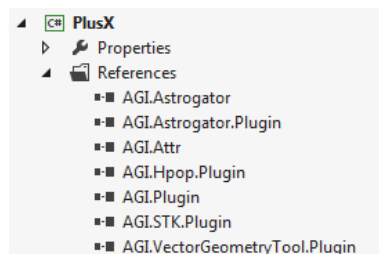
- 4) Save and close *AssemblyInfo.cs*.

2.3 – Add COM References

- 1) Right-click References in the Solution Explorer and select *Add Reference*.
- 2) Select *AGI AgGator 11*, *AGI AgGatorPlugin 11* and *AGIStkPlugin 11*. Those three libraries contain the methods we need to get connected to STK 11 using COM:



- 3) When you click OK, the following items (namespace) are added to your project references:



- 4) Add using directive for those AGI namespaces, as well as the System.Runtime.InteropServices namespace (which promotes interaction with external type libraries):

```
using AGI.Plugin;
using AGI.STK.Plugin;
using AGI.Astrogator;
using AGI.Astrogator.Plugin;
using System.Runtime.InteropServices;
```

The *Thrust.cs* class should appear like this:

```

PlusX* Thrust.cs*
PlusX.Thrust
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 using AGI.Attr;
8 using AGI.Plugin;
9 using AGI.STK.Plugin;
10 using AGI.Astrogator;
11 using AGI.Astrogator.Plugin;
12 using System.Runtime.InteropServices;
13
14 namespace PlusX
15 {
16     public class Thrust
17     {
18     }
19 }
20
    
```

Thrust.cs is the only class for this (and for the following ones) namespace. We still need to define our project as a COM component.

2.4 – Add GUID and ProgID

Add the following class attributes to your class (in namespace area before class):

```

namespace PlusX
{
    [Guid("<GUID>")]
    [ProgId("<ProgID>")]
    [ClassInterface(ClassInterfaceType.None)]

    public class Thrust
    {
    ...
    }
}
    
```

, where:

- <GUID> is a **Globally Unique IDentifier**, used to uniquely identify your component in the Windows registry. On the Visual Studio Tools menu, use the Create GUID function to generate a new GUID for your plugin, while Visual Studio Express Edition does not have a built-in GUID generation tool.

There are many free GUID generators available online (<https://www.guidgenerator.com/> , www.guidgen.com/). Here some GUIDs:

```

8b7cf9ae-7966-4970-8b70-712f05a0172b
62a95e66-646a-4722-8c5d-9b5697b26f2c
7a4a2ac8-1e33-43a2-8865-22a43c55c4b4
27eba233-d94c-42ad-b493-8fb5ef67e61a
c62018e7-050b-42e1-843d-6bdc9b1dcc0f
69d9972a-c93c-4027-9bab-c75651b84046
1f67dbdc-49d3-45d1-b14d-66e8262b02fe
c0bdb412-3858-4433-8382-67c18f8fe167
a768eba1-dabf-4705-9cc6-2bb4c8473b77
94cc07b9-93f2-4d0d-94d3-def5ba5f2785
    
```

How to create a custom Thruster Set

- <ProgID> is a unique string used to identify your plugin to STK. We here put the thruster name as identifier:

When you finish, your GUID and ProgID should resemble this format:

```

14 namespace PlusX
15 {
16     [Guid("b040438e-c17d-40c9-b404-f485f1599f81")]
17     [ProgId("PlusX")]
18     [ClassInterface(ClassInterfaceType.None)]
19
20     public class Thrust
21     {
22     }
23 }
24
  
```






2.5 - Implement the IAgGatorPluginEngineModel Interface

The next step is to change the class to implement the `IAgGatorPluginEngineModel` interface. This interface is used to create a COM component that acts as an Astrogator Engine Model. The engine model plugin is used to set the thrust magnitude, Isp and mass flow rate of an engine. The plugin component will be called at certain event times defined by the interface (see the members of the interface). During these calls, the component may request input values and set output values that can affect the computation.

- 1- After the class name, add a colon, then add the interface name:

```
public class Thrust : IAgGatorPluginEngineModel
```

Right-click the interface (`IAgGatorPluginEngineModel`). Select *Implement Interface*, then *Implement Interface* again. When you implement the interface, its members will be exposed. IAgUiPlugin interface's members are the following:

 Evaluate	Triggered on every force model evaluation during the propagation of a step. Use the input interface to access engine model settings. Returning false will turn this callback off.
 Free	Triggered just before the plugin is freed from use to allow for any additional cleanup.
 Init	Triggered when the plugin is initialized to allow for any additional needed initialization. Must return true to turn on use of plugin.
 PreNextStep	Triggered just before the next propagation step is attempted. Use the input interface to access engine model settings. Returning false will turn this callback off.
 PrePropagate	Triggered just before propagation starts. Use the input interface to access engine model settings.

Having the interface implemented means that all the interface's members are ready to be properly configured in the class body:

How to create a custom Thruster Set

```

20 public class ThrustThrust : IAgGatorPluginEngineModel
21 {
22     public bool Evaluate(AgGatorPluginResultEvalEngineModel ResultEvalEngineModel)
23     {
24         throw new NotImplementedException();
25     }
26
27     public void Free()
28     {
29         throw new NotImplementedException();
30     }
31
32     public bool Init(IAgUtPluginSite Site)
33     {
34         throw new NotImplementedException();
35     }
36
37     public string Name
38     {
39         get { throw new NotImplementedException(); }
40     }
41
42     public bool PreNextStep(AgGatorPluginResultState ResultState)
43     {
44         throw new NotImplementedException();
45     }
46
47     public bool PrePropagate(AgGatorPluginResultState ResultState)
48     {
49         throw new NotImplementedException();
50     }
51 }
52
  
```

There is also a public property named Name to set the name of the plugin used in messages.

3 - Customizing the interface

3.1 – Add Global Variables

- 1) Add the following variables after the class definition:

```

public class Thrust : IAgGatorPluginEngineModel
{
    private IAgUtPluginSite _site = null;
    private AgGatorPluginProvider _gatorPrv = null;
    private AgGatorConfiguredCalcObject _semimajorAxis = null;
}
  
```

Site interfaces represent the services made available to the plugin by the application. Each application implements and extends the *IAgUtPluginSite* interface.

The Astrogator plugin provider allows a plugin to request a calculation object from the Component Browser.

The Calc Object will evaluate the SMA value at runtime.

3.2 – Setup *Init* method

- 1) Edit the *Init* method in this way:

```
public bool Init(IAgUtPluginSite Site)
{
    _site = Site;

    if (_site != null)
    {
        this._gatorPrv = ((IAgGatorPluginSite)(_site)).GatorProvider;

        if (_gatorPrv != null)
        {
            _semimajorAxis = _gatorPrv.ConfigureCalcObject("Semimajor_Axis");
            return true;
        }
    }

    return false;
}
```

This will assure that the GatorProvider method returns data. The *_semimajorAxis* variable is feed by the Semimajor_Axis calculation object in STK.

3.3 – Setup *Evaluate* method

- 1) Edit the Evaluate method in this way:

```
public bool Evaluate(AgGatorPluginResultEvalEngineModel ResultEvalEngineModel)
{
    if (ResultEvalEngineModel != null)
    {
        double thrust = 0.089;
        double isp = 1650;
        double semimajorAxis = _semimajorAxis.Evaluate(ResultEvalEngineModel);

        if (semimajorAxis >= 42164)
        {
            // target SMA reached
            thrust = 0;
        }

        ResultEvalEngineModel.SetThrustAndIsp(thrust, isp);
    }
}
```

Here we just push constant values (thrust and isp) for the engine model using the *SetThrustAndIsp* method. If the target SMA is reached, the engine stops firing.

3.4 – Setup *Free* method

- 1) Edit the *Free* method by leaving it empty:

```
public void Free()  
{  
}
```

3.5 – Setup *PreNextStep* and *PrePropagate* methods

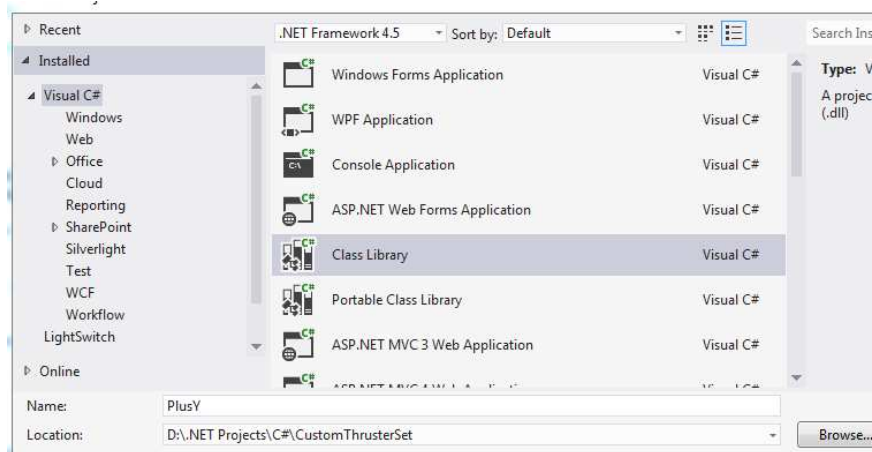
- 1) Edit the methods this way:

```
public bool PreNextStep(AgGatorPluginResultState ResultState)  
{  
    return true;  
}  
  
public bool PrePropagate(AgGatorPluginResultState ResultState)  
{  
    return true;  
}
```

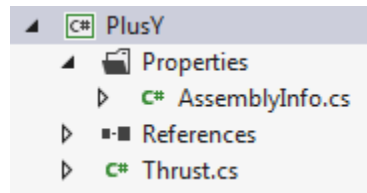
4 – Create the PlusY Class

Well, we just finished configuring the engine that thrust in the +X direction, but we still need two other engines to accomplish our goal (we only need three out of six engines for our strategy). The PlusY thruster follows a sinusoidal profile, so we need to change the *Evaluate* method to reflect the new situation. Let's do the follow:

- 1) Add a new project by right clicking the *Solution* item in the Solution Explorer. Choose Class Library as project type and name it PlusY:



- 2) Press OK, then right click the *Class1.cs* item in the Solution Explorer and rename it as *Thrust*:



- 3) Repeat steps from Section 2.2 to 3.1 (included) to configure the *PlusY* project with the following exceptions:

- a) Be sure that (Section 2.4) the GUID is different than the other ones and the ProgId is set to *PlusY*:

```
namespace PlusY
{
    [Guid("b040438e-c17d-44c9-b404-f415f1599f81")]
    [ProgId("PlusY")]
    [ClassInterface(ClassInterfaceType.None)]
}
```

- b) In Section 3.1 define two additional variables to get the argument of latitude and the inclination of the spacecraft over time from STK:

```
public class Thrust : IAgGatorPluginEngineModel
{
    private IAgUtPluginSite _site = null;
    private AgGatorPluginProvider _gatorPrv = null;
    private AgGatorConfiguredCalcObject _argOfLat = null;
    private AgGatorConfiguredCalcObject _inc = null;
```

- c) Instead of using the code in Section 3.2, use the following to get the required calculation objects:

```
public bool Init(IAGUtPluginSite Site)
{
    _site = Site;

    if (_site != null)
    {
        this._gatorPrv = ((IAGGatorPluginSite)(_site)).GatorProvider;

        if (_gatorPrv != null)
        {
            _argOfLat = _gatorPrv.ConfigureCalcObject("Argument_of_Latitude");
            _inc = _gatorPrv.ConfigureCalcObject("Inclination");

            if ((_argOfLat != null) && (_inc != null))
            {
                return true;
            }
        }
    }

    return false;
}
```

- a) Instead of using the code in Section 3.3, use the following to get the thrust level over time:

```
public bool Evaluate(AGGatorPluginResultEvalEngineModel ResultEvalEngineModel)
{
    if (ResultEvalEngineModel != null)
    {
        double thrust;
        double maxThrust = 0.089;
        double isp = 1650;
        double argOfLat = _argOfLat.Evaluate(ResultEvalEngineModel);
        double inc = _inc.Evaluate(ResultEvalEngineModel);

        thrust = maxThrust * (Math.Cos(argOfLat));
        if (thrust <= 0)
        {
            // only positive values for thrust are accepted by STK
            thrust = 0;
        }
        if (Math.Abs(inc * 180 / Math.PI) < 0.01)
        {
            // inclination is within threshold
            thrust = 0;
        }

        ResultEvalEngineModel.SetThrustAndIsp(thrust, isp);
    }

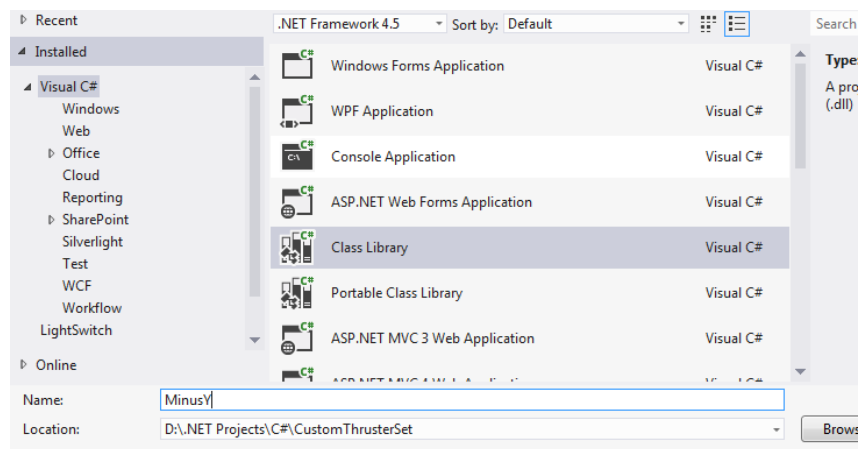
    return true;
}
```


This is a sinusoidal function of the Argument of Latitude, with inclination = 0 as stopping condition: until the inclination is positive, this engine will fire southward during the ascending portion of the orbit to reduce the inclination. Note that STK only accepts positive values of the thrust.

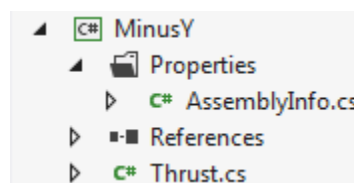
5 – Create the MinusY Class

This is the last engine to define in order to implement our basic thruster set. Let's do the follow:

- 1) Add a new project by right clicking the *Solution* item in the Solution Explorer. Choose Class Library as project type and name it PlusY:



- 2) Press OK, then right click the *Class1.cs* item in the Solution Explorer and rename it as *Thrust*:



- 3) Repeat steps from Section 2.2 to 3.1 (included) to configure the *MinusY* project with the following exceptions:

- a) Be sure that (Section 2.4) the GUID is different than the other ones and the ProgId is set to *PlusY*:

```
namespace MinusY
{
    [Guid("3d2d9468-a42a-47d3-b79e-a6a2988d4a85")]
    [ProgId("MinusY")]
    [ClassInterface(ClassInterfaceType.None)]
```

- b) In Section 3.1 define two additional variables to get the argument of latitude and the inclination of the spacecraft over time from STK:

```
public class Thrust : IAgGatorPluginEngineModel
{
    private IAgUtPluginSite _site = null;
    private AgGatorPluginProvider _gatorPrv = null;
    private AgGatorConfiguredCalcObject _argOfLat = null;
    private AgGatorConfiguredCalcObject _inc = null;
```

- c) In Section 3.2 set the code to get the required calculation objects:

```
public bool Init(IAgUtPluginSite Site)
{
    _site = Site;

    if (_site != null)
    {
        this._gatorPrv = ((IAgGatorPluginSite)(_site)).GatorProvider;

        if (_gatorPrv != null)
        {
            _argOfLat = _gatorPrv.ConfigureCalcObject("Argument_of_Latitude");
            _inc = _gatorPrv.ConfigureCalcObject("Inclination");

            if ((_argOfLat != null) && (_inc != null))
            {
                return true;
            }
        }
    }

    return false;
}
```

d) In Section 3.3 set the code to evaluate the Thrust:

```
public bool Evaluate(AgGatorPluginResultEvalEngineModel ResultEvalEngineModel)
{
    if (ResultEvalEngineModel != null)
    {
        double thrust;
        double maxThrust = 0.95;
        double isp = 3250;
        double argOfLat = _argOfLat.Evaluate(ResultEvalEngineModel);
        double inc = _inc.Evaluate(ResultEvalEngineModel);

        thrust = - maxThrust * (Math.Cos(argOfLat));
        if (thrust <= 0)
        {
            // only positive values for thrust are accepted by STK
            thrust = 0;
        }
        if (Math.Abs(inc * 180 / Math.PI) < 0.01)
        {
            // inclination is within threshold
            thrust = 0;
        }

        ResultEvalEngineModel.SetThrustAndIsp(thrust, isp);
    }

    return true;
}
```

This is a sinusoidal function of the Argument of Latitude (the same as the PlusY class, but with a minus sign before the function), with inclination = 0 as stopping condition: until the inclination is positive, this engine will fire northward during the descending portion of the orbit to reduce the inclination. Note that STK only accepts positive values of the thrust.

According with our custom thrust profile, the thruster set is now defined. If the remaining thruster are also needed, just create the remaining three classes as shown before.

6 – Register the plugin

Once the plugin is written and compiled, we ultimately need to do 2 additional steps

1. Register the .dll files with Windows.
2. Register the plugin with STK to notify the tool that it should look for your specific plugin.

There can be some confusion since both use the same terminology, but don't necessarily mean the same thing:

“Registering with Windows” is a process where registry keys are generated in the Windows registry (regedit). This essentially relates the GUID assigned in the code with the physical location on the machine where the .dll is located. Windows registration is accomplished by running regasm.exe from a command line or checking the “Register for COM Interop” flag in Visual Studio (more on these later).

“Registering for STK/” is the process of placing an XML in one of a handful of directories in order to alert STK that it should be looking for a specific plugin. The plugins' names that appear in the STK are related to the files registered with windows via the ProgID.

6.1 – STK Registration

This step is covered first because it is relatively simple, and it is easy to verify that it was successful. This step involves placing an .xml file in one of those three possible directory locations:

- 1- Single User Folder (usually *C:\Users\<username>\STK 11\Config\Plugins*).
Plugins registered here will be available only to the single user with access to this My Documents folder. Usually does not require administrative privileges.
- 2- All Users Folder (usually *C:\ProgramData\AGI\STK 11\Plugins*).
Plugins registered here will be available to all STK users on this machine. Usually does not require administrative privileges.
- 3- STK Install Folder (usually *C:\Program Files\AGI\STK 11\Plugins* for 64 bit install or *C:\Program Files (x86)\AGI\STK 11\Plugins* for 32 bit install).
Plugins registered here will be available to all STK users on this machine. Usually requires administrative privileges to save/modify files here.

On startup, STK searches these directories for .xml files that identify what plugins are available.

For the thruster set under consideration we need an XML file for each of the engine model (i.e. dll files) we compiled. Each of the XML file has the following structure:

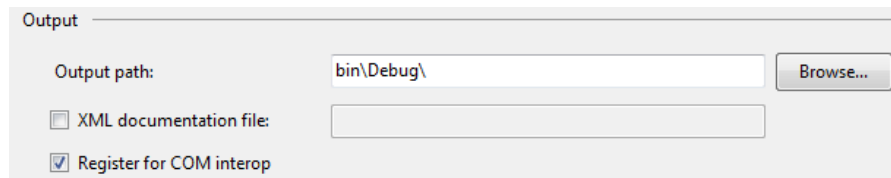
```
<?xml version = "1.0"?>
<AGIRegistry version = "1.0">
  <CategoryRegistry>
    <Category Name = "Engine Models">
      <Plugin DisplayName = "Custom Thruster Minus Y" ProgID = "MinusY"/>
    </Category>
  </CategoryRegistry>
</AGIRegistry>
```

, where the ProgID field shall match the ProgID of the relevant dll:

```
[Guid("3d2d9468-a42a-47d3-b79e-a6a2988d4a85")]
[ProgId("MinusY")]
[ClassInterface(ClassInterfaceType.None)]
```

6.2 – Windows Registration

There are two methods of registering a COM plugin with STK. The first method is applicable if you have the source code and are able to compile the solution/project using Visual Studio (or whatever IDE you are using). All you need to do is simply check the “Register for COM Interop” flag under the Project Properties/Build tab.



When the code is compiled, VS automatically registers the appropriate .dlls from the plugin with Windows. Note that you need Admin rights to do this. If you do not have privileges, you will get an error.

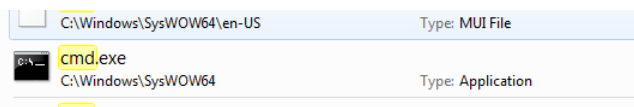
The second method is valid when you only have the .dll files. This requires manual registration and administrative privileges as well. Also you need to know which version of Microsoft .NET Framework was used to compile the plugin, and that version shall be installed in the target machine.

Most machines are x64 bit, so these instructions are catered for that.

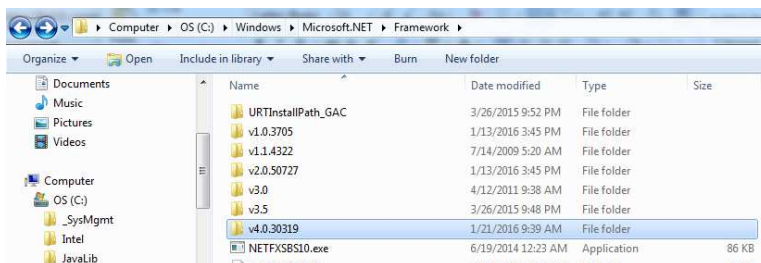
1. Navigate to the directory where you placed the .xml file. Create a folder specific to your plugin within the Plugins folder. Name it as “Custom Thruster Set” and copy the .dll files (one for each engine model) into this folder. Note you can pick any directory, however, it’s convenient to keep the .xml and the .dlls in the same location.

How to create a custom Thruster Set

2. In Windows Explorer, navigate to C:\Windows\SysWOW64. Run cmd.exe using the “Run as Administrator” option.



3. CD into C:\Windows\Microsoft.NET\Framework. There will be some folders called v#.#.###. These represent the available versions of .NET that are installed on your machine. CD into the version that the plugin was compiled against.

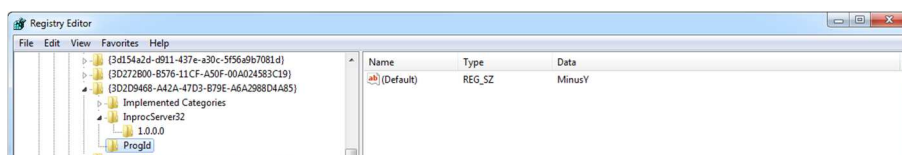
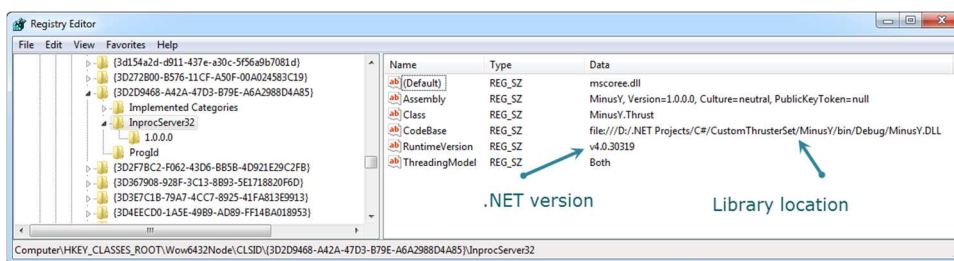


4. Run regasm /codebase "<filename>" where <filename> is the FULL path to the plugin .dll (file name included).

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319>regasm /codebase "D:\.NET Projects\C#\CustomThrusterSet\MinusY\bin\Debug\MinusY.dll"
```

5. You may get a warning regarding unsigned assemblies. This would be an issue if you have multiple plugins using the same .dll name, so beware.

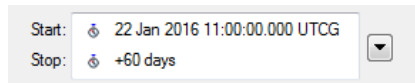
You can verify that the plugin was registered properly using Windows regedit.exe. If you happen to know the GUID, then simply search for the GUID code. Within that folder there should be a directory called InprocServer32. If you do not have the GUID, you can search for the ProgID instead. Recall the ProgID is listed in the .xml file:



7 – Create a Suitable STK scenario

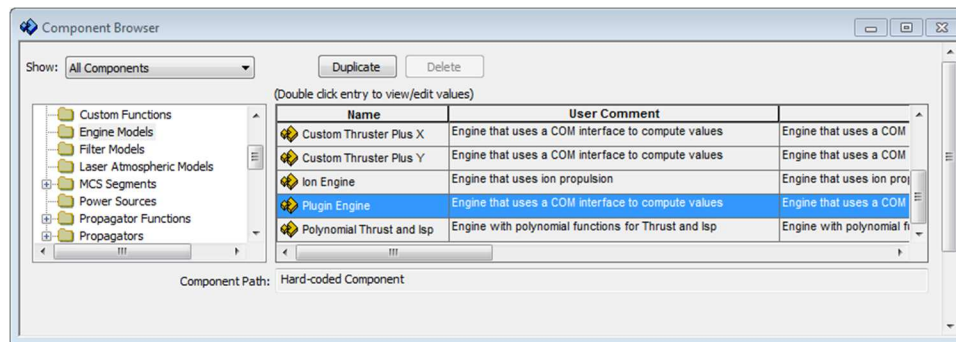
Now that we have our plugin set up and running, we need to create an STK scenario that actually use what we designed.

- 1) Create a new scenario and name it *CustomThrusterSet*. Give to it a duration of 2 months:

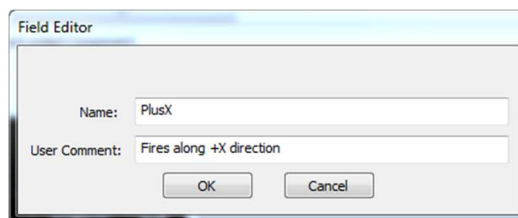


Start: 22 Jan 2016 11:00:00.000 UTCG
Stop: +60 days

- 2) Go in Utility -> Component Browser. Browse to the *Engine Model* folder and duplicate the Plugin Engine template:

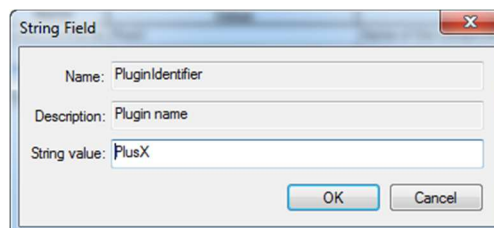


- 3) Name the component as *PlusX* and give it a description:



Name: PlusX
User Comment: Fires along +X direction

- 4) Press OK, then open the newly created component. As string value, put the ProgID of the plugin (PlusX in this case):

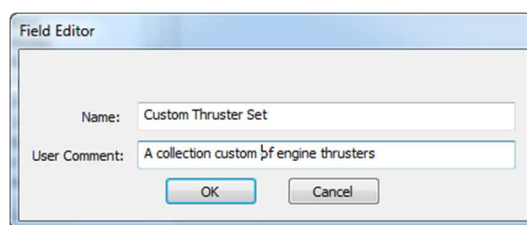


Name: PluginIdentifier
Description: Plugin name
String value: PlusX

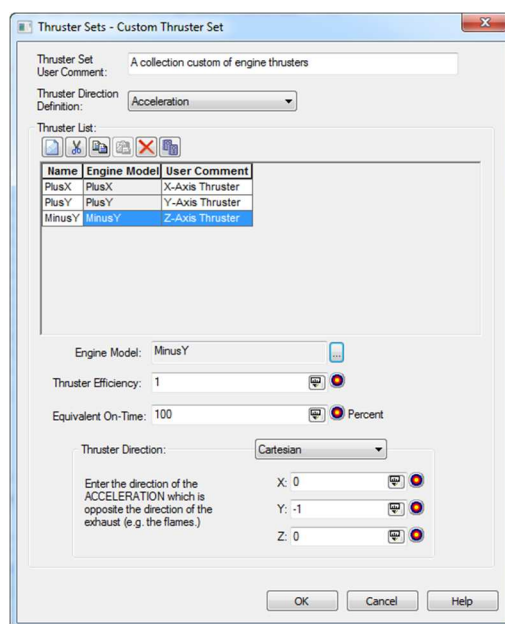
- 5) Do the same for the other 2 engine models, changing the engine name and its ProgID according with the engine to implement:

MinusY	Fires along -Y direction	Engine that uses a COM interface to
Plugin Engine	Engine that uses a COM interface to compute values	Engine that uses a COM interface to
PlusX	Fires along +X direction	Engine that uses a COM interface to
PlusY	Fires along +Y direction	Engine that uses a COM interface to

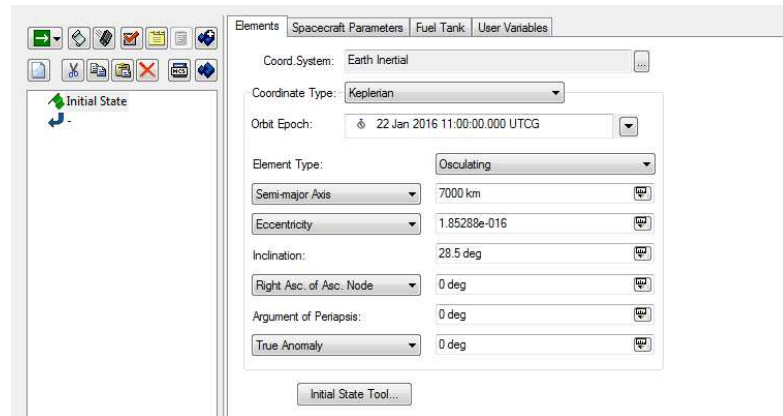
- 6) In the Component Browser, browse to the *Thruster Sets* folder. Choose the Thruster Set item and duplicate it. Change the name to *Custom Thruster Set*:



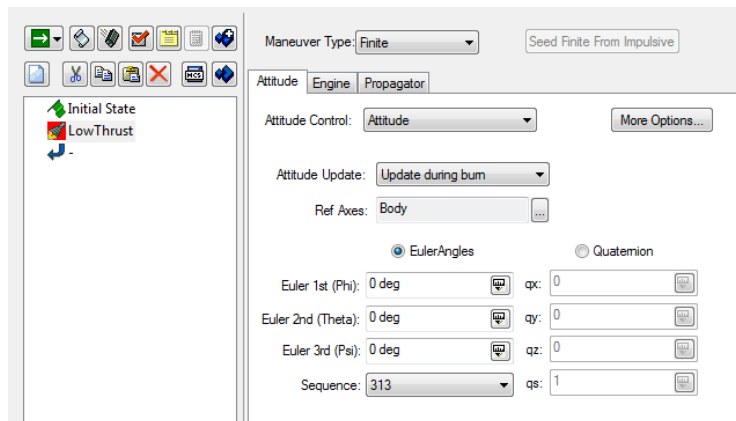
- 7) Press OK, then open the newly created component. By default, there will be three engines configured. For each engine change the name and the engine model fields. Also be sure that the thruster direction is compliant with the engine under consideration:



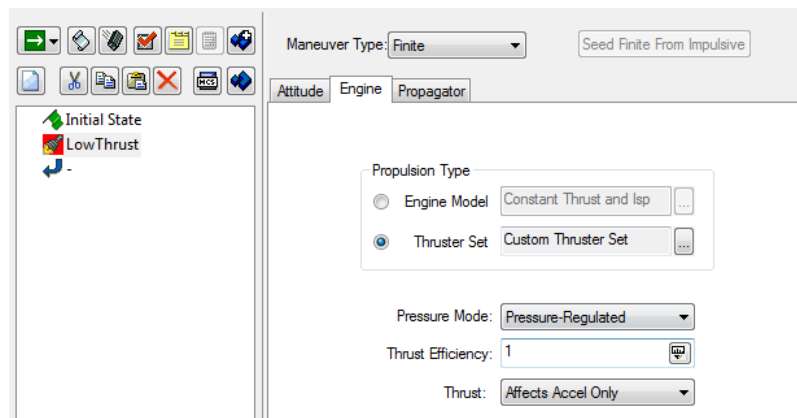
- 8) Create a new Astrogator satellite and name it *LowThrust*. In the MCS, remove the default *Propagate* segment and give the proper initial state to the satellite:



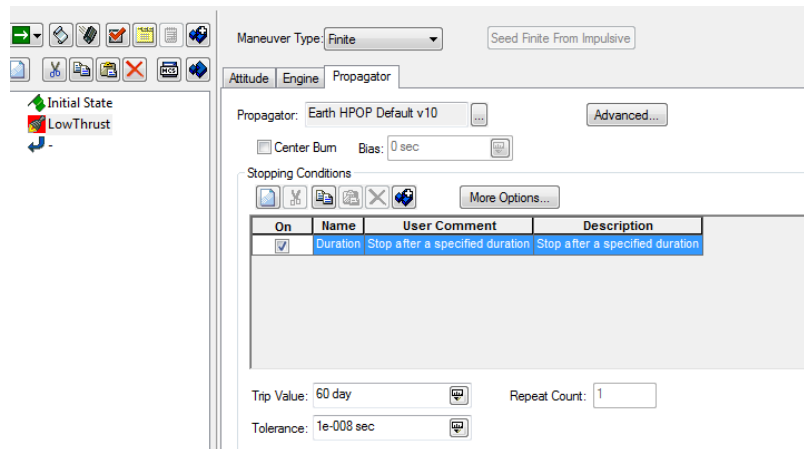
- 9) Add a new Maneuver segment and name it LowThrust. Set the Maneuver Type option to be Finite. Change the Attitude Control option to Attitude, then set Body as Ref Axes:



- 10) In the Engine tab, set Thruster Set as Propulsion Type and use the *Custom Thruster Set* component:



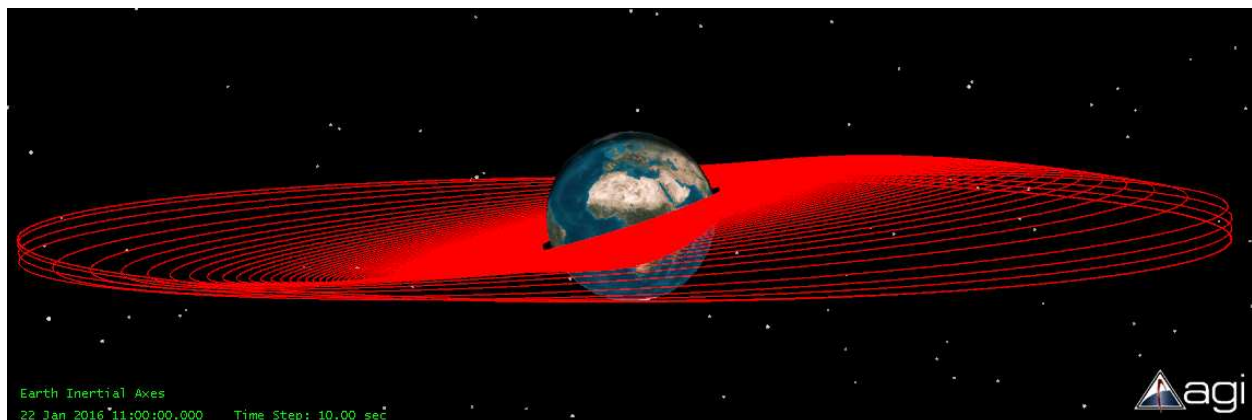
11) In the Propagator tab, set the Trip Value field to 60 days:



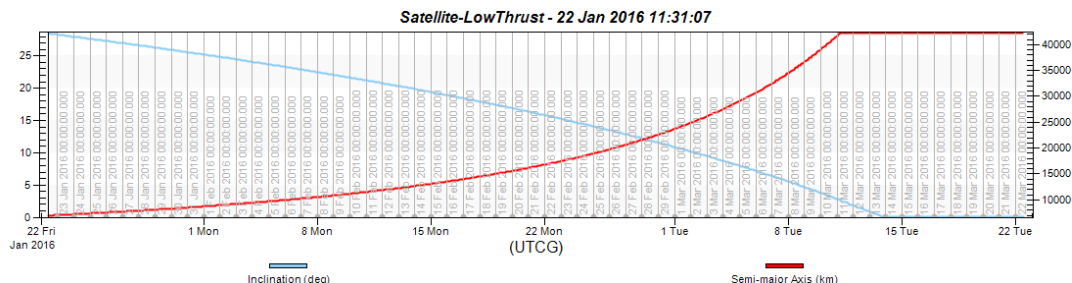
12) Apply and run the MCS; it will take some time to propagate. You can optionally check the Message Viewer to see if everything is up and running:

EngineModel plugin PlusX initialized and working.	Inform...	01-
EngineModel plugin PlusY initialized and working.	Inform...	01-
EngineModel plugin MinusY initialized and working.	Inform...	01-

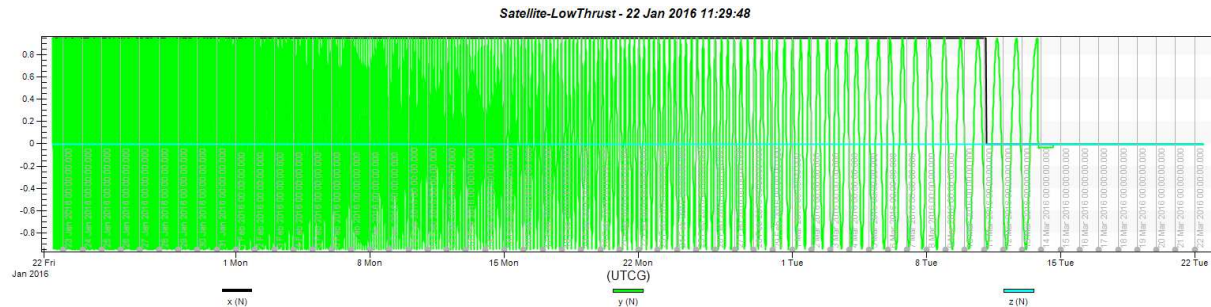
You should now see the low thrust trajectory in the 3D window:



After the propagation we can get some useful information from the satellite's data provider. Here's a screenshot of SMA and Inclination over time:



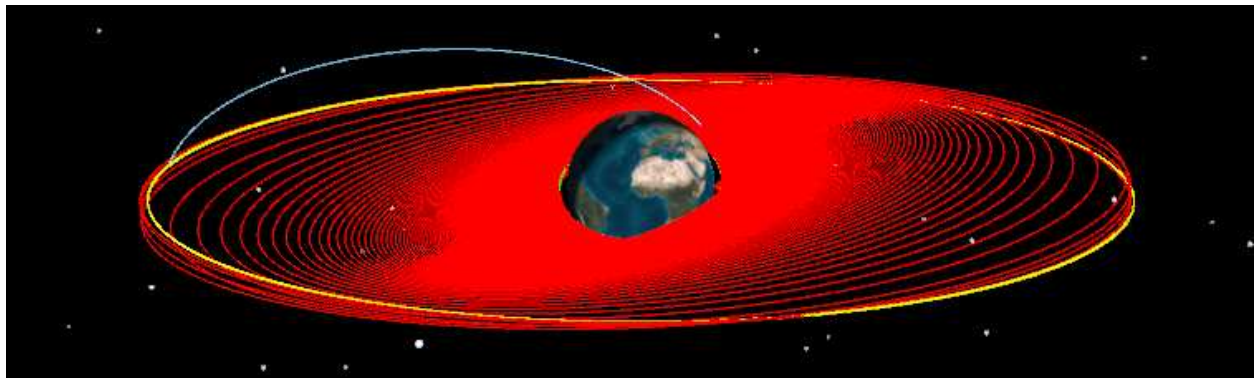
The following screenshot is reporting the thrust profile over time:



Also, the Maneuver Summary report shows the amount of fuel (209 kg) needed for the transfer maneuver:

Satellite-LowThrust							
Maneuver Number	Segment	Start Time (UTC)	Stop Time (UTC)	Duration (sec)	Delta V (m/sec)	Fuel Used (kg)	Thruster/Engine
1	LowThrust	22 Jan 2016 11:00:00.000	22 Mar 2016 11:00:00.000	5184000.000	5623.383873	209.799	Custom_Thruster_Set
Global Statistics							
Total Duration				5184000.000			
Total Delta V					5623.383873		
Total Fuel Used						209.799	

For the sake of comparison we may be interested in comparing the amount of fuel needed for a chemical rocket to perform the same orbit transfer. TO do this, we need to model another Astrogator satellite using the default chemical thrust and impulsive maneuvers. I won't explain all the steps here since this tutorial comes with the completed STK scenario attached.



Chemical maneuver is very fast (about 6 hours) at the expense of a pretty large fuel compsumption (about 1000 kg with the default Astrogator engine):

Satellite-Chemical								
Maneuver Number	Segment	Start Time (UTC)	Stop Time (UTC)	Duration (sec)	Delta V (m/sec)	Fuel Used (kg)	Thruster/Engine	
1	TransferOrbitManeuver_Burn1	22 Jan 2016 15:50:28.648	22 Jan 2016 15:50:28.648	3224.875	2336.654955	548.076	Constant_Thrust_and_Isp	
2	ToGeoManeuver_Burn2	22 Jan 2016 21:08:35.455	22 Jan 2016 21:08:35.455	2706.460	1812.654281	459.970	Constant_Thrust_and_Isp	
Global Statistics								
Total Duration				5931.335				
Total Delta V					4149.309236			
Total Fuel Used						1008.046		