GMAT API Cookbook

Release R2025a

Claire Conway, Joshua Raymond, and Darrel Conway

Mar 18, 2025

CONTENTS:

Note: This project is under active development.

The GMAT API Cookbook is a collection of use case examples that demonstrate common activities performed by GMAT API users. This document is a "living document:" it is intended to act as a central location for API examples that previously were only available through request of knowledgeable API users.

CHAPTER

ONE

PROPAGATION WITH THE GMAT API

1.1 Problem

GMAT's propagator is used to model spacecraft motion within a specified force model. The propagation can be done through either numerical integration or by an ephemeris. Duration of propagation can either be set directly or through a final goal reached by the spacecraft (i.e. propagating for 3 days, propagating to apoapsis, etc.). API users need to be able to perform a propagation through the API and be able to modify propagator settings.

1.2 Solution

For this example, we will focus on using a numerical integrator. Key objects in performing propagation are the Spacecraft, ForceModel, Integrator and Propagator. The Propagator is created with desired step settings and then a desired Integrator is attached to it. The types of integrators and analytic propagators supported by GMAT can be found in GMAT's User Guide under the "Propagator" resource reference. The integrator is then used to perform propagation using its Step method. An example of propagator use through the API is shown below.

1.2.1 Example API Propagation

Step 1: Configure the Spacecraft

We'll need a spacecraft to propagate. The following lines provide a basic low-Earth orbiting spacecraft configuration:

Listing 1.1: Spacecraft setup used in some examples

```
sat = gmat.Construct("Spacecraft", "LeoSat")
sat.SetField("DateFormat", "UTCGregorian")
sat.SetField("Epoch", "12 Mar 2020 15:00:00.000")
sat.SetField("CoordinateSystem", "EarthMJ2000Eq")
sat.SetField("DisplayStateType", "Keplerian")
sat.SetField("SMA", 7005)
sat.SetField("ECC", 0.008)
sat.SetField("INC", 28.5)
sat.SetField("RAAN", 75)
sat.SetField("AOP", 90)
sat.SetField("TA", 45)
sat.SetField("DryMass", 50)
sat.SetField("Cd", 2.2)
sat.SetField("Cr", 1.8)
```

(continues on next page)

(continued from previous page)

```
sat.SetField("DragArea", 1.5)
sat.SetField("SRPArea", 1.2)
```

Step 2: Configure the Force Model

Next we'll set up a force model. For this example, we'll use an Earth 8x8 potential model, with Sun and Moon point masses and Jacchia-Roberts drag. In GMAT, forces are collected in the ODEModel class. That class is scripted as a "ForceModel" in the script language. The API accepts either class name. The force model is built and its (empty) contents displayed using the following API commands:

```
# Create the ODEModel container
fm = gmat.Construct("ForceModel", "TheForces")
fm.Help()
```

In this example, the spacecraft is in Earth orbit. The largest force for the model is the Earth gravity field. We'll set it to an 8x8 field and add it to the force model using the code

```
# An 8x8 JGM-3 Gravity Model
earthgrav = gmat.Construct("GravityField")
earthgrav.SetField("BodyName","Earth")
earthgrav.SetField("Degree",8)
earthgrav.SetField("Order",8)
earthgrav.SetField("PotentialFile","JGM2.cof")

# Add the force into the ODEModel container
fm.AddForce(earthgrav)
```

Next we'll build and add the Sun, Moon, and Drag forces, and then show the completed force model.

```
# The Point Masses
moongrav = gmat.Construct("PointMassForce")
moongrav.SetField("BodyName","Luna")
sungrav = gmat.Construct("PointMassForce")
sungrav.SetField("BodyName","Sun")

# Drag using Jacchia-Roberts
jrdrag = gmat.Construct("DragForce")
jrdrag.SetField("AtmosphereModel","JacchiaRoberts")

# Build and set the atmosphere for the model
atmos = gmat.Construct("JacchiaRoberts")
jrdrag.SetReference(atmos)

# Add all of the forces into the ODEModel container
fm.AddForce(moongrav)
fm.AddForce(sungrav)
fm.AddForce(jrdrag)

# Show the settings on the force model
fm.Help()
```

In GMAT, the force model scripting shows the settings for each force. In the API, you can examine the settings for the individual forces using the Help method:

```
earthgrav.Help()
```

or, with a little work, you can see the GMAT scripting for the complete force model:

```
print(fm.GetGeneratingString(0))
```

Step 3: Configure the Integrator

Finally, in order to propagate, we need an integrator. For this example, we'll use a Prince-Dormand 7(8) Runge-Kutta integrator. The propagator is set using the code

```
# Build the propagation container that connect the integrator, force model, and␣
↪spacecraft together
pdprop = gmat.Construct("Propagator","PDProp")

# Create and assign a numerical integrator for use in the propagation
gator = gmat.Construct("PrinceDormand78", "Gator")
pdprop.SetReference(gator)

# Set some of the fields for the integration
pdprop.SetField("InitialStepSize", 60.0)
pdprop.SetField("Accuracy", 1.0e-12)
pdprop.SetField("MinStep", 0.0)
```

Step 4: Connect the Objects Together

Next the propagator needs its assigned force. This assignment is made by passing the force model to the propagator:

```
# Assign the force model to the propagator
pdprop.SetReference(fm)
```

The propagator also needs to know the object that is propagated. For this example, that is the spacecraft constructed above:

```
pdprop.AddPropObject(sat)
```

Step 5: Initialize the System and Propagate a Step

Finally, the system can be initialized and fired to see a single propagation step. The top level system initialization is performed using the API's Initialize function. The propagator uses a propagator specific method, PrepareInternals, to setup the propagation specific interconnections and data structures:

```
# Perform top level initialization
gmat.Initialize()
# Perform the integration subsystem initialization
pdprop.PrepareInternals()

# Refresh the integrator reference
gator = pdprop.GetPropagator()
```

We can then propagate, and start accumulating the data for display later:

```
# Take a 60 second step, showing the state before and after, and start buffering
# Buffers for the data
time = []
```

(continues on next page)

(continued from previous page)

```
pos = []
vel = []

gatorstate = gator.GetState()
t = 0.0
r = []
v = []
for j in range(3):
    r.append(gatorstate[j])
    v.append(gatorstate[j+3])
time.append(t)
pos.append(r)
vel.append(v)

print("Starting state:\n", t, r, v)

# Take a step and buffer it
gator.Step(60.0)
gatorstate = gator.GetState()
t = t + 60.0
r = []
v = []
for j in range(3):
    r.append(gatorstate[j])
    v.append(gatorstate[j+3])
time.append(t)
pos.append(r)
vel.append(v)

print("Propped state:\n", t, r, v)
```

Finally, we can run for a few orbits and show the results:

```
for i in range(360):
    # Take a step and buffer it
    gator.Step(60.0)
    gatorstate = gator.GetState()
    t = t + 60.0
    r = []
    v = []
    for j in range(3):
        r.append(gatorstate[j])
        v.append(gatorstate[j+3])
    time.append(t)
    pos.append(r)
    vel.append(v)

import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (15, 5)
positions = plt.plot(time, pos)
```

The velocities can also be plotted:

```
velocities = plt.plot(time, vel)
```

1.3 Discussion

This example has showcased one way to generate the position and velocity of a spacecraft over a specified time span. This script is now easily modifiable should a user need to adjust step sizes, adjust the spacecraft initial state, provide a more extensive force model, etc. to quickly rerun GMAT through the API and collect new sets of data.

CHAPTER

TWO

STATE MANAGEMENT WITH THE GMAT API

2.1 Problem

GMAT contains state data for various objects that can be accessed for data collection. API users need to also have access to state data and understand the initialization of said data as the state data in GMAT can be a bit confusing.

2.2 Solution

This chapter demonstrates the access and manipulation of state data from a Spacecraft. The internal state is retrieved from the GetState method on the Spacecraft, and this state is used to convert outputs to requested state types in the Spacecraft's coordinate system. Here we will cover examples of getting Cartesian and Keplerian states. As will be shown, care must be taken for whether or not this state has been properly initialized. The Spacecraft state can also be manipulated and gathered in different coordinate systems created by the API user.

Step 1: Configure the Spacecraft

We'll need an object that provides the state. We begin by creating a basic spacecraft, along with a reference to the state data inside of the spacecraft:

```
sat = gmat.Construct("Spacecraft","MySat")
iState = sat.GetState()
```

The state reference here, iState, operates on the member of the Spacecraft object that GMAT uses when running a simulation. The "internal state" referenced by iState is the Earth-centered mean-of-J2000 equatorial representation of position and velocity of the spacecraft MySat. The data is contained in a GmatState object, which can be seen by accessing iState from the Python command prompt like this:

```
iState
```

GmatState objects are used to collect together an epoch and a vector of data defining the object's location in space at that epoch. These data can be accessed directly:

```
print("The state epoch is ", iState.GetEpoch(), "in A1ModJulian, the state has ", iState.
↪GetSize(), " elements, and contains the data:\n ", iState.GetState(), "\n")
```

The data shown here is the default GmatState vector data for a spacecraft. The epoch is January 1, 2000 at 12:00:00.000 in TAIModJulian time, or 21545.00000039794 in A.1 ModJulian time. Note that GMAT uses A.1 Mod Julian as its internal epoch system. The state has 6 elements. The position and velocity data at this point are filled in with the dummy entries -999.999.

Step 2: Working with Cartesian and Keplerian Representations

A spacecraft in GMAT has a second collection of data: the state data for the spacecraft in the coordinate system set on the spacecraft. These data are the spacecraft's "display state", named that way because they are the data displayed to the user on the GMAT graphical user interface. Users interact with the display state similarly to the way they interact with the scripting language. Data for a Keplerian state can be set using the SetField() method, as shown here:

```
sat.SetField("StateType", "Keplerian")
sat.SetField("SMA", 7015)
sat.SetField("ECC", 0.0011)
sat.SetField("INC", 98.6)
sat.SetField("RAAN", 75)
sat.SetField("AOP", 90)
sat.SetField("TA", 33.333)
```

At this point it can appear to the user that the data is set, but it really is not. The spacecraft object cannot interpret the state data. The data set using SetField needs more information than a spacecraft object can provide by itself. Specifically, the uninitialized spacecraft here does not have a connected coordinate system. Cartesian state data set on the spacecraft does not have connections defining the coordinate origin, nor the structures needed to set the orientation of the axes defining directions. Additionally, the spacecraft does not have the the gravitational constant needed to interpret Keplerian data at this point.

In this uninitialized state, the spacecraft uses its GmatState buffer to hold the data entries. We can see that the data is not yet fully populated by posting queries to the spacecraft:

```
print("The internal state buffer just holds preinitialization data (Keplerian here):\n
↪", iState.GetState())
print("but access to the Keplerian state shows that it is not correct:\n  ", sat.
↪GetKeplerianState(), "\n")
```

The Keplerian state data is not correct because the GMAT objects are not yet initialized. Once we initialize the system, the Keplerian state will be correct, and the internal state will be updated to the EarthMJ2000Eq system. The interobject connections necessary for these settings are made by calling the API's Initialize() function:

```
gmat.Initialize()
print("The initialized internal state buffer is EarthMJ2000Eq:\n  ", iState.GetState())
print("and the Keplerian state is correct:\n  ", sat.GetKeplerianState(), "\n")
```

Changes made to the state variables are now applied to the state as expected:

```
sat.SetField("SMA", 8000)
print("Internal state:\n ", iState.GetState())
print("Cartesian:\n ", sat.GetCartesianState())
print("Keplerian:\n ", sat.GetKeplerianState())
print()
sat.SetField("INC", 45)
print("Internal state:\n ", iState.GetState())
print("Cartesian:\n ", sat.GetCartesianState())
print("Keplerian:\n ", sat.GetKeplerianState())
print()
sat.SetField("TA", 50)
print("Internal state:\n ", iState.GetState())
print("Cartesian:\n ", sat.GetCartesianState())
print("Keplerian:\n ", sat.GetKeplerianState())
print()
```

Step 3: Changing Coordinate Systems

The previous section shows how to access Cartesian and Keplerian representations of the system. In this section we will work with a couple of different coordinate systems: an Earth fixed coordinate system named "ECF" and accessed using the Python reference ecf, and a solar ecliptic system named "SolarEcliptic", referenced as sec. These coordinate systems are built using the code:

```
ecf = gmat.Construct("CoordinateSystem", "ECF", "Earth", "BodyFixed")
sec = gmat.Construct("CoordinateSystem", "SolarEcliptic", "Sun", "MJ2000Ec")
```

In this section, the spacecraft sat defined previously will be used with the Earth fixed coordinate system, and a copy of that spacecraft will be used with the solar ecliptic system. GMAT's objects support a method, Copy(), that copies one object into another object of the same type. Rather than set up a new spacecraft from scratch, we'll use that framework to get started by creating a new spacecraft and then setting the coordinate systems so that the original spacecraft uses the ECF coordinate system and the new spacecraft uses the solar ecliptic system.

```
solsat = gmat.Construct("Spacecraft","SolarSat")
solsat.Copy(sat)

# Now set coordinate systems
sat.SetField("CoordinateSystem","ECF")
solsat.SetField("CoordinateSystem","SolarEcliptic")
```

We've reset the coordinate system names on the spacecraft at this point, but have yet to reset the associated objects because the Initialize() function that connects objects together has not been called since making the reassignment. The data reflects this state of the system:

```
# Show the data after setting the new coordinate systems, before initialization
print("The spacecraft ", sat.GetName(), " initialization state is ", sat.IsInitialized())
print("The internal state buffer:  ", iState.GetState())
print("The ECF Cartesian State:    ", sat.GetCartesianState())
print("The ECF Keplerian State:    ", sat.GetKeplerianState())
print()
print("The spacecraft ", solsat.GetName(), " initialization state is ", sat.
↪IsInitialized())
print("The internal state buffer (SolarSat):  ", solsat.GetState().GetState())
print("The SolarEcliptic Cartesian State:     ", solsat.GetCartesianState())
print("The SolarEcliptic Keplerian State:     ", solsat.GetKeplerianState())
    print()
```

Note that the initialization state reported here is a bug: resetting object references should toggle the initialization flag, but did not.

Once we initialize the system, replacing the coordinate system references with the correct objects, the data is once again correct:

```
# Connect the GMAT objects together
gmat.Initialize()

# And show the data in the new coordinate systems
print("The internal state buffer:  ", iState.GetState())
print("The ECF Cartesian State:    ", sat.GetCartesianState())
print("The ECF Keplerian State:    ", sat.GetKeplerianState())
print()
print("The internal state buffer (SolarSat):  ", solsat.GetState().GetState())
print("The SolarEcliptic Cartesian State:     ", solsat.GetCartesianState())
print("The SolarEcliptic Keplerian State:     ", solsat.GetKeplerianState())
```

2.3 Discussion

Utilizing the state management foundations covered in this chapter, API users can gather Spacecraft state data throughout missions in desired representations. The other state and coordinate system types supported by GMAT can be found in the GMAT User Guide under the Spacecraft Orbit State and CoordinateSystem resources, respectively.

CHAPTER

THREE

STM AND COVARIANCE PROPAGATION

3.1 Problem

Spacecraft propagation advances the position and velocity of the spacecraft through time, and models mass flow during finite burn maneuvers. Users may also need to propagate the spacecraft's orbital state transition matrix (STM) and the orbital covariance matrix. This chapter shows how to add the STM and covariance propagation to the propagation example shown in *Propagation with the GMAT API*.

3.2 Solution

The GMAT Spacecraft and propagation models work together to build and propagate the state transition and covariance matrices.

3.2.1 Example STM Propagation

Step 1: Configure the Spacecraft

We'll need a spacecraft to propagate. Listing 3.1 shows an abbreviated block of the spacecraft configuration from *Propagation with the GMAT API*.

Listing 3.1: Abbreviated Spacecraft configuration (see Listing 1.1 for full version)

```
sat = gmat.Construct("Spacecraft", "LeoSat")
sat.SetField("DateFormat", "UTCGregorian")
...
sat.SetField("Cr", 1.8)
sat.SetField("DragArea", 1.5)
sat.SetField("SRPArea", 1.2)
```

The spacecraft initializes the state transition matrix to an identity matrix, and the covariance matrix to a 6x6 matrix with large values set nonzero along its diagonal. The STM setting is required so that the propagated and unpropagated STM requirements are met

$$r(t1) = \Phi(t0, t1) r(t0)$$
$$\Phi(t0, t0) = I$$

The covariance matrix is generally set to more reasonable numbers before proceeding with covariance propagation, either through an estimation process or by direct user intervention. Listing 3.2 shows the latter approach.

Listing 3.2: Covariance initialization of sat using the API

```
# Initialize the covariance matrix
sat.SetRealParameter("Covariance", 0.000001, 0, 0)
sat.SetRealParameter("Covariance", 0.0000015, 1, 1)
sat.SetRealParameter("Covariance", 0.000001, 2, 2)
sat.SetRealParameter("Covariance", 2.5e-11, 3, 3)
sat.SetRealParameter("Covariance", 2.5e-11, 4, 4)
sat.SetRealParameter("Covariance", 2.5e-11, 5, 5)
```

Step 2: Update the propagation subsystem

GMAT's propagators configure the data that is integrated using a component called the propagation state manager (PSM). The PSM assembles the data that is integrated into a vector that the numerical integrator uses to advance the GMAT simulation through time. The STM and covariance matrix are set to participated in the integration through calls to the PSM, shown in Listing 3.3.

Listing 3.3: Propagation State Manager Configuration for Covariance Propagation

```
# Setup STM and covariance propagation
psm = pdprop.GetPropStateManager()
psm.SetProperty("Covariance",sat)
psm.SetProperty("STM",sat)
```

Step 3: Spacecraft Updating

The propagation chapter of this document glosses over an object synchronization item that is required for successful covariance propagation. After a propagation step has been performed, the propagation vector managed by the PSM needs to be passed to the spacecraft object so that the internal spacecraft data is in sync with the propagation. This synchronization is performed by directing the initialized propagator to pass the state vector into the spacecraft. The propagator performs this action by calling methods on the PSM. Listing 3.4 shows the calls that accomplish this data synchronization.

Listing 3.4: Data synchronization between the propagator and the spacecraft

```
# Perform top level and integrator initialization
gmat.Initialize()
pdprop.PrepareInternals()
propagator = pdprop.GetPropagator()

# Push the data into the spacecraft runtime elements
propagator.UpdateSpaceObject()
```

Note, in particular, the last line in this block. The call to UpdateSpaceObjects() is new in this chapter, and is the call that feeds the data from the propagation subsystem into the spacecraft object.

Step 4: Accessing the STM and Covariance

The steps shown above configure the GMAT API for STM and covariance propagation. The data is viewed by accessing it from the spacecraft object. The code in Listing 3.5 shows the access methods needed to use the matrix data. The code shown here accesses the data for printing.

Listing 3.5: Code that displays the STM and covariance to the user

```
print(sat.GetName(), " State and matrix data:\n")
print("Epoch:  ", sat.GetEpoch())
print("Position:  ", sat.GetField("X"), ", ", sat.GetField("Y"), ", ",
    sat.GetField("Z"))
print("Velocity:  ", sat.GetField("VX"), ", ", sat.GetField("VY"), ", ",
    sat.GetField("VZ"))

# Access the state transition matrix data
stm = sat.GetRmatrixParameter("STM")
print("\nState Transition Matrix:\n")
for i in range(6):
    cstr = ""
    for j in range(6):
        cstr = cstr + "    " + str(stm.GetElement(i,j))
    print(cstr)
print()

# Access the covariance matrix data
cov = sat.GetCovariance().GetCovariance()
print("\nCovariance:\n")
for i in range(6):
    cstr = ""
    for j in range(6):
        cstr = cstr + "    " + str(cov.GetElement(i,j))
    print(cstr)
```

3.3 Discussion

This example has provides an overview of the methods used to access the STM and Covariance matrices using the GMAT API. The appendix to this chapter contains a complete working of these features.

3.4 Appendix: A Complete Example

```
#!/bin/python3

# The Propagation chapter example code from the GMAT API Cookbook

from load_gmat import gmat
import matplotlib.pyplot as plt


# Define core objects

def ConfigureCoreObjects():
    '''
    Objects built in the earlier exercise
    '''
```

(continues on next page)

(continued from previous page)

```
    # Configure the spacecraft
    sat = gmat.Construct("Spacecraft", "LeoSat")
    sat.SetField("DateFormat", "UTCGregorian")
    sat.SetField("Epoch", "12 Mar 2020 15:00:00.000")
    sat.SetField("CoordinateSystem", "EarthMJ2000Eq")
    sat.SetField("DisplayStateType", "Keplerian")
    sat.SetField("SMA", 7005)
    sat.SetField("ECC", 0.008)
    sat.SetField("INC", 28.5)
    sat.SetField("RAAN", 75)
    sat.SetField("AOP", 90)
    sat.SetField("TA", 45)
    sat.SetField("DryMass", 50)
    sat.SetField("Cd", 2.2)
    sat.SetField("Cr", 1.8)
    sat.SetField("DragArea", 1.5)
    sat.SetField("SRPArea", 1.2)

    # Create the ODEModel container
    fm = gmat.Construct("ForceModel", "TheForces")

    # An 8x8 JGM-3 Gravity Model
    earthgrav = gmat.Construct("GravityField")
    earthgrav.SetField("BodyName","Earth")
    earthgrav.SetField("Degree",8)
    earthgrav.SetField("Order",8)
    earthgrav.SetField("PotentialFile","JGM2.cof")

    # Add the force into the ODEModel container
    fm.AddForce(earthgrav)

    # The Point Masses
    moongrav = gmat.Construct("PointMassForce")
    moongrav.SetField("BodyName","Luna")
    sungrav = gmat.Construct("PointMassForce")
    sungrav.SetField("BodyName","Sun")

    # Drag using Jacchia-Roberts
    jrdrag = gmat.Construct("DragForce")
    jrdrag.SetField("AtmosphereModel","JacchiaRoberts")

    # Build and set the atmosphere for the model
    atmos = gmat.Construct("JacchiaRoberts")
    jrdrag.SetReference(atmos)

    # Add all of the forces into the ODEModel container
    fm.AddForce(moongrav)
    fm.AddForce(sungrav)
    fm.AddForce(jrdrag)

    # Build the propagation container that connect the integrator, force model, and␣
↪spacecraft together
```

(continues on next page)

(continued from previous page)

```
    pdprop = gmat.Construct("Propagator","PDProp")

    # Create and assign a numerical integrator for use in the propagation
    gator = gmat.Construct("PrinceDormand78", "Gator")
    pdprop.SetReference(gator)

    # Set some of the fields for the integration
    pdprop.SetField("InitialStepSize", 60.0)
    pdprop.SetField("Accuracy", 1.0e-12)
    pdprop.SetField("MinStep", 0.0)

    # Assign the force model to the propagator
    pdprop.SetReference(fm)
    pdprop.AddPropObject(sat)

    return sat, pdprop

def StepAMinute(sat, propagator):
    propagator.Step(60.0)

def UpdateData(propagator, time, pos, vel):
    gatorstate = propagator.GetState()
    if len(time) == 0:
       t = 0.0
    else:
        t = time[len(time) - 1] + 60.0
    r = []
    v = []
    for j in range(3):
        r.append(gatorstate[j])
        v.append(gatorstate[j+3])
    time.append(t)
    pos.append(r)
    vel.append(v)

def ShowStateStmCov(sat):

    print(sat.GetName(), " State and matrix data:\n")
    print("Epoch:  ", sat.GetEpoch())
    print("Position:  ", sat.GetField("X"), ", ", sat.GetField("Y"), ", ",
        sat.GetField("Z"))
    print("Velocity:  ", sat.GetField("VX"), ", ", sat.GetField("VY"), ", ",
        sat.GetField("VZ"))

    # Access the state transition matrix data
    stm = sat.GetRmatrixParameter("STM")
    print("\nState Transition Matrix:\n")
    for i in range(6):
        cstr = ""
        for j in range(6):
            cstr = cstr + "    " + str(stm.GetElement(i,j))
        print(cstr)
```

(continues on next page)

(continued from previous page)

```
    print()

    # Access the covariance matrix data
    cov = sat.GetCovariance().GetCovariance()
    print("\nCovariance:\n")
    for i in range(6):
        cstr = ""
        for j in range(6):
            cstr = cstr + "    " + str(cov.GetElement(i,j))
        print(cstr)
    print()


# Load the preliminaries
sat, pdprop = ConfigureCoreObjects()

# ---------------------------------------------------------------------------
# New code
# ---------------------------------------------------------------------------

# Initialize the covariance matrix
sat.SetRealParameter("Covariance", 0.000001, 0, 0)
sat.SetRealParameter("Covariance", 0.0000015, 1, 1)
sat.SetRealParameter("Covariance", 0.000001, 2, 2)
sat.SetRealParameter("Covariance", 2.5e-11, 3, 3)
sat.SetRealParameter("Covariance", 2.5e-11, 4, 4)
sat.SetRealParameter("Covariance", 2.5e-11, 5, 5)

# Setup STM and covariance propagation
psm = pdprop.GetPropStateManager()
psm.SetProperty("Covariance",sat)
psm.SetProperty("STM",sat)

# Perform top level and integrator initialization
gmat.Initialize()
pdprop.PrepareInternals()
propagator = pdprop.GetPropagator()

# Push the data into the spacecraft runtime elements
propagator.UpdateSpaceObject()

# Matplotlib graphics buffers
time = []
pos = []
vel = []

# Save initial state information into the graphics buffers
UpdateData(propagator, time, pos, vel)

# Show initial information
ShowStateStmCov(sat)
```

(continues on next page)

(continued from previous page)

```
# Propagate 10 minutes and buffer data at each step
for i in range(10):
    StepAMinute(sat, propagator)
    # Push the propagation data onto the spacecraft
    propagator.UpdateSpaceObject()
    UpdateData(propagator, time, pos, vel)

# Show the results
plt.rcParams['figure.figsize'] = (15, 5)
f1 = plt.figure(1)
positions = plt.plot(time, pos)
f2 = plt.figure(2)
velocities = plt.plot(time, vel)

f1.show()
f2.show()

ShowStateStmCov(sat)

print("Press Enter")
input()
```

CHAPTER

FOUR

EXECUTING FINITE BURNS

4.1 Problem

GMAT's thruster resources allow users to perform finite burns with control over their burn duration, burn direction, etc. in order to propagate maneuvers. A simple GMAT script example of the activation and deactivation of a finite burn can be found below:

```
BeginMissionSequence;
%  Propagate for 1/10 of a day, without thrusters on.
Propagate 'Prop 0.1 Days' prop(Sc) {Sc.ElapsedSecs = 8640};

%  Turn on thrusters....they will remain on through all events until the
%  "EndFiniteBurn fb(Sc)" command is executed.
BeginFiniteBurn 'Turn on Thruster' fb(Sc);

%  Propagate for 1 day, while thrusters are turned on.
Propagate 'Prop 1 day' prop(Sc) {Sc.ElapsedDays = 1};

%  Turn off thrusters
EndFiniteBurn 'Turn off Thruster' fb(Sc);

%  Propagate for 1 day
Propagate 'Prop 1 Day' prop(Sc) {Sc.ElapsedDays = 1};
```

API users need to be able to perform finite burn maneuvers while easily being able to modify propagation and burn sequences in order to make corrections to achieve desired maneuver goals.

4.2 Solution

To utilize GMAT's finite burn capabilities through the API, users will need tank, thruster, FiniteBurn, and FiniteThrust objects. In this guide, we will focus on using a ChemicalTank and ChemicalThruster, but electric tanks and thrusters are also available options. The hardware will be attached to the desired Spacecraft, and the thrust force will be added to the ForceModel in use to propagate the Spacecraft with burns active. With everything set up, API users can turn thrusters on and off through the "IsFiring" boolean on a thruster and setting a Spacecraft's and corresponding FiniteBurn's maneuvering flags. An example covering these steps is shown below.

4.2.1 Example API Finite Burn

Step 1: Instantiate Spacecraft

We need to create a spacecraft that will perform the maneuver. Most of the default Spacecraft settings will be used. The dry mass is reduced to allow more acceleration on the spacecraft during the maneuver. We will also set up a simple default Earth point mass ForceModel and a propagator.

```
# Setup a spacecraft and basic propagation system
sat = gmat.Construct("Spacecraft", "Sat")
sat.SetField("DryMass", 80.0)

fm = gmat.Construct("ForceModel", "FM")
epm = gmat.Construct("PointMassForce", "EPM")
fm.AddForce(epm)

prop = gmat.Construct("Propagator", "Prop")
gator = gmat.Construct("PrinceDormand78", "Gator")
prop.SetReference(gator)
prop.SetReference(fm)
```

Step 2: Set Finite Burn specific settings

To model fuel use and thrusting on a Spacecraft, we will attach a ChemicalTank and ChemicalThruster to the Spacecraft. Meanwhile for the burn sequence itself, we create a FiniteBurn object and connect it to the ChemicalThruster. Finally, a burn force is created as a FiniteThrust object and connected to both the FiniteBurn and the Spacecraft that is maneuvering. This force will later be added to the ForceModel so that the force can be applied.

Note that the ChemicalThruster is where we set the thrust direction for the burn. For this case, we will simply use the default settings, applying thrust in the positive velocity direction of the Spacecraft with respect to the Earth.

```
# Setup the spacecraft hardware
tank = gmat.Construct("ChemicalTank", "Fuel")
tank.SetField("FuelMass", 20.0)
thruster = gmat.Construct("ChemicalThruster", "Thruster")
thruster.SetField("Tank", "Fuel")
thruster.SetField("DecrementMass", True)

sat.SetField("Tanks", "Fuel")
sat.SetField("Thrusters", "Thruster")

# Ensure the system is up to date
gmat.Initialize()

# Build the burn model used for the burn
burn = gmat.Construct("FiniteBurn", "TheBurn")
burn.SetField("Thrusters", "Thruster")
burn.SetSolarSystem(gmat.GetSolarSystem())
burn.SetSpacecraftToManeuver(sat)

# Build the force that applies the burn
def setThrust(s, b):
    bf = gmat.FiniteThrust("Thrust")
    bf.SetRefObjectName(gmat.SPACECRAFT, s.GetName())
    bf.SetReference(b)
```

(continues on next page)

(continued from previous page)

```
    gmat.ConfigManager.Instance().AddPhysicalModel(bf);
    return bf

burnForce = setThrust(sat, burn)
```

Step 3: Wire the objects together

Here we initialize our setup and grab the ChemicalThruster object specifically owned by our Spacecraft. This ensures we are working with the thruster acting on and decreasing the fuel of our Spacecraft.

```
gmat.Initialize()
prop.AddPropObject(sat)
prop.PrepareInternals()

# Access the thruster cloned onto the spacecraft
theThruster = sat.GetRefObject(gmat.THRUSTER, "Thruster")

# Check the spacecraft mass
print(sat.GetName(), " current mass:  ", sat.GetField("TotalMass"), " kg\n")
```

Step 4: Propagate

In order to switch between a non-firing and firing state, we will need to set whether our ChemicalThruster is firing, our Spacecraft is maneuvering, and what Spacecraft our FiniteBurn is affecting. With everything turned on, we then add our FiniteThrust object from Step 2 to the ForceModel. In this example, our burn will be used to raise the SMA.

```
# Take a few steps
elapsed = 0.0
dt = 60.0
gator = prop.GetPropagator()

# Step for 10 minutes
print("\n Propagate 10 minutes without a        burn\n-----------------------------------
↪--------------\n")

print("Firing: ", theThruster.GetField("IsFiring"), "\n")
state = gator.GetState()
r = math.sqrt(state[0]**2 + state[1]**2 + state[2]**2)
vsq = state[3]**2 + state[4]**2 + state[5]**2
sma = r * mu / (2.0 * mu - vsq * r)
print(elapsed, sma, state, sat.GetField("TotalMass"))
for i in range(10):
        gator.Step(dt)
        elapsed = elapsed + dt
        state = gator.GetState()
        r = math.sqrt(state[0]**2 + state[1]**2 + state[2]**2)
        vsq = state[3]**2 + state[4]**2 + state[5]**2
        sma = r * mu / (2.0 * mu - vsq * r)
        print(elapsed, sma, state, sat.GetField("TotalMass"))
        gator.UpdateSpaceObject()

# Step for 30 minutes with a burn
print("\n Propagate 30 minutes with a burn\n-----------------------------------------------
↪----\n")
```

(continues on next page)

(continued from previous page)

```
# ----------------------------
# Finite Burn Specific Settings
# ----------------------------
# Turn on the thruster
theThruster.SetField("IsFiring", True)
sat.IsManeuvering(True)
#sat.SetPropItem("MassFlow")
burn.SetSpacecraftToManeuver(sat)
#burn.TakeAction("SetData")

# Add the thrust to the force model
prop.AddForce(burnForce)
psm = prop.GetPropStateManager()
psm.SetProperty("MassFlow")
# ----------------------------

prop.PrepareInternals()
gator = prop.GetPropagator()

print("Firing: ", theThruster.GetField("IsFiring"), "\n")
# Now propagate thruogh the burn
for i in range(30):
        gator.Step(dt)
        elapsed = elapsed + dt
        state = gator.GetState()
        r = math.sqrt(state[0]**2 + state[1]**2 + state[2]**2)
        vsq = state[3]**2 + state[4]**2 + state[5]**2
        sma = r * mu / (2.0 * mu - vsq * r)
        print(elapsed, sma, state, sat.GetField("TotalMass"))
        gator.UpdateSpaceObject()
```

Step 5: Turn off the Thruster

We now simply undo our work from Step 4, turning off the burn.

```
fm = prop.GetODEModel()
fm.DeleteForce(burnForce)
theThruster.SetField("IsFiring", False)
sat.IsManeuvering(False)

prop.PrepareInternals()
gator = prop.GetPropagator()
```

4.3 Discussion

This example provides all the core elements required to develop more complex maneuvers through finite burns. By using the IsFiring field on thrusters and managing ForceModel forces, finite burns can be controlled through propagation to apply staggered burn sequences. Here a simple loop stepping through 30 time steps was used for the burn, but a user could also set up different goals through Python (for example, having the burn run until a desired SMA was achieved).

CHAPTER

FIVE

ACCESSING GMAT COMMANDS

5.1 Problem

GMAT's mission control sequence (MCS) controls the order of actions taken when a simulation is run. The MCS is a linked list of objects all derived from the GmatCommand base class. API users may need access to these commands in order to determine the state of the system at specific nodes in the MCS. This page shows how to do this, using the Ex_HohmannTransfer sample script.

5.2 Solution

The solver control sequences for targeting and for optimization share setting of variables through the Vary command. Targeting specifies the target goals using the Achieve command. Optimization uses the Minimize command to set the objective function that is minimized, and the NonlinearConstraint command to constrain the optimization solution.

5.2.1 Setup and Some Comments

The targeting example used in this chapter uses the Hohmann transfer sample script supplied with GMAT. Create a folder one level up from the run folder, name it scripts, and copy "Ex_HohmannTransfer.script" from the GMAT samples folder into that folder.

The purpose of this chapter is exploration, using the GMAT API, of the mission control sequence. The MCS in the Hohmann Transfer sample script is shown in Listing 5.1.

Listing 5.1: The Full Hohmann Transfer MCS

```
BeginMissionSequence;

Propagate 'Prop to Perigee' DefaultProp(DefaultSC) {DefaultSC.Periapsis};

% Burn in the velocity direction to reach an alternate Apoapsis point
Target 'Raise and Circularize' DC {SolveMode = Solve, ExitMode = DiscardAndContinue};
   Vary 'Vary TOI.V' DC(TOI.Element1 = 0.5, {Perturbation = 0.0001, Lower = 0, Upper = 3.
↪14159, MaxStep = 0.2});
   Maneuver 'Apply TOI' TOI(DefaultSC);
   Propagate 'Prop to Apogee' DefaultProp(DefaultSC) {DefaultSC.Apoapsis};
   Achieve 'Achieve RMAG' DC(DefaultSC.Earth.RMAG = 42165, {Tolerance = 0.1});
   Vary 'Vary GOI.V' DC(GOI.Element1 = 0.5, {Perturbation = 0.0001, Lower = 0, Upper = 3.
↪14159, MaxStep = 0.2});
   Maneuver 'Apply GOI' GOI(DefaultSC);
```

(continues on next page)

(continued from previous page)

```
   Achieve 'Achieve ECC' DC(DefaultSC.ECC = 0, {Tolerance = 0.1});
EndTarget;  % For targeter DC

Propagate 'Prop 1 Day' DefaultProp(DefaultSC) {DefaultSC.ElapsedSecs = 86400};
```

The GMAT MCS shown here consists of two pieces. The core MCS is a set of four commands:

Listing 5.2: The Top Level MCS

```
BeginMissionSequence;
Propagate 'Prop to Perigee' DefaultProp(DefaultSC) {DefaultSC.Periapsis};
Target 'Raise and Circularize' DC {SolveMode = Solve, ExitMode = DiscardAndContinue};
Propagate 'Prop 1 Day' DefaultProp(DefaultSC) {DefaultSC.ElapsedSecs = 86400};
```

Each line in this list is an instance of a class derived from the GmatComand class. GmatCommand provides two methods used to manage the MCS: a GetNext() method that returns a pointer to the next command in the list, and a GetPrevious() method that returns the command preceding the command. Each list is terminated - on both ends - by a null pointer, indicates as a "None" object in Python.

The Target command on line 3 is an example of a "BranchCommand" object in the GMAT code. Branch commands are commands that can pass processing in more than one direction: either to the next node in the sequence, or to a branch of commands scripted to perform some specific set of actions. In this example, the Target command branches to the Targeter Control Sequence that specifies the actions followed to tune a pair of maneuvers that move a spacecraft from low Earth orbit to geosynchronous orbit.

This chapter shows how to access these commands and examine them. Manipulation of the commands, with a focus on manipulations for the targets and optimizers, will be presented in a later chapter.

5.2.2 Top Level Command Access

This section shows how to view the main MCS.

Step 1: Load the script

GMAT Scripts are loaded and run in the APIusing the LoadScript function:

```
from load_gmat import gmat

# Load the Hohmann transfer script into GMAT.
retval = gmat.LoadScript("../scripts/Ex_HohmannTransfer.script")

if retval == False:
   print("The script failed to load.")
   exit()
```

This function returns the status of the load: True if the script was loaded, False if the load failed.

Step 2: Access the first command in the list

Access to a loaded script in GMAT is made through the GMAT Moderator. From the API, access to the moderator, and then to the first command (or node) in the MCS is straightforward:

```
# Connect to the GMAT engine through the Moderator singleton
mod = gmat.Moderator.Instance()
```

(continues on next page)

(continued from previous page)

```
# Access the first node of the MCS; by convention, this is a NoOp command
startnode = mod.GetFirstCommand()
```

The command list in GMAT always starts with a node that performs no work - a NoOp command in the core code. Commands in GMAT have both a type and optionally a name. The type field identifies the command's class. The Name field stores the command string assigned to the command. One way to access these attributes of the current command is by simply printing the Python object directly. The can be accessed individually using the GetTypeName() and GetName() methods:

Listing 5.3: Attributes of the first MCS node

```
# View the attributes of the node
print(startnode)
print("Type:  ", startnode.GetTypeName())
print("Name:  ", startnode.GetName())
```

This code produces the output

```
Object of type NoOp named
Type:   NoOp
Name:
```

The NoOp command does not have a name, so those fields are empty in the output.

Step 3: Access the next command in the list

The MCS itself is pointed to by this NoOp command. Access to the MCS is made like this:

Listing 5.4: Accessing the first scripted command

```
# Now access the first scripted command
node = startnode.GetNext()
```

The node object now points to the first command in the MCS, as can be verified like this:

Listing 5.5: Attributes of the current MCS node

```
# View the attributes of the current node
print(node)
print("Type:  ", node.GetTypeName())
print("Name:  ", node.GetName())
```

with output

```
Object of type BeginMissionSequence named
Type:   BeginMissionSequence
Name:
```

The node object points to the first scripted command, "BeginMissionSequence," as expected.

Step 4: Access the previous command in the list

GMAT's MCS is implemented as a doubly linked list, meaning that the commands are linked both to subsequent commands and to commands that come before the current command. Subsequent commands are accessed using GetNext(), and shown above. The predecessors to the current command are accessed using GetPrevious(). For example, the node object set in Listing 5.4 points back to the NoOp command at the start of the list, and can be shown like this:

```
print(node.GetPrevious())
```

producing the expected output:

```
Object of type NoOp named
```

Step 5: Display the MCS

The methods presented above can be used to view the complete top level MCS. The function WalkTheMCS() shown in Listing 5.6 shows how to move through the control sequence and access each node's descriptive attributes.

Listing 5.6: A Utility Function to Show the MCS

```
def WalkTheMCS(node):
   '''
      Simple function to show a GmatCommand linked list, omitting children

      inputs:
         node  The starting node for the command list

      returns:
         None.  The command list is written to the display
   '''
   count = 1

   while node != None:
      indent = "  "
      print(indent, count, ":  ", '{:20}'.format(node.GetTypeName()), node.GetName())
      node = node.GetNext()
      count = count + 1
```

Whenthis function is called with the node pointing to the BeginMissionSequence command, the ouput shown in Listing 5.7 is displayed.

Listing 5.7: The MCS shown using WalkTheMCS()

```
   1 :   BeginMissionSequence
   2 :   Propagate           Prop to Perigee
   3 :   Target              Raise and Circularize
   4 :   Propagate           Prop 1 Day
```

Compare this output with Listing 5.1 to see how the methods called from the API align with the GMAT scripting.

5.2.3 Branch Command Access

The MCS for the Hohmann transfer script includes a targeting loop. The sequence of commands in that loop are not shown when running the WalkTheMCS() function because the targeter commands are branched off of the main mission control sequence. That branch of commands, called the solver control sequence, is scripted off of the Target command like this in the sample script:

Listing 5.8: The Solver Control Sequence for the Hohmann Transfer

```
1 Target 'Raise and Circularize' DC {SolveMode = Solve, ExitMode = DiscardAndContinue};
2    Vary 'Vary TOI.V' DC(TOI.Element1 = 0.5, {Perturbation = 0.0001, Lower = 0, Upper = 3.
  ↪14159, MaxStep = 0.2});
```

(continues on next page)

(continued from previous page)

```
   Maneuver 'Apply TOI' TOI(DefaultSC);
   Propagate 'Prop to Apogee' DefaultProp(DefaultSC) {DefaultSC.Apoapsis};
   Achieve 'Achieve RMAG' DC(DefaultSC.Earth.RMAG = 42165, {Tolerance = 0.1});
   Vary 'Vary GOI.V' DC(GOI.Element1 = 0.5, {Perturbation = 0.0001, Lower = 0, Upper = 3.
↪14159, MaxStep = 0.2});
   Maneuver 'Apply GOI' GOI(DefaultSC);
   Achieve 'Achieve ECC' DC(DefaultSC.ECC = 0, {Tolerance = 0.1});
EndTarget;  % For targeter DC
```

The Target command here is apart of the top level MCS. The remaining lines are the solver control sequence command, and access to the solver commands requires code that accesses that solver control sequence branch.

Step 1: Locate a Branch Command Object

In GMAT, every command that manages branching is derived from the BranchCommand class. That feature can be used to find the branch commands in the MCS by checking each command to determine if it is a branch command:

```
node = startnode.GetNext()
while node != None:
   branchstatus = "does not branch"
   if node.IsOfType("BranchCommand"):
      branchstatus = "is a branch command"
   print(node.GetTypeName(), branchstatus)
   node = node.GetNext()
```

This code block locates the Target command in the Hohmann transfer script and identifies it as a command that has a branch:

```
BeginMissionSequence does not branch
Propagate does not branch
Target is a branch command
Propagate does not branch
```

Step 2: Access the child command from the Branch Command

Now that the branch command has been located, the first command in the branch is retrieved using the GetChildCommand() method, like this:

```
child = node.GetChildCommand()
```

Step 3: Walk through the command branch

Using the child node, the solver control sequence in the Hohmann transfer script can be examined using the same techniques followed for the top level MCS. This can be done using an updated, and now recursive, function to walk the MCS:

```
def WalkTheMCSWithBranches(node, depth = 1, parent = None):
   '''
      Simple function to show a GmatCommand linked list, including children.

      Note that there is a hack in this implementation.  Ideally, we would
      check nodes against the parent pointer, but in Python, we have
      references to SWIG pointers to the GmatCommand object pointers.  It's a
      windy little maze of rooms that all look the same, and that need to be
      unwound when time permits.
```

(continues on next page)

(continued from previous page)

```
      inputs:
         node  The starting node for the command list
         depth The current depth of the call (used for indentation)
         parent The node that has node as a child, or None for a top level node

      returns:
         None.  The command list is written, including branches, to the display
   '''
   count = 1
   indent = "   " * depth

   while node != None:

      print(indent, count, ":  ", '{:20}'.format(node.GetTypeName()), node.GetName())

      # Handle the kiddos
      if node.IsOfType("BranchCommand"):

         child = node.GetChildCommand()
         WalkTheMCSWithBranches(child, depth + 1, node)

      node = node.GetNext()
      count = count + 1

      if parent != None:
         # Note: This is a hack for now.  We want to compare command pointers,
         # but they are buried several levels deep and are not easily accessed
         if node.GetTypeName() == parent.GetTypeName() and node.GetName() == parent.
↪GetName():
            node = None
```

Calling WalkTheMCSWithBranches(node) generates the full mission control sequence with branches when called with node set to the BeginMissionSequence command:

```
1 :   BeginMissionSequence
2 :   Propagate            Prop to Perigee
3 :   Target               Raise and Circularize
  1 :   Vary                 Vary TOI.V
  2 :   Maneuver             Apply TOI
  3 :   Propagate            Prop to Apogee
  4 :   Achieve              Achieve RMAG
  5 :   Vary                 Vary GOI.V
  6 :   Maneuver             Apply GOI
  7 :   Achieve              Achieve ECC
  8 :   EndTarget
4 :   Propagate            Prop 1 Day
```

5.2.4 Viewing Run Results

The instructions provided above show how to access the commands in the MCS. Each command supports a field that reports the state of the spacecraft in the MCS the last time the command was accessed. The summary for a specific command is accessed through the "Summary" field name. Before the script has been run, the summary text data is not populated, and the field returns text to that effect. For example, if the node points to the last propagate command, then accessing the Summary field produces the message

```
>>> print(node.GetField("Summary"))
******  Changes made to the mission will not be reflected ******
******  in the data displayed until the mission is rerun  ******

        Propagate Command: Unnamed
        Command summary is not supported for Propagate (in Single Step Mode)
        or when the command did not execute due to control logic statements.
        Please see the next valid command.
```

After the script has run, the summary contains more useful data:

```
>>> gmat.RunScript()
True
>>> print(node.GetField("Summary"))
******  Changes made to the mission will not be reflected ******
******  in the data displayed until the mission is rerun  ******

        Propagate Command: Unnamed
        Spacecraft       : DefaultSC
        Coordinate System: EarthMJ2000Eq

        Time System   Gregorian                    Modified Julian
        ------------------------------------------------------------------------
        UTC Epoch:    02 Jan 2000 18:32:14.796     21546.2723934740
        TAI Epoch:    02 Jan 2000 18:32:46.796     21546.2727638444
        TT  Epoch:    02 Jan 2000 18:33:18.980     21546.2731363444
        TDB Epoch:    02 Jan 2000 18:33:18.980     21546.2731363440

        Cartesian State                      Keplerian State
        ---------------------------          ---------------------------------
        X  =   7059.8054404060 km            SMA  =   42165.000291122 km
        Y  =   41006.405025033 km            ECC  =   0.0000007749653
        Z  =   6820.5046046645 km            INC  =   12.852951589765 deg
        VX =  -2.9804993875227 km/sec        RAAN =   306.15458093975 deg
        VY =   0.5912233799383 km/sec        AOP  =   180.88186603791 deg
        VZ =  -0.4695068593848 km/sec        TA   =   312.46888463718 deg
                                             MA   =   312.46895014326 deg
                                             EA   =   312.46891739023 deg

        Spherical State                      Other Orbit Data
        ---------------------------          ---------------------------------
        RMAG =   42164.978228331 km          Mean Motion        =   7.291900370e-05 deg/
↪sec
        RA   =   80.231521342877 deg         Orbit Energy       =  -4.7266742410520 km^
↪2/s^2
        DEC  =   9.3089268671491 deg         C3                 =  -9.4533484821041 km^
↪2/s^2
```

(continues on next page)

(continued from previous page)

```
        VMAG =   3.0746314210029 km/s          Semilatus Rectum   =   42165.000291097 km
        AZI  =   98.901788705887 deg           Angular Momentum   =   129641.76692671 km^
↪2/s
        VFPA =   90.000032753035 deg           Beta Angle         =  -17.350375964570 deg
        RAV  =   168.78023389434 deg           Periapsis Altitude =   35786.831314710 km
        DECV =  -8.7836287310829 deg           VelPeriapsis       =   3.0746321949380 km/s
                                               VelApoapsis        =   3.0746274294752 km/s
                                               Orbit Period       =   86166.636794673 s

        Planetodetic Properties
        ---------------------------
        LST       =   80.231355220914 deg
        MHA       =   19.777879001376 deg
        Latitude  =   9.3164365259040 deg
        Longitude =   60.453476219538 deg
        Altitude  =   35787.400879520 km


        Spacecraft Properties
        -----------------------------
        Cd                     =   2.200000
        Drag area              =   15.00000 m^2
        Cr                     =   1.800000
        Reflective (SRP) area  =   1.000000 m^2
        Dry mass               =   850.000000000000 kg
        Total mass             =   850.000000000000 kg
        SPADDragScaleFactor    =   1.000000
        SPADSRPScaleFactor     =   1.000000
```

There are a few additional features worth mentioning:

- When the node used to generate the summary data is a branch command, the summary is reported for that node and for each node in the branch.
- Commands also have an option to report the summary, starting from the selected node, through the end of the control sequence. You can access that report through the "MissionSummary" field of the command.

5.3 Discussion

This chapter provides an overview showing how to use the GMAT API to access the commands in a script and to begin viewing the data associated with the commands. Subsequent chapters show how to change data in selected commands.

CHAPTER

SIX

A COLLECTION OF COMMAND FUNCTIONS

This chapter gathers together functions used when accessing GMAT commands from the API. These functions collect, in a single utility file, the functions used in sunbsequent chapters. The chapter itself is just a listing of the code. Copy the following code into a file named CommandFunctions.py for later use.

6.1 The CommandFunction Code

Listing 6.1: Command access function utilities

```
#!/usr/bin/python3

'''
        Functions used to access Mission Control Sequence commands
'''

from load_gmat import gmat

def GetMissionSequence():
        '''
                Access method for the mission control sequence of a loaded script

                returns:
                        A reference to the first scripted command of the MCS
        '''

        mod = gmat.Moderator.Instance()
        mcs = mod.GetFirstCommand().GetNext()

        if mcs is None:
                print("Please load a script before accessing the MCS")

        return mcs

def FindSolverCommand(node, number = 1):
        '''
                Function used to find a Solver branch command.

                inputs:
                        node The starting node in a control sequence
                        number        A counter allowing access to later nodes.  Set to␣
↪1 for the
```

(continues on next page)

(continued from previous page)

```
                                first SolverBranchCommand, 2 for the second, etc.

                returns:
                        The requested node, or None if the node was not found
        '''
        counter = 0
        retnode = None

        while node != None:
                if node.IsOfType("SolverBranchCommand"):
                        counter = counter + 1
                        if counter == number:
                                retnode = node
                                break
                node = node.GetNext()

        return retnode

def FindCommand(node, type, number = 1):
        '''
                Function used to find a child command of a set type.

                inputs:
                        parent  A branch command node in a control sequence
                        type         The type of node requested
                        number        A counter allowing access to later nodes.  Set to␣
↪1 for the
                                first instance, 2 for the second, etc.

                returns:
                        The requested node, or None if the node was not found
        '''
        counter = 0
        retnode = None

        while node != None:
                if node.IsOfType(type):
                        counter = counter + 1
                        if counter == number:
                                retnode = node
                                break
                node = node.GetNext()

        return retnode

def FindCommandByName(node, name):
        '''
                Function used to find a Solver branch command.

                inputs:
                        node        The starting node in a control sequence
                        name        The name set on the command
```

(continues on next page)

(continued from previous page)

```
                returns:
                        The requested node, or None if the node was not found
        '''
        retnode = None

        while node != None:
                if node.GetName() == name:
                        retnode = node
                        break
                node = node.GetNext()

        return retnode

def FindChild(parent, type, number = 1):
        '''
                Function used to find a child command of a set type.

                inputs:
                        parent  A branch command node in a control sequence
                        type         The type of node requested
                        number        A counter allowing access to later nodes.  Set to␣
↪1 for the
                                first instance, 2 for the second, etc.

                returns:
                        The requested node, or None if the node was not found
        '''
        counter = 0
        retnode = None
        node = parent.GetChildCommand()

        while node != None:
                if node.IsOfType(type):
                        counter = counter + 1
                        if counter == number:
                                retnode = node
                                break
                node = node.GetNext()

        return retnode


def FindChildByName(parent, name, number = 1):
        '''
                Function used to find a child command with a specific name.

                inputs:
                        parent  A branch command node in a control sequence
                        name         The name of node requested
                        number        A counter allowing access to later nodes.  Set to␣
↪1 for the
```

(continues on next page)

(continued from previous page)

```
                        first instance, 2 for the second, etc.

        returns:
                The requested node, or None if the node was not found
    '''
    counter = 0
    retnode = None
    node = parent.GetChildCommand()

    while node != None:
        if node.GetName() == name:
            retnode = node
            break
        node = node.GetNext()

    return retnode
```

CHAPTER

SEVEN

CONTROL OF SCRIPTED SOLVERS

This chapter uses targeting and optimization scripts to demonstrate control of the targeting and optimization processes using the GMAT API. It uses the sample scripts for a Hohmann transfer and for a minimum fuel lunar orbit insertion to show the steps used for control of the related processes. This chapter build on the command access capabilities described in *Accessing GMAT Commands*.

Some of the code presented in this chapter uses the functions defined in *Command access function utilities*, imported through the CommandFunctions package.

7.1 Problem

GMAT's optimizers and targeters provide a set of tools that can be used to tune the parameters used when solving analysis problems. These solver capabilities are scripted in a "Solver Control Sequence." The Solver control sequence is an ordered list of GMAT commands that work together to solve a scripted analysis problem. An example can be found in the Ex_HohmannTransfer sample script. The solver control sequence for from that script is shown in Listing 7.1.

Listing 7.1: The Hohmann Transfer targeting sequence

```
Target 'Raise and Circularize' DC;
   Vary 'Vary TOI.V' DC(TOI.Element1 = 0.5, {Perturbation = 0.0001, ...
       Lower = 0, Upper = 3.14159, MaxStep = 0.2});
   Maneuver 'Apply TOI' TOI(DefaultSC);
   Propagate 'Prop to Apogee' DefaultProp(DefaultSC) {DefaultSC.Apoapsis};
   Achieve 'Achieve RMAG' DC(DefaultSC.Earth.RMAG = 42165, {Tolerance = 0.1});
   Vary 'Vary GOI.V' DC(GOI.Element1 = 0.5, {Perturbation = 0.0001, ...
       Lower = 0, Upper = 3.14159, MaxStep = 0.2});
   Maneuver 'Apply GOI' GOI(DefaultSC);
   Achieve 'Achieve ECC' DC(DefaultSC.ECC = 0, {Tolerance = 0.1});
EndTarget;
```

Sometimes the scripted control sequence needs to be adjusted, either because the goals of the sequence have changed or because the variables are not conditioned properly, leading to an inability to converge on a solution to the analysis problem. API users need to be able to tune the solver sequence through API calls to the underlying objects.

7.2 Solution

The solver control sequences for targeting and for optimization share setting of variables through the Vary command. Targeting specifies the target goals using the Achieve command. Optimization uses the Minimize command to set the objective function that is minimized, and the NonlinearConstraint command to constrain the optimization solution. The API provides access to each of these commands so that their settings can be viewed and changed. The sections below show how to work with each component.

7.2.1 Vary Command Access

Scripting for the Vary command is shown in Listing 7.2.

Listing 7.2: A single Vary command

```
Vary 'Vary TOI.V' DC(TOI.Element1 = 0.5, {Perturbation = 0.0001, ...
       Lower = 0, Upper = 3.14159, MaxStep = 0.2, ...
       AdditiveScaleFactor = 0.0, MultiplicativeScaleFactor = 1.0});
```

The right side of the equality setting for each field in the command can be set in the API. The API also provides access to the most recent evaluated value for the variable. Table 7.1 shows the field names and access attributes available through the API.

Table 7.1: Fields on the Vary Command

Field	Access	Description and notes
SolverName	R	The name of the targeter or optimizer controlling this variable
InitialValue	R/W	Variable value used to start the targeting or optimization
Perturbation	R/W	Perturbation applied to compute the Jacobian numerically. 0.0 not allowed
Lower	R/W	The minimum allowed value for the variable, for solvers that allow control over the minimum.
Upper	R/W	The maximum allowed value for the variable, for solvers that allow control over the maximum.
MaxStep	R/W	The largest change allowed in the magnitude of the variable
AdditiveScaleFactor	R/W	A factor added to the variable value
MultiplicativeScaleFactor	R/W	A factor used to scale the variable
CurrentValue	R	The most recent value of the variable during a run.

API users access these settings after loading a script. They can be accessed prior to a run, or after the script has been run in preparation for a subsequent run. :numref`VaryExample` shows how to set some of the field settings from the API.

Listing 7.3: Sample code for modifying a Vary command

```
from load_gmat import gmat
import CommandFunctions as cf

# Load the Hohmann transfer script into GMAT.
retval = gmat.LoadScript("../scripts/Ex_HohmannTransfer.script")

mcs = cf.GetMissionSequence()

# Locate the Target command
solverCmd = cf.FindSolverCommand(mcs)

# Find the 1st Vary command
vary1 = cf.FindChild(solverCmd, "Vary")

# Change some settings
val = 2.4

 # Set from a Python variable
vary1.SetField("InitialValue", val)
```

(continues on next page)

(continued from previous page)

```
# Set from a numeric value
vary1.SetField("Perturbation", 0.0005)
```

Control of the Vary command parameters is a stepping stone to control of GMAT's targeting and optimization processes, discussed below.

7.2.2 Use Case 1: Targeting

Targeting in GMAT is driven by commands between an opening "Target" command and a closing "EndTarget" command. Example script is shown in Listing 7.4.

Listing 7.4: The start and end of a targeter control sequence

```
Target DC
   % Control sequence, with at least one Vary and one Achieve command, goes here
EndTarget
```

That target control sequence contains the variables, manipulated by a Vary command (described above), a timeline of actions to execute, and the targeting goals, captured in Achieve commands. The Target command provides a field used to find the name of the targeter used, and a simple Boolean field that checks to see if the targeter control sequence has succeeded in finding a valid solution.

Table 7.2: Fields on the Target Command

Field	Access	Description and notes
SolverName	R	The name of the targeter used in the run
Targeter	R	The name of the targeter used in the run
TargeterConverged	R	Boolean flag indicating if targeter convergence has been achieved

The Achieve command (scripting: Listing 7.5) controls the goals of the targeting run.

Listing 7.5: Example of an Achieve command

```
Achieve DC(sat.SMA = 42165.0, {Tolerance = 0.1});
```

The user sets desired values for the targeter goals and tolerances on those values. The targeter retrieves those settings and used them, along

Table 7.3: Fields on the Achieve Command

Field	Access	Description and notes
SolverName	R	The name of the targeter controlling this goal
Goal	R	String identifying the targeter goal
GoalValue	R/W	The desired value of the goal
Tolerance	R/W	The precision needed in the achieved value relative to the desired value in the converged solution
Achieved-Value	R	The value of the goal following a run. Note that this is a numeric field only accessed using GetNumber()

The Achieve fields provide full control over teh goals of the targeting run. Listing 7.6 illustrated their use.

Listing 7.6: Sample code setting the tolerance on an Achieve command and accessing the Target command fields

```
achieve2 = cf.FindChild(solverCmd, "Achieve", 2)
achieve2.SetField("Tolerance", 1e-7)

print("Variable 1:  Initial Value:", vary1.
gmat.RunScript()

print("The targeter", solverCmd.GetField("Targeter"),
   "completed its run with convergence status",
   solverCmd.GetField("TargeterConverged"))

print("The goal", achieve2.GetField("Goal"), "=",
   achieve2.GetField("GoalValue"), "has a tolerance of",
   achieve2.GetField("Tolerance"), "and a final value",
   achieve2.GetNumber("AchievedValue"))
```

The full sample Python code used when writing this section can be found in *Complete example for targeter control*.

7.2.3 Use case 2: Optimization

Optimization in GMAT is driven by commands between an opening "Optimize" command and a closing "EndOptimize" command. Example script for these commands is shown in Listing 7.7.

Listing 7.7: The start and end of an optimizer control sequence

```
Optimize opt
   % Control sequence, with at least one Vary and one Minimize or NonlinearConstraint␣
↪command, goes here
EndOptimize
```

That optimization control sequence contains the variables, manipulated by a Vary command (described above), a timeline of actions to execute, and the optimization objective function, captured in a Minimize command, along with any constraints to be applied to the optimization process, specified using NonlinearConstraint commands. The Optimize command provides a field used to find the name of the optimizer used, and a simple Boolean field that checks to see if the optimization control sequence has succeeded in finding a valid solution. These fields are described in Table 7.4.

Table 7.4: Fields on the Optimize Command

Field	Access	Description and notes
SolverName	R	The name of the optimizer used in the run
OptimizerName	R	The name of the optimizer used in the run
OptimizerConverged	R	Boolean flag indicating if optimizer convergence has been achieved

Linear and nonlinear equality and inequality constraints are specified in NonlinearConstraint commands. The fields of those commands, shown in Table 7.5, provide similar control for optimization to the Achieve controls for targeting.

Listing 7.8: Example of an optimization constraint

```
NonlinearConstraint opt(MMSRef.Luna.SMA = 2300);
```

The constraints, scripted as in Listing 7.8, are specified in the format <arg1> operator <arg2>. The API provides methods for evaluating both the left side argument, arg1, and the right side argument, arg2 using the fields shown in Table 7.5. The value for arg2 can also be specified using the RHSValue field. Tolerances can be set on nonlinear constraints using the Tolerance setting. Users should be cautious with this setting: some optimizers support it at the individual constraint level, while other do not.

Table 7.5: Fields on the NonlinearConstraint Command

Field	Access	Description and notes
SolverName	R	The name of the optimizer controlling this goal
ConstraintArg1	R	The text description of the left hand side of the constraint
LHSValue	R	The most recent (or desired) value of argument 1. Note that the argument to the left of the operator is not a number, and cannot be set.
ConstraintArg2	R	The text description of the right hand side of the constraint
RHSValue	R/W	The most recent (or desired) value of argument 2
Operator	R/W	The operator used to compare the left and right arguments. Valid operators are >=, =, and <=.
Tolerance	R/W	Tolerance setting for equality constrains, for optimizers that support that setting at the constraint level.

The final component used in optimization problems if the objective function, defining the function that needs to be minimized during optimization. In GMAT scripting, that function is set using the Minimize command exemplified in Listing 7.9.

Listing 7.9: Scripting for the objective function definition for optimization

```
Minimize opt(myCostFunction);
```

The API provides access to reading the objective function settings using the fields shown in Table 7.6.

Table 7.6: Fields on the Minimize Command

Field	Access	Description and notes
SolverName	R	The name of the optimizer controlling this goal
OptimizerName	R	The name of the optimizer controlling this goal
ObjectiveName	R	The scripted cost function
Cost	R	The most recent value of the objective function

An example showing the use of these settings is shown in Listing 7.10.

Listing 7.10: Sample code (simplified) showing control of constraints and access to the objective function

```
# See full example (below) for some definitions
```

(continues on next page)

(continued from previous page)

```
nlc =  cf.FindChildByName(solverCmd, "SMA = 2300")
nlc.SetField("RHSValue", 2100.0)

# Run with the scripted settings
gmat.RunScript()

minCommand = cf.FindChild(solverCmd, "Minimize")

print("\n\nObjective function:  ", minCommand.GetField("ObjectiveName"),
      "=" , minCommand.GetNumber("Cost"), "\n")

print("Constraint results:", nlc.GetField("ConstraintArg1"),
      nlc.GetField("Operator"), nlc.GetField("ConstraintArg2"),
      " ==> ", nlc.GetNumber("LHSValue"), nlc.GetField("Operator"),
      nlc.GetNumber("RHSValue"))
```

The full sample Python code used when writing this section can be found in *Complete example for optimizer control*.

7.3 Discussion

This chapter provides usage examples for controlling GMAT scripts that target and optimize trajectories using the Python API.

7.4 Complete Targeter Script

Listing 7.11: Complete example for targeter control

```
#!/usr/bin/python3

'''
        Example showing how to manipulate variables in a solver control sequence
        using the GMAT API.

        This example uses the Hohmann transfer sample script supplied with GMAT.
        Create a folder one level up from the run folder, name it scripts, and copy
        Ex_HohmannTransfer.script from the GMAT samples folder into that folder.
'''

from load_gmat import gmat
import CommandFunctions as cf

# For reporting, set up a local log file
gmat.UseLogFile("./VarExampleLog.txt")
gmat.EchoLogFile()


# Load the Hohmann transfer script into GMAT.
retval = gmat.LoadScript("../scripts/Ex_HohmannTransfer.script")
```

(continues on next page)

(continued from previous page)

```
if retval == False:
        print("The script failed to load.")
        exit()

# Connect to the GMAT engine and access the MCS
mcs = cf.GetMissionSequence()

# Locate the Target command
solverCmd = cf.FindSolverCommand(mcs)

# Find the 1st Vary command
vary1 = cf.FindChild(solverCmd, "Vary")
# Find the 2nd Vary command
vary2 = cf.FindChild(solverCmd, "Vary", 2)

print("Variable 1:  Initial Value:", vary1.GetNumber("InitialValue"),
        "Perturbation:", vary1.GetNumber("Perturbation"),
        "Max Step:", vary1.GetNumber("MaxStep"),)
print("Variable 2:  Initial Value:", vary2.GetNumber("InitialValue"),
        "Perturbation:", vary2.GetNumber("Perturbation"),
        "Max Step:", vary2.GetNumber("MaxStep"))

print("The targeter", solverCmd.GetField("Targeter"),
        "entered its run with convergence status",
        solverCmd.GetField("TargeterConverged"))

input("\nPress Enter to continue...\n")

# Run with the scripted settings
gmat.RunScript()

print("The targeter", solverCmd.GetField("Targeter"),
        "completed its run with convergence status",
        solverCmd.GetField("TargeterConverged"))

input("\nPress Enter to continue...\n")

val = vary1.GetNumber("CurrentValue")

# Now change the settings
vary1.SetField("InitialValue", val)
vary1.SetField("Perturbation", 0.0005)
vary2.SetField("InitialValue", 1.4)
vary2.SetField("MaxStep", 0.02)

achieve2 = cf.FindChild(solverCmd, "Achieve", 2)
achieve2.SetField("Tolerance", 1e-7)

print("Variable 1:  Initial Value:", vary1.GetNumber("InitialValue"),
        "Perturbation:", vary1.GetNumber("Perturbation"),
        "Max Step:", vary1.GetNumber("MaxStep"),)
print("Variable 2:  Initial Value:", vary2.GetNumber("InitialValue"),
```

(continues on next page)

(continued from previous page)

```
        "Perturbation:", vary2.GetNumber("Perturbation"),
        "Max Step:", vary2.GetNumber("MaxStep"))

input("\nPress Enter to continue...\n")

gmat.RunScript()

print("The targeter", solverCmd.GetField("Targeter"),
        "completed its run with convergence status",
        solverCmd.GetField("TargeterConverged"))

print("The goal", achieve2.GetField("Goal"), "=",
        achieve2.GetField("GoalValue"), "has a tolerance of",
        achieve2.GetField("Tolerance"), "and a final value",
        achieve2.GetNumber("AchievedValue"))
```

7.5 Complete Optimizer Script

Listing 7.12: Complete example for optimizer control

```
#!/usr/bin/python3

'''
        Example showing how to manipulate variables in a solver control sequence
        using the GMAT API.

        This example uses the Hohmann transfer sample script supplied with GMAT.
        Create a folder one level up from the run folder, name it scripts, and copy
        Ex_HohmannTransfer.script from the GMAT samples folder into that folder.
'''

from load_gmat import gmat
import CommandFunctions as cf

# For reporting, set up a local log file
gmat.UseLogFile("./VarExampleLog.txt")
gmat.EchoLogFile()

# Load the lunar optimal fuel transfer script into GMAT.
retval = gmat.LoadScript("../scripts/Ex_MinFuelLunarTransfer.script")

if retval == False:
        print("The script failed to load.")
        exit()

# Connect to the GMAT engine and access the MCS
mcs = cf.GetMissionSequence()

# Locate the Target command
solverCmd = cf.FindCommandByName(mcs, "Optimal Transfer")
```

(continues on next page)

(continued from previous page)

```
solverCmd.SetField("ShowProgressWindow", False)

optName = solverCmd.GetField("OptimizerName")
opt = gmat.GetObject(optName)

minCommand = cf.FindChild(solverCmd, "Minimize")

constraints = []

# Collect the constraint list
index = 1
node = cf.FindChild(solverCmd, "NonlinearConstraint", index)

while node:
        constraints.append(node)
        index = index + 1
        node = cf.FindChild(solverCmd, "NonlinearConstraint", index)

# Run with the scripted settings
gmat.RunScript()

print("\n\nObjective function:  ", minCommand.GetField("ObjectiveName"),
     "=" , minCommand.GetNumber("Cost"), "\n")

if len(constraints) > 0:
        print("Constraints:")
        for i in range(len(constraints)):
                print("   ", constraints[i].GetField("ConstraintArg1"),
                        constraints[i].GetField("Operator"),
                      constraints[i].GetField("ConstraintArg2"), " ==> ",
                      constraints[i].GetNumber("LHSValue"),
                      constraints[i].GetField("Operator"),
                      constraints[i].GetNumber("RHSValue"))

input("\nPress Enter to continue...\n")

nlc =  cf.FindChildByName(solverCmd, "SMA = 2300")
if nlc is None:
        print("The command named 'SMA = 2300' was not found")
        exit()
nlc.SetField("RHSValue", 2100.0)

nlc =  cf.FindChildByName(solverCmd, "Inc = 65")
if nlc is None:
        print("The command named 'Inc = 65' was not found")
        exit()
nlc.SetField("RHSValue", 85)

nlc =  cf.FindChildByName(solverCmd, "ECC = 0.01")
if nlc is None:
        print("The command named 'ECC = 0.01' was not found")
```

(continues on next page)

(continued from previous page)

```
        exit()
nlc.SetField("RHSValue", 0.005)

if len(constraints) > 0:
        print("\nConstraints reset:")
        for i in range(len(constraints)):
                print(constraints[i].GetName(), "now set to ", constraints[i].GetNumber(
↪"RHSValue"))

input("\nPress Enter to continue...\n")

# Run with the scripted settings
gmat.RunScript()

print("\n\nObjective function:  ", minCommand.GetField("ObjectiveName"), "=" ,␣
↪minCommand.GetNumber("Cost"), "\n")

if len(constraints) > 0:
        print("Constraints:")
        for i in range(len(constraints)):
                print("   ", constraints[i].GetField("ConstraintArg1"),
                        constraints[i].GetField("Operator"),
                      constraints[i].GetField("ConstraintArg2"), " ==> ",
                      constraints[i].GetNumber("LHSValue"),
                      constraints[i].GetField("Operator"),
                      constraints[i].GetNumber("RHSValue"))
print("\nTa da!\n\n")
```