DRAFT GMAT Application Programming Interface

Release R2020a

Darrel Conway and John McGreevy

Apr 13, 2020

CONTENTS:

1 Introduction **1**

2 System Design **3**
- 2.1 GMAT Architectural Components . . . 3
- 2.2 GMAT API Prototypes . . . 5
- 2.3 API Examples . . . 6
- 2.4 The API Design . . . 11
- 2.5 GMAT Code Updates . . . 12
- 2.6 API Support Functions . . . 14

3 GMAT API User's Guide **17**
- 3.1 Conventions in the API Examples and Use Cases . . . 17
- 3.2 Setting up the GMAT API . . . 18
- 3.3 Object Usage with the GMAT API . . . 22
- 3.4 Overview . . . 22
- 3.5 Functions Used in the GMAT API . . . 24

4 Script Usage **27**
- 4.1 API Functions for Script Users . . . 27
- 4.2 Examples . . . 28

5 Tutorial: Accessing GMAT Propagation and Navigation Features **31**
- 5.1 Verifying Setup . . . 31
- 5.2 Getting Started with the API . . . 32
- 5.3 Spacecraft Configuration . . . 33
- 5.4 Force Model Setup . . . 35
- 5.5 Propagator Setup . . . 38
- 5.6 The GMAT API and Plug-in Modules . . . 40
- 5.7 Measurement Modeling . . . 42

6 Example: MONTE-GMAT Interoperability for OSIRIS_REx **47**
- 6.1 Ephemeris Sharing . . . 47
- 6.2 Maneuver Planning . . . 48
- 6.3 Navigation Data Sharing . . . 48

7 API Best Practices **49**
- 7.1 General Practices . . . 49
- 7.2 Java and MATLAB Best Practices . . . 50
- 7.3 Python Best Practices . . . 50

A Review and Prototype Information **51**

A.1 Open issues . . . 51
A.2 Review Responses . . . 52
A.3 Comparison of the SWIG Prototype with the Production API . . . 57
A.4 API Examples in Java . . . 62

B GMAT API Cheat Sheet **65**
B.1 Loading the API . . . 65
B.2 Asking for Help . . . 65
B.3 GMAT Objects . . . 65
B.4 GMAT Script Access . . . 67

C API Notebook Walkthroughs **69**
C.1 State Management with the GMAT API . . . 69
C.2 Propagation with the GMAT API . . . 73

D Bibliography **81**

E Change History **83**

CHAPTER

ONE

INTRODUCTION

The General Mission Analysis Tool, GMAT, is a general purpose spacecraft mission design, analysis, and operations tool. GMAT is in active development, and has been used for multiple spacecraft missions. GMAT is the operational tool for maneuver design and navigation on several missions in the Goddard Space Flight Center's Flight Dynamics Facility. It is used for mission design at GSFC and at numerous other organizations throughout the world. GMAT is a free and open source tool, available at the GMAT wiki *[GmatWiki]*.

Core capabilities of GMAT can be accessed using an Application Programming Interface (API). This document describes the GMAT API, and includes sample usage from Python and MATLAB using Java.

GMAT is coded using an object oriented approach documented in the GMAT Architectural Specification *[Architecture]*. The system has been under development since 2002. Users interact with GMAT through a spacecraft domain specific language built into the system, modeled on the MATLAB programming language. The GMAT API opens the system's object model to users that want to interact directly with the core system components, outside of the scripted interfaces used when running the application.

The materials presented in this document are divided into two sections:

- The first section documents the design of the API. In it, you will find an extremely high level overview of the GMAT code and a matching API overview, a discussion of the philosophy governing the API that includes use cases, and a description of the additions to the GMAT code base added for the API that enable the examples and features requested by the user community.
- The second section is a user's guide for the GMAT API. It contains instructions for installing the API code, a "Getting Started" tutorial for initial use of the API, and additional hints, tips, and use case descriptions designed to help you start using the GMAT API.

This documentation concludes with appendices that provide additional API guidelines, review notes, and other information for developers and API users.

CHAPTER

TWO

SYSTEM DESIGN

2.1 GMAT Architectural Components

The GMAT API exposes core GMAT classes and processes to API users. In order to understand the API, it is useful to understand at a high level how GMAT works, and how the API encapsulates the GMAT design for use outside of the program.

2.1.1 The GMAT Architecture

The GMAT system consists of a set of components that set up a framework for executing spacecraft mission analysis simulations, and a set of components used to define and run the simulation. The former is referred to as the GMAT engine. The latter defines the components that users script when they run a simulation. Fig. 2.1 shows the connections between the components of the GMAT engine. The main control element of GMAT is a component called the Moderator. GMAT's Moderator provides an interface into the inner working of GMAT, and manages user interactions with the system. Simulations are run in a Sandbox, using clones of user objects created in factories and stored in the GMAT configuration. Users interact with GMAT through interpreters that convert script or GUI descriptions of simulation components into the objects used for the simulation. The results of a simulation are passed, through a Publisher, to subscribing components, which write files or display data for the user. In a nutshell, that is the architecture shown in Fig. 2.1.

Another view of the components used in GMAT is shown in the component stack diagram in Fig. 2.2. GMAT is built on a set of utility functions used for string, vector and matrix manipulations, core numerical operations, file manipulations, and general purpose time and state representations. The GMAT Engine components are built on these utilities, as are the classes defining the objects used in a GMAT simulation. User interfaces into the GMAT system are built on top of these core elements of the system.

All of the user configured simulation components are built on a class, GmatBase, that provides the serialization interfaces used to set the properties of the components through object fields. GmatBase provides the interfaces used when reading and writing simulation objects, either to script files or to panels on the GMAT graphical user interface. Resources - objects like the spacecraft model, coordinate systems, force models, propagators, environmental elements, hardware components, and numerical engines used for estimating, targeting, and optimizing - are all built on top of the GmatBase class, as is the mission timeline scripted as a sequence of GMAT commands. The mission timeline is referred to as the GMAT mission control sequence in the documentation.

Most users of the GMAT API do not interact directly with the components of the GMAT engine. Those components are described in the GMAT Architectural Specification *[Architecture]*. Typical users of the GMAT API fall into two groups: users that use GMAT components in their work, and users that manipulate scripted components prior to, during, or after the execution of a script. The API design documented below focusses on these users.

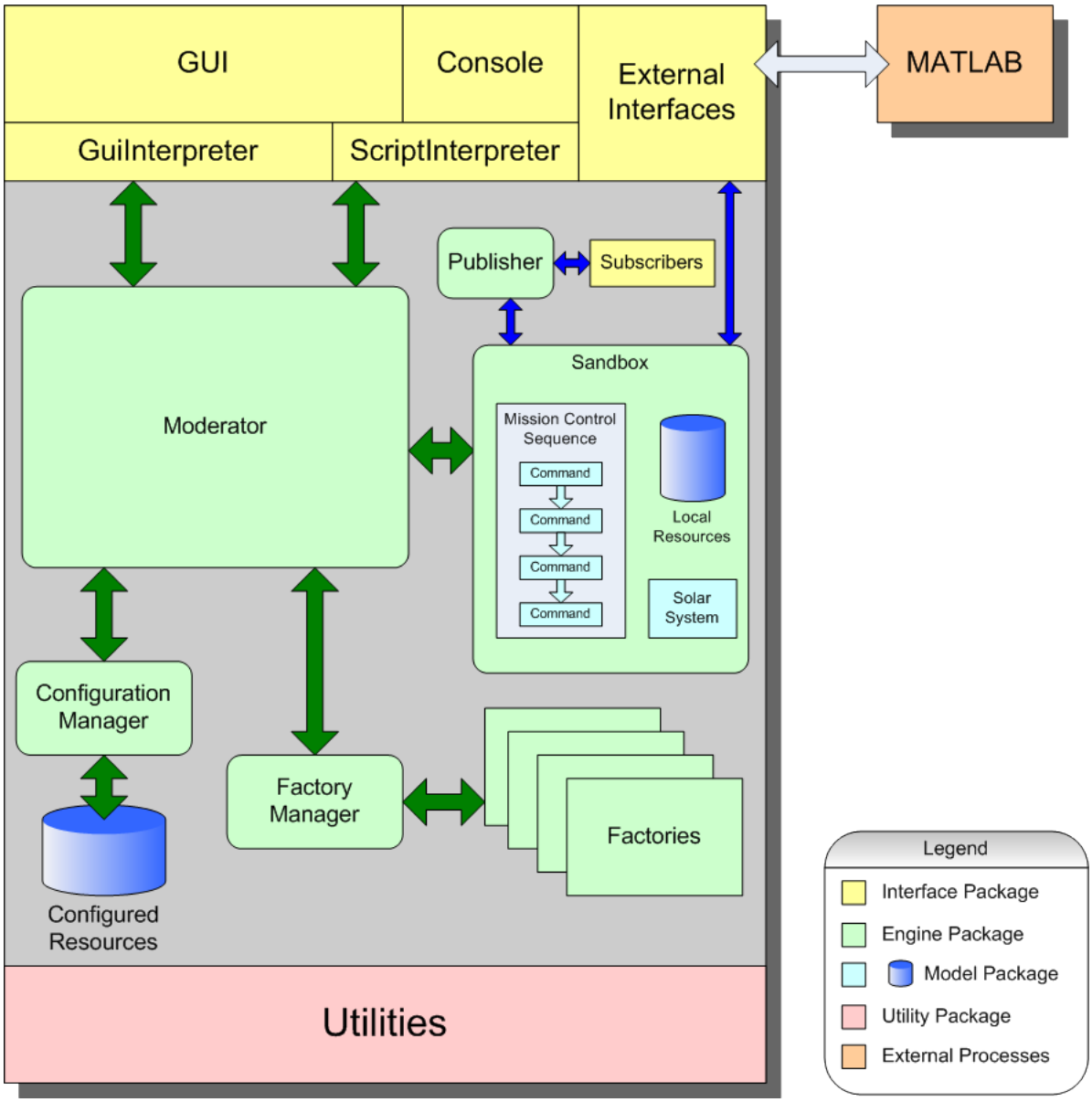


Fig. 2.1: The GMAT engine, showing interactions between the components.

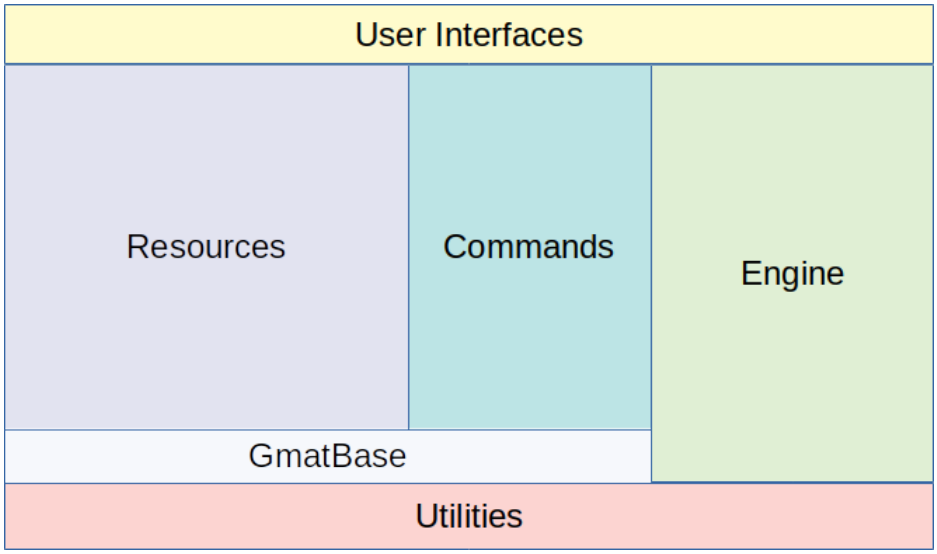


Fig. 2.2: The GMAT component stack.

2.2 GMAT API Prototypes

The GMAT development team has experimented with making the GMAT code available for external use two previous times, targeting information gathering leading to a production GMAT API. These experiments provide lessons that are incorporated into the design presented in this document.

2.2.1 A short history of the GMAT API

In 2011, core features of GMAT were exposed for access to internal projects at Goddard Space Flight Center (GSFC) using a plugin library *[CInterfaceAPI]*. That plugin, libCInterface, allows access to objects initialized in the GMAT Sandbox. Calls to those objects provide derivative information used by the Orbit Determination Toolbox (ODTBX) project at GSFC, and by other system users. This application was the first attempt at an API into GMAT computations for external users. While this early interface provided a needed piece of functionality, the nature of the C Interface plugin required code changes and a system rebuild when a new component needed to be accessed by users of the plugin. The C Interface code generated during this development cycle is exposed for MATLAB access using the MATLAB shared library loading mechanism. The C Interface plugin is still part of the GMAT delivery packages at this writing. Once the API is sufficiently complete, the C Interface plugin will be eligible for replacement by the newer, more complete API.

A new experiment building a GMAT API prototype was built as an internal research and development project at GSFC in 2016 *[SWIGExperiment]*. The results of that study identified the Simplified Wrapper and Interface Generator (SWIG) as a useful mechanism for creating a robust GMAT API capable of exporting most of the GMAT code for use by external projects. The 2016 API prototype has been used to test interfaces with ODTBX, and works as expected in that context. Users of the new approach to access of GMAT capabilities can create and use a much more complete set of components than were available in the C Interface plugin. The component exposure using this interface opens GMAT's code to more general use, and allows access from a variety of programming languages including Python, Java, and C#. The SWIG approach allows API generation from both core GMAT code and from GMAT plugin components, a feature unavailable with the prototype C Interface plugin. The SWIG API generated from this work is available in an API branch of the GMAT repository at GSFC.

2.2.2 User Experiences

Users at GSFC were surveyed at the start of the production API work that produced the design documented here. The users surveyed include users familiar with both of the API experiments described above. Feedback was collected for the usability of the earlier API systems, and for projected needs at GSFC. The highlights of the user experiences informed the API design process. Key elements that are targeted for the production API can be grouped into three use styles, three application frameworks (four is C++ is included), two near term needs, and a single overriding usage requirement:

- Use Styles
  - Users of GMAT scripts that want to change scripted data during a run
  - Analysts that want an easy-to-use toolbox of validated astrodynamics components
  - Users that want to interact at a detailed level with instances of GMAT classes
- GSFC Desired Application Frameworks
  - Python
  - Java
  - MATLAB (via Java)
  - C++ (See note below)
- Near Term Needs
  - Dynamics Modeling and Propagation
    - Dynamics models must include Jacobian data
    - Propagation should be available for all of GMAT's propagators
  - Measurement Models from the Estimation Plugin
- Usage
  - Users need to be able to use the API without detailed knowledge of GMAT code
    - The API needs usage documentation
    - Users need online access to available API and object settings

These feedback considerations provide guidance for the GMAT API described here.

Note: C++ and the API

GMAT is coded in C++. The tool used to generate the API, SWIG, provides interface code that exposes the native C++ code to users on other development platforms. SWIG presumes that C++ coders will simply call into the native code directly. There is no "C++ API" per se, but the functions added to GMAT to support the API on the target platforms can be called from a developer's C++ code. Some functionality, like the help system, designed for interactive platform use, is of limited use for users of compiled C++ code.

2.3 API Examples

Four Python sample use cases were coded using the prototype SWIG API to act as a guide to addressing the changes that are needed for the production system. These cases ranged from a trivial time system conversion use case to a full propagation use case. These cases, shown in *Comparison of the SWIG Prototype with the Production API*, were then reworked into the API syntax documented here. The following section previews the changes coming to the GMAT API

by presenting each of these cases as planned for the API. The examples presented here are in Python. Java examples are presented in *API Examples in Java*.

2.3.1 Case 1: Time System Conversion

GMAT supports five time systems: A.1 Atomic Time (A1), International Atomic Time (TAI), Coordinated Universal Time (UTC), Barycentric Dynamical Time (TDB), and Terrestrial Time (TT). Times in GMAT are stored internally in a modified Julian format, referenced to January 5, 1941 at noon. Conversions between these time systems are performed using a time system converter, coded in the TimeSystemConverter singleton class. The time system converter also provides routines to convert between modified Julian representations and Gregorian representations.

The simplest usage of the time system converter using GMAT's API consists of two lines of code; lines 4 and 8 shown here:

Listing 2.1: Python code for time conversions using the GMAT API

```
1 import gmatpy as gmat
2
3 # Get the converter
4 timeConverter = gmat.theTimeSystemConverter
5
6 # Convert an epoch
7 UTCEpoch = 21738.22145
8 TAIepoch = timeConverter.Convert(UTCepoch, UTC, TAI)
```

This code shows two features of the API. The first line shows how a Python user loads the GMAT system and initializes it for use. As part of the initialization process, several components are created in the GMAT module that are single instance objects, following a singleton design pattern. These singletons are accessed from the API using names prefixed by the character string "the". Line 4 is an example of this usage. The time system converter singleton is accessed using the object name theTimeSystemConverter. In the python code, the object "timeConverter" is connected to the singleton, and then used to convert a UTC epoch to the TAI time system on line 8.

At this point, the singleton is ready for the user to ineract with it directly. Using the API, the conversion is immediately available:

```
>>> timeConverter.ConvertMjdToGregorian(UTCEpoch)
'12 Jul 2000 17:18:53.280'
>>> TAIEpoch = timeConverter.Convert(UTCEpoch,2,1)
>>> timeConverter.ConvertMjdToGregorian(TAIEpoch)
'12 Jul 2000 17:19:25.280'
```

The interactive call shows the correct time system difference arising from the number of leap seconds needed to convert from UTC time to TAI time. Users can access the leap second count at a specified epoch directly as well:

```
>>> timeConverter.NumberOfLeapSecondsFrom(TAIEpoch)
32.0
```

2.3.2 Case 2: Coordinate System Conversion

The time system converter is a stand alone component in GMAT. It does not require external components to perform conversions. Coordinate systems are more complex. They require connections to other objects in order to compute data. Table 2.1 shows the settings, required and optional, to define a GMAT coordinate system.

Table 2.1: Coordinate System Settings

Field	Type	Required?	Description
AxisType	AxisSystem object	Yes	Defines the orientation of the coordinate axes
Origin	SpacePoint object	Yes	Defines the coordinate system origin
Primary	SpacePoint object	No	Reference body used in coordinate systems that need a primary body
Secondary	SpacePoint object	No	Reference body used in coordinate systems that need a secondary body
J2000 Body	Internal reference object	Yes	Reference origin in GMAT (always set to Earth)
Solar System	Solar System object	Yes	The solar system used in the run

Coordinate systems are defined in GMAT as a collection of objects, using a core composite component that collect together the axis system defining the directions for the coordinate system axes, the bodies used to set the coordinate system origin and the axis references, and core GMAT settings used to tie the coordinate system into the rest of the executing GMAT code. Many of the user objects in GMAT have a structure like this: a core object that uses other objects to form a composite component consistent with the rest of the running GMAT system.

GMAT performs conversions between coordinate systems using a coordinate system converter, coded in the CoordinateConverter class. The CoordinateConverter class maintains state information about the most recent conversion performed. This state data would cause issues using a state converter in a single instance context, because the state data from one conversion could be accessed in code requesting the state data from a second conversion. For that reason, the CoordinateConverter class does not provide a singleton instance, and a separate object must be created for each use.

A basic use case for the coordinate system converter takes a state in Earth-centered Mean-of-J2000 Equatorial coordinates and converts the state into Earth-centered Earth-fixed coordinates. The Python code demonstrating this conversion using the GMAT API is

Listing 2.2: Python code for coordinate system conversions

```
import gmatpy as gmat

# Setup the GMAT data structures for the conversion
mjd   = gmat.A1Mjd(22326.977184)
rvIn  = gmat.Rvector6(6988.427, 1073.884, 2247.333, 0.019982, 7.226988, -1.554962)
rvOut = gmat.Rvector6(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

# Create the converter
csConverter = gmat.CoordinateConverter()

# Create the input and output coordinate systems
eci  = gmat.Create("CoordinateSystem",
    "ECI", "Earth", "MJ2000Eq")
ecef = gmat.Create("CoordinateSystem",
    "ECEF", "Earth", "BodyFixed")

csConverter.Convert(UTCepoch, rvIn, eci, rvOut, ecef)
```

The key features shown in this example for the API are:

- Default objects (like the solar system and Earth objects) are set automatically.
- GMAT objects are built using the Create() command.

2.3.3 Cases 3 and 4: Force Modeling and Propagation

Force models in GMAT – or, more properly, dynamics models – are built by creating an object from the force container class, ODEModel, and adding the constituent forces to that container. The ODEModel object is responsible for accumulating the dynamics into a derivative vector. The dynamics are computed when the GetDerivatives() method is called on the object. The resulting computation is stored in a class member, accessible using the GetDerivativeArray() method.

The ODEModel class is one component of a more complicated propagation subsystem in GMAT. That subsystem is designed with spacecraft propagation in mind. Force modeling requires an associated spacecraft object. During initialization, an instance of the GMAT helper class, PropagationStateManager (PSM), is used to collect data and assemble the state vector used to evaluate the dynamics. The PSM determines the size of the state vector by checking to see if the 6 element Cartesian state, mass flow from spacecraft tanks, and the state transition matrix or state Jacobian (A-matrix) are needed during the propagation. Once the size of the propagation state vector is determined, the complete vector is assembled and initialized, and only then can the dynamics be evaluated.

The steps required for this initialization are largely implemented behind the scenes in the GMAT API. Users that want to manage this setup by hand are referred to the sample code in *Comparison of the SWIG Prototype with the Production API*. The third example in that chapter shows force model configuration. API users can access GMAT's dynamics models by configuring the forces piece by piece and assigning them to an ODEModel container. Here is an example of this process for an Earth point mass force model:

Listing 2.3: Python code for force modeling using the API

```
import gmatpy as gmat

dynamics = gmat.Create("ODEModel", "EPointMassDynamics")
epm = gmat.PointMassForce("EarthPointMass")
dynamics.AddForce(epm)

# Evaluating only the 6 element Cartesian state
pstate = [7000.0, 0.0, 1000.0, 0.0, 8.0, -0.25]

gmat.Initialize()

pderiv = dynamics.GetDerivativeArray()
dynamics.GetDerivatives(pstate, 0.0)
```

This basic example uses a hidden Spacecraft object for part of the configuration. When this type of automatically generated object is used, the API notifies the user that the object was created, and supplies the user with the object's name. In general, API users will want to create and assign a spacecraft to the force model, so that spacecraft properties that affect the forces will be under the control of API user. A more complete example of this procedure, adding a solar radiation pressure force and a spacecraft used for the corresponding ballistic data, might look like this:

Listing 2.4: Python code for force modeling requiring a spacecraft

```
import gmatpy as gmat

# Set up the spacecraft
sat = gmat.Create("Spacecraft", "Sat")
sat.SetField("Cr", 1.4)
sat.SetField("SRPArea", 6.5)
sat.SetField("DryMass", 225)

# Make a force container
dynamics = gmat.Create("ODEModel", "EPM_SRP")
dynamics.SetObject(sat)
```

(continues on next page)

(continued from previous page)

```

# Set the forces
epm = gmat.PointMassForce("EarthPointMass")
srp = gmat.SolarRadiationPressure("SRP")
dynamics.AddForce(epm)
dynamics.AddForce(srp)

# Evaluating only the 6 element Cartesian state
pstate = [7000.0, 0.0, 1000.0, 0.0, 8.0, -0.25]

gmat.Initialize()

pderiv = dynamics.GetDerivativeArray()
dynamics.GetDerivatives(pstate, 0.0)
```

The setup for dynamics modeling above extends with little additional configuration to numerical integration. The integration piece of the configuration adds the lines

Listing 2.5: Propagator setup and use in the API

```
prop = gmat.Create("PrinceDormand78", "MyIntegrator")
prop.SetPhysicalModel(dynamics)

gmat.Initialize()

for i in range(10):
   prop.Step(60.0)
```

To summarize: The goal of the GMAT API is to make API based configuration as simple as possible, while maintaining full access to the capabilities of GMAT. Towards that end, the design of the API can be illustrated through a representative example implementation of a propagation problem. A reference propagation in the GMAT API looks like this:

Listing 2.6: The Propagation example in Python

```
import gmatpy as gmat

# Set up the spacecraft
sat = gmat.Create("Spacecraft", "Sat")
sat.SetField("Cr", 1.4)
sat.SetField("SRPArea", 6.5)
sat.SetField("DryMass", 225)

# Make a force container and set its spacecraft
dynamics = gmat.Create("ODEModel", "EPM_SRP")
dynamics.SetObject(sat)

# Set the forces
epm = gmat.PointMassForce("EarthPointMass")
srp = gmat.SolarRadiationPressure("SRP")
dynamics.AddForce(epm)
dynamics.AddForce(srp)

# Propagator configuration
prop = gmat.Create("PrinceDormand78", "MyIntegrator")
prop.SetPhysicalModel(dynamics)
```

(continues on next page)

(continued from previous page)

```
22
23 gmat.Initialize()
24
25 # Number of steps to take
26 count = 60
27
28 for i in range(count):
29     prop.Step(60.0)
```

2.3.4 Cases 5: Working with a GMAT Script

The script examples above show how an API user interacts with GMAT components directly. This final example shows how a user can work with an existing GMAT script that needs to change settings on one of the scripted objects. For this example, the GMAT sample mission that demonstrates finite burns is used. The full script is the Ex_FiniteBurn.script file in the GMAT samples folder. Part of that script includes the definition of a chemical thruster, shown through the thrust vector portion here:

Listing 2.7: The Thruster in GMAT's FiniteBurn Sample Mission

```
1 Create ChemicalThruster engine1;
2 GMAT engine1.CoordinateSystem = Local;
3 GMAT engine1.Origin = Earth;
4 GMAT engine1.Axes = VNB;
5 GMAT engine1.ThrustDirection1 = 1;
6 GMAT engine1.ThrustDirection2 = 0;
7 GMAT engine1.ThrustDirection3 = 0;
8 ...
```

An API user might want to change the thrust direction before running the script. The following API code loads the script, adds an orbit normal component to the thrust direction and then runs the script.

Listing 2.8: Changing an Object and then Running a Script

```
1 import gmatpy as gmat
2
3 gmat.InterpretScript("../samples/Ex_FiniteBurn.script")
4
5 Thruster = gmat.GetConfiguredObject("engine1")
6 Thruster.SetField("ThrustDirection2", 1.0)
7
8 gmat.RunScript()
```

2.4 The API Design

Fig. 2.2 shows an overview of the GMAT component stack. The stack for the GMAT API, shown in Fig. 2.3, has a similar appearance. Users interact with the GMAT API through an interface layer built using the Simplified Wrapper and Interface Generator, SWIG. SWIG generates interfaces and shared libraries for Python and Java, and can generate similar interface code for other languages when needed. Classes in GMAT's code base are exposed through this interface using language specific wrappers. Users interact with the GMAT classes through these wrappers.

Using the SWIG interface code, users can work directly with GMAT classes on a class by class/object by object level. Users that work this way need a pretty complete understanding of object linkages and interactions in GMAT. Using

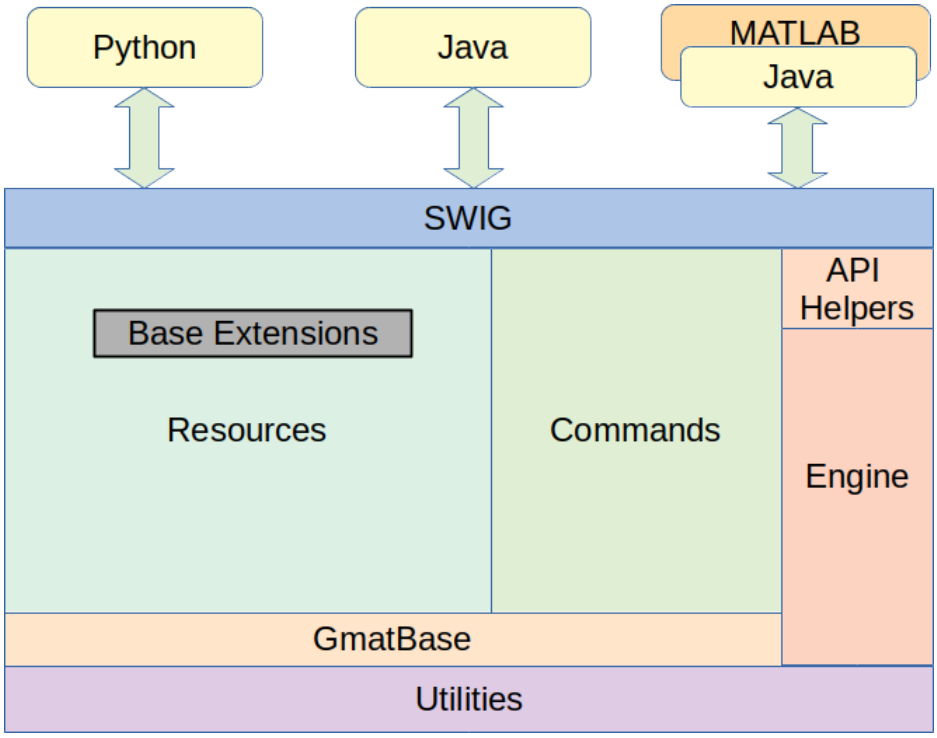


Fig. 2.3: The GMAT API stack.

that expertise, they either imitate many of the steps that are performed by the GMAT engine when GMAT is run or make calls to the components of the engine to perform the required actions.

Most users would rather work at a less detailed level than this object by object interaction. There are two groups of users in this category: those that are familiar with GMAT and want to use the API to run GMAT scripts, making API calls to adapt their scripts along the way, and those that want to use capabilities provided by GMAT inside of models that they are running in a tool like MATLAB or Python, or in a compiled application written in a language like Java. The API provides a set of helper functions that encapsulate the GMAT engine behind calls that simplify the management tasks of the GMAT engine for these users. These API helpers are exposed through the SWIG interface layer for use by these API users.

A driving feature of the GMAT API is the incorporation of usability features for the API user community. During the prototyping exercise for the API, the development team found that the SWIG system provides a simple mechanism for exposing GMAT components in the Python, Java, and MATLAB environments. However, users working in those systems still found it difficult to use the prototype API because of a lack of on line documentation and apparent inconsistencies in the methods in GMAT. The production API addresses the first of these issues through the incorporation of class and object level help functions for classes that are identified as “API ready.” Interface inconsistencies are addressed through the addition of methods to the source code that simplify the class and object interfaces, leaving in place where necessary the interfaces that appear to API users to be inconsistent because of internal code needs in the GMAT system.

2.5 GMAT Code Updates

The GMAT source code was written to be fast and to meet the needs of analysts modeling spacecraft guidance, navigation and control. The code includes functions used to set up mission simulations through a set of interfaces that require that users know the underlying data structures in the code. API users do not necessarily have that knowledge, nor do they have a graphical user interface to guide them through setup for the components accessed using the API.

Several code elements presented here are designed for integration into the base class for user components to meet these needs, adding functionality to the GmatBase class used for simulation components in GMAT, to facilitate API usage.

2.5.1 SetField() and GetField()

Parameter settings on GMAT objects are made through type specific methods with names like SetStringParameter(), SetRealParameter(), etc. API users can use those methods. A second set of methods, SetField(), will be added to the code to adapt user objects to less strongly typed interfaces. The SetField() methods are overloaded to adapt to multiple types of input data. The method checks the type for the field that being set, and sets it accordingly.

Table 2.2: The Field Setting Methods, with Examples for a Spacecraft Named sat

Method Signature	Example	Parameter Description
bool SetField(name, value)	sat.SetField("X", 7100.0)	name is the field name used in GMAT scripting value is the value that is set
bool SetField(id, value)	*% Spacecraft.X has ID 13* sat.SetField(13, 7100.0)	id is the id of the field in GMAT's object value is the value that is set

The SetField() method takes either a field name or an integer ID value to identify the field to be set, and a string, integer, real number, or Boolean value as the new field setting. On success, the call to SetField() returns true. If the value passed to the SetField() method is incompatible with the data structure that is being set, or if the field is not a supported field on the object, the method returns a false boolean value.

Parameter settings on GMAT objects are accessed through type specific methods with names like GetStringParameter(), GetRealParameter(), etc. API users can use those methods. A second set of methods, GetField(), have been added to the code to adapt user objects to less strongly typed interfaces. The GetField() methods return a string containing the value of the field. The API users then convert that string into the form needed for their code.

Table 2.3: The Field Access Methods, with Examples for a Spacecraft Named sat

Method Signature	Example	Parameter Description
string GetField(name)	sat.GetField("X")	*name* is the field name used in GMAT scripting
string GetField(id)	sat.GetField(13)	*id* is the ID of the field in GMAT's object
real GetNumber(name)	sat.GetNumber("X")	*name* is the field name used in GMAT scripting
real GetNumber(id)	sat.GetNumber(13)	*id* is the ID of the field in GMAT's object

The GetField() method takes either a field name or an integer ID value to identify the field to be retrieved. On success, the current field value is returned in a string. If the field is not a supported field on the object, the method returns an empty string.

The GetNumber() method returns numerical data as a real number rather than as a string. If the field is not a numerical field on the object, the method reports an empty exception that identifies the type of the field.

2.5.2 Object help

Objects created in the API support a help function that is used to retrieve information about the settings available for the object, along with other information provided in the object's class.

API users retrieve help for created objects by calling the Help() method on the object. For example, a call to the Help() method for an ImpulsiveBurn object, IB, returns the following data *(Note: this is a mock-up of the data returned)*:

```
>>> IB = gmat.ImpulsiveBurn("Impulse1")
>>> print(IB.Help())

Impulse1 is an ImpulsiveBurn object with 9 Fields:

   CoordinateSystem     Type: Object   Value: Local
   Origin               Type: Object   Value: Earth;
   Axes                 Type: Enum     Value: VNB;
   Element1             Type: Real     Value: 25;
   Element2             Type: Real     Value: 0;
   Element3             Type: Real     Value: 0;
   DecrementMass        Type: Boolean  Value: false;
   Isp                  Type: Real     Value: 300;
   GravitationalAccel   Type: Real     Value: 9.81;
```

GMAT programmers may provide additional help for specific objects on a case by case basis by overriding the Help() method. The API Help command calls into the object's Help() method when an object is passed to it, producing the same output. In other words, the output from IB.Help() is the same as the output from gmat.Help(IB).

2.6 API Support Functions

API users can work directly with GMAT objects, setting interobject connections and initializing objects to prepare them for use. However, much of the GMAT object model is built on composite objects, consisting of a core component that uses other GMAT components to jointly complete a task. In many cases, it is easy to miss making connections that are made automatically in the GMAT engine, leading to confusion about the component usage. The API includes helper functions that ease the burden of GMAT internal connections for users of typical components.

Table 2.4 provides a list of functions supplied as part of the GMAT API that help work with GMAT components without the need for detailed understanding of the GMAT code base. These functions work with the GMAT engine, hiding the engine functionality from the user while providing services that the user would otherwise need to code by hand with objects built with the API. Each function is described briefly below.

Table 2.4: Helper Functions in the GMAT API

Function	Options	Description
Initialize()		Initializes GMAT objects and establishes object to object connections. This command is reentrant. Subsequent calls reconnect objects and continue the process of preparing the system for use.
Status()		Returns a string reporting any known configuration issues. Users call this function if the Initialize() call reported an issue.
Create()	ObjectType, ObjectName	Creates an object of the input type, with the input name. The created object is then managed inside of the API.
ShowObjects()		Lists all of the named objects that exist in the current run.
ShowClasses()	Category	Lists all of the creatable object types of the selected category. Example: ShowClasses("Propagator") lists all of the propagators available for creation.
Clear()	ObjectName	Clears the GMAT engine by deleting all of the user created components managed in the API. Selecting an object by name in the call to Clear() results in deletion of that objects, leaving the other objects available for use.
Help()	Topic	Shows the API help. Top level help is displayed if no topic is selected. The top level help includes a list of topics for the system. If the user enters a topic, the corresponding help is displayed.

Initialize The Initialize() function is used to initialize GMAT objects. Each call to Initialize() generates a pass through the GMAT objects constructed using the Create() command, setting the inter-object connections that can be set and tracking any issues that prevent component use. The function returns true if the initialization succeeded, and false, along with a list of encountered issues, if there were problems.

Status The Status() method returns a list of issues found when the Initialize() method was called. If there are no known issues, the call to Status reports the number of managed objects in the current run.

Create The Create() function is used to create objects that are managed inside of the GMAT engine running under the API, including components exposed to the API from plugin libraries. All named objects created using the Create() method are built through the GMAT Moderator and passed to the GMAT ConfigurationManager singleton. These actions are invisible to the API user, and provide the mechanism used to manage objects in the API.

ShowObjects The ShowObjects() function is used to show a list of all objects managed in the API. Note that objects created through direct calls to a class's constructor rather than through the Create() command are not managed in the API, and will not be part of the returned list.

ShowClasses The ShowClasses() function is used to list all of the available classes of objects available for creation is an input category.

Clear The Clear() function is used to clear the configuration of created objects. When an object name is included as a parameter in the function call, all other objects will remain available for use.

Help The Help() function is used to get help from inside of the API.

CHAPTER

THREE

GMAT API USER'S GUIDE

This chapter contains information and general guidelines for using the GMAT API, along with example use cases, presented in a tutorial fashion.

3.1 Conventions in the API Examples and Use Cases

The usage section of this documentation shows API users how to perform several common tasks using the GMAT API.

3.1.1 General Conventions

- Numbers that would normally display as 16 digits are truncated to fit on the page when necessary.
- Extraneous white space has been removed from some output.
- Interactive and scripted code segments are shown offset in special blocks, like this:

```
>>> import gmatpy as gmat
```

  Interactive Python blocks, like the one shown above, include Python's triple bracket marker. Blocks from Python script files do not have this marker:

```
import gmatpy as gmat
```

  Interactive MATLAB elements are displayed similarly, with MATLAB's Command Window line marker:

```
>> load_gmat
```

Note that the typesetting for MATLAB is also different from the Python settings.

3.1.2 User Workspace and GMAT Typography

This documentation provides examples in the two application environments supported by the core GMAT API development team: Python and MATLAB (via Java). The API is built using SWIG, and can be built for other platforms. This documentation does not address other platforms.

Work performed using the API involves interactions with objects created in GMAT that are accessed in the user's application environment. This results in references in the user's environment of objects in the GMAT environment. The following conventions are observed in this document to help clarify this component dichotomy:

- Objects created in GMAT use camel-cased names. User references to those objects are presented in lowercase. For instance, the Python script line

```
mysat = gmat.Construct("Spacecraft", "MySat")
```

creates a GMAT Spacecraft object named "MySat," stored in GMAT, that a user accesses through their mysat environment variable.

3.2 Setting up the GMAT API

The GMAT API is included in the GMAT release code beginning with the GMAT R2020a release of the system on SourceForge. The API subsystem included in these packages has been tested using Python and MATLAB calls to the API.

3.2.1 Installation

Installation of the API is complete when the GMAT release bundle is installed on the user's workstation.

Direct Usage

The API can be used immediately by users that want to run it from the GMAT bin directory. Simply open a session in MATLAB or Python, change directories to the GMAT bin folder, and load the GMAT system into the executing environment. The examples shown later in this document can then be run directly in that environment.

Running the API Outside of the GMAT Folders

Users that want to run the API from a different location from the GMAT system need to perform additional steps to configure the system to address two features of the system:

1. GMAT uses a set of data files for its core functions.
2. Running environments need to be able to find the GMAT API interfaces.

The following paragraphs address the configuration settings for these two items.

File Location Access

GMAT uses a text data file, gmat_startup_file.txt, to identify and locate GMAT plug-in components, planetary ephemerides, gravitational potentials, and a large variety of other data files required during a run. The GMAT API operates using this data file by default, but that causes problems when running outside of the GMAT folder structure. Rather than change the GMAT startup file, API users can accomplish the folder structure definition by creating an API specific startup file. A template for this file, api_startup_file_template.txt is in the api folder in the main GMAT folder. Follow these steps to configure that file for use:

1. Copy the startup file template into the GMAT bin folder, renaming it api_startup_file.txt.
2. Open the new api_startup_file.txt file with a text editor.
3. Change the "<TopLevelGMATFolder>" entry near the top of the file to the absolute path to your top level GMAT folder.

   For example, if GMAT is installed (on Windows; please excuse the backslashes) in

   C:MyGmatR2019aBeta2

   then set the ABSOLUTE_PATH variable to

ABSOLUTE_PATH = C:/MyGmat/R2019aBeta2

Note I've replaced backslashes with forward slashes here. For the startup file, Windows users need not do this. Mac and Linux users should use their native forward slash formatting for this absolute path.

4. Edit the PLUGIN entries as needed, turning on desired (and available) plug-in components.
5. Save the file.

Once the api_startup_file is configured, you are ready to configure your Python or MATLAB environment.

External Access from Python

Note: External access from Python requires configuration of an API startup file, as described in the *File Location Access* text, above.

The API is loaded from a Python process running outside of the GMAT folders using the load_gmat.py module found in the GMAT api folder. The simplest way to proceed is to edit that file in place:

1. Open load_gmat.py in a text editor.
2. Change the "<TopLevelGMATFolder>" entry near the top of the file to the absolute path to your top level GMAT folder. Make sure that the path is enclosed in quotation marks.

   Windows users will also need to change backslash characters in this string either to double backslashes or to forward slashes so that the Python interpreter can handle the path correctly.
3. Save the file.
4. Copy the edited load_gmat.py file into the folder that is used for the API run.

   Note one advantage of editing the load_gmat file in the api folder is that this file can be copied into any folder that needs access to the API. In other words, once the file has the absolute path set, it can be copied to any folder that needs to act as the home folder for an API run.

Test the configuration to make certain that the API can be run. Note that, rather than directly importing load_gmat, you will want to preserve the imported symbols. This preservation is done using the syntax "from module import *", as shown in this Linux example:

```
$ cd APIFromHere/
$ python3
>>> from load_gmat import *
>>> sat = gmat.Construct("Spacecraft","Sat")
>>> gmat.ShowObjects()
Current GMAT Objects

   EarthMJ2000Eq
   EarthMJ2000Ec
   EarthFixed
   EarthICRF
   SolarSystemBarycenter
   Sat

>>>
```

External Access from MATLAB

Note: External access from MATLAB requires configuration of an API startup file, as described in the *File Location Access* text, above.

The API is loaded from a MATLAB console running outside of the GMAT folders using the load_gmat.m module found in the GMAT bin folder. That file configures MATLAB to use the API. The simplest way to use it is to add the GMAT bin folder to your MATLAB path, either in the running MATLAB environment or in the MATLAB configuration on your workstation. You can test the configuration to make certain that the API can be run following the steps below, which run the API from the folder APIFromHere outside of the GMAT installation folders:

```
$ cd APIFromHere/
$ matlab -nodesktop
MATLAB is selecting SOFTWARE OPENGL rendering.

                                                  < M A T L A B (R) >
                                        Copyright 1984-2019 The MathWorks, Inc.
                                      R2019a Update 6 (9.6.0.1214997) 64-bit␣
↪(glnxa64)
                                                   September 25, 2019


To get started, type doc.
For product information, visit www.mathworks.com.

>> addpath('/home/djc/gsfcgmat/GmatDevelopment/GMAT-R2019a-Linux-x64/bin/')
>> load_gmat
No script provided to load.

ans =

Instance of GMAT Moderator is initialized. No script ready to run.

>> sat = gmat.gmat.Construct('Spacecraft','MySat')

sat =

Object of type Spacecraft named MySat

>> gmat.gmat.ShowObjects()

ans =

Current GMAT Objects

   EarthMJ2000Eq
   EarthMJ2000Ec
   EarthFixed
   EarthICRF
   SolarSystemBarycenter
   MySat

The SolarSystem contains the following bodies:

   [Sun, Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto, Luna]
```

(continues on next page)

(continued from previous page)

```
>>
```

3.2.2 Loading the API

Packaging

The GMAT API is packaged in a set of libraries encapsulating specific pieces of functionality.

Table 3.1: The component packages built for GMAT's API

Package Name	Contents	Description
gmat	GMAT Utilities and Core Components	Utility classes used in the rest of GMAT, that do not depend on other GMAT components, built from gmatutil code. Classes used for most of the GMAT spacecraft modeling, along with the GMAT "engine" that drives the main application, built from gmatbase code.
station	Ground station models	Code used to model ground stations
navigation	Estimation Components	Classes used for the GMAT orbit determination subsystem.

Initialization

There are some differences in how to initialize the API between Python, Java, and MATLAB, however they all follow the same basic process. The API is first loaded into the interfacing language, and then the GMAT executive is initialized through the Moderator. The process of initializing the API is detailed in the sections below for each supported language. The initialization example for each language creates an instance of the Moderator object named `myMod` that has been initialized. The `result` variable is a flag which returns `True` if the initialization was successful, or `False` if unsuccessful.

Python

The API is loaded from the GMAT bin folder into Python with an import command

Python initialization example:

```
import gmatpy as gmat
```

At this point, the GMAT Python API is not initialized. A user can initialize it using the Setup() command, or by making any call to an API specific function.

The API can be loaded into from any folder from Python if by following the *External Access from Python* instructions, above.

Java

Java requires an extra step compared to Python. Not only does the GMAT API need to be imported, a shared library must also be loaded

```
import gmat.*;

public class main {

    public static void main(String[] args) {

        Moderator myMod;
        boolean result;

        // Load the GMAT library
        System.loadLibrary("gmat"); // On Windows

        myMod = Moderator.Instance();
        result = myMod.Initialize("gmat_startup_file.txt");
    }
}
```

At this point, the GMAT Java API is initialized.

MATLAB

While the MATLAB interface uses the Java API, there are a few extra steps required due to how Java packages are used inside MATLAB. The . To simplify this, a MATLAB script which performs all the initialization, load_gmat.m, is included with the GMAT installation in the <*install-dir*>/bin directory. The load_gmat script also optionally takes as an input the filename of a GMAT script to load, and a filename to a custom startup file.

```
[myMod, gmatStartupPath, result] = load_gmat("sample.script");
```

The extra steps in initializing the GMAT API through MATLAB is because every GMAT library and JAR file needs to be loaded explicitly, and it needs to be loaded by the Java instance inside MATLAB instead of by MATLAB itself. For more details, see the comments inside the load_gmat.m file.

3.3 Object Usage with the GMAT API

The goal of this section is to help you start using the GMAT API quickly, while introducing features of the interface in a natural progression of steps needed to solve a simple orbital state conversion problem.

3.4 Overview

The GMAT API is built using the Simplified Wrapper and Interface Generator, SWIG. SWIG connects code written in C and C++ with a variety of high level languages. The production GMAT API provides Python and Java connections for GMAT functionality. C++ programmers can also use the API specific code interfaces through direct calls into the GMAT libraries. The following sections describe the provided interfaces. Following that, three levels of interface usage are described:

1. Usage based primarily on GMAT scripting, with API generated component changes.
2. High level usage of GMAT components to meet specific user needs.
3. Class and object level access to GMAT components for expert users.

3.4.1 GMAT API Interfaces

The production GMAT API is built with Python and Java wrappers. The Java wrappers are used to also provide the MATLAB interface into the API.

The Python Interface

The Python wrappers are identified by the suffix "_py." Python imports are provided as .py files, which connect to associated shared library files. The Python API is packaged in the gmatpy folder contained in the GMAT bin folder. The API is loaded by importing that folder into the Python environment.

The Java and MATLAB Interface

The Java interface is provided in Java Archive (jar) files and associated libraries. These files include the GMAT base code API interfaces and the interfaces to the components used in the navigation subsystem, tabulated below.

Table 3.2: Java archives for the GMAT API

Java Wrapper	Python Wrapper	Contents
gmat.jar	gmat_py.py	GMAT Base and utility code
station.jar	station_py.py	Groundstation components
navigation.jar	navigation_py.py	Orbit determination components

MATLAB users load the GMAT API by calling the load_gmat.m MATLAB script in the GMAT bin folder.

3.4.2 Interface Complexity

The GMAT API is designed for three different styles of usage: Users working with configurations based on GMAT scripting, users that work with GMAT objects through API calls, and users that work at a low level with GMAT objects directly.

GMAT Script Drivers

One use for the GMAT API is to act as a front end for GMAT mission runs. In this context, the user starts a runtime environment for the controlling language (e.g. a Python session or MATLAB), loads and initializes GMAT using the API, and then loads a GMAT script into the running environment. At this point the user might manipulate setting on the GMAT objects used in the script to tailor the run. Once the configuration is ready, the API is used to run the script.

More details can be found in the *Script Usage* chapter.

High Level Access

A second use of the GMAT API is as a tool for using intermediate level GMAT objects to model portions of an analysis problem, and to feed the modeled results back to the driving system for further analysis. When used this way, the GMAT API provides proven and tested components used to meet the user's needs for building blocks for a problem running outside of GMAT. Examples of this type of usage include

- Converting state data from one coordinate system into another
- Accessing the GMAT force models for accelerations and Jacobians of a given state.
- Accessing the Navigation measurement models for retrieve calculated measurement values.

The walk-through provided in the *Tutorial: Accessing GMAT Propagation and Navigation Features* chapter covers the techniques needed for high level object access and usage.

Component Level Access

Some API users need access to the details of GMAT's components in order to control them at a fine grained level during use, to extend them with new computed data during use, or to monitor and report their state during use. These users may want to configure the objects by hand, and may want to manipulate the objects differently from the ways anticipated by the GMAT developers. The GMAT API allows for this level of access to GMAT's components. The *Doxygen* generated object level documentation provides a guide to this type of usage.

3.5 Functions Used in the GMAT API

The GMAT API provides several functions that simplify GMAT use from Java and Python. GMAT's user classes include member functions, called methods, that provide interfaces into GMAT objects that help simplify calls into the GMAT objects. These features of the API are described in this chapter. Script users may also want to refer to the *Script Usage* chapter for information about functions tailored to running GMAT scripts through the API.

3.5.1 General API Functions

API specific code in can be broken into three blocks: General API functions used to interact with the system, functions specific to driving GMAT using script files, and methods that are implemented on the GMAT classes to simplify object interactions using the API.

Table 3.3: General Purpose Functions Controlling the API

Function	Example	Return Value	Description
Help	gmat.Help("MySat")	string	Returns help for the input item
Setup	gmat.Setup("StartFile.txt")	void	Initializes the GMAT system with a custom startup file
Initialize	gmat.Initialize()	none	Sets up interconnections between objects and reports on missing pieces
ShowObjects	gmat.ShowObjects()	string	Lists the configured objects
ShowClasses	gmat.ShowClasses("Burn")	string	Lists the classes available of a given type
Construct	**gmat.Construct(** "Spacecraft","Sat")	object	Creates an instance of a class with a given name and adds it to the GMAT configuration
Copy	gmat.Copy(sat, "Sat2")		Creates a new object using the settings on an existing object
GetObject	gmat.GetObject("Sat")	object	Retrieves an object from the configuration

3.5.2 GMAT Object Methods

GMAT's user classes have been updated with member functions (or methods) that facilitate use for the objects by API users. These methods are shown in Table 3.4.

Table 3.4: Methods added to GMAT objects for API users

Function	Example	Return Value	Description
Help	Sat.Help()	string	Retrieves help for an object
SetField	Sat.SetField("X",0.0)	bool	Sets a field on an object
GetField	Sat.GetField("X")	string	Retrieves the setting for a field as a string
GetNumber	Sat.GetNumber("X")	double	Retrieves the setting for numerical field

CHAPTER

FOUR

SCRIPT USAGE

The GMAT API can be used as a front end for driving the GMAT application in a "headless" mode. You might want to do this to run GMAT remotely, to script product generation, or to perform a large scale run like a Monte-Carlo run or a scan through a set of parameters. This section introduces the API features that make these processes possible.

4.1 API Functions for Script Users

The GMAT API includes five functions, shown in Table 4.1 specifically designed for script based usage.

Table 4.1: Functions Used to Run Scripts in the API

Function	Example	Return Value	Description
LoadScript	LoadScript("script")	bool	Loads a script into the GMAT system
RunScript	RunScript()	bool	Runs a loaded script
SaveScript	SaveScript("script")	bool	Saves the configured objects to a file
GetRuntimeObject	GetRuntimeObject ("Sat")	GmatBase*	Retrieves an object from a GMAT run
GetRunSummary	GetRunSummary()	String	Retrieves a listing of the spacecraft data from a script run for each command in the script

4.1.1 Background: The GMAT Run Script Process

In GMAT, when a user runs a script three steps are taken:

1. The script is read and GMAT objects are created that match the objects described by the script.
    - GMAT Resources are stored in the GMAT Configuration.
    - GMAT Commands are connected together to create the Mission Control Sequence.
2. The objects from the script are copied ("cloned") into a memory location, the Sandbox, used for the run.
    - This creates a new set of objects used for the run.
    - After cloning, these run time objects are connected together as needed and initialized.
3. The Mission Control Sequence is executed sequentially, starting from the first command in the sequence. Command execution manipulates the run time objects to simulate the scripted mission.

API users drive this process using the functions in Table 4.1.

4.1.2 Driving a Script From the API

API users take the following steps to execute a script:

1. Start the API environment (e.g. Python or MATLAB) from the GMAT bin directory or start the application and change the current directory to the GMAT bin directory.
2. Load GMAT into the environment. The MATLAB implementation includes a scripted function for this process.
   - **Python**: import gmatpy
   - **MATLAB**: load_gmat()
3. Read a script into the loaded GMAT engine.
   - **Python**: gmatpy.LoadScript(*script_path_and_name*)
   - **MATLAB**: gmat.gmat.LoadScript(*script_path_and_name*)
4. Run the script. This step performs the run time object cloning and initialization and then executes the Mission Control Sequence.
   - **Python**: gmatpy.RunScript()
   - **MATLAB**: gmat.gmat.RunScript()

4.2 Examples

The sections above describe in general terms how to run GMAT scripts from the API. The following sample usage shows these features in Python and MATLAB.

4.2.1 Example: Running a Sample Mission

The first example shows how to run a sample mission from the API and retrieve data generated from the run. The example runs the sample mission Ex_GEOTransfer.script in the GMAT samples folder, then accesses the targeted maneuvers from the script and computes the total delta-V needed for the run.

Python

Use of the API in Python is performed through direct calls to the functions described above.

Listing 4.1: Sample Run: Calculating the Delta-V for the GEO Transfer, Run in Python

```
$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import gmatpy as gmat
>>> gmat.LoadScript("../samples/Ex_GEOTransfer.script")
True
>>> gmat.RunScript()
True
>>> TOI = gmat.GetRuntimeObject("TOI")
>>> MCC = gmat.GetRuntimeObject("MCC")
>>> MOI = gmat.GetRuntimeObject("MOI")
```

(continues on next page)

(continued from previous page)

```
>>> toidv = float(TOI.GetField("Element1"))
>>> mccdv = (float(MCC.GetField("Element1"))**2+float(MCC.GetField("Element2
↪"))**2)**0.5
>>> moidv = float(MOI.GetField("Element1"))
>>> deltaV = abs(toidv)+mccdv+abs(moidv)
>>> print("Total Delta-V Cost: ", deltaV, " km/s")
Total Delta-V Cost:  4.394839062410714  km/s
>>> exit()
```

MATLAB

Loading the GMAT API in MATLAB is a moderately complicated procedure, so the API developers have wrapped the load process in a MATLAB function, load_gmat.m. The function takes two optional arguments: the name of a script, and the startup file used to initialize GMAT. The example shown below leaves both inputs blank so that it closely matches the Python example, above.

Listing 4.2: Sample Run: Calculating the Delta-V for the GEO Transfer, Run in MATLAB

```
>> load_gmat()
Initialize Moderator Status: 1
No script provided to load.

ans =

Instance of GMAT Moderator is initialized. No script ready to run.

>> gmat.gmat.LoadScript("../samples/Ex_GEOTransfer.script")

ans =

  logical

   1

>> gmat.gmat.RunScript()

ans =

  logical

   1

>> TOI = gmat.gmat.GetRuntimeObject("TOI");
>> MCC = gmat.gmat.GetRuntimeObject("MCC");
>> MOI = gmat.gmat.GetRuntimeObject("MOI");
>> toidv = str2num(TOI.GetField("Element1"));
>> mccdv = sqrt(str2num(MCC.GetField("Element1"))^2+str2num(MCC.GetField("Element2"))^
↪2);
>> moidv = str2num(MOI.GetField("Element1"));
>> DeltaV = abs(toidv)+mccdv+abs(moidv)

DeltaV =

   4.394839062410714
```

(continues on next page)

(continued from previous page)

```
>> exit
```

CHAPTER

FIVE

TUTORIAL: ACCESSING GMAT PROPAGATION AND NAVIGATION FEATURES

The use case demonstration for the first GMAT API Beta build focuses on using the API for models that exercise core GMAT functions. For the purposes of this release of the API, the target use case suite shows how to setup and use force models, propagators, and measurement models. The implemented functionality in the Beta release includes the ability to drive GMAT scripts as well. That capability is described separately in *Script Usage*.

The goal of the use cases described here is to build a GMAT measurement model from core components, using the API functions. GMAT's measurement models require a propagation component in order to solve the light transit time portion of the modeling. Propagation requires configuration of both a numerical integrator and of a force model, along with a spacecraft that supplies model parameters. This guide will walk Python and MATLAB users through the process of configuring these components in order to assemble a range measurement.

The problem demonstrated in the example code here provides modeling for an Earth orbiting spacecraft, "EarthOrbiter." The spacecraft is an 80 kg vehicle modeled in a polar orbit with a 6600 km semimajor axis. That ensures that we need to configure a fair number of system parameters in order to build the simulation.

5.1 Verifying Setup

The GMAT API is included in releases of GMAT beginning with the R2020a release of the system. Once GMAT is installed, the API can be accessed from inside of the folder containing the GMAT application.

- For Python, change directories to the GMAT bin folder and access the API help system:

```
>>> import gmatpy as gmat
>>> gmat.Help()

---------------------------------------
GMAT Application Programmer's Interface
---------------------------------------
...
```

- For MATLAB users
  - Start MATLAB
  - Change directories to the installed GMAT bin directory
  - Load the API and access the help system:

```
>> load_gmat()
No script provided to load.
```

(continues on next page)

(continued from previous page)

```
ans =

Instance of GMAT Moderator is initialized. No script ready to run.

>> gmat.gmat.Help()

ans =


----------------------------------------
GMAT Application Programmer's Interface
----------------------------------------
...
```

5.2 Getting Started with the API

Note: This section introduces several API functions and tools for users new to GMAT's API by experimenting interactively with the system in Python. This section introduces interaction with the GMAT API, but is not necessary for the force model, propagation, and measurement configuration described below.

The modeling for this use case uses an Earth centered coordinate system with axes oriented in the mean of J2000 Equatorial frame. The coordinate system can be built using the Construct function of the API. The Python code for this object creation is:

```
import gmatpy as gmat
eartheq = gmat.Construct("CoordinateSystem", "EarthMJ2000Eq", "Earth", "MJ2000Eq")
```

Users can view the list of objects that have been built during a run using the ShowObjects() function:

```
>>> gmat.ShowObjects()
Current GMAT Objects

   EarthMJ2000Eq

>>>
```

The code above created a coordinate system object, an axis system object, and connected those objects together for later use. The object cannot be fully exercised at this point because it is missing some key connections to the modeled space environment, consisting of a solar system model, member planets and the Sun and Moon, along with other internal objects used to tie a user's components together.

For this example, the solar system is not yet connected to the new coordinate system, and the bodies needed for use - the Earth, for example - are also not yet connected. The coordinate system object's state can be checked using its "IsInitialized()" method:

```
>>> eartheq.IsInitialized()
False
```

Objects that are ready for use return True from this call. The API prepares the objects for use with the Initialize() function:

```
>>> gmat.Initialize()
>>> eartheq.IsInitialized()
True
```

The Initialize() function prepares all of the objects that the user has created for use, and reports any objects that could not be prepared because of missing settings.

Objects can be removed individually from GMAT using the Clear(*ObjectName*) function, or all once using the Clear() function without specifying an object:

```
>>> gmat.ShowObjects()
Current GMAT Objects

   EarthMJ2000Eq

>>> gmat.Clear("EarthMJ2000Eq")
'The object EarthMJ2000Eq has been removed from GMAT.'
>>> gmat.ShowObjects()
Current GMAT Objects

>>> eartheq = gmat.Construct("CoordinateSystem", "EMJ2k", "Earth", "MJ2000Eq")
>>> gmat.ShowObjects()
Current GMAT Objects

   EMJ2k

>>> gmat.Clear()
'All configured objects have been removed from GMAT.'
>>> gmat.ShowObjects()
Current GMAT Objects


>>>
```

The full set of GMAT API commands are described in *Functions Used in the GMAT API*.

The functions used above are presented working interactively in the GMAT API. In the remainder of this section, the configuration is built in a Python script included with the GMAT release in the API folder. Matching MATLAB .m scripts are included in the API folder for users more comfortable working in that environment.

5.3 Spacecraft Configuration

Note: The Spacecraft Configuration and Force Model Setup examples are located in the Ex_R2020a_BasicForceModel.py file. An example of the full configuration, showing one configuration for the exercises, is in the Ex_R2020a_CompleteForceModel file.

To use one of these files, copy the file you plan to use into GMAT's bin folder.

Spacecraft configuration requires that we define the initial spacecraft state data for the orbiter, and then configure the spacecraft properties needed for the force modeling.

5.3.1 Construction and The Initial State

Using this coordinate system, the spacecraft state can be configured. For this example, the spacecraft is in a circular polar orbit at the moon on July 20, 2020. The orbital state is set using the scripting

```
# Load GMAT into memory
import gmatpy as gmat

# Spacecraft configuration preliminaries
earthorb = gmat.Construct("Spacecraft", "EarthOrbiter")
earthorb.SetField("DateFormat", "UTCGregorian")
earthorb.SetField("Epoch", "20 Jul 2020 12:00:00.000")

earthorb.SetField("CoordinateSystem", "EarthMJ2000Eq")
earthorb.SetField("DisplayStateType", "Keplerian")

# Orbital state
earthorb.SetField("SMA", 6600)
earthorb.SetField("ECC", 0.05)
earthorb.SetField("INC", 78)
earthorb.SetField("RAAN", 45)
earthorb.SetField("AOP", 90)
earthorb.SetField("TA", 180)
```

5.3.2 Additional Spacecraft Parameters

The force model used for this example may include full field Earth gravity, point mass effects from the Sun and Moon ("Luna" for GMAT), Jacchia-Roberts drag, and solar radiation pressure. The latter forces require settings for the reflectivity and drag coefficients of the spacecraft, its surface areas for those forces, and the spacecraft mass. These settings are made using the SetField method, and resemble the corresponding GMAT scripting:

```
# Spacecraft ballistic properties for the SRP and Drag models
earthorb.SetField("SRPArea", 2.5)
earthorb.SetField("Cr", 1.75)
earthorb.SetField("DragArea", 1.8)
earthorb.SetField("Cd", 2.1)
earthorb.SetField("DryMass", 80)
```

For reference, the GMAT scripting for this configuration is

```
Create Spacecraft EarthOrbiter;
GMAT EarthOrbiter.DateFormat = UTCGregorian;
GMAT EarthOrbiter.Epoch = '20 Jul 2020 12:00:00.000';
GMAT EarthOrbiter.CoordinateSystem = MoonEc;
GMAT EarthOrbiter.DisplayStateType = Keplerian;
GMAT EarthOrbiter.SMA = 4000;
GMAT EarthOrbiter.ECC = 0.05;
GMAT EarthOrbiter.INC = 78;
GMAT EarthOrbiter.RAAN = 45;
GMAT EarthOrbiter.AOP = 90;
GMAT EarthOrbiter.TA = 180;
GMAT EarthOrbiter.DryMass = 80;
GMAT EarthOrbiter.Cd = 2.1;
GMAT EarthOrbiter.Cr = 1.75;
GMAT EarthOrbiter.DragArea = 1.8;
GMAT EarthOrbiter.SRPArea = 2.5;
```

Note: Differences between Python and MATLAB code

The code shown above and throughout this section is Python code. In the MATLAB version of this example, there are several differences worth noting because of the platform differences:

1. GMAT is loaded using the MATLAB load_gmat.m script. This script loads the GMAT libraries into MATLAB, initializes the GMAT system by reading a startup file and loading plugins and configuration data identified in that file, and optionally loads a GMAT script into memory.

2. Access to the GMAT functions is made using a call into a nested "gmat" class built for the underlying Java code. In general, where Python users type

   ```
   sat = gmat.Construct("Spacecraft","Sat")
   ```

   MATLAB users enter

   ```
   sat = gmat.gmat.Construct('Spacecraft','Sat');
   ```

3. Type identification requires a call that sets the object type. In the Python implementation of the API, changes from the base GmatBase type to the derived type is automatic. In MATLAB/Java, the user needs to perform the cast. Where Python users type

   ```
   sat = gmat.Construct("Spacecraft","Sat")
   ```

   to work with a Spacecraft object, MATLAB users that need to interact with the object **as a Spacecraft** need to enter

   ```
   sat = gmat.gmat.Construct('Spacecraft','Sat');
   sat = gmat.Spacecraft.SetClass(sat);
   ```

   A MATLAB class, GMATAPI, is provided in the bin folder which contains static functions that automatically handle the change from the base GmatBase type to the derived type just like in the Python API. MATLAB users can now type

   ```
   sat = GMATAPI.Construct('Spacecraft','Sat');
   ```

   This type setting becomes important when objects are passed to other objects using methods that require specific object type, as is the case when setting forces on a dynamics model or spacecraft on a propagation state manager (examples below).

5.4 Force Model Setup

GMAT hides the complexity of force modeling in the internal ODEModel class, which is aliased to the label "ForceModel" in GMAT scripting. The GMAT scripting for the force model used here is

```
Create ForceModel FM;
GMAT FM.CentralBody = Earth;
GMAT FM.PrimaryBodies = {Earth};
GMAT FM.GravityField.Earth.Degree = 8;
GMAT FM.GravityField.Earth.Order = 8;
GMAT FM.GravityField.Earth.PotentialFile = 'JGM3.cof';
```

5.4.1 Basic Force Model Configuration

Using the API is similar for force configuration, but not identical. The GMAT scripting hides the creation of individual forces and their collection into the ODEModel force container. Settings on the forces in an ODEModel are made by passing those settings from the force container to the corresponding force. API users access the forces directly, setting their parameters and force by force and passing the configured forces into the ODEModel container. The model scripted above is configured using the scripting

```
# Force model settings
fm = gmat.Construct("ForceModel", "FM")
fm.SetField("CentralBody", "Earth")

# An 8x8 JGM-3 Gravity Model
earthgrav = gmat.Construct("GravityField")
earthgrav.SetField("BodyName","Earth")
earthgrav.SetField("PotentialFile","../data/gravity/earth/JGM3.cof")
earthgrav.SetField("Degree",8)
earthgrav.SetField("Order",8)

# Add force to the dynamics model
fm.AddForce(earthgrav)
```

Note the difference in the calls to Construct in this example. The dynamics model is created using the line

```
fm = gmat.Construct("ForceModel", "FM")
```

and the gravity model component, with the code

```
earthgrav = gmat.Construct("GravityField")
```

The dynamics model has a name, "FM." The gravity field does not have a name. Objects constructed with names are managed by the GMAT code running inside of the API library. Objects that do not have names are not managed by the library. The object ownership for those objects is the responsibility of the code that creates the object. For the dynamics model under construction here, the user has responsibility for the gravity field in this call:

```
# Unnamed: The user is responsible for this object
earthgrav = gmat.Construct("GravityField")
```

and then passes that responsibility to the dynamics model with this call:

```
# Add force to the dynamics model, passing ownership to the dynamics
fm.AddForce(earthgrav)
```

5.4.2 Connecting the Spacecraft

Before the force model can be used, it needs to be connected to the spacecraft that provides state data and force model parameters. GMAT does this using a component called a Propagation State Manager (PSM). The PSM component is not exposed to script users. It is built inside of the scripted Propagator object that connects together integrators and force models.

Users that want to work directly with a force model can do so by creating a Propagation State Manager object and working directly with it. The force model built above can be tested using this approach:

```
psm = gmat.PropagationStateManager()
psm.SetObject(earthorb)
psm.BuildState()
```

The last line here, "psm.BuildState()", creates an internal state object that connects spacecraft properties to a vector of data used by the force model. The propagation state manager is connected to the force model, and its state set as the force model's state, using the scripting

```
fm.SetPropStateManager(psm)
fm.SetState(psm.GetState())
```

5.4.3 Testing the Model

The steps above produce a GMAT force model configuration that can be used from the user's application framework. All that remains is initialization of the objects, post initialization preparation, and then calls that exercise the model. Initialization connects the force model to GMAT's underlying resources, including the solar system objects and core elements of the system infrastructure:

```
# Assemble all of the objects together
gmat.Initialize()
```

GMAT's propagation subsystem, which includes the force model components, requires two additional steps before it can be used. First the state vector needs to be set up for the force and propagation modeling. This step determines and sets the size of the state and derivative vectors, and sets up mappings between the spacecraft that are modeled and that state vector. The second step passes the parameters needed for modeling into the force model and propagator objects that are used.

For direct access to the force modeling, the user needs to execute these steps directly:

```
# Finish force model setup:
##  Map spacecraft state into the model
fm.BuildModelFromMap()
##  Load physical parameters needed for the forces
fm.UpdateInitialData()
```

Users can display the Cartesian form of the state vector used in the modeling by accessing the state vector from the spacecraft:

```
# Now access state and get derivative data
pstate = earthorb.GetState().GetState()
print("State Vector: ", pstate)
```

Finally, the force model can be exercised either in its raw form as used by the integrators by calling the GetDerivatives() method:

```
fm.GetDerivatives(pstate)
dv = fm.GetDerivativeArray()
print("Derivative:   ", dv)
```

or by calling it for a specific spacecraft object through the GetDerivativesForSpacecraft() method:

```
vec = fm.GetDerivativesForSpacecraft(earthorb)
print("SCDerivative: ", vec)
```

When these pieces are assembled together, a run of the Ex_R2020a_BasicForceModel script shows the input state and derivative outputs to the user:

Note: numbers have been truncated for display purposes

```
$ python3 Ex_R2020a_BasicForceModel.py
State Vector:  [1018.819261, -1018.819261, -6778.562873, 5.226958, 5.226958, -1.
↪374825e-15]

Derivative:    [5.226958, 5.226958, -1.374825e-15, -0.00121383, 0.00121392, 0.
↪00809840]

SCDerivative:  5.226958 5.226958 -1.374825e-15 -0.00121383 0.00121392 0.00809840
```

5.4.4 Exercises

1. Add point mass forces for the Sun and Moon to the force model. The GMAT class for point mass forces is named "PointMassForce".
2. Use the propagation state manager to turn on the A-Matrix computation for the force model by passing the "AMatrix" setting to the propagation state manager using its SetProperty method.
3. Add a Jacchia-Roberts drag model and a solar radiation pressure model to the force model.

5.5 Propagator Setup

Note: The Propagator Setup example shown here is located in Ex_R2020a_PropagationStep.m file. It uses a basic force model by importing from the Ex_R2020a_BasicFM file, a stripped down version of the force model used in the previous section. An example of the full configuration, showing one solution for the exercises, is in the PropagateLoop file.

To use one of the propagation files, copy the file you plan to use into GMAT's bin folder. Also copy the Ex_R2020a_BasicFM file.

In GMAT scripting the lines of script for a propagator,

```
Create Propagator PDProp
GMAT PDProp.FM = FM;
GMAT PDProp.Type = PrinceDormand78;
GMAT PDProp.InitialStepSize = 60;
GMAT PDProp.Accuracy = 1.0e-12;
GMAT PDProp.MinStep = 0.0;
```

create an object from the GMAT class PropSetup. This object is a container for an object that performs propagation either numerically through an Integrator object or analytically through an object implementing an analytic algorithm. The latter objects are used, in GMAT, for ephemeris propagators. The former are used for Runge-Kutta integrators, predictor-correctors, and other numerical integration algorithms that require associated dynamics models. When the propagator requires a dynamics model, that model is also managed by a PropSetup object. The key feature to know for propagator configuration in the GMAT API is that a "Propagator" is actually a PropSetup object that contains the propagation component and, for numerical integrators, a dynamics model.

Working interactively, an API user can see this relationship in Python:

```
>>> import gmatpy as gmat
>>> pdprop = gmat.Construct("Propagator","PDProp")
>>> pdprop
<gmatpy.gmat_py.PropSetup; proxy of <Swig Object of type 'PropSetup *' at␣
↪0x7f26b76a57e0> >
```

5.5.1 Propagator Component Setup

When a Propagator is scripted, a PropSetup is created that the user then configures for use. Using the provided MATLAB example, the code that loads the force model and builds the PropSetup is

```
% Load GMAT into memory
[myMod, gmatStartupPath, result] = load_gmat();

Ex_R2020a_BasicFM;

% Build the propagation container class
pdprop = GMATAPI.Construct("Propagator","PDProp");
```

The PropSetup constructed here is a container for the objects used in propagation. The next step configuring this container is creation and assignment of an integrator, performed using the steps

```
% Create and assign a numerical integrator for use in the propagation
gator = GMATAPI.Construct("PrinceDormand78");
pdprop.SetReference(gator);
```

The dynamics model also needs to be set on the PropSetup:

```
% Assign the force model imported from Ex_R2020a_BasicFM
pdprop.SetReference(fm);
```

Once the local references are set, the integrator settings can be made similarly to the dynamics model setting in the previous section:

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

5.5.2 Spacecraft and Final Initialization

In the preceding section, the propagation state manager was built as a separate component and configured to connect the spacecraft to the dynamics model. When working with a PropSetup component, the propagation state manager is integrated into the component. As an alternative to the manual steps to configure the propagation state manager, the PropSetup provides a function, PrepareInternals(), that handles this configuration for each propagated object added through the AddPropObject() function, and completes the initialization of the component and its integrator:

```
% Setup the spacecraft that is propagated
pdprop.AddPropObject(earthorb);
pdprop.PrepareInternals();
```

GMAT's PropSetup component works by creating copies of the propagator and dynamics models. Those copies need to be set for the application environment so that the user can use them after configuration. The PropSetup provides a simple mechanism for accessing its copies. The code that refreshes the local variables for them to be used, is

```
% Refresh the 'gator reference
gator = pdprop.GetPropagator();
```

5.5.3 Running the Propagator

The propagator can now be used. A 60-second propagation is performed, showing the state data before and after the step, using the code

```
% Take a 60 second step, showing the state before and after
gator.GetState()
gator.Step(60);
gator.GetState()
```

These calls produce this output from Ex_R2020a_PropagationStep.m:

```
>> Ex_R2020a_PropagationStep
Initialize Moderator Status: 1
No script provided to load.

ans =

   1.0e+03 *

   1.018819261603825
  -1.018819261603827
  -6.778562873085272
   0.005226958779502
   0.005226958779502
  -0.000000000000000


ans =

   1.0e+03 *

   1.330028382856595
  -0.703241487939055
  -6.763990149325915
   0.005142965479731
   0.005288543267909
   0.000485610549370

>>
```

5.5.4 Exercises

1. Modify the Ex_R2020a_PropagationStep example to use a force model that includes the point mass Sun and Moon forces and solar radiation pressure.
2. Wrap the propagator in a loop so that propagation extends for a full day, displaying the epoch and position at each propagation step.

5.6 The GMAT API and Plug-in Modules

GMAT plug-in modules package new functionality into shared libraries that GMAT loads when it starts up. The API's copy of GMAT loads these modules when they are identified in the GMAT startup file. Standard GMAT functions work on components from plugins, but the API calls have several restrictions.

5.6.1 Wrapped Plugins

The Station and Estimation plugin libraries in GMAT include SWIG wrapper code for the contained classes. This reduces the restrictions on those components.

As an example of the restrictions on wrapped plugin code, consider the Station plugin, which implements GMAT's GroundStation class. Users of GMAT's GroundStation class can access the full feature set for the class. The user is required in Python to cast constructed components to the derived class type by hand. The Python auto-cast feature in the GMAT core code is not accessible from the plugin component as seen below:

```
>>> import gmatpy as gmat
>>> station = gmat.Construct("GroundStation","Station")
>>> station
<gmatpy.gmat_py.GmatBase; proxy of <Swig Object of type 'GmatBase *' at␣
↪0x7fccdc983f60> >
>>> station=gmat.GroundStation.SetClass(station)
>>> station
<gmatpy.station_py.GroundStation; proxy of <Swig Object of type 'GroundStation *' at␣
↪0x7fccdc983fc0> >
>>>
```

Note that in this code, the station object returned from the call to the Construct() function is set as a GmatBase object. In order to treat it as a GroundStation object, the user needed to call the GroundStation.SetClass() method on the object in order for Python to identify the object's subclass correctly.

MATLAB API users are not required to explicity cast the class, provided they use the GMATAPI MATLAB class, as shown below:

```
>> load_gmat();
No script provided to load.
>> station = GMATAPI.Construct("GroundStation","Station")

station =

Object of type GroundStation named Station

>> station.getClass()

ans =

class gmat.GroundStation

>>
```

5.6.2 Unwrapped Plugins

Plugin code that is not wrapped in SWIG can be accessed using the API, but only in a more restricted manner. As an example, at this writing the VF13ad optimizer is available as a GMAT component for users inside of Goddard Space Flight Center. The associated plugin builds a component with class name "VF13ad" that provides the optimization functionality. The VF13ad optimizer is derived from an Optimizer base class in the GMAT core code. API users can access that component as a GmatBase object, or as an Optimizer object, but not as a VF13ad object, as can be seen here:

```
>>> import gmatpy as gmat
>>> vf13 = gmat.Construct("VF13ad","VF13")
>>> vf13
```

(continues on next page)

(continued from previous page)

```
<gmatpy.gmat_py.GmatBase; proxy of <Swig Object of type 'GmatBase *' at␣
↪0x7f615f50c2a0> >
>>> vf13 = gmat.VF13ad.SetClass(vf13)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'gmatpy' has no attribute 'VF13ad'
>>> vf13 = gmat.Optimizer.SetClass(vf13)
>>> vf13
<gmatpy.gmat_py.Optimizer; proxy of <Swig Object of type 'Optimizer *' at␣
↪0x7f614d9cff90> >
>>> exit()
```

The underlying object remains a VF13ad component:

```
>>> vf13.Help()

VF13ad  VF13

   Field                                    Type    Value
   ---------------------------------------------------------

   ShowProgress                          Boolean   true
   ReportStyle                              List   Normal
   ReportFile                           Filename   <not set>
   MaximumIterations                     Integer   200
   Tolerance                                Real   1e-05
   UseCentralDifferences                 Boolean   false
   FeasibilityTolerance                     Real   0.001
```

but, from the perspective of an API user, is manipulated as an Optimizer or GmatBase object.

5.7 Measurement Modeling

Note: The Measurement Modeling example shown here is located in Ex_R2020a_RangeMeasurement.m file (or the Ex_R2020a_RangeMeasurement.py file for Python users).

GMAT's Measurement Models are driven through the TrackingFileSet class. A TrackingFileSet defines a measurement as a tracking configuration consisting of a signal path and measurement type. The signal path is defined by the nodes that a measurement signal traverses to create the measurement. For example, the path may be ground station -> spacecraft -> ground station for a range measurement.

The user configures the hardware for each node, assigning antennae, transmitters, receivers, and transponders as needed to the stations and spacecraft used in the measurement. Error models are configured and assigned to each measurement, media corrections are toggled, and ancillary components configured - like propagators is light time correction is applied - to complete the configuration the user needs. The system is complex because the processes involved have many options. This guide, and the sample script, step through the process element by element to build the model.

5.7.1 Spacecraft, Ground Stations, and Propagators

```
% Construct the PropSetup to control propagation (for light time)
prop = GMATAPI.Construct("PropSetup", "prop");
prop.GetODEModel().SetField("ErrorControl", "None");
prop.GetPropagator().SetNumber("MinStep", 0);

% Create objects for generating measurement
simsat = GMATAPI.Construct("Spacecraft", "SimSat");
gds = GMATAPI.Construct("GroundStation", "GDS");

% Configure Spacecraft initial conditions
simsat.SetField("DateFormat", "A1ModJulian");
simsat.SetField("Epoch", "21550");

% Configure GroundStation
gds.SetField("StateType", "Spherical");
gds.SetField("HorizonReference", "Ellipsoid");
gds.SetField("Location1", 0);
gds.SetField("Location2", 90.0);
gds.SetField("Location3", 0);
```

5.7.2 Hardware Components

```
% Create communication hardware
% Hardware for ground station
ant1 = GMATAPI.Construct("Antenna", "Antenna1");
tmit = GMATAPI.Construct("Transmitter", "Transmitter1");
tmit.SetField("Frequency", 2067.5);
rec = GMATAPI.Construct("Receiver", "Receiver1");

% Hardware for spacecraft
ant2 = GMATAPI.Construct("Antenna", "Antenna2");
tpond = GMATAPI.Construct("Transponder", "Transponder1");
tpond.SetField("TurnAroundRatio","240/221");

% Set fields
% Use Antenna1 for Transmitter1 and Receiver1
tmit.SetField("PrimaryAntenna","Antenna1");
rec.SetField("PrimaryAntenna","Antenna1");

% Use Antenna2 for Transponder1
tpond.SetField("PrimaryAntenna","Antenna2");

% Add Antenna2 and Transponder1 to spacecraft
simsat.SetField("AddHardware","{Antenna2, Transponder1}");

% Add Antenna1, Transmitter1, and Receiver1 to station
gds.SetField("AddHardware","{Antenna1, Transmitter1, Receiver1}");
```

5.7.3 Error Models

```
% Define range measurements and error model
tem = GMATAPI.Construct("ErrorModel", "TheErrorModel");
% Specify these measurements are range measurements in km
tem.SetField("Type","Range");
tem.SetField("NoiseSigma", 0.050); % Standard deviation of noise
tem.SetField("Bias",0); % Bias in measurement

% Define doppler range rate measurements and error model
tem2 = GMATAPI.Construct("ErrorModel", "TheErrorModel2");
% Specify these measurements are doppler range rate measurements
tem2.SetField("Type","RangeRate");
tem2.SetField("NoiseSigma", 5e-5); % Standard deviation of noise
tem2.SetField("Bias",0); % Bias in measurement

% Add ErrorModels to the ground station
gds.SetField("ErrorModels","{TheErrorModel, TheErrorModel2}");
```

5.7.4 The Measurement

```
% Create a TrackingFileSet to manage the observations
tfs = GMATAPI.Construct("TrackingFileSet", "SimData");
tfs.SetField("FileName","TrkFile_API_GN.gmd"); % Still needed even though it's' not␣
↪written to
tfs.SetField("UseLightTime", false);
tfs.SetField("UseRelativityCorrection", false);
tfs.SetField("UseETminusTAI", false);

% Define signal paths and measurement type(s)
% 2-way measurements are used here along the path GDS -> SimSat -> GDS
% Add range measurements to TrackingFileSet
tfs.SetField("AddTrackingConfig", "{{GDS,SimSat,GDS}, Range}");
% Add doppler range rate measurements to TrackingFileSet
tfs.SetField("AddTrackingConfig", "{{GDS,SimSat,GDS}, RangeRate}");
tfs.SetPropagator(prop); % Tell TrackingFileSet the propagator to use
```

5.7.5 Exercising the Model

```
% Initialize the GMAT objects
gmat.gmat.Initialize()

% Calculate the measurement
tdas = tfs.GetAdapters();
numMeas = tdas.size();

tda = tdas.get(0);
md0 = tda.CalculateMeasurement();
disp("GMAT Range Measurement Value:")
disp(md0.getValue().get(0))


% Make sure this is correct
satState = simsat.GetState();
gsPos = gds.GetMJ2000Position(satState.GetEpochGT()).GetDataVector();
```

(continues on next page)

(continued from previous page)

```
satPos = satState.GetState();
r = gsPos - satPos(1:3);
rNorm = norm(r);
disp("Numerical Range Measurement Value (no lighttime):")
disp(2*rNorm)

disp("")

xid = simsat.GetParameterID("CartesianX");
tda.CalculateMeasurementDerivatives(simsat,xid);
for ii = 0:5
    deriv(ii+1) = tda.ApiGetDerivativeValue(0,ii);
end
disp("GMAT Range Measurement Derivatives:")
disp(deriv)

disp("")

tda = tdas.get(1);
md1 = tda.CalculateMeasurement();
disp("GMAT RangeRate Measurement Value:")
disp(md1.getValue().get(0))
```

5.7.6 Exercises

1. Modify the one day propagation script to report the range measurement at each step where a valid measurement can be computed.
2. Add a second ground station the Ex_R2020a_RangeMeasurement example and report its measurement data.

CHAPTER

SIX

EXAMPLE: MONTE-GMAT INTEROPERABILITY FOR OSIRIS_REX

At this writing, the Origins, Spectral Interpretation, Resource Identification, Security-Regolith Explorer (OSIRIS-REx) spacecraft is in orbit at the near-Earth asteroid Bennu (formerly 1999 RQ36). The mission plans to bring at least a 2.1-ounce sample of the asteroid back to Earth for study. The mission is using the Mission Analysis, Operations and Navigation Toolkit Environment, MONTE, for navigation and trajectory planning.

In this example, data will be shared between MONTE and GMAT using interfaces built with MONTE's native Python framework and GMAT's API, accessed through Python.

6.1 Ephemeris Sharing

Both GMAT and MONTE have ephemeris reading and writing capabilities. GMAT supports four types of spacecraft ephemerides: Goddard specific "Code-500", STK time-position-velocity (.e), CCSDS OEM, and SPICE SPK formats. MONTE supports SPICE based SPK ephemerides, so that format is used for data interchange between the systems.

6.1.1 General Ephemeris Sharing

Ephemeris sharing between GMAT and MONTE is straightforward: use the system providing the ephemeris to generate the file, and then import it into the other system. The steps for this procedure are described below, beginning with the GMAT to MONTE procedure.

Generating an Ephemeris in GMAT

Tutorial: Accessing GMAT Propagation and Navigation Features describes the steps needed to propagate a spacecraft using the GMAT API. The sample code shown here starts from a similar propagation configuration, with a spacecraft in the following initial state:

Reading the GMAT Ephemeris in MONTE

Generating an Ephemeris in MONTE

Reading the MONTE Ephemeris in GMAT

A Round Trip

6.1.2 Ephemeris Sharing for OSIRIS-REx

6.2 Maneuver Planning

6.3 Navigation Data Sharing

CHAPTER

SEVEN

API BEST PRACTICES

7.1 General Practices

API users work more closely with the core GMAT code than do users that run GMAT through the console for GUI applications. This feature of the API system adds responsibility for understanding how the system manages objects to the list of items an API user must consider. The following items capture some of the lessons we have learned from using the API.

7.1.1 Understand Object Ownership

The API provides a function, Construct(), that builds GMAT objects and retains object ownership in the GMAT module. Objects created using Construct() remain GMAT's responsibility for management. Sometimes, API users may need to create objects directly by calling the object's constructor, like this:

Listing 7.1: Creating a propagation state manager in Python

```
psm = gmat.PropagationStateManager()
```

or, in MATLAB:

Listing 7.2: Creating a propagation state manager in MATLAB

```
psm = gmat.gmat.PropagationStateManager()
```

The objects created this way are managed on the client side of the interface: in either Python or the MATLAB Java systems. The garbage collectors on the client side will delete the underlying objects when it determines that the object is no longer needed. This can cause memory management issues for objects that are passed to other GMAT objects. The API code provides a mechanism to assign ownership to the component that needs it, using the setSwigOwnership() method in Java code:

Listing 7.3: Managing ownership for a propagation state manager in MATLAB

```
% Create the state manager
psm = gmat.PropagationStateManager();

% Hand the manager to a force model, and assign ownership to the GMAT object
fm.SetPropStateManager(psm);
psm.setSwigOwnership(false());
```

or the thisown setting in Python:

Listing 7.4: Managing ownership for a propagation state manager in Python

```
% Create the state manager
psm = gmat.PropagationStateManager();

% Hand the manager to a force model, and assign ownership to the GMAT object
fm.SetPropStateManager(psm);
psm.thisown = False
```

For either of these mechanisms, a false setting indicates that the client does not own the object.

7.2 Java and MATLAB Best Practices

- Adding the bin folder to your MATLAB path allows you to run the GMAT API from any other working directory
- Use the GMATAPI MATLAB class contained in the bin folder when using the API helper functions Construct(), Copy(), GetObject(), or GetRuntimeObject(). These functions in the GMATAPI MATLAB class will automatically perform class casting, so the object returned is the more specific type instead of just being of type GmatBase. The GMATAPI MATLAB class also contains a SetClass() function which will also automatically perform the class casting on any GmatBase object provided.

7.3 Python Best Practices

- The import function loads GMAT by loading all of the libraries in the gmatpy folder (gmat, station, etc). These libraries can be imported separately if you do not need all of the API functions in your application.
- The import can rename the interface calls for user convenience. In this document we often load the engine using

  import gmatpy as gmat

- The GMAT startup file is loaded the first time a GMAT API function is called. Users that want to use a startup file that is different from the default file, gmat_startup_file.txt, can load their file using the Setup(*path_and_startup_file_name*) function call.

  This approach is used for running the API from folders outside of the GMAT bin folder, as described in *Running the API Outside of the GMAT Folders*.

APPENDIX

A

REVIEW AND PROTOTYPE INFORMATION

A.1 Open issues

- Should the util library be wrapped separately from the base library?

  Resolution: The util code is included in the core GMAT API packaging.

- When will base be made into smaller components?
  - May be useful for on board project
  - Helps ensure modularity

  Resolution: There is no repackaging plan at this

- The CMake files need some work, so that the wrappers are written into the build folder (application/bin by default).

  Resolution: The current best practice is to build a GMAT release to get the API package set.

- Is there a tie-in between GMAT/CSALT on board and the GMAT API?

  Resolution: Not at this time.

- Are there export issues between the languages? (e.g. Java supports x but not Python)

  Resolution: Language differences are addressed in this document.

- Is there a way to identify "approved" API features from those not yet tested?
  - Tell SWIG to expose only some pieces of a class without a lot of hand edits?
  - Tell Doxygen to make an API specific build?
  - Identify API pieces in the full Doxygen build?

  Resolution: This item is still under consideration. There is no API marking of features at this time.

- GMAT already has a `Create` object, producing a name conflict with the API `Create` function. For now, I'm using `Construct`.

  Resolution: Construct is the API creation mechanism.

- We need to report when things fail. Example: Setup("ThisIsntAStartupFile.txt") fails to initialize the Moderator.

  Resolution: This is currently ad hoc: issues GMAT encountered that throw exceptions are reported, but general failures are not.

A.2 Review Responses

The GMAT API was reviewed by interested parties on February 15, 2019. Five RFAs (Request for Action) reports are filed based on this review. The design updates based on these RFAs are summarized here.

RFA ID	Title	Reporter
APIRFA01	Unsafe C++ API	Joel Parker
APIRFA02	Python version compatibility	Joel Parker
APIRFA03	Handling of persistent objects in GMAT memory when using the API	Jacob Englander
APIRFA04	Complexity of "Create" Interface	Steve Hughes
APIRFA05	Rename and simplify the Setup() command	Steve Hughes
APIRFA06	API Style Guide	Steve Hughes

A.2.1 API RFA 01 - Unsafe C++ API

Requested action

The target languages for the API were presented as including a C++ API, which is available "for free" since GMAT is written in C++ natively. But if this is intended for end users, the stable "public" API will be interspersed with the existing internal-only API, which may cause unintended consequences.

The team should look into the need for a public-facing C++ API, and if there is a straightforward way (e.g. in SWIG) to separate the public "safe" C++ API from the internal core version.

Supporting Rationale

Users expecting a production-quality safe API in C++ may be confused or misled by the availability of unsafe, internal-only calls if the two are mixed. This could lead users to inadvertently using unsafe calls, or avoiding the C++ API altogether.

Response

The "C++ API" for the design is a bit of a misnomer for two reasons. First, SWIG is a tool that wraps C++ code for **other** languages. As such, there is no API product for C++ users. Second, the typical use case for the API is one that follows the procedure

- Open interactive session on the target platform
- Load GMAT into the active session
- Build objects (either by hand or by loading a platform specific script)
- Exercise the objects (either interactively or through platform specific scripts)

C++ users will benefit from the new Create() and Initialize() functions provided for API users. These functions automate the object interconnections for GMAT components, initially for those targeted by the API use cases, and eventually over a much broader range of classes.

For C++ users, there is no interactive environment like that supplied by Python and MATLAB. C++ works on a "compile and run" paradigm; as such, it does not supply a mechanism for the interactive use that makes API functions like the GMAT API's Help() functions useful. C++ users can call that function, but they only see the output after compiling and running the compiled program, making the functionality limited at best. Similarly for the other additions to GMAT for the API. The functionality is generally available, but of limited use when working in GMAT's native C++ language.

The design document has been updated (see the note in *User Experiences*), intended to clarify how C++ fits into the API.)

A.2.2 API RFA 02 - Python version compatibility

Requested action

The reliance of the current GMAT API on a specific major.minor version of Python is not ideal, as it could lead to extensive configuration issues on the user side to match versions when using GMAT, alongside other tools that potentially have differing requirements. The team should investigate mitigations for this, and confirm that: a) this specific minor-version reliance is necessary, b) that the chosen version is the best one, and c) that there are no better options that could be selected to enhance ease of configuration. Investigating the approach of other Python APIs may help here.

Supporting Rationale

The current GMAT API relies on a specific Python major.minor version as of the time of the API release. GMAT will not likely be the only tool being used on end-user machines that requires Python, so that specific reliance may cause problematic configuration issues. Further, GMAT will have a different release schedule from other tools, requiring reconfiguration on each upgrade to make sure things don't get broken. It may result in less adoption of the GMAT API if setup is known to be difficult and fragile.

Response

The GMAT Python interface relies on specific Python major/minor versioning. The SWIG documentation distinguishes between Python 2.x and Python 3.x for some features, but does not distinguish minor versions. However, SWIG is generating C++ wrapper code which is then compiled and linked. The issue comes with the link step: the shared library links to a specific Python library, which is then a dependency for its use. More about Python binary compatibility can be found at https://docs.python.org/3.7/c-api/stable.html.

The API team did try building the current API using the Py_LIMITED_API option discussed at the link. That option is incompatible with the current release of SWIG.

The development team will track this issue during implementation. If a solution is found, it will be implemented and recommended for inclusion with GMAT's Python interface build. If not, the Python linkage will be made to the same library as is used for the GMAT Python interface in order to avoid library conflicts.

A.2.3 API RFA 03 - Handling of persistent objects in GMAT memory when using the API

Requested action

Potential user actions that might cause memory leaks or other issues as objects to go out of scope between GMAT and user code should be explored and tested as necessary.

If I understand correctly from the review, when the user instantiates an object in their Python (or MATLAB or C++) workspace, GMAT creates that object and passes it by reference to the calling program. My question is, what happens if the user deletes that object in their calling program or if it just goes out of scope.

For instance, if the calling program does:

Listing 1.1: RFA 03 Example 1

```
myThing = GMAT.createThing()
myThing.doStuff()
del myThing
```

The above code will result in the Python object myThing being deleted and therefore the reference to GMAT's Thing is deleted. But what happens to the Thing that GMAT created dynamically? Is it deleted when Python's reference to it is deleted? Or does it persist?

Similarly, suppose you are doing this in a loop

Listing 1.2: RFA 03 Example 2

```
for (some range):
    myThing = GMAT.createThing()
    myThing.doStuff()
```

In the above code, myThing is an automatic variable that vanishes as soon as it goes out of scope. Since its scope is one particular iteration of the loop, myThing will be created and destroyed n times. What happens to the GMAT object on the other end of the myThing reference? Does it get deleted, too, or do we get n of them in memory? If the latter, is there any way to get rid of them or do we have to have the calling program close GMAT and re-open it?

Does the answer change if the calling program is in C++ vs Python or MATLAB? I know that I could create and destroy objects the "right" way by calling into GMATbase if I were an expert in the GMAT codebase, but when you create a clean C++ API then this issue could come up.

Supporting Rationale

I'm worried about creating a memory leak if the user creates and destroys many GMAT objects in the calling program.

Response

Objects created by SWIG generated code are C++ objects contained inside of a target platform wrapper. By default, these objects are managed using the platform's memory management facilities. For example, if a constructor for a GMAT object is accessed directly from Java, the resulting object wrapper is managed in Java. When the wrapper goes out of scope, the Java garbage collector can delete it. When the wrapper is deleted, it calls the destructor of the contained object. That call can be overridden using a SWIG setting that releases object management to the underlying C++ code on an object by object basis.

The GMAT API will manage this setting inside of the Create() function. Objects constructed using the Create() function will me managed inside of the GMAT configuration manager. Repeated calls to Create() for the same named object will return the reference/pointer to the object created on the first call, as long as that object remains in the configuration. Memory management for objects created using constructor calls outside of the Create() function are the responsibility of the platform making the call.

As with all C/C++ programming, we will need to watch the memory management closely as work proceeds, and provide as much support as we can to making it transparent to the user when possible, and simple to understand when necessary.

A.2.4 API RFA 04 - Complexity of "Create" Interface

Requested action

The examples for object initialization were relatively simple:

Listing 1.3: RFA 04 Example 1

```
Create("Spacecraft", "EO1")
Create("CoordinateSystem", "ECI", "Earth", "MJ2000Eq")
```

In general, there are a lot of special cases where certain data is required (for example, if the Coordinate system axes above were "ObjectReferenced" instead of "MJ2000Eq" additional information is required on the Primary and Secondary bodies.

The design needs to clarify how required fields for object creation are handled, especially when the choice of one setting changes on a model dramatically changes what other information is required to configure that model such as when a Propagator Type is SPK instead of an integrator. Additionally,

The design also needs to clarify how required vs. optional information will be handled at object creation. For example, Spacecraft has numerous settings. For complex objects what will be required at construction, and what will be set via "Set()" methods.

Supporting Rationale

The API will be quite difficult to use, and possibly error prone, if the complexity of object initialization is not well designed and well documented.

Response

Object creation is a straigtforward call to the constructor for the object. The Create command is used to make constructor calls and place object management under control of the GMAT engine by placing the objects into the configuration database. (See the description of memory management for RFA 3, above).

The concern here is more about object initialization, performed by the Setup() command in the draft document, now renamed Initialize() (see RFA 5, below). As stated in the *description of that command*, the object to object interconnections are set when Initialize() is called prior to object use. Missing elements are identified in the return from that call.

As the reviewer notes, there is a lot of complexity in some of the GMAT objects. Part of the challenge of coding the changes needed for the API is covering that complexity. The development team will concentrate efforts on making high usage objects as robust as possible in this regard, with a focus on clear configuration messages made to the users for the objects used in the use cases.

While we will make our best efforts to address the object complexity issues during the initial year of development, we expect that we will revisit this issue once use case 1 has been implemented, in hopes of refining the strategy that addresses it at a larger scope than a case by case approach.

A.2.5 API RFA 05 - Rename and simplify the Setup() command

Requested action

The Setup()/initialization process is confusing and possible dangerous because omitting the call may result in an undesirable state. The code example in the review was:

Listing 1.4: RFA 05 Example 1

```
import gmat_py as gmat
gmat.Setup("MyCustomStartupFile.txt")
csConverter = gmat.CoordinateConverter()
eci  = gmat.Create("CoordinateSystem", "ECI", "Earth", "MJ2000Eq")
ecef = gmat.Create("CoordinateSystem", "ECEF", "Earth", "BodyFixed")
gmat.Setup()
csConverter.Convert(mjd, rvIn, eci, rvOut, ecef)
```

During the review, it sounded like it is relatively easy to eliminate the first call to setup by modifying the import line to contain startup file configuration.

The second call confused a lot of people, and there is concern that forgetting to call SetUp() could result in an issue in the configuration that is not obvious to the user. The main confusion is that typically in an API, once an object is constructed, it is ready for use. That is not the case in the current design. It appears the SetUp() call is similar to BeginMissionSequence, and is required due to the existing GMAT design. If that is not correct, consider designs that do not require a Setup() call. If the existing design of GMAT requires such a call, consider using a more descriptive set of function names and provide a way to create and initialize new objects during execution. For example

Listing 1.5: RFA 05 Example 1

```
import gmat_py as gmat
csConverter = gmat.CoordinateConverter()
eci  = gmat.Create("CoordinateSystem", "ECI", "Earth", "MJ2000Eq")

gmat.InitializeObjects()

csConverter.Convert(mjd, rvIn, eci, rvOut, ecef)
// Based on previous execution, determined we need a new model, so create
// during execution and use. Need to initialize that object but leave the
// state of the rest of the objects alone.
ecef = gmat.Create("CoordinateSystem", "ECEF", "Earth", "BodyFixed")

// only initialize ecef, other objects are not re-initialized
gmat.InitializeObjects(ecef)
```

Finally, if an object has not been initialized before use, the API must provide that information back to the user (probably as an exception).

Supporting Rationale

The single biggest concern for users/reviewers was the call to SetUp(). Existing GMAT design may require a Setup()-like call, but if that is the case, care must be taken in the design and documentation should be clear, and user errors need to be trapped to avoid un-intended non-obvious failure modes.

Response

The originally proposed Setup() command has been renamed Initialize(). Based on the work one team member performed after the original design document was written, we agree that we can incorporate the core GMAT initialization when we load the system into the target platform's environment, so you'll see that the examples in the design document no longer include that first call.

The purpose of the Initialize() call is to establish object-to-object interconnections, and to validate that the objects are ready for use. This is different from the BeginMissionSequence behavior in GMAT, though. GMAT has a requirement

when running a script that all of the scripted components be set and that their references also be connected. In GMAT, this occurs after objects are loaded into the GMAT Sandbox. That step is performed, in the API, using the Initialize() command.

One result of initialization here is the generation of a list of connections that were needed but not found. We may present this as an exception, but may present it in a more user friendly format.

A.2.6 API RFA 06 - API Style Guide

Requested action

Develop and maintain an API style guide as development progresses.

Supporting Rationale

Low-level models and functions in GMAT do not always have a consistent interface style between them. We should ensure as we expose models via the API, that the interface style and "wrappers" are consistent between models to make the API easy to use.

Response

The development team will include a style guide in the user documentation. A placeholder for it has been added to the User's Guide table of contents.

A.3 Comparison of the SWIG Prototype with the Production API

The prototype SWIG generated API developed in 2016 can be used with GMAT R2018. This appendix shows the differences between that version of the SWIG generated API and the production API presented in this document for the three examples described in *API Examples*.

A.3.1 Time System Conversion

Listing 1.6: Time System Conversion using GMAT R2018a SWIG Configuration

```
import gmat_py as gmat

# Initializing the Moderator configures the converter
gmat.Moderator.Instance().Initialize('gmat_startup_file.txt')

# Get the converter
timeConverter = gmat.TimeSystemConverter.Instance()

# Convert an epoch
UTCEpoch = 21738.22145
TAIepoch = timeConverter.Convert(UTCepoch, 2, 1)
```

Listing 1.7: Time System Conversion using the Production API

```
import gmat_py as gmat

# Get the converter
timeConverter = gmat.theTimeSystemConverter

# Convert an epoch
UTCEpoch = 21738.22145
TAIepoch = timeConverter.Convert(UTCepoch, UTC, TAI)
```

A.3.2 Coordinate System Conversion

Listing 1.8: Coordinate System Conversion using GMAT R2018a SWIG Configuration

```
import gmat_py as gmat

# Initialize to set default objects needed to configure the converter
mod = gmat.Moderator.Instance()
mod.Initialize('gmat_startup_file.txt')

# Setup the GMAT data structures for the conversion
mjd   = gmat.A1Mjd(22326.977184)
rvIn  = gmat.Rvector6(6988.426918, 1073.884261, 2247.332981, 0.019982, 7.226988, -1.
↪554962)
rvOut = gmat.Rvector6(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

# Create the converter
csConverter = gmat.CoordinateConverter()

# Get the solar system and central body
ss = mod.GetSolarSystemInUse()
earth = ss.GetBody("Earth")

# Create the input and output coordinate systems
eci  = gmat.CoordinateSystem.CreateLocalCoordinateSystem(
    "ECI", "MJ2000Eq", earth, None, None, earth, ss)
ecef = gmat.CoordinateSystem.CreateLocalCoordinateSystem(
    "ECEF", "BodyFixed", earth, None, None, earth, ss)

csConverter.Convert(UTCepoch, rvIn, eci, rvOut, ecef)
```

Listing 1.9: Coordinate System Conversion using the Production API

```
import gmat_py as gmat

# Setup the GMAT data structures for the conversion
mjd   = gmat.A1Mjd(22326.977184)
rvIn  = gmat.Rvector6(6988.426918, 1073.884261, 2247.332981, 0.019982, 7.226988, -1.
↪554962)
rvOut = gmat.Rvector6(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

# Create the converter
```

(continues on next page)

(continued from previous page)

```
csConverter = gmat.CoordinateConverter()

# Create the input and output coordinate systems
eci  = gmat.Create("CoordinateSystem",
    "ECI", "Earth", "MJ2000Eq")
ecef = gmat.Create("CoordinateSystem",
    "ECEF", "Earth", "BodyFixed")

csConverter.Convert(UTCepoch, rvIn, eci, rvOut, ecef)
```

A.3.3 Force Modeling

Listing 1.10: Force modeling using GMAT R2018a SWIG Configuration

```
import gmat_py as gmat
mod = gmat.Moderator.Instance()
mod.Initialize('gmat_startup_file.txt')

# Spacecraft setup
sc = gmat.Spacecraft("sc")

# Evaluating only the 6 element Cartesian state
state = [7000.0, 0.0, 1000.0, 0.0, 8.0, -0.25]
pstate = gmat.new_doubleArray(6)
for i in range(len(state)):
    gmat.doubleArray_setitem(pstate, i, state[i])

# Internal required objects
ss = mod.GetDefaultSolarSystem()
earth = ss.GetBody("Earth")
eci  = gmat.CoordinateSystem.CreateLocalCoordinateSystem(
        "ECI", "MJ2000Eq", earth, None, None, earth, ss)

# The state manager
psm = gmat.PropagationStateManager()
psm.SetObject(sc)
psm.BuildState()
psm.MapObjectsToVector()

# The force model
dynamics = gmat.ODEModel("EPointMassDynamics")
dynamics.SetForceOrigin(earth)

epm = gmat.PointMassForce("EarthPointMass")
dynamics.AddForce(epm)

# Manage memory - dynamics now owns the epm, so Python should not delete it
epm.thisown = 0

dynamics.SetPropStateManager(psm)
dynamics.SetSolarSystem(ss)
dynamics.SetInternalCoordSystem(eci)
dynamics.BufferState()
dynamics.BuildModelFromMap()

```

(continues on next page)

(continued from previous page)

```
dynamics.UpdateInitialData()
dynamics.Initialize()

pderiv = dynamics.GetDerivativeArray()
dynamics.GetDerivatives(pstate, 0.0)
```

Listing 1.11: Force Modeling using the Production API

```
import gmat_py as gmat

dynamics = gmat.Create("ODEModel", "EPointMassDynamics")
epm = gmat.PointMassForce("EarthPointMass")
dynamics.AddForce(epm)

# Evaluating only the 6 element Cartesian state
pstate = [7000.0, 0.0, 1000.0, 0.0, 8.0, -0.25]

gmat.Initialize()

pderiv = dynamics.GetDerivativeArray()
dynamics.GetDerivatives(pstate, 0.0)
```

A.3.4 Force Modeling and Propagation

Listing 1.12: Propagation using GMAT R2018a SWIG Configuration

```
import gmat_py as gmat
mod = gmat.Moderator.Instance()
mod.Initialize('gmat_startup_file.txt')

# Setup the state for propagation
state = [7000.0, 0.0, 1000.0, 0.0, 8.0, -0.25]

# Load some pieces we need to configure the system
ss = mod.GetDefaultSolarSystem()
sc = gmat.Spacecraft("sc")

psm = gmat.PropagationStateManager()
psm.SetObject(sc)
psm.BuildState()
psm.MapObjectsToVector()

# Setup a Earth/Sun/Moon force model
# note: Use Moderator for the forces and Python memory management won't segfault
dynamics = gmat.ODEModel("Forces")

epm = mod.CreatePhysicalModel("PointMassForce", "EarthPointMass")
spm = mod.CreatePhysicalModel("PointMassForce", "SunPointMass")
mpm = mod.CreatePhysicalModel("PointMassForce", "MoonPointMass")
spm.SetStringParameter("BodyName", "Sun")
mpm.SetStringParameter("BodyName", "Luna")

dynamics.AddForce(epm)
dynamics.AddForce(spm)
```

(continues on next page)

(continued from previous page)

```
dynamics.AddForce(mpm)

# Manage memory
epm.thisown = 0
spm.thisown = 0
mpm.thisown = 0

# Reference object setup
dynamics.SetPropStateManager(psm)
dynamics.SetSolarSystem(ss)

# ODE model initialization
dynamics.BufferState()
dynamics.BuildModelFromMap()
dynamics.UpdateInitialData()
rv = dynamics.Initialize()

# Propagator configuration
prop = gmat.PrinceDormand78("Propagator")
prop.SetSolarSystem(ss)
prop.SetPropStateManager(psm)
prop.SetPhysicalModel(dynamics)
prop.Initialize()

# Set the propagation state
pstate = dynamics.GetState()
for i in range(len(state)):
    gmat.doubleArray_setitem(pstate, i, state[i])

for i in range(count):
    prop.Step(60.0)
```

Listing 1.13: Propagation using the Production API

```
import gmat_py as gmat

# Setup the state for propagation
state = [7000.0, 0.0, 1000.0, 0.0, 8.0, -0.25]

# Setup a Earth/Sun/Moon force model
# note: Use Moderator for the forces and Python memory management won't segfault
dynamics = gmat.Create("ODEModel", "Forces")

epm = mod.Create("PointMassForce", "EarthPointMass")
spm = mod.Create("PointMassForce", "SunPointMass")
mpm = mod.Create("PointMassForce", "MoonPointMass")
spm.SetField("BodyName", "Sun")
mpm.SetField("BodyName", "Luna")

dynamics.AddForce(epm)
dynamics.AddForce(spm)
dynamics.AddForce(mpm)

# Propagator configuration
prop = gmat.PrinceDormand78("Propagator")
prop.SetPhysicalModel(dynamics)
```

(continues on next page)

(continued from previous page)

```

gmat.Initialize()

# Set the propagation state
pstate = dynamics.GetState()
for i in range(len(state)):
    gmat.doubleArray_setitem(pstate, i, state[i])

for i in range(count):
    prop.Step(60.0)
```

A.4 API Examples in Java

API Examples shows Python scripting for four common GMAT API use cases. This section shows those same use cases in Java.

A.4.1 Time System Conversion

Listing 1.14: Time System Conversion in Java

```
import gmat.*;

public class TimeConvNew {

    public static void main(String[] args) {

        // Get the converter
        TimeSystemConverter timeConverter = gmat.theTimeSystemConverter;

        // Convert an epoch
        double UTCepoch = 21738.22145;
        double TAIepoch = timeConverter.Convert(UTCepoch, UTC, TAI);
   }
}
```

A.4.2 Coordinate System Conversion

Listing 1.15: Coordinate Conversion in Java

```
import gmat.*;

public class CoordConvNew {

    public static void main(String[] args) {

        // Initialize GMAT
        gmat.Setup("MyCustomStartupFile.txt");

        // Setup the GMAT data structures for the conversion
        A1Mjd mjd = new A1Mjd(22326.977184);
```

(continues on next page)

(continued from previous page)

```
        Rvector6 rvIn = new Rvector6(6988.427, 1073.884, 2247.333, 0.019982, 7.226988,
↪ -1.554962);
        Rvector6 rvOut = new Rvector6(0.0, 0.0, 0.0, 0.0, 0.0, 0.0);

        // Create the converter
        CoordinateConverter csConverter = new CoordinateConverter();

        // Create the input and output coordinate systems
        CoordinateSystem eci = gmat.Create("CoordinateSystem",
                "ECI", "Earth", "MJ2000Eq");
        CoordinateSystem ecef = gmat.Create("CoordinateSystem",
                "ECEF", "Earth", "BodyFixed");

        csConverter.Convert(mjd, rvIn, eci, rvOut, ecef);
  }
}
```

A.4.3 Force Modeling

Listing 1.16: Force Model Creation and Use in Java

```
import gmat.*;

public class ForceModelNew {

  public static void main(String[] args)
  {
    ODEModel dynamics = gmat.Create("ODEModel","FM");

    PointMassForce epm = gmat.Create("PointMassForce","EPM");
    dynamics.AddForce(epm);

    gmat.Initialize();

    dynamics.GetDerivatives(state, dt);
    double[] derivatives = dynamics.GetDerivativeArray();
  }
}
```

A.4.4 Propagation

Listing 1.17: Propagation in Java

```
import gmat.*;

public class PropExampleNew {

    public static void main(String[] args) {

        // Setup the state for propagation
        double[] state = {7000.0, 0.0, 1000.0, 0.0, 8.0, -0.25};
        // Setup a Earth/Sun/Moon force model
        // note: Use Moderator for the forces and Python memory management won't seg␣
↪fault
```

(continues on next page)

(continued from previous page)

```
        ODEModel dynamics = gmat.Create("ODEModel", "Forces");

        PhysicalModel epm = gmat.Create("PointMassForce", "EarthPointMass");
        PhysicalModel spm = gmat.Create("PointMassForce", "SunPointMass");
        PhysicalModel mpm = gmat.Create("PointMassForce", "MoonPointMass");
        spm.SetStringParameter("BodyName", "Sun");
        mpm.SetStringParameter("BodyName", "Luna");

        dynamics.AddForce(epm);
        dynamics.AddForce(spm);
        dynamics.AddForce(mpm);

        // Propagator configuration
        PrinceDormand78 prop = new PrinceDormand78("Propagator");
        prop.SetPhysicalModel(dynamics);

        gmat.Initialize();

        // Set the propagation state
        dynamics.SetState(state);

        for (int i = 0; i < 10; i++) {
            prop.Step(60.0);
        }
    }
}
```

APPENDIX

B

GMAT API CHEAT SHEET

B.1 Loading the API

Python	MATLAB
import gmatpy as gmat	load_gmat

B.2 Asking for Help

Python	MATLAB
gmat.Help()	gmat.gmat.Help()
gmat.Help(<topic>)	gmat.gmat.Help(<topic>)
object.Help()	object.Help()
Examples	
gmat.Help("ScriptUsage")	gmat.gmat.Help("Objects")
burn.Help()	burn.Help()

B.3 GMAT Objects

B.3.1 Listing Available Classes

Python	MATLAB
gmat.ShowClasses()	gmat.gmat.ShowClasses()
gmat.ShowClasses(<type>) | gmat.gmat.ShowClasses(<type>)	
Examples	
gmat.ShowClasses()	gmat.gmat.ShowClasses()
gmat.ShowClasses("PhysicalModel")	gmat.gmat.ShowClasses("Propagator")

B.3.2 Listing Created Objects

Python	MATLAB
gmat.ShowObjects()	gmat.gmat.ShowObjects()
gmat.ShowObjects(<type>)	gmat.gmat.ShowObjects(<type>)
Examples	
gmat.ShowObjects()	gmat.gmat.ShowObjects()
gmat.ShowObjects("Spacecraft")	gmat.gmat.ShowObjects("Burn")

B.3.3 Object Creation

Python	MATLAB
obj = gmat.Construct(<type>,<name>)	obj = gmat.gmat.Construct(<type>, <name>)
Examples	
Python	MATLAB
burn = gmat.Construct("ImpulsiveBurn", "Burn")	burn = gmat.gmat.Construct("ImpulsiveBurn","Burn")

B.3.4 Object Field Access

Python	MATLAB
value = obj.GetField(<FieldLabel>)	value = obj.GetField(<FieldLabel>)
obj.SetField(<FieldLabel>,<value>)	obj.SetField(<FieldLabel>,<value>)
obj.GetNumber(<FieldLabel>)	obj.GetNumber(<FieldLabel>)
Examples	
V = burn.GetField("Element1")	V = burn.GetField("Element1")
burn.SetField("Element1",1.5)	SetField("Element1",1.5)
burn.SetField("Origin","Mars")	SetField("Origin","Mars")
V = burn.GetNumber("Element1")	V = burn.GetNumber("Element1")

B.3.5 Object to Object Connections

Python	MATLAB
obj.SetReference(<RefObject>)	obj.SetReference(<RefObject>)
Examples	
jrdrag.SetReference(atmos)	jrdrag.SetReference(atmos)

B.4 GMAT Script Access

B.4.1 Loading a GMAT script

Python	MATLAB
gmat.LoadScript(<script>)	gmat.gmat.LoadScript(<script>)
Examples	
gmat.LoadScript("../samples/Ex_GEOTransfer.script")	gmat.gmat.LoadScript("../samples/Ex_GEOTransfer.script")

B.4.2 Running a GMAT script

Python	MATLAB
gmat.RunScript()	gmat.gmat.RunScript()

B.4.3 Saving a GMAT script

Python	MATLAB
gmat.SaveScript(<script>)	gmat.gmat.SaveScript(<script>)
Examples	
gmat.SaveScript("../scripts/New_GEOTransfer.script")	gmat.gmat.SaveScript("../scripts/New_GEOTransfer.script")

B.4.4 Accessing Run Data After a Run

Python	MATLAB
gmat.GetRuntimeObject(<name>)	gmat.gmat.GetRuntimeObject(<name>)
gmat.GetRunSummary()	gmat.gmat.GetRunSummary()
Examples	
gmat.GetRuntimeObject("geoSat")	gmat.gmat.GetRuntimeObject("geoSat")

APPENDIX

C

API NOTEBOOK WALKTHROUGHS

API builds include several *[Jupyter]* notebooks illustrating specific features of the interface. These notebooks can be found in the api/Jupyter folder of the GMAT build. Users with access to the Jupyter system can run these notebooks interactively. Static versions of the notebooks are included in this chapter.

C.1 State Management with the GMAT API

The state data in GMAT can be a bit confusing. This notebook introduces the state variables as used for a GMAT Spacecraft, and provides some pointers on the manipulation of the state data.

C.1.1 Prepare the GMAT Environment

Before the API can be used, it needs to be loaded into the Python system and initialized using a GMAT startup file. This can be done from the GMAT bin folder by importing the gmatpy module, but using that approach tends to leave pieces in the bin folder that may annoy other users. Running from an outside folder takes a few steps, which have been captured in the run_gmat.py file imported here:

```
from run_gmat import *
```

C.1.2 Configure a Spacecraft

We'll need an object that provides the state. Here's 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 here, 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:

```
iState
```

```
<gmatpy.gmat_py.GmatState; proxy of <Swig Object of type 'GmatState *' at␣
↪0x7f13ceed97e0> >
```

GmatState objects are used to collect together an epoch and a vector of data. These data can be accessed directly:

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

```
The state epoch is  21545.000000397937 , the state has  6  elements, and contains the␣
↪data  [-999.999, -999.999, -999.999, -999.999, -999.999, -999.999]
```

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 are filled in with the dummy entries -999.999. ## 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. 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 at first glance 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 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.

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):
↪", iState.GetState())
print("but access to the Keplerian state shows that it is not correct:", sat.
↪GetKeplerianState())
```

```
The internal state buffer just holds preinitialization data (Keplerian here):   [7015.
↪0, 0.0011, 98.6, 75.0, 90.0, 33.333]
but access to the Keplerian state shows that it is not correct: 0                0   ␣
↪           0               0               0               0
```

The GMAT objects are not yet initialized, so the Keplerian state data is not correct. 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 Initialize() function:

```
.. code:: ipython3
```

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

```
The initialized internal state buffer is EarthMJ2000Eq:   [-150.99058171804361, -3946.
↪626071010534, 5789.742898439815, -2.23046049968889, -5.931020059857665, -4.
↪095581409074377]
```

(continues on next page)

(continued from previous page)

```
and the Keplerian state is correct:   7015.000000000001 0.00110000000000004 98.
↪59999999999999 75                90.00000000000402 33.33299999999598
```

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

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

```
Internal state:   [-172.19168264352888, -4500.785255607168, 6602.700383110265, -2.
↪088638988531676, -5.553902317709759, -3.835168124650226]
Cartesianian      -172.1916826435289 -4500.785255607168 6602.700383110265 -2.
↪088638988531676 -5.553902317709759 -3.835168124650226
Keplerian:        8000.000000000007 0.001100000000000707 98.59999999999999 75
↪       90.00000000001161 33.3329999999884

Internal state:   [-5697.7414619496, -3020.2186545041395, 4721.90552145557, 1.
↪1208679033712874, -6.413887097497282, -2.7427113896944304]
Cartesianian      -5697.7414619496 -3020.21865450414 4721.90552145557 1.
↪120867903371287 -6.413887097497282 -2.74271138969443
Keplerian:        8000.000000000011 0.001100000000000964 45.00000000000001 75.
↪00000000000001 90.00000000001836 33.33299999998167

Internal state:   [-5094.78342738948, -4974.9069027511405, 3633.5822378210464, 2.
↪5169011956354206, -5.379735538828468, -3.8235178821457656]
Cartesianian      -5094.78342738948 -4974.90690275114 3633.582237821046 2.
↪516901195635421 -5.379735538828468 -3.823517882145766
Keplerian:        8000.000000000012 0.001100000000001075 45.00000000000001 75.
↪00000000000001 90.00000000002314 49.99999999999523
```

C.1.3 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 an 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 ECI 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())
```

```
The spacecraft  MySat  initialization state is  True
The internal state buffer:   [-5094.78342738948, -4974.9069027511405, 3633.
↪5822378210464, 2.5169011956354206, -5.379735538828468, -3.8235178821457656]
The ECF Cartesian State:     -5094.78342738948 -4974.90690275114 3633.582237821046 2.
↪516901195635421 -5.379735538828468 -3.823517882145766
The ECF Keplerian State:     8000.000000000012 0.001100000000001075 45.00000000000001␣
↪75.00000000000001 90.00000000002314 49.99999999999523

The spacecraft  SolarSat  initialization state is  True
The internal state buffer (SolarSat):   [-5094.78342738948, -4974.9069027511405, 3633.
↪5822378210464, 2.5169011956354206, -5.379735538828468, -3.8235178821457656]
The SolarEcliptic Cartesian State:      -5094.78342738948 -4974.90690275114 3633.
↪582237821046 2.516901195635421 -5.379735538828468 -3.823517882145766
The SolarEcliptic Keplerian State:      8000.000000000012 0.001100000000001075 45.
↪00000000000001 75.00000000000001 90.00000000002314 49.99999999999523
```

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())
```

(continues on next page)

(continued from previous page)

```
print("The SolarEcliptic Cartesian State:     ", solsat.GetCartesianState())
print("The SolarEcliptic Keplerian State:     ", solsat.GetKeplerianState())
```

```
The internal state buffer:   [-5094.78342738948, -4974.9069027511405, 3633.
↪5822378210464, 2.5169011956354206, -5.379735538828468, -3.8235178821457656]
The ECF Cartesian State:     3980.769626359613 -5904.072200723337 3633.84663580491 5.
↪31337371013498 1.221190102125526 -3.82343374194999
The ECF Keplerian State:     7197.708272712511 0.1106817774544226 47.10837940070086
↪152.2889386356222 322.0637563061007 179.5887814511714

The internal state buffer (SolarSat):   [-5094.78342738948, -4974.9069027511405, 3633.
↪5822378210464, 2.5169011956354206, -5.379735538828468, -3.8235178821457656]
The SolarEcliptic Cartesian State:      -26505087.9080278 144694001.6268158 4700.
↪442019894719 -27.27732399951776 -11.92620879192113 -1.367891168160935
The SolarEcliptic Keplerian State:      144849901.1130946 0.2292154440704447 2.
↪702016602265948 280.4191667873194 286.9680459339144 252.9931176051724
```

C.2 Propagation with the GMAT API

This document walks you through the configuration and use of the GMAT API for propagation.

C.2.1 Prepare the GMAT Environment

Before the API can be used, it needs to be loaded into the Python system and initialized using a GMAT startup file. This can be done from the GMAT bin folder by importing the gmatpy module, but using that approach tends to leave pieces in the bin folder that may annoy other users. Running from an outside folder takes a few steps, which have been captured in the run_gmat.py file imported here:

```
from run_gmat import *
```

C.2.2 Configure a Spacecraft

We'll need an object to propagate. Here's a basic one:

```
sat = gmat.Construct("Spacecraft","LeoSat")

sat.SetField("DateFormat", "UTCGregorian")
sat.SetField("Epoch", "27 Sep 2019 15:05:00.000")
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)
```

C.2.3 Configure the Forces

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. The force model is built and its (empty) contents displayed using

```
fm = gmat.Construct("ForceModel", "TheForces")
fm.Help()
```

```
ForceModel  TheForces

   Field                                    Type    Value
   ------------------------------------------------------

   CentralBody                           Object    Earth
   PrimaryBodies                    ObjectArray    {}
   PolyhedralBodies                 ObjectArray    {}
   PointMasses                      ObjectArray    {}
   Drag                                  Object    None
   SRP                                    OnOff    Off
   RelativisticCorrection                 OnOff    Off
   ErrorControl                            List    RSSStep
   UserDefined                      ObjectArray    {}
```

```
''
```

C.2.4 Add the Potential Field

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 forces into the ODEModel container
fm.AddForce(earthgrav)
```

C.2.5 Add the other forces

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)

fm.Help()
```

```
ForceModel  TheForces

   Field                                    Type   Value
   ------------------------------------------------------

   CentralBody                            Object   Earth
   PrimaryBodies                     ObjectArray   {Earth}
   PolyhedralBodies                  ObjectArray   {}
   PointMasses                       ObjectArray   {Luna, Sun}
   Drag                                   Object   JacchiaRoberts
   SRP                                     OnOff   Off
   RelativisticCorrection                  OnOff   Off
   ErrorControl                             List   RSSStep
   UserDefined                       ObjectArray   {}
```

```
''
```

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

```
earthgrav.Help()
```

```
GravityField

   Field                                    Type   Value
   ------------------------------------------------------

   Degree                                Integer   8
   Order                                 Integer   8
   StmLimit                              Integer   100
   PotentialFile                        Filename   JGM2.cof
   TideFile                             Filename
   TideModel                              String   None
```

```
''
```

or, with a little work, the scripting for the complete force model:

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

```
Create ForceModel TheForces;
GMAT TheForces.CentralBody = Earth;
GMAT TheForces.PrimaryBodies = {Earth};
GMAT TheForces.PointMasses = {Luna, Sun};
GMAT TheForces.SRP = Off;
GMAT TheForces.RelativisticCorrection = Off;
GMAT TheForces.ErrorControl = RSSStep;
GMAT TheForces.GravityField.Earth.Degree = 8;
GMAT TheForces.GravityField.Earth.Order = 8;
GMAT TheForces.GravityField.Earth.StmLimit = 100;
GMAT TheForces.GravityField.Earth.PotentialFile = 'JGM2.cof';
GMAT TheForces.GravityField.Earth.TideModel = 'None';
GMAT TheForces.Drag.AtmosphereModel = JacchiaRoberts;
GMAT TheForces.Drag.HistoricWeatherSource = 'ConstantFluxAndGeoMag';
GMAT TheForces.Drag.PredictedWeatherSource = 'ConstantFluxAndGeoMag';
GMAT TheForces.Drag.CSSISpaceWeatherFile = 'SpaceWeather-All-v1.2.txt';
GMAT TheForces.Drag.SchattenFile = 'SchattenPredict.txt';
GMAT TheForces.Drag.F107 = 150;
GMAT TheForces.Drag.F107A = 150;
GMAT TheForces.Drag.MagneticIndex = 3;
GMAT TheForces.Drag.SchattenErrorModel = 'Nominal';
GMAT TheForces.Drag.SchattenTimingModel = 'NominalCycle';
GMAT TheForces.Drag.DragModel = 'Spherical';
```

C.2.6 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)
```

C.2.7 Connect the Objects Together

```
# Assign the force model imported from BasicFM
pdprop.SetReference(fm)
```

(continues on next page)

(continued from previous page)

```
# Setup the spacecraft that is propagated
psm = pdprop.GetPropStateManager()
psm.SetObject(sat)
```

```
True
```

C.2.8 Initialize the System and Propagate a Step

Finally, the system can be initialized and fired to see a single propagation step. Some of the code displayed here will be folded into the API's Initialize() function. For now, the steps needed to initialize the system for a propagation step are:

```
# Perform top level initialization
gmat.Initialize()

# Refresh the object references from the propagator clones
fm = pdprop.GetODEModel()
gator = pdprop.GetPropagator()

psm.BuildState()

# Pass the state manager to the dynamics model
fm.SetPropStateManager(psm)
fm.SetState(psm.GetState())

# Assemble all of the force model objects together
fm.Initialize()

# Finish the force model setup
fm.BuildModelFromMap()
fm.UpdateInitialData()

# Initialize the Propagator components
pdprop.Initialize()
gator.Initialize()
```

```
True
```

Note: Alternatively, the above code can be replaced with a call to the PrepareInternals() function on the propagator. This also removes the need to manually configure the PropagationStateManager. The condensed code can be seen below:

```
# Perform top level initialization
gmat.Initialize()

# Setup the spacecraft that is propagated
pdprop.AddPropObject(sat)
pdprop.PrepareInternals()

# Refresh the object references from the propagator clones
fm = pdprop.GetODEModel()
gator = pdprop.GetPropagator()
```

and we can then propagate, and start accumulating the data

```
# Take a 60 second step, showing the state before and after, and start buffering
# Buffers for the data
time = []
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:  ", 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:   ", t, r, v)
```

```
Starting state:   0.0 [-5455.495852919224, -3637.0485868833093, 2350.0571814448517]␣
↪[3.1318014092807545, -6.423940627548709, -2.545227573417657]
Propped state:    60.0 [-5256.1378692623475, -4014.4902800853415, 2192.4488937848546]␣
↪[3.51104729424038, -6.153042432990329, -2.7064897093764597]
```

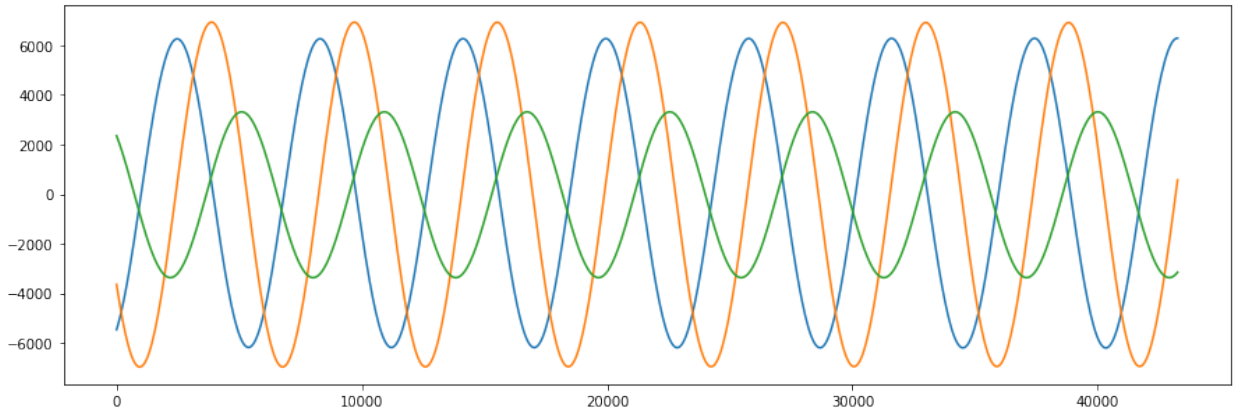
Finally, we can run for a few orbits and show the results

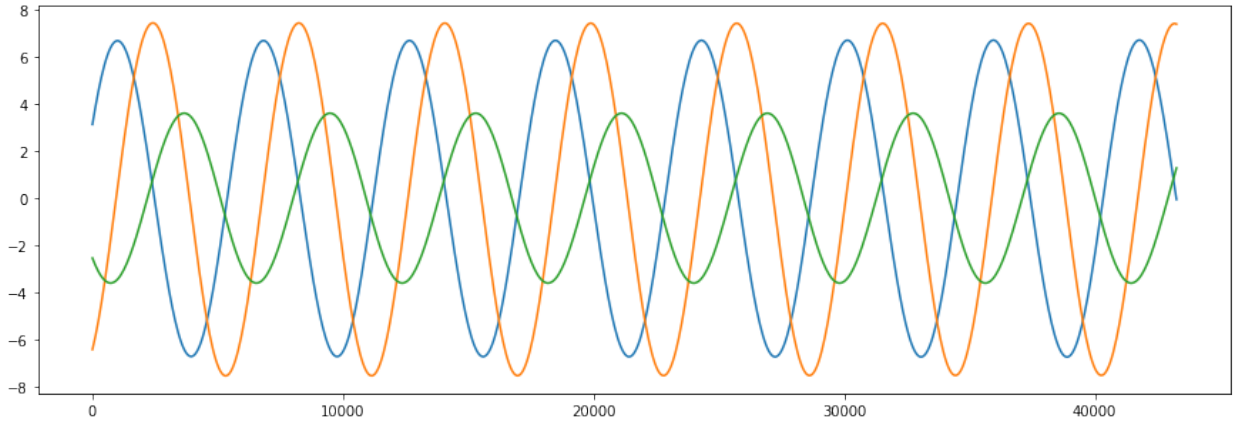
```
for i in range(360):
    # Take a step and buffer it
    gator.Step(60.0)
    gatorstate = gator.GetState()
    t = t + 60.0
    r = []
    v = []
    for j in range(3):
        r.append(gatorstate[j])
        v.append(gatorstate[j+3])
    time.append(t)
    pos.append(r)
    vel.append(v)

import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
plt.rcParams['figure.figsize'] = (15, 5)
plt.plot(time, pos)
plt.show()
plt.plot(time, vel)
plt.show()
```





APPENDIX

D

BIBLIOGRAPHY

[GmatWiki] gmatcentral.org

The GMAT Wiki at gmatcentral.org is the main public facing interface for GMAT development and release activities.

[Architecture] The GMAT Development Team, "General Mission Analysis Tool (GMAT) Architectural Specification," NASA GSFC.

The GMAT Architectural Specification provides a good overview for the GMAT system. The document is included with each GMAT release. The document overview can be viewed at http://gmatcentral.org:8090/display/GW/Architectural+Specification.

[CInterface] D. Conway, "GMAT API Tradeoff Study," Thinking Systems, Inc., February 2012.

The GMAT CInterface plugin was an artifact of the original GMAT API study described here. One recommendation arising from this study was a set of automatic API generation tools, including SWIG.

[SWIGExperiment] D. Conway, "GMAT API Consultation Support," Thinking Systems, Inc., December 2016.

Goddard personnel, assisted by contractors at Thinking Systems, Inc. and Emergent Space Technologies, performed an in-house study of the use of SWIG as a tool for a production GMAT API. This document describes that study.

[SWIG] Simplified Wrapper and Interface Generator (SWIG)

SWIG is an open source software development tool that connects programs written in C and C++ with other high-level programming languages. The GMAT API is generated using the SWIG tool.

[Doxygen] www.doxygen.nl

The detailed design information for GMAT is generated using the open source Doxygen documentation generation tool.

[Jupyter] www.jupyter.org

Jupyter is an interactive tool that provides writers with the ability to intersperse documentation and Python code. The resulting notebook files can be run interactively, enriching the user's learning experience through editable Python code that can be executed inside the notebook.

APPENDIX

E

CHANGE HISTORY

Rev 0	December 05, 2018	Original draft specification.
Rev 1	March 29, 2019	Draft specification incorporating review comments.
Rev 2	October 11, 2019	Updates for the first beta release.
Rev 3	January 15, 2020	Updates for GMAT R2020a.