AAS 20-580

USING THE GMAT APPLICATION PROGRAMMER'S INTERFACE

Darrel J. Conway[*]
John McGreevy[†]

The General Mission Analysis Tool (GMAT) is an open-source astrodynamics tool, developed in partnership between NASA's Goddard Space Flight Center (GSFC), industry partners, and academic institutions.[1] GMAT is in operational use in the GSFC Flight Dynamics Facility, providing guidance and navigation support for active missions, and as a mission design tool at NASA and industrial organizations.

The GMAT R2020a release includes a new GMAT Application Programming Interface (API) that provides access to core GMAT capabilities from Python and Java, and, through the Java interface, from MATLAB. GMAT's API is a tested and documented beta-quality feature of the 2020 release. It includes sample scripts that can be executed in Python or MATLAB, tutorial walk-throughs presented as interactive notebooks, and more than 70 pages of user documentation. The GMAT API can also be run interactively. When working with the tool in this mode, users have access to a live help system that presents available options to the user to simplify use of the system. Configuration for the GMAT API is straightforward on Linux, Windows, and Mac workstations. This paper presents these features and several use cases for the GMAT API.

One core feature of the GMAT API is the access to GMAT's core components for direct manipulation and use. Access to GMAT's components enables the development of tutorial materials, presented as Jupyter notebooks, that introduce users to GMAT's components and walk users through interactive use of these components using the native user interface elements of Python or MATLAB. The GMAT API provides a toolkit to build graphical tools hosted in language native environments. Access to GMAT's computational models for propagation and navigation models enable use of those features for analysis of a user's data, including live streaming of measurements into custom tools to validate those data streams in real time. This API provides users with easy access to a library of astrodynamics utilities that are rigorously tested and trusted operationally by a wide range of NASA and private missions, reducing the need for individual users to implement and test their own utilities. These utilities range from time system conversions, to coordinate system conversions, all the way to simulating measurements including corrections such as light-time delays and relativistic corrections.

All of the GMAT scripting capabilities are accessible using the GMAT API. The script interfaces enable simple configuration of large scale analysis problems, ranging from scans of parameters to explore a solution space to more complex problems like Monte-Carlo analyses. These capabilities are presented with sample problems designed to illustrate their use.

Current work on the GMAT API includes integration of GMAT and Monte into a unified environment, driven from Monte's Python based tool set. The status of this work is also described in this paper.

[*]Senior Scientist and CEO, Thinking Systems, Inc., 437 W Thurber Rd, Suite 6, Tucson, AZ 85705.
[†]Aerospace Engineer, Emergent Space Technologies, Inc., 7901 Sandy Spring Road, Suite 511, Laurel, MD 20707.

INTRODUCTION

NASA's General Mission Analysis Tool (GMAT) is a mature astrodynamics tool used for mission analysis and design, planning, and operations.[1] The GMAT system contains components used to plan and optimize maneuvers, perform orbit estimation, and generate output used to monitor spacecraft on orbit. The GMAT code base contains all of the elements needed to support these capabilities either from a graphical user interface (GUI) or from a text based console application. GMAT users can add custom capabilities through a plug-in system designed to allow extension of GMAT for new mission needs. Recent improvements to the system extend this plug-in system to include additions to the GMAT GUI.

GMAT's capabilities are fully contained and accessible through the GMAT GUI and console front ends. Users of the system began asking for access to GMAT's tested components from external tools several years ago. In response, the GMAT team prototyped approaches to address these user requests, resulting in a "C-Interface" plugin with extremely limited capabilities and a study of available tools that could be used to make GMAT's components accessible to other systems.[2] The recommendations of the latter study were then used to build a replacement for the C-Interface code in the Orbit Determination Toolbox (ODTBX) project at GSFC, validating the recommended approach.[3] These early efforts to make an Application Programming Interface (API) provided a set of preliminary requirements and potential implementation options for a system API. The API study and subsequent ODTBX experiment identified the Simplified Wrapper and Interface Generator (SWIG)[4] as a toolset that could generate the desired API functions from the GMAT code base.

In late 2018 work on a production GMAT API started with the goals of providing access to most of the core GMAT system for users working in MATLAB and Python, with the eventual goal of making these components available both as a toolkit and as an avenue for interoperability between GMAT and other astrodynamics tools. The R2020a release of GMAT[5] includes this capability in beta form, along with user documentation and sample use cases for the system. In this paper, we describe this work and use existing examples to illustrate current functionality of the GMAT API. We conclude by describing the ongoing development of the system.

Design Goals

GMAT's API user base includes mission analysts working at NASA GSFC and their contractors. These users are familiar with GMAT and have had exposure to the earlier GMAT API prototypes. Given this user experience, the API developers surveyed this user community at the start of the API development process in order to identify the most useful feature set for early builds of the API, and to identify the target API platforms and features. The resulting feature set was broken into four categories: style of usage (identified through use cases), application frameworks used to call the API, ease of use features, and near term needs. The high level requirements identified are summarized below.

- The GMAT API must support the following use cases
    - GMAT API users need to be able to load a script, edit scripted parameters, run the script using the edited data, and retrieve the resulting data.
    - Mission analysts need an easy-to-use toolbox of validated astrodynamics components.

  - Programmers and ”power users” need the ability to interact at a detailed level with objects built from GMAT classes

- The GMAT API can be called from the following tools and application frameworks
  - Python
  - Java
  - MATLAB
  - C++

- API features simplifying ease of use
  - API users need to be able to use the API without detailed knowledge of GMAT code
  - The API must be documented to simplify use
  - The API must have online access to help, including interactive access to the available object settings

- The initial builds of the GMAT API will satisfy the following near term needs
  - The API will provide access to GMAT’s time system, coordinate system, and state representation conversion utilities
  - The API will provide access to GMAT’s dynamics modeling and propagation components
    - ∗ GMAT Dynamics models must be accessible
    - ∗ Propagation should be available for GMAT’s propagators
  - Measurement models must be accessible from GMAT’s Estimation Plugin

The R2020a release of GMAT includes a build of the GMAT API that satisfies all of these requirements. The API included in GMAT R2020a is considered beta quality while testing continues on the system.

API Development

Working from the results of the earlier API exploration and the identified needs of the user community at GSFC, we began implementation of the SWIG based GMAT API in the fourth quarter of 2018. The resulting API supports general purpose exposure of the core GMAT system, demonstrated through a use case that provides the following functions and features:

- Interactive use in Python and MATLAB via Java
- Component access
  - Access to GMAT’s conversion utilities
  - Dynamics modeling
  - Propagation
  - Measurement modeling

- Script manipulation and use
- Interactive help

The goal of the API development is to meet the needs of identified users. In support of this goal, the development team scheduled periodic demonstrations of the system. The core functionality identified above was demonstrated in May 2019 and refined over the course of the year. The system, including more than 80 pages of user documentation and example implementations of the use case features, was ready for release in October 2019 and was included in the next formal GMAT release, GMAT R2020a. The remainder of this paper focuses on the contents of this release.

Funding for the GMAT API is provided by the NASA Engineering and Safety Center as part of an effort to make several NASA tools operate together. Now that the core API feature set is in place, the development team is using the API to help the GMAT system work with other NASA developed tools, including the Monte system from JPL. We provide a brief description of that work as well.

COMPONENT ACCESS

GMAT is built on an object model documented at a high level in its Architectural Specification[6] and in detail in files generated using the Doxygen[7] documentation generation tool. The code is written in standard C++. The wrappers for the GMAT code use SWIG to generate the Python and Java interface files used by the API. Direct low level access to GMAT's objects can be made through these interfaces by experts in the system. That access was used for the early API experiments described above.

One goal of the GMAT API project is to simplify the learning curve for access to GMAT's capabilities. With that goal in mind, the development team added API specific functions to simplify system configuration, and customized select portions of the SWIG wrapper code to make the system easier to use. Table 1 lists some of these functions.

Table 1. API Functions

Function	Purpose
ShowClasses	Lists the classes available for use
ShowObjects	Lists the objects that have been created
Construct	Creates a GMAT object
GetObject	Retrieves a created object
Copy	Makes a copy of an object
Initialize	Prepares the constructed objects for use, and links the objects together
Clear	Deletes a constructed GMAT object
Help	Provides help, either at the global level or for specific objects

The team also added new elements to the GMAT code that simplify user access to GMAT's components, and to the settings on those components.

Running Interactively

GMAT runs scripts in a two-step process. The first step parses a script file and translates the file into a set of individual objects that are stored in an in-memory database of objects and a linked list of actions defining a mission time line. When the user tells GMAT to run the script, the second step

Table 2. GMAT Class Extensions for the API

Method	Purpose
Help	Retrieves help for an object
SetField	Sets a field on an object
GetField	Retrieves the setting for a field as a string
GetNumber	Retrieves the value of a numerical field‘

is applied: the objects from the configuration database are copied into a local sandbox, and the steps defined in the mission time line are applied sequentially to the copied objects.

API users that work with the GMAT components have a different work flow. They construct objects from their native application framework and then manipulate those objects directly. This work flow can be scripted in the framework's language. Two examples of that process – propagation and measurement modeling – are described below. Users can also run the GMAT API in console applications provided by the application framework. This approach lets a user explore GMAT's capabilities interactively and try actions to see what happens while writing a more scripted piece of analysis.

As an example, consider a user that needs the Earth-fixed representation of a mean-of-J2000 state. Using the API, the user can execute the steps shown in Listing 1.

Listing 1. Converting an Inertial State to a Body Fixed State

```
$ python3
>>> from load_gmat import *
>>>
>>> today = gmat.UtcDate(2020, 7, 25, 12, 38, 00.000)
>>> j2000state = gmat.Rvector6(6988.427, 1073.884, 2247.333, 0.019982,
    7.226988, -1.554962)
>>> fixedState = gmat.Rvector6()
>>>
>>> eci = gmat.Construct("CoordinateSystem","ECI","Earth","MJ2000Eq")
>>> ecf = gmat.Construct("CoordinateSystem","ECF","Earth","BodyFixed")
>>> csConverter = gmat.CoordinateConverter()
>>>
>>> gmat.Initialize()
>>>
>>> csConverter.Convert(today.ToA1Mjd(), j2000state, eci, fixedState,
    ecf)
True
>>> print(fixedState)
-3969.46495427452 -5845.748088443061 2261.066486400762
    4.850432360714961 -4.64877091287663 -1.554952194642377
```

This session shows the basic steps a user follows for the conversion in the Python interpreter. After launching the interpreter and loading the GMAT API, the user enters the epoch and inertial state data, and sets up a container for the fixed state. Next they create the coordinate systems and converter necessary for the coordinate transformation. The call to initialize the system connects these objects to underlying GMAT components, including the Earth object needed for the coordinate system computations. Finally, the converter is called to perform the coordinate system conversion, and the result is displayed to the user.

Example: Propagation using Python

The GMAT API provides direct access to GMAT force modeling and propagation for MATLAB and Python users. Working in interactive mode, a user can experiment with force model configuration directly and view the results using the help system. For example, the configuration of a force model to use a 4x4 Earth geopotential can be built interactively, as shown in listing 2.

Listing 2. Force Model Configuration

```
>>> from load_gmat import *
>>> fm = gmat.Construct("ForceModel","FM")
>>>
>>> earthgrav = gmat.Construct("GravityField")
>>> earthgrav.SetField("PotentialFile","JGM2.cof")
>>> earthgrav.SetField("BodyName","Earth")
>>> earthgrav.Help()

GravityField EarthGrav

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

  Degree                       Integer 4
  Order                        Integer 4
  StmLimit                     Integer 100
  PotentialFile               Filename JGM2.cof
  TideFile                    Filename
  TideModel                     String None

>>> fm.AddForce(earthgrav)
```

Once the user has their force model configuration determined, the API calls needed can be placed in a script file that can then be imported for further use. The integrator configuration shown in Listing 3 uses such a configuration, imported from the file BasicFM.py.

In GMAT, a numerical propagator consists of a force model and integrator collected together in a Propagator container. The Propagator collects together all of the pieces needed for propagation: a spacecraft, force model, and integrator, and manages the connections between these pieces. Then when the components are initialized, the links between these pieces are set. For propagation, a further initialization is performed using the PrepareInternals method, which sets up internal structures and the state vector used in propagation. The listing shows the full configuration needed.

Listing 3. Integrator Configuration and Use

```
# Load the force model used for the propagation
from BasicFM import *

# Spacecraft used for the propagation
earthorb = gmat.Construct("Spacecraft", "EarthOrbiter")
```

```
# Build the propagation container
pdprop = gmat.Construct("Propagator","PDProp")

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

# Assign the integrator and force model imported from BasicFM
pdprop.SetReference(gator)
pdprop.SetReference(fm)

# Perform top level initialization
gmat.Initialize()

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

# Refresh the integrator reference and take a step
gator = pdprop.GetPropagator()
gator.Step(60)
```

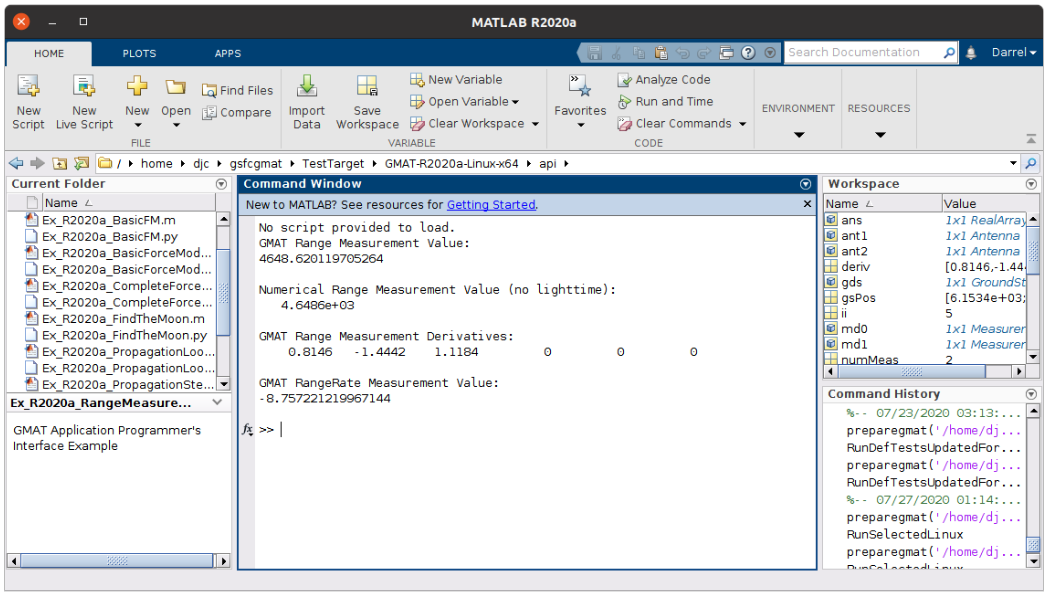
The code shown here is part of the GMAT R2020a release. The force model configuration is in the Ex_R2020a_BasicFM.py file in the api folder, and the propagation is demonstrated in Ex_R2020a_PropagationStep.py. Matching MATLAB .m files are also included in the release package.

Example: Measurement Modeling in MATLAB

Part of GMAT's capabilities as an orbit determination tool includes the ability to simulate measurements. The types of measurements GMAT supports, as of R2020a, include range, range rate, and angle types. GMAT also provides the ability to add various corrections to the simulated measurement values, including corrections due to light-time delays, ionospheric and tropospheric delays, and relativistic effects, among others. The ability to simulate measurements with or without these corrections is exposed in the API.

Ex_R2020a_RangeMeasurement.m is an example MATLAB API script contained in the R2020a release where range and range rate measurements are configured completely through the GMAT API, then are calculated for a ground station station and spacecraft pair. The range measurement obtained from GMAT is then validated with the two way range calculated in MATLAB from the position vectors of the ground station and the spacecraft. The script also provides the syntax for adding either ionosphere or troposphere delay modeling on the ground station, or corrections for light-time delay and the effects of general relativity on the signal, though these are either commented out or not enabled by default in the example script. Figure 1 shows this example running in MATLAB.

While the above mentioned example constructs all the resources using the API, it is also possible to instead configure some or all of the resources in a GMAT script. If set up that way, the GMAT script is loaded and initialized through the API, which allows an API user to use a preexisting script they may already have that they want to reuse through the API.



Current Folder

```
Ex_R2020a_BasicFM.m
Ex_R2020a_BasicFM.py
Ex_R2020a_BasicForceMod...
Ex_R2020a_BasicForceMod...
Ex_R2020a_CompleteForce...
Ex_R2020a_CompleteForce...
Ex_R2020a_FindTheMoon.m
Ex_R2020a_FindTheMoon.py
Ex_R2020a_PropagationLoo...
Ex_R2020a_PropagationLoo...
Ex_R2020a_PropagationSte...
```

Ex_R2020a_RangeMeasure...

GMAT Application Programmer's Interface Example

Command Window

```
No script provided to load.
GMAT Range Measurement Value:
4648.620119705264

Numerical Range Measurement Value (no lighttime):
   4.6486e+03

GMAT Range Measurement Derivatives:
    0.8146   -1.4442    1.1184         0         0         0

GMAT RangeRate Measurement Value:
-8.757221219967144

>> |
```

Workspace

Name	Value
ans	1x1 RealArray
ant1	1x1 Antenna
ant2	1x1 Antenna
deriv	[0.8146,-1.44
gds	1x1 GroundSt
gsPos	[6.1534e+03;
ii	5
md0	1x1 Measurer
md1	1x1 Measurer
numMeas	2

Command History

```
%-- 07/23/2020 03:13:...
preparegmat('/home/dj...
RunDefTestsUpdatedFor...
preparegmat('/home/dj...
RunDefTestsUpdatedFor...
%-- 07/27/2020 01:14:...
preparegmat('/home/dj...
RunSelectedLinux
preparegmat('/home/dj...
```

Figure 1. Calculating Range in MATLAB

GMAT SCRIPTS AND THE API

GMAT is a script based application. Users configure the system using a custom scripting language, or using panels on GMAT's graphical user interface that interact with underlying objects through GMAT's script interfaces. When GMAT loads a script, the application builds a database of objects in memory that match the scripted settings. Running the script takes these pristine objects, makes copies of them, and then performs operations on these copies, generating output files and other analysis data based on the instructions in the user's script. The GMAT application tracks progress through the system in a log file that records the status of the run as it progresses.

Users familiar with GMAT's scripting language requested that the API provide the ability to load GMAT scripts, change input settings, and run the script with the changed values. The script manipulation capabilities are provided through a few API specific commands, along with the object access capabilities described above. The GMAT log file location is also provided through an API call.

A user that wants to run a script in the API and then view the results needs to access the object used in the run rather than the original object built from the script. The API provides a function specific to that feature of GMAT runs using another API function, GetRuntimeObject().

The script manipulation capabilities are best illustrated through a few examples. The Python listing below shows a very basic application of the script access functions in the GMAT API.

Listing 4. Running a GMAT Script using the API

```python
# Import the GMAT system
import gmatpy as gmat

# Load a script into the GMAT API and run it
gmat.LoadScript("ToLuna.script")
gmat.RunScript()

# Retrieve and display the results of the run
MoonDistance = gmat.GetRuntimeObject("MoonDistance")
print (MoonDistance.GetRealParameter("Value"))
```

The script identified here, “ToLuna.script,” is a basic launch-coast-burn script that starts with a spacecraft in a low-Earth parking orbit from a launch vehicle. The spacecraft propagates (coasts) for a set time, applies an impulsive maneuver (burn), and then propagates to one of three conditions: apogee, perilune, or eight days of propagation. The first stopping condition encountered stops the propagation.

This Python scripting performs the same function as running the script in GMAT, and then writes the resulting spacecraft-Moon separation distance to the terminal running the script:

```
$ python3 RunToLuna.py
228234.63471634657
```

This example shows the basic approach to running GMAT scripts using the API.

The API’s script driving capabilities enable a variety of run time capabilities ranging from parameter scans through Monte-Carlo analysis.

Example: A Parameter Scan

The scripting shown above provides the basic structure used to seek a close approach to the Moon by varying the parking orbit insertion epoch, coast time and burn delta-V. All of these actions are straightforward using the API. The script file includes the settings

Listing 5. Script Settings for the Lunar Encounter Search

```
...
Create ImpulsiveBurn TOI;
...
GMAT TOI.Element1 = 3.1;
...
Create Variable StartEpoch LeoTime;
...
StartEpoch = StartEpoch + Sat.A1ModJulian;
Sat.A1ModJulian = StartEpoch;
...
Propagate DefaultProp(Sat) {Sat.ElapsedSecs = LeoTime};
Maneuver TOI(Sat);
Propagate DefaultProp(Sat) {Sat.Earth.Apoapsis, Sat.Luna.Periapsis,
    Sat.ElapsedDays = 8};
```

The three variables for the scan are the launch insertion state epoch, controlled with the StartEpoch variable, the coast time, controlled using the LeoTime variable, and the maneuver delta-V, set with the Element1 field on the TOI maneuver object. The lunar encounter distance is evaluated at the end of the last propagation. The API can be used to scan through various values of these three settings, and the best case values stored for display and later use. The pertinent portion of the API driving code for this scan is shown in listing 6.

Listing 6. The Lunar Encounter Search in Python

```
burn = gmat.GetObject("TOI")
Time = gmat.GetObject("LeoTime")
start = gmat.GetObject("StartEpoch")

for i in range(10) :
  for j in range(20) :
    for k in range(25) :
      StartEpoch = k / 2.0
      start.SetRealParameter("Value", StartEpoch / 24.0)
      Time.SetRealParameter("Value", 1500.0 + i * 100.0)
      burn.SetRealParameter("Element1", 3.0 + j * 0.01)
```

The full parameter scan uses the settings in this listing to scan through the three search parameters, reporting the resulting parameter values and lunar encounter distance whenever the encounter distance is smaller than the previous closest approach value. Once the scan has completed, the script writes the results and generates a GMAT script that, when run, has the best case solution in the script. The output from the run (abbreviated for the purposes of this paper) shows the scans as they find the closest lunar approach:

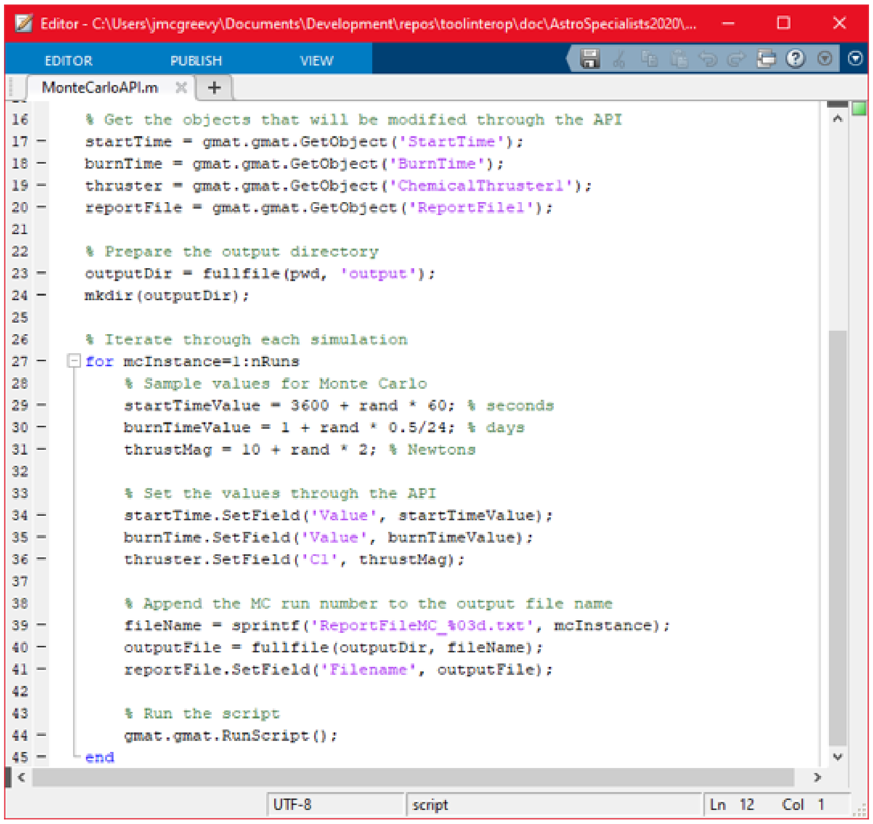
```
$ python3 Ex_R2020a_FindTheMoon.py

Launch 0.0 Coast: 1500.0 Delta-V: 3.0 Moon Dist: 280079.277
Launch 0.02083 Coast: 1500.0 Delta-V: 3.0 Moon Dist: 277059.147
Launch 0.04166 Coast: 1500.0 Delta-V: 3.0 Moon Dist: 276559.174
Launch 0.0 Coast: 1500.0 Delta-V: 3.01 Moon Dist: 275932.283
...
Launch 0.125 Coast: 1800.0 Delta-V: 3.17 Moon Dist: 28176.146
Launch 0.14583 Coast: 1800.0 Delta-V: 3.17 Moon Dist: 23557.035
Launch 0.125 Coast: 1800.0 Delta-V: 3.18 Moon Dist: 18796.546
Launch 0.14583 Coast: 1900.0 Delta-V: 3.17 Moon Dist: 15522.861
Launch 0.16666 Coast: 1900.0 Delta-V: 3.17 Moon Dist: 6363.771

Saving solution to Ex_R2020a_ToLuna_solution.script
```

The GMAT R2020a release package includes the GMAT script and drivers described here, in the api folder of the build. The script is the file Ex_R2020a_ToLuna.script, the Python driver for the parametric scan is Ex_R2020a_FindTheMoon.py. The release also includes a matching MATLAB driver, Ex_R2020a_FindTheMoon.m.

Example: Monte-Carlo Analysis

The API can also be used to perform a Monte-Carlo analysis based on an existing GMAT script. After first using the API to load a GMAT script into GMAT's sandbox, the API can then be used to modify specific values in the sandbox according to the desired probability distribution(s), and then the API can run the modified mission in GMAT. Because the modified mission is still being executed by GMAT, the API is still running a fully-featured GMAT mission, including any targeters, optimizers, and reporting tools in the script, including resources and commands that have not yet been exposed through the API.

MonteCarloAPI.m[8] is an example in MATLAB of the GMAT API being used to perform a Monte-Carlo simulation on a single finite burn in low earth orbit. The parameters that are varied in this example are the initial epoch and duration of the burn, along with its magnitude. First, a GMAT script, MonteCarloAPI.script, with the nominal maneuver is configured. This GMAT script prepares the spacecraft, the force model and propagator, the maneuver properties, and the output report file. The nominal mission sequence is also defined in the script, consisting of propagation to the start of the maneuver, during the maneuver, and for a period of time after the maneuver.

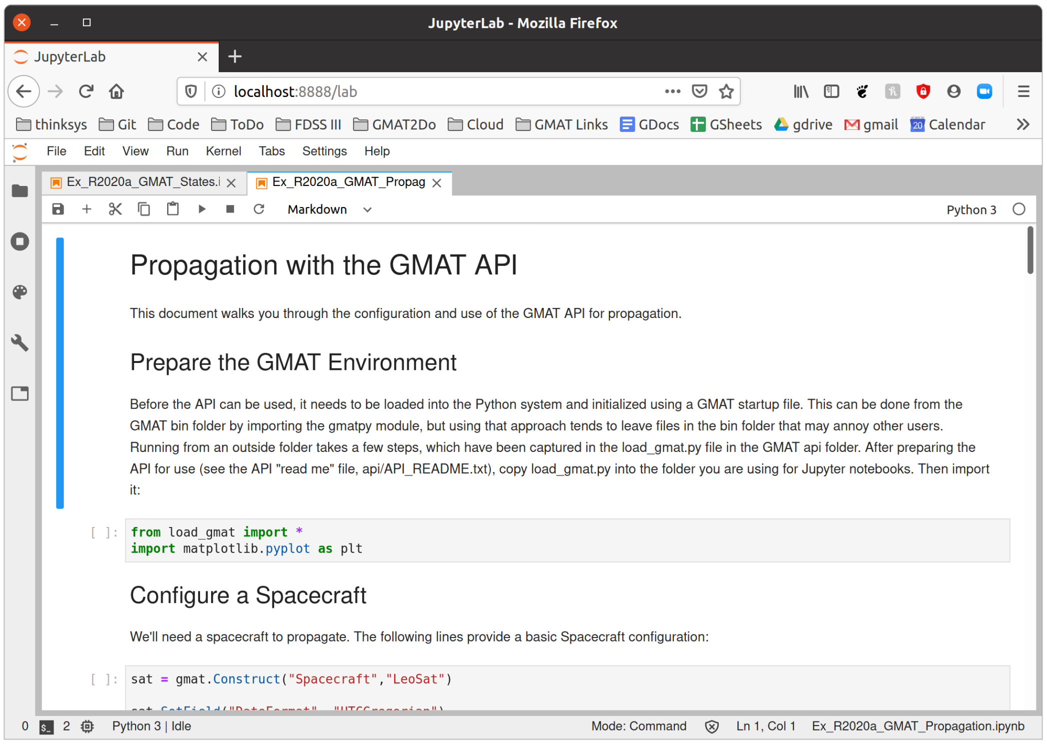

Editor - C:\Users\jmcgreevy\Documents\Development\repos\toolinterop\doc\AstroSpecialists2020\...
EDITOR
PUBLISH
VIEW
MonteCarloAPI.m
UTF-8
script
Ln 12
Col 1

```
% Get the objects that will be modified through the API
startTime = gmat.gmat.GetObject('StartTime');
burnTime = gmat.gmat.GetObject('BurnTime');
thruster = gmat.gmat.GetObject('ChemicalThruster1');
reportFile = gmat.gmat.GetObject('ReportFile1');

% Prepare the output directory
outputDir = fullfile(pwd, 'output');
mkdir(outputDir);

% Iterate through each simulation
for mcInstance=1:nRuns
    % Sample values for Monte Carlo
    startTimeValue = 3600 + rand * 60; % seconds
    burnTimeValue = 1 + rand * 0.5/24; % days
    thrustMag = 10 + rand * 2; % Newtons

    % Set the values through the API
    startTime.SetField('Value', startTimeValue);
    burnTime.SetField('Value', burnTimeValue);
    thruster.SetField('C1', thrustMag);

    % Append the MC run number to the output file name
    fileName = sprintf('ReportFileMC_%03d.txt', mcInstance);
    outputFile = fullfile(outputDir, fileName);
    reportFile.SetField('Filename', outputFile);

    % Run the script
    gmat.gmat.RunScript();
end
```

Figure 2. MATLAB Script Using the GMAT API for Monte-Carlo Analysis

Through the API, the above mentioned parameters for the maneuver are modified. For each iteration, the start time and duration of the burn are set through GMAT variables named Start-Time and BurnTime, respectively, and the thrust magnitude is set through the ’C1’ field on Chem-icalThruster1. The values used to set these fields come from random sampling using MATLAB’s random number generator. The file name the ReportFile writes to is also set through the API, using a uniquely numbered file name for each iteration. Finally, the modified script is run through the RunMission() function, where GMAT executes the script with the values for the current iteration, and the output files are generated according to the GMAT script. Figure 2 shows the portion of the MATLAB script responsible for setting these values and running the GMAT script through the API. The output files can them be analyzed after the run is complete, either with another utility, or loaded back into MATLAB, and analyzed in the same script that operated the GMAT API.

TUTORIAL DEVELOPMENT


JupyterLab - Mozilla Firefox
JupyterLab
localhost:8888/lab
thinksys Git Code ToDo FDSS III GMAT2Do Cloud GMAT Links GDocs GSheets gdrive gmail Calendar
File Edit View Run Kernel Tabs Settings Help
Ex_R2020a_GMAT_States.i
Ex_R2020a_GMAT_Propag
Markdown
Python 3

Propagation with the GMAT API

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

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 files in the bin folder that may annoy other users. Running from an outside folder takes a few steps, which have been captured in the load_gmat.py file in the GMAT api folder. After preparing the API for use (see the API "read me" file, api/API_README.txt), copy load_gmat.py into the folder you are using for Jupyter notebooks. Then import it:

```
[ ]: from load_gmat import *
     import matplotlib.pyplot as plt
```

Configure a Spacecraft

We'll need a spacecraft to propagate. The following lines provide a basic Spacecraft configuration:

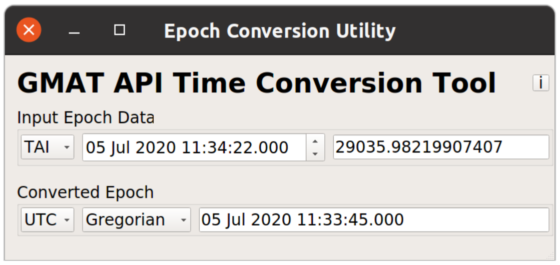
```
[ ]: sat = gmat.Construct("Spacecraft","LeoSat")
```

0 2 Python 3 | Idle Mode: Command Ln 1, Col 1 Ex_R2020a_GMAT_Propagation.ipynb

Figure 3. A Jupyter Notebook Using the GMAT API

The GMAT API lends itself to the creation of tutorials and walkthrough demonstrations using the Python Jupyter notebook system. Figure 3 shows one of the notebooks included in the GMAT release. This notebook lets a user walk through the process of configuring and running the propagation setup described earlier. A second notebook in the release shows how states are managed in GMAT, a topic that confused some users.

GUI TOOL DEVELOPMENT



Input Epoch Data

TAI | 05 Jul 2020 11:34:22.000 | 29035.98219907407

Converted Epoch

UTC | Gregorian | 05 Jul 2020 11:33:45.000

Figure 4. Time Conversions using PyQt

The spacecraft panel in the GMAT GUI provides a framework accessing GMAT capabilities for conversions and data manipulation, but the process of using that capability is somewhat cumbersome, requiring that a user launch GMAT and understand how to access those features of the system. The API provides a mechanism for producing a more lightweight, customized user interface into GMAT's code. One example is shown in Figure 4.

The time conversion GUI[9] shown in the figure is written in PyQt. It was built to facilitate comparisons of epochs for some testing performed by a GMAT developer that needed a quick way to examine reported epochs.

INTEROPERABILITY WITH ODTBX

The GMAT API can also be used in combination with other software tools. The Orbit Determination Toolbox (ODTBX) is another open source software tool developed by NASA Goddard to perform orbit determination for early mission analysis.[10] ODTBX is primarily written in MATLAB and also has Java components, making it well suited to use GMAT's MATLAB and Java API.

The estimators in ODTBX call into user provided functions that contain dynamics and measurement models used in ODTBX's estimation algorithm. The function containing the dynamics model, called a dynfun, provides the state derivatives, and optionally provides the Jacobian and process noise spectral density matrices, while the function containing the measurement model, called a datfun, provides the measurement values and noise covariance, and optionally provides the measurement partial derivatives. If the partial derivatives are not provided by either function, ODTBX numerically calculates them. An ODTBX user can write these functions as wrapper functions which call into the GMAT API to use GMAT's dynamics and measurement models to calculate the state derivatives, measurement values, and their partial derivatives. The wrapper functions then output the results from GMAT in the format the ODTBX estimators expect. All this allows ODTBX users both to easily use the high-fidelity dynamics and measurement models that GMAT provides through ODTBX, and to use the same GMAT script that was used in ODTBX through the GMAT application for future mission analysis. Using the GMAT API also can provide performance improvements as the GMAT models are run in native code, which is faster than the equivalent code in Java or MATLAB.

Listing 7 shows an example ODTBX dynfun which calls into the GMAT API. A GMAT API ODEModel object is passed to the function as an additional input argument, fm, which contains the GMAT forces to be evaluated. This ODEModel is configured via API calls in the MATLAB script which calls the ODTBX estimation routine, and because the dynfun function uses the GMAT API to also calculate the Jacobian, the ODEModel needs to be configured to compute that matrix as well. The GetDerivatives() function is where the actual call to the API to compute the derivatives, and the remainder of the dynfun is rearranging the GMAT API output into the format that ODTBX expects, and populating a process noise spectral density matrix.

Listing 7. Example ODTBX dynfun for GMAT

```
function [xDot, A, Q] = gmat_dynfun(t, x, fm)

nt = numel(t);

xDot = zeros(6, nt);
A = zeros(6, 6, nt);

for tIndex=1:nt
    % Get derivatives
    fm.GetDerivatives(x(:,tIndex), t(tIndex), 1); % Calculate derivatives
    deriv = fm.GetDerivativeArray(); % Get calculated derivatives

    xDot(:,tIndex) = deriv(1:6);

    % Reshape the vector of state derivatives into the A matrix
    % Need to skip the first 6 elements as they do not contain Jacobian
       data
    A(:,:,tIndex) = reshape(deriv(7:42), 6, 6)';
end

% Populate process noise spectral density matrix
Q = repmat(diag([0 0 0 1e-9 1e-9 1e-9].^2), 1, 1, nt);
```

SYSTEM STATUS AND FUTURE DEVELOPMENT

The GMAT API included in the GMAT R2020a release is a beta delivery of the system. The release is beta because it is still undergoing refinement and testing. These steps are expected to be completed for the GMAT R2021a release. Interested users are encouraged to download the build and experiment with it, and to provide suggestions and feedback to the authors to help improve the system.

The GMAT API development team is working to use GMAT and other NASA tools together. Data sharing between GMAT and JPL's Monte system has been demonstrated. We are working now on sharing dynamics models between the systems as a step towards a consistent suite of tools across NASA.

ACKNOWLEDGMENTS

The GMAT Application Programming Interface development work is funded through the NASA Engineering and Safety Center. Goddard Space Flight Center manages the work. The authors are grateful for the support of these organizations.

REFERENCES

[1] The GMAT Development Team, *GMAT Wiki*, 2020. https://gmat.atlassian.net/wiki.
[2] D. J. Conway, "GMAT API Tradeoff Study," 2012.
[3] D. J. Conway, "GMAT API Consultation Support," 2016.
[4] The SWIG Development Team, *SWIG-4.0 Documentation*, 2019. http://www.swig.org/.
[5] The GMAT Development Team, *GMAT: The General Mission Analysis Tool*, 2020. https://sourceforge.net/projects/gmat/.
[6] The GMAT Development Team, *General Mission Analysis Tool (GMAT) Architectural Specification*, 2020.
[7] D. van Heesch, *Doxygen*, 2020. https://www.doxygen.nl/.
[8] J. McGreevy, "GMAT API Monte Carlo Example," Available upon request, 2020.
[9] D. J. Conway, "GMAT API Time Conversion Tool," Available for download: www.thinksysinc.com/gmatapi/EpochConverter.zip, 2020.
[10] The ODTBX Development Team, *ODTBX: The Orbit Determination Toolbox*, 2020. https://sourceforge.net/projects/odtbx/.