MODELSPACE USER'S GUIDE

VERSION 25.06

Table of Contents

Introduction

Hello, and thank you for choosing ModelSpace to ensure your project's success. This welcome packet will walk you through the first steps to set up and begin using your ModelSpace environment.

If you have any questions, please utilize the included support resources. Any question that is not answered by the documentation or Stack Overflow will happily be answered at help@attxengineering.com.

Our ATTX team believes our work begins when you start using ModelSpace, and we want to be with you every step of the way to help your mission succeed. We are looking forward to working with your team!

Support Resources Available

ATTX Assist

ATTX has provided a ChatGPT-based AI chatbot which is fully trained on ModelSpace. The chatbot, called ATTX Assist, is so knowledgeable that we use it internally for our own software development. It is a great resource for any questions you may have! The more specific your question, the better the response you will receive.

ATTX Assist is trained on the ATTX code base as well as this User's Guide and ATTX Stack Overflow questions and answers so it is constantly growing in knowledge and support capability.

ATTX Stack Overflow

The ATTX Stack Overflow is the single most useful resource for answering questions and receiving help. It is actively monitored by ATTX engineers for quick responses. It is also shared across the ATTX community, which helps to build a collective knowledge base. Checking to see if your question has already been asked is the recommended first step for any issues. If it hasn't, we request that you strip proprietary information from your problem and post it there. That way, it may be useful to other engineers who encounter the same question.
https://stackoverflowteams.com/c/attx/questions

NOTE: The ATTX stack overflow is shared by the ATTX community. It is your team's responsibility to ensure you do not post ANY proprietary information or data to the

ATTX Stack Overflow. Doing so could potentially expose your information to other ATTX partners. All proprietary information should be stripped from questions prior to posting on the ATTX Stack Overflow.

Documentation

The ATTX documentation is an extensive source of information on how to use ModelSpace. It is designed to be a handy reference for both new and experienced developers. The docs contain everything from low-level code documentation, to high-level descriptions of utilities and models, to tutorials that walk your team through setting up, building, and performing analysis in ModelSpace scripts

.

While this User's guide is provided as an all-in-one source to help users succeed with ModelSpace, the Doxygen documentation on the ATTX website should be viewed as the ultimate source of authority on interacting with ModelSpace. In each section, the User's Guide has links to the specific element of documentation associated with that element. The overall documentation for ModelSpace can be found at this link.

The ATTX Help Email Address

ATTX also provides a help email address, which can be used anytime. Simply email help@attxengineering.com and the ATTX team will help solve your problem in any way we can. Of course, feel free to email individual engineers on the ATTX team directly as well. We sincerely want to see your project succeed. Please do not hesitate to ask for help any time you need it.

Installing ModelSpace

This section of the User's guide walks users through the ModelSpace installation process. ModelSpace runs natively in Linux, but is friendly to Windows and Mac via WSL, Docker, and virtual machines. Instructions on how to install each are included here.

Running on Windows with WSL (Recommended)

The easiest and most straightforward way to run ModelSpace on Windows is using the Windows Subsystem for Linux, or WSL. WSL is a great solution as a compromise between creating a software-friendly environment (Linux) and using ModelSpace seamlessly within your team's OS.

ATTX has configured a script to automatically configure and build WSL for the ModelSpace environment. The script can be downloaded from the modelspace release folder, and can be run using Windows PowerShell using administrator permission. Steps are as follows:

1. Download the entire modelspace_XX.YY folder containing a single release from Box.
2. Open a Windows PowerShell instance by right clicking and selecting “Run as Administrator”
3. Change into the modelspace directory on your windows computer by running cd <path_to_your_directory>
   Run the script ms_windows.ps1 in PowerShell using the following command:

```None
powershell -ExecutionPolicy Bypass -File ms_windows.ps1
```

4. If WSL and Ubuntu 22.04 are already installed, the script will install ModelSpace on your system. **If they are not installed, the script will install them and exit – to run ModelSpace, you will need to re-run the install script.**
5. Once install is complete, the script will generate a desktop icon from which the ModelSpace GUI can be run. Scripts can be coded in VSCode by connecting to WSL, as detailed in the following steps. If just using the GUI, users can stop here.

Supporting Installs

While not strictly necessary, ATTX highly recommends using the Visual Studio Code development environment for ModelSpace work. VSCode can be found/installed here: https://code.visualstudio.com/

Connecting to WSL

Once WSL is verified, the next step is to establish a basic working environment from which to use ModelSpace.

1. Open VSCode on your Windows computer. Press F1 to bring up the VSCode command window
2. Run the command WSL: Connect to WSL
3. If the command does not work, go to the extensions window and install the WSL extension for VSCode

4. Once connected, open a terminal in VSCode by clicking Terminal->New Terminal in the top toolbar, or by hitting ctrl + shift + `. This terminal will operate exactly as an Ubuntu 22.04 OS without using Docker

Installing in Native Linux Environments

On computers running Linux natively, which includes Linux Docker containers and Virtual Machines, ModelSpace can be installed easily via a prepared install script. To install, only the following steps are necessary:

1. Download the file ms_linux.zip from the latest release folder on the ATTX Box, which is where all files are posted to ATTX partners
2. Run the following commands to install ModelSpace on your system

```
None
unzip ms_linux.zip
cd modelspace/
./install.sh
```

ModelSpace should now be installed on your system. It can be verified by following the "Verifying Your Installation" subsection.

Installing on Mac

Because Mac does not have WSL or native support for ModelSpace, users have two options:

1. Install ModelSpace using a virtual machine (supported, but less common. Instructions are not provided by default for this case)
2. Install ModelSpace via Docker devcontainer on Mac. This instruction set covers how to do so.

Because these installation instructions rely on Docker, users should first follow the instructions on installing Docker for Mac. Users should also first install Visual Studio Code for Mac.

Once those software packages are installed, open Visual Studio. Go to the "Extensions" tab on the left and install the Dev Containers extension.

Clone the ATTX “modelspace custom” git repository and open the repository folder in Visual Studio Code.

Download the “ms_mac.zip” from the ModelSpace Box and place it in the modelspace custom repository location.

Now close out of Visual Studio Code and re-open. Visual Studio will prompt whether you want to re-open the repository in a development container. Select yes, and ModelSpace will automatically install for you. It may take a few minutes.

ModelSpace is now installed in your devcontainer instance! The gui may be run from anywhere in the devcontainer environment by running “ms-gui” from the terminal.

Visuals can be accessed in the devcontainer instance using noVNC. To access, in a browser window, type http://localhost:6080/ into the address bar. A window should come up that has a large button in the center that says "connect." Click the button, and type password "attx" into the prompt. A black screen will pop up -- that's good. When a script with visuals is running, the visuals will appear there.

As always, ATTX is here to support – if you run into any issues with the install, we will assist and ensure your environment is working.

Verifying Your Installation

1. At this point, ModelSpace is installed on your system and can be run from anywhere. Now let's test it. Run the command "code script.py" to open an empty vscode window and put the following code in it (alternatively, you can copy an example script from the scripts posted with the examples/ folder in the release):

```python
import sys
from modelspace.ModelSpacePy import SimulationExecutive,
CartesianVector3
from modelspace.SpicePlanet import SpicePlanet
from modelspace.Spacecraft import Spacecraft

exc = SimulationExecutive()
```

```
exc.parseArgs(sys.argv)

earth = SpicePlanet(exc, "earth")
sc = Spacecraft(exc, "sc")
sc.params.planet_ptr(earth)

exc.startup()

sc.initializePositionVelocity(CartesianVector3([7000000,0,0
]), CartesianVector3([0, 7000, 10]))

exc.run()
```

2. Run the script -- it should execute and show successful completion!

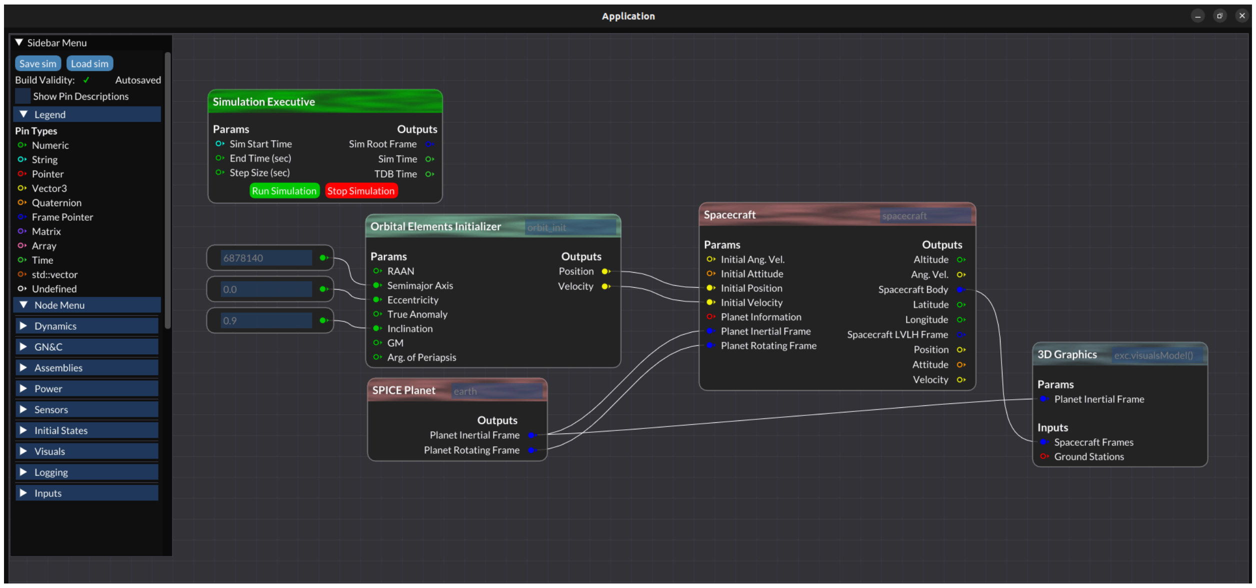
Congratulations! You are now ready to start building your own simulations using ModelSpace. Remember that ATTX is with you every step of the way to help. Please, do not hesitate to contact us.

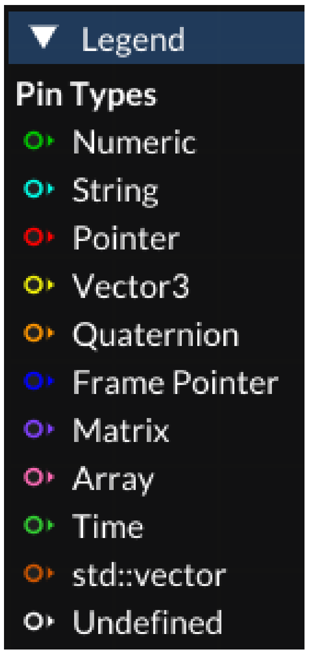
For users looking to explore ModelSpace via example scripts, a number of example scripts demonstrating various scenarios are made available on the ModelSpace Box.

Using the ModelSpace GUI

The application window below shows the default view you load into with the ModelSpace GUI. The GUI utilizes logic based flow diagrams and is designed to be an innovative method for users to get started with building ModelSpace simulations. To run the ModelSpace GUI from your working directory, run the command below from a Terminal:

```Shell
ms-gui
```



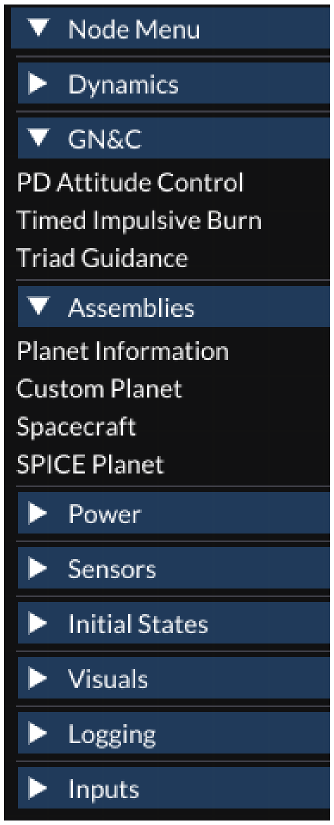


The GUI has two main areas. The Sidebar Menu hosts the Legend for connectors as well as grouping menus for all available models. The checkered field hosts the models actually being used by the simulation you're building. The top of the Sidebar Menu contains the ability to Save and Load Simulations from examples or previously saved work. It also contains the Build Validity indicator (a green check if the simulation is valid, and a red X when it's invalid), as well as a toggle to enable pin descriptors.

The Legend defines the type of data that is expected at various connectors within the simulation:

- Numeric: Can be integers, doubles, floats, etc.
- String: Text input, no need for quote literals [""]
- Pointer: Connect similar data type to pass info (i.e. Ground Station position to Ground Station visuals)
- Vector3: Utilizes a 3 unit Cartesian Vector [xyz]

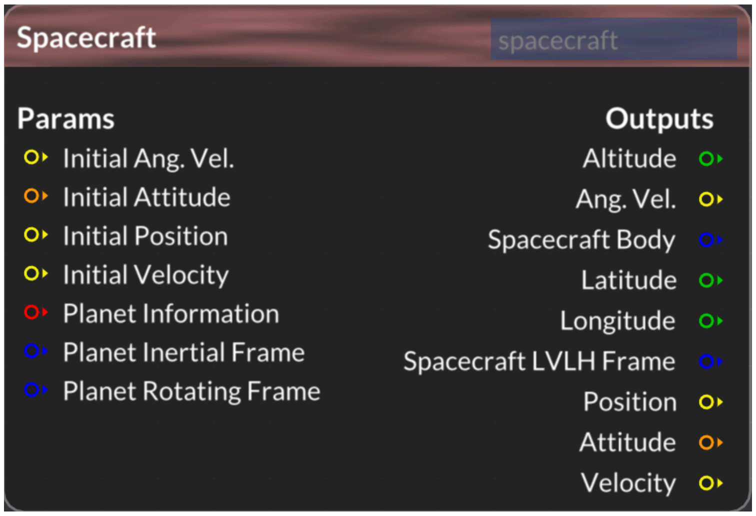
- Quaternion: Utilizes a 4 unit Quaternion [wxyz]
- Frame Pointer: Connects similar assemblies (e.g. Planets)
- Matrix: Utilizes a 9 unit Matrix [3 by 3]
- Time: Essentially equivalent to numeric input
- Undefined: A catch-all category for connecting data from different models (e.g. Logging various data types for data analysis)



The Node Menu groups together similar model types to simplify the ease-of-use of the GUI:

- Dynamics: Utilizes ATTX's pre-built library of models (e.g. Aero. Drag, Gravity Gradient, Solar Radiation Pressure, etc.)
- GN&C: Simple guidance, navigation, and control models. As you build more complex GN&C, you'll find yourself either building and utilizing your own custom models or running the simulation entirely from script.
- Assemblies: Important models necessary for a valid simulation (e.g. Spacecraft, Planets, etc.)
- Power: Models that process the power input/output of a spacecraft (e.g. solar panels, batteries, etc.)
- Sensors: Commonly utilized components (e.g. IMU, Gyro, Star Trackers, etc.)
- Initial States: An initializer for Orbital Elements
- Visuals: Both LivePlot and high-fidelity 3D Graphics
- Logging: A logger that outputs connected data to .csv
- Inputs: User-generated inputs for the simulation

Adding Models

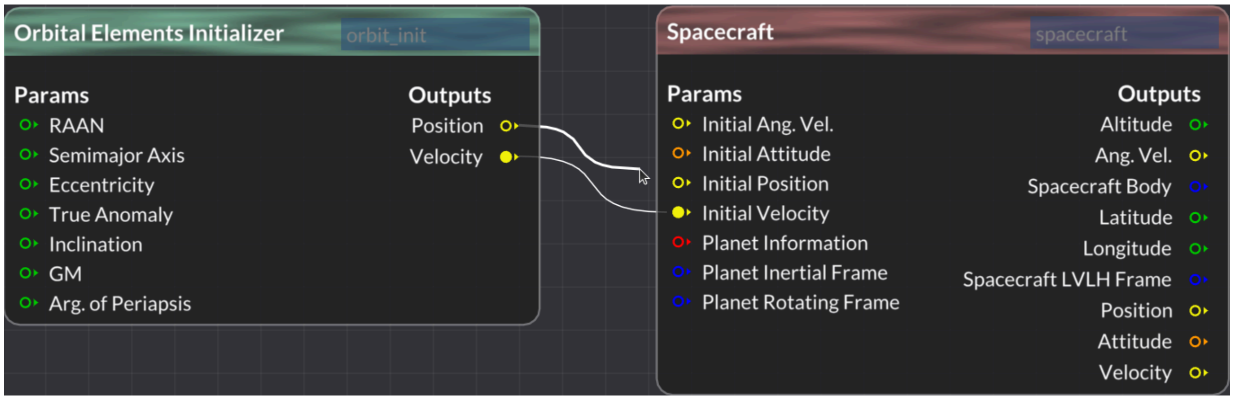


Left-click on any model to load it into the simulation. On the left side of the model block are Parameters / Inputs and the right side has Outputs. The pin type and the corresponding text help the user understand what type of data it would like to have passed into the model. These fields start with an open pin and have a value assigned by default, however when the user connects a pin to the model, the pin type closes and the user input overwrites the default values. The text box at the top right enables you to rename this model for easily distinguishing between similar models within a simulation (e.g. Spacecraft1 and Spacecraft2).

Connecting Models

To connect 2 models, just click on the open pin next to the variable of the information you're trying to pass and drag over to the intended models open connector pin, as shown in the image below. This will link those two models together within the simulation and pass along the intended information. The GUI will not enable you to connect 2 connectors of inconsistent pin types.

Note: Most Input/Parameter connectors are only intended to be connected from one other model block. However, the Outputs of a model can connect to multiple other models. The key models that vary from this rule are logging and visuals which are designed to accept multiple inputs.



Simulation Must-Haves

For a simple simulation to be valid, the user must populate and connect these models:

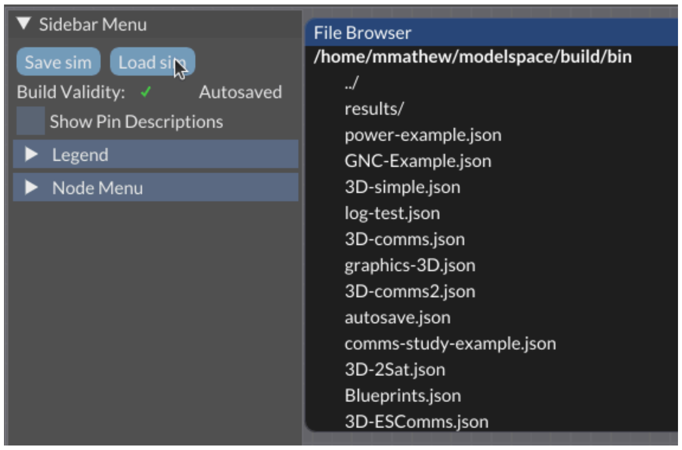
- Initial Sates/ Orbital Elements Initializer, with a minimum of Semimajor axis, Eccentricity, and Inclination defined through 3 connected Input/ Numeric Input
- Assemblies/ Planet, the simplest to utilize being SPICE Planet
- Assemblies/ Spacecraft, with parameters connected from the Orbital Elements Initializer and the Planet Frames
- Visuals/ 3D Graphics, with the Planet and Spacecraft Frames connected

Building and Verifying Simulations

Once a simulation is built, the Sidebar Menu on the left (pictured below) will inform the user if the build is valid or not. If the green check mark is shown, go ahead and run the simulation! If it's a red X, something is incorrect within your simulation and please ensure that the simulation is valid. For a build to be valid, please verify that:

- No loose models are visible that are not connected to any other models
- Model parameters are correctly initialized with correct input types

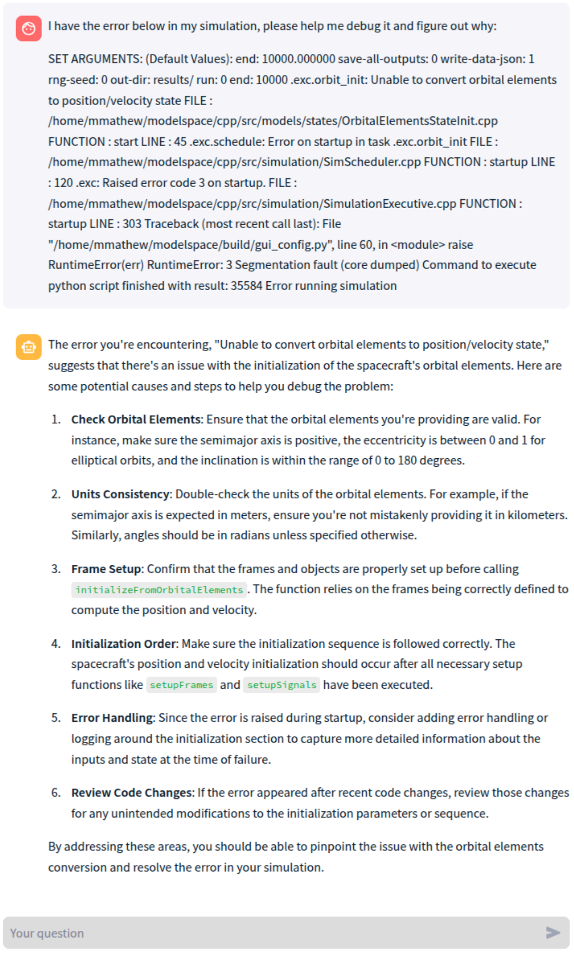
Saving and Loading Simulations



The Sidebar Menu enables the user to load in previously developed simulations and make modifications to save with ease. When a simulation is saved, it writes a .JSON file which can be easily shared with and used by other users. ModelSpace also continuously saves your work to a file named autosave.json, please do not overwrite this file.

Debugging Invalid Sims

If you have a simulation that has invalid build validity and you're unsure how to resolve it, never fear! ATTX Assist is our AI-powered debugging tool that is perfect for situations like this. Just go into your VS Code terminal and copy (Shift + Ctrl + C) the error log and ask the tool to help you resolve it. In the example on the left and below, I had an invalid orbital elements initializer and ATTX Assist was able to help me resolve my issues with ease.



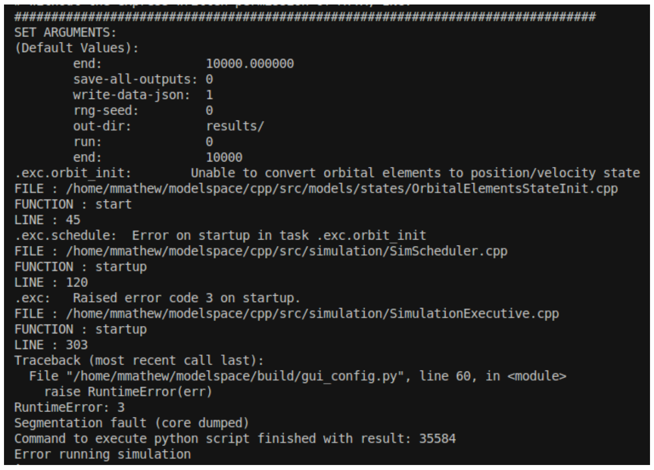
I have the error below in my simulation, please help me debug it and figure out why:

SET ARGUMENTS: (Default Values): end: 10000.000000 save-all-outputs: 0 write-data-json: 1 rng-seed: 0 out-dir: results/ run: 0 end: 10000 .exc.orbit_init: Unable to convert orbital elements to position/velocity state FILE : /home/mmathew/modelspace/cpp/src/models/states/OrbitalElementsStateInit.cpp FUNCTION : start LINE : 45 .exc.schedule: Error on startup in task .exc.orbit_init FILE : /home/mmathew/modelspace/cpp/src/simulation/SimScheduler.cpp FUNCTION : startup LINE : 120 .exc: Raised error code 3 on startup. FILE : /home/mmathew/modelspace/cpp/src/simulation/SimulationExecutive.cpp FUNCTION : startup LINE : 303 Traceback (most recent call last): File "/home/mmathew/modelspace/build/gui_config.py", line 60, in <module> raise RuntimeError(err) RuntimeError: 3 Segmentation fault (core dumped) Command to execute python script finished with result: 35584 Error running simulation

The error you're encountering, "Unable to convert orbital elements to position/velocity state," suggests that there's an issue with the initialization of the spacecraft's orbital elements. Here are some potential causes and steps to help you debug the problem:

1. **Check Orbital Elements**: Ensure that the orbital elements you're providing are valid. For instance, make sure the semimajor axis is positive, the eccentricity is between 0 and 1 for elliptical orbits, and the inclination is within the range of 0 to 180 degrees.
2. **Units Consistency**: Double-check the units of the orbital elements. For example, if the semimajor axis is expected in meters, ensure you're not mistakenly providing it in kilometers. Similarly, angles should be in radians unless specified otherwise.
3. **Frame Setup**: Confirm that the frames and objects are properly set up before calling `initializeFromOrbitalElements`. The function relies on the frames being correctly defined to compute the position and velocity.
4. **Initialization Order**: Make sure the initialization sequence is followed correctly. The spacecraft's position and velocity initialization should occur after all necessary setup functions like `setupFrames` and `setupSignals` have been executed.
5. **Error Handling**: Since the error is raised during startup, consider adding error handling or logging around the initialization section to capture more detailed information about the inputs and state at the time of failure.
6. **Review Code Changes**: If the error appeared after recent code changes, review those changes for any unintended modifications to the initialization parameters or sequence.

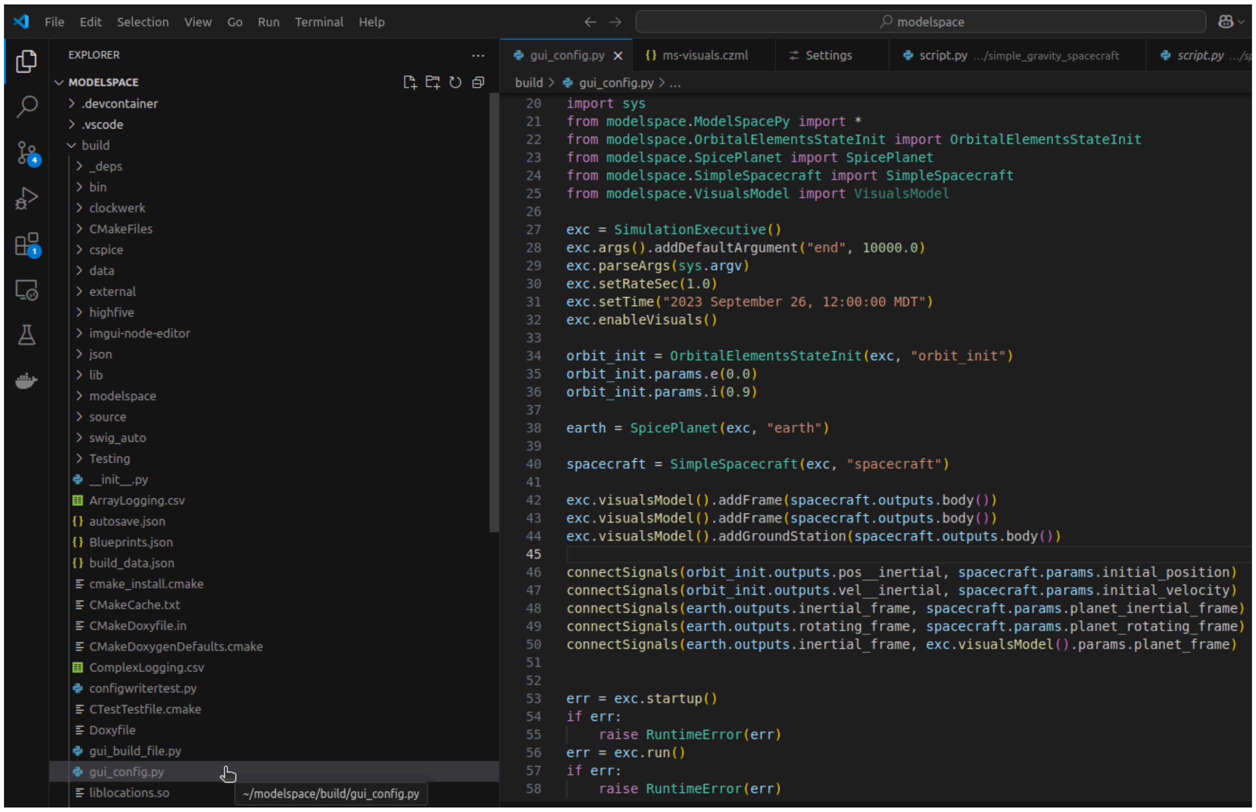
By addressing these areas, you should be able to pinpoint the issue with the orbital elements conversion and resolve the error in your simulation.



```
############################################################################
SET ARGUMENTS:
(Default Values):
        end:             10000.000000
        save-all-outputs: 0
        write-data-json:  1
        rng-seed:        0
        out-dir:         results/
        run:             0
        end:             10000
.exc.orbit_init:        Unable to convert orbital elements to position/velocity state
FILE : /home/mmathew/modelspace/cpp/src/models/states/OrbitalElementsStateInit.cpp
FUNCTION : start
LINE : 45
.exc.schedule:  Error on startup in task .exc.orbit_init
FILE : /home/mmathew/modelspace/cpp/src/simulation/SimScheduler.cpp
FUNCTION : startup
LINE : 120
.exc:   Raised error code 3 on startup.
FILE : /home/mmathew/modelspace/cpp/src/simulation/SimulationExecutive.cpp
FUNCTION : startup
LINE : 303
Traceback (most recent call last):
  File "/home/mmathew/modelspace/build/gui_config.py", line 60, in <module>
    raise RuntimeError(err)
RuntimeError: 3
Segmentation fault (core dumped)
Command to execute python script finished with result: 35584
Error running simulation
```

Turning GUI Output into Scripts

The ModelSpace GUI has many key user-friendly capabilities, but it's also designed to scale with a user's efforts as they start to develop more complicated simulations. As shown in the image below, everything the user builds within the GUI is auto-coded into a python script named "gui_config.py" within VSCode! This enables the user to easily run monte carlo simulations, interface with external tools/APIs and extend their simulations. For examples of how to extend your spacecraft simulation through python scripting, checkout the "/python/scripts/examples/" folder within your ModelSpace VSCode project.



```python
import sys
from modelspace.ModelSpacePy import *
from modelspace.OrbitalElementsStateInit import OrbitalElementsStateInit
from modelspace.SpicePlanet import SpicePlanet
from modelspace.SimpleSpacecraft import SimpleSpacecraft
from modelspace.VisualsModel import VisualsModel

exc = SimulationExecutive()
exc.args().addDefaultArgument("end", 10000.0)
exc.parseArgs(sys.argv)
exc.setRateSec(1.0)
exc.setTime("2023 September 26, 12:00:00 MDT")
exc.enableVisuals()

orbit_init = OrbitalElementsStateInit(exc, "orbit_init")
orbit_init.params.e(0.0)
orbit_init.params.i(0.9)

earth = SpicePlanet(exc, "earth")

spacecraft = SimpleSpacecraft(exc, "spacecraft")

exc.visualsModel().addFrame(spacecraft.outputs.body())
exc.visualsModel().addFrame(spacecraft.outputs.body())
exc.visualsModel().addGroundStation(spacecraft.outputs.body())

connectSignals(orbit_init.outputs.pos__inertial, spacecraft.params.initial_position)
connectSignals(orbit_init.outputs.vel__inertial, spacecraft.params.initial_velocity)
connectSignals(earth.outputs.inertial_frame, spacecraft.params.planet_inertial_frame)
connectSignals(earth.outputs.rotating_frame, spacecraft.params.planet_rotating_frame)
connectSignals(earth.outputs.inertial_frame, exc.visualsModel().params.planet_frame)


err = exc.startup()
if err:
    raise RuntimeError(err)
err = exc.run()
if err:
    raise RuntimeError(err)
```

More GUI Examples

For scenario examples utilizing the ModelSpace GUI, check out our YouTube page.

Building ModelSpace Scripts

Scripting allows you to utilize the full potential of ModelSpace including more complicated scenarios, custom classes, monte carlo simulations, full GNC implementation and much more. The scripting is done in python while the models and base code are developed in C++. This simplifies the user interface, particularly for users with little to no coding experience. While there is no one right way to make a script, the following section outlines the typical structure of a basic ModelSpace script for your reference.

Elements of a Basic Script

The beginning of every well written script includes a description of the purpose and implementation of the script as well as the author information. All of the ATTX examples which can be found in the "/python/scripts/examples/" folder include this important element. After that are the "imports" which tell python which ModelSpace elements you want to access in this script. These are usually filled in as you go along and realise what you actually need. Imports in ModelSpace are set up to be easy for the user to intuitively know how to import a necessary component. All models follow the pattern of

```python
from modelspace.ModelName import ModelName
```

The non-models and utilities are all imported from the same place following the pattern of

```python
from modelspace.ModelSpacePy import SimulationExecutive, CsvLogger
```

The first step in almost every script is to set up the simulation executive. The simulation executive is a class within the ModelSpace framework responsible for managing and executing simulations. It includes various components necessary for simulation, such as scheduling, logging, and visual management. The simulation executive handles parsing arguments, setting simulation parameters, and controlling the overall simulation flow. It provides functionalities for setting the integrator type, simulation rate, and end time, as well as initializing and stepping through the simulation. Overall, the simulation executive acts as the central hub for coordinating and running simulations effectively. The most common set up of the simulation executive looks like

```
Python
exc = SimulationExecutive()   # Create our executive -- by convention named exc
exc.parseArgs(sys.argv)       #This interperets command-line inputs
exc.setRateHz(1)              #We  can  setRateHz  or  setRateSec  --  default  is  1
second
```

These lines first create a simulation executive named exc, parse any command line arguments then set the rate at which the simulation will run. The usual next step after creating the simulation executive is to define the celestial bodies in your simulation. Generally, this is the Earth and sometimes the Sun and even asteroids, but it can be any combination of celestial bodies based on your need. The easiest way to establish these is to use the spice planet model. This can be done as follows

```
Python
earth = SpicePlanet(exc, "earth")
sun = SpicePlanet(exc, "sun")
```

The next step is to create the spacecraft needed for the simulation. This could be one or multiple and is usually done using the Spacecraft model as follows.

```
Python
sc = Spacecraft(exc, "sc")
sc.configFromPlanet(earth)
```

The configFromPlanet method of the spacecraft class is used to establish the relationship between the spacecraft and the planet, essentially configuring the spacecraft to orbit around the planet. This involves setting up the necessary references and parameters so that the spacecraft's orbit can be accurately modeled with respect to the planet's gravitational influence.

Next, any necessary models are configured. These could include things like sensors, actuators, solar panels, GNC algorithms, etc. This simple example does not include any additional models so we will move on. Once all models have been connected and properly initialized, the next step is to set up an output. ModelSpace supports logging to csv, HDF5, and MATLAB mat objects. Please see “Logging” for more information on each type of logger. The general interface is the same for both and can be seen below.

```python
states = CsvLogger(exc, "sc_states.csv")

states.addParameter(exc.time().base_time, "sim_time")
states.addParameter(sc.outputs.pos_sc_pci, "sc_inrtl_position")
states.addParameter(sc.planetRelativeModel().outputs.latitude_detic,
"latitude")
states.addParameter(sc.planetRelativeModel().outputs.longitude, "longitude")
states.addParameter(sc.planetRelativeModel().outputs.altitude_detic,
"altitude_m")
states.addParameter(sc.outputs.vel_sc_pci,"sc_inrtl_velocity")

exc.logManager().addLog(states, 1)
```

There are three main parts separated in the above code. First, an instance of the CsvLogger is created and the file name is taken as an input. Next, the data is passed to the data logger using the addParameter method with an additional argument specifying the desired name for the data. This name will be saved in the first row of the file to serve as a column header. Lastly, the data logger is passed to the log manager to set the rate at which the data will be logged.

After the data logging has been set up, this is usually where visualizations are created if desired. The simplest way to achieve this is to use VizKit as follows,

```python
vk = VizKitPlanetRelative(exc)
vk.target(sc.outputs.body())
vk.planet(earth.outputs.inertial_frame())
exc.logManager().addLog(vk, Time(100))
```

The setup is similar to the data logger in that it has the same three steps. First, the instance of the VizKit visual is created, then the spacecraft and planet objects are passed to the model, and finally the visual is passed to the log manager to set the rate at which the visual will be updated. When the script is run the visual will automatically appear and play as the simulation is executed.

Once all of those steps are complete we can call

```Python
exc.startup()
```

This is responsible for initializing the execution environment, which recursively initializes all models within the framework. This setup is crucial for configuring spacecraft to orbit a planet because it ensures that all necessary components and models, including the spacecraft and any planetary models, are properly initialized and ready for further configuration, such as setting initial positions, velocities, and orbital parameters. This basically sets up the scenario and awaits any final inputs that depend on the scenario setup such as spacecraft initial states.

As alluded to above, we can now establish the initial state of our spacecraft. The most intuitive way to do this is with orbital elements as follows

```Python
sc.initializeFromOrbitalElements(a, e, i, raan, w, f)
```

The last step in every script is to run the simulation using

```Python
exc.run()
```

This function executes the simulation loop, which automatically progresses the simulation time and updates the state of all models and components configured within the environment. The simulation continues until it reaches its predefined end time or a termination condition is met.

To run the script you can either download the vscode python extension and use the green play button in the top right corner of the screen or you can use the terminal to navigate to the location of the script and run

```Python
python3 <scriptname>.py
```

When the simulation is run ModelSpace will output some useful information,, including the end time and the execution speed versus real time, to the terminal.

Simulation Structure

The ModelSpace framework contains multiple elements, all of which work together to generate a high fidelity simulation. Each element which users may interact with is documented in this section.

Simulation Executive

The simulation executive acts as a single, central hub for everything which occurs in ModelSpace. It is a collection of utilities which provides the following functionality:

- Parsing command-line arguments
- Managing rates and run time
- Providing time and time conversions
- Establishing an organization and address scheme for simulation data
- Providing a handle to high fidelity visuals
- Setting up logging for data
- Scheduling and sequencing models
- Providing simulation dynamics and state integration

Creating a SimulationExecutive

Description	**C++ Example**	**Python Example**
Constructor - Creates a SimulationExecutive object.	C/C++ `SimulationExecutive exc;`	Python `exc = SimulationExecutive()`

Parsing Arguments

Description	**C++ Example**	**Python Example**

Parsing Command Line Arguments - Parses command line arguments which are provided from script start in C++ and Python.	C/C++ `exc.parseArgs(argc, argv)`	Python `# This parsing method used only for C++`
Parsing Arguments from Vector - Parses arguments from a vector of strings.	C/C++ `// This method only used in Python`	Python `exc.parseArgs(sys.argv)`

Setting Simulation Parameters

Description	**C++ Example**	**Python Example**
Setting Integrator - Sets the integrator type. Current valid options are 1 or FORWARD_EULER for forward euler and 4 or RK4 for RK4	C/C++ `exc.integrator(1);`	Python `exc.integrator(1)`
Getting Integrator - Gets the integrator type.	C/C++ exc.integrator();	Python exc.integrator()
Setting Simulation Rate (Hz) - Sets the simulation run rate in Hz.	C/C++ exc.setRateHz(100);	Python exc.setRateHz(100)

Setting Simulation Rate (Seconds) - Sets the simulation run rate in seconds.	C/C++
	`Time rateSec(1);`
	`exc.setRateSec(rateSec);`
		Python
		`rateSec = Time(1)`
		`exc.setRateSec(rateSec)`

Setting Simulation End Time - Sets the simulation end time. Note: better to set this from the command line with --end=<endtime>	C/C++
	`double endTime = 100.0;`
	`exc.end(endTime);`
		Python
		`endTime = 100.0`
		`exc.end(endTime)`

Getting Simulation End Time - Gets the simulation end time.	C/C++ `Time endTime = exc.end();`	Python `endTime = exc.end()`

Running the Simulation

Description	**C++ Example**	**Python Example**
Starting the Simulation - Initializes the simulation executive and its components.	C/C++ `int result = exc.startup();`	Python `result = exc.startup()`
Stepping the Simulation - Steps the scheduler by a single step.	C/C++ `result = exc.step();`	Python `result = exc.step()`
Stepping the Simulation with Step Size - Steps the scheduler by a specified step size.	C/C++ `Time stepSize(1);`	Python `stepSize = Time(1);`

	`result = exc.step(stepSize);`	`result = exc.step(stepSize)`

Searching

Description	C++ Example	Python Example
Searching the Simulation Tree - Searches the simulation tree for a match.	C/C++ `std::vector<std::string> results = exc.search("query");`	Python `results = exc.search("query")`
Searching the Frame Tree - Searches only the frame tree for a match.	C/C++ `std::vector<std::string> results = exc.searchFrameTree("query");`	Python `results = exc.searchFrameTree("query")`
Searching the Simulation Architecture Tree - Searches the simulation architecture tree for a match.	C/C++ `std::vector<std::string> results = exc.searchSimTree("query");`	Python `results = exc.searchSimTree("query")`

Logging and Visuals

Description	C++ Example	Python Example

Adding a Logger with Rate - Registers and sets up a logger with a specified rate in Hz.	C/C++
	`CsvLogger logger(...);`
	`exc.addLog(logger, 50);`
		Python
		`logger = CsvLogger(...);`
		`exc.addLog(logger, 50)`

Enabling Visuals - Enables visuals for the simulation.	C/C++ `exc.enableVisuals();`	Python `exc.enableVisuals()`
Disabling Visuals - Disables visuals for the simulation.	C/C++ `exc.disableVisuals();`	Python `exc.disableVisuals()`

Additional Operations

Description	**C++ Example**	**Python Example**
Checking Termination Status - Returns the termination flag of the simulation.	C/C++ `bool terminated = exc.isTerminated();`	Python `terminated = exc.isTerminated()`
Getting Run Number - Gets the run number for the simulation.	C/C++ `unsigned int runNumber = exc.runNumber();`	Python `runNumber = exc.runNumber()`

Accessing Arguments - Accesses the arguments of the simulation.	C/C++ `ArgParser* args = exc.args();`	Python `args = exc.args()`
Accessing Dispersions - Accesses the dispersions of the simulation.	C/C++ `DispersionEngine* dispersions = exc.dispersions();`	Python `dispersions = exc.dispersions()`

Time and Timing

The simulation, through the SimTimeManager and the SimulationExecutive, provides users with access to simulation time in multiple formats:

- **base_time** is the core unit of time in the simulation. It begins at 0.0 seconds at simulation start, and increments with every simulation step. This value is provided as a Time object
- **tdb_time** is the "real world time" unit of time in the simulation, and is equivalent to the ephemeris time value which JPL's SPICE utilizes for determining planet states. This value is provided as a Time object
- **utcTime** is a function provided by the time manager which returns the current simulation time as a UTC time string in ISOC format
- **jdTime** is a function provided by the time manager which returns the current simulation time in JD format
- **gpsTime** is a function provided by the time manager which returns the current simulation time in GPS time format as a string

Simulation Steps

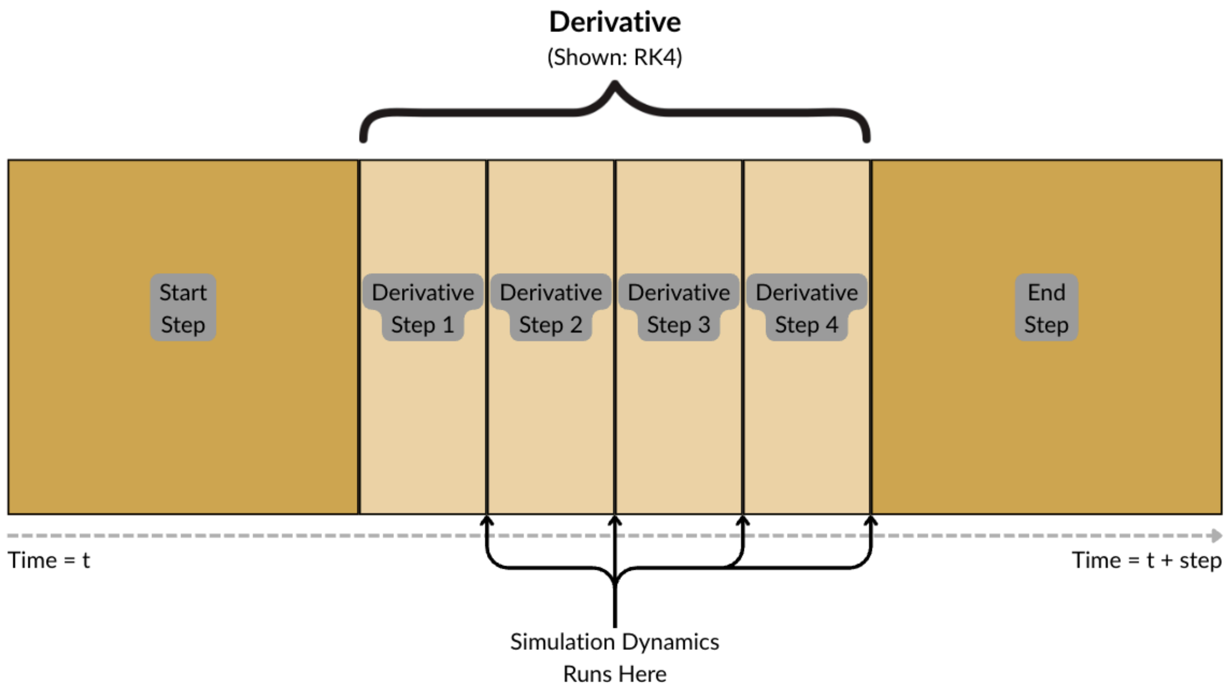
Execution of a ModelSpace simulation is broken down into a series of simulation steps. Each step, the simulation moves time forward such that simulation time at the end of the step is equal to the previous time plus the step size.

The simulation can be stepped once by calling the SimulationExecutive step() function. It can be run with repeated steps until sim termination by calling the SimulationExecutive run() function.

Each simulation step is broken down to a number of substeps, each of which corresponds to a time when models in the simulation can run. The substeps are as follows:

- **START_STEP** occurs at the start of each simulation step. In START_STEP, simulation time is equal to the time at the start of the step, and no states have yet been updated. START_STEP runs once per simulation step always.
- **DERIVATIVE** occurs after START_STEP and is the simulation substep where forces, moments, and accelerations which feed simulation dynamics in the sim integrator are calculated. At the end of the DERIVATIVE step the simulation dynamics engine runs and updates frame states. Time is updated in this step. This step runs as many times per simulation step as there are steps in the simulation integrator – that is, 4 times per step for RK4, once for Forward Euler, etc.
- **END_STEP** occurs at the end of every simulation step after DERIVATIVE has been run as many times as necessary. Time during END_STEP is equal to the time at START_STEP plus the simulation step size. Simulation logging occurs at the end of END_STEP, once all models have run, by default.

Figure 1 shows the simulation steps via diagram.



Models

Models form the core of the ModelSpace simulation, and are the unit responsible for calculating forces and moments which feed ModelSpace dynamics, and producing measurements which represent sensors and provide ModelSpace users with information. ModelSpace models are designed specifically to be easy to build – the user is not directly responsible for running ModelSpace models or managing dynamics so long as they are built into the ModelSpace framework. Further, models have abstracted, mappable inputs and outputs which make them easy to connect, like legos.

Each model has two sets of inputs:

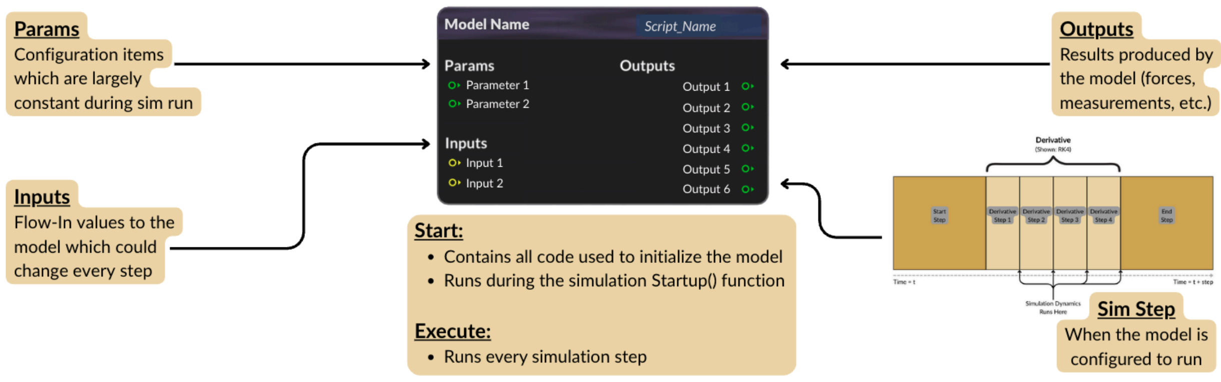
- **PARAMS** are model inputs which serve as configuration items typically set once at simulation start. PARAMS represent items like drag coefficients, masses, gravitational parameter, etc. In ModelSpace params should only be set before simulation startup – after startup, the behavior of modifying params is not guaranteed.
- **INPUTS** are model inputs which are expected to change from step to step. Inputs are most typically mapped to the outputs of other models which live upstream of their model.

And each model has only one set of outputs, called **OUTPUTS.** OUTPUTS can represent values like forces and moments, which are typically mapped to simulation nodes, and can also represent items such as sensor outputs.

In addition to PARAMS, INPUTS, and OUTPUTS, models are scheduled according to the simulation steps – that is, they can run in START_STEP, DERIVATIVE, and END_STEP. In addition, models can be scheduled to run in ALL – that is, all of the listed steps, STARTUP_ONLY, and UNSCHEDULED.

Basic Model CONOPS per simulation step looks like this: each Model is run once during simulation startup (this is when the Model start function is called), and then is executed each simulation step according to when it is scheduled (this is when the Model execute function is called). For instance, a Model scheduled for START_STEP would run its start function on simulation startup and its execute function in every simulation START_STEP. In the Model's execute function, it would take the PARAMS configured for it and the INPUT values at that step and map them to its own OUTPUTS.

Figure 2 shows the various items associated with Models.


Params
Configuration items which are largely constant during sim run
Inputs
Flow-In values to the model which could change every step
Model Name
Script_Name
Params
Parameter 1
Parameter 2
Inputs
Input 1
Input 2
Outputs
Output 1
Output 2
Output 3
Output 4
Output 5
Output 6
Start:
Contains all code used to initialize the model
Runs during the simulation Startup() function
Execute:
Runs every simulation step
Outputs
Results produced by the model (forces, measurements, etc.)
Derivative
Start Step
Derivative Step 1
Derivative Step 2
Derivative Step 3
Derivative Step 4
End Step
Simulation Dynamics Runs Here
Sim Step
When the model is configured to run

Logging

In general, there are three steps to configuring logging in ModelSpace regardless of the language. The three steps are:

- Creation of the logger — right now that can be CsvLogger or Hdf5Logger
- Addition of parameters — the same (as defined below) regardless of logger type
- Passing the logger to the executive — the same regardless of logger type

CsvLogger: Configures ModelSpace to log output data to comma delimited CSV files. CSV logging is the easiest to read, as it outputs data in human-readable text format. However, it is also space inefficient, and can suffer from precision issues (ModelSpace logs to 15 decimal places of precision by default, but text files are always inexact)
Hdf5Logger: Configures ModelSpace to log output data to the binary HDF5 format. HDF5 is an efficient binary method for saving large amounts of data, and the format ATTX recommends for ModelSpace logging. HDF5 saves data in exact form, but is not typically human readable. However, VSCode offers some fantastic extensions to overcome that limitation – ATTX recommends the H5web extension for viewing HDF5 data. ATTX has also provided a data loader to load HDF5 files in Python – more on that in the "Processing Data" section.

Creation of the Logger

This example shows creating a logger as Csv in Python and Hdf5 in C++, but those can be easily reversed. Creation of the logger involves creating it as a child of the executive and providing a name, which will become the logged filename
C++

```
C/C++
CsvLogger state(exc, "states.h5");
```

Python

```Python
state = CsvLogger(exc, "states.csv")
```

Adding Parameters to the Logger

Adding parameters to the logger should be done as defined below. Simple C++ and Python examples are provided here.

C++

```C/C++
// Add time as a logged parameter by object directly, with name time
state.addParameter(exc.time()->base_time, "time");

// Add a hypothetical model velocity output to model via address
state.addParameter(".exc.fs_model.outputs.vel_tgt_ref__ref", "sc_vel");
```

Python

```Python
# Add time as a logged by object with name time
state.addParameter(exc.time().base_time,"time")

# Add veolocity by address with name sc_vel
state.addParameter(".exc.fs_model.outputs.vel_tgt_ref__ref", "sc_vel")
```

Passing the Logger to the Executive

The final step in configuring logging is to pass the logger to the executive, which will automatically run it at a configured rate. Simple examples are provided below.

C++

```C/C++
// Set our log to run at a rate of 1 Hz
exc.logManager()->addLog(state, 1);
```

Python

```python
exc.logManager().addLog(state, 1)# Set our log to run at a rate of 1 Hz
```

Adding Logging Parameters

The following functions to add logging parameters are the same regardless of which logger is applied.

Description	C++ Example	Python Example
Adding Parameter by Object - Adds a logging parameter by passing the object itself	C/C++ `logger.addParameter(object, "paramName");`	Python `logger.addParameter(object, "paramName")`
Adding Parameter by Address String - Adds a logging parameter by passing an address string. This exists for almost all objects in the sim, and can be handy for accessing objects more easily.	C/C++ `logger.addParameter("paramAddress", "paramName");`	Python `logger.addParameter("paramAddress", "paramName")`
Adding Parameter with recursive logging. This logs the parameter and all of its children — for example, passing a model to this would log all inputs and outputs to the model.	C/C++ `logger.addParameter(object, "paramName", true);`	Python `logger.addParameter(object, "paramName", true)`

Processing Data

Data logged in ModelSpace can easily be processed in Python or MATLAB. Loading data is straightforward in both tools:

```python
# PYTHON: Loading .csv files. Can be loaded by any methodology, but ATTX recommends and pre-installs pandas
import pandas as pd
df = pd.read_csv("results/filename.csv")       # Data is now pandas dataframe

# PYTHON: Loading .h5 files. ATTX provides a function to automatically load H5 files in the same format as CSV for portability
from modelspaceutils.analysisutils import readH5Dataframe
df = readH5Dataframe('results/filename.h5')    # Data is now pandas dataframe
```

```
% MATLAB: Load HDF5 data as native variables
% Here variable_name is the name you set the variable to log to in addParameter
data = h5read('Filename.h5', '/variable_name');
```

Additionally, ATTX provides helper functions in Python to assist with data processing and visualization. Data can be visualized in Google Earth using the plotGoogleEarth tool:

```python
# Assume we have a dataframe with items lat_deg, lon_deg, alt_m
plotGoogleEarth(df['lat_deg'], df['lon_deg'], df['alt_m'])
```

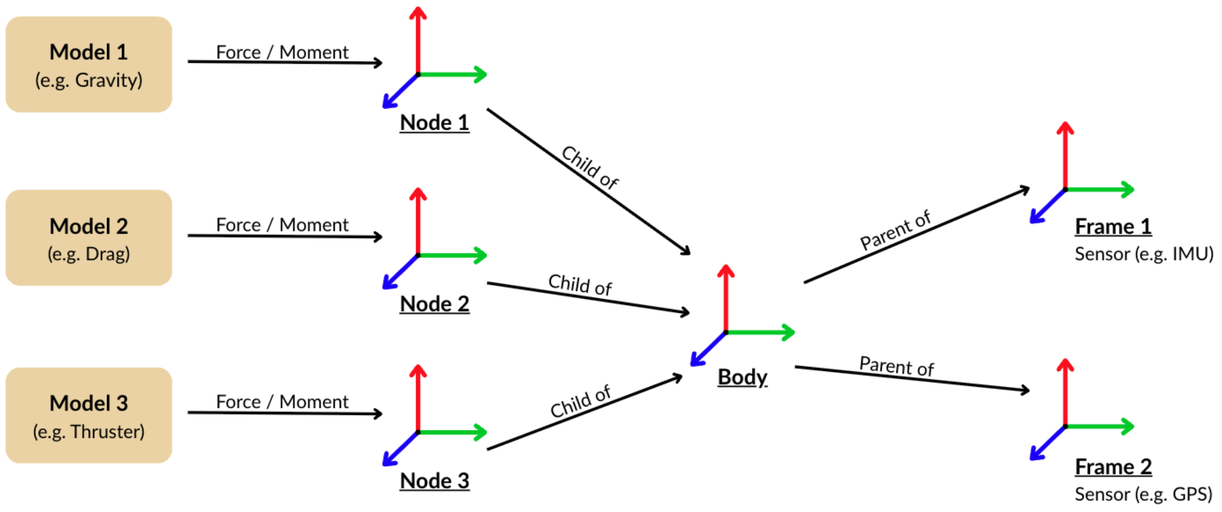
Dynamics

Once forces are calculated in models, a natural question arises: how do we translate these forces into dynamics which can be integrated over time to produce simulation results? The answer to that question is the ModelSpace dynamics engine. This section details how the simulation dynamics work at a high level. A detailed description of the dynamics library is provided in the following section.

The ModelSpace dynamics engine is based on the concept of Frames, which are collections of position, velocity, attitude, and angular velocity information. Frames come in multiple flavors, which includes Bodies (which are frames with mass to represent i.e. a spacecraft body) and Nodes, which exist to translate forces onto frames and produce accelerations.

Connecting the force output from a model to a Node is the method for connecting Models to the simulation dynamics engine. Typically, each force on a body (such as gravity, drag, solar radiation pressure, etc) has its own node connected to a Model force output. These Nodes are children of a Body. In this configuration, ModelSpace can automatically determine the accelerations on every frame in the simulation and integrate them to determine states over time.

At the end of each simulation derivative step, the simulation calculates and integrates the sum of forces on each body and the acceleration of each frame. It does so beginning with the SimulationExecutive root frame and recursively traveling to each frame. For ModelSpace to integrate a frame, it must be a child of the SimulationExecutive root frame.



Manipulating Signals

The fundamental unit of interaction in ModelSpace is the signal, which are defined by the SIGNAL macro in models as shown below:

```
C/C++
        START_PARAMS
              SIGNAL(m,                          double,                    1.0)
              SIGNAL(b,                       double,                 0.0)
          END_PARAMS

          START_INPUTS
              SIGNAL(x,                       double,                 0.0)
          END_INPUTS

          START_OUTPUTS
              SIGNAL(y,                       double,                 0.0)
          END_OUTPUTS
```

Signals are unique in that they can be set, gotten, and connected to one another such that they reflect one another's value.

Accessing and using signals follows the pattern <model name>.<input/output type>.<name>, where the value held by signals is accessed by calling them with empty parenthesis (), and set by passing a value in parenthesis:

```C/C++
// Examples setting signals
drag.inputs.in_value(value_to_set_signal_to);
gravity.params.mu(300000.0);

// Examples getting signals
double mu = gravity.params.mu();
```

Getting and setting signal values is based on their value at any given time, but ModelSpace is constantly running and updating values every step. How do we ensure that values are always tracking? For that, we use the connectSignals macro. Signals connected by connectSignals always reflect one another – that is, it allows the input to a model to track the output from another model. Graphically, the connectSignals macro is reflected in dragging connections in the ModelSpace GUI, and in fact connecting signals in the GUI calls connectSignals.

```
C/C++
// Connect signals -- this makes the input to atmosphere always track the
output to planet relative
connectSignals(planet_relative.outputs.altitude, atmosphere.inputs.altitude)
```

From this example, we can see the benefit of the connectSignals function – by making the inputs to models track the outputs of others, we can very easily construct large and complex simulation without adding complexity to any models.

Simulation Script Arguments

When run from the command line, ModelSpace offers a number of script arguments which may be modified at runtime. Script arguments provided in the simulation by default are:

- end (default 10,000 seconds): The end time for the simulation, in seconds of simulation time.
- run (default 0): The run number for the simulation. This value is primarily used for Monte Carlo simulation. Modifying it changes the output of dispersions defined in ModelSpace. Monte Carlo sets are generated by running the simulation while iterating through options, i.e. run=1, run=2, etc. Setting run to 0 (the default) takes the nominal value of all dispersion objects.
- out-dir (default 'results'): The output directory to which the simulation should write log files, configuration json information, etc.
- rng-seed (default 0): Seed for the random number generator. Modifying this value changes the output draw of dispersions in the sim, for all run numbers.
- write-data-json (default true): Flag to determine whether simulation parameters should be written to json.
- save-all-outputs (default false): Flag to determine whether the all sim outputs should be logged. This flag can generate ridiculous amounts of data, so don't set it unless you really want *everything* logged.
- enable-visuals (default false): Command line flag to enable visuals in ModelSpace. Passing this argument is the same as setting exc.enableVisuals() in the python script for ModelSpace.

Arguments set at the command line should be set with a leading double dash (--). Example uses:

```
None
# Set sim to end at 3,000 seconds
python3 script.py --end=3000

# Enable visuals And set output to a directory called "res"
python3 script.py --enable-visuals=true --out-dir=res

# Execute run 1 of the simulation
python3 script.py --run=1
```

The default values provided for simulation executive script arguments can be overridden by setting a new default value in the simulation. Calling addDefaultArgument(<arg_name>, <new_default_value>) produces a new default value for the simulation end. Command line arguments are parsed when calling exc.parseArgs, so new default values must be set before calling parseArgs, as shown below:

```
Python
exc = SimulationExecutive()
exc.args().addDefaultArgument("end", 100)      # Set default end to 100 seconds
exc.parseArgs(sys.argv)                        # Must occur after addDefaultArg
```

Running Monte Carlo Simulation

ModelSpace comes with a built-in engine to run Monte Carlo simulation. The engine is based on a run number input, and is designed to work with in-house launcher solutions (i.e. slurm). A Monte Carlo launcher shell script is also provided.

Defining Variables

Variables are defined as “dispersion” objects, which can be accessed directly as single values for a simulation run, but with values which are modifiable. An example of both a uniform and gaussian distribution are shown below:

```python
# Uniform
semimajor_axis=exc.dispersions().createUniformInputDispersion("semimajor_axis",
10678140.0, 10878040.0, 10778240.0)   # (name, min, max, default) for uniform
# Gaussian
eccentricity=exc.dispersions().createUniformInputDispersion("eccentricity",
0.001, 1e-6, 0.1)     # (name, mean, std, default) for gaussian

# Access variables
semimajor_axis()()  # Returns value held by dispersion for this run.
```

Once dispersions are established, a single Monte Carlo run can be executed as:

```
python3 script.py --run=<run_number>
```

In ModelSpace, run 0 is a special, undispersed run (always takes the default value) and all other runs are dispersed. Run 0 is the default which executes when no run argument is passed.

The Monte Carlo example script shows how to build Monte Carlo simulation, and also provides the generic Monte Carlo launcher shell script.

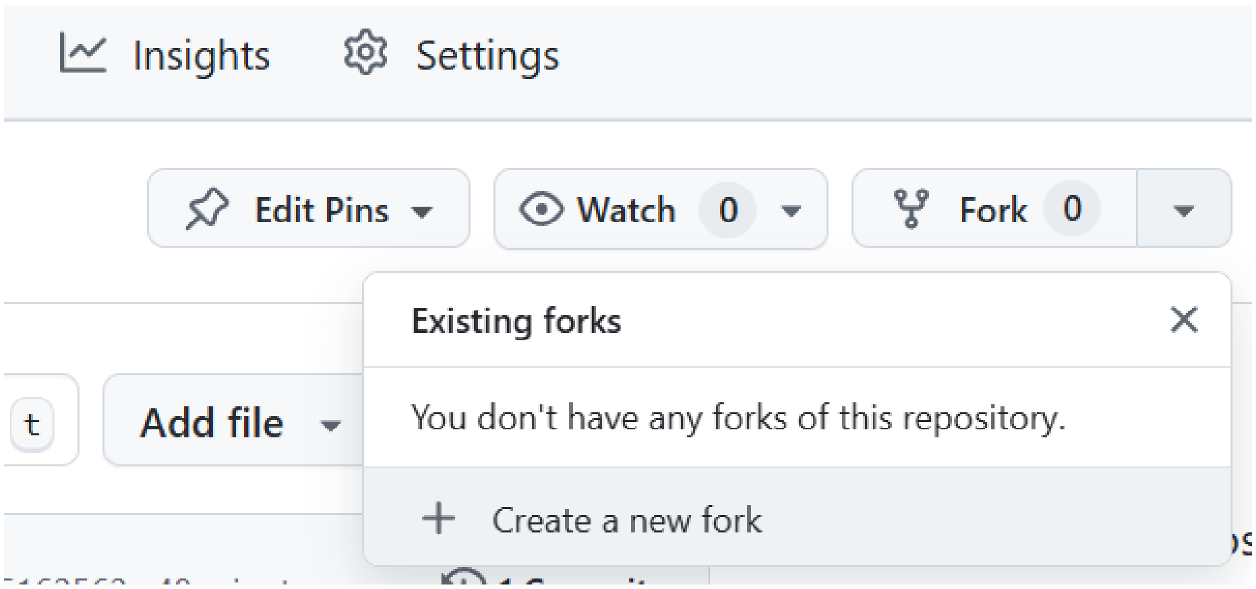
Building Custom Models

ModelSpace is designed for extension so users can create custom models of their vehicle systems, sensors, actuators, and the simulation environment. These custom models can be built into the ModelSpace GUI and utilized in scripting. The following tutorial walks users through the custom model development process with an end-to-end example using a "Slope Intercept Model"

While not strictly required, ATTX recommends building custom models using the ModelSpace development kit on the ATTX GitHub. The dev kit comes with everything necessary to compile custom models as well as integrate them with the GUI and run scripts.

Forking the ModelSpace Custom Repo

To build a custom model, first fork the ModelSpace development kit by going to forks →create new fork and building a new public or private fork within your organization's Git.



While forking isn't required, you will not be able to push changes back to modelspace-custom (as it is a public/shared resource). Forking is the easiest way to track and maintain your organization's custom code.

Once your repository is forked, clone it on your local (WSL, Docker, Linux, or other ModelSpace environment system. If you followed the Windows install instructions, your

WSL Ubuntu 22.04 instance is where you should clone). Cloning modelspace-custom is shown here:

```
None
git clone https://github.com/attx-engineering/modelspace-custom.git
```

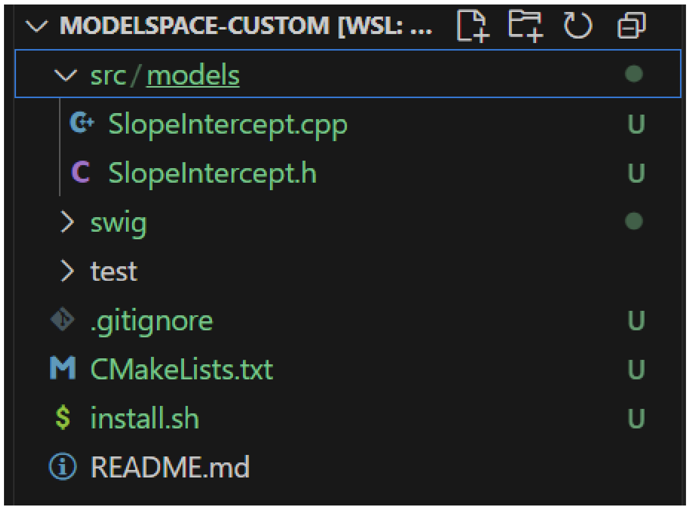
Your cloned repository comes with a number of files and directories designed to make building custom models easy. Those are:

- CMakeLists.txt: This file contains the build instructions used to generate your custom models and code
- src: This directory is where all of your source code will go. All custom source code should go in src, and all models should go in src/models.
- swig: This directory contains files used to generate Python bindings for custom models
- test: ATTX provides a test harness for users to test their models using GoogleTest. All tests should go in this directory
- install.sh: Handy script to install all dependencies necessary to build custom models

Making a Custom Model

Now let's build our custom Slope Intercept Model. This model is an implementation of the basic math equation:

Y = mx + b



```
MODELSPACE-CUSTOM [WSL: ...
  src / models
    SlopeIntercept.cpp    U
    SlopeIntercept.h      U
  swig
  test
  .gitignore              U
  CMakeLists.txt          U
  install.sh              U
  README.md
```

Models are defined in a c++ .h and .cpp file. Create the model h and cpp file in src/models – once created, your file browser in VSCode should look like the image on the left.

Now we'll implement the .h file for our slope intercept model. The .h file defines the entire interface to the model (this is why ATTX provides .h files as model documentation). It also includes the definition for all functions to be implemented in the model – more on that later.

Because our model represents slope-intercept form, we will define it to have the following items:
PARAMS:
 b - The constant slope intercept offset in y=mx + **b**
 m - The constant multiplier in y=**m**x + b
INPUTS:
 x - The input variable in y=m**x** + b
OUTPUTS:
 y - The output variable **y**=mx + b

Remember from documentation on signals, that PARAMS are constant or near-constant variables which are set once at sim start, and INPUTS are variables which are expected to change constantly.

In addition to our PARAMS, INPUTS, and OUTPUTS, we need to implement the following functions for our model:

- start: This function is run once and configures the model for run, typically by using PARAMS
- execute: This function is called every simulation step and actually does the model work of taking PARAMS/INPUTS and mapping them to OUTPUTS
- activate: This function activates the model – models are activated by default, so this only needs to be called once a model is deactivated
- deactivate: This function deactivates the model, meaning it no longer runs when the simulation steps.

Ultimately, our model header looks like the below. Note the definition of params, inputs, and outputs, which form our model's main interface:

```
C/C++
/*
Metadata for MS GUI:
imdata = {"displayname" : "Slope Intercept Model",
          "exclude" : False,
          "category" : "Custom"
}
aliases = {"m" : "Multiplier",
           "b" : "Offset",
           "x" : "Input",
          "y" : "Output"
}
*/

#ifndef MODELS_SLOPE_INTERCEPT_H
#define MODELS_SLOPE_INTERCEPT_H

#include "simulation/Model.h"

namespace modelspace {

    /**
     * @brief   Simple implementation of y = mx + b
    */
    MODEL(SlopeIntercept)
    public:
        START_PARAMS
            /** The multiplier in y=mx+b */
            SIGNAL(m,                       double,                 1.0)
            /** The offset in y=mx+b */
            SIGNAL(b,                       double,                 0.0)
        END_PARAMS

        START_INPUTS
            /** The model input every step */
            SIGNAL(x,                       double,                 0.0)
        END_INPUTS

        START_OUTPUTS
            /** The output from y=mx+b */
            SIGNAL(y,                       double,                 0.0)
        END_OUTPUTS

        int16 activate() override;
        int16 deactivate() override;

    protected:
        int16 start() override;
        int16 execute() override;
    };

}

#endif
```

And our cpp implementation looks like the following:

```
C/C++
#include "SlopeIntercept.h"
#include "simulation/SimulationExecutive.h"

using namespace clockwerk;
using namespace cfspp;

namespace modelspace {
        // These constructor functions can be largely ignored. These are
boilerplate defaults
     // that only need to be modified when including a model within a model, or
setting
      // variable values on construction. The only item worth noting is the
START_STEP
      // in the first two instances of the constructor. This sets the default
step for
    // the model to run in (see User's Guide for more info)
    SlopeIntercept::SlopeIntercept(Model &pnt, const std::string &m_name)
           : Model(pnt, START_STEP, m_name) {}
      SlopeIntercept::SlopeIntercept(SimulationExecutive &e, const std::string
&m_name)
           : Model(e, START_STEP, m_name) {}
        SlopeIntercept::SlopeIntercept(Model &pnt, int schedule_slot, const
std::string &m_name)
           : Model(pnt, schedule_slot, m_name) {}
      SlopeIntercept::SlopeIntercept(SimulationExecutive &e, int schedule_slot,
const std::string &m_name)
           : Model(e, schedule_slot, m_name) {}
    SlopeIntercept::~SlopeIntercept() {}

    int16 SlopeIntercept::activate() {
             // Define behavior to occur when the model is activated after
deactivation
        // Here we just run execute, so outputs reflect the inputs once more.
        active(true);
        execute();

             // Model functions always return an error code. NO_ERROR means
everything
        // went fine, and is an alias for 0. Anything nonzero is intepreted by
        // the sim as an error
        return NO_ERROR;
    }

    int16 SlopeIntercept::deactivate() {
        // Here behavior to deactivate the model is written. At a minimum it
should
        // set the active flag to false, but we can also set up the model to
show
        // it is deactivated. In slope intercept, we will set the outputs to 0.
        active(false);
        outputs.y(0.0);

        // Model functions always return an error code. NO_ERROR means
everything
        // went fine, and is an alias for 0. Anything nonzero is intepreted by
        // the sim as an error
        return NO_ERROR;
    }

    int16 SlopeIntercept::start() {
        // Code used to configure the model goes here. This function runs once
        // at simulation start... Slope Intercept model doesn't have anything
for
        // this, but this is where files should be loaded and items should be
        // pre-calculated.

        // Model functions always return an error code. NO_ERROR means
everything
        // went fine, and is an alias for 0. Anything nonzero is intepreted by
        // the sim as an error
        return NO_ERROR;
    }

    int16 SlopeIntercept::execute() {
        // Code which runs every model step goes here. Per the above and the
User's
        // Guide, valid steps are START_STEP, DERIVATIVE, and END_STEP. In
general
        // here is where you should write values to all outputs on the basis of
        // inputs and params.
        outputs.y(params.m()*inputs.x() + params.b());

        // Model functions always return an error code. NO_ERROR means
everything
```

```
        // went fine, and is an alias for 0. Anything nonzero is intepreted by
        // the sim as an error
        return NO_ERROR;
    }

}
```

Building a Custom Model

Modelspace is built using cmake. First create a build directory within your custom repository, which will hold our binaries, and cd into it.

```
None
mkdir build
cd build
```

Now run cmake. Cmake should be run under the following conditions

- First build of the repo
- Any target files (.h, .cpp, .i) are added or removed
- The .h header file for a model or app is changed

```
None
cmake ..
```

Finally, compile ModelSpace. This should run in approx. 1 minute and produce no errors. The argument `-j<n>` sets the number of cores which should be used to build. Any number up to 20 is usually fine, depending on the system -- higher is faster.

```
None
make -j4
```

ATTX also provides a unit test harness with the custom repository to simplify model testing. Tests can be built as per the provided example, and you can extend the test directory with as many tests as is helpful.

```None
make test
```

Using Custom Models in the GUI

Once our model is built without error, we next want to use it in the GUI. Recall from the model .h file the following comment block:

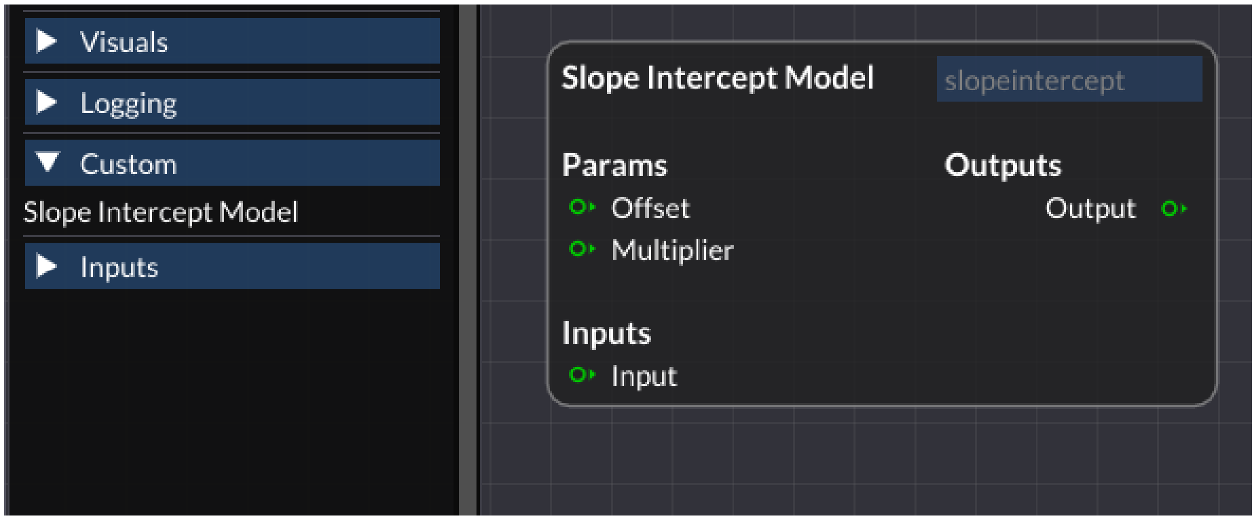
```C/C++
/*
Metadata for MS GUI:
imdata = {"displayname" : "Slope Intercept Model",
          "exclude" : False,
          "category" : "Custom"
}
aliases = {"m" : "Multiplier",
           "b" : "Offset",
           "x" : "Input",
           "y" : "Output"
}
*/
```

This block defines interaction between the model and the GUI. The first block of braces, imdata, is required for using the gui and defines the model name and category. Category should be left constant and displayname should be updated to reflect the current model. The second set of braces, aliases, is optional, and is used to make model inputs and outputs more appealing in the GUI. It takes signal names, defined in the .h file, and maps them to their display name. If a signal is not included in aliases, it displays exactly as defined in signal.

Finally, we want to actually run the GUI with our custom models. For the GUI to pick up our models, it needs to know where they are defined – that is managed by the argument custom-model-file. To run the GUI with our custom models, execute the following on the command line:

```None
# Assumes running from the build directory where custom models are built.
# In general takes the path to a json with other definitions
ms-gui --custom-model-file=custom_models.json
```

Once run, your custom model should show up in the GUI under the “Custom” tab with a black header, as shown:



Using Custom Models in Python

Once custom models are built, ModelSpace automatically wraps them into Python for use in scripting. Custom models can be interacted with in modelspace exactly as built-in models can, as shown in the example:

```Python
from modelspace.SlopeIntercept import SlopeIntercept
from modelspace.ModelSpacePy import SimulationExecutive

exc = SimulationExecutive()
slope_intercept = SlopeIntercept(exc)

slope_intercept.params.m(2.0)
slope_intercept.params.b(3.0)

slope_intercept.inputs.x(5.0);

exc.startup()
exc.step()

print(slope_intercept.outputs.y())
```

ModelSpace Dynamics

Matrix and Vector Math Library

ModelSpace's core mathematics is built atop a **templated, safe, real-time-compliant** C++ `Matrix` library. All vectors, attitude representations, and higher-level linear-algebra constructs derive from this `Matrix` class, which enforces:

- **Compile-time dimensionality** via template parameters `<T, R, C>` for type `T` and sizes `R×C`.
- **Safe embedded usage**, avoiding dynamic allocation and performing bounds-checked operations where necessary.
- **SWIG-wrapped Python interoperability**, so the same API is available in Python scripts.

> **Note:** For full, formal API details, see the Doxygen pages:
> – Matrix: *[Matrix Doxygen →]*
> – Matrix Math Functions: *[Matrix Math Functions Doxygen →]*

---

2. The `Matrix` Class

2.1. Class Template & Macros

The class is declared as:

```
template<typename T, unsigned int R, unsigned int C>
class Matrix { … };
```

- **T**: floating-point type (e.g. `double`, `float`)
- **R**, **C**: integer number of rows and columns

To simplify declarations, ModelSpace provides macros up to 6×6 matrices:

[Table: Common Matrix Macros]

Template	Macro
`Matrix<double,3,3>`	`Matrix3D`
`Matrix<float,3,1>`	`Matrix31F`
…	…

[Insert full macro table from `macros.h` *here]*

2.2. Constructing Matrices

Description	C++	Python
Default (zeros)	`Matrix3D mat;`	`mat = Matrix3D()`
Uniform value	`Matrix3D mat(5);`	`mat = Matrix3D(5)`
Initializer list	`Matrix3D mat({{1,2,3},{4,5,6},{7,8,9}});`	`mat = Matrix3D([[…],[…],[…]])`
From C-array	`double a[3][3] = {{…}}; Matrix3D mat(a);`	`mat = Matrix3D(a)`

Copy	`Matrix3D mat2(mat1);`	`mat2 = mat1`

[Add code-formatted examples and screenshots of the GUI matrix editor here]

2.3. Accessing & Modifying Elements

Safe setter/getter

```
mat.set(row, col, value);      // returns error code
mat.get(row, col, outValue);   // returns error code
```

Unsafe direct access

```
double v = mat[row][col];      // no bounds checks
```

Batch operations

```
mat.setFromArray(ptr);         // row-major array
mat.getAsArray(ptr);
```

[Insert table or GUI screenshot highlighting setter/getter methods]

2.4. Common Operations

Operation	C++	Python
Dump to console	`mat.dump();`	`mat.dump()`
String representation	`std::string s = mat.str();`	`s = mat.str()`
Transpose	`mat.transpose(outMat);`	`outMat = mat.transpose()`
Determinant	`mat.det(detVal);`	`detVal = mat.det()`

Inverse	`mat.inverse(invMat);`	`invMat = mat.inverse()`
Trace	`mat.trace(traceVal);`	`traceVal = mat.trace()`
Max / Min	`mat.max(maxVal, idx); mat.min(minVal, idx);`	-

[Placeholder for detailed examples: "Compute and visualize inverse of a 3×3 matrix"]

---

3. The `CartesianVector` Class

As a specialization, `CartesianVector<T, L>` inherits **all** `Matrix<T,L,1>` functionality and adds vector-specific methods:

```
template<typename T, unsigned int L>
class CartesianVector : public Matrix<T, L, 1> { … };
```

3.1. Declarations & Macros

Template	**Macro**
`CartesianVector<double,3>`	`CartesianVector3D`
`CartesianVector<float,2>`	`CartesianVector2F`
…	…

[Insert full macro table here]

3.2. Constructing Vectors

Description	C++	Python
Default (zeros)	`CartesianVector3D v;`	`v = CartesianVector3D()`
Uniform value	`CartesianVector3D v(5);`	`v = CartesianVector3D(5)`
Initializer list (array)	`CartesianVector3D v({1,2,3});`	`v = CartesianVector3D([1,2,3])`
From `std::array`	`std::array<double,3> a{…};` `CartesianVector3D v(a);`	`v = CartesianVector3D(a)`
Copy	`CartesianVector3D v2(v1);`	`v2 = v1`

[Placeholder for GUI screenshot of vector construction wizard]

3.3. Element Access

Safe:

```
v.set(idx, value);
v.get(idx, outValue);
```

Unsafe:

```
double v0 = v[idx];
```

3.4. Vector-Specific Operations

Operation	C++	Python
Norm	`v.norm(n);/n = v.norm();`	`n = v.norm()`
Norm²	`v.normSquared(n2);`	`n2 = v.normSquared()`

Unit vector (out)	`v.unit(u);`	`u = v.unit()`
Unitize (in-place)	`v.unitize();` / `v.normalize();`	`v.unitize()`
Dot / Cross	`dot(a,b,out);` / `cross(a,b,out);`	-
Tilde (skew-sym.)	`tilde(v, M);`	-

[Placeholder for code snippet: "Compute cross product and visualize with `tilde` *matrix"]*

---

4. Examples & Further Reading

Matrix-Vector Multiplication

```
Matrix3D A({{1,2,3},{4,5,6},{7,8,9}});
CartesianVector3D v({1,0,0});
auto result = A * v;  // calls overloaded operator
```

Combining Operations

```
Matrix3D M = Matrix3D::eye();
M.set(0,1,2.0);
auto Mt = M.transpose();
double trace;
Mt.trace(trace);
```

Attitude Math Library

4.1 Underlying Principles

ModelSpace implements four common attitude parameterizations—Direction Cosine Matrices (DCMs), Quaternions, Modified Rodrigues Parameters (MRPs), and 3-2-1 Euler angles—based directly on the conventions in *Analytical Mechanics of Space Systems* (4th Ed.) by Schaub & Junkins. At their core, all are orthogonal (or normalized) mappings between two Cartesian frames, and each can be converted to any other representation without loss of information (except at known singularities for Euler angles and MRPs).

> **Note:** for a concise overview of attitude kinematics and dynamics, see Dr. Schaub's paper "Attitude Kinematics and Dynamics Reference."
> **Further reading:** full Doxygen documentation is available for each class (see links in each subsection).

4.2 Attitude Math Operations in ModelSpace

ModelSpace overloads operators so that **Multiplication** of two attitude objects chains rotations:

```
auto C = DCM<double>(…);

auto Q = Quaternion<double>(…);
auto result = C * Q;        // implicitly converts Q→DCM, then multiplies
```

Multiplying an attitude object by a vector rotates it into the target frame:

```
CartesianVector<double,3> v_B{…};
auto v_A = quat_A_B * v_B;    // rotates v_B (frame B) into frame A
```

All pass-by-reference return values use the name `result`, and follow the same naming conventions as in Schaub & Junkins (e.g. DCM [AB], quat_A_B, etc.).

> **Placeholder:** *Insert table summarizing multiplication & rotation functions here.*

4.3 DCM Class

Definition: `template<typename T> class DCM : public Matrix<T,3,3>`

- **Default constructor** → identity matrix
- **Constructors** from C-style array or `std::array<…>`
- **Inverse** (transpose) via `inverse()`
- **Kinematics:** `rate(omega, dcmdot)` computes C˙\dot CC˙ from angular velocity
- **Conversions:**
    - `toQuaternion(…)` / `toQuaternion()`
    - `toEuler321(…)` / `toEuler321()`
    - `toMRP(…)` / `toMRP()`

Doxygen: [DCM Doxygen link]

> **Placeholder:** *Insert screenshot of DCM properties in the GUI.*

4.3.1 Examples

```
// create identity DCM
DCM<double> C_id;

// initialize from raw values
DCM<double> C({{1,0,0}, {0,1,0}, {0,0,1}});

// compute inverse
auto C_inv = C.inverse();

// rotate vector
CartesianVector<double,3> v_B{{1,0,0}};
auto v_A = C * v_B;
```

> **Placeholder:** *Include Python equivalent once implemented.*

4.4 Quaternion Class

Definition: `template<typename T> class Quaternion : public CartesianVector<T,4>`

- **Default constructor** → zero rotation (1,0,0,0)
- **Constructors** from C-array or `std::array<…>`
- **Kinematics:** `rate(omega, quatdot)` computes q dot
- **Conversions:**
  - `toDCM(…)` / `toDCM()`
  - `toMRP(…)` / `toMRP()`
  - `rotationAngle(double &angle)`

Doxygen: [Quaternion Doxygen link]

> **Placeholder:** *Insert screenshot of Quaternion widget showing components.*

4.4.1 Examples

```
// default quaternion
Quaternion<double> q_id;

// compose two quaternions
Quaternion<double> q1({1,0,0,0}), q2({0.707,0.707,0,0});
auto q3 = q1 * q2;

// convert to DCM
auto C = q3.toDCM();

// rotate vector by quaternion
```

```
CartesianVector<double,3> v{0,1,0};
auto v_rot = q3 * v;
```

4.5 Euler321 Class

Definition: `template<typename T> class Euler321 : public CartesianVector<T,3>`

- **Default constructor** → all angles zero
- **Constructors** from C-array or `std::array<…>`
- **Kinematics:** `rate(omega, eulerdot)` computes $\dot{\phi}, \dot{\theta}, \dot{\psi}$
- **Conversion:** `toDCM(…)` / `toDCM()`

Doxygen: [Euler321 Doxygen link]

> **Placeholder:** *Include table of singularity conditions (cos θ = 0) here.*

4.5.1 Examples

```
// set a 45° yaw, 30° pitch, 10° roll (in radians)
Euler321<double> ea({0.1745, 0.5236, 0.7854});

// convert to DCM
auto C = ea.toDCM();

// update rates given body rates omega
CartesianVector<double,3> omega{0.01,0.02,0.03};
Matrix<double,3,1> edot;
ea.rate(omega, edot);
```

4.6 MRP Class

Definition: `template<typename T> class MRP : public CartesianVector<T,3>`

- **Default constructor** → σ=[0,0,0]
- **Constructors** from C-array or `std::array<…>`
- **Shadow set:** `shadow()` toggles to σ′
- **Kinematics:** `rate(omega, mrpdot)`
- **Conversions:**
    - `toDCM(…)` / `toDCM()`
    - `toQuaternion(…)` / `toQuaternion()`

Doxygen: [MRP Doxygen link]

> **Placeholder:** *Insert graphic showing how MRPs avoid singularities within unit sphere.*

4.6.1 Examples

```
// initialize MRP
MRP<double> s({0.1, 0.2, -0.1});

// compute shadow if |σ|>1
if(s.norm()>1) s.shadow();

// convert to quaternion
auto q = s.toQuaternion();

// rotate a vector
CartesianVector<double,3> v{1,0,0};
auto v_rot = s * v;   // implicit conversion via DCM
```

Frames, Bodies, and Nodes

This section of the ModelSpace User Guide introduces the foundational building blocks for simulating rigid-body dynamics: Frames, Bodies, and Nodes. Using these building blocks, ModelSpace is able to construct highly complex simulations accounting for multiple spacecraft or planetary bodies, sensors, reference frames, and more without passing that complexity on to the user. By simply defining and configuring frames in the ModelSpace simulation, users can lean on the ModelSpace dynamics engine to construct simulations without detailed knowledge of dynamics themselves.

Dynamic Principles Overview

ModelSpace separates *kinematics* (how things move) from *dynamics* (why things move). In Newtonian mechanics, the state of a rigid body is captured by four key quantities:

- Position **r** (3 components)
- Velocity **v** (3 components)
- Attitude (orientation) (e.g., quaternion or DCM; 4 or 9 components)
- Angular Velocity **ω** (3 components)

These define a 6-DOF state. In ModelSpace:

- Kinematics: Recursive chaining of frames to compute root-relative position, velocity, attitude, and angular velocity.
- Dynamics: Application of forces and moments (via Nodes) to Bodies (frames with mass/inertia) to compute accelerations and integrate states.

Frame Class

Official Frame class documentation

ModelSpace defines three core dynamic containers:

- **FRAME**: fundamental 6-DOF container for kinematics in a chained frame hierarchy.
- **BODY**: extends Frame to include mass and inertia for resolving forces into accelerations; base unit for vehicles.
- **NODE**: specialized fixed Frame for applying forces/moments and sensing accelerations at offsets.

Overview & Naming
The C++ *Frame<T>* class is templated on type T (e.g., double) and tracks key parameters relative to its parent:

- *pos_f_p__p*: Position of child frame origin in parent coordinates.
- *vel_f_p__p*: Velocity of child frame origin in the parent frame.
- *quat_f_p*: Attitude quaternion of child relative to parent.
- *ang_vel_f_p__f*: Angular velocity of child relative to parent, in child coordinates.

Freedom & Joints

- Locked frame: no translation/rotation relative to parent.
- Fully free: translation and rotation allowed.

Frames are connected in ModelSpace via a class called a Joint, which defines the translational and rotational degrees of freedom with which a Frame can move with respect to its parent. Frames which are fully locked cannot move relative to their parent – this is the default behavior for frames, if a Joint behavior is not specified. A good example for fully locked behavior is an IMU frame, which is locked to a parent spacecraft. Frames can also be allowed to move relative to their parent – a good example of this behavior is a spacecraft moving relative to a parent planet inertial frame.

Constructors & Creation

Description	C++	Python
Name only	Frame<double> f1("f1");	f1 = Frame("f1")
Name + parent	Frame<double> f2("f2", &f1);	f2 = Frame("f2", parent=f1)
Name + parent + freedom (free = true)	Frame<double> f3("f3", &f1, true);	f3 = Frame("f3", parent=f1, free=True)

Parent-Relative vs Root-Relative APIs

You can directly manipulate parent-relative signals (`pos_f_p__p`, etc.), or use root-relative functions:

```C/C++
f3.setRootRelPosition({1.0,2.0,3.0});
auto pos_root = f3.rootRelPosition();
```

```Python
f3.setRootRelPosition([1.0,2.0,3.0])
pos_root = f3.rootRelPosition()
```

Accessors & Mutators

Operation	**C++**	**Python**
Get translational joint	`auto tj = f.tJoint();`	`tj = f.tJoint()`
Get rotational joint	`auto &rj = f.rJoint();`	`rj = f.rJoint()`
Set parent	`f.parent(&other);`	`f.parent(other)`
Set position relative to root	`f.setRootRelPosition({1,2,3});`	`f.setRootRelPosition()`
Set velocity relative to root	`f.setRootRelVelocity({0.5,0.5,0.5});`	`f.setRootRelVelocity([0.5,0.5,0.5])`
Set attitude relative to root	`f.setRootRelAttitude(Quaternion<double>({1,0,0,0}));`	`f.setRootRelAttitude()`
Set angular velocity relative to root	`f.setRootRelAngularVelocity({0.1,0.2,0.3});`	`f.setRootRelAngularVelocity([0.1,0.2,0.3])`
Get position relative to root	`auto p = f.rootRelPosition();`	`p = f.rootRelPosition()`

Get velocity relative to root	`auto v = f.rootRelVelocity();`	`v = f.rootRelVelocity()`
Get acceleration relative to root	`auto a = f.rootRelAcceleration();`	`a = f.rootRelAcceleration()`
Get attitude quaternion relative to root	`auto q = f.rootRelQuaternion();`	`q = f.rootRelQuaternion()`
Get DCM relative to root	`auto C = f.rootRelDCM();`	`C = f.rootRelDCM()`
Get angular velocity relative to root	`auto w = f.rootRelAngularVelocity();`	`w = f.rootRelAngularVelocity()`
Get angular acceleration relative to root	`auto alpha = f.rootRelAngularAcceleration();`	`alpha = f.rootRelAngularAcceleration()`

Body Class

Official Body class documentation

The `Body<T>` class derives from `Frame<T>` and adds mass/inertia for 6-DOF dynamics. For full reference, see the Body Doxygen documentation: [Body Doxygen]

Constructors & Creation

Description	**C++**	**Python**
Name only	`Body<double> b1("body1");`	`b1 = Body("body1")`

Name + parent	`Body<double> b2("body2", &f1);`	`b2 = Body("body2", parent=f1)`

Mass & Inertia

Operation	C++	Python
Set mass	`b1.mass(10.0);`	`b1.mass = 10.0`
Set inertia matrix	`b1.inertia(Matrix3D{{1,0,0},{0,1,0},{0,0,1}});`	`b1.inertia = []`

Dynamics Resolution

- `calcFrameTreeExtForcesMoments()` to accumulate forces/moments from Nodes and child Bodies.
- `calcFrameTreeExtAcceleration()` to compute accelerations via mass/inertia.

Node Class

Official Node class documentation

The `Node<T>` class derives from `Frame<T>` (fixed to parent) and models force/moment application points.

Constructors & Creation

Description	C++	Python
Name only	Node<double> n1("node1");	n1 = Node("node)