Developing a Multilingual Auto-coding Interface Control for the MAVERIC-II Dynamics Simulator

Mason Nixon
Leidos, Inc.
Huntsville, AL
mason.e.nixon@nasa.gov

ABSTRACT

Simulation model development in certain high-level languages such as Python, MATLAB®, or Simulink® are unparalleled by their convenience and rapid turnover time. However, legacy simulation engines often depend on more traditional languages such as FORTRAN or C/C++. The NASA Marshall Aerospace Vehicle Representation in C version II (MAVERIC-II) is a modular, legacy-derived computer program used for high-fidelity, 6 degree-of-freedom (6DOF) simulation for aerospace vehicle flights and analyses of guidance and control performance with built-in mathematical modeling of environmental effects such as wind, atmosphere, and gravity as well as dispersion capability for Monte Carlo analysis. MAVERIC-II is modular in the sense that each component software element of the simulation engine may be supplanted for a higher or lower fidelity version. The design flow of the development of these models is often performed in high-level languages as mentioned previously, which must then be translated into C or C++ code to be integrated into MAVERIC-II. Using principles of model-based design, we propose a unified method of auto-coding and interfacing between several languages and MAVERIC-II, which may be generalized further to any type of 6DOF simulation engine.

ABOUT THE AUTHORS

Mason Nixon grew up in Wetumpka, AL. He graduated Summa Cum Laude in 2011 from Auburn University in Electrical Engineering and Wireless Electrical Engineering and was also ranked first in his class. Mr. Nixon then went on to receive his Master's degree from the Georgia Institute of Technology in Electrical Engineering with a focus in Control Systems and Human Automation with a minor in Computer Science in 2013. For the first 6 years of his professional career, Mr. Nixon worked at the United States Army Space and Missile Defense Command / Army Forces Strategic Command where he supported mission analyses through simulation as well as design and development for the four SNaP satellite missions, the Kestrel Eye satellite program, and other small satellite missions. He concluded his tenure at SMDC as the Principal Investigator and Program Manager for the Space Lab. In support of the Space Lab, he oversaw a pilot program, called ACES RED, which has the goal of building a low-cost small satellite testing platform. He and his team built, tested, and delivered flight hardware that was launched on a Space X Falcon 9 mission to attach to the International Space Station in the spring of 2019. He currently works as a Senior Guidance, Navigation, Control, and Communications Engineer with Leidos, Inc. and is currently in pursuit of a Ph.D. in Control Systems at the University of Alabama in Huntsville.

Developing a Multilingual Auto-coding Interface Control for the MAVERIC-II Dynamics Simulator

Mason Nixon
Leidos, Inc.
Huntsville, AL
mason.e.nixon@nasa.gov

HISTORY / BACKGROUND

During the late 1990s, through needs derived from the X-33 technology demonstrator program, through NASA's Marshall Spaceflight Center (MSFC), Jim McCarter developed the simulation tool, Marshall Aerospace Vehicle Representation in C (MAVERIC) (M. Hanson et al., 1998; Orr et al., 2014). MAVERIC is a high-fidelity 3 / 6 DOF nonlinear time domain simulation tool with the ability to quickly simulate a vehicle's launch or flight, including all environmental, propulsion, aerodynamic, and any other disturbance forces that it encounters throughout its mission. The initial version of MAVERIC lacked generality and modularity to serve as a general purpose simulation tool and so a subsequent version was developed to support NASA's Space Launch Initiative (SLI) program (McCarter, 2011). The new version incorporated the object-oriented simulation kernel (OSK), the C++ Model Developer (CMD), and the Tools to Facilitate the Rapid Assembly of Missile Engagement Simulations (TFRAMES) architectures developed by Ray Sells (Sells, 1995, 2017; Sells et al., 2001) and was called MAVERIC-II. Since the original X-33 version had become obsolete, the new version became known as simply MAVERIC. The program continued on to support many other NASA initiatives such as all of the Ares vehicles, specifically the built and tested Ares-I X, Lockheed Martin and Boeing's Orbital Space Plane (OSP) programs, the X-43C, the Lunar Lander, launch vehicle concepts, the more recent Space Launch System (SLS), and many others (Ahmad et al., 2019; J. M. Hanson & Hall, 2008; Jang et al., 2010, 2008; McCarter, 2011; Shtessel et al., 2010; Vanzwieten & Johnson, 2014).

The primary uses for MAVERIC are for guidance, navigation, and control (GN&C) development and evaluation of various vehicle designs and mission profiles, with a primary focus on GN&C algorithms for use in flight software (Orr et al., 2014). For example, in (Hall & Shtessel, 2006), Hall and Shtessel perform a comparison between the X-33 autopilot algorithm and a novel sliding mode controller (SMC) and a sliding mode disturbance observer (SMDO). The power of MAVERIC lies in the ability to quickly simulate parameter variations that allows for Monte Carlo simulations of 10s of thousands of scenarios in short order to be able to characterize a vehicle's performance in a variety of scenarios. Coupled with this feature is the ability to develop flight software in a low-level coding language (such as C, C++) and be able to rigorously test it in simulation, but also to then extract the same code for hardware-in-the-loop simulations, and perhaps even for use on the actual flight hardware (Orr et al., 2014). This enables better configuration control over the software and a means of quickly evaluating flight software changes without having to test on the actual flight hardware, which can be dangerous and expensive.

In order to facilitate the use of MAVERIC and better represent actual flight-like conditions, high-fidelity models must be developed for all of the relevant forces and perturbations that may be applied to the vehicle. Wind models, gravity models, vehicle sensors, actuators, lag times, etc. all must be accounted for and the degree to which they are modeled is typically a trade-off between computation time and the resultant error in measurements for a given mission scenario. As an example, during a fast maneuver, perhaps vehicle fuel sloshing must be modeled to examine coupling effects in the vehicle's attitude dynamics. Modeling these effects requires significant computation and so in a slow maneuver, modeling these phenomena becomes less of a priority. Notwithstanding, the development of each of these models requires their own software development cycles and validation. Oftentimes, these evaluations occur in environments better suited to a containerized development, for example by using MATLAB®, Simulink®, or Python. Using techniques gleaned from the field of model-based design, we may ease many of the pain points of designing, developing, and integrating these models into the MAVERIC simulation and ensure fast, accurate, and repeatable development cycles.

Model-Based Design

Widely promulgated by the MathWorks, Inc., model-based design is a method of development workflow for the creation of dynamic systems to better facilitate the V-model systems engineering process wherein model designs are linked directly to design requirements (The MathWorks, 2021b). A key benefit of the model-based design methodology is that the developed model may be run for simulation or system-level testing at any step of the development process (Bergmann, 2014). The power in this is derived from the fact that from an initial feasibility study, to simulation, to prototyping, and verification; all may leverage the same model. Several other benefits include (Bergmann, 2014):

- Consistent Documentation & Implementation
- Elimination / Reduction of Coding Translation Errors
- Continuous Verification & Validation
- Reusable Code
- Automatic Code Generation

The model-based design paradigm consists of three primary steps (The MathWorks, 2021a). These steps include: first identifying the components to be designed and the design goals, then performing analysis of the system behavior via simulation, and finally component design and verification. With a sufficient interface control definition, these steps may be iterated at any point in the later design phases with little to no system-level software changes; only the code for the model itself needs to be modified.

The use of model-based design has been widely adopted within the aerospace industry, from large-scale projects, such as NASA's SLS (GN&C) (Ahmad et al., 2019; Oliver et al., 2018) to smaller- to medium-scaled design problems (Bergmann, 2014; Kelemenová et al., 2013; Ruff et al., 2012). From low-complexity, to highly complex systems, model-based design gives the design engineer a powerful procedure that enables a cost-effective and efficient means of system development. Note that although much of the literature focuses on implementations using Simulink® or MATLAB®, for generalization, we seek to extrapolate these methods to other toolsets, such as Python.

PROBLEM STATEMENT

The MAVERIC simulation tool, and particularly the object-oriented CMD OSK framework provides for a suitable use case for the model-based design approach. Specifically, we have a platform that allows for the creation of low- to high-fidelity models. MAVERIC allows for modularity of each subsystem model where flight code, model code, etc. may be compiled into separately executable code, which thus enables separate development cycles for the system-level code versus the individual components. Focusing on the principle of automatic code generation, in order to create a fast, repeatable, and accurate method, we propose an auto-coding strategy that leverages a multilingual model-based design approach.

INTERFACE CONTROL

Since several branches and versions of the MAVERIC code base are currently being used in the simulation community, we try to make the descriptions of the architecture as generic as possible to remain relevant to a broader audience.

MAVERIC Architecture

The program is laid out in a file structure (See Figure 1) that separates the MAVERIC simulation-specific code from vehicle-specific code with a respective **m** or **v**. Each mission scenario is broken out in to a simulation case that has its own specific input files and source code. These cases can be selected at runtime upon start-up of the program executable (typically called *mav*) in a command-line interface (CLI). Data that is configured to be output will show up in the console window as the simulation runs and written to output files called PLOT files.

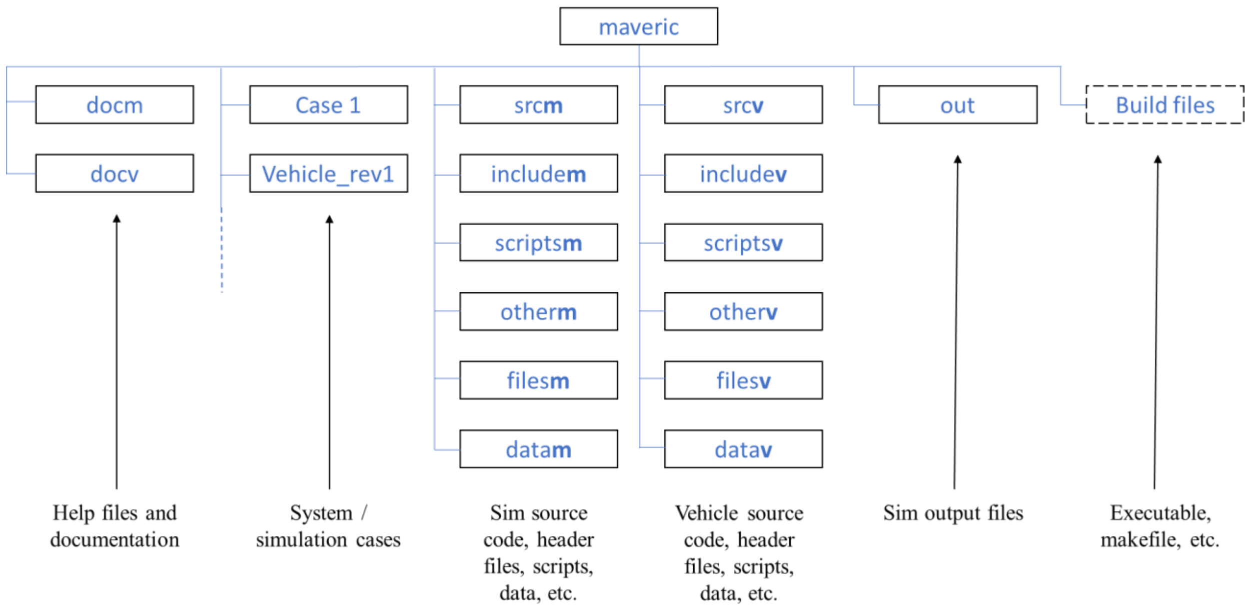

maveric
docm
docv
Case 1
Vehicle_rev1
srcm
includem
scriptsm
otherm
filesm
datam
srcv
includev
scriptsv
otherv
filesv
datav
out
Build files
Help files and documentation
System / simulation cases
Sim source code, header files, scripts, data, etc.
Vehicle source code, header files, scripts, data, etc.
Sim output files
Executable, makefile, etc.

Figure 1. MAVERIC File Structure

Model Interface

Although there are many different models that may be developed or configured as mentioned previously, we will focus specifically on vehicle models for sensors and actuators for the purposes of this assessment, however, the principles and solutions developed should easily extrapolate to any model within the MAVERIC framework. The basic structure for these models is defined as in the OSK CMD architecture (Sells, 2017) where the model object is defined by three methods: read, init(ialize), update.

The read() method, or function, is used to gather up static model parameters and information defined within text-based DAT files. The init() method is intended to determine the current state of the model based on a global parameter that synchronizes state information between different models throughout the entire simulation. The update() function is where one may define the dynamics of the model and where the reporting of its respective output data occurs. This data flow is described in Figure 2.

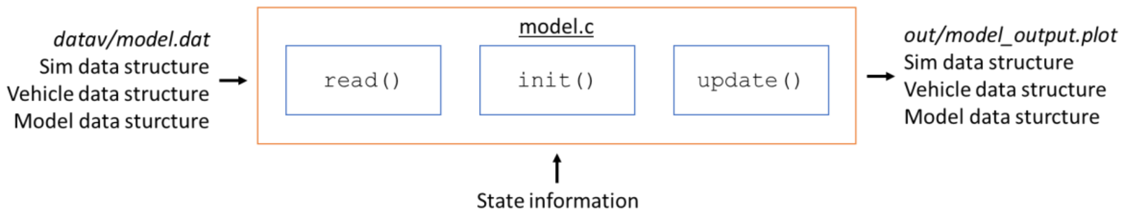


Figure 2. Model Data Flow

Since we aim to create an auto-coding interface, there are several other file changes and configurations that must occur in order to be able to access the new model. For instance, the model header file must be added to the includev/models.h and the associated read, init, and update functions must be declared within includev/plugins.incl and defined in datav/plugin.dat. Additionally, any mission-specific or case-specific data files must be called out in *system*/mission.dat and *system*/cases.dat, respectively, where *system* is the relevant case or vehicle system directory. For any desired outputs, these must be specified within *system*/output.dat and datav/outputlist.dat.

To create a clearer requirement for the auto-coding, we can further decompose the problem into developing the model dynamics code. Unfortunately, due to the variety of potential model types, this leaves open the need for building up the associated data structures and interfaces mentioned above in a manual process. Nonetheless, to alleviate in the development of the model algorithms and maintain their configuration control, we still may provide this benefit, in addition to reducing overall development times.

DESIGN PROCESS

In this section, we will explore the process of model development including defining the system to be modeled, describing the inputs and outputs of the model, discussing the level of fidelity to meet different objectives, and finally, the implementation, verification, and validation.

System Definition

Simulation is a means of recreating aspects of the real, physical world. We can describe the dynamics of the world (i.e., a dynamical system) using differential equations and we can decompose systems into lumped element models. A dynamical system can be simply defined as a system that changes over time and is subject to both its own internal evolution and/or external stimuli. In order to construct these models, we must first define the states of the phenomena we are modeling, then define any driving parameters, and finally we must define or describe how the model evolves over time, i.e., describe the state derivatives.

A system state is defined as a set of variables whose values, when combined with input signals and dynamical equations of the system can be used to determine the output state of the system at some future time (Dorf & Bishop, 1990). A very simple example, say for the single-dimensional motion of a unit mass, where the position and velocity states are $x_1 = x,\ x_2 = \dot{x}_1$, respectively, can be represented by the following nonlinear dynamics.

$$\boldsymbol{x_1 = x_1(t) = x(t), \qquad x_2 = x_2(t) = \frac{dx_1(t)}{dt} = \dot{x}_1} \qquad (1)$$

$$\boldsymbol{\rightarrow \begin{cases} \dot{x}_1 = x_2, & x_1(0) = x_{10} \\ \dot{x}_2 = u(x_1, x_2, t) + f(x_1, x_2, t), & x_2(0) = x_{20} \end{cases}}$$

where $u = u(x_1, x_2, t)$ is some input control force, $f = f(x_1, x_2, t)$ is some external force such as dry and/or viscous friction, and x_{10} and x_{20} are the associated initial conditions for the states, x_1 and x_2, respectively. This system of equations in (1) contains the relevant states, x_1 and x_2, describes how the system evolves over time (i.e., the state derivatives, $\dot{x}_1$ and $\dot{x}_2$), and specifies the driving parameters, which in this case are a control force input, u, and a disturbance force, f.

Inputs/Outputs

Once we have described the system dynamics, we need to determine what relevant external inputs and outputs our model should manipulate. For instance, the inputs in (1) are the input forces, u, and the outputs could be the position, velocity, and perhaps the friction force over time. The outputs could be any information about the system desired by the operator of the simulation.

Sampling Rate

Since we are discretizing the dynamics of the system (i.e., sampling), we must also select a sufficiently large sampling rate in order to observe whatever relevant phenomena we seek to observe. An example could be a system that describes an audio signal of voice data that is being captured. In order to sufficiently recreate the voice data, we can cite the Nyquist-Shannon sampling theorem (Nyquist, 1928; Shannon, 1949), which states that we must sample the signal at a rate that is at least twice the highest frequency of the data that was sampled to allow for perfect reconstruction. Typically voice data is between the threshold of human hearing, which is roughly 20 Hz to 20 kHz. So, in this case, we must sample faster than 2×20k, or 40k samples per second.

Implementation, Verification, & Validation

Once we have identified the relevant system inputs, outputs, and described the states and how they evolve over time, we can attempt to implement the model in computer software. Simulink® offers a convenient toolset for developing continuous-time and discrete-time models with built-in numerical integration to allow for fast prototyping and simulation capability. The block-based nature of this method is intuitive to control systems engineers, wherein block diagrams are common tools of the practice. Models can be strung together by simply connecting functional blocks and data streams. Even very complex models can be built up from the basic blocks that are provided in the base software. Some limitations of using Simulink® to describe a model include the high cost of the tool itself, the inability to natively (without additional tools or toolboxes) generate code from the model to a low-level language supported by MAVERIC, the inability to easily use a custom-defined numerical integration technique, and the general closed-source nature of the software make customization a bit more cumbersome.

Another method for developing models leverages Ray Sell's OSK toolset (Sells, 2017) in a programming language such as Python. In this case, both the OSK and Python are open source. Additionally, the Python language provides a multitude of user support online with a large variety of no-cost add-ons and modules. One can easily incorporate any numerical integration into the framework and the modular nature of the OSK lends itself very well to the development of complex systems of systems. Limitations of this method are related to the general lack of speed of the Python language (as compared to lower-level languages such as C/C++, FORTRAN, or assembly), no graphical method for implementing models, and the lack of dedicated support for the framework and programming environment.

Of course, the model could be written directly in C or C++, or even MATLAB® code, but in this case the development time is typically increased and suitable tools for assessing the model are not readily available as is the case with the above methods.

Regardless of which method of implementation one choses, following the tenets of model-based design, the model must then be verified and validated. Verification is the process under which the model is evaluated to determine whether it is compliant with given requirements or specifications (NASA, 2016). For example, say that we must develop an engine model that allows us to measure a motor rotational velocity. A verification would assess whether or not a given model does in fact provide an engine model with an output of rotor shaft velocity. Validation is an assurance that the model fulfills its intended use (NASA, 2016). So, for the motor example, perhaps we are simulating the engine to determine its operating frequency for a modal analysis. In this case, the model must be developed in such a way that the dynamical equations result in a determination of the operating frequencies of the engine.

For a MAVERIC simulation, we must have a suitable model to incorporate into the simulation software that is written in C and some C++. Therefore, whether we chose Simulink®, Python, MATLAB®, or some other method in the design phase, we must ultimately end up with C/C++ code. As mentioned, another principle of model-based design leverages the power of auto-coding. In the next section, we examine a case study for the development of a system model using several methods and then describe the means by which we can automatically produce C/C++ code to be incorporated into MAVERIC.

CASE STUDY

To clarify the design methods outlined in the previous section, we now will work through a relevant example, or case study. As noted in (Trumper & Dubowsky, 2005), second-order systems (or systems where the highest derivative is of order two) represent electrical, mechanical, thermal, fluidic systems with two energy storage elements. Through simplifying assumptions, in the linear case, higher-order systems can very often be approximated by using a second-order system. A good example of this for a servo drive pulley system is studied in (Hoyt, 2009). Therefore, we will make use of this fact by developing a model for a generic second-order system and subsequently defining methods for auto-coding to meet our goal of generating MAVERIC-compatible model code. The canonical form for a second-order oscillator can be described as

$$\boldsymbol{\frac{d^2x}{dt^2} + 2\zeta\omega\frac{dx}{dt} + \omega^2 x = \omega^2 u} \tag{2}$$

where $0 \leq \zeta \leq 1$ is the damping coefficient, $\omega = 2\pi f$ is the frequency of oscillation (radians), and f is the frequency in Hertz. This system can then be transformed into a state-based representation for $x_1 = x,\ \dot{x}_1 = \dot{x} = x_2$ as

$$\begin{gathered}\dot{x}_2 + 2\zeta\omega\dot{x}_1 + \omega^2 x_1 = \omega^2 u \rightarrow \\ \begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}0 & 1\\ -\omega^2 & -2\zeta\omega\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + \begin{bmatrix}0\\ \omega^2\end{bmatrix} u\end{gathered} \tag{3}$$

This system will be the basis for our modeling using both code-based and block-based methods in the subsequent sections.

To define the multi-lingual interface control, our resultant auto-coding output must fit in to the data flow represented in Figure 2. The read() function is paired with an associated data (.DAT) file for the model. This data file is populated with relevant initial condition data. This approach is useful since, instead of having to recompile source code, we can simply update the data file to calibrate our model. For example, if we are defining an environment model that describes the atmospheric density for a given altitude, we can use a MAVERIC data table to encapsulate this data, as described in (Sells, 1995). For our simple second-order oscillator, the data for input using the read() function can be the values for ω, ζ, and the initial values for our two states.

The init() function can then define any parameters needed for initialization of the model. Note that both this function and the read() function are only executed once at the beginning of the MAVERIC simulation. This would be a good place to assign the initial values for the states that we read in using the read() function.

And finally, the update() function is where we should define the dynamics of the model, i.e., how our model evolves over time. This is also where we should assign to any variables we would like to designate as simulation output variables. Two variables to assess our second-order oscillator are time and the state, x_1, so these would make for appropriate output variables. The subsequent sections describe the auto-coding methods we can use to create the update() function for our model.

Block-Based Methods

A common tool for developing models that may be simulated is MathWorks' Simulink®. There are a multitude of workflows one can use to generate code output from a Simulink® model depending on each user's specific use case. Since we are concerned with integration into MAVERIC, we can narrow these workflows a bit. We can additionally impose the constraint that we want to minimize the number of additional toolboxes, add-ons, etc. in order to achieve this goal. Note that there may be better workflows for our use case available and they could possibly save more development time in the long-run, but for the sake of brevity, we will explore only one workflow that will leverage only MATLAB®, Simulink®, and the Simulink Coder™. The interested reader may reference (Fraticelli, 2012; Vikström, 2009).

First, we must construct the model. See Figure 3.

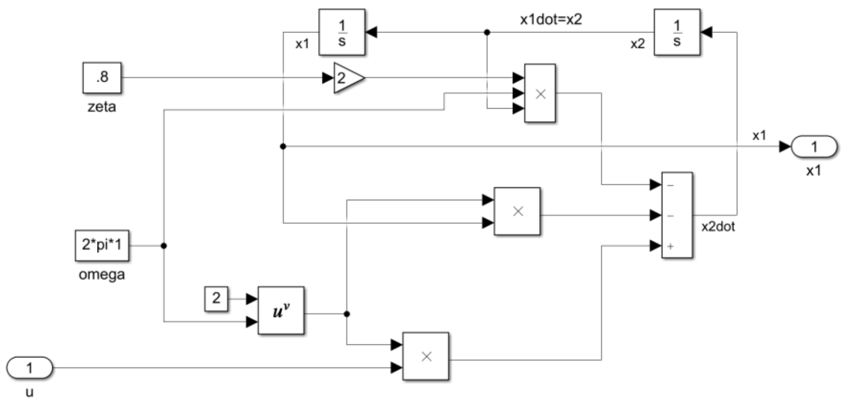


Figure 3. Second-order Oscillator Block Diagram

The model may be verified and validated at this point using any given input and inspecting its output. For $\zeta = 0.8$, since we have $0 < \zeta < 1$, according to systems theory, we have an underdamped system and thus the system output should have some degree of overshoot from our setpoint. A validation thus entails assurance that the model does in fact result in such a response. We can apply a step function input to step the system from 0 up to 1. The result is shown in Figure 4, which is indeed what we expect for an underdamped system. If this is indeed the system response we seek, then the model has also been verified to be compliant with our needs.

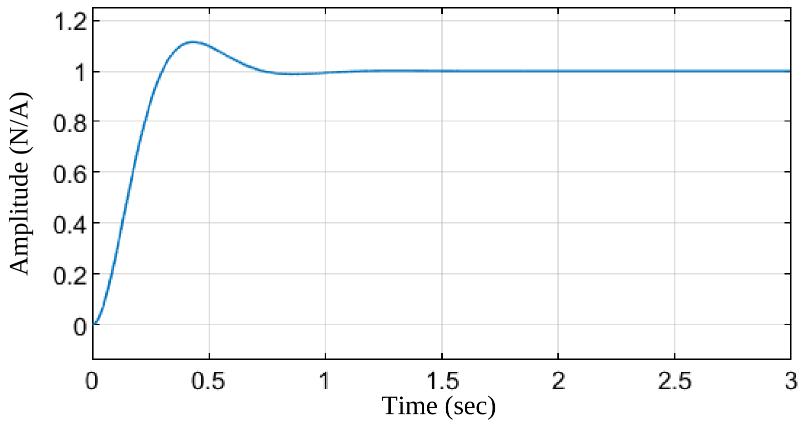
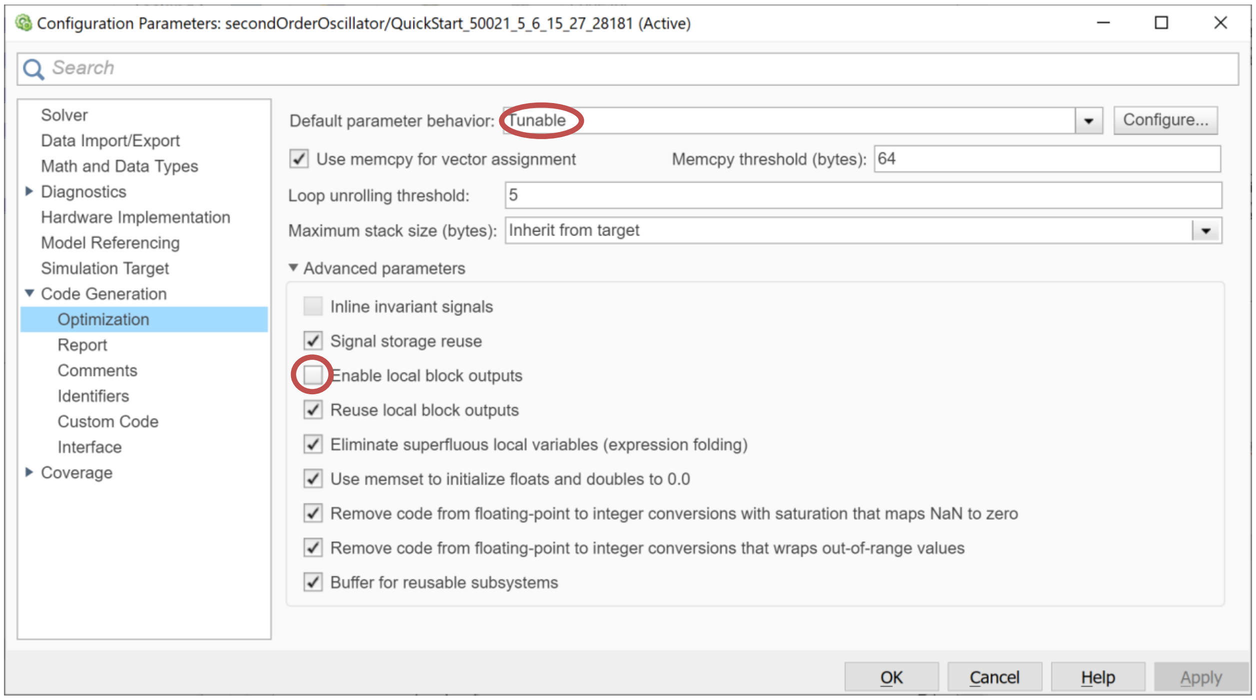


Figure 4. Step Input System Response

After we have our model constructed with an input and output port, we can then start the process to generate code with the code generation app. Note that many tutorials exist that have been published by MathWorks, Inc. on the process and should likely be consulted for the best approach since many variations of this process exist depending on the particular version of MATLAB® being used. For reference, we are using version R2019b. In order to have non-fixed values assigned to our gain parameters and constants, the default parameter behavior in the model configuration parameters should be changed from "Inline" to "Tunable." Additionally, the "Enable local block outputs" should be unchecked. Afterwards, the code may be generated (See Figure 5).



Configuration Parameters: secondOrderOscillator/QuickStart_50021_5_6_15_27_28181 (Active)

Default parameter behavior: Tunable

- [✓] Use memcpy for vector assignment — Memcpy threshold (bytes): 64
- Loop unrolling threshold: 5
- Maximum stack size (bytes): Inherit from target

Advanced parameters

- [ ] Inline invariant signals
- [✓] Signal storage reuse
- [ ] Enable local block outputs
- [✓] Reuse local block outputs
- [✓] Eliminate superfluous local variables (expression folding)
- [✓] Use memset to initialize floats and doubles to 0.0
- [✓] Remove code from floating-point to integer conversions with saturation that maps NaN to zero
- [✓] Remove code from floating-point to integer conversions that wraps out-of-range values
- [✓] Buffer for reusable subsystems

Figure 5. Code Generation Configuration Parameters

The code output is created within a file named after your model with the .c extension, e.g. ours is called secondOrderOscillator.c. This code is split into three functions: initialize, step, and terminate. We can take the step function code and use this in our MAVERIC update() function. In order to use this code directly, certain data types and headers are required for their direct use. In our case, we need secondOrderOscillator.h, and secondOrderOscillator_private.h, and their associated dependencies. By default, all of these dependencies should be included in the same directory as the C file and headers. To ensure MAVERIC can read all of these files appropriately, they should be placed in the associated subdirectories. For example, the headers for the model should be placed in the includev directory as described in INTERFACE CONTROL. A helpful hint in making this work properly is to leverage the Makefile that is generated by the coder as a reference for the files that must be linked and included in the compilation.

The model may be modified as needed within Simulink® and further verified as needs are refined throughout the development process and the code may then be easily regenerated into MAVERIC code by repeating the process described above. In the next section, we will explore high-level (e.g., MATLAB® code or Python code) code-based methods for low-level (e.g., C) code generation.

Code-Based Methods

Note that model-based design is typically used synonymously with Simulink® model design, but we can further generalize this to refer to mathematically defined models coded in high-level (relative to C/C++ code) languages such as MATLAB® or Python. Benefits of this approach are the ability to rapidly code algorithms as well as increased readability, even for non-coders due to the simplicity of the coding language. We will walk through an example using Python here, although the results may be extrapolated to be used in MATLAB® or some other language. We will also take advantage of Ray Sell's open source OSK Python code, which has the added benefit of very similar construction to MAVERIC code, as is described in (Sells, 2017).

Again, our first step is to construct the model. The OSK framework is extremely modular and object-oriented from the bottom up. To create a model in the Python version of OSK, we merely must define a class with four functions: __init__(), init(), update(), and rpt(). Note the similarities in construction to the MAVERIC model flow described in Figure 2. Taking the dynamics described in (3), our model definition in class form in Python can be written as described below in Figure 6 (Sells, 2021).

```
class SecondOrder( Block): # oscillator
  def __init__( self): # constructor, executed when object created
    Block.__init__( self) # boiler-plate to use osk internals
    self.x1 = self.addIntegrator() # create integrators
    self.x2 = self.addIntegrator()

  def init( self): # executed at start of simulation
    self.x1[0] = 0.0 # index 0 is state
    self.x2[0] = 0.0 # index 0 is state
    self.u = 1.0 # for step input
    # frequency of oscillation will be wn * sqrt( 1 - zeta^2)
    self.wn = 2.0 * math.pi * 1.0 # 1 Hz
    self.zeta = .8

  def update( self): # executed each time step as sim. executes
    u = self.u # shorthands for more compact expression
    wn = self.wn
    zeta = self.zeta
    self.x1[1] = self.x2[0] # index 1 is derivative
    self.x2[1] = ( u - self.x1[0]) * wn * wn - 2.0 * zeta * wn * self.x2[0]

  def rpt( self): # executed each time step as sim. executes
      print( "hello_2nd_Order %8.3f %8.6f" % ( Sim.clock.t, self.x1[0]))
      outDict['t'].append(Sim.clock.t)
      outDict['x1'].append(self.x1[0])
```

Figure 6. Python Second-Order Oscillator Class (Sells, 2021)

As was the case with the block-based methods, we can continuously verify the model with various inputs and parameters. For the step input, we see the results in Figure 7. Note that the results depend on the sampling time for the integrator used as well as which integrator is selected.

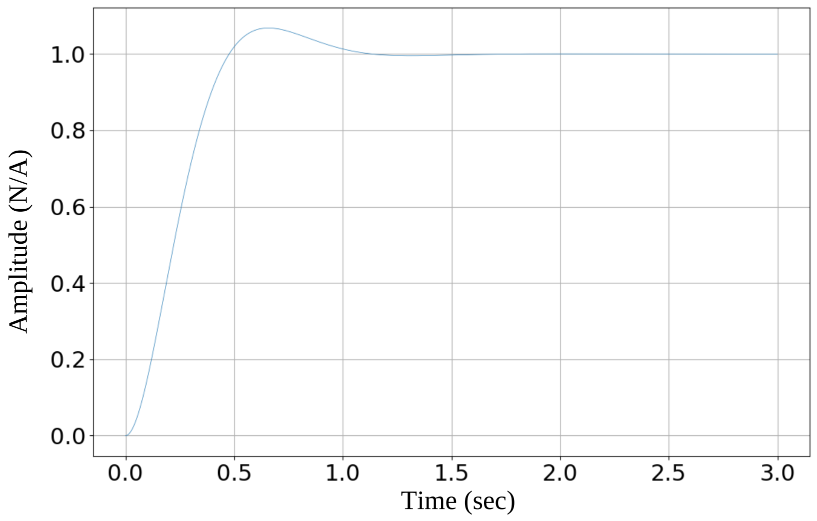


Figure 7. Python Second-Order Oscillator Step Response

This result is consistent with what we achieved in the block-based model, which for our purposes, serves as a sufficient validation. The next step is the auto-coding.

Similar to the Simulink® potential workflows, due to the open source nature of the Python programming language, many options exist for the development of MAVERIC models in the C language. Since Python 2 is no longer supported as of January 2020, we can reduce the number of options to Python 3 support. Additionally, we can restrict our options to methods leveraging the least number of third-party dependencies or modules. Fortunately, Python 3 supports a method for achieving just such a task using a process termed *Pure Embedding* (Python Software Foundation, 2021). This method allows us to directly embed the Python interpreter into our C code and thus, execute any arbitrary Python code. We can then take our update() function that has been validated into its own .py script file and wrap it in the C caller described in (Python Software Foundation, 2021), incorporate the Python.h header file, and then perform the compilation and linking.

For a given installation version X.Y of Python (on a Unix-based system – which is typically where a MAVERIC instance is being compiled), we can determine the appropriate compile flags using the command:

```
$ /opt/bin/pythonX.Y-config --cflags
-I/opt/include/pythonX.Ym  -I/opt/include/pythonX.Ym  -DNDEBUG  -g  -fwrapv  -O3  -Wall  -Wstrict-
prototypes
```

Similarly, the linker flags can be determined using the command:

```
$ /opt/bin/pythonX.Y-config --ldflags
-L/opt/lib/pythonX.Y/config-X.Ym -lpthread -ldl -lutil -lm -lpythonX.Ym -Xlinker -export-dynamic
```

These options can then be incorporated into the MAVERIC make_makefile script. Any subsequent changes made to the model update function can be made directly within the Python script; the Makefile generation process does not need to be performed again, since the C code we already generated calls the Python script directly.

A similar approach may be taken for MATLAB® scripts by structuring the code into the functions discussed previously and shown in Figure 2. This interface control is thus extendable to any language or development environment with C code generation functionality where the process can be encoded in a similar functional form as the examples we have assessed.

CONCLUSIONS

In summation, we have described an interface control for auto-code generation for use with two different software platforms: Simulink® and Python. Depending on the availability of auto-code generation software, the methods presented could conceivably be extrapolated to other software languages and platforms. Additionally, although we present these methods as applied towards the MAVERIC-II simulator, any software-based simulation environment with a similar construction could leverage these model-based engineering methods. These methods enable utilization of the principles of model-based design and allow for faster design that is continuously verifiable and able to be re-/validated and re-/verified at any development stage as well as modular and consistent throughout the design process.

REFERENCES

Ahmad, N., Anzalone, E., Kim, Y., Von der Porten, P., Compton, J., Hough, S., Park, T., Wall, J., & Garcia, C. (2019). SLS GNC Model-based Design Approach. *NASA MSFC Annual Flight Software Workshop*, 1–12.

Bergmann, A. (2014). Benefits and Drawbacks of Model-based Design. *KMUTNB International Journal of Applied Science and Technology*, *7*(3), 15–19. https://doi.org/10.14416/j.ijast.2014.04.004

Dorf, R. C., & Bishop, R. H. (1990). *Modern Control Systems* (6th ed.). Prentice Hall.

Fraticelli, J. C. M. (2012). *Simulink Code Generation*.

Hall, C. E., & Shtessel, Y. B. (2006). Sliding Mode Disturbance Observer-Based Control for a Reusable Launch Vehicle. *Journal of Guidance, Control, and Dynamics*, *29*(6), 1315–1328. https://doi.org/10.2514/1.20151

Hanson, J. M., & Hall, C. E. (2008). Learning about ares I from Monte Carlo simulation. *AIAA Guidance, Navigation and Control Conference and Exhibit*, 1–19. https://doi.org/10.2514/6.2008-6622

Hanson, M., Coughlin, D. J., Mulqueen, J. A., & Mccarter, J. W. (1998). Ascent, Transition, Entry, and Abort Guidance Algorithm Design for the X-33 Vehicle. *AIAA Guidance, Navigation, and Control Conference*, 1–11.

Hoyt, C. (2009). *Developing equivalent second order system models and robust controls for servo-drive related systems*. University of Arkansas.

Jang, J.-W., Alaniz, A., Hall, R. A., Bedrossian, N. S., Hall, C. E., Ryan, S., & Jackson, M. (2010). Ares I Flight Control System Design. *Technical Paper*, *August*, 1–14. https://doi.org/10.2514/6.2010-8442

Jang, J.-W., Hall, R., Bedrossian, N., & Hall, C. (2008). Ares-I Bending Filter Design Using A Constrained Optimization Approach. *AIAA GNC Conference*, 1–16. https://doi.org/10.2514/6.2008-6289

Kelemenová, T., Kelemen, M., Miková, Ľ., Maxim, V., Prada, E., Lipták, T., & Menda, F. (2013). *Model Based Design and HIL Simulations*. *1*(7), 276–281. https://doi.org/10.12691/ajme-1-7-25

McCarter, J. (2011). *MAVERIC Training Sessions*. DESE Research, Inc.

NASA. (2016). NASA System Engineering Handbook Revision 2. *National Aeronautics and Space Administration*, 297. https://www.nasa.gov/sites/default/files/atoms/files/nasa_systems_engineering_handbook_0.pdf

Nyquist, H. (1928). Certain Topics in Telegraph Transmission Theory. *Transactions of the American Institute of Electrical Engineers. Reprint as Classic Paper in: Proc. IEEE, Vol. 90, No. 2, Feb 2002*, *47*(2), 617–644. https://doi.org/10.1109/T-AIEE.1928.5055024

Oliver, T. E., Anzalone, E., Park, T., & Geohagan, K. (2018). SLS model based Design: A navigation perspective. *Advances in the Astronautical Sciences*, *164*, 981–993.

Orr, J. S., Wall, J. H., & Hall, C. E. (2014). Space Launch System Ascent Flight Control Design. *American Astronautical Society (AAS) Guidance, Navigation, and Control Conference*.

Python Software Foundation. (2021). *Embedding Python in Another Application*. Python 3 Docs. https://docs.python.org/3/extending/embedding.html

Ruff, R., Stephens, C., & Mahapatra, S. (2012). Applying model-based design to large-scale systems development: Modeling, simulation, test, & deployment of a multirotor vehicle. *AIAA Modeling and Simulation Technologies Conference 2012*, *August*, 1–17. https://doi.org/10.2514/6.2012-4939

Sells, R. (1995). Missile Six Degree-of-freedom Simulation Development: A User-focused Approach. *Society for Computer Simulation Summer Computer Simulation Conference*.

Sells, R. (2017). A code architecture to streamline the missile simulation life cycle. *AIAA Modeling and Simulation Technologies Conference*. https://doi.org/10.2514/6.2017-1768

Sells, R. (2021). *Python OSK Training Session*. DESE Research, Inc.

Sells, R., Salter, A., Edgemon, D., Arsenal, R., & Al, H. (2001). Toward Establishing an Architectural Standard for Dynamical System Simulations. *Society for Computer Simulation Summer Computer Simulation Conference*,

1–7.
Shannon, C. E. (1949). Communication in the presence of noise. *Proceedings of the Institute of Radio Engineers. Reprint as Classic Paper in: Proc. IEEE, Vol. 86, No. 2, (Feb 1998)*, *37*(1), 10–21. https://doi.org/10.1109/JPROC.1998.659497
Shtessel, Y., Hall, C. E., Baev, S., & Orr, J. (2010). Flexible Modes Control Using Sliding Mode Observers: Application to Ares I. *AIAA Guidance, Navigation, and Control Conference*, *August*, 1–22.
The MathWorks, I. (2021a). *Design a System in Simulink*. https://www.mathworks.com/help/simulink/gs/design-a-system-in-simulink.html
The MathWorks, I. (2021b). *Model-Based Design with Simulink*. https://www.mathworks.com/help/simulink/gs/model-based-design.html
Trumper, D., & Dubowsky, S. (2005). *Modeling Dynamics and Control I: Second-order Systems*. MIT OpenCourseWare. https://ocw.mit.edu/courses/mechanical-engineering/2-003-modeling-dynamics-and-control-i-spring-2005/
Vanzwieten, T., & Johnson, M. (2014). *Time and Frequency-Domain Cross-Verification of SLS 6DOF Trajectory Simulations*. 0–1.
Vikström, A. (2009). *A study of automatic translation of MATLAB code to C code using software from the MathWorks*. 1–51.