Simulink® Coder™

Getting Started Guide

MATLAB®&SIMULINK®

R2019b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Simulink® Coder™ Getting Started Guide

Revision History

April 2011	Online only	New for Version 8.0 (Release 2011a)
September 2011	Online only	Revised for Version 8.1 (Release 2011b)
March 2012	Online only	Revised for Version 8.2 (Release 2012a)
September 2012	Online only	Revised for Version 8.3 (Release 2012b)
March 2013	Online only	Revised for Version 8.4 (Release 2013a)
September 2013	Online only	Revised for Version 8.5 (Release 2013b)
March 2014	Online only	Revised for Version 8.6 (Release 2014a)
October 2014	Online only	Revised for Version 8.7 (Release 2014b)
March 2015	Online only	Revised for Version 8.8 (Release 2015a)
September 2015	Online only	Revised for Version 8.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 8.10 (Release 2016a)
September 2016	Online only	Revised for Version 8.11 (Release 2016b)
March 2017	Online only	Revised for Version 8.12 (Release 2017a)
September 2017	Online only	Revised for Version 8.13 (Release 2017b)
March 2018	Online only	Revised for Version 8.14 (Release 2018a)
September 2018	Online only	Revised for Version 9.0 (Release 2018b)
March 2019	Online only	Revised for Version 9.1 (Release 2019a)
September 2019	Online only	Revised for Version 9.2 (Release 2019b)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Contents

Product Overview

Simulink Coder Product Description

Generate C and C++ code from Simulink and Stateflow models

Simulink Coder (formerly Real-Time Workshop®) generates and executes C and C++ code from Simulink models, Stateflow® charts, and MATLAB® functions. The generated source code can be used for real-time and non-real-time applications, including simulation acceleration, rapid prototyping, and hardware-in-the-loop testing. You can tune and monitor the generated code using Simulink or run and interact with the code outside MATLAB and Simulink.

Key Features

- ANSI/ISO C and C++ code and executables for discrete, continuous, or hybrid Simulink and Stateflow models
- Integer, floating-point, and fixed-point data types using row- and column-major layout
- Code generation for single-rate, multirate, and asynchronous models
- Single-task, multitask, and multicore code execution with or without an RTOS
- External mode simulation for parameter tuning and signal monitoring using XCP, TCP/IP, and serial communication protocols
- Incremental and parallel code generation builds for large models

Code Generation by Using Simulink Coder

In this section...

Code Generation Technology

MathWorks® code generation technology produces C or C++ code and executable programs for algorithms. You can write algorithms programmatically by using MATLAB or graphically in the Simulink environment. You can generate code for MATLAB functions and Simulink blocks that are useful for real-time and embedded applications. Generated source code and executable programs for floating-point algorithms match the functional behavior of MATLAB code execution and Simulink simulations to a high degree of fidelity. Using the Fixed-Point Designer product, you can generate fixed-point code that provides a bitwise match to model simulation results. Such broad support and high degree of accuracy are possible because code generation is tightly integrated with the MATLAB and Simulink execution and simulation engines. The built-in accelerated simulation modes in Simulink use code generation technology.

Code generation technology and related products provide tooling that you can apply to the V-model for system development. The V-model is a representation of system development that highlights verification and validation steps in the development process. For more information, see "Validation and Verification for System Development" on page 1-6.

To learn about model design patterns that include Simulink blocks, Stateflow charts, and MATLAB functions, and map to commonly used C constructs, see "Modeling Patterns for C Code" (Embedded Coder).

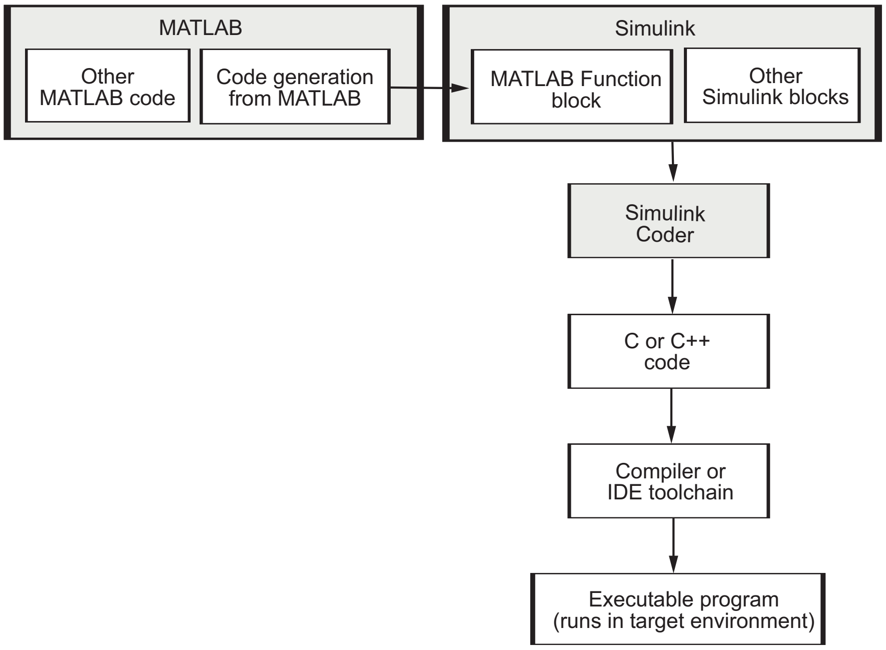
Code Generation Workflow by Using Simulink Coder

Use MathWorks code generation technology to generate standalone C or C++ source code for rapid prototyping, simulation acceleration, and hardware-in-the-loop (HIL) simulation:

- By developing Simulink models and Stateflow charts, and then generating C/C++ code from the models and charts by using the Simulink Coder product

- By integrating MATLAB code for code generation in MATLAB Function blocks in a Simulink model, and then generating C/C++ code by using the Simulink Coder product

You can generate code for most Simulink blocks and many MathWorks products on page 1-3. This figure shows the product workflow for code generation by using Simulink Coder. Other products that support code generation, such as Stateflow software, are available.

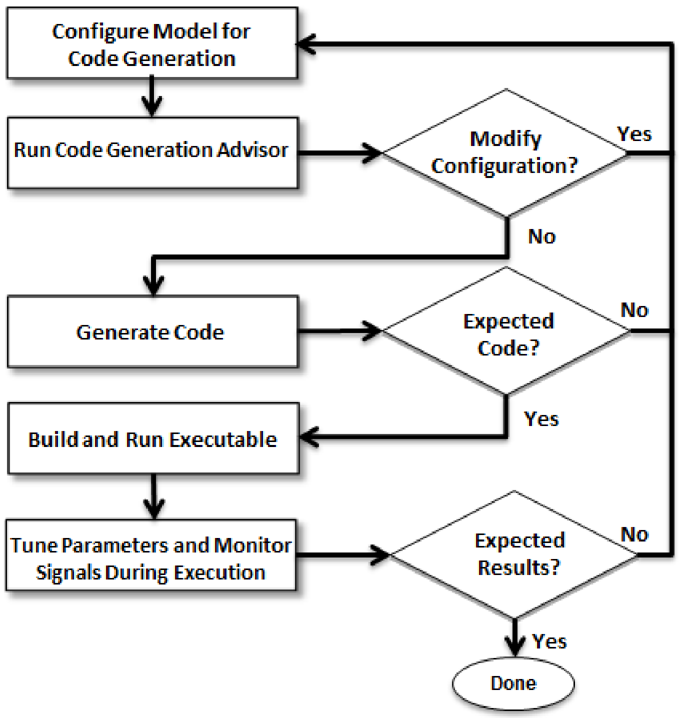


The code generation workflow is a part of the V-model on page 1-6 for system development. The process includes code generation, code verification, and testing of the executable program in real-time. For rapid prototyping of a real-time application, typical tasks are:

- Configure the model for code generation in the model configuration set.
- Check the model configuration for execution efficiency using the Code Generation Advisor.
- Generate and view the C code.
- Create and run the executable of the generated code.

- Verify the execution results.
- Build the target executable.
- Run the external model target program.
- Connect Simulink to the external process for testing.
- Use signal monitoring and parameter tuning to further test your program.

Here is a typical workflow for applying the software to the application development process.


Configure Model for Code Generation
Run Code Generation Advisor
Modify Configuration?
Yes
No
Generate Code
Expected Code?
No
Yes
Build and Run Executable
Tune Parameters and Monitor Signals During Execution
Expected Results?
No
Yes
Done

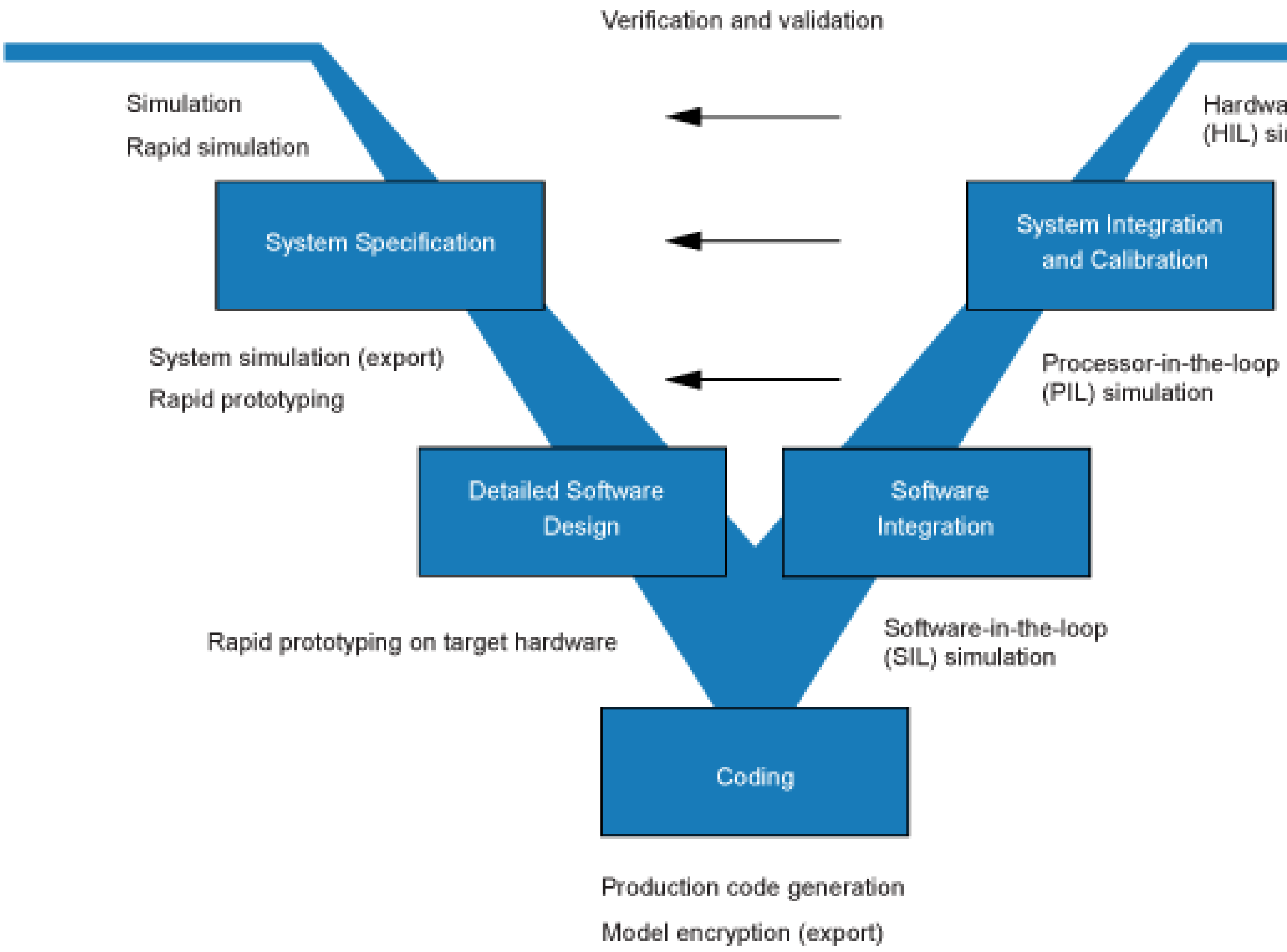
For more information on how to perform these tasks, see “Generate C Code for a Model” on page 2-2.

Validation and Verification for System Development

An approach to validating and verifying system development is the V-model.

V-Model for System Development

The V-model is a representation of system development that highlights verification and validation steps in the system development process. The left side of the 'V' identifies steps that lead to code generation, including system specification and detailed software design. The right side of the V focuses on the verification and validation of steps cited on the left side, including software and system integration.


Verification and validation
Simulation
Rapid simulation
System Specification
System simulation (export)
Rapid prototyping
Detailed Software Design
Rapid prototyping on target hardware
Coding
Production code generation
Model encryption (export)
Software-in-the-loop (SIL) simulation
Software Integration
Processor-in-the-loop (PIL) simulation
System Integration and Calibration
Hardwa
(HIL) si

Depending on your application and its role in the process, you might focus on one or more of the steps called out in the V-model or repeat steps at several stages of the V-model. Code generation technology and related products provide tooling that you can apply to the V-model for system development. For more information about how you can apply MathWorks code generation technology and related products to the V-model process, see "Types of Simulation and Prototyping in the V-Model" on page 1-8.

Types of Simulation and Prototyping in the V-Model

This table compares the types of simulation and prototyping identified on the left side of the V-model diagram shown in “V-Model for System Development” (Embedded Coder).

	Simulation	Rapid Simulation	System Simulation, Rapid Prototyping	Rapid Prototyping on Target Hardware
Purpose	Test and validate functionality of concept model	Refine, test, and validate functionality of concept model in nonreal time	Test new ideas and research	Refine and calibrate design during development process
Execution hardware	Development computer	Development computer Standalone executable runs outside of MATLAB and Simulink environments	PC or nontarget hardware	Embedded computing unit (ECU) or near-production hardware
Code efficiency and I/O latency	Not applicable	Not applicable	Less emphasis on code efficiency and I/O latency	More emphasis on code efficiency and I/O latency

	Simulation	Rapid Simulation	System Simulation, Rapid Prototyping	Rapid Prototyping on Target Hardware
Ease of use and cost	Can simulate component (algorithm or controller) and environment (or plant) Normal mode simulation in Simulink enables you to access, display, and tune data during verification Can accelerate Simulink simulations	Easy to simulate models of hybrid dynamic systems that include components and environment models Ideal for batch or Monte Carlo simulations Can repeat simulations with varying data sets, interactively or programmatically by using scripts, without rebuilding the model Can connect to Simulink to monitor signals and tune parameters	Might require custom real-time simulators and hardware Might be done with inexpensive, off-the-shelf PC hardware and I/O cards	Might use existing hardware for less expense and more convenience

Target Environments and Applications

About Target Environments

The code generator produces make or project files to build an executable program for a specific target environment. The generated make or project files are optional. If you prefer, you can build an executable program for the generated source files by using an existing target build environment, such as a third-party integrated development environment (IDE). Applications of generated code range from calling a few exported C or C++ functions on a development computer to generating a complete executable program that uses a custom build process for custom hardware, in an environment completely separate from the development computer running MATLAB and Simulink.

The code generator provides built-in system target files that generate, build, and execute code for specific target environments. These system target files offer varying degrees of support for interacting with the generated code to log data, tune parameters, and experiment with or without Simulink as the external interface to your generated code.

Types of Target Environments

Before you select a system target file, identify the target environment on which you expect to execute your generated code. The most common target environments include environments listed in this table.

Target Environment	Description
Development computer	The computer that runs MATLAB and Simulink. A development computer is a PC or UNIX®[a] environment that uses a non-real-time operating system, such as Microsoft® Windows® or Linux®[b]. Non-real-time (general purpose) operating systems are nondeterministic. For example, those operating systems might suspend code execution to run an operating system service and then, after providing the service, continue code execution. Therefore, the executable for your generated code might run faster or slower than the sample rates that you specified in your model.
Real-time simulator	A different computer from the development computer. A real-time simulator can be a PC or UNIX environment that uses a real-time operating system (RTOS), such as: • Simulink Real-Time system • A real-time Linux system • A Versa Module Eurocard (VME) chassis with PowerPC® processors running a commercial RTOS The generated code runs in real time. The exact nature of execution varies based on the particular behavior of the system hardware and RTOS. A real-time simulator connects to a development computer for data logging, interactive parameter tuning, and Monte Carlo batch execution studies.
Embedded microprocessor	A computer that you eventually disconnect from a development computer and run as a standalone computer as part of an electronics-based product. Embedded microprocessors range in price and performance, from high-end digital signal processors (DSPs) to process communication signals to inexpensive 8-bit fixed-point microcontrollers in mass production (for example, electronic parts produced in the millions of units). Embedded microprocessors can: • Use a full-featured RTOS • Be driven by basic interrupts • Use rate monotonic scheduling provided with code generation

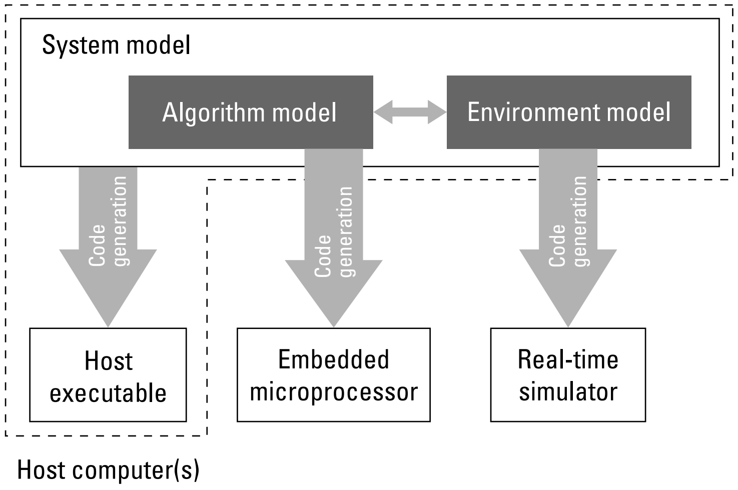
a. UNIX is a registered trademark of The Open Group in the United States and other countries.
b. Linux is a registered trademark of Linus Torvalds.

A target environment can:

- Have single- or multiple-core CPUs
- Be a standalone computer or communicate as part of a computer network

You can deploy different parts of a Simulink model on different target environments. For example, it is common to separate the component (algorithm or controller) portion of a model from the environment (or plant). Using Simulink to model an entire system (plant and controller) is often referred to as closed-loop simulation and can provide many benefits, such as early verification of a component.

The following figure shows example target environments for code generated for a model.



Applications of Supported Target Environments

This table lists ways that you can apply code generation technology in the context of the different target environments.

Application	Description
Development Computer	

Application	Description
"Acceleration" (Simulink)	Techniques to speed up the execution of model simulation in the context of the MATLAB and Simulink environments. Accelerated simulations are especially useful when run time is long compared to the time associated with compilation and checking whether the target is up to date.
Rapid Simulation	Execute code generated for a model in non-real-time on the development computer, but outside the context of the MATLAB and Simulink environments.
Shared Object Libraries (Embedded Coder)	Integrate components into a larger system. You provide generated source code and related dependencies for building a system in another environment or in a shared library to which other code can dynamically link.
"Protect Models to Conceal Contents"	Generate a protected model for use by a third-party vendor in another Simulink simulation environment.
Real-Time Simulator	
Real-Time Rapid Prototyping	Generate, deploy, and tune code on a real-time simulator connected to the system hardware, for example, physical plant or vehicle. being controlled. Crucial for validating whether a component can control the physical system.
Shared Object Libraries (Embedded Coder)	Integrate generated source code and dependencies for components into a larger system that is built in another environment. You can use shared library files for intellectual property protection.

Application	Description
Hardware-in-the-Loop (HIL) Simulation	Run a simulation that pairs physical hardware, such as a controller, with a virtual real-time implementation of physical components on a real-time target computer, including a plant, sensors, actuators, and the environment. Use HIL simulations to test and validate physical hardware and a controller algorithm by including the effects of component response in real time to realistic stimuli. Testing commonly compares the HIL simulation results to system requirements. Validation compares HIL simulation results to user requirements. Often HIL simulations are referred to as closed-loop simulations due to the component response to the physical environment stimuli.
Embedded Microprocessor	
“Code Generation” (Embedded Coder)	From a model, generate code that is optimized for speed, memory usage, simplicity, and compliance with industry standards and guidelines.
“Software-in-the-Loop Simulation” (Embedded Coder)	Compile generated or external source code intended for production and execute the code as a separate process from the rest of the Simulink model on your development computer. Goals include initial source code testing and verification by comparing SIL and model simulation results or comparing SIL results to requirements by using back-to-back testing. Commonly used with external code integration, bit-accurate fixed-point math, and coverage analysis.

Application	Description
“Processor-in-the-Loop Simulation” (Embedded Coder)	Cross-compile generated or external source code intended for production on a development computer, and then download and run the object code on a target processor or an equivalent instruction set simulator. Goals include verification by comparing PIL simulation results against model or SIL simulation results and collecting execution time profiling data. Commonly used with external code integration, bit-accurate fixed-point math, and coverage analysis.
Hardware-in-the-loop (HIL) Simulation	Run a simulation that pairs physical hardware, such as a controller, with a virtual real-time implementation of physical components on a real-time target computer, including a plant, sensors, actuators, and the environment. Use HIL simulations to test and validate physical hardware and a controller algorithm by including the effects of component response in real time to realistic stimuli. Testing commonly compares the HIL simulation results to system requirements. Validation compares HIL simulation results to user requirements. Often HIL simulations are referred to as closed-loop simulations due to the component response to the physical environment stimuli.

Getting Started Tutorials

Generate C Code for a Model

To generate C or C++ code from Simulink models, Stateflow charts, and MATLAB functions, use the Simulink Coder product. Use the generated code in applications such as simulation acceleration, rapid prototyping, and hardware-in-the-loop (HIL) simulations.

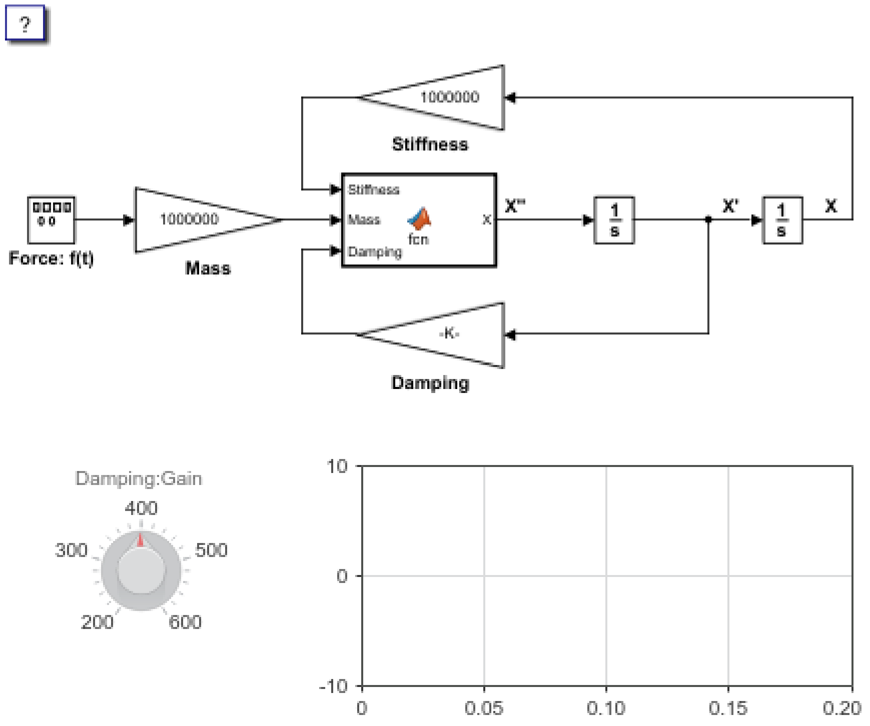
If you are new to Simulink Coder or your application code customization requirements are minimal, you can use graphical tools and default model configuration settings to quickly generate code.

Generating and reviewing code can be as simple as preparing the model for code generation with the Quick Start tool. Then, using code tools accessible from the Simulink Editor, you can configure data interfaces, initiate code generation, and verify the generated code.

This tutorial uses example model `rtwdemo_secondOrderSystem`.

Open the model by entering the model name in the Command Window.

```
rtwdemo_secondOrderSystem
```



The model implements a second-order physical system called an ideal mass-spring-damper system. Gain blocks represent components of the system equation: `Mass`, `Stiffness`, and `Damping`. The equation for the system is $mX'' + cX' + kX = f(t)$.

- m = mass of the system (1.0E-6 kg)
- c = damping ratio (4.0e-4 Ns/m)
- k = spring stiffness (1.0 N/m)
- $f(t)$ = forcing function in the x-direction (N)

A Signal Generator block injects a square wave form with an amplitude of 4 and frequency of 20 Hz. The block uses simulation time as the source of values for the waveform time variable. Because the model is configured with a fixed-step solver, which is required for code generation, Simulink uses the same step size for an entire simulation. The consistent step size provides a uniformly sampled representation of the ideal waveform.

The example model shows how you can use MATLAB Function blocks to integrate existing MATLAB function code into Simulink models from which you can generate embeddable C code. The MATLAB function block in the example model integrates a MATLAB function that computes the sum of the component variables.

The Integrator blocks compute integrals of the MATLAB Function block output with respect to time. The solver computes the output of the Integrator block at the current time step, by using the current input value and the value of the state at the previous time step. To support this computational model, the Integrator block saves its output at the current time step for use by the solver to compute its output at the next time step. The block also provides the solver with an initial condition for use in computing the block's initial state at the beginning of a simulation. The default initial condition and the setting for this example model is 0.

The dashboard blocks, Knob and Dashboard Scope, provide visual tooling for tuning the damping and monitoring the waveform. The Knob block is connected to the `Damping` Gain block. The Dashboard Scope block connects to signals `Force: f(t):1` and `X`.

You use this model to learn how to:

1. Generate code by using the Simulink Coder Quick Start tool.
2. Verify whether generated executable program results match simulation results.
3. Tune a parameter during program execution.
4. Deploy prototype code and artifacts.

To start the tutorial, see “Generate Code by Using Simulink Coder Quick Start Tool” on page 2-5.

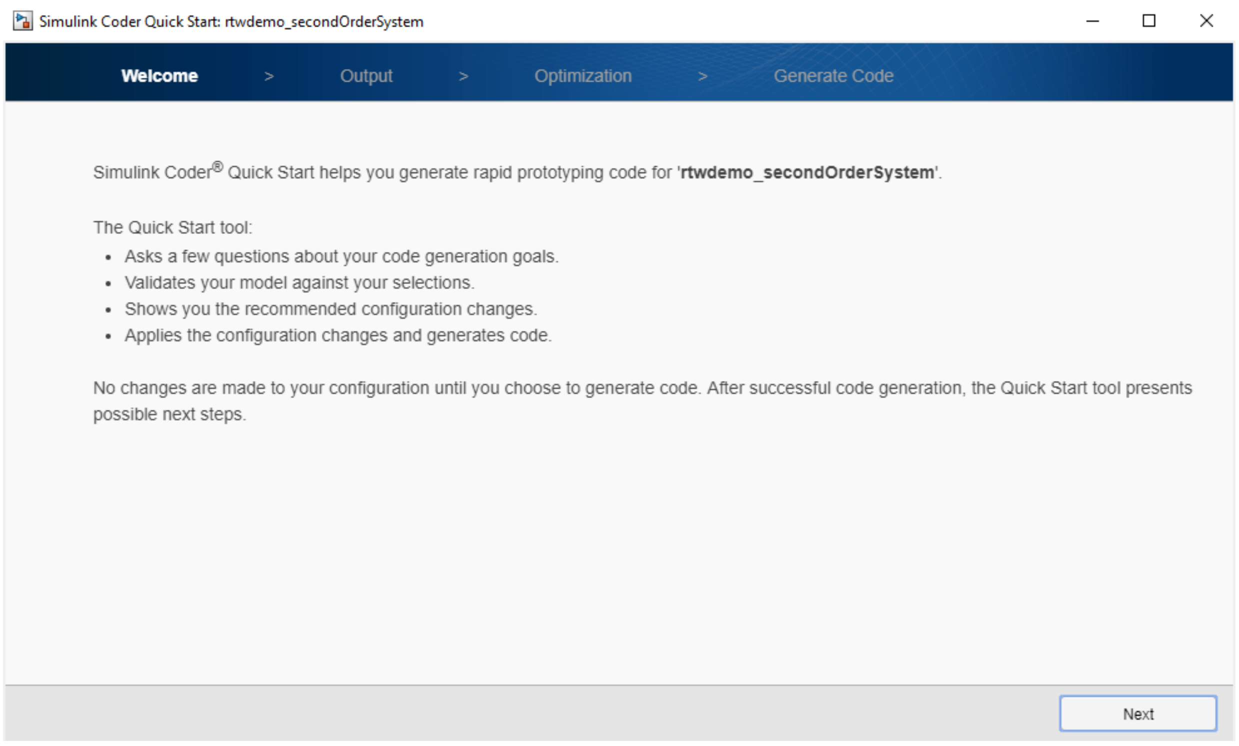
Generate Code by Using Simulink Coder Quick Start Tool

Prepare model `rtwdemo_secondOrderSystem` for code generation and generate C89/C90 compliant C code by using the Simulink Coder Quick Start tool. Then, inspect the generated code.

Generate C Code with Quick Start Tool

The Quick Start tool chooses fundamental code generation settings based on your goals and application. For example, the Quick Start tool configures the model with a fixed-step solver, which is required for code generation.

1. Open model `rtwdemo_secondOrderSystem` by entering the model name in the Command Window.
2. Save a copy of the model to a writeable location on the MATLAB path.
3. If the **C Code** tab is not already open, in the Apps gallery, under **Code Generation**, click **Simulink Coder**.
4. Open the Simulink Coder Quick Start tool. On the **C Code** tab, click **Quick Start**.


Simulink Coder Quick Start: rtwdemo_secondOrderSystem
Welcome > Output > Optimization > Generate Code
Next

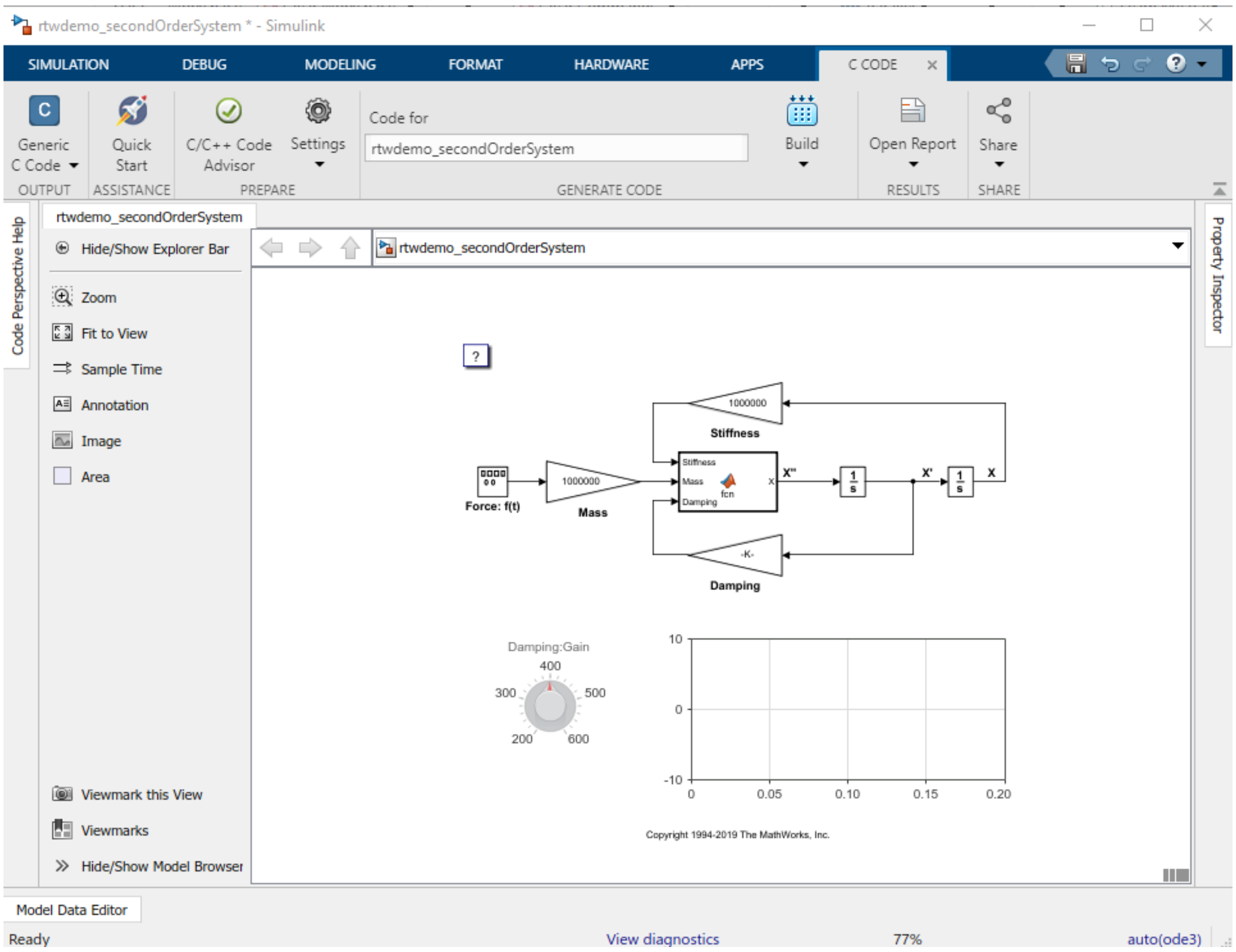
Simulink Coder® Quick Start helps you generate rapid prototyping code for '**rtwdemo_secondOrderSystem**'.

The Quick Start tool:

- Asks a few questions about your code generation goals.
- Validates your model against your selections.
- Shows you the recommended configuration changes.
- Applies the configuration changes and generates code.

No changes are made to your configuration until you choose to generate code. After successful code generation, the Quick Start tool presents possible next steps.

5. Advance through the steps of the Quick Start tool. Each step asks questions about the code that you want to generate. For this tutorial, use the default settings. The tool validates your selections against the model and presents the parameter changes required to generate code.
6. In the **Generate Code** step, apply the proposed changes and generate code from the model by clicking **Next**.
7. Click **Finish**. In the Simulink Editor, return to the **C Code** tab. Configure code generation customizations, regenerate code, and check results in the code generation report.



Inspect the Generated Code

The code generator creates folder `rtwdemo_secondOrderSystem_ert_rtw` in your current working folder and places source code files in that folder. The generated code is in two primary files: `rtwdemo_secondOrderSystem.c` and `rtwdemo_secondOrderSystem.h`. The file `rtwdemo_secondOrderSystem.c` contains the algorithm code, including the ODE solver code. Model data and entry-point functions are accessible to a caller by including `rtwdemo_secondOrderSystem.h`. The `rtwdemo_secondOrderSystem.h` file includes the `extern` declarations for block outputs, continuous states, model output, entry points, and timing data.

In your current folder, the code generator creates an `slprj` folder. This folder contains the file `rtwtypes.h`, which defines standard data types that the generated code uses by default. This sibling folder contains generated files that can or must be shared between multiple models.

The code that you generate from a model includes entry-point functions, which you call from application code, such as an external main program. For a rate-based model, these functions include an initialization function, an execution function, and, optionally, terminate and reset functions. The functions exchange data with your application code through a data interface that you control.

1. Open the code generation report. In the **C Code** tab, click **Open Report**.
2. Open the **Code Interface Report** section. Review the entry-point functions that the code generator produces for the model. For the initialize, execution (step), and terminate functions, the code generator uses these names:

   - `rtwdemo_secondOrderSystem_initialize`
   - `rtwdemo_secondOrderSystem_step`
   - `rtwdemo_secondOrderSystem_terminate`

   The functions have a `void-void` interface, which means that they do not pass arguments. The functions gain access to data through shared data structures. Examples of such data include system-level input and output that the functions exchange with application code.
3. Review the entry-point functions in the generated code. In the left pane of the code generation report, under **Generated Code**, click file name `rtwdemo_secondOrderSystem.c`. Use the **Find** field to find instances of the string `secondOrderSystem_step`. Use the arrows to the right of the **Find** field to step through each instance. Do the same for the header file `rtwdemo_secondOrderSystem.h`. Then, review code for the initialize and terminate functions.

Next, verify whether model simulation results match generated executable program results.

Verify Generated Executable Program Results

Verify whether generated executable program results for the model match simulation results.

Configure Model for Verification

1. Configure the Dashboard Scope block to monitor the values of signals `Force: f(t): 1` and `X`. Double-click the Dashboard Scope block. In the Block Parameters dialog box, confirm that:

   - The block is connected to signals `Force: f(t): 1` and `X`. To connect a Dashboard block to a signal, in the model canvas, select the signal. In the Block Parameters dialog box, select the signal name.
   - **Min** is set to `-10`.
   - **Max** is set to `10`.

   Apply changes and close the dialog box.
2. Configure the Knob block so that you can use the knob to change the value of the damping gain. Double-click the Knob block. In the Block Parameters dialog box, confirm that:

   - The block is connected to parameter `Damping:Gain`. To connect a Dashboard block to a parameter, in the model canvas, select the block that uses the parameter. In the Block Parameters dialog box, select the parameter name.
   - **Minimum** is set to `200`.
   - **Maximum** is set to `600`.
   - **Tick Interval** is set to `100`.

   Apply changes and close the dialog box.
3. Open the Model Configuration Parameters dialog box. On the **C Code** tab, click **Settings**.
4. Configure the model such that Simulink and the generated executable program log workspace data in the Simulation Data Inspector. Click **Data Import/Export**. Confirm that the model is configured with these settings:

Parameter Selected	Name Set To
Time	`tout`
States	`xout`
Output	`yout`
Signal logging	`logsout`
Data stores	`dsmout`
Record logged workspace data in Simulation Data Inspector	

5 Configure the model for building an executable program. Click **Code Generation**. Confirm that parameter “Generate code only” is cleared.

6 Configure and validate the toolchain for building the executable program. Confirm that parameter “Toolchain” is set to `Automatically locate an installed toolchain`. Then, search for and click the **Validate Toolchain** button. The Validation Report indicates whether the checks passed.

7 Configure parameters and signals so that the data is stored in memory and is accessible while the the executable program runs. To efficiently implement a model in C code, you do not allocate memory for every parameter, signal, and state in the model. If the model algorithm does not require data to calculate outputs, code generation optimizations eliminate storage for data. To allocate storage for the data so that you can access it during prototyping, you disable some optimizations.

   Click **Code Generation** > **Optimization**. Confirm that:

   - “Default parameter behavior” is set to `Tunable`. With this setting, block parameters, such as the **Gain** parameter of a Gain block, are tunable in the generated code.
   - “Signal storage reuse” is cleared. With this setting, the code generator allocates storage for signal lines. While running the executable program, you can monitor the values of these signals.

8 Configure the code generator to produce nonfinite data (for example, `NaN` and `Inf`) and related operations. Click **Code Generation** > **Interface**. Confirm that parameter “Support: non-finite numbers” is selected.

9 Configure a communication channel. For Simulink® to communicate with an executable program generated from a model, the model must include support for a communication channel. This example uses XCP on TCP/IP as the transport layer for a communication channel. Confirm these parameter settings:

- “External mode” is selected.
- “Transport layer” is set to `XCP on TCP/IP`. This selection specifies `ext_xcp` for parameter **Mex-file name**.
- “Static memory allocation” is selected. You cannot clear this parameter.
- “Static memory buffer size” specifies the amount of XCP slave memory that is allocated for signal logging.

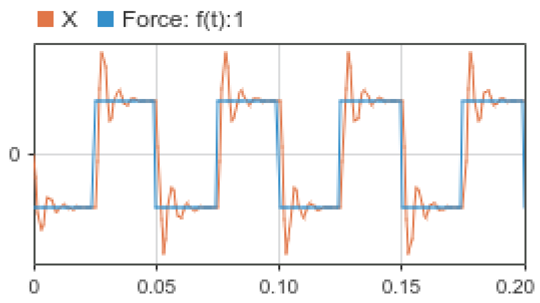
10 Disable MAT-file logging. Load the data into the Simulation Data Inspector from the MATLAB base workspace. Confirm that parameter “MAT-file logging” is cleared.

11 Apply your configuration changes, close the Model Configuration Parameters dialog box, and save the model.

Simulate Model and View Results

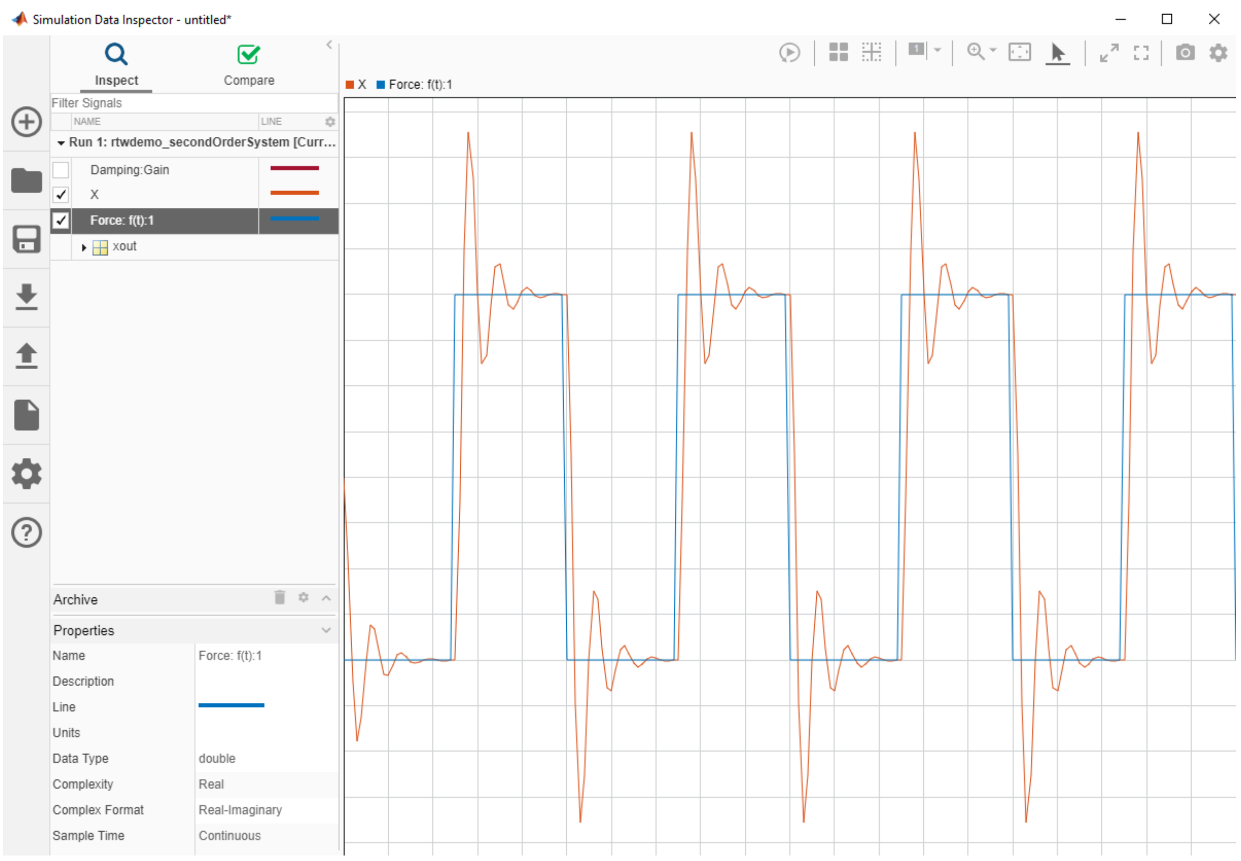
1 From the Simulink Editor, in the **Simulation** tab, click **Run**. The clock on the **Run** button indicates that simulation pacing is enabled. Simulation pacing slows down a simulation so that you can observe system behavior. Visualizing simulations at a slower rate makes it easier to understand the underlying system design and identify design issues while demonstrating near real-time behavior.

   During the simulation, the Dashboard Scope block displays the behavior of signals `Force: f(t):1` and `X`.



2 In the Simulink Editor, in the **Simulation** tab, click **Data Inspector**. The Simulation Data Inspector opens with data from your simulation run imported.

3 Expand the run (if not already expanded). Then, to plot the data, select data signals `X` and `Force: f(t):1`.



Name	Force: f(t):1
Description	
Line	
Units	
Data Type	double
Complexity	Real
Complex Format	Real-Imaginary
Sample Time	Continuous

Leave these results in the Simulation Data Inspector. Later, you compare the simulation data to the output data produced by the executable program generated from the model.

Build and Run Executable Program and View Results

Use the Monitor and Mapping tool to build and run the model executable program.

1 Click **Monitor & Tune**. Simulink:

   a Builds the executable program. During the build process `Building` appears on the bottom-left corner of the Simulink Editor window. When the code generation report appears and the text reads `Ready`, the process is complete.

- In Windows, the code generator creates and places these files in your current working folder:
  - `rtwdemo_secondOrderSystem.exe` – Executable program file
  - `rtwdemo_secondOrderSystem.pdb` – Debugging symbols file for parameters and signals
- In Linux, the code generator creates and places DWARF format debugging information in the ELF executable program file, `rtwdemo_secondOrderSystem`, and places the file in your current working folder.

b Deploys the executable program as a separate process on your development computer.

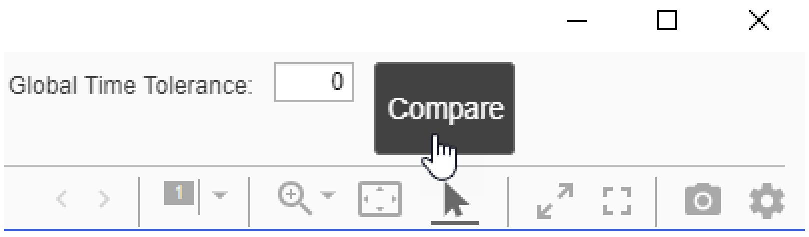
c Connects the Simulink model to the executable program.

d Runs the model executable program code.

Compare Simulation and Executable Program Results

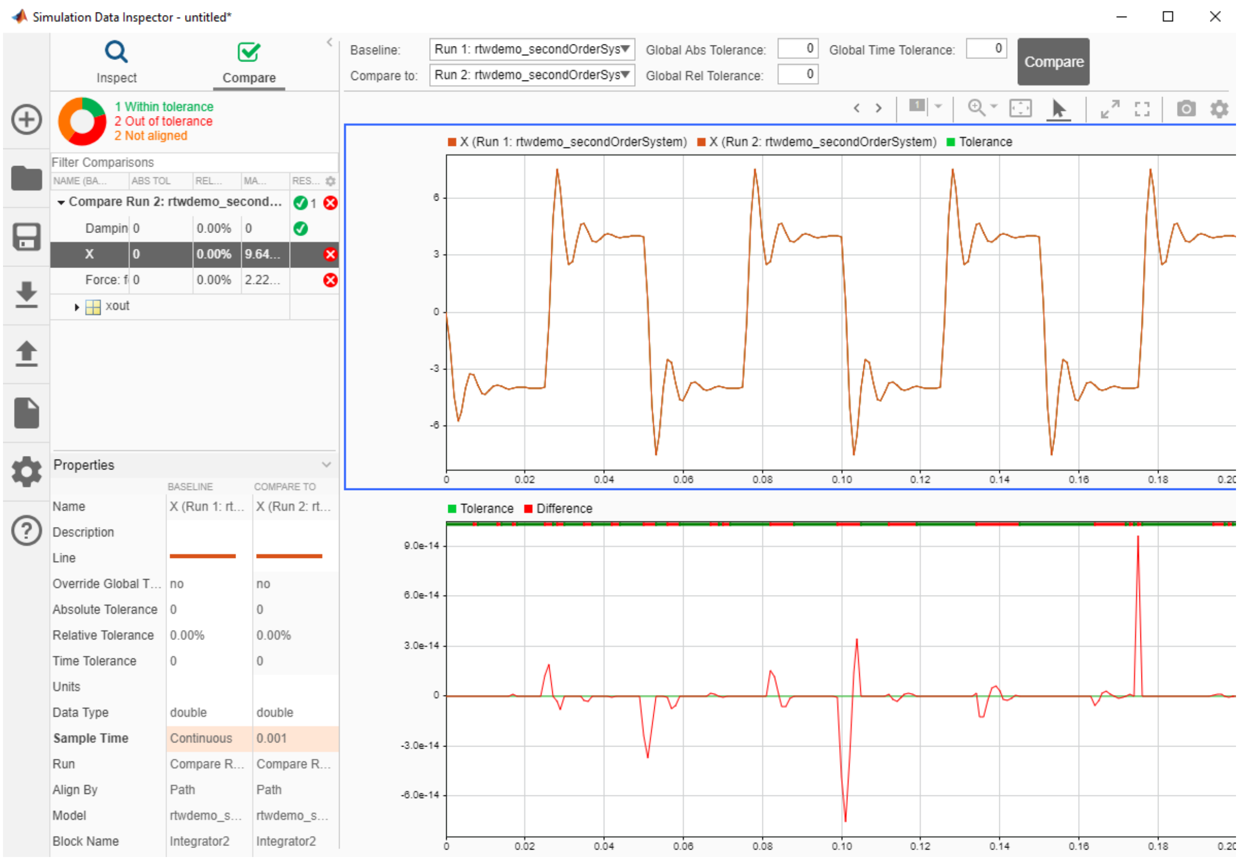
Use the Simulation Data Inspector to compare the executable program results with the simulation results.

1. In the Simulation Data Inspector, inspect the results of your executable program run, `Run 2: rtwdemo_secondOrderSystem`.
2. Click **Compare**.
3. Select the data runs that you want to compare. For this example, from the **Baseline** list, select `Run 1: rtwdemo_secondOrderSystem`. From the **Compare to** list, select `Run 2: rtwdemo_secondOrderSystem`.
4. In the upper-right corner of the Simulation Data Inspector, click **Compare**.



5. The Simulation Data Inspector indicates that the output for `X` and `Force: f(t):1` from the executable program code is out of tolerance from the simulation data

output. To see a plot of the results for X, under **File Comparisons**, select the row for X.


Simulation Data Inspector - untitled*

Baseline: Run 1: rtwdemo_secondOrderSys
Compare to: Run 2: rtwdemo_secondOrderSys
Global Abs Tolerance: 0
Global Rel Tolerance: 0
Global Time Tolerance: 0

1 Within tolerance
2 Out of tolerance
2 Not aligned

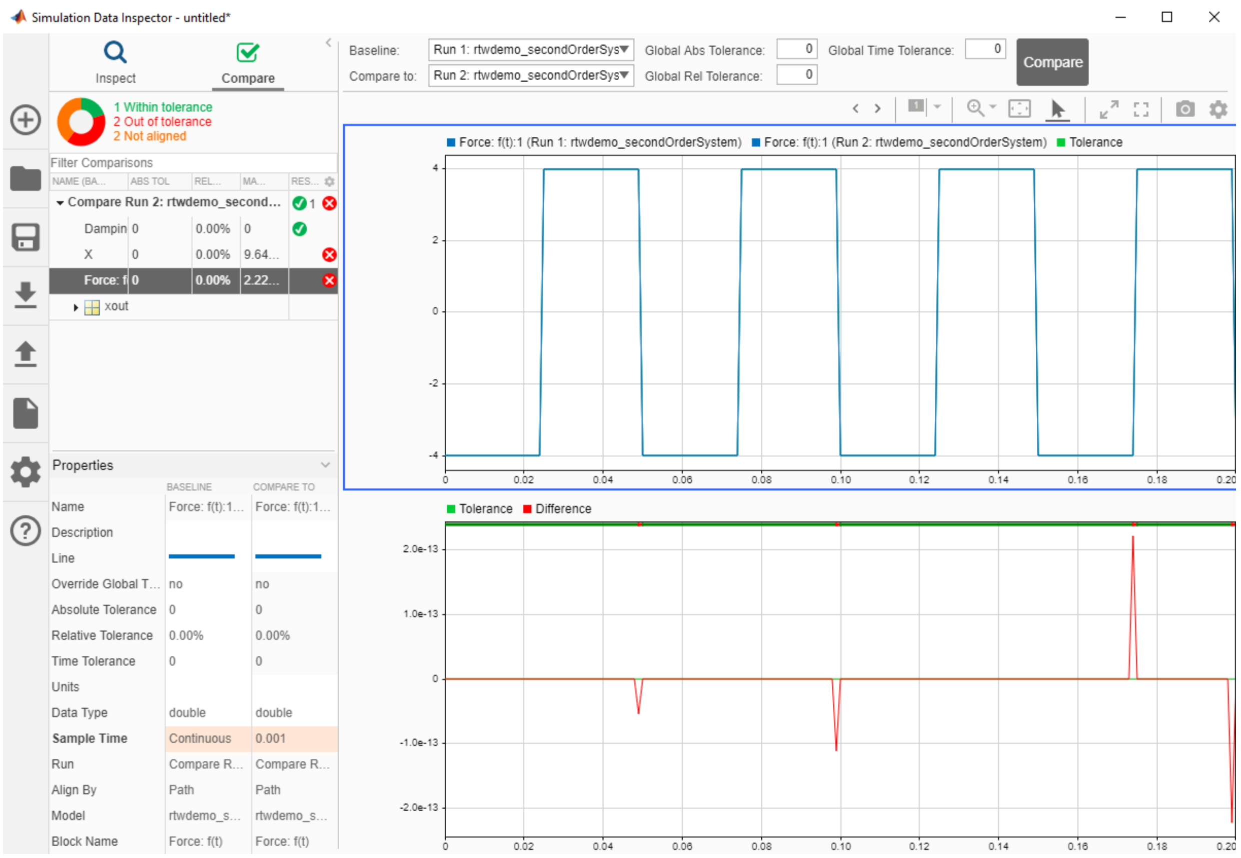
Filter Comparisons

NAME (BA...	ABS TOL	REL...	MA...	RES...
Compare Run 2: rtwdemo_second...				✓ 1 ✗
Dampin	0	0.00%	0	✓
X	0	0.00%	9.64...	✗
Force: f	0	0.00%	2.22...	✗
xout				

Properties

	BASELINE	COMPARE TO
Name	X (Run 1: rt...	X (Run 2: rt...
Description		
Line		
Override Global T...	no	no
Absolute Tolerance	0	0
Relative Tolerance	0.00%	0.00%
Time Tolerance	0	0
Units		
Data Type	double	double
Sample Time	Continuous	0.001
Run	Compare R...	Compare R...
Align By	Path	Path
Model	rtwdemo_s...	rtwdemo_s...
Block Name	Integrator2	Integrator2

6 Inspect the comparison plot for `Force: f(t):1`. Under **File Comparisons**, select the row for `Force: f(t):1`.


Simulation Data Inspector - untitled*

Inspect Compare

Baseline: Run 1: rtwdemo_secondOrderSys
Compare to: Run 2: rtwdemo_secondOrderSys
Global Abs Tolerance: 0
Global Rel Tolerance: 0
Global Time Tolerance: 0
Compare

1 Within tolerance
2 Out of tolerance
2 Not aligned

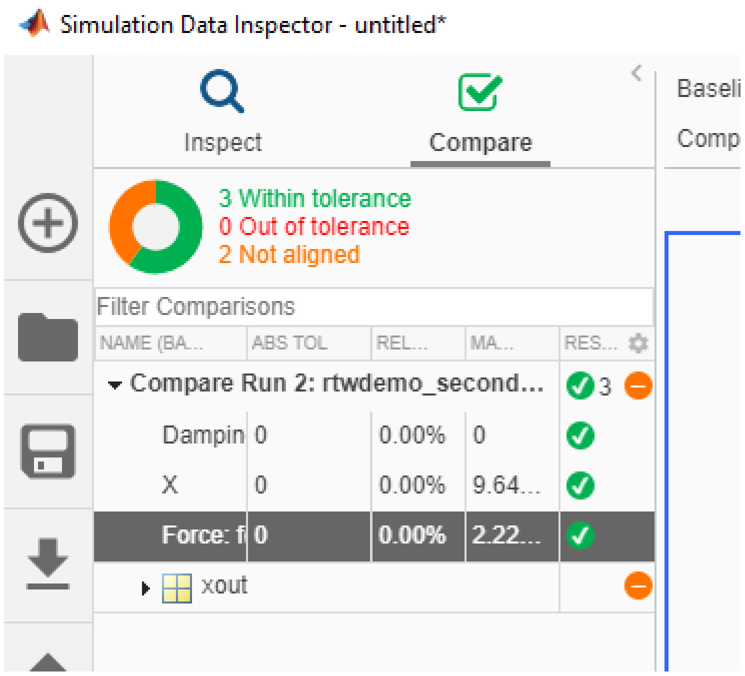
Filter Comparisons

NAME (BA...	ABS TOL	REL...	MA...	RES...
Compare Run 2: rtwdemo_second...				✓ 1 ✗
Dampin	0	0.00%	0	✓
X	0	0.00%	9.64...	✗
Force: f	0	0.00%	2.22...	✗
xout				

Properties

	BASELINE	COMPARE TO
Name	Force: f(t):1...	Force: f(t):1...
Description		
Line		
Override Global T...	no	no
Absolute Tolerance	0	0
Relative Tolerance	0.00%	0.00%
Time Tolerance	0	0
Units		
Data Type	double	double
Sample Time	Continuous	0.001
Run	Compare R...	Compare R...
Align By	Path	Path
Model	rtwdemo_s...	rtwdemo_s...
Block Name	Force: f(t)	Force: f(t)

7 Determine whether the numerical discrepancies are significant by specifying an absolute relative tolerance value. For this tutorial, set **Global Abs Tolerance** to `1e-12`. Then, click **Compare**. The comparisons for `X` and `Force: f(t):1` are within tolerance.


Simulation Data Inspector - untitled*

3 Within tolerance
0 Out of tolerance
2 Not aligned

NAME (BA...	ABS TOL	REL...	MA...	RES...
Compare Run 2: rtwdemo_second...				✓ 3 ⊖
Dampin	0	0.00%	0	✓
X	0	0.00%	9.64...	✓
Force: f	0	0.00%	2.22...	✓
xout				⊖

For more information,about numerical consistency verification and tolerances, see "Numerical Consistency of Model and Generated Code Simulation Results".

Next, tune a parameter during program execution.

Tune Parameter During Program Execution

In this section...

Interact with a generated executable program while the program runs in nonreal time on your development computer by tuning a parameter and observing the results.

Configure Data Accessibility and Communication Channel

This part of the tutorial assumes that you have configured your copy of example model `rtwdemo_secondOrderSystem` as described in "Configure Model for Verification" on page 2-9.

Build and Run Executable Program

1. To allow time for you to monitor changes that you make to the parameter, set the simulation stop time to `Inf`. In the Simulink Editor, click the **Hardware** tab. Click the down arrow to the right of **Monitor & Tune**. At the bottom of the list enter `Inf` for the simulation stop time.
2. Click **Monitor & Tune**. The software:
   a. Builds the executable program.
   b. Deploys the program as a separate process on your development computer.
   c. Connects the Simulink model to the program.
   d. Runs the model executable program code.

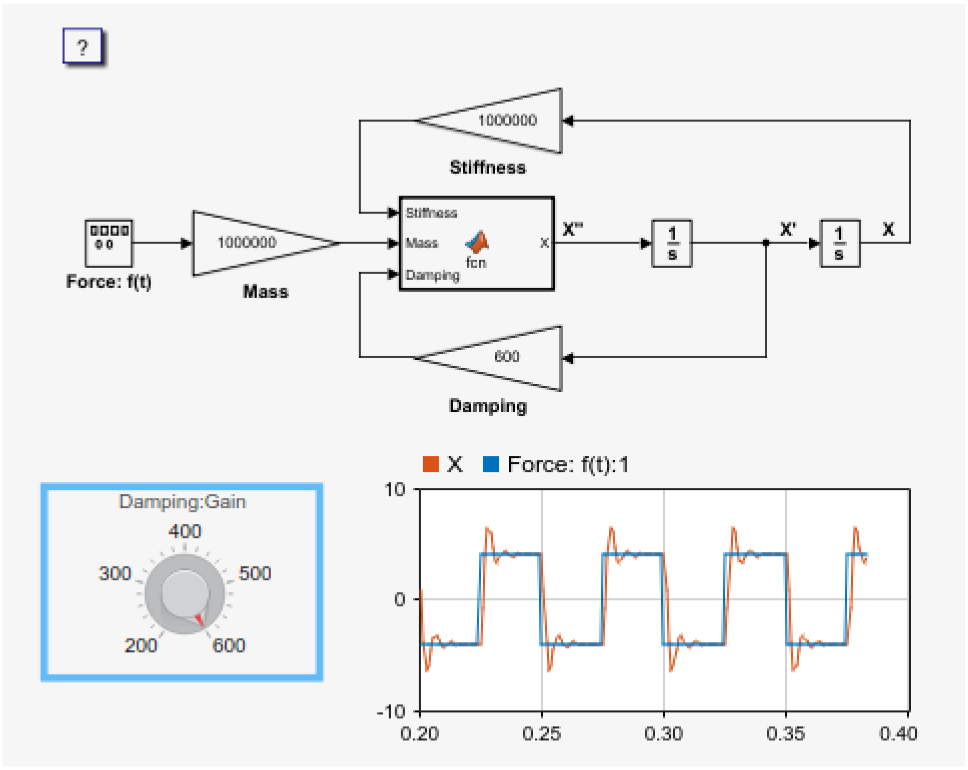
To stop the simulation, in the **Hardware** tab, click **Stop**.

Tune Parameter and Observe Results

Experiment with the value of a block parameter during execution. Observe the impact of the change.

While the executable program is simulating on your development computer, in the model canvas, use the Knob block to change the value of the damping gain. For example, change the value to 600. Observe:

- The changes in the plot are displayed in the Dashboard Scope block.
- In the Model Data Editor, the value for the **Gain** parameter of the `Damping` Gain block changes. In the bottom-left corner of the Simulink Editor, click the **Model Data Editor** tab.



Next, package the generated program code and artifacts for deployment.

Deploy Prototype Code and Artifacts

Package prototype code and artifacts in a Zip file so that you can share or relocate project results.

Package Generated Code and Artifacts in a Zip File

1. In the Simulink Editor, in the **C Code** tab, click **Share**.
2. Under **Package Code & Artifacts**, specify a file name for the zip file. By default, the code generator uses the model name and file extension `.zip`. For this example, use the default name.
3. Click **Generate Code and Package**. The code generator produces zip file `rtwdemo_secondOrderSystem.zip`.
4. Explore the contents of the generated zip file.

Explore Other Options

To explore more ways to customize, verify, and deploy generated rapid-prototyping code and artifacts, see the information that is listed in this table.

Goal	More Information
Configure data accessibility for rapid prototyping	"Access Signal, State, and Parameter Data During Execution"
Model multirate systems	"Scheduling"
Create multiple model configuration sets and share configuration parameter settings across models	"Configuration Reuse" (Simulink)
Control how signals are stored and represented in the generated code	"How Generated Code Stores Internal Signal, State, and Parameter Data"
Generate block parameter storage declarations and interface block parameters to your code	"Create Tunable Calibration Parameter in the Generated Code"
Interface with legacy code for simulation and code generation	"External Code Integration"
Generate code compatible with C++	"Programming Language"

Goal	More Information
Create a protected model that hides block and line information for sharing with a third party	“Model Protection”
Customize the build process	“Build Process Customization”