

Simulink® Compiler™

User's Guide



MATLAB® & SIMULINK®

R2020b



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® Compiler™ User's Guide

© COPYRIGHT 2020 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2020	Online only	New for Version 1.0 (Release 2020a)
September 2020	Online only	Revised for Version 1.1 (Release 2020b)

1

Simulink Compiler

Deploy an App Designer Simulation with Simulink Compiler	1-2
Deploying A Simulation App with Simulink Compiler	1-2
Running the Deployed Application	1-6
Deploying A Simulation App with Simulink Compiler	1-8
Deploy Simulations with Tunable Parameters	1-12
Prepare a Script to Deploy Simulations with Parameter Tuning	1-12
Comparing Simulink Coder and Simulink Compiler	1-15
Differences	1-15
Common Questions about Simulink Compiler and Simulink Coder	1-15
Rapid Accelerator Limitations	1-17
Rapid Accelerator Mode	1-17
Limitations	1-17
Debug an Application for Deployment	1-19
Debug Application in Simulink	1-19
Debug Application	1-19
Export Simulink Model to Standalone FMU	1-20
Generate, Modify and Deploy a MATLAB App for a Simulink Model	1-24
Generate and Deploy a MATLAB App for a Model	1-24
Generate and Deploy a MATLAB App for a Model	1-30
Simulation Callbacks for Deployable Applications	1-35
Deploy an App with Live Simulation Results of Lorenz System	1-35
Deploy an App with Live Simulation Results of Lorenz System	1-40

Simulink Compiler

- “Deploy an App Designer Simulation with Simulink Compiler” on page 1-2
- “Deploying A Simulation App with Simulink Compiler” on page 1-8
- “Deploy Simulations with Tunable Parameters” on page 1-12
- “Comparing Simulink Coder and Simulink Compiler” on page 1-15
- “Rapid Accelerator Limitations” on page 1-17
- “Debug an Application for Deployment” on page 1-19
- “Export Simulink Model to Standalone FMU” on page 1-20
- “Generate, Modify and Deploy a MATLAB App for a Simulink Model” on page 1-24
- “Generate and Deploy a MATLAB App for a Model” on page 1-30
- “Simulation Callbacks for Deployable Applications” on page 1-35
- “Deploy an App with Live Simulation Results of Lorenz System” on page 1-40

Deploy an App Designer Simulation with Simulink Compiler

In this section...

“Deploying A Simulation App with Simulink Compiler” on page 1-2

“Running the Deployed Application” on page 1-6
--

This example walks you through the workflow of creating a simulation app in App Designer and using Simulink Compiler to deploy it. The example explains the code that is used to build the app.

To open the example, type the following in the MATLAB® command window, or click the View MATLAB Code button.

```
openExample('simulinkcompiler/DeployingASimulationAppUsingSimulinkCompilerExample')
```

Deploying A Simulation App with Simulink Compiler

In this example, we use an app that is prepared in the App Designer and deploy it with Simulink Compiler.

Open and Explore Model

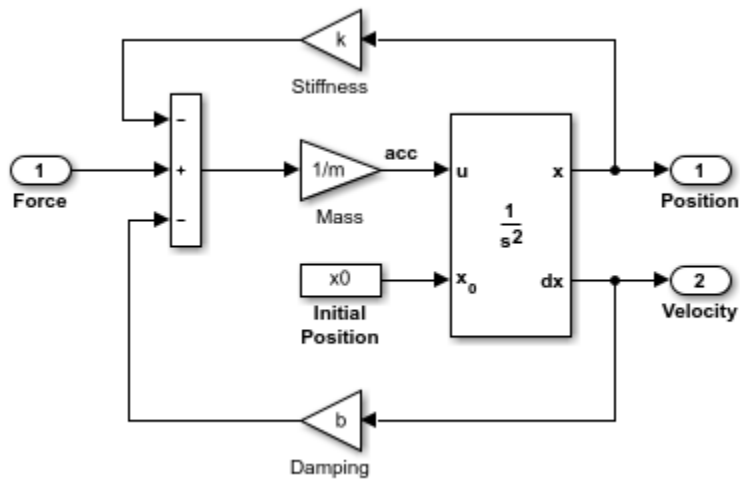
In this example, we use the model of a mass springer damper system. The mass-spring-damper model consists of discrete mass nodes distributed throughout an object and interconnected via a network of springs and dampers. This model is well-suited for modelling object with complex material properties such as non-linearity and elasticity. In this example we use the mass spring damper system. The system is parametrized by mass (m), spring stiffness (k), damping (b) and the initial position (x0). The input to the system is the applied force.

To explore this model with different values of the tunable parameters, create the following model workspace variables:

- Mass - m.
- Spring stiffness - k.
- Damping - b.
- Initial position - x0.

To create the model workspace variables, go to the **Modelling** tab and select **Model Workspace** in the **Data Repositories** in the **Design** section. Use the **Add MATLAB Variables** icon to add the above four variables. Add the appropriate initial values, for example, 3, 128, 2 and 0.5 respectively.

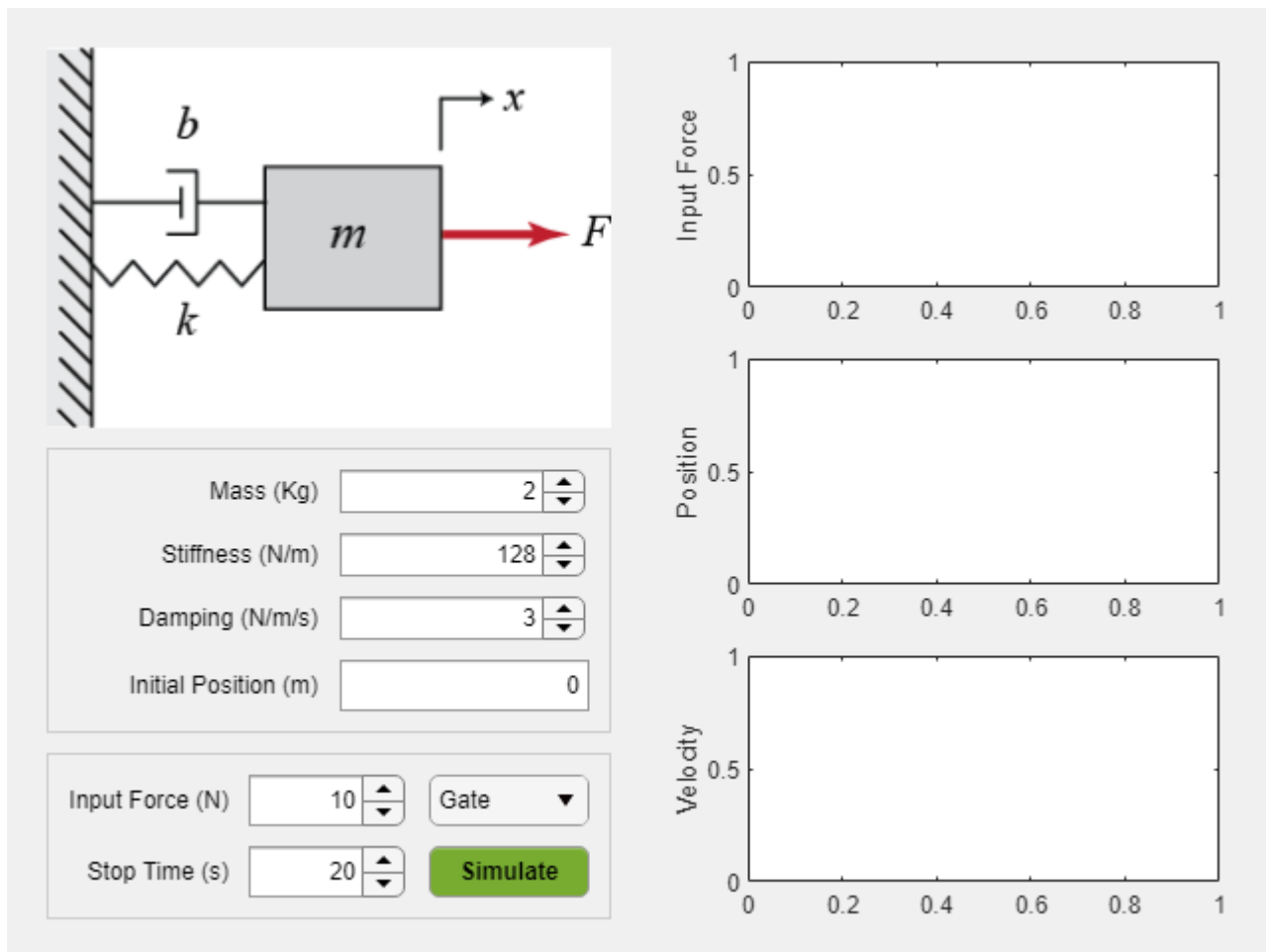
```
open_system('MassSpringDamperModel');
```



Create the App in App Designer

Use the MATLAB APP Designer to create an app to simulate the model with different parameter values and input signals. To learn more about how to create an app using the App Designer, see "Create and Run a Simple App Using App Designer" Use the `MassSpringDamperApp.mlapp` file to use the app.

MassSpringDamperApp



App Details

The main part for the app is the simulate button callback function. It has the following salient parts: setup the `SimulationInput` object, configure it for deployment, simulate, and plot the simulation results.

The functionality of the application to change and experiment with the tunable parameters is defined in the callback function `SimulateButtonPushed`. This callback function enables you to change, experiment and analyze different simulations by modifying the values in the app designer.

SimulateButtonPushed Callback Function Code

This section explains the code written to create the app, `MassSpringDamperApp`. The callback function `SimulateButtonPushed` is called in the app designed in the App Designer. This callback function defines how the model is simulated. We use the `Simulink.SimulationInput` object to set the variables to the model and use these variables to change the values and analyze the model.

Create the `Simulink.SimulationInput` Object in the `SimulateButtonPushed` Function

In the `SimulateButtonPushed` function, create a `SimulationInput` object, `SimInp` for the model `MassSpringDamperModel`. Use the `setModelParameters` method on the `SimulationInput` object. In this example, we set the `StopTime` model parameter for the simulation.


```
function SimulateButtonPushed(app, event)
    try
        app.toggleUIC('off', 'Simulating ...')

        simInp = Simulink.SimulationInput('MassSpringDamperModel');

        stopTimeStr = num2str(app.StopTimeSpinner.Value);
        simInp = simInp.setModelParameter('StopTime', stopTimeStr);
```

Set the Values of the Tunable Parameters and the Input Signal

To set the input signal to the model, use the `ExternalInput` property of the `Simulink.SimulationInput` object, `simInp`. Use the `setVariables` method to set the values of the four tunable parameters. Create the force input signal and set it as the `ExternalInput` in the simulation input object.

```
simInp.ExternalInput = app.externalInput();

simInp = simInp.setVariable('k', app.StiffnessSpinner.Value);
simInp = simInp.setVariable('m', app.MassSpinner.Value);
simInp = simInp.setVariable('b', app.DampingSpinner.Value);
simInp = simInp.setVariable('x0', app.InitialPositionEditField.Value);
```

Configure for Deployment

Now that we have assigned all the values to the variables and set the input signal, the `Simulink.SimulationInput` object is required to be configured for deployment. Use the `simulink.compiler.configureForDeployment` function of Simulink Compiler. This function handles all the settings required for the script to be compatible for deployment by setting the simulation mode to rapid accelerator, and by setting the parameter `RapidAcceleratorUpToDateCheck` to off.

```
simInp = simulink.compiler.configureForDeployment(simInp);
```

Simulate and Plot the Results

Use the configured `Simulink.SimulationInput` object to run the simulation with the `sim` command. Plot the results from the simulation using the `Simulink.SimulationOutput` object, `simOut`.

```
simOut = sim(simInp);

t = simOut.y.time;
yp = simOut.y.signals(1).values;
yv = simOut.y.signals(2).values;
plot(app.PositionUIAxes, t, yp);
plot(app.VelocityUIAxes, t, yv);

catch ME
    errordlg(ME.message);
end
app.toggleUIC('on', 'Simulate');
end
```

Test Out the Application in App Designer

Before deploying the application, ensure that the app runs in the App Designer. Click the **Simulate** button on the app to verify that the application works by simulating the model for different values.

Compile Script for Deployment

To compile the app, use the `mcc` command, followed by the script name.

```
mcc -m MassSpringDamperApp.mlapp
```

Running the Deployed Application

Install MATLAB Runtime and Package the Deployable

To run the deployed executable, you need an appropriate runtime environment. For more information, see “MATLAB Runtime”.

Ensure that the path environment variable is free of other instances of MATLAB Runtime from previous installs. If there are any, remove them.

To install MATLAB Runtime, follow the instructions on “Install and Configure the MATLAB Runtime”.

Compile the deployable for the first time as follows:

- 1 Enter `deploytool` command in the MATLAB Command Window and select **Application Compiler**.
- 2 In the **Main File** section, add the file to be deployed, `MassSpringDamperApp.mlapp`
- 3 In the **Packaging Options** section on the toolstrip, select **Runtime included in package** and enter the `deployed_installer` in the text box.
- 4 Click **Package** in the **Package** section of the toolstrip.
- 5 Once the package is ready, use the `deployed_installer` in the `for_redistribution` folder to install the proper runtime environment for running the deployed application.

Run the Deployed Application

You can run the deployed script only on the platform that the deployed script was developed on.

It is recommended to run the deployed application from the Windows Command Prompt. Running the deployed application from the command prompt also enables the script to print errors when something is wrong in the deployed application. These errors can help troubleshoot the problem.

Note The `MassSpringDamperApp.mlapp` contains `errordlg`, and `errordlg` is not supported on Web Apps.

See Also

`Simulink.SimulationInput | deploytool | mcc | simulink.compiler.configureForDeployment`

More About

- “Create and Deploy a Script with Simulink Compiler”
- “Deploy Simulations with Tunable Parameters” on page 1-12
- “Simulation Callbacks for Deployable Applications” on page 1-35

Deploying A Simulation App with Simulink Compiler

In this example, we use an app that is prepared in the App Designer and deploy it with Simulink Compiler.

Open and Explore Model

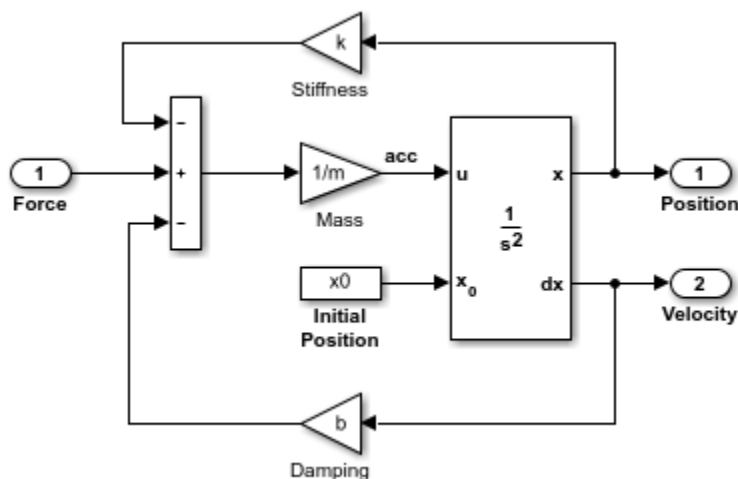
In this example, we use the model of a mass spring damper system. The mass-spring-damper model consists of discrete mass nodes distributed throughout an object and interconnected via a network of springs and dampers. This model is well-suited for modelling object with complex material properties such as non-linearity and elasticity. In this example we use the mass spring damper system. The system is parametrized by mass (m), spring stiffness (k), damping (b) and the initial position (x_0). The input to the system is the applied force.

To explore this model with different values of the tunable parameters, create the following model workspace variables:

- Mass - m .
- Spring stiffness - k .
- Damping - b .
- Initial position - x_0 .

To create the model workspace variables, go to the **Modelling** tab and select **Model Workspace** in the **Data Repositories** in the **Design** section. Use the **Add MATLAB Variables** icon to add the above four variables. Add the appropriate initial values, for example, 3, 128, 2 and 0.5 respectively.

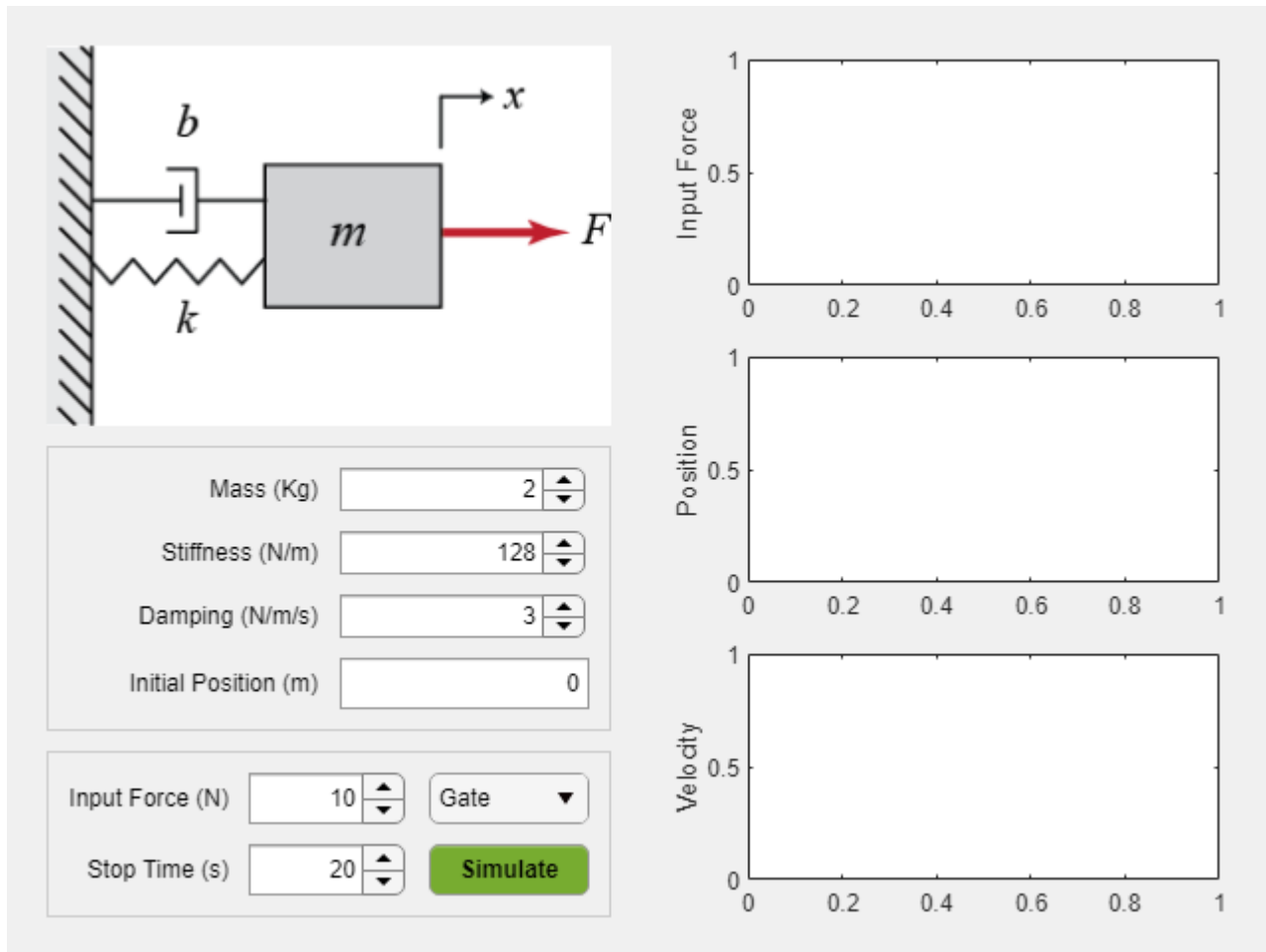
```
open_system('MassSpringDamperModel');
```



Create the App in App Designer

Use the MATLAB APP Designer to create an app to simulate the model with different parameter values and input signals. To learn more about how to create an app using the App Designer, see “Create and Run a Simple App Using App Designer” Use the `MassSpringDamperApp.mlapp` file to use the app.

MassSpringDamperApp

**App Details**

The main part for the app is the simulate button callback function. It has the following salient parts: setup the `SimulationInput` object, configure it for deployment, simulate, and plot the simulation results.

The functionality of the application to change and experiment with the tunable parameters is defined in the callback function `SimulateButtonPushed`. This callback function enables you to change, experiment and analyze different simulations by modifying the values in the app designer.

SimulateButtonPushed Callback Function Code

This section explains the code written to create the app, `MassSpringDamperApp`. The callback function `SimulateButtonPushed` is called in the app designed in the App Designer. This callback function defines how the model is simulated. We use the `Simulink.SimulationInput` object to set the variables to the model and use these variables to change the values and analyze the model.

Create the `Simulink.SimulationInput` Object in the `SimulateButtonPushed` Function

In the `SimulateButtonPushed` function, create a `SimulationInput` object, `simInp` for the model `MassSpringDamperModel`. Use the `setModelParameters` method on the `SimulationInput` object. In this example, we set the `StopTime` model parameter for the simulation.

```
function SimulateButtonPushed(app, event)
    try
        app.toggleUIC('off', 'Simulating ...')

        simInp = Simulink.SimulationInput('MassSpringDamperModel');

        stopTimeStr = num2str(app.StopTimeSpinner.Value);
        simInp = simInp.setModelParameter('StopTime', stopTimeStr);
```

Set the Values of the Tunable Parameters and the Input Signal

To set the input signal to the model, use the `ExternalInput` property of the `Simulink.SimulationInput` object, `simInp`. Use the `setVariables` method to set the values of the four tunable parameters. Create the force input signal and set it as the `ExternalInput` in the simulation input object.

```
simInp.ExternalInput = app.externalInput();

simInp = simInp.setVariable('k', app.StiffnessSpinner.Value);
simInp = simInp.setVariable('m', app.MassSpinner.Value);
simInp = simInp.setVariable('b', app.DampingSpinner.Value);
simInp = simInp.setVariable('x0', app.InitialPositionEditField.Value);
```

Configure for Deployment

Now that we have assigned all the values to the variables and set the input signal, the `Simulink.SimulationInput` object is required to be configured for deployment. Use the `simulink.compiler.configureForDeployment` function of Simulink Compiler. This function handles all the settings required for the script to be compatible for deployment by setting the simulation mode to rapid accelerator, and by setting the parameter `RapidAcceleratorUpToDateCheck` to off.

```
simInp = simulink.compiler.configureForDeployment(simInp);
```

Simulate and Plot the Results

Use the configured `Simulink.SimulationInput` object to run the simulation with the `sim` command. Plot the results from the simulation using the `Simulink.SimulationOutput` object, `simOut`.

```
simOut = sim(simInp);

t = simOut.y.time;
    yp = simOut.y.signals(1).values;
    yv = simOut.y.signals(2).values;
    plot(app.PositionUIAxes, t, yp);
    plot(app.VelocityUIAxes, t, yv);

    catch ME
        errordlg(ME.message);
    end
    app.toggleUIC('on', 'Simulate');
end
```

Test Out the Application in App Designer

Before deploying the application, ensure that the app runs in the App Designer. Click the **Simulate** button on the app to verify that the application works by simulating the model for different values.

Compile Script for Deployment

To compile the app, use the `mcc` command, followed by the script name.

```
mcc -m MassSpringDamperApp.mlapp
```

Deploy Simulations with Tunable Parameters

With Simulink Compiler, you can deploy simulations that use tunable parameters.

As you construct a model, you can experiment with block parameters, such as the coefficients of a Transfer Fcn block, to help you decide which blocks to use. You can simulate the model with different parameter values, and capture and observe the simulation output.

You can change the values of most numeric block parameters during a simulation. This technique allows you to quickly test parameter values while you develop an algorithm. You can:

- Tune and optimize control parameters.
- Calibrate model parameters.
- Test control robustness under different conditions.

The following example shows how to set a tunable parameter in a model, write a standalone application that can be used to tune the parameters, and analyze the simulations. For more information on tunable parameters, see “Tune and Experiment with Block Parameter Values”.

Prepare a Script to Deploy Simulations with Parameter Tuning

In this example, create a MATLAB function to simulate the model `sldemo_suspn_3dof` with the values of `Simulink.SimulationInput`. Save the script as `deployParameterTuning.m` on the MATLAB path.

Prepare a Function to Deploy

Create a function called `deployParameterTuning` containing the code shown below. This code creates a `Simulink.SimulationInput` object for the model `sldemo_suspn_3dof`. `mb` is the value that we pass through the `setVariable` method for the tunable parameter, `Mb`. To configure this script to be deployed, use the function `simulink.compiler.configureForDeployment`. `simulink.compiler.configureForDeployment` configures the `Simulink.SimulationInput` object for by deployment by setting its simulation mode to Rapid Accelerator and by restricting inputs that require rebuilding the deployed app.

```
function deployParameterTuning(oName, mb)

    if ischar(mb) || isstring(mb)
        mb = str2double(mb);
    end

    if isnan(mb) || ~isa(mb, 'double') || ~isscalar(mb)
        disp('The value of mb given to deployParameterTuning must be a double scalar or a string')
    end

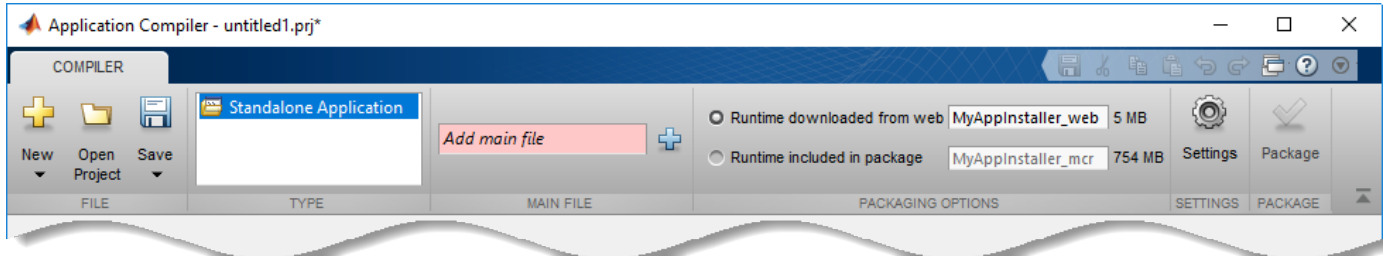
    in = Simulink.SimulationInput('sldemo_suspn_3dof');
    in = in.setVariable('Mb', mb);
    in = simulink.compiler.configureForDeployment(in);
    out = sim(in);

    save(oName, 'out');


end
```


Deploy the Prepared Function

- 1 On the **Apps** tab, in the **Apps** section, click the arrow. In **Application Deployment**, click **Application Compiler**.



Alternately, you can open the **Application Compiler** app by entering `applicationCompiler` at the MATLAB prompt.

- 2 In the **Compiler** project window, specify the main file of the MATLAB application that you want to deploy.
 - a In the **Main File** section, click .
 - b In the **Add Files** window, browse to path where you have saved the prepared function, and select `deployParameterTuning.m`. Click **Open**.

The function `deployParameterTuning.m` is added to the list of main files.

- 3 Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:
 - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.
 - **Runtime included in package** — Generates an installer that includes the MATLAB Runtime installer.
- 4 Customize the packaged application and its appearance:
 - **Application information** — This section lists editable information about the deployed application. You can also customize the standalone applications appearance by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer”.
 - **Command line input type options** — This section lists selection of input data types for the standalone application. For more information, see “Determine Data Type of Command-Line Input (For Packaging Standalone Applications Only)”.
 - **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See “Change the Installation Path” .
 - **Files required for your application to run** —Files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project”.
 - **Files installed for your end user** — This section lists the files that are installed with your application. These files include:
 - A generated `readme.txt` file
 - The generated executable for the target platform

See “Specify Files to Install with Application”.

- **Additional runtime settings** —This section lists platform-specific options for controlling the generated executable. See “Additional Runtime Settings”.
- 5 To generate the packaged application, click **Package**. In the Save Project dialog box, specify the location to save the project.
 - 6 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the packaging process is complete, examine the generated output.

- `PackagingLog.txt` — Log file generated by MATLAB Compiler.
- Three folders are generated in the target folder location: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`. For more information about the files generated in these folders, see .

See Also

`Simulink.SimulationInput` | `configureForDeployment` | `deploytool` | `mcc` | `sim` | `simulink.compiler.genapp`

More About

- “Simulink Compiler Workflow Overview”
- “Tune and Experiment with Block Parameter Values”
- “Code Regeneration in Accelerated Models”

Comparing Simulink Coder and Simulink Compiler

Simulink Compiler enables you to share Simulink simulations as standalone executables. You can build the executables by packaging the compiled Simulink model and the MATLAB code to set up, run, and analyze a simulation. Standalone executables can be complete simulation apps that use MATLAB graphics and UIs designed with MATLAB App Designer. To cosimulate with an external simulation environment, you can generate standalone Functional Mockup Unit (FMU) binaries that adhere to the Functional Mockup Interface (FMI) standard.

Simulink Coder 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, 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.

Differences

The following table states the major comparisons between Simulink Compiler and Simulink Coder. Use this table to understand the differences between the applications and usage of the two products.

Outputs and Support	Simulink Compiler	Simulink Coder
Main Use Case	Deploy simulations as standalone executables on desktop or production servers	Generate portable C/C++ code for Simulink model that can be deployed on embedded platforms or desktop
Output	Executable or software component or shared library	Portable and readable C/C++ source code
Simulink Block Support	All the blocks supported in Rapid Accelerator mode in Simulink	A subset of Simulink blocks
Supported Blocksets	All the blocksets supported by Rapid Accelerator mode in Simulink	A subset of Simulink blocks
Production	MATLAB Production Server	Embedded Coder
Simulink Graphics Support	Supports MATLAB Graphics.	None
Library Dependencies	MATLAB Runtime	None

Common Questions about Simulink Compiler and Simulink Coder

The following table answers some of the common questions about using Simulink Compiler and Simulink Coder, such as the memory required, performance, and other questions about support.

Common Questions	Simulink Compiler	Simulink Coder
What files are produced?	Shared executables or libraries, along with the required MATLAB Runtime components.	Source code (*.c & *.h) that can be compiled into shared libraries and executables

Common Questions	Simulink Compiler	Simulink Coder
Which platforms can these files be deployed to?	All platforms supported by MATLAB (Windows, Mac, and Linux)	Any platform that supports ANSI/ISO C/C++ code
Does it generate readable code?	No, only non-readable shared libraries	Yes, readable source code
Is it faster than Simulink?	Runs at the same speed as Rapid Accelerator mode in Simulink.	Has the potential to be faster, depending on the algorithm. The generated code is not faster for optimized MATLAB functions (such as FFT, or Image Processing, and Computer Vision functions) Use GPU Coder GPU Coder™ to generate CUDA source code that potentially runs faster on NVIDIA GPUs.
Does it take advantage of hardware accelerators?	Supports the same hardware as MATLAB, including GPUs and AVX. Multicore and clusters are supported via Parallel Computing Toolbox	C code running on local multicore machines can be supported using the OpenMP API. Use GPU Coder to generate CUDA source code that runs on NVIDIA GPUs. Use HDL Coder™ to generate Verilog or VHDL that runs on FPGAs.
What is the difference in memory use on a desktop?	Highly dependent on the executables. MATLAB Runtime itself uses more memory than the Simulink Coder.	Highly dependent on the MATLAB code.
What file I/O formats does each software support?	Supports all formats supported in MATLAB	Limited file support: text files, audio, and video formats. Does not support image formats.

See Also

More About

- “Simulink Compiler Workflow Overview”

Rapid Accelerator Limitations

Rapid Accelerator Mode

The rapid accelerator mode creates a *Rapid Accelerator standalone executable* from your model. This executable includes the solver and model methods, but it resides outside of MATLAB and Simulink. It uses external mode to communicate with Simulink.

MATLAB and Simulink run in one process, and if a second processing core is available, the standalone executable runs there.

Limitations

- Rapid Accelerator mode does not support:
 - Algebraic loops.
 - Targets written in C++.
 - Interpreted MATLAB Function blocks.
 - Noninlined MATLAB language or Fortran S-functions. You must write S-functions in C or inline them using the Target Language Compiler (TLC) or you can also use the MEX file.
 - Debugger or Profiler.
 - Run time objects for Simulink.RunTimeBlock and Simulink.BlockCompOutputPortData blocks.
- Model parameters must be one of these data types:
 - `boolean`
 - `uint8` or `int8`
 - `uint16` or `int16`
 - `uint32` or `int32`
 - `single` or `double`
 - Fixed-point
 - Enumerated
- You cannot pause a simulation in Rapid Accelerator mode.
- If a Rapid Accelerator build includes referenced models (by using Model blocks), set up these models to use fixed-step solvers to generate code for them. The top model, however, can use a variable-step solver as long as the blocks in the referenced models are discrete.
- In certain cases, changing block parameters can result in structural changes to your model that change the model checksum. An example of such a change is changing the number of delays in a DSP simulation. In these cases, you must regenerate the code for the model. See “Code Regeneration in Accelerated Models” for more information.
- For root inports, Rapid Accelerator mode supports only base as the `Srcworkspace`.
- For root inports, when you specify the minimum and maximum values that the block should output, Rapid Accelerator mode does not recognize these limits during simulation.
- In Rapid Accelerator mode, To File or To Workspace blocks inside function-call subsystems do not generate any logging files if the function-call port is connected to Ground or unconnected.
- Simulink Compiler does not support the use of Scope block and non-virtual bus.

- Simulink Compiler does not support lcc-win64.
- Simulink Compiler does not support initialize, terminate, and reset blocks on referenced models.

See Also

More About

- “Simulink Compiler Workflow Overview”
- “Rapid Accelerator Mode”
- “Select Blocks for Rapid Accelerator Mode”
- “Parameter Tuning in Rapid Accelerator Mode”

Debug an Application for Deployment

This topic provides high-level tips on debugging the standalone applications. The following lists highlight the solutions and tips for most frequently encountered errors.

Debug Application in Simulink

Use the following tips while preparing the standalone application to be deployed:

- To ensure that the model runs successfully in rapid accelerator mode correctly, run the rapid accelerator target in a writable directory.
- While writing the script, ensure that the `sim` command uses the `Simulink.SimulationInput` object as the input.
- If you see the error "Unable to resolve the name `Simulink.SimulationInput`", check that the model is on the path.
- If the dependent files are located in another directory, attach them by using the flag `-a`. For example, `mcc -m scriptName.m -a myDataFile.dat`.
- If you are using a function as string, either:

- Add a function pragma `##function`.

```
set(gca, 'ButtonDownFcn', 'foo'); % function foo is a string here.
```

```
##function foo
```

```
set(gca, 'ButtonDownFcn', 'foo'); % function foo is a string here.
```

- Write it as an anonymous function

```
set(gca, 'ButtonDownFcn', @foo);
```

Debug Application

- Callback functions in the model might include functions that are not deployable. Ensure that the functions in the callbacks of the model are deployable.
- Callback functions are not invoked at runtime. Ensure that the deployed simulation application does not use callback functions that are required to be invoked at runtime.

See Also

More About

- "Simulink Compiler Workflow Overview"

Export Simulink Model to Standalone FMU

This example shows how to export Simulink component to standalone Co-Simulation FMU 2.0 with Simulink Compiler®. For a detailed explanation of the model, see:

- “Modeling a Fault-Tolerant Fuel Control System”

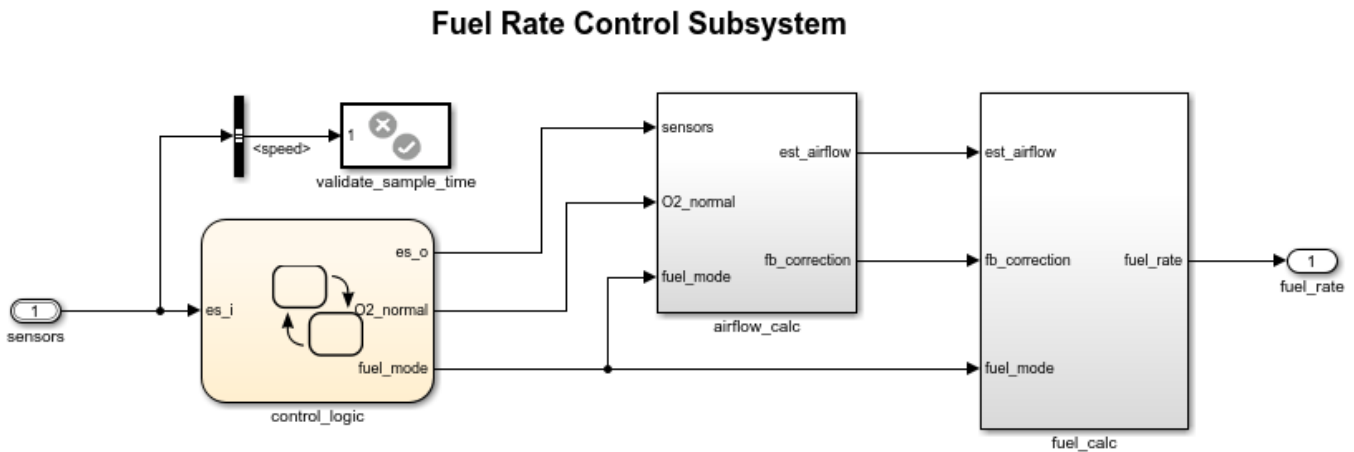
In this example, the air-fuel ratio control system is composed of three Simulink models:

- Fuel Rate Control Component: `fmudemo_export_fuelsys_controller`,
- Engine Gas Dynamics Component: `fmudemo_export_fuelsys_plant`, and
- top-level model `fmudemo_export_fuelsys_top`.

Once the controller and plant components are export to FMU format, they can be integrated using the top-level model. The generated FMUs can also be imported into other simulation tools that support FMI. For a list of Tools that support FMI, see: <https://fmi-standard.org/tools/>.

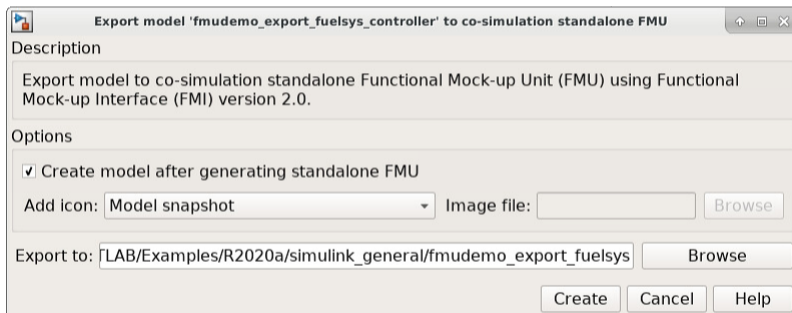
Export Fuel Rate Control Component to FMU

Open the `fmudemo_export_fuelsys_controller` example model.



Copyright 1990-2020 The MathWorks, Inc.

From **Simulation** tab, click drop-down button for **Save**. In **Export Model To** section, click **Standalone FMU...** In FMU Export dialog, configure wrapper model and icon settings, and specify save location for generated FMU.

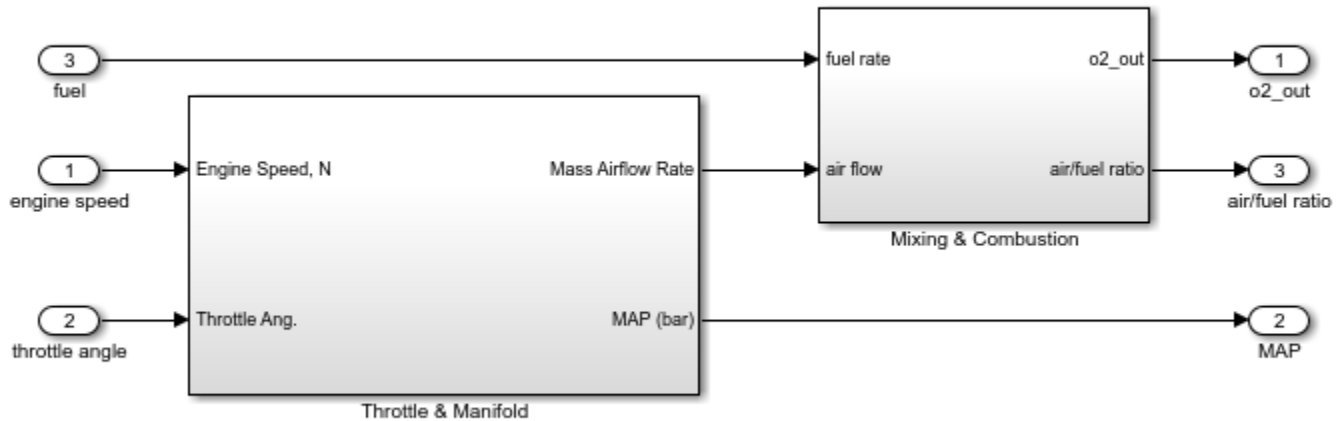


Click **Create** to export to FMU. The `fmudemo_export_fuelsys_controller.fmu` file can be found at specified save location.

Export Engine Gas Dynamics Component to FMU

Open the `fmudemo_export_fuelsys_plant` example model.

Engine Gas Dynamics



Copyright 1990-2020 The MathWorks, Inc.

FMU can also be exported using command-line. In MATLAB command line window, use `exportToFMU2CS` command:

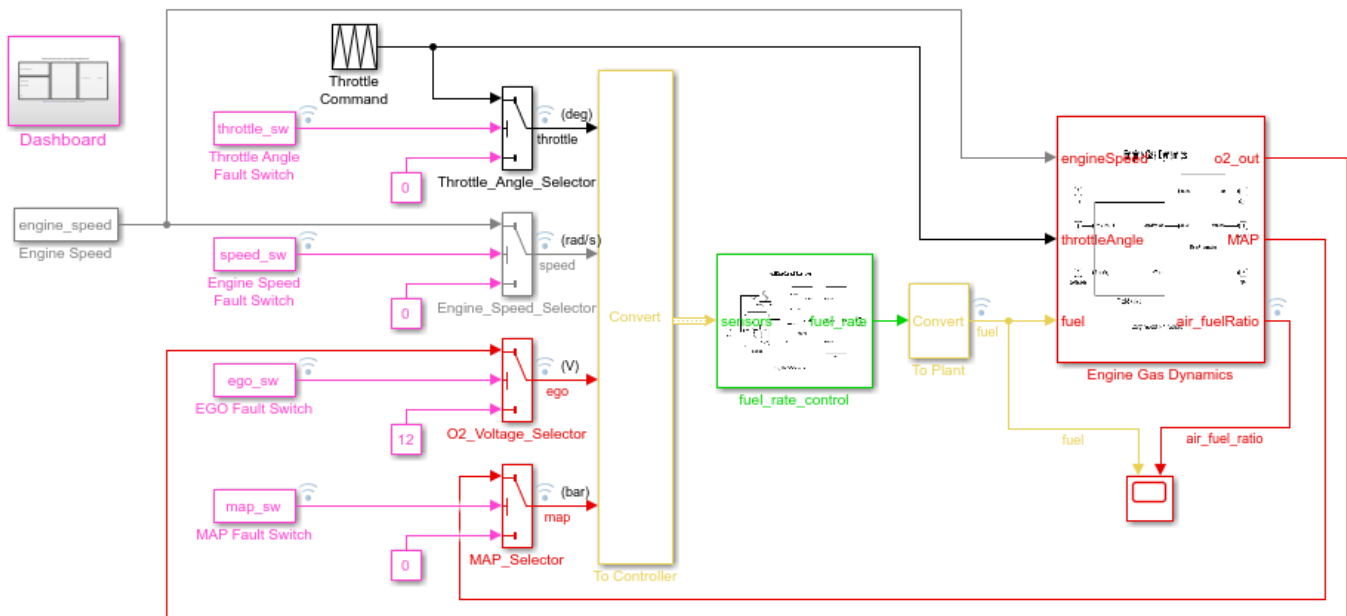
```
% Export model to Standalone Co-Simulation FMU 2.0
exportToFMU2CS('fmudemo_export_fuelsys_plant', 'CreateModelAfterGeneratingFMU', 'off', 'AddIcon')
```

You can use optional arguments **CreateModelAfterGeneratingFMU**, **AddIcon**, **SaveDirectory** to configure FMU export settings. For more information, call `help ExportToFMU2CS`.

Integrate FMU components in Simulink

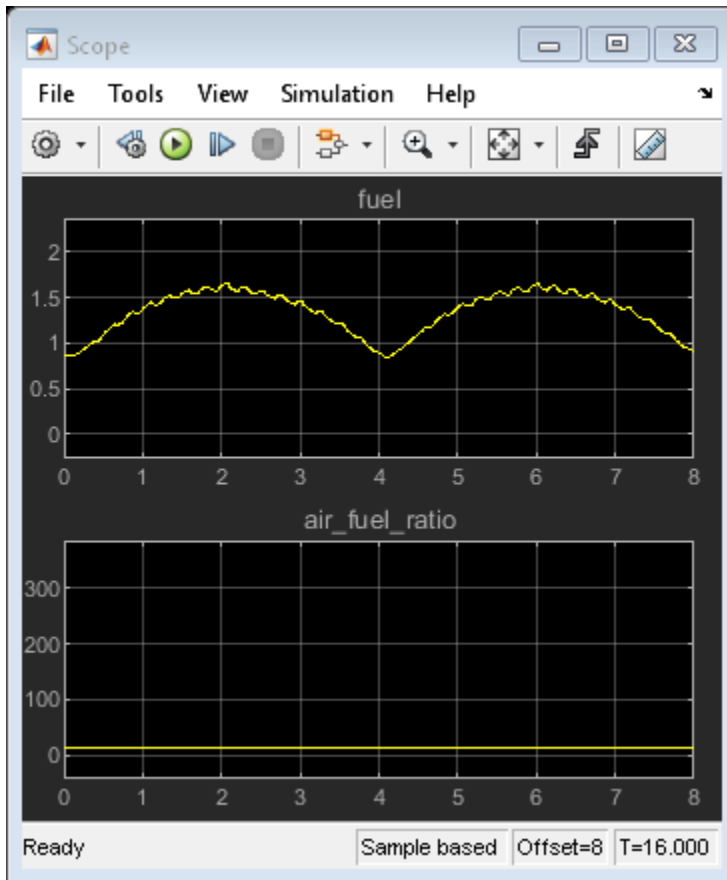
Once both FMUs are successfully exported, you may use the top model `fmudemo_export_fuelsys_top` to fully integrate the system for testing.

Fault-Tolerant Fuel Control System



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2020 The MathWorks, Inc.



Generate, Modify and Deploy a MATLAB App for a Simulink Model

The `simulink.compiler.genapp` enables you to automatically generate a MATLAB app for a Simulink model. You can compile and deploy the automatically generated app using the `mcc` command. The following example generates an app for a model, compiles and deploys it, and explores how you can customize the app using the MATLAB App Designer.

Generate and Deploy a MATLAB App for a Model

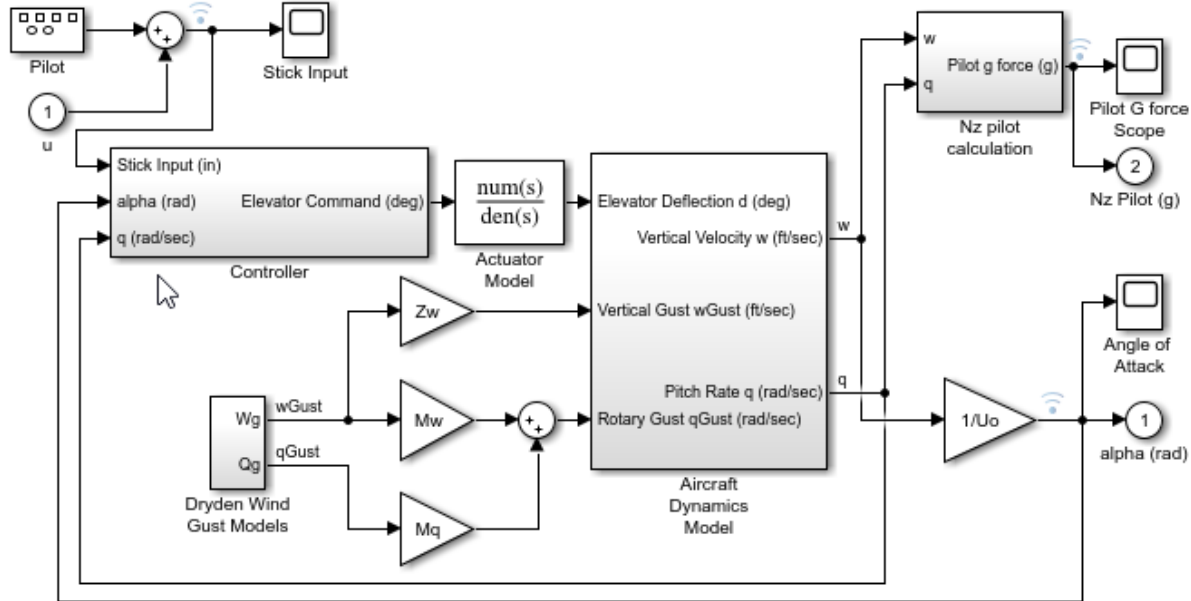
This example shows you how to use the `simulink.compiler.genapp` function to generate a MATLAB app for a model, that is deployable. Typically when a Simulink model is functionally complete, it is often used to run multiple simulations different input and parameter values. To try simulations for your model with different input and parameter values, you can generate a MATLAB App. You can also deploy this generated app for use outside of MATLAB.

This example illustrates the use of `simulink.compiler.genapp` function to generate a starter app for the model `f14`, using the generated app to tune the parameters of the model and simulate it, and customizing the app in the MATLAB App Designer.

Open the Model

The `simulink.compiler.example.AppGeneration` command loads the example project on your path. This project contains all the files required for this example including the model. Open the model `f14`.

```
simulink.compiler.example.AppGeneration;  
open_system(f14)
```



F-14 Flight Control

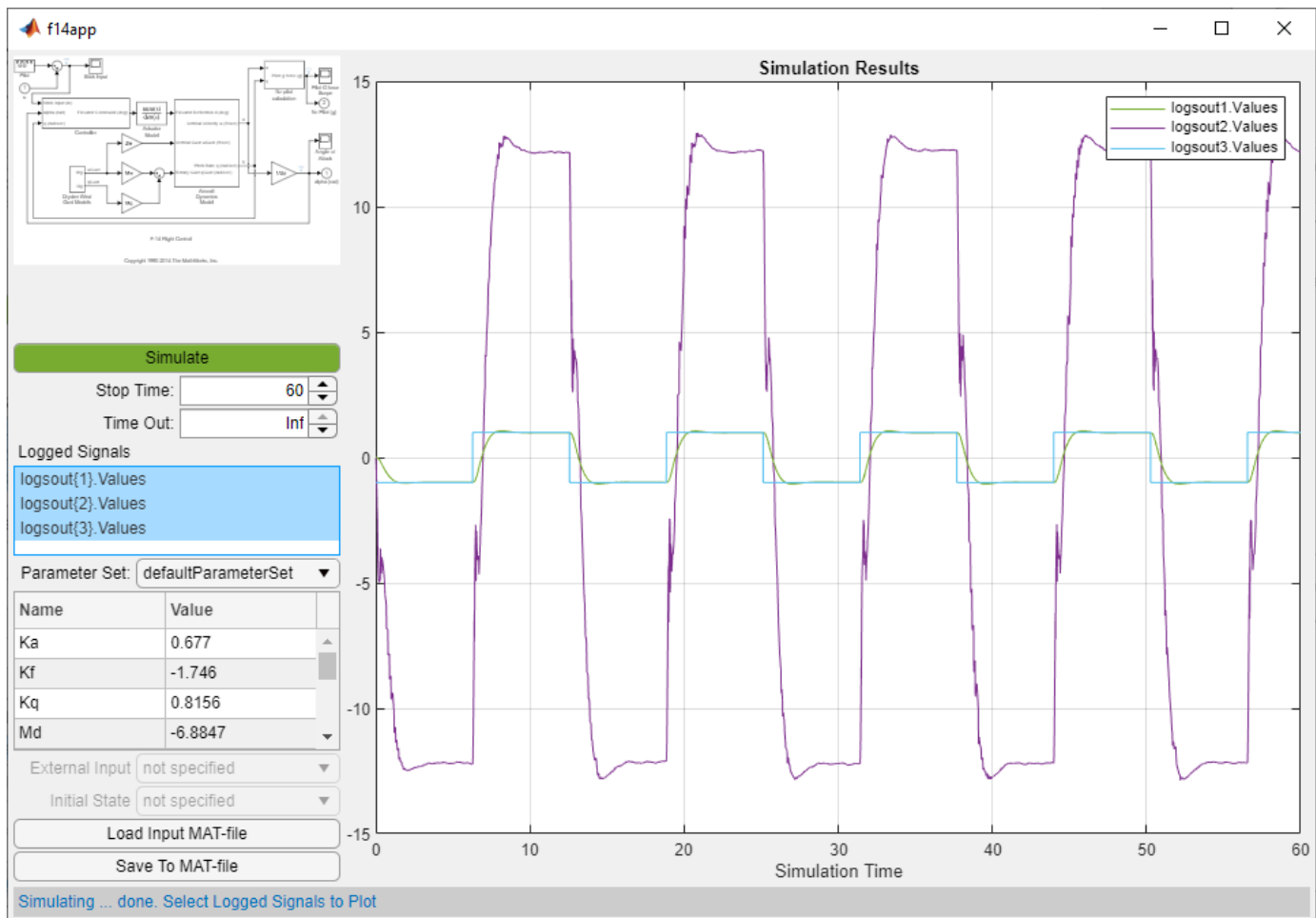
Copyright 1990-2014 The MathWorks, Inc.

Generate a MATLAB App for the Model

Use the `simulink.compiler.genapp` function to generate an app for the `f14` model. Running the `simulink.compiler.genapp` function with the model name as an argument generates an App named `f14app`. Simulink Compiler uses a default template to generate the app. The generated app provides an ability to tune the parameters and simulate the model for which the app is generated. The generated app also provides the plot of the simulation results. All the files are generated into the `f14app` directory.

```
simulink.compiler.genapp('f14', 'AppName', 'f14app');
```

After generating the app, Click **Simulate** to simulate the app.



Along with the app, the following artifacts are generated :

- `f14app.mlapp` file -- This file contains the code for the generated app. Open this file in App Designer for editing.
- Files starting with the `default` prefix -- Functions returning default values used by the app such as, model name, model image aspect ratio, model image file, and input MAT-file name.
- Model Image, (`f14app_image.svg`) -- Image of the Simulink Model.
- Inputs used in the simulation (`f14app_inputs.mat`) -- MAT file containing all the inputs that are used in the simulation of the model.
- App labels file `setLabels.m` -- File specifying label contents.
- Default Simulink logo (`SimulinkLogo.png`) -- File used as a placeholder for the model image.
- `pragma.m` directives file -- File used by Simulink Compiler to generate the deployable app.
- Set of MATLAB functions as M-files -- Files that the app uses to control user interface of the app.

Compile and Deploy the Generated App

You can use the MATLAB App Designer to compile and deploy the app. You can also use `deploytool`. For more information on compiling and deploying with App Designer, see *Develop Apps Using App Designer, Web Apps and Application Compiler*.

In this example, we compile the app, with the `mcc` command followed by the app name.

```
mcc -m f14App
```

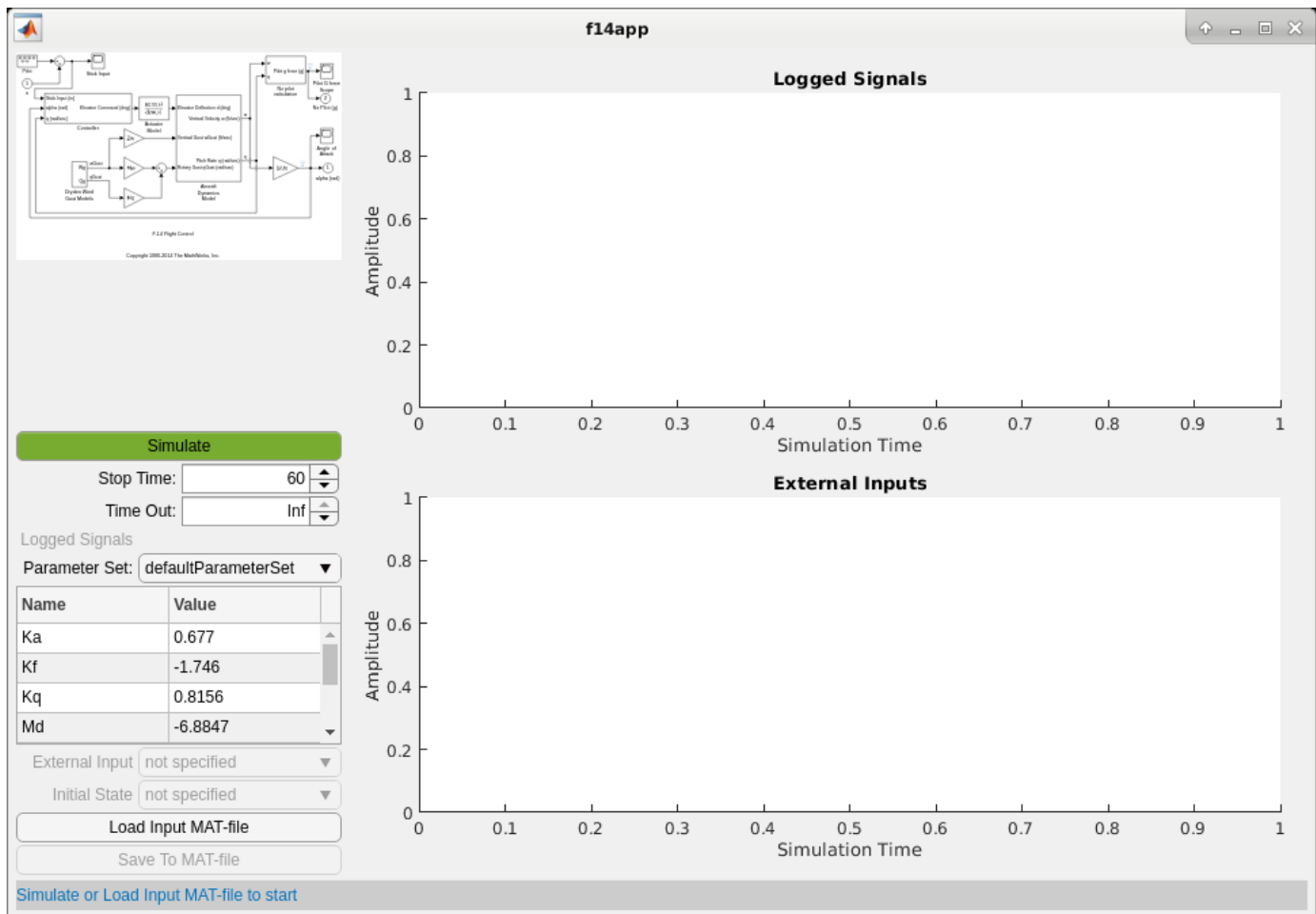
Customize the Generated App

You can also customize the generated app. To customize the app, use the App designer. The generated app `f14` is generic, but it allows you to easily customize it in the App Designer. In this section, we are going to replace the one axis in the generated app with two axes. Open the generated app in the app designer.

```
appdesigner('f14app');
```

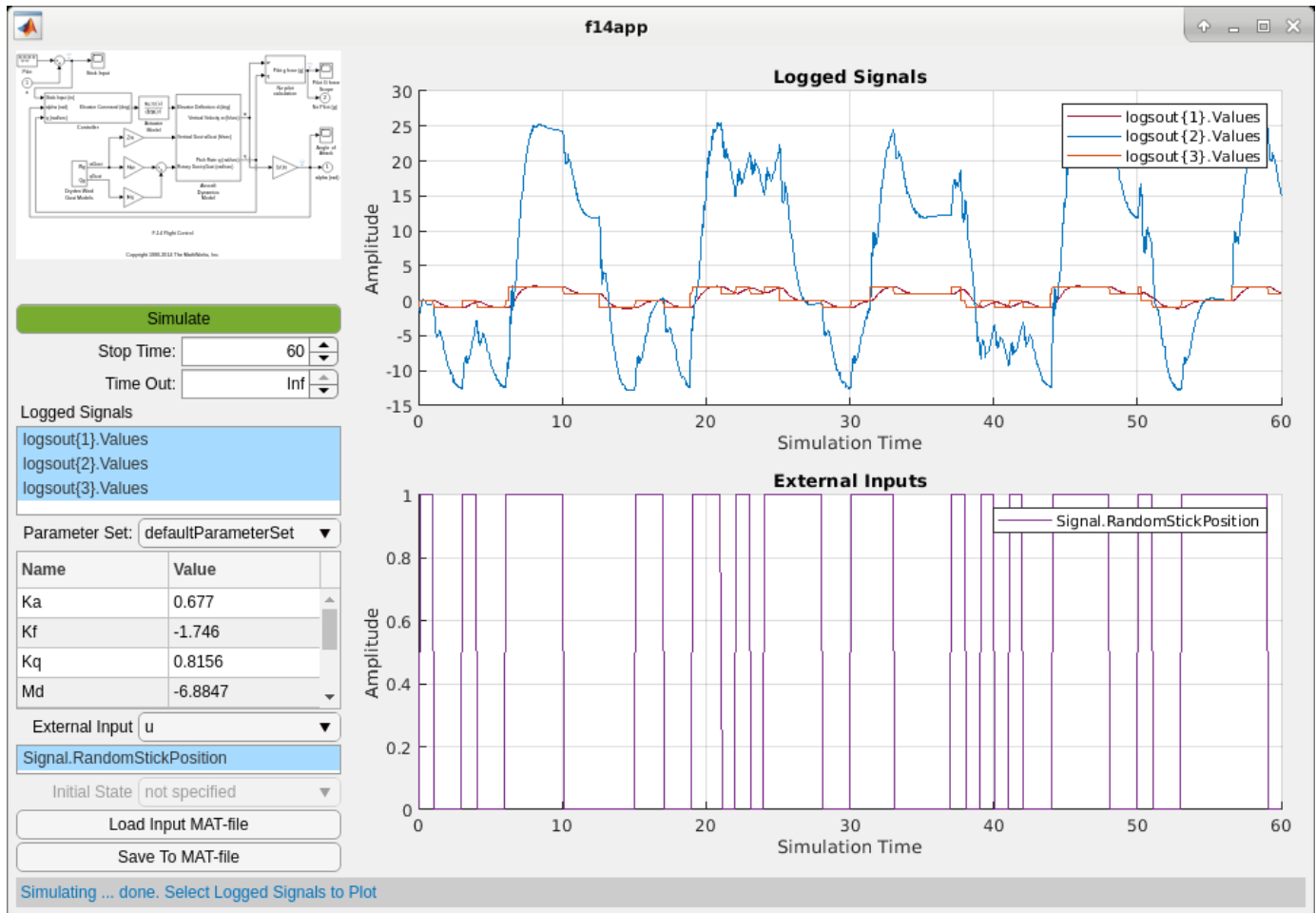
The idea of customization is to replace the Axes component with two Axes components. To get two Axes components, use a grid with two rows and one column and placing the two components in the grid cells (the customized app, `f14customapp` is included as a part of this example project file for your reference). You can follow:

- 1** In **Design View**, select and delete the default **Axes** component.
- 2** Go to **Component Library** on the left side of the window. From the **Component Library**, drag and drop a **Grid Layout** component under the **Containers** section in place of the removed Axes component. In **Component Browser** on the right, on the **Inspection** tab under the **Grid Layout**, update `ColumnWidth` and `RowHeight` properties to `'1x'` and `'1x,1x'`, respectively. This updates the grid to have two vertical cells. Drag two Axes components from the **Common** section in **Component Library** and place them in the two grid cells.
- 3** The external input references to `UIAxes` (the original Axes component) have to be updated to `UIAxes2`. If you added the top Axes first, `UIAxes` refers to `Logged Signals` and `UIAxes2` to `External Inputs`. You can check by switching to **Design View** and verifying which Axes gets the focus when the component is selected in **Component Browser**.
- 4** Now find and replace these occurrences in the **Code View**. Using the Find & Replace dialog, replace `UIAxes` with `UIAxes2` or additions. Once you complete the replacements, add the following line code to the `cbkSimulate(app, event)` function. After the line for `UIAxes`:
`app.SimulationHelper.UserInterface.clearGridAndLegend(app.UIAxes2)`.
- 5** Save the app.



Use the Modified App to Simulate the Model

Now that you have modified the app to show two axes, you can use that app to simulate the model. You can then compile and deploy the app. To simulate the app, click **Load Input MAT-file** and choose the `u.mat` file to attach an external input signal to Inport 1 of the f14 model. This activates the External Input drop down and displays the loaded signal, `Signal.RandomStickPosition` in the list box under the drop down. Select the loaded signal to display in the bottom Axes component. Click **Simulate**. After the simulation completes, the two Axes components update. You can observe the effect of the loaded input signal on the logged signals in the top Axes.



See Also

[deploytool](#) | [mcc](#) | [sim](#) | [simulink.compiler.configureForDeployment](#) | [simulink.compiler.genapp](#)

More About

- “Deploy an App Designer Simulation with Simulink Compiler” on page 1-2
- “Ways to Build Apps”

Generate and Deploy a MATLAB App for a Model

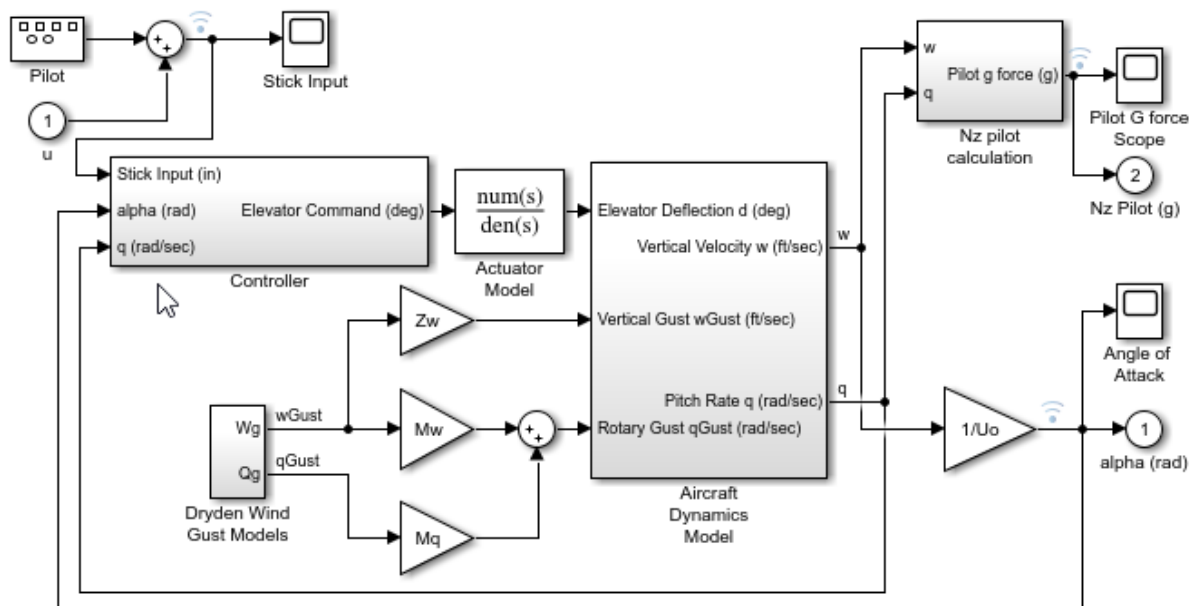
This example shows you how to use the `simulink.compiler.genapp` function to generate a MATLAB app for a model, that is deployable. Typically when a Simulink model is functionally complete, it is often used to run multiple simulations different input and parameter values. To try simulations for your model with different input and parameter values, you can generate a MATLAB App. You can also deploy this generated app for use outside of MATLAB.

This example illustrates the use of `simulink.compiler.genapp` function to generate a starter app for the model `f14`, using the generated app to tune the parameters of the model and simulate it, and customizing the app in the MATLAB App Designer.

Open the Model

The `simulink.compiler.example.AppGeneration` command loads the example project on your path. This project contains all the files required for this example including the model. Open the model `f14`.

```
simulink.compiler.example.AppGeneration;
open_system('f14')
```



F-14 Flight Control

Copyright 1990-2014 The MathWorks, Inc.

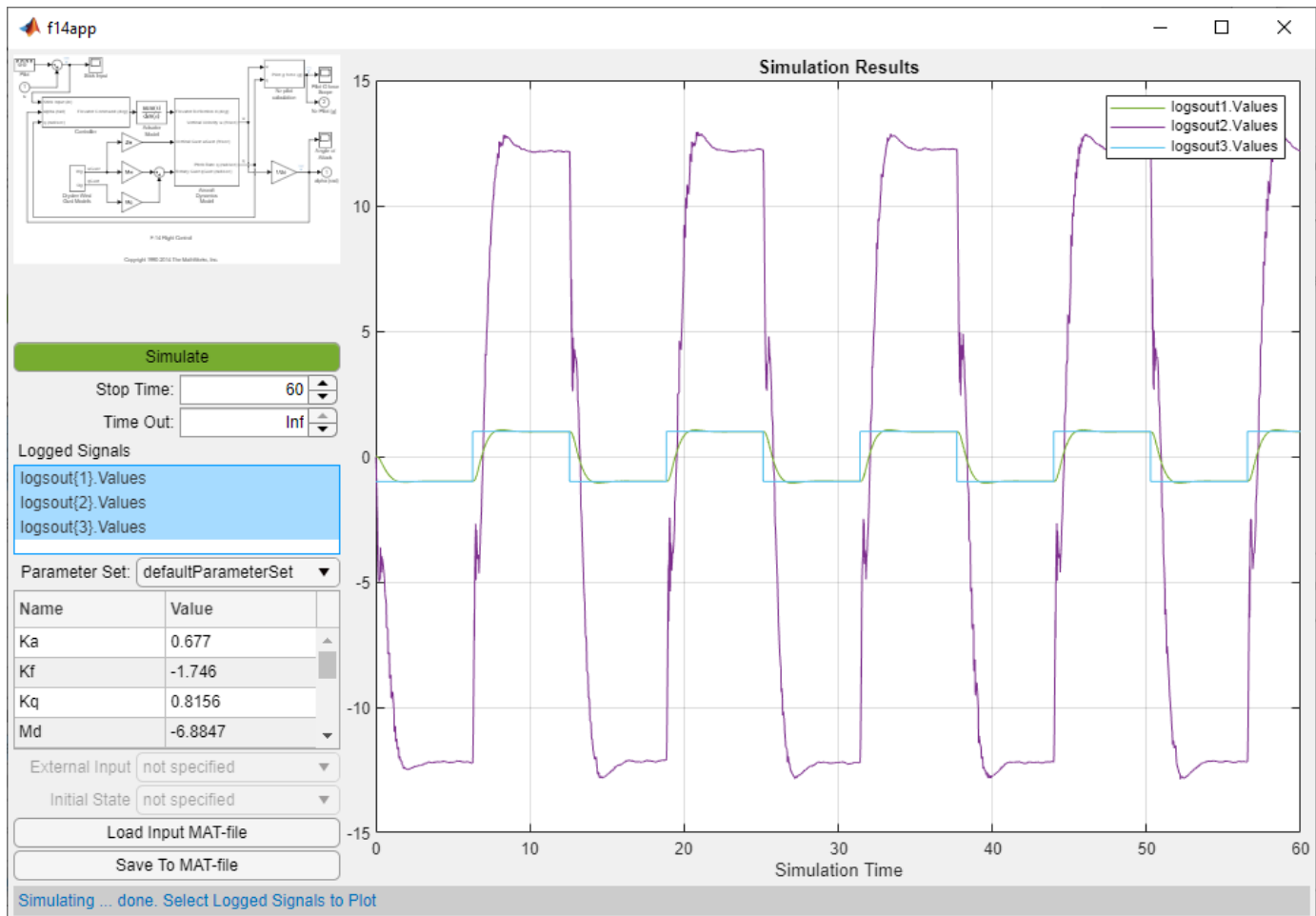
Generate a MATLAB App for the Model

Use the `simulink.compiler.genapp` function to generate an app for the `f14` model. Running the `simulink.compiler.genapp` function with the model name as an argument generates an App named `f14app`. Simulink Compiler uses a default template to generate the app. The generated app

provides an ability to tune the parameters and simulate the model for which the app is generated. The generated app also provides the plot of the simulation results. All the files are generated into the f14app directory.

```
simulink.compiler.genapp('f14', 'AppName', 'f14app');
```

After generating the app, Click **Simulate** to simulate the app.



Along with the app, the following artifacts are generated :

- f14app.mLapp file -- This file contains the code for the generated app. Open this file in App Designer for editing.
- Files starting with the default prefix -- Functions returning default values used by the app such as, model name, model image aspect ratio, model image file, and input MAT-file name.
- Model Image, (f14app_image.svg) -- Image of the Simulink Model.
- Inputs used in the simulation (f14app_inputs.mat) -- MAT file containing all the inputs that are used in the simulation of the model.
- App labels file setLabels.m -- File specifying label contents.
- Default Simulink logo (SimulinkLogo.png) -- File used as a placeholder for the model image.

- `pragma.m` directives file -- File used by Simulink Compiler to generate the deployable app.
- Set of MATLAB functions as M-files -- Files that the app uses to control user interface of the app.

Compile and Deploy the Generated App

You can use the MATLAB App Designer to compile and deploy the app. You can also use `deploytool`. For more information on compiling and deploying with App Designer, see [Develop Apps Using App Designer](#), [Web Apps and Application Compiler](#).

In this example, we compile the app, with the `mcc` command followed by the app name.

```
mcc -m f14App
```

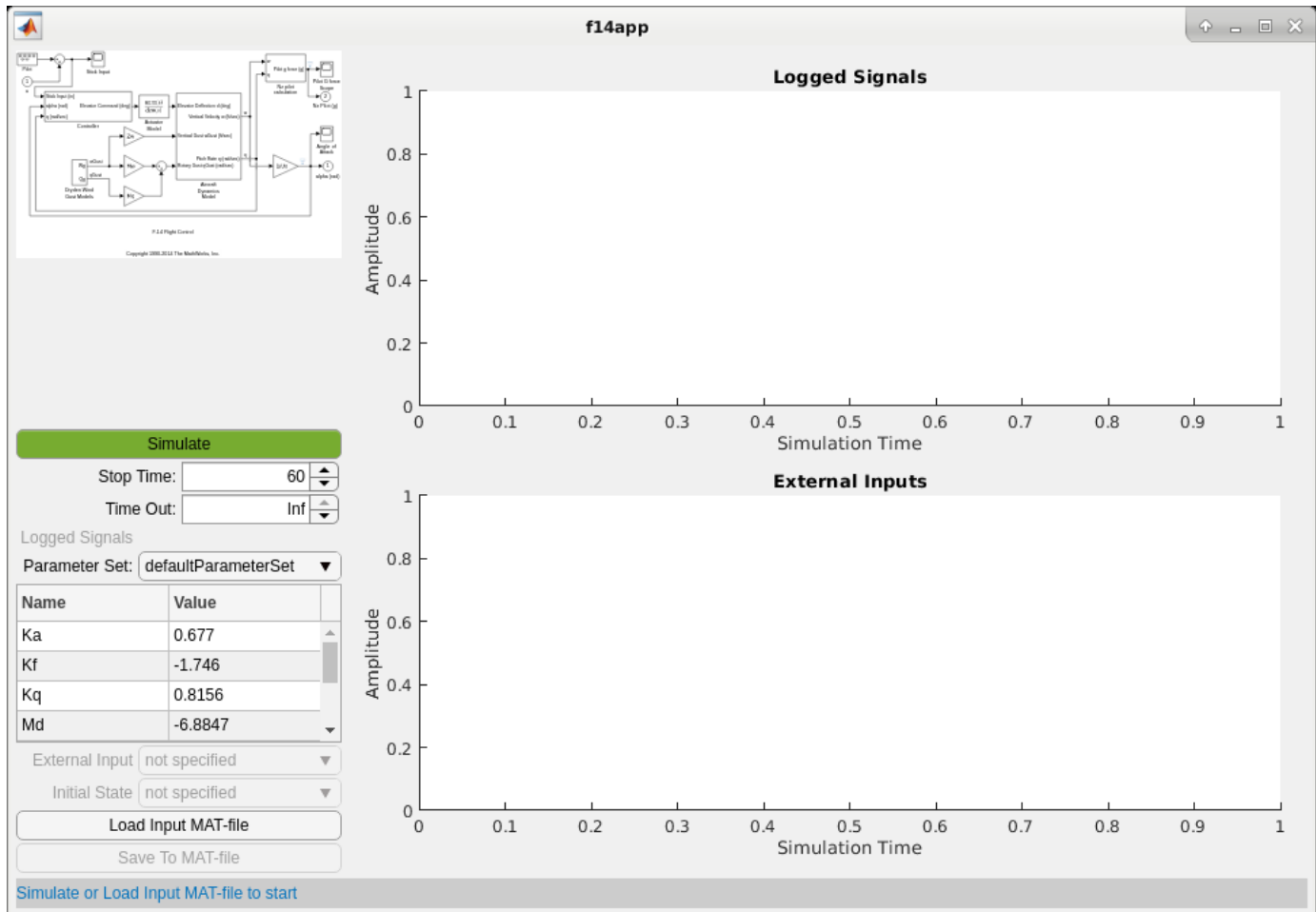
Customize the Generated App

You can also customize the generated app. To customize the app, use the App designer. The generated app `f14` is generic, but it allows you to easily customize it in the App Designer. In this section, we are going to replace the one axis in the generated app with two axes. Open the generated app in the app designer.

```
appdesigner('f14app');
```

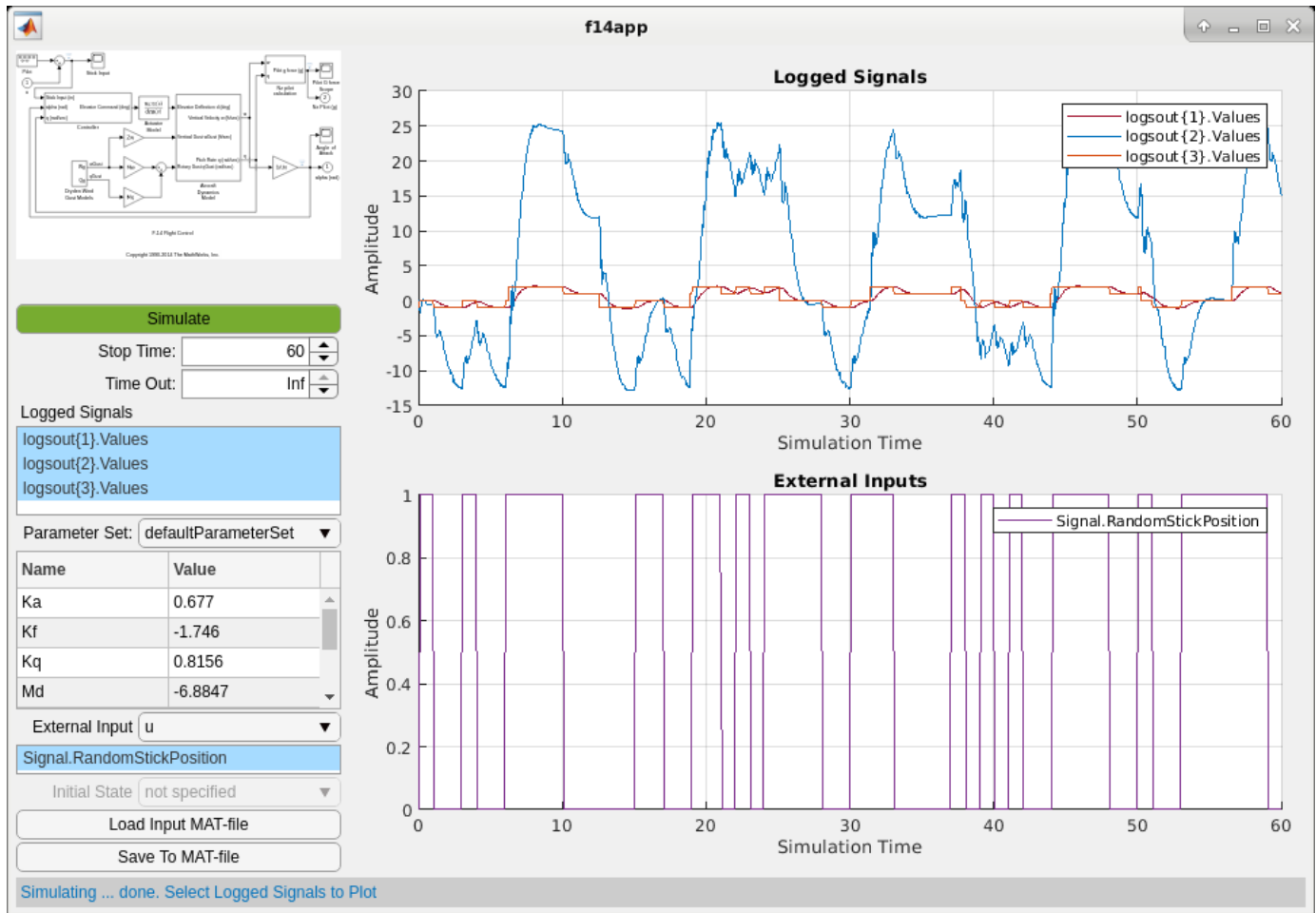
The idea of customization is to replace the Axes component with two Axes components. To get two Axes components, use a grid with two rows and one column and placing the two components in the grid cells (the customized app, `f14customapp` is included as a part of this example project file for your reference). You can follow:

- 1** In **Design View**, select and delete the default **Axes** component.
- 2** Go to **Component Library** on the left side of the window. From the **Component Library**, drag and drop a **Grid Layout** component under the **Containers** section in place of the removed Axes component. In **Component Browser** on the right, on the **Inspection** tab under the **Grid Layout**, update `ColumnWidth` and `RowHeight` properties to `'1x'` and `'1x,1x'`, respectively. This updates the grid to have two vertical cells. Drag two Axes components from the **Common** section in **Component Library** and place them in the two grid cells.
- 3** The external input references to `UIAxes` (the original Axes component) have to be updated to `UIAxes2`. If you added the top Axes first, `UIAxes` refers to `Logged Signals` and `UIAxes2` to `External Inputs`. You can check by switching to **Design View** and verifying which Axes gets the focus when the component is selected in **Component Browser**.
- 4** Now find and replace these occurrences in the **Code View**. Using the Find & Replace dialog, replace `UIAxes` with `UIAxes2` or additions. Once you complete the replacements, add the following line code to the `cbkSimulate(app, event)` function. After the line for `UIAxes`:
`app.SimulationHelper.UserInterface.clearGridAndLegend(app.UIAxes2)`.
- 5** Save the app.



Use the Modified App to Simulate the Model

Now that you have modified the app to show two axes, you can use that app to simulate the model. You can then compile and deploy the app. To simulate the app, click **Load Input MAT-file** and choose the `u.mat` file to attach an external input signal to Inport 1 of the f14 model. This activates the External Input drop down and displays the loaded signal, `Signal.RandomStickPosition` in the list box under the drop down. Select the loaded signal to display in the bottom Axes component. Click **Simulate**. After the simulation completes, the two Axes components update. You can observe the effect of the loaded input signal on the logged signals in the top Axes.



Simulation Callbacks for Deployable Applications

With certain functions in Simulink Compiler, you can register callbacks during simulation. The `simulink.compiler.setExternalInputsFcn` and `simulink.compiler.setExternalOutputsFcn` functions enable you to set the values at the root inport blocks and to obtain the values at the root outport blocks at every simulation step. With the `simulink.compiler.setPostStepFcn` function, you can register a callback that is invoked after every simulation step, thus using it to post process the outputs.

The following example uses the `simulink.compiler.setExternalOutputsFcn` and the `simulink.compiler.setPostStepFcn`, to provide an ongoing tracing of the simulation outputs.

Deploy an App with Live Simulation Results of Lorenz System

This example shows an app that uses callbacks for simulation inputs and outputs to view the simulation of a Simulink model of Lorenz System, and is then deployed with Simulink Compiler

Open and Examine the Project File

In this example, we use a Simulink Project that contains all the files required to run this example. The project contains a Simulink Model of Lorenz System and A MATLAB App, created in App Designer that simulates the model with different input and output values. To learn more about how to create an app using the App Designer, see “Create and Run a Simple App Using App Designer”.

`simulink.compiler.example.LorenzSystem`

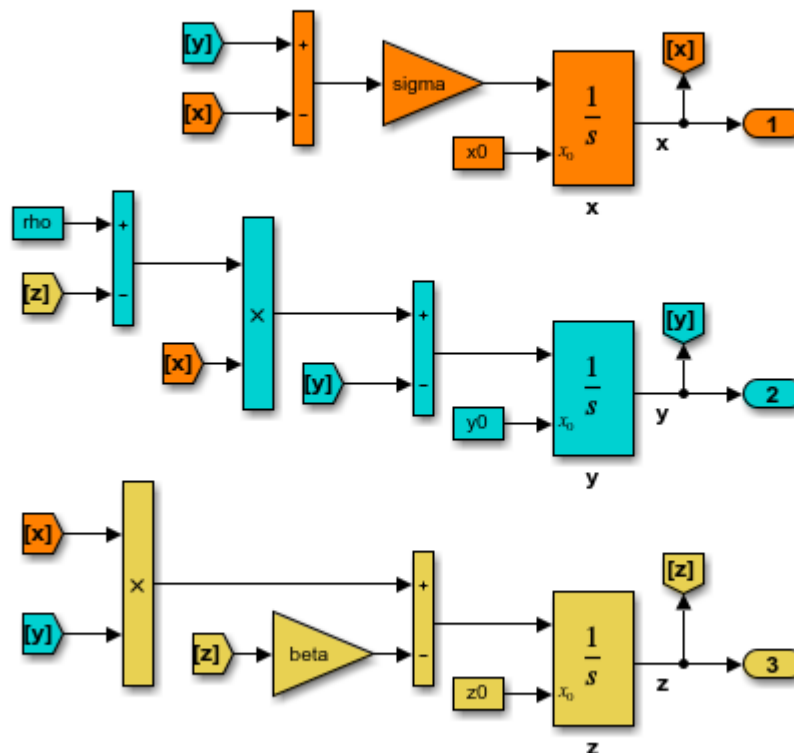
Project - LorenzSystem

Views: All | Project (8) | Modified (1) | Layout: Tree

Name	Status	SVN	Revision	Classification
AppStatus.m	✓	●	2	Design
LorenzAttractor.gif	✓	●	2	Other
LorenzEquations...	✓	●	2	Other
LorenzSystemAp...	✓	●	2	Design
LorenzSystemMo...	✓	●	2	Other
LorenzSystemMo...	✓	●	2	Design
onShutdown.m	✓	●	2	Design
onStartup.m	✓	●	2	Design

Labels: Classification

Details



app designer. The essential part of this app is the behavior of the **Simulate** button. It has the following salient parts: creating the `SimulationInput` object, configuring it for deployment, using simulation callbacks to read the output port data and plot the data at each time step. The following section explains how these three functions are used to see the live results of the simulation in the deployed app

Creating the `Simulink.SimulationInput` object

In the function `createSimulationInput`, we define an empty `Simulink.SimulationInput` object for the model. We use this `Simulink.SimulationInput` object to set simulation callbacks and variables for the model to simulate with.

The simulation callback functions are used to register the callbacks. The `simulink.compiler.setPostStepFcn` function registers a callback that is invoked after every simulation step. The `simulink.compiler.setExternalOutputsFcn` registers a callback that dynamically processes the values for every output port at root level of a model during simulation.

We use the `setVariable` method of the `Simulink.SimulationInput` object to provide the parameter values to the app. These values for the simulation are obtained from the edit fields of the UI of the app. To enable deployment of the app, we use the `simulink.compiler.configureForDeployment` function. (Comment the line of code that calls `simulink.compiler.configureForDeployment` function for faster debugging)

```
function simInp = createSimulationInput(app)
    % Create an empty SimulationInput object
    simInp = Simulink.SimulationInput('LorenzSystemModel');

    % Specify the simulation callbacks
    simInp = simulink.compiler.setPostStepFcn(simInp, @app.postStepFcn);
    simInp = simulink.compiler.setExternalOutputsFcn(simInp, @app.processOutputs);

    % Load the parameters values from the ui edit fields
    simInp = simInp.setVariable('rho', app.rhoUIC.Value);
    simInp = simInp.setVariable('beta', app.betaUIC.Value);
    simInp = simInp.setVariable('sigma', app.sigmaUIC.Value);
    simInp = simInp.setVariable('x0', app.x0UIC.Value);
    simInp = simInp.setVariable('y0', app.y0UIC.Value);
    simInp = simInp.setVariable('z0', app.z0UIC.Value);

    % Configure simInp for deployment
    % DEBUG TIP: Comment out the line below for
    % faster/easier debugging when running in MATLAB
    simInp = simulink.compiler.configureForDeployment(simInp);
end % createSimulationInput
```

Simulation Callback functions

The simulation callback functions register callbacks to enable you to read values from the output ports and to write values to the root input ports. These functions register callbacks at every simulation time step thus allowing you to view live results of the simulation.

The `processOutputs` Callback

The `simulink.compiler.setExternalOutputsFcn` line refers to the function `postprocessOutputs`. This is a callback function that processes the values for every root output port block of model during simulation. The `postprocessOutputs` function is called once per port,

and per the port's sample time. When `processOutputs` function is called, it reads the values for every root output block and caches away those values. The `postStepFcn` obtains the cached values to update the plot.

```
function processOutputs(app, opIdx, ~, data)
    % Called during sim to process the external output port data,
    % will be called once per port per its sample hit.
    switch opIdx
        case 1
            app.txyzBuffer.x = data;
        case 2
            app.txyzBuffer.y = data;
        case 3
            app.txyzBuffer.z = data;
        otherwise
            error(['Invalid port index: ', num2str(opIdx)]);
    end
end
```

The PostStepFcn Callback

The `postStepFcn` is a callback function that is invoked after every simulation step. The time argument is the time for the previous simulation step. This function obtains the cached output block values for every time, and passes those values to the `updateTrace` function to plot the cached values at simulation time.

```
function postStepFcn(app, time)
    % Called during sim after each simulation time step
    app.updateSimStats(time);
    if app.status == AppStatus.Starting
        app.switchStatus(AppStatus.Running);
        app.simStats.WallClockTimeAfterFirstStep = tic;
    end
    if app.stopRequested
        app.switchStatus(AppStatus.Stopping);
        stopRequestedID = [mfilename('class'), ':StopRequested'];
        throw(MException(stopRequestedID, 'Stop requested'));
    end
    %-----
    app.txyzBuffer.t = time;
    x = [app.txyzBuffer.x];
    y = [app.txyzBuffer.y];
    z = [app.txyzBuffer.z];
    app.updateTrace(x, y, z);
    app.updateMarker('head', x, y, z);
    %-----
    drawnow limitrate;
end % postStepFcn
```

Test Out the Application in App Designer

Before deploying the application, ensure that the app runs in the App Designer. Click **Simulate** to verify that the application works by simulating the model for different values.

Compile App for Deployment

. You can use the MATLAB App Designer to compile and deploy the app. You can also use `deploytool`. For more information on compiling and deploying with App Designer, see [Develop Apps Using App Designer, Web Apps and Application Compiler](#).

In this example, we compile the app, with the `mcc` command followed by the app name.

```
mcc -m LorenzSystemApp
```

See Also

`Simulink.SimulationInput` | `configureForDeployment` | `deploytool` | `mcc` | `sim` | `simulink.compiler.genapp` | `simulink.compiler.setExternalInputsFcn` | `simulink.compiler.setExternalOutputsFcn` | `simulink.compiler.setPostStepFcn`

More About

- “Simulink Compiler Workflow Overview”
- “Deploy an App Designer Simulation with Simulink Compiler” on page 1-2
- “Deploy Simulations with Tunable Parameters” on page 1-12

Deploy an App with Live Simulation Results of Lorenz System

This example shows an app that uses callbacks for simulation inputs and outputs to view the simulation of a Simulink model of Lorenz System, and is then deployed with Simulink Compiler

Open and Examine the Project File

In this example, we use a Simulink Project that contains all the files required to run this example. The project contains a Simulink Model of Lorenz System and A MATLAB App, created in App Designer that simulates the model with different input and output values. To learn more about how to create an app using the App Designer, see “Create and Run a Simple App Using App Designer”.

`simulink.compiler.example.LorenzSystem`

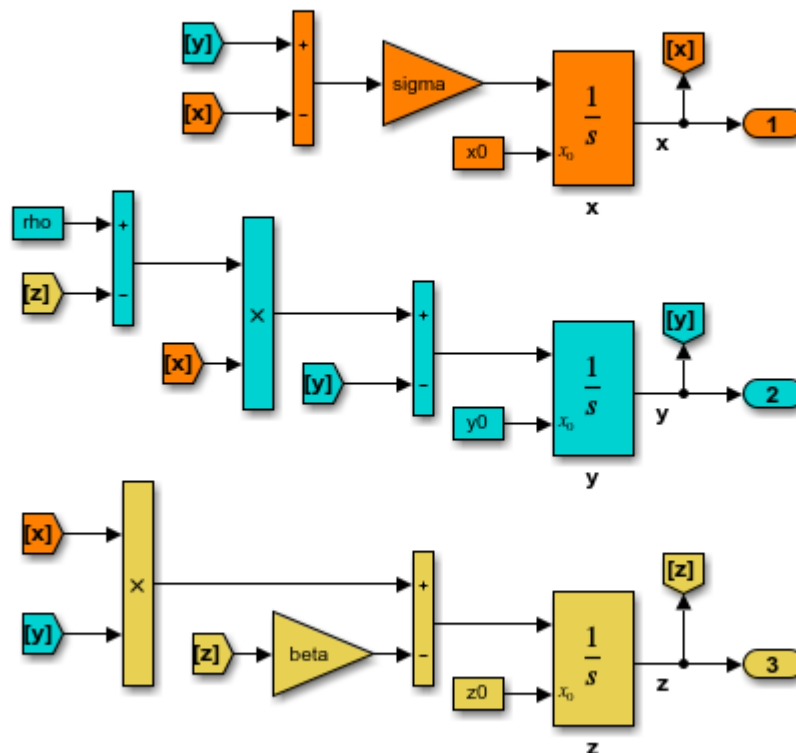
Project - LorenzSystem

Views: All | Project (8) | Modified (1) | Layout: Tree

Name	Status	SVN	Revision	Classification
AppStatus.m	✓	●	2	Design
LorenzAttractor.gif	✓	●	2	Other
LorenzEquations...	✓	●	2	Other
LorenzSystemAp...	✓	●	2	Design
LorenzSystemMo...	✓	●	2	Other
LorenzSystemMo...	✓	●	2	Design
onShutdown.m	✓	●	2	Design
onStartup.m	✓	●	2	Design

Labels: Classification

Details



app designer. The essential part of this app is the behavior of the **Simulate** button. It has the following salient parts: creating the `SimulationInput` object, configuring it for deployment, using simulation callbacks to read the output port data and plot the data at each time step. The following section explains how these three functions are used to see the live results of the simulation in the deployed app

Creating the `Simulink.SimulationInput` object

In the function `createSimulationInput`, we define an empty `Simulink.SimulationInput` object for the model. We use this `Simulink.SimulationInput` object to set simulation callbacks and variables for the model to simulate with.

The simulation callback functions are used to register the callbacks. The `simulink.compiler.setPostStepFcn` function registers a callback that is invoked after every simulation step. The `simulink.compiler.setExternalOutputsFcn` registers a callback that dynamically processes the values for every output port at root level of a model during simulation.

We use the `setVariable` method of the `Simulink.SimulationInput` object to provide the parameter values to the app. These values for the simulation are obtained from the edit fields of the UI of the app. To enable deployment of the app, we use the `simulink.compiler.configureForDeployment` function. (Comment the line of code that calls `simulink.compiler.configureForDeployment` function for faster debugging)

```
function simInp = createSimulationInput(app)
    % Create an empty SimulationInput object
    simInp = Simulink.SimulationInput('LorenzSystemModel');

    % Specify the simulation callbacks
    simInp = simulink.compiler.setPostStepFcn(simInp, @app.postStepFcn);
    simInp = simulink.compiler.setExternalOutputsFcn(simInp, @app.processOutputs);

    % Load the parameters values from the ui edit fields
    simInp = simInp.setVariable('rho', app.rhoUIC.Value);
    simInp = simInp.setVariable('beta', app.betaUIC.Value);
    simInp = simInp.setVariable('sigma', app.sigmaUIC.Value);
    simInp = simInp.setVariable('x0', app.x0UIC.Value);
    simInp = simInp.setVariable('y0', app.y0UIC.Value);
    simInp = simInp.setVariable('z0', app.z0UIC.Value);

    % Configure simInp for deployment
    % DEBUG TIP: Comment out the line below for
    % faster/easier debugging when running in MATLAB
    simInp = simulink.compiler.configureForDeployment(simInp);
end % createSimulationInput
```

Simulation Callback functions

The simulation callback functions register callbacks to enable you to read values from the output ports and to write values to the root input ports. These functions register callbacks at every simulation time step thus allowing you to view live results of the simulation.

The `processOutputs` Callback

The `simulink.compiler.setExternalOutputsFcn` line refers to the function `postprocessOutputs`. This is a callback function that processes the values for every root output port block of model during simulation. The `postprocessOutputs` function is called once per port,

and per the port's sample time. When `processOutputs` function is called, it reads the values for every root output block and caches away those values. The `postStepFcn` obtains the cached values to update the plot.

```
function processOutputs(app, opIdx, ~, data)
    % Called during sim to process the external output port data,
    % will be called once per port per its sample hit.
    switch opIdx
        case 1
            app.txyzBuffer.x = data;
        case 2
            app.txyzBuffer.y = data;
        case 3
            app.txyzBuffer.z = data;
        otherwise
            error(['Invalid port index: ', num2str(opIdx)]);
    end
end
```

The PostStepFcn Callback

The `postStepFcn` is a callback function that is invoked after every simulation step. The time argument is the time for the previous simulation step. This function obtains the cached output block values for every time, and passes those values to the `updateTrace` function to plot the cached values at simulation time.

```
function postStepFcn(app, time)
    % Called during sim after each simulation time step
    app.updateSimStats(time);
    if app.status == AppStatus.Starting
        app.switchStatus(AppStatus.Running);
        app.simStats.WallClockTimeAfterFirstStep = tic;
    end
    if app.stopRequested
        app.switchStatus(AppStatus.Stopping);
        stopRequestedID = [mfilename('class'), ':StopRequested'];
        throw(MException(stopRequestedID, 'Stop requested'));
    end
    %-----
    app.txyzBuffer.t = time;
    x = [app.txyzBuffer.x];
    y = [app.txyzBuffer.y];
    z = [app.txyzBuffer.z];
    app.updateTrace(x, y, z);
    app.updateMarker('head', x, y, z);
    %-----
    drawnow limitrate;
end % postStepFcn
```

Test Out the Application in App Designer

Before deploying the application, ensure that the app runs in the App Designer. Click **Simulate** to verify that the application works by simulating the model for different values.

Compile App for Deployment

. You can use the MATLAB App Designer to compile and deploy the app. You can also use `deploytool`. For more information on compiling and deploying with App Designer, see [Develop Apps Using App Designer, Web Apps and Application Compiler](#).

In this example, we compile the app, with the `mcc` command followed by the app name.

```
mcc -m LorenzSystemApp
```