Wolfram *SystemModeler*™

Getting Started

WOLFRAM

Vertrieb durch:
ADDITIVE GmbH • Max-Planck-Straße 22b • 61381 Friedrichsdorf
http://additive-mathematica.de/ • eShop: http://eshop.additive-net.de
Verkauf: +49-6172-5905-134 mathematica@additive-net.de
Support: +49-6172-5905-20 support@additive-net.de

ADDITIVE
SOFT- & HARDWARE FÜR TECHNIK & WISSENSCHAFT

Contents

1 Introduction **1**

2 Hello World **5**

2.1 Hello World Model 5
2.1.1 Exercise 11

3 Multidomain—A Servo Mechanism **13**

3.1 DC Motor 13
3.2 Stiff and Weak Axis 20
3.2.1 Exercise 23
3.3 Control System 23
3.4 Sensitivity Analysis 28

4 Component-Based—Simple Circuit **31**

4.1 Block-Based Circuit 31
4.2 Component-Based Circuit 34
4.2.1 Exercise 38

5 Custom Component—Chain Pendulum **39**

5.1 Chain Link Component 39
5.2 Chain Pendulum Model 43
5.2.1 Exercise 44

6 External Function—Chirp Signal 45

6.1 Chirp Function . . . 45
6.2 Modeling . . . 46
6.2.1 Exercise . . . 49

7 Hierarchical Model—Tank System 51

7.1 Flat Tank . . . 51
7.2 Component-Based Tank . . . 53
7.2.1 Interfaces . . . 55
7.2.2 Tank Components . . . 56
7.2.3 Controllers . . . 57
7.2.4 Small Tank System . . . 59
7.3 Tank with Continuous PID Controller . . . 60
7.4 Three Tank System . . . 62

8 System—Inverted Pendulum 63

8.1 Inverted Pendulum . . . 63

9 Modelica Aspects and Modeling Advice 65

9.1 Initial Values . . . 65
9.1.1 Start Attribute . . . 65
Guess Values . . . 66
9.1.2 Initial Equations and Algorithms . . . 66
9.2 Events . . . 67
9.3 The MultiBody Library . . . 68
9.3.1 Initial Values . . . 68
9.3.2 Angle and Position of Objects . . . 69
9.3.3 Animations . . . 69
9.3.4 CAD Shapes . . . 70
9.4 General Advice . . . 71

Chapter 1: **Introduction**

This document contains examples that are useful for getting started with Wolfram *SystemModeler*™. The examples have a detailed step by step description of how to build and simulate the models.

It is recommended that you go through the examples in the order given. Note that all examples are also available in the `IntroductoryExamples` library. You can browse the package structure of the `IntroductoryExamples` library by using the **Class Browser**.

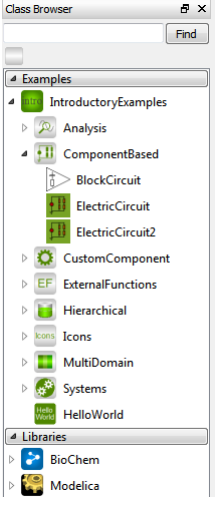

Figure 1-1: Browsing the package structure of the `IntroductoryExamples` library using the **Class Browser**.

Double-clicking the name of a package will open the package as a new tree and show its contents in the **Class Browser**. Double-clicking on the name of a model will open the model in a class window.

Additional information about the models in the `IntroductoryExamples` library is integrated into the packages and models and may be viewed by right-clicking any package or model in the **Class Browser** and choosing **View Documentation** from the popup menu.

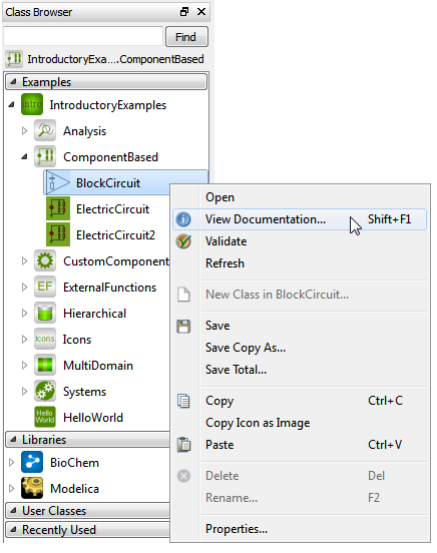

Figure 1-2: Viewing the documentation of a model.

The documentation of the classes will be shown in the **Class Documentation Browser**.

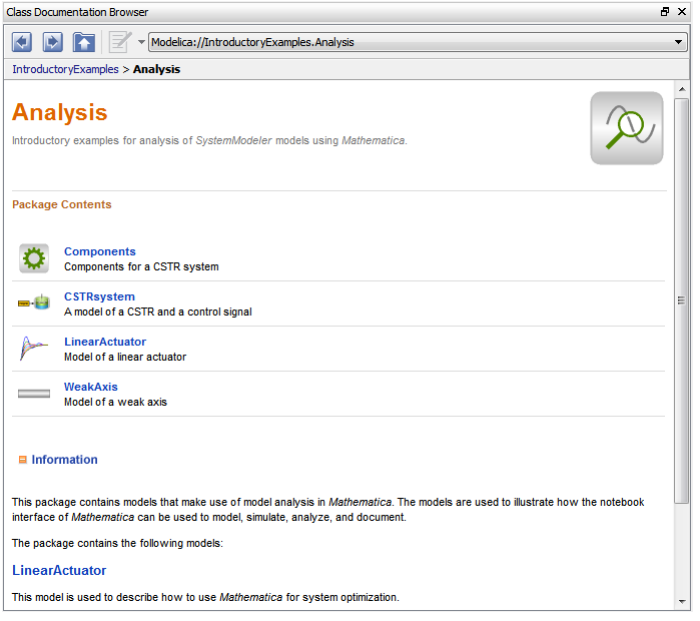

Figure 1-3: The documentation of the `IntroductoryExamples.Analysis` package.

Chapter 2: **Hello World**

The most basic Modelica model is a differential equation. In this example, a differential equation is implemented and simulated. Also, the process of creating an icon representing the model graphically is described in detail.

2.1 Hello World Model

There is a long tradition that the first example in any computer language is a trivial program printing the string "Hello World". Since Modelica, the language used in *SystemModeler*, is an equation-based language, printing a string does not make much sense. Instead our Hello World Modelica program solves a trivial differential equation:

$$\dot{x} = -x$$

The variable x in this equation is a dynamic variable (and a state variable) whose value can change over time. The time derivative is the derivative of x, written as `der(x)` in Modelica. All Modelica programs consist of a class declaration (`block`, `model`, `package`, etc.). In this example we will declare the program as a model.

We begin by creating a new model at the top level of the Modelica package hierarchy, i.e. the model will not be located inside a package. Choose **New Class** from the **File** menu.

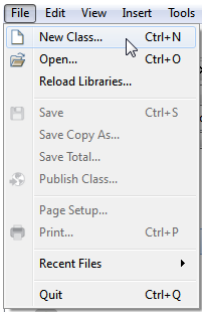

Figure 2-1: Choosing **New Class** from the **File** menu.

This will open the **New Class** dialog box, in which we will specify a name and description for the model. Give the model the name "HelloWorld" and the description "A differential equation".

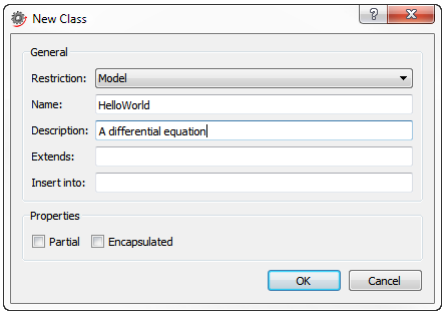

Figure 2-2: Specifying a name and description for a new model.

After clicking the **OK** button, the model will be created and become visible in the **Class Browser**. At the same time, the model will also be opened in a class window. Click the **Modelica Text View** button in the toolbar to switch to the **Modelica Text View** of the class window.

Figure 2-3: The **Modelica Text View** button in the toolbar of the *Model Center*.

The textual representation of the model should look as follows.

```
model HelloWorld "A differential equation"
  ¤;
end HelloWorld;
```

The symbol ¤ in the second row contains hidden graphical information about the model and is automatically updated whenever you edit the model in any of the graphical views of the class window. Note that the description that we entered in the dialog box has been added to the model. Now it is just a matter of adding the variable and the equation. We will do that by editing the definition of the model directly in the **Modelica Text View**.

```
model HelloWorld "A differential equation"
  ¤;
  Real x(start=1);
equation
  der(x)=-x;
end HelloWorld;
```

Note that when we declare the variable we also set its initial value to `1` by specifying a value for its parameter `start`.

The `HelloWorld` model is now ready. Before simulating the model, we may want to verify its correctness by clicking the **Validate Class** button in the toolbar.

Figure 2-4: The **Validate Class** button in the toolbar of the *Model Center*.

This will generate a report in the **Messages** view, located below the class window.

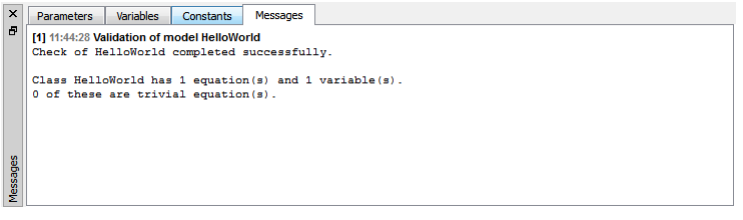

Figure 2-5: The **Messages** view of the *Model Center*.

If everything was typed in correctly, you should find a report similar to the one in Figure 2-5.

To perform the simulation of the model, we need to start *Simulation Center*, the simulation environment of *SystemModeler*. Click the **Simulation Center** button in the toolbar.

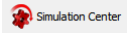

Figure 2-6: The **Simulation Center** button in the toolbar of the *Model Center*.

Simulation Center will start, and the `HelloWorld` model will automatically be translated into an executable. An experiment is created for the `HelloWorld` model in the **Experiment Browser** of *Simulation Center*.

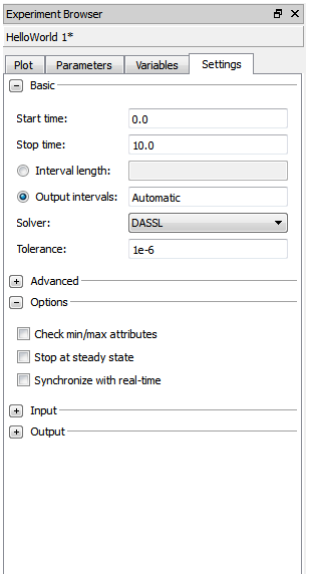

Figure 2-7: The **Settings** view of the `HelloWorld` experiment in *Simulation Center*.

In the **Experiment Browser** you can specify simulation settings, parameter values, and initial values for variables, but we will leave it as is for now. Instead we will click the **Simulate** button to start the simulation.

Figure 2-8: The **Simulate** button in the toolbar of *Simulation Center*.

After the simulation is completed, the **Plot** view of the **Experiment Browser** becomes visible. Click the checkbox in front of the variable *x* to plot the result.

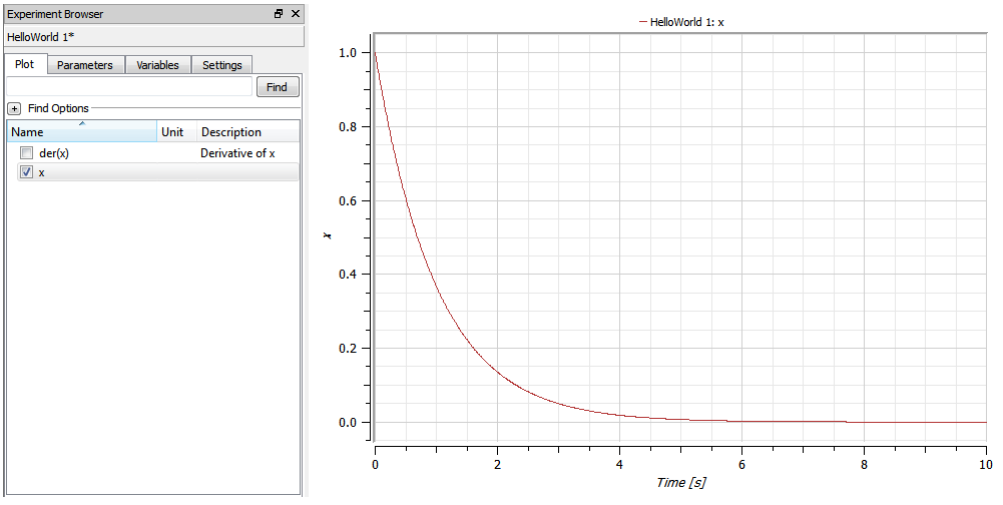

Figure 2-9: Plotting the variable *x* of the `HelloWorld` model in *Simulation Center*.

We will now return to the *Model Center* in order to create an icon for the model. Switch to the icon view of the class window by clicking the **Icon View** button in the toolbar.

Figure 2-10: The **Icon View** button in the toolbar of the *Model Center*.

To create an icon, we will use the drawing tools available in the toolbar of the *Model Center*.

Figure 2-11: The drawing tools in the toolbar of the *Model Center*.

By choosing the rectangle tool, it is possible to draw rectangles. Draw a rectangle covering the white area of the icon view. Double-click the rectangle to view and edit the properties of the rectangle.

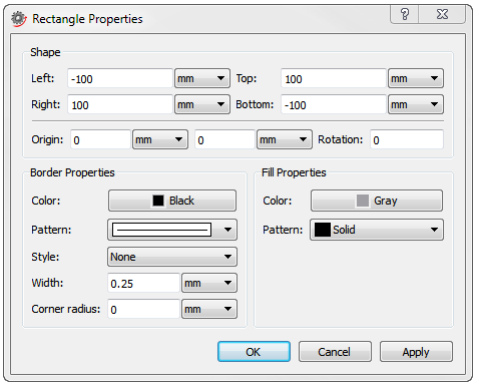

Figure 2-12: Editing the properties of a rectangle.

Change the fill color to **Gray**, select **Solid** as the fill pattern and click the **OK** button. Next, press the **Esc** key to clear the selection in the icon view. Finally, choose the **Text Tool** and draw a text item covering the entire rectangle, and change the text to "Hello World" in the **Text Properties** dialog box. The reason why it was necessary to clear the selection before drawing the text item deserves an explanation. Without clearing the selection, we would have ended up moving the rectangle instead of adding a text item, as all drawing tools can also be used to move selected items.

The icon of the model should now look similar to the one below.

Figure 2-13: The icon of the `HelloWorld` model.

The `HelloWorld` model will from now on be represented by this icon everywhere it is used.

2.1.1 Exercise

Change the model equation, for instance by adding a parameter and trying out the result.

Chapter 3: **Multidomain—A Servo Mechanism**

This example shows how to develop a servo mechanism model step by step in *SystemModeler*. It illustrates the multi-engineering capabilities and shows how you can use *Simulation Center* to analyze models created in *Model Center*, synthesize controllers, and carry out comparison studies.

3.1 DC Motor

A simple dynamic model of a controlled DC motor consists of a variable voltage source, a resistor, an inductor, and an electromotoric force element representing the coupling between electric energy and mechanical energy provided by the magnetic field in the DC motor. The motor axis is represented by a rotating mass or inertia.

All of these components can be found in the Modelica Standard Library, included with *SystemModeler*. With the help of drag-and-drop, they can be used to compose the model as illustrated in the figure below.

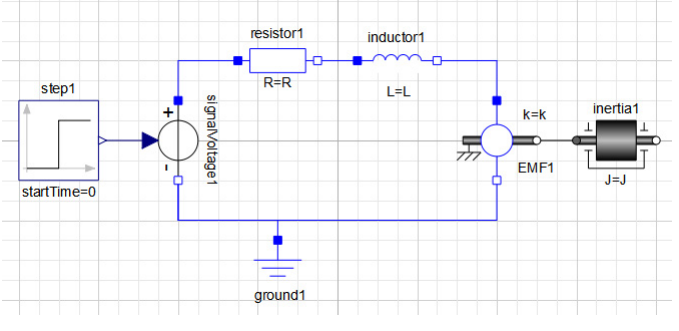

Figure 3-1: The **Diagram View** of a DC motor in the *Model Center*.

To build this model, we need to create a new model, find the appropriate components, drag and drop the components into the diagram area, and finally connect the components using the **Connection Line Tool**.

We begin by creating a new model with the name `DCMotor`. The components that we will use are all available in the Modelica Standard Library. To locate the components, we can either search for them, or if we know their exact location, open the package that contains them in the **Class Browser**. We will show how to do both.

To locate the `Step` source component, we will use the **Class Browser** to search for it. Type "`step`" (without the quotation marks) in the text box of the **Class Browser** and press the **Enter** key or click the **Find** button to the right of the text box.

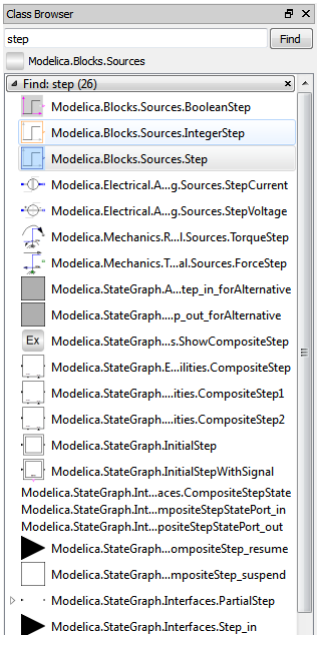

Figure 3-2: Searching for a step source component using the **Class Browser** in the *Model Center*.

If everything went well, you should have at least 24 matches for "step" in the **Class Browser**. The component we want to use is the `Modelica.Blocks.Sources.Step` component, highlighted in the figure above.

To add this component to our `DCMotor` model, drag it from the **Class Browser** and drop it on the **Diagram View** of the class window.

The signal voltage component is located in the `Modelica.Electrical.Analog.Sources` package. As we know the exact location of the component, we will use the tree view of the **Class Browser** and expand the branches of the tree all the way down to the branch that represents the package `Sources`, in which the component is located.

Start by expanding the `Modelica` package. This is done by clicking the symbol to the left of the package icon and name.

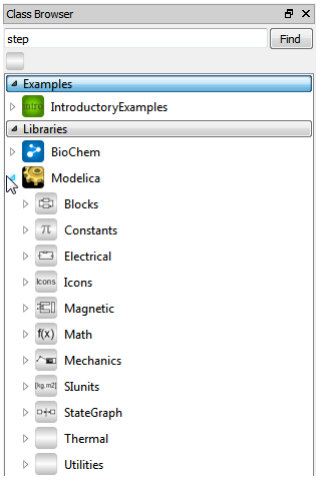

Figure 3-3: Expanding the `Modelica` package in the **Class Browser**.

As you can see, the `Modelica` package contains several packages. We will continue by expanding the `Electrical` package, followed by the `Analog` package, and finally the `Sources` package, in which we will find the signal voltage component.

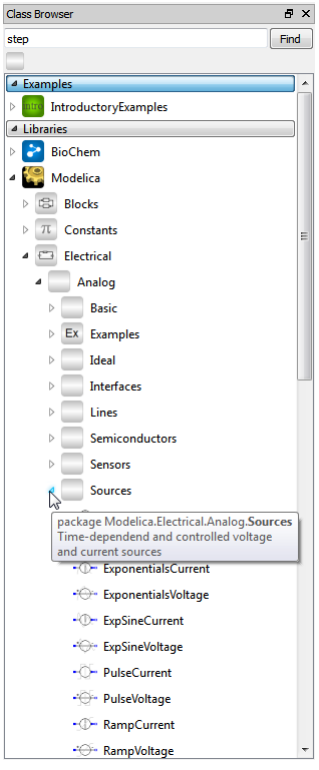

Figure 3-4: Expanding the `Modelica.Electrical.Analog.Sources` package.

Add the `SignalVoltage` component to the `DCMotor`, by dragging it to the **Diagram View** of the class window.

We have now added two of the seven components. The remaining four electrical components (`Resistor`, `Inductor`, `Ground`, and `EMF`) can all be found in the `Modelica.Electrical.Analog.Basic` package. As we already have the `Modelica.Electrical.Analog` package expanded, we can easily locate the `Basic` package and expand it in order to find the resistor, ground, inductor, and EMF components.

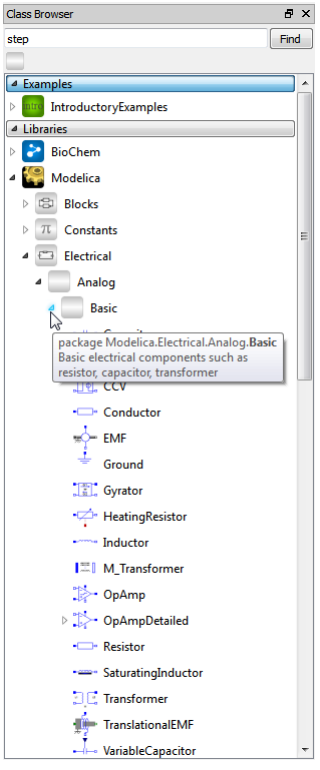

Figure 3-5: Expanding the `Modelica.Electrical.Analog.Basic` package.

When you have added the electrical components to the `DCMotor` model, there is only one component left to add, the inertia. It is located in the `Modelica.Mechanics.Rotational.Components` package. You can choose if you want to search for it or browse to it directly by expanding the `Modelica`, `Mechanic`, `Rotational`, and `Components` packages.

Once you have added the inertia component, all that remains to complete the model of the DC motor is to connect the components. Components are connected using the **Connection Line Tool**.

Figure 3-6: The **Connection Line Tool** in the toolbar of the *Model Center*.

For instance, to connect the ground to the negative pin of the signal voltage component, place the mouse cursor above the ground pin, press the left mouse button and hold it down while moving the mouse cursor to the negative pin of the signal voltage component. To make the connection, release the mouse button.

Continue connecting all the components until the **Diagram View** of the `DCMotor` resembles the picture in Figure 3-1.

While dropping and connecting the components, *Model Center* generates the Modelica code corresponding to the actions. Switch to the **Modelica Text View** to view the textual representation of the model. In the textual representation of the model, each component is declared, and each connection between two components is represented by connect equations in the equation section.

```
model DCMotor
  Modelica.Blocks.Sources.Step step;
  Modelica.Electrical.Analog.Sources.SignalVoltage
signalVoltage1;
  Modelica.Electrical.Analog.Basic.Resistor resistor1;
  Modelica.Electrical.Analog.Basic.Inductor inductor1;
  Modelica.Electrical.Analog.Basic.EMF EMF1;
  Modelica.Mechanics.Rotational.Components.Inertia inertia1;
  Modelica.Electrical.Analog.Basic.Ground ground1;

equation
  connect(EMF1.flange,inertia1.flange_a);
  connect(EMF1.n,signalVoltage1.n);
  connect(signalVoltage1.n,ground1.p);
  connect(inductor1.n,EMF1.p);
  connect(resistor1.n,inductor1.p);
  connect(signalVoltage1.p,resistor1.p);
  connect(step1.y,signalVoltage1.v);
end DCMotor;
```

The order of the declarations and equations depends on in which order you dropped the components and made the connections. Therefore, the order of the declarations and equations may be slightly different in your model. Also, for readability, all graphical annotations have been removed from the definition of the `DCMotor` above.

The `DCMotor` model is now complete and possible to simulate. Click the **Simulation Center** button to start *Simulation Center*. In *Simulation Center*, set the simulation time to 25 seconds by editing the **Stop time** in the **Settings** view of the `DCMotor` experiment.

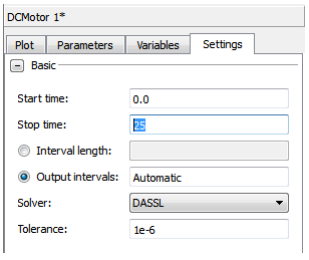

Figure 3-7: Setting the simulation time to 25 seconds for the `DCMotor` model.

Start the simulation and, when completed, select the variables to plot in the **Experiment Browser** as illustrated in the figure below.

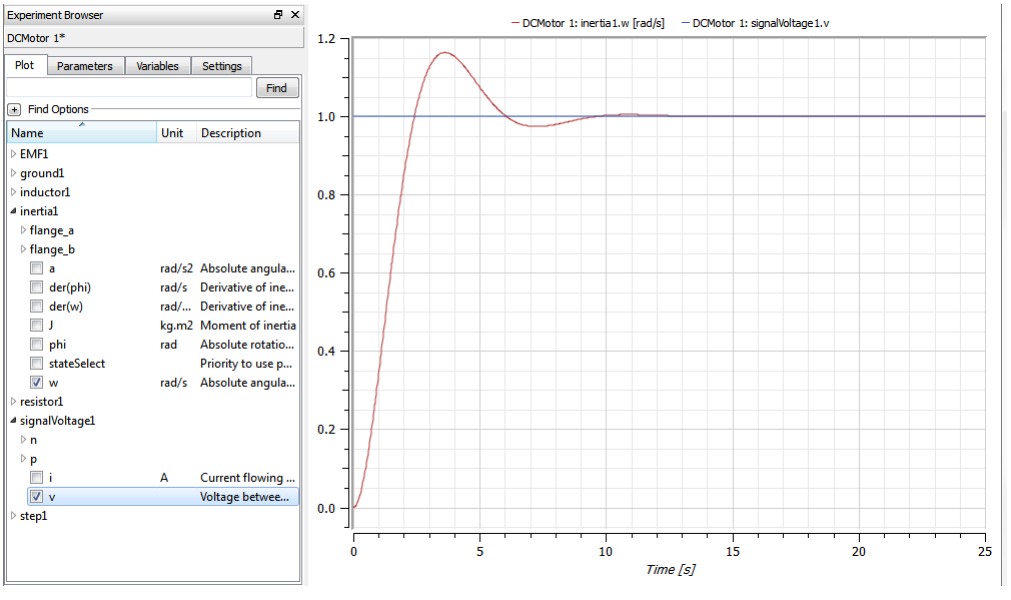

Figure 3-8: Plotting `inertia1.w` and `signalVoltage1.v` for the `DCMotor` model with default parameter values.

Finally, we get the result, with the plot of `inertia.w` and `signalVoltage1.v` versus time.

It is also easy to change parameter values in order to modify the system behavior. We will change the resistance of the resistor, the inductance of the inductor, and the moment of the inertia in order to yield a damped step response instead of an oscillative step response.

Switch to the **Parameters** view in the **Experiment Browser**. To edit a parameter value in the **Parameters** view, double-click the current value. Set the resistance of `resistor1` to 10 ohm, the inductance of `inductor1` to 0.1 H, and the moment of `inertia1` to 0.3 kgm^2.

Simulate the model again and study the updated plot of the angular velocity of the inertia.

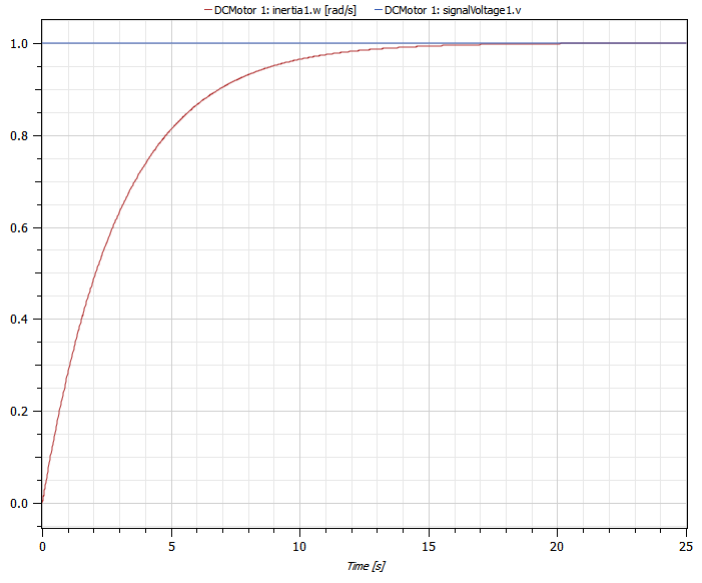

Figure 3-9: Plotting `inertia1.w` and `signalVoltage1.v` for the `DCMotor` model with customized parameter values.

3.2 Stiff and Weak Axis

In this section we will begin by developing a stiff axis model, study its step response by adding a step torque as illustrated below, and show how the axis can be more accurately modeled by including an additional weakness to the stiff axis model.

We begin by developing the stiff axis model. The components (`Step`, `Torque`, `Inertia`, and `IdealGear`) of the model can all be found by expanding the `Modeli-`

`ca.Blocks.Sources`, `Modelica.Rotational.Sources` and `Modelica.Mechanics.Rotational.Components` packages in the **Class Browser**, or by simply searching for them. You can give the model any name you want. The different stages of the model are also available in the `IntroductoryExamples.MultiDomain` package. Note however that all models in `IntroductoryExamples` are read-only models and cannot be modified, so there is a point in developing the models yourself if you want to be able to do everything that is involved in the steps of this example.

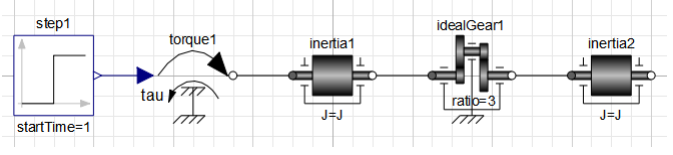

Figure 3-10: The **Diagram View** of the `IntroductoryExamples.MultiDomain.StiffAxis` model.

By selecting the `idealGear1` component, we are able to edit the parameters of the component in the **Parameters** view, located at the bottom part of the editor.

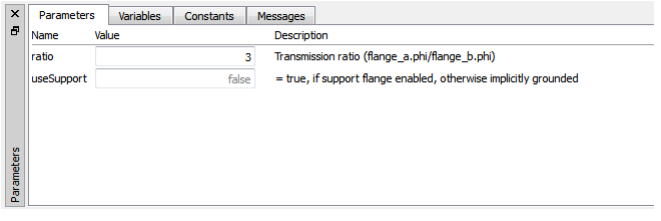

Figure 3-11: Editing the transmission ratio of an ideal gear component in the *Model Center*.

Give the gear ratio parameter a value of 3. This means that angles and angular velocity are amplified three times and the torque is attenuated by a factor of three from one side of the gear to the other. Also, change the start time of the step source by changing the value of the parameter `startTime` to 1 second. The text in the `idealGear1` icon in the **Diagram View** should change from "ratio = ratio" to "ratio = 3".

After simulating the system for 6 seconds, we observe that a constant torque results in a constant angular acceleration, i.e. a ramp in angular velocity and a square curve for the angle of the axis, as seen below.

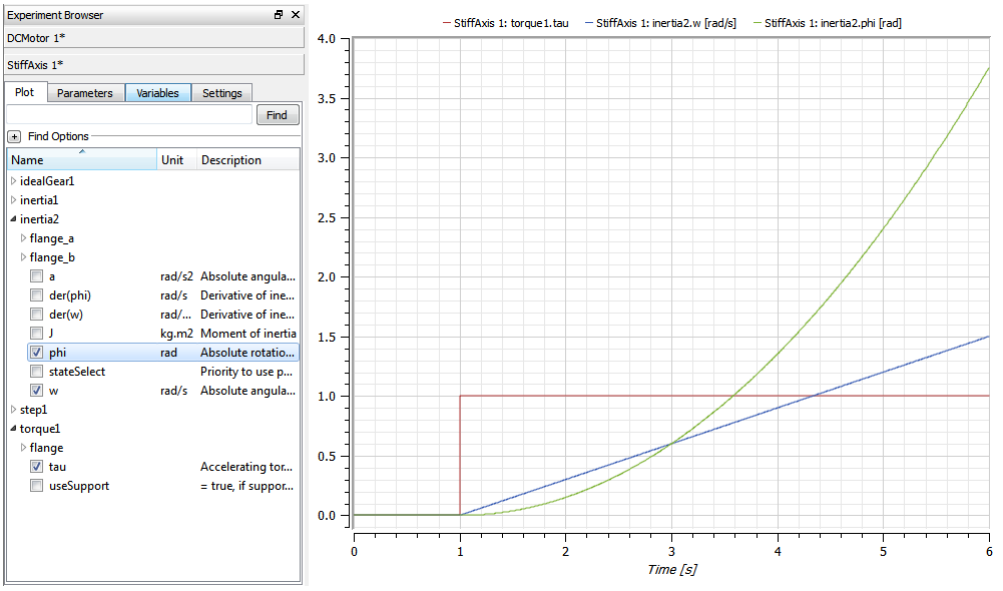

Figure 3-12: Plotting the torque, the angle of `inertia2`, and the angular velocity of `inertia2` for the `IntroductoryExamples.MultiDomain.StiffAxis` model.

By including an additional weakness, the axis can be more accurately modeled. This is possible by substituting the above axis model with a model consisting of two rotating masses connected by a torsion spring, according to the figure below. The torsion spring is found in the `Modelica.Mechanics.Rotational.Components` package.

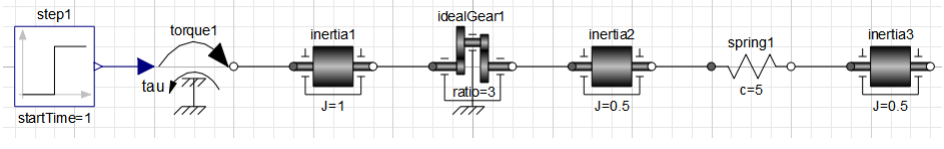

Figure 3-13: The **Diagram View** of the `IntroductoryExamples.MultiDomain.WeakAxis` model.

Notice that `inertia1` and `inertia2` have been given a moment of 0.5 kgm^2, and the spring constant of `spring1` is set to 5 Nm/rad. We simulate this subsystem for 6 seconds and then study the result. A comparison with the stiff axis model shows that we have similar behavior but with an added deflection. Note that `inertia3`, and not `inertia2` as earlier, is the last element of the axis. Therefore, we plot the rotational velocity and angle for `inertia3` in order to do a fair comparison.

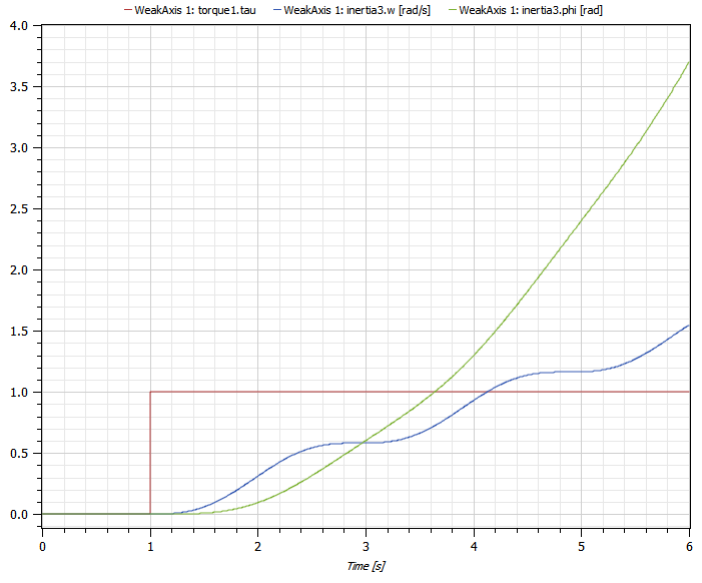

Figure 3-14: Plotting the torque, the angle of `inertia3`, and the angular velocity of `inertia3` for the `IntroductoryExamples.MultiDomain.WeakAxis` model.

3.2.1 Exercise

Make a simple DC motor with a torsional spring to the outgoing shaft and another inertia element. Simulate and study the results. Adjust some parameters and compare results. You may also want to add an input torque and connect it to `inertia2`, then study the system.

3.3 Control System

We end this chapter by developing a stiff and weak servo mechanism, using the DC motor model and axis models developed earlier in this chapter.

The structure of the control system is shown in the schematic picture below. This system consists of an input signal, a sensor, a feedback loop, and a regulator. The physical system consists of the DC motor and one of the axis systems. Since the physical system has negative static gain, the PI gain must also be negative.

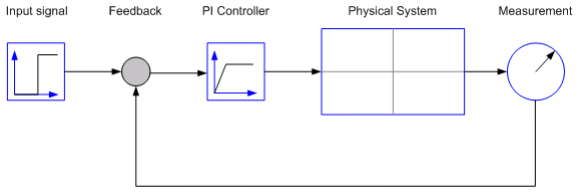

Figure 3-15: Simplified representation of a control system.

We connect all three subsystems as seen in the figure above. The default choices of regulator parameters are $k = 1$ and $T = 1$, where the PI regulator transfer function is:

$$G_{PI} = kTs + \frac{1}{Ts}$$

We begin by developing a control system for the DC motor and the stiff axis developed earlier. As seen in the figure below, three new components are introduced: a feedback component, a PI controller, and a speed sensor.

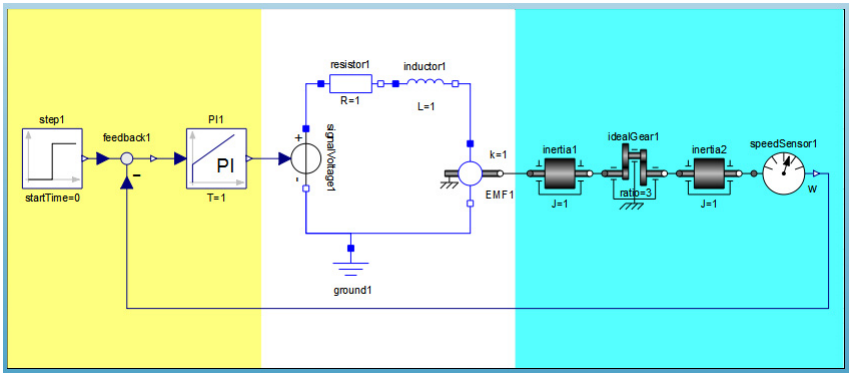

Figure 3-16: The **Diagram View** of the `IntroductoryExamples.MultiDomain.StiffServoMechanism` model.

These components can be found in the following packages:

- The PI controller is found in the `Modelica.Blocks.Continuous` package.
- The feedback component is found in the `Modelica.Blocks.Math` package.
- The speed sensor is found in `Modelica.Mechanics.Rotational.Sensors`.

When simulating this model, we will pay attention to the response for the angular velocity of both the motor axis and the gear axis shown in the figure below. The model was simulated for 25 seconds.

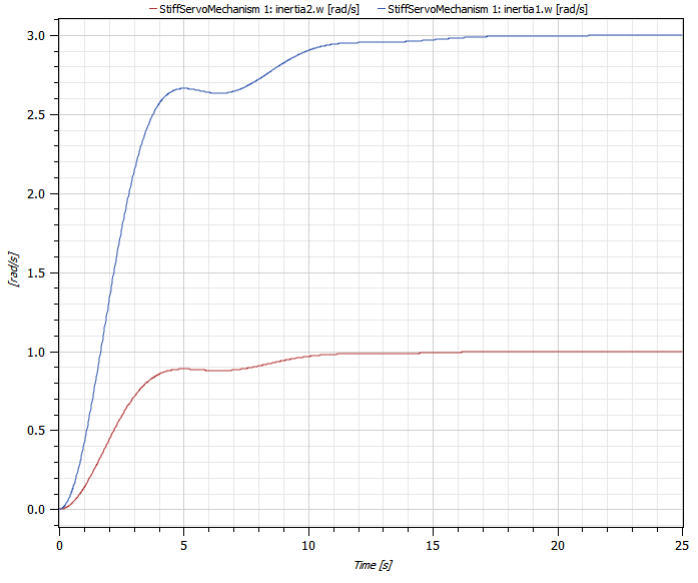

Figure 3-17: Plotting the angular velocity of `inertia1` and `inertia2` for the `IntroductoryExamples.MultiDomain.StiffServoMechanism` model.

Until now we have used default parameters for the controller. By varying the controller gain *k* we can control the response. In this case we vary the gain from 1 to 2 by intervals of 0.25. We can compare the results of all the simulations by creating a new experiment for each simulation and then plotting the results in the same window. New experiments are created by choosing **New** from the **File** menu in *Simulation Center*. Set the appropriate parameter values for each experiment, simulate, and plot the results.

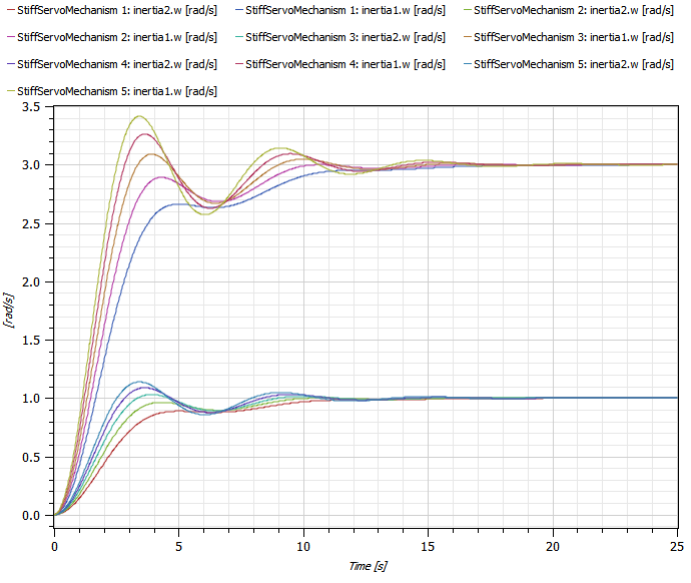

Figure 3-18: Plotting the angular velocity of `inertia1` and `inertia2` for the `IntroductoryExamples.MultiDomain.StiffServoMechanism` model with different controller gain.

By studying the angular velocity response for the motor and gear axes using different regulator gains, we conclude that by choosing $k = 1.5$ we get a sufficiently fast response with few oscillations.

Finally, we develop a control system for the DC motor and the weak axis system.

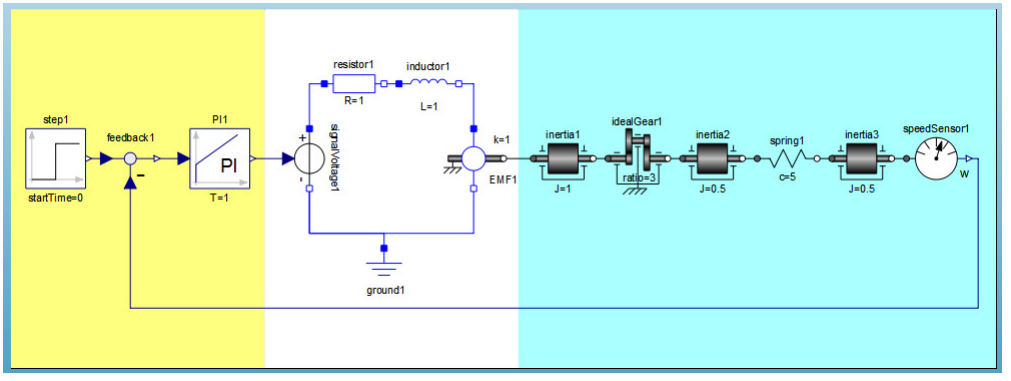

Figure 3-19: The **Diagram View** of the `IntroductoryExamples.MultiDomain.WeakServoMechanism` model.

Before simulating, we set the controller gain to $k = 1.5$ and compare the results with the results of the stiff axis system.

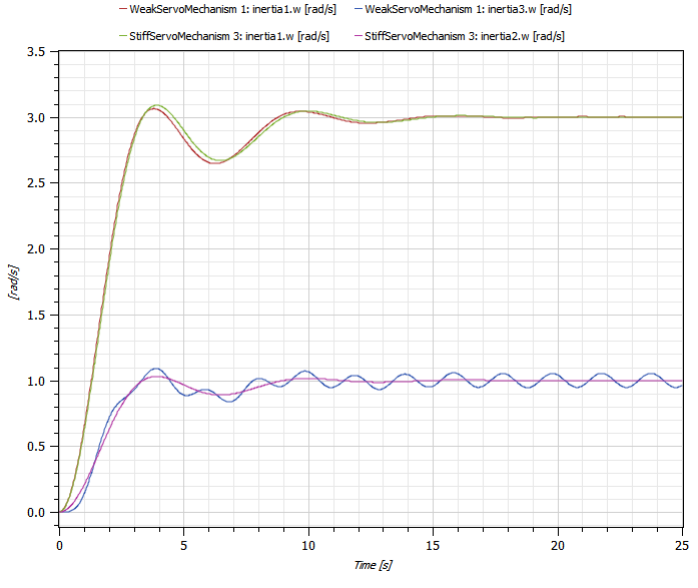

Figure 3-20: Comparison between the inertias of the `StiffServoMechanism` model and the `WeakServoMechanism` model with a regulator gain of $k = 1.5$.

As seen above, the controller design made using the stiff axis model also performs well for the more accurate weak axis.

3.4 Sensitivity Analysis

In this section we will study how sensitive our control design is to changes of different system parameters. This is done using the CVODES solver that supports forward sensitivity analysis. The sensitivity $s_i(t)$ for a state $y_i(t)$ with respect to the parameter p is given by:

$$s_i(t) = \frac{\partial y_i(t)}{\partial p}$$

In other words, at each time instance the sensitivity represents how much the solution for the state $y_i(t)$ would change for a small change of parameter p.

Let us study the sensitivity of our control design with respect to the three inertias. To do so, we select the CVODES solver in the experiment settings and check the **SA** checkboxes in the **Parameters** view for `inertia1.J`, `inertia2.J` and `inertia3.J`.

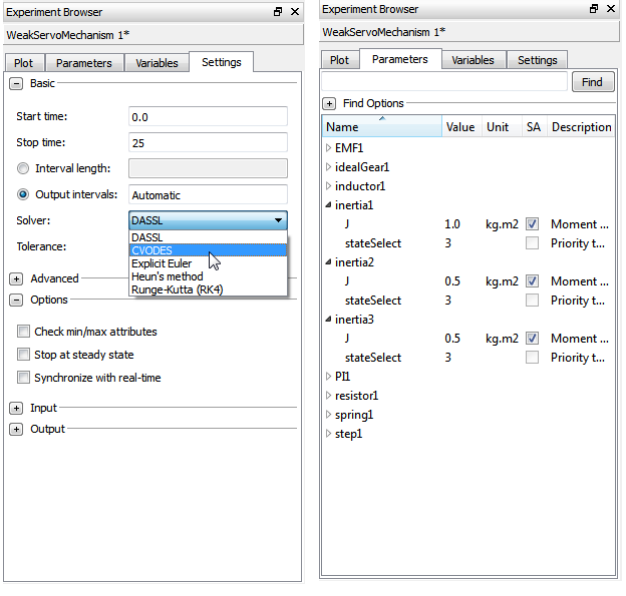

Figure 3-21: Selecting the CVODES solver and selecting `inertia1.J`, `inertia2.J` and `inertia3.J` for sensitivity analysis.

When the simulation has finished, we can find the result of the sensitivity analysis in the **Plot** view as an expandable tree below each state. Figure 3-22 shows the sensitivities for `inertia3.w` with respect to `inertia1.J`, `inertia2.J`, and `inertia3.J`. There we can see that `inertia1.J` has a minor impact on the solution of `inertia3.w` in the beginning of the simulation. This impact diminishes toward the end of the simulation. Furthermore, `inertia2.J` has a negligible impact on the solution during the whole simulation. The inertia `inertia3.J`, on the other hand, has a significantly larger impact on the solution. From this we can conclude that our control design is most sensitive to changes of `inertia3.J`.

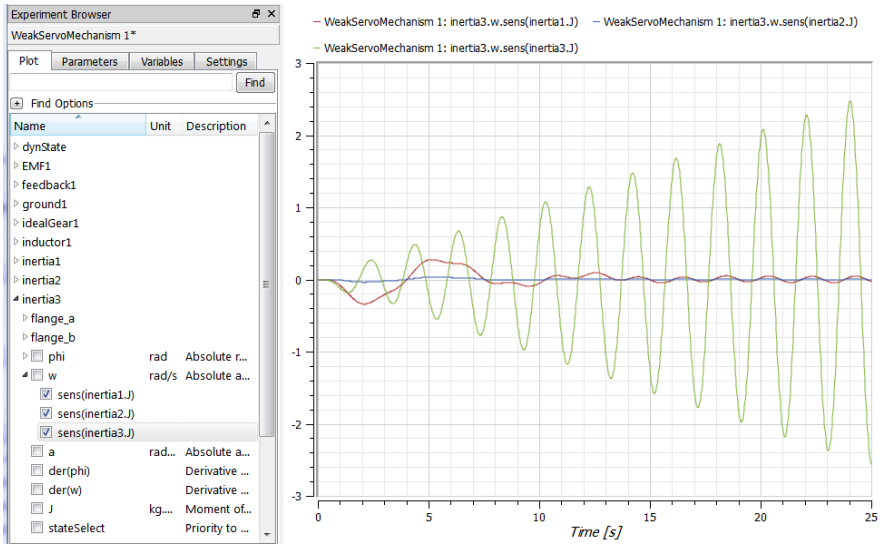

Figure 3-22: The sensitivity of `inertia3.w` with respect to `inertia1.J`, `inertia2.J`, and `inertia3.J`.

To verify our results, we perform the following simulations:

- WeakServoMechanism 1: the original settings.
- WeakServoMechanism 2: `inertia1.J` increased by 50%.
- WeakServoMechanism 3: `inertia2.J` increased by 50%.
- WeakServoMechanism 4: `inertia3.J` increased by 50%.

The result is shown in Figure 3-23, where we confirm our analysis:

- WeakServoMechanism 2: changing `inertia1.J` has some impact in the beginning of the simulation, but it diminishes towards the end.

- WeakServoMechanism 3: changing `inertia2.J` has almost no impact at all.
- WeakServoMechanism 4: changing `inertia3.J` has the most impact; it leads to a phase shift of the oscillations as well as increased amplitude.

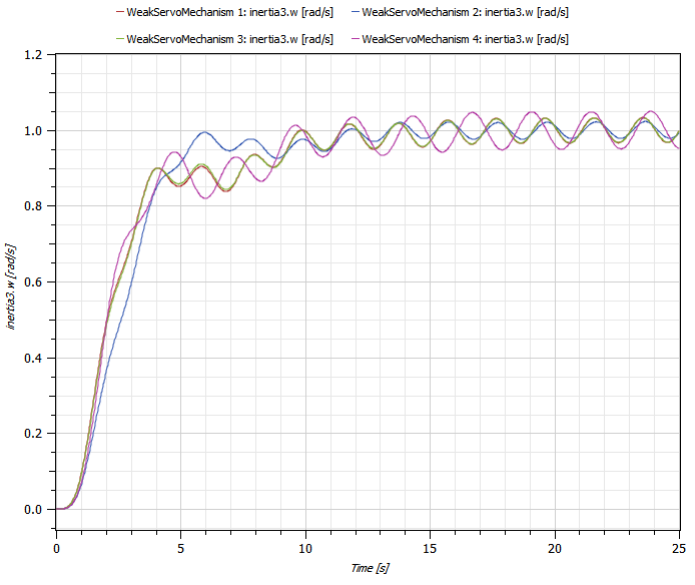

Figure 3-23: The result of changing `inertia1.J`, `inertia2.J`, or `inertia3.J`.

Chapter 4: **Component-Based—Simple Circuit**

Block-based modeling is well suited for problems that have a well-defined causality, i.e. direction of flow. An example of this type of signal-based system is a control system. However, in most cases the causality is not predefined; for instance, a motor could also be used as a generator, depending on whether or not the input signal is the current or torque. Another basic example is the AC circuit below.

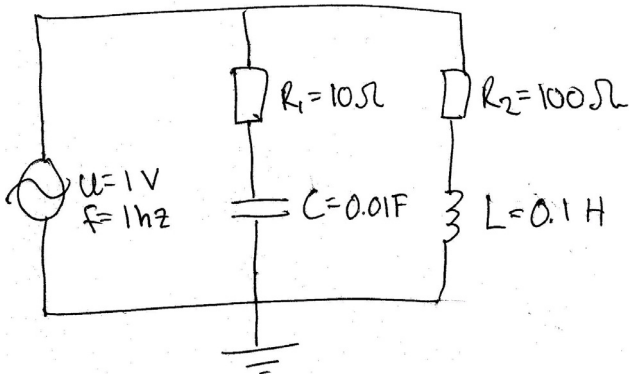

Figure 4-1: The draft schematics of an AC circuit model.

In this example, the circuit above will be used to illustrate the difference between a block-based approach and a component-based approach to modeling.

4.1 Block-Based Circuit

We begin by creating a block-based model. Before we actually start implementing the model, we have to:

- Decide on input and output signals for the system.

- Set up the system of equations.
- Derive the output as a function of the input.

In this example, we want to study the current through the signal voltage as a function of the voltage. To calculate this we have three equations:

$$u(t)=R_1 i_1(t)+\frac{1}{C}\int i_1(t)dt$$

$$u(t)=R_2 i_2(t)+L\frac{di_2(t)}{dt}$$

$$i(t)=-i_1(t)-i_2(t)$$

In these equations i is the total current through the signal voltage, and i_1 and i_2 are the currents running through `resistor1` and `resistor2` respectively. Rewrite the above equations to not contain derivatives:

$$i_1(t)=\frac{1}{R_1}\left(u(t)-\frac{1}{C}\int i_1(t)dt\right)$$

$$i_2(t)=\int\frac{u(t)-R_2 i_2(t)}{L}dt$$

With these equations we can now implement the block-based model, as shown below.

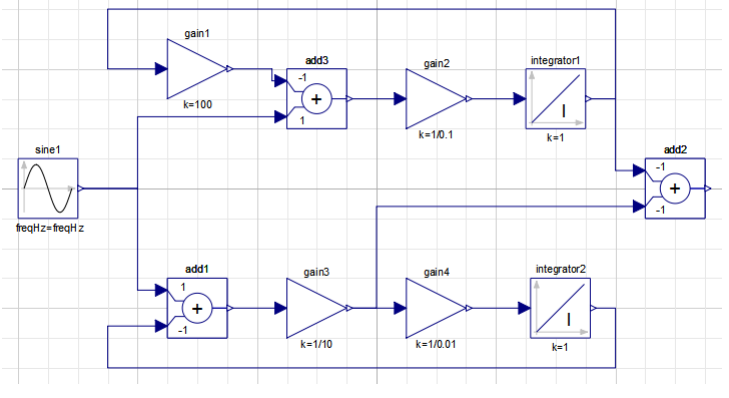

Figure 4-2: The **Diagram View** of the `IntroductoryExamples.ComponentBased.BlockCircuit` model.

Create a new model and locate the components in the **Class Browser**. All components required to implement the system with a block-based approach can be found in the following packages:

- `Modelica.Blocks.Sources`
- `Modelica.Blocks.Math`
- `Modelica.Blocks.Continous`

To view the components in the `Modelica.Blocks.Sources` package in the **Class Browser**, expand the `Modelica` package, followed by `Blocks` and `Sources`, by clicking the symbol to the left of each package icon and name.

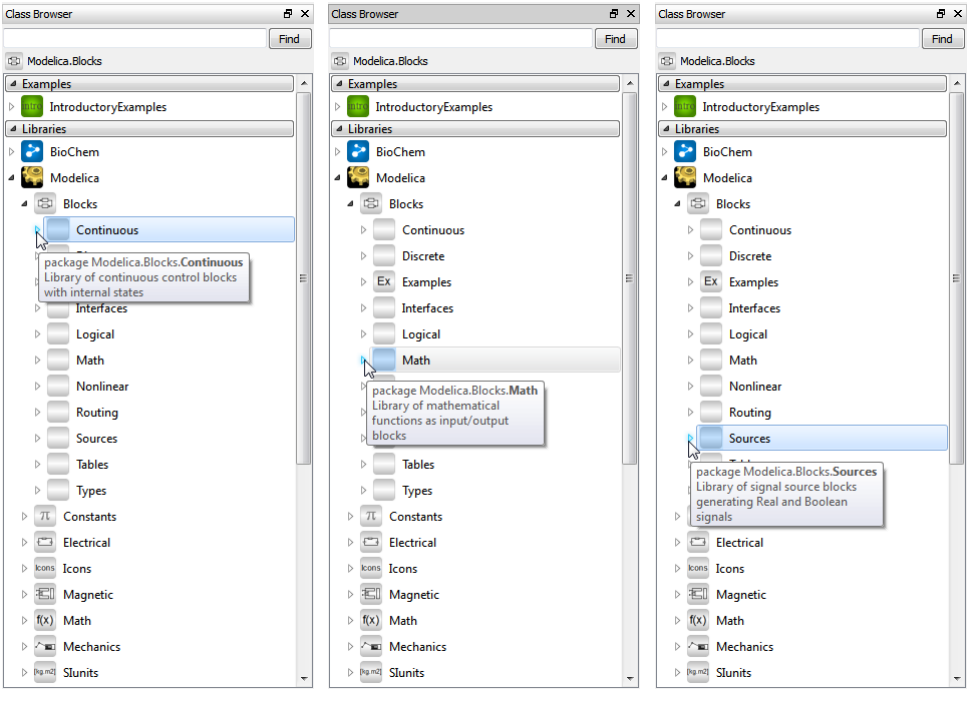

Figure 4-3: Expanding the `Continuous`, `Math`, and `Sources` packages within `Modelica.Blocks`.

Place the components in the **Diagram View** of your model by dragging them from the **Class Browser** and dropping them in the view. Complete the model by connecting the components.

Switch to *Simulation Center* and simulate the model for 10 seconds. The output current is the result of `add2`. The signals i_1 and i_2 are from `gain3` and `integrator1` respectively. The picture below shows the resulting current.

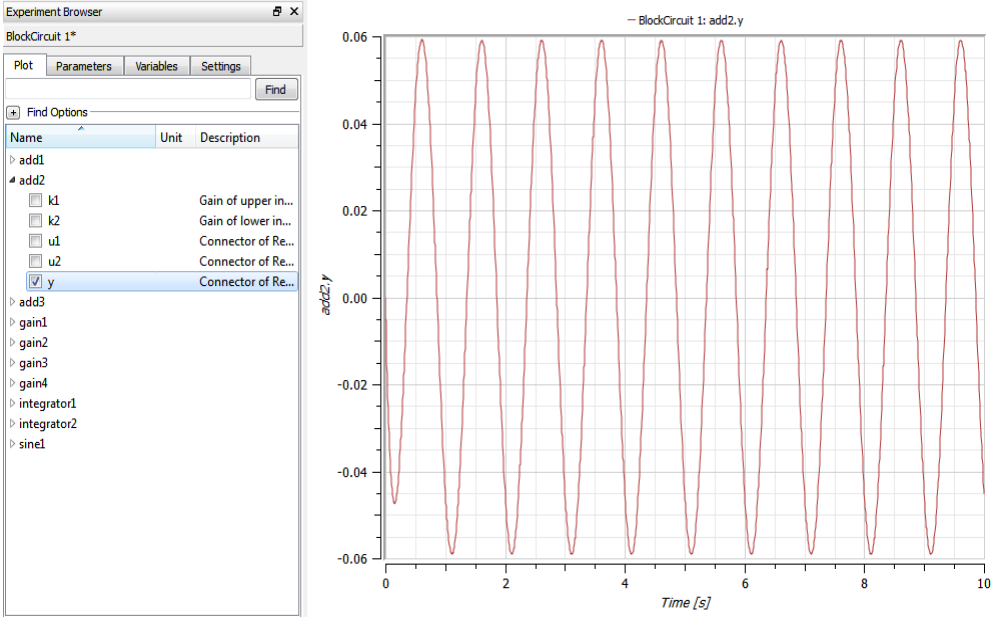

Figure 4-4: Plotting `add2.y` for the `IntroductoryExamples.ComponentBased.BlockCircuit` model with default parameters values.

4.2 Component-Based Circuit

Naturally, implementing a component-based model of the system shown in Figure 4-1 requires only drag-and-drop as well as connecting the components and setting parameters. This leaves us with a model that looks just like the drawing with which we started.

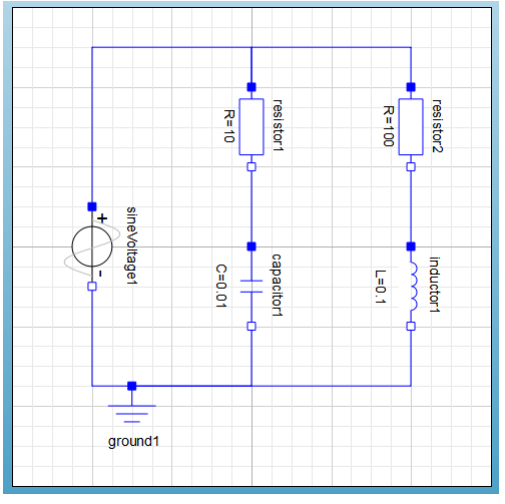

Figure 4-5: The **Diagram View** of the `IntroductoryExamples.ComponentBased.ElectricCircuit` model.

If you would like to build the model yourself, the sine voltage component is located in the `Modelica.Electrical.Analog.Sources` package, and the rest of the components in the `Modelica.Electrical.Analog.Basic` package. Note that some of the parameter values differ from the default, so in order to obtain the same simulation results you will have to change these as well.

Now we can simulate and plot the resulting current through the signal voltage, and as expected it looks just like the result plotted from the block model.

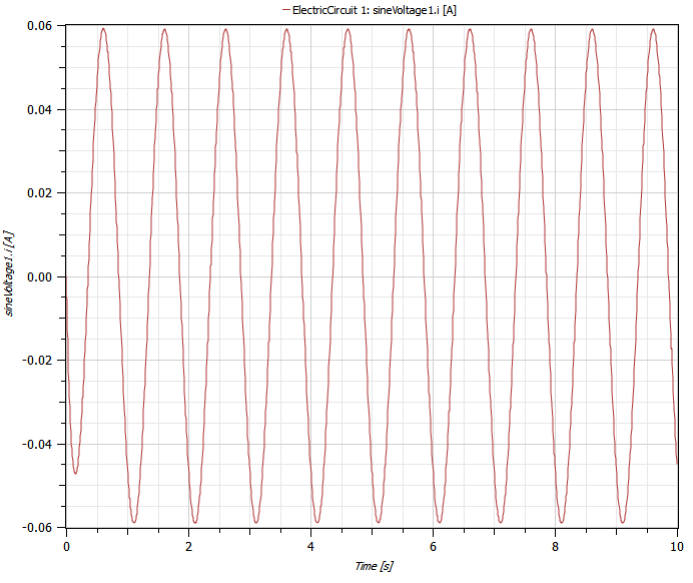

Figure 4-6: Plotting the current going through the source, for the `IntroductoryExamples.ComponentBased.ElectricCircuit` model with default parameters values.

We will end this chapter by adding a second capacitor to the model as shown below. The capacitor component is located in the `Modelica.Electrical.Analog` package.

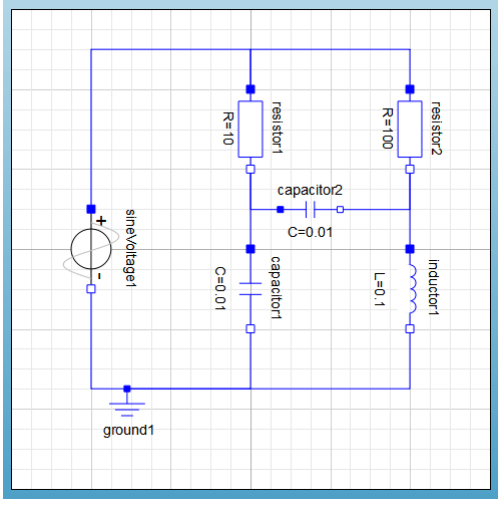

Figure 4-7: The **Diagram View** of the `IntroductoryExamples.ComponentBased.ElectricCircuit2` model.

After simulation, we compare the resulting currents with one another.

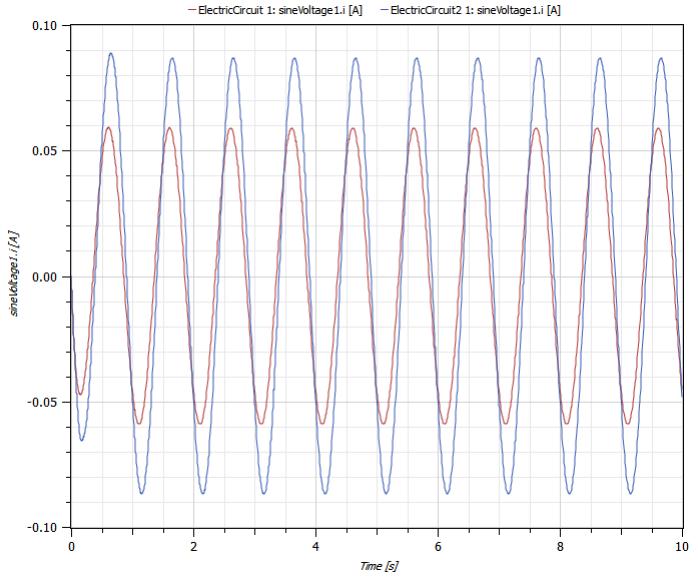

Figure 4-8: Comparison between the two currents going through the source in the `ElectricCircuit` and `ElectricCircuit2` models.

4.2.1 Exercise

Develop a block-based model for the second circuit.

Chapter 5: **Custom Component—Chain Pendulum**

This chapter illustrates how to create and reuse a custom component. A chain pendulum can be seen as a concatenation of chain links, where each chain link consists of a body rotating around one end. We will show how to create a chain link component that will be reused in the chain pendulum model.

5.1 Chain Link Component

The components needed to build the chain link are available in the Modelica Standard Library included in *SystemModeler*. The chain link is inspired by the pendulum example in the `Modelica.Mechanics.MultiBody` library and consists of a body rotating around a revolute joint. To add friction to the rotation, a damper is connected to the revolute joint.

To build the *model* of a chain link, we need to create a new model, find the appropriate components, drag and drop the components into the diagram area, and connect the components using the **Connection Line Tool**. These first steps are explained in detail in Section 3.1. Furthermore, for the *model* to be used as a *component*, it also needs connector interfaces to permit connection to other components. We will show how the connector interfaces are easily added with the **Connection Line Tool**. Parameters are also added to the component to make it more flexible.

We begin by creating a new model that we call `ChainLink`. The components needed are a revolute joint `Revolute` located in `Modelica.Mechanics.MultiBody.Joints`, a body `BodyBox` located in `Modelica.Mechanics.MultiBody.Parts`, and a rotational damper `Damper` located in `Modelica.Mechanics.Rotational.Components`. Accessing the components in the **Class Browser** is explained more in detail in Section 3.1.

To add a component to the `ChainLink` model, drag it from the **Class Browser** and drop it on the **Diagram View** of the class window. The model is depicted in Figure 5-1.

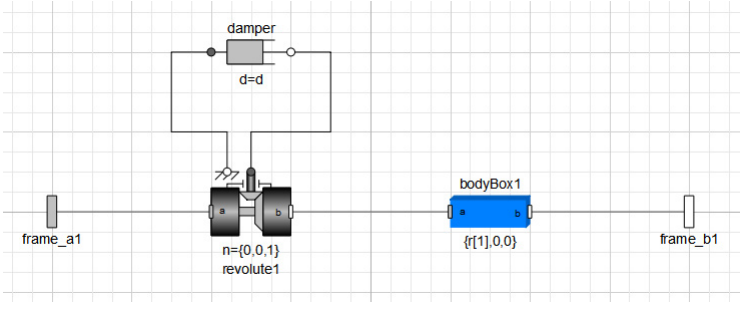

Figure 5-1: Diagram View of `ChainLink` model.

Once you have added the three components, you need to connect them with each other. Components are connected using the **Connection Line Tool**.

Figure 5-2: The **Connection Line Tool** in the toolbar of the *Model Center*.

Only connectors with similar properties can be linked to each other. This rule is supported by the **Connection Line Tool**. If the user tries to connect two incompatible connectors, the connection line will be disabled, as seen in Figure 5-3.

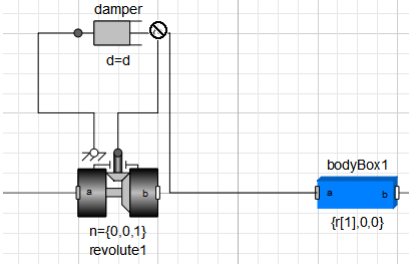

Figure 5-3: Example when the connection line is disabled between two incompatible connectors.

The MultiBody connectors are called "frames" and represent coordinate systems. Since we want the body to rotate around one end, we choose to connect `frame_a` of `bodyBox1` to `frame_b` of the revolute joint. The damper is connected to the revolute joint by using the one-dimensional mechanical systems connectors, called "flanges".

For instance, to connect `frame_a` (gray connector) of `bodyBox1` to `frame_b` (white connector) of the revolute joint component, place the mouse cursor above `frame_a` of `bodyBox1`, press the left mouse button, and hold it down while moving the mouse cursor

to `frame_b` of the revolute joint component. To make the connection, release the mouse button.

The two flanges of the damper, which are connectors for one-dimensional mechanical systems, are connected to the flanges of the revolute joint in the same manner.

Since we want to use this model as a component, we need to add compatible connectors so the model can be linked to other components. With the **Connection Line Tool**, this task is simple. Place the mouse cursor above `frame_b` of `bodyBox1`, press the left mouse button, and hold it down while moving the cursor to the desired position of the new connector. To create the connection, right-click on the mouse button and choose **Create Connector**. A compatible connector is created. A connector to `frame_a` of the revolute joint is created using the same method.

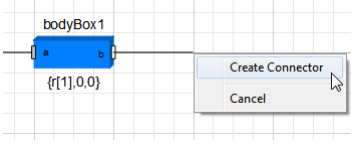

Figure 5-4: Adding a component connector with the **Connection Line Tool**.

We now would like to add parameters to the component to make it more flexible. This will be done in the text view. While dropping and connecting the components, the model editor generates the Modelica code corresponding to the actions. Switch to the **Modelica Text View** to view the textual representation of the model. In the textual representation of the model, each component is declared, and each connection between two components is represented by connect equations in the equation section.

In the textual view, we declare dimension vector `r` of `bodyBox1` (length, width, height) and the damping coefficient `d` as parameters.

```
model ChainLink
     Modelica.Mechanics.MultiBody.Joints.Revolute
revolute1(useAxisFlange=true);
     Modelica.Mechanics.MultiBody.Parts.BodyBox bodyBox1(r=r);
     Modelica.Mechanics.Rotational.Components.Damper
damper1(d=d);
     Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_a1;
     Modelica.Mechanics.MultiBody.Interfaces.Frame_b frame_b1;
     parameter Real r[3]={1,0.1,0.1};
     parameter Real d=1.0;

equation
```

```
      connect(bodyBox1.frame_b,frame_b1);
      connect(revolute1.frame_b,bodyBox1.frame_a);
      connect(revolute1.frame_a,frame_a1);
      connect(damper1.flange_b,revolute1.axis);
      connect(damper1.flange_a,revolute1.support);
end ChainLink;
```

We now create an icon for the component. Switch to the **Icon View** and draw the component icon with the help of the **Graphic Tools** toolbar.

Figure 5-5: The **Graphic Tools** toolbar.

Note that the connectors were automatically added in the icon window when created with the **Connection Line Tool**.

We describe the chain link with an ellipse. To change the ellipse properties, double-click the ellipse object or select it with the mouse and press **Return**.

We use the **Text Tool** to display the name of the component by adding a text item with text "`%name`". To add any parameter display, the user should type "`%`" followed by the parameter name. We display the components parameters `r` and `d` by adding two text boxes with the text "`%r`" and "`%d`".

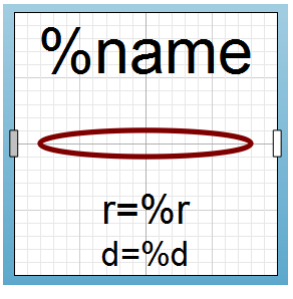

Figure 5-6: Icon View of the chain link component.

5.2 Chain Pendulum Model

Once we have the chain link component, the chain pendulum model is represented with a concatenation of four chain links connected to each other and to the initial frame. The **Diagram View** of the chain pendulum model is represented in Figure 5-7**.**

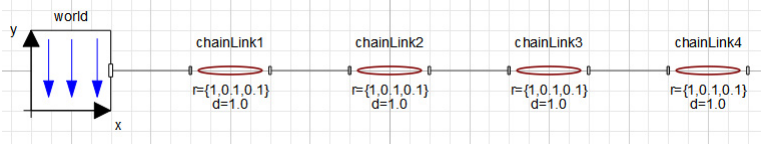

Figure 5-7: Diagram View of the chain pendulum model.

Switch to *Simulation Center* and simulate the model for 10 seconds. The pendulum animation can be visualized after simulation by clicking the **Animation** button in the toolbar.

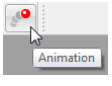

Figure 5-8: Animation is viewed by clicking the **Animation** button in the toolbar.

Figure 5-9 displays the animation of the pendulum at time 4.77 seconds.

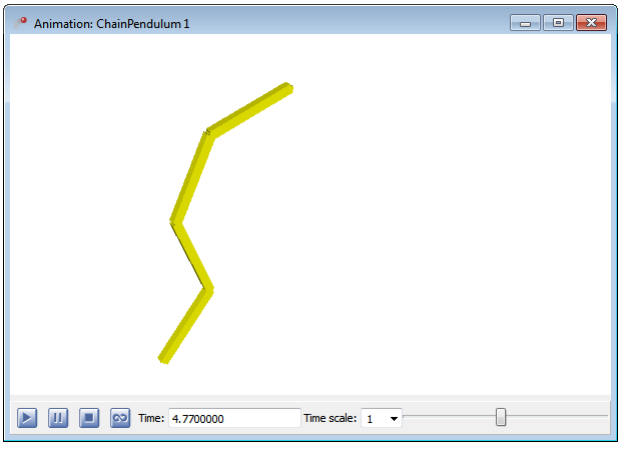

Figure 5-9: View of the chain pendulum animation at time 4.77 seconds.

The position of the end of the pendulum in the *x* and *y* directions is displayed in Figure 5-10.

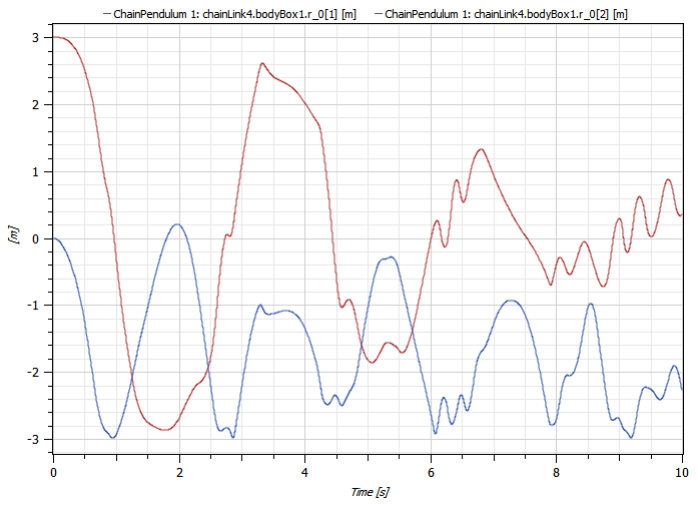

Figure 5-10: Plotting the horizontal and vertical position of the end of the pendulum.

5.2.1 Exercise

The interested reader can create a more general chain pendulum component with the number of chain links as a parameter.

Hint: you can use a `for` loop to connect the chain links.

Chapter 6: **External Function—Chirp Signal**

While it is easy to write Modelica functions, it is sometimes convenient to call a subroutine written in C or FORTAN. This example shows how to use an external function written in C.

6.1 Chirp Function

A chirp signal is a sinusoid with a frequency that changes continuously over:

- a certain band:

$$\Omega : \omega_1 \le \omega \le \omega_2$$

- a certain time period:

$$0 \le t \le M$$

We will use the following signal:

$$u(t) = A\cos\left(\omega_1 t + (\omega_2 - \omega_1)\frac{t^2}{2M}\right)$$

The instantaneous frequency in this signal is obtained by differentiating the argument with respect to time *t*:

$$\omega_i = \omega_1 + \frac{t}{M}(\omega_2 - \omega_1)$$

We see that the instantaneous frequency increases from the lower bound of the frequency band to the higher. When applying the signal to a system, it gives good control over the excited frequency band and is therefore often used for system identification. In this exam-

ple, we will define the chirp function in C and then use it as an external function in Modelica.

6.2 Modeling

We begin by creating a Modelica function called `Chirp` that will make an external call to a C function with the same name (for details on how to create models, see any of the previous examples).

```
function Chirp
   input Modelica.SIunits.AngularVelocity w_start;
   input Modelica.SIunits.AngularVelocity w_end;
   input Real A;
   input Real M;
   input Real t;
   output Real u "output signal";
   external "C" annotation(Include="#include \"Chirp.c\"");
end Chirp;
```

The function has five input signals, one output signal, and a call to the external function `Chirp.c`. The declaration assumes that the function `Chirp.c` is declared with these five inputs and returns a double. If, for some reason, you wish to switch the order of the variables in calling the function, this is possible by changing the declaration to, for instance, the following.

```
external "C" Chirp(t,A,M,w_start,w_end)
annotation(Include="#include \"Chirp.c\"");
```

However, in this case we define a function that uses the same variables in the same order.

```
double Chirp(double w1, double w2, double A, double M, double
time)
{
   double res;
   res=A*cos(w1*time+(w2-w1)*time*time/(2*M));
   return res;
}
```

The function can be written in any text editor and stored with the name `Chirp.c`. In this case, the C function should be stored in the same library as the Modelica function. It can be placed in other places too, but the annotation should then be changed accordingly. For instance, you can place the function directly in the root of C.

```
external "C" annotation(Include="#include \"c:\\Chirp.c\"");
```

As soon as the C function is saved, the Modelica function is ready to use. To do this, we define a Modelica block and call the `Chirp` function within it.

```
block ChirpSignal
   Modelica.Blocks.Interfaces.RealOutput u;
   parameter Modelica.SIunits.AngularVelocity w_start=0;
   parameter Modelica.SIunits.AngularVelocity w_end=10;
   parameter Real A=1;
   parameter Real M=10;
equation
  u=Chirp(w_start, w_end, A, M, time);
end ChirpSignal;
```

Note that we have set default parameters so that the signal will increase from 0 to 10 rad/s in 10 seconds, as shown by this simulation result.

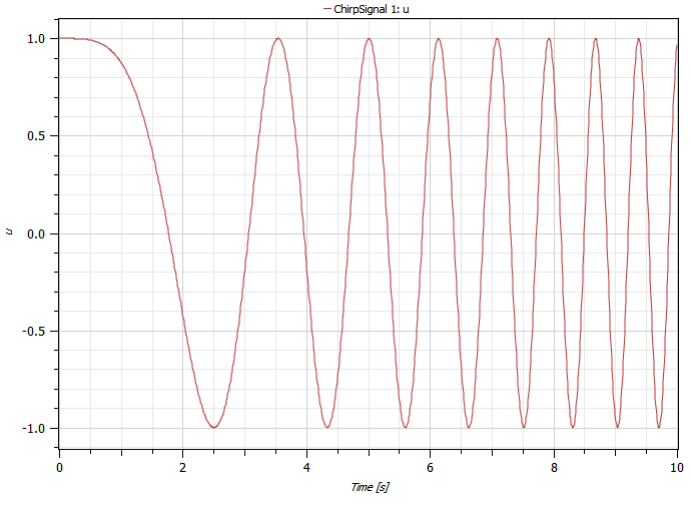

Figure 6-1: Plotting the chirp signal *u* of the `IntroductoryExamples.ExternalFunctions.ChirpSignal` model.

The attentive reader will note that the variable *u* was declared as the predefined Modelica connector `Modelica.Blocks.Interfaces.RealOutput`, which is used in most block models in the Modelica `Blocks` library. As a result, `ChirpSignal` can be used in other models as an input source. For instance, we can use it to test the resonance frequency of the following electric circuit.

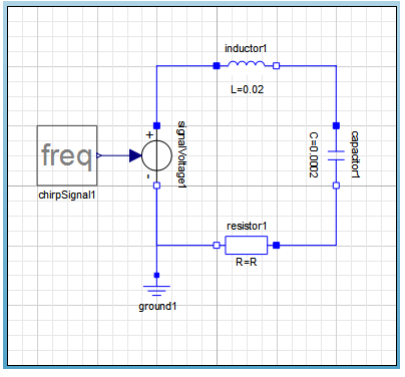

Figure 6-2: The **Diagram View** of the `IntroductoryExamples.ExternalFunctions.SeriesCircuit` model.

Note that the default parameters of the electrical components have been changed according to the figure above. We have also changed the parameters of the chirp to sweep from 0 to 1000 rad/s.

Next we simulate and study the current.

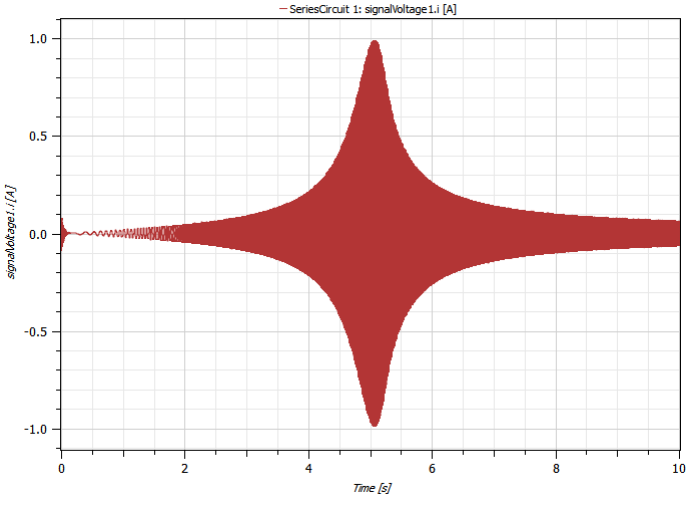

Figure 6-3: Plotting the current *i* of the `IntroductoryExamples.ExternalFunctions.SeriesCircuit` model.

As seen, we get a top in the curve at around 5 seconds, which corresponds to:

$$\omega_i = \omega_1 + \frac{t}{M}(\omega_2 - \omega_1) = 0 + \frac{5}{10}(1000 - 0) = 500\,\text{rad/s}$$

This can also be verified by calculating the resonant frequency for the circuit analytically as follows:

$$\omega_0 = \frac{1}{\sqrt{LC}} = \frac{1}{\sqrt{0.02*0.002}} = 500\,\text{rad/s}$$

Of course, for more complicated systems it might be difficult to calculate the resonant frequency analytically, and in these cases a chirp signal can be very useful.

6.2.1 Exercise

The chirp signal can easily be implemented as one single Modelica block without using an external function. This is left to the interested reader as an exercise.

Chapter 7: **Hierarchical Model—Tank System**

This example illustrates how you can build a hierarchical model using *SystemModeler*, as well as make new libraries. A flat tank model is first developed, followed by a similar component-based tank model. We then see the flexibility that this gives us to test new scenarios.

7.1 Flat Tank

The system we will begin with is a one-tank system with a controller, as illustrated in the picture below.

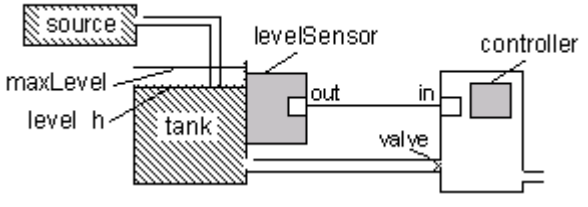

Figure 7-1: A graphical representation of a tank system.

To implement the model, we need to set up the system equations. The water level *h* in the tank is a function of the flow in and out of the tank and the tank area:

$$\dot{h}=\frac{q_{in}-q_{out}}{A}$$

In this example we choose an input flow that is constant the first 150 seconds, after which it triples:

$$q_{in}=\begin{cases} flowLevel, & t<150\text{s} \\ 3(flowLevel), & t\geq 150\text{s} \end{cases}$$

where *flowLevel* is a parameter. By controlling the output flow, we will try to keep the tank level at a desired reference value *ref*. In order to do this we implement a PI controller:

$$q_{out}=K\left(error(t)+\frac{1}{T}\int_0^t error(s)ds \right)$$

where K is the controller gain and T is the time constant of the controller. Finally, we limit the output flow to a minimum value `minV` and a maximum value `maxV`. With this information, we can implement the flat Modelica code.

```
model FlatTank
  parameter Real flowLevel(unit="m3/s")=0.02;
  parameter Real area(unit="m2")=1;
  parameter Real flowGain(unit="m2/s")=0.05;
  parameter Real K=2 "Gain";
  parameter Real T(unit="s")=10 "Time constant";
  parameter Real minV=0,maxV=10;
  parameter Real ref=0.25 "Reference level for control";
  Real h(start=0,unit="m") "Tank level";
  Real qInflow(unit="m3/s") "Flow through input valve";
  Real qOutflow(unit="m3/s") "Flow through output valve";
  Real error "Deviation from reference level";
  Real outCtr "Control signal without limiter";
  Real x "State variable for controller";
equation
  assert(minV >= 0, "minV must be greater or equal to zero");
  der(h)=(qInflow - qOutflow)/area;
  qInflow=if time > 150 then 3*flowLevel else flowLevel;
  qOutflow=Functions.LimitValue(minV, maxV, -flowGain*outCtr);
  error=ref - h;
  der(x)=error/T;
  outCtr=K*(error + x);
end FlatTank;

function LimitValue
  input Real pMin;
```

```
  input Real pMax;
  input Real p;
  output Real pLim;
algorithm
  pLim:=if p > pMax then pMax else if p < pMin then pMin else p;
end LimitValue;
```

By simulating the model for 250 seconds, we can see that the tank level starts to increase, reaching and then surpassing the desired reference level. Once the desired level is surpassed, the outflow is opened, and after 150 seconds the level is stabilized. However, at this moment the input flow is suddenly increased, thus increasing the water level before the controller manages to stabilize it again. This is illustrated in the figure below.

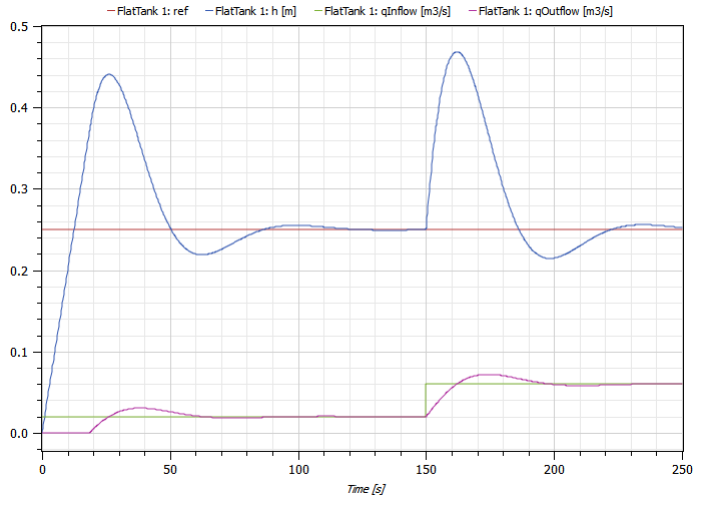

Figure 7-2: Plotting the tank level and the flows in and out of the flat tank with default parameters values.

7.2 Component-Based Tank

Implementing a component-based tank will require a bit more work to begin with, but as soon as we start experimenting with the tank and testing different scenarios we will regain the invested time.

When using the object-oriented component-based approach to modeling, we first try to understand the system structure and decomposition in a hierarchical top-down manner. Once the system components and interactions between these components have been roughly

identified, we can apply the first traditional modeling phases of identifying variables and equations to each of these model components.

By studying Figure 7-1 we see that the tank system has a natural component structure.

We can identify five components in the figure: the tank itself, the liquid source, the level sensor, the valve, and the controller. However, since we will choose very simple representations of the level sensor and the valve, i.e. just a simple scalar variable for each, we let these variables be simple variables of type `Real` in the tank model instead of creating two new classes, each containing a single variable.

The next step is to determine the interactions and communication paths between the components. It is fairly obvious that fluid flows from the source tank via a pipe. Fluid also leaves the tank via an outlet controlled by the valve. The controller needs measurements of the fluid level from the sensor. Thus, a communication path from the sensor of the tank and the controller needs to be established.

In order to connect communication paths, connector instances must be created for those components that are connected, and connector classes must be declared when needed. In fact, the system model should be designed such that the only communication between a component and the rest of the system is via connectors.

Finally, we should think about reuse and generalizations of certain components. For example, do we expect that several variants of a component will be needed? In the case of the tank system, we expect to plug in several variants of the controller, starting with a PI controller. Thus, it is useful for us to create a base class for tank system controllers.

The structure of the tank system model developed using the object-oriented component-based approach is clearly visible in Figure 7-3 below.

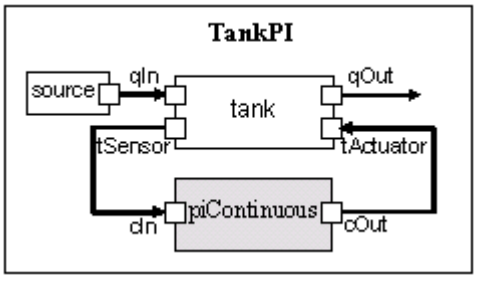

Figure 7-3: A graphical representation of an object-oriented component-based tank system.

We can identify three different types of classes that will be used in the model: interfaces, functions, and components. Therefore, we develop a package containing three subpack-

ages. To create a package, right-click on the **User Classes** root in the **Class Browser** and select **New Class**, as shown in Figure 7-4. You can also right-click on the package you want to add your package to and select **New Class**.

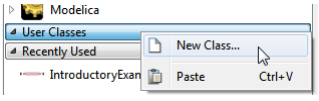

Figure 7-4: Menu to create a new class.

In the dialog box that opens, set the class restriction to **Package** and give the package the name `Hierarchical`. Click the **OK** button to create the new package. The package will appear in the **User Classes** tree of the **Class Browser**. By right-clicking the name of the new package, we can create and add models and packages to it.

7.2.1 Interfaces

We are now ready to create the interfaces, called connectors. Begin by creating a new package `Interfaces` within the `Hierarchical` package. Unless already expanded, expand the `Hierarchical` package in the **Class Browser** to view its contents. Create connector classes within the `Interfaces` package by specifying **Connector** as the class restriction in the **New Class** dialog box.

Create one connector class for reading the fluid level (see the `HelloWorld` example to see how to edit models textually and make model icons).

```
connector ReadSignal "Reading fluid level"
   Real val(unit="m");
end ReadSignal;
```

Create a second connector class for the signal to the actuator for setting valve position.

```
connector ActSignal "Signal to actuator for setting valve
position"
   Real act;
end ActSignal;
```

Finally, create a connector class for the liquid flow at inlets and outlets.

```
connector LiquidFlow "Liquid flow at inlets or outlets"
   Real lflow(unit="m3/s");
end LiquidFlow;
```

7.2.2 Tank Components

The next step is to create the three components of the system. Begin by creating a `Components` package within the `Hierarchical` package. In the `Components` package, create a tank model named `Tank`. The tank model has four interfaces (connectors in Modelica): `qIn` for input flow, `qOut` for output flow, `tSensor` for providing fluid level measurements, and `tActuator` for setting the position of the valve at the outlet of the tank. The central equation regulating the behavior of the tank is the mass balance equation, which in the current simple form assumes constant pressure. The output flows are related to the valve position through the `flowGain` parameter and the `LimitValue` function. This function guarantees that the flow does not exceed what corresponds to the open/closed positions of the valve.

```
model Tank;
   parameter Real area(unit="m2")=0.5;
   parameter Real flowGain(unit="m2/s")=0.05;
   parameter Real minV=0,maxV=10;
   Real h(start=0.0,unit="m") "Tank level";
   Hierarchical.Interfaces.ReadSignal tSensor "Connector, sensor
reading tank level (m)";
   Hierarchical.Interfaces.LiquidFlow qIn "Connector, flow (m3/s)
through input valve";
   Hierarchical.Interfaces.LiquidFlow qOut "Connector, flow (m3/
s) through output valve";
   Hierarchical.Interfaces.ActSignal tActuator "Connector,
actuator controlling input flow";

equation
assert(minV >= 0, "minV - minimum Valve level must be >= 0 ");
  der(h)=(qIn.lflow - qOut.lflow)/area;

equation
  qOut.lflow=Functions.LimitValue(minV, maxV, -
flowGain*tActuator.act);
  tSensor.val=h;
end Tank;
```

The model uses the already-defined connector as well as the `LimitValue` function, which has not been defined yet. This is defined by creating the following function within a new package, named `Functions`. As it is a function, we set its class restriction to **Function** when creating it.

```
function LimitValue;
  input Real pMin;
  input Real pMax;
  input Real p;
  output Real pLim;
algorithm
  pLim:=if p > pMax then pMax else if p < pMin then pMin else p;
end LimitValue;
```

The fluid entering the tank must originate somewhere. Therefore, we have a liquid source component in the tank system `Flow` which increases sharply at t = 150 to three times the previous flow level. This creates an interesting control problem that the tank controller must handle. The following model is created in the `Components` package.

```
model LiquidSource;
  parameter Real flowLevel=0.02;
  Hierarchical.Interfaces.LiquidFlow qOut;

equation
  qOut.lflow=if time > 150 then 3*flowLevel else flowLevel;
end LiquidSource;
```

7.2.3 Controllers

Finally, the controllers need to be specified. We will initially choose a PI controller, but later replace it with other kinds of controllers. The behavior of a PI (proportional and integrating) controller is primarily defined by the following two equations:

$$\frac{dx}{dt}=\frac{error}{T}$$

$$outCtr=K*(error+x)$$

Here x is the controller state variable, *error* is the difference between the reference level and the actual level of liquid obtained from the sensor, T is the time constant of the controller, *outCtr* is the control signal to the actuator for controlling the valve position, and K is the gain factor. These two equations are placed in the controller class `PIcontinuous-Controller`, which extends the `BaseController` class defined later.

```
model PIcontinuousController
  extends BaseController(K=2,T=10);
  Real x "State variable of continuous PI controller";
```

```
equation
  der(x)=error/T;
  outCtr=K*(error + x);
end PIcontinuousController;
```

Both the PI and PID controllers to be defined later inherit the partial controller class `BaseController`, containing common parameters, state variables, and two connectors: one to read the sensor and one to control the valve actuator.

```
partial model BaseController;
  parameter Real Ts(unit="s")=0.1 "Time period between discrete
samples";
  parameter Real K=2 "Gain";
  parameter Real T(unit="s")=10 "Time constant";
  parameter Real ref "Reference level";
  Real error "Deviation from reference level";
  Real outCtr "Output control signal";
  IntroductoryExamples.Hierarchical.Interfaces.ReadSignal cIn
"Input sensor level,  connector";
IntroductoryExamples.Hierarchical.Interfaces.ActSignal cOut
"Control to actuator, connector";

equation
  error=ref - cIn.val;
  cOut.act=outCtr;
end BaseController;
```

7.2.4 Small Tank System

When this is finished, we can compose our tank system model with drag-and-drop.

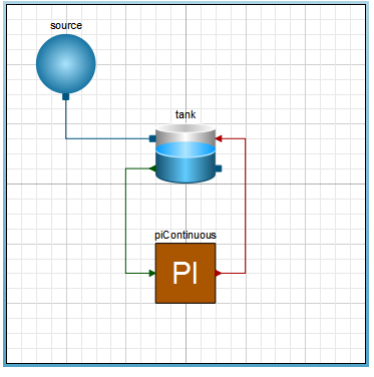

Figure 7-5: The **Diagram View** of the `IntroductoryExamples.Hierarachical.TankPI` model.

Simulating for 250 seconds yields the same result as the flat tank system.

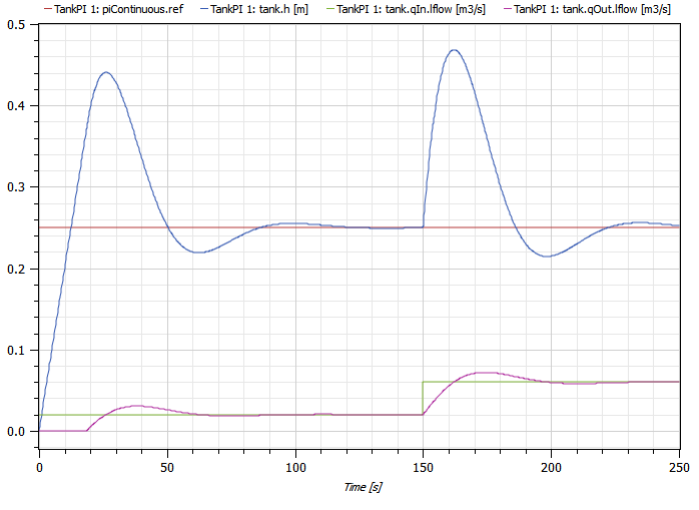

Figure 7-6: Plotting the tank level and the flows in and out of the PI-controlled tank with default parameters values.

7.3 Tank with Continuous PID Controller

We now define a `TankPID` system, which is the same as the `TankPI` system except that the PI controller has been replaced by a PID controller. Here we see a clear advantage of the object-oriented component-based approach over the traditional model-based approach, since system components can easily be replaced and changed in a plug-and-play manner.

A PID (proportional, integrating, derivating) controller model can be derived in a similar way as the PI controller. The basic equations for a PID controller are the following:

$$\frac{dx}{dt} = \frac{error}{T}$$

$$y = T\frac{derror}{dt}$$

$$outCtr = K(error + x + y)$$

Using these equations and the `BaseController` class we create the PID controller.

```
model PIDcontinuousController
  extends BaseController(K=2,T=10);
  Real x "State variable of continuous PID controller";

equation
  der(x)=error/T;
  y=T*der(error);
  outCtr=K*(error + x + y);
end PIDcontinuousController;
```

We can now compose a PID-controlled tank system using drag-and-drop.

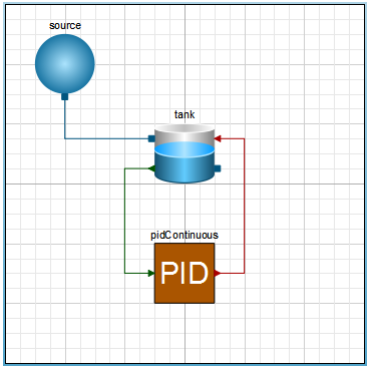

Figure 7-7: The **Diagram View** of the `IntroductoryExamples.Hierarachical.TankPID` model.

We simulate for 250 seconds again and compare with the previous result.

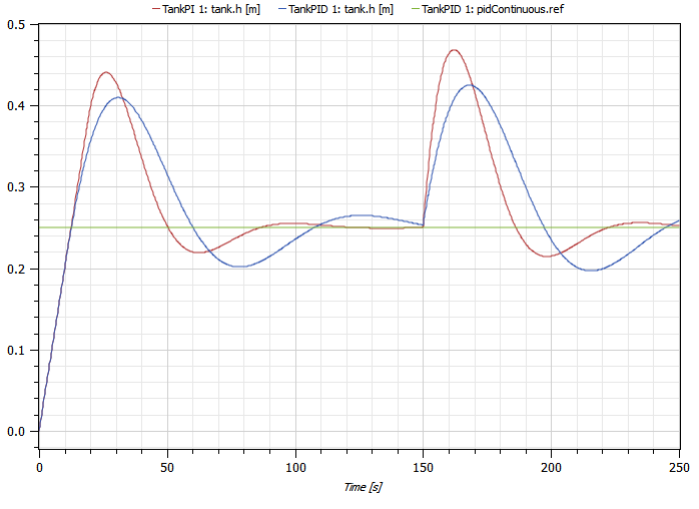

Figure 7-8: Comparison of tank levels between the `TankPI` model and the `TankPID` model.

7.4 Three Tank System

Finally, thanks to the object-oriented component-based approach, we can compose a larger system with ease.

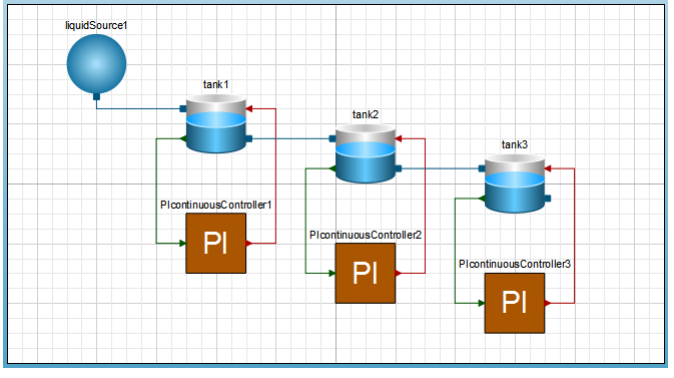

Figure 7-9: The **Diagram View** of the `IntroductoryExamples.Hierarachical.TankSystem` model.

Simulating this system, we can study how the level of each tank is controlled.

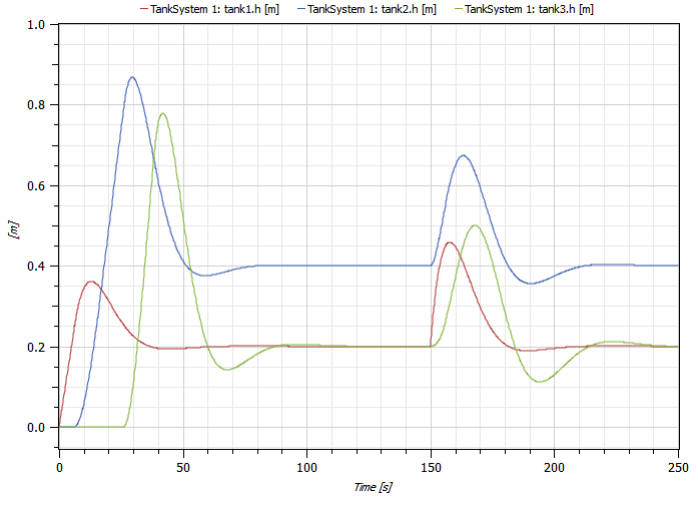

Figure 7-10: Evolution of the tank levels from the `TankSystem` model.

Note that the second tank has a reference level of 0.4 meters while the other tanks have a reference level of 0.2 meters.

Chapter 8: **System—Inverted Pendulum**

This chapter describes a complete system model of an inverted pendulum.

8.1 Inverted Pendulum

A classical engineering problem is to control an inverted pendulum. The pendulum system consists of an electrical motor, a gear, and a pendulum connected to a cart. The position of the cart is controlled using a controller with LQ (Linear Quadratic) design. The first priority of the controller is to make sure that the pendulum stays in an upright position.

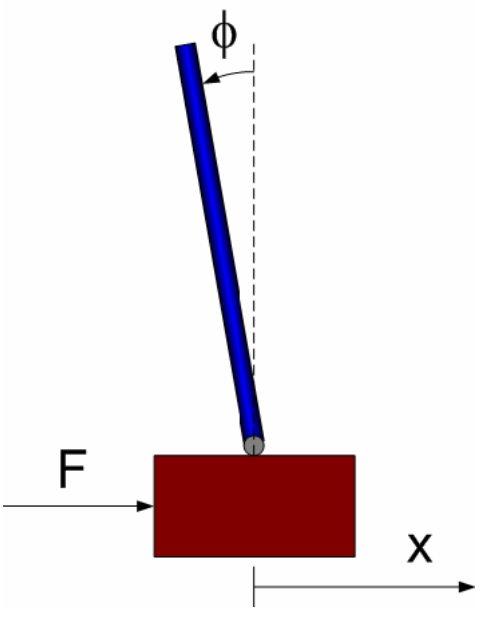

Figure 8-1: Control signals for the pendulum.

To be able to control the pendulum, the position of the cart *x* and the pendulum angle *phi* are measured. The movement of the cart can be controlled via a force *F*, using the flange input.

Simulate the system for 20 seconds and view the result in an animation window. The figure below shows an animation at time 11.3 seconds, using the pulse reference signal (`referenceType=2`).

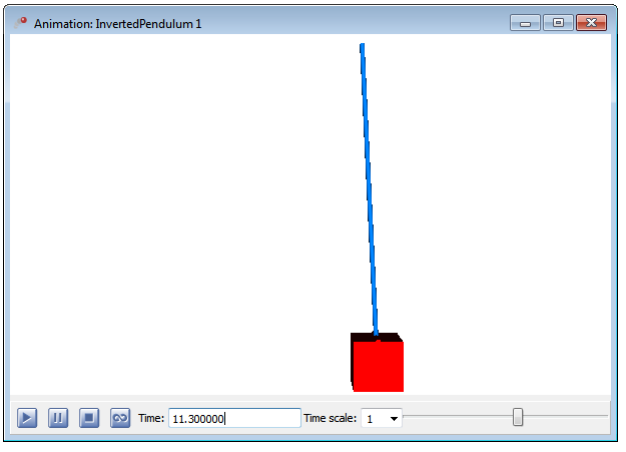

Figure 8-2: Animation of the pendulum system.

The position of the cart can be changed by the `referenceType` parameter. With the time table option (`referenceType=3`), you can create your own arbitrary signal. Change different parameters of the system, e.g. limit the maximum output signal from the controller (`controller.uMax`) or the length of the pendulum (`pendulum.l_pendulum`), and see what happens.

Chapter 9: **Modelica Aspects and Modeling Advice**

This chapter explains some important aspects of the Modelica language and gives you advice on things to keep in mind when working with *SystemModeler*.

9.1 Initial Values

There are several ways to set initial values in a Modelica model. Here, the most common ways are explained.

9.1.1 Start Attribute

Start values for state variables can be set either by using the `start` attribute or by including equations in the initial equation/algorithm section. If a variable is a state variable[1], a `start` value is often needed in order to make sure that the variable has a suitable value at the start of a simulation. The default start value is `0` for numerical (`Integer` and `Real`) variables, `false` for `Boolean` variables, and the empty string `""` for `String` variables.

While a state variable will always have the value of the start attribute at the start of a simulation, a variable that is not a state variable may not. For non-state variables, the start attribute value is treated as a guess value. In the example below, `x` is a state variable (since used in a `der` expression) and the start value is set to `1`, hence the value of `x` will be `1` at the start of the simulation.

```
model A
  Real x(start=1);
```

1. How the compiler chooses state variables depends on several factors, e.g. variables that appear within the `der` operator are preferred. To control the selection of states, the `stateSelect` and `fixed` attributes for variables may be used.

```
equation
  der(x) = -x + 1;
end A;
```

Guess Values

The start attribute can also be used to help the solver in finding the correct value for a non-state variable. The equation `z^2=6-z` has two solutions, `2` and `-3` respectively. To help the solver find the desired solution, a start (guess) value may be given for `z`. This value should be chosen to be as close as possible to the real solution. For example, if a start value of `4` is used for `z`, the solution will be `z=2`. If a start value of `-5` is used, the solution will be `z=-3`. See model `B1` and `B2` below.

```
model B1
  Real z(start=4);
equation
  z^2=6-z;
end B1;

Result: z=2
```

```
model B2
  Real z(start=-5);
equation
  z^2=6-z;
end B2;

Result: z=-3
```

Guess values are useful when working with nonlinear equations, and in particular if the objective is to start in a steady state. In that case, start values for state variables may be used as guess values.

9.1.2 Initial Equations and Algorithms

Another method to set initial values of variables is to use the initial equation or initial algorithm section.

```
model C
  Real x;
initial algorithm
  x:=1;
equation
  der(x) = -x + 1;
end C;
```

In model `C`, `x` will always start with its value equal to `1` while in model `A` (defined in Section 9.1.1), the start value of `x` could easily be changed by a modification. It is not possible to change an initial equation section by a modification. An initial equation section is more powerful than the start attribute, since you could calculate a start value using equations. For

example, in model `D` the variable `x` will start in steady state since `der(x)=0` is used as an initial equation.

```
model D
  Real x;
  Real y(start=3);
initial equation
  der(x)=0;
equation
  der(x) = -x + y + 1;
  der(y)= -y +2;
end D;
```

9.2 Events

An event is generated during simulation, for example when the Boolean expression of an if equation changes its value. When an event occurs, the solver stops and iterates to find the exact point in time of the event. These iterations can take some time and it is advised to try to minimize them if possible.

One method to avoid unnecessary iterations is to use the built-in operator `noEvent` in order to give a hint to the solver that event generation is not needed. This operator can only be used if an expression is continuous during the event (not necessarily differentiable).

```
model EventTest
  Real x1;
  Real x2;
equation
  x1=if time<3 then time else 3 "Event generated at time=3";
  x2=noEvent(if time<3 then time else 3) "No event generated";
end EventTest;
```

The `noEvent` operator can also be used to protect against illegal evaluation. In the following example `sin(x)/x` is calculated. To guard against division by zero, `noEvent` is used around `abs(x)>0` to make sure that `sin(x)/x` is not evaluated if `x=0`.

```
model GuardEval
  Real x;
  Real y;
equation
  x=time - 1;
```

```
  y=if noEvent(abs(x) > 0) then sin(x)/x else 1;
end GuardEval;
```

9.3 The MultiBody Library

`Modelica.MultiBody` is a Modelica library for 3D mechanical modeling. Working with 3D models can be tricky, since all parts need to be connected in space and initial values need to be correct in order for the system to start. *SystemModeler* contains visualization for models created with the `MultiBody` library to aid the user in analyzing the movement in a convenient way. Below are some important things to consider when working with the `MultiBody` library. An introduction to the library as well as information about its components can be found by accessing the embedded documentation of the `Modelica.MultiBody` package and its classes. Figure 9-1 shows an example using the `MultiBody` library.

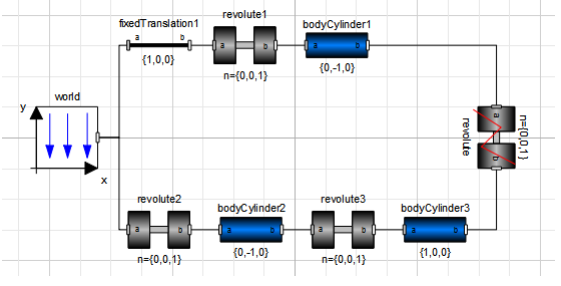

Figure 9-1: Example of a planar loop.

9.3.1 Initial Values

It is important to get the system to initialize the states properly to get a physically correct model. In this example, we have chosen to use the states in the `revolute3` component by setting the parameter `stateSelect` in `revolute3` to `StateSelect.always`. When the model is translated or built, appropriate states are selected. The initial values of these states are shown in the **Variables** tab in *Simulation Center*.

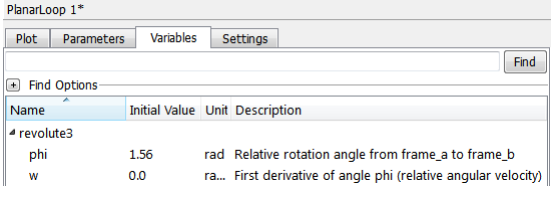

Figure 9-2: Selected states for the model in Figure 9-1.

As seen in Figure 9-2, the angle of the joint `revolute1` will start with the speed 0.0 rad/s and at the angle 0.78539 rad. These values can also be changed directly in the model in the **Variables** tab in *Model Center*.

9.3.2 Angle and Position of Objects

It is very important to get a physically correct model, otherwise the system will likely fail to start a simulation. Try to use simple directions, e.g. `r={1,0,0)`, `r={0,-1,0}`, for components like `BodyCylinder`, `BodyBox`, etc. If another direction is needed, change `phi` in a joint or use the `FixedRotation` and `FixedTranslation` components available in `MultiBody.Parts`.

9.3.3 Animations

Some of the `MultiBody` components contain animation information. These are the `Body`, `BodyBox`, `BodyCylinder`, and `BodyShape` components in `MultiBody.Parts`. There are also visualizers that show properties like velocity, acceleration, etc. in `MultiBody.Visualizers`.

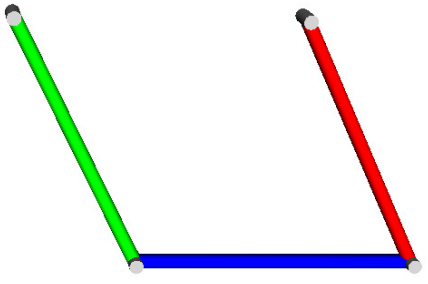

Figure 9-3: Animation of a simple loop model containing three bars from the model depicted in Figure 9-1.

As indicated by their respective names, the `BodyBox` is box shaped and the `BodyCylinder` is cylinder shaped. If more variants of animated shapes are needed, e.g. sphere, cone, gearwheel, etc., the `BodyShape` can be used, setting the parameter `shapeType` to the desired shape. If only a pure animation is needed without physical properties, the `Shape` object can be used.

Figure 9-4 shows an example of a pendulum with a sphere at the end, where the `BodyShape` component has been used to animate the sphere.

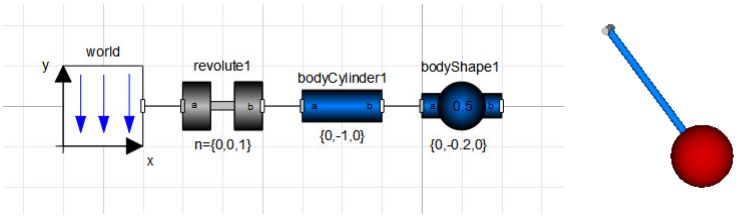

Figure 9-4: A pendulum model with a sphere shape at the end.

9.3.4 CAD Shapes

The documentation for the `MultiBody` library states that external shapes in the form of DXF files can be specified by using `shapeType` "`1`", "`2`", ..., "`n`", with the corresponding DXF file named "1.dxf", "2.dxf", ..., "n.dxf". As an extension to that, *SystemModeler* also supports CAD data in the OBJ file format (see http://en.wikipedia.org/wiki/Obj). Furthermore, it is possible to use an arbitrary filename instead of just numbers. If the experiment is saved, the simulation searches for CAD files in that directory. If that fails, it looks in the directory where the model is saved. Another alternative is to specify the full path in the model.

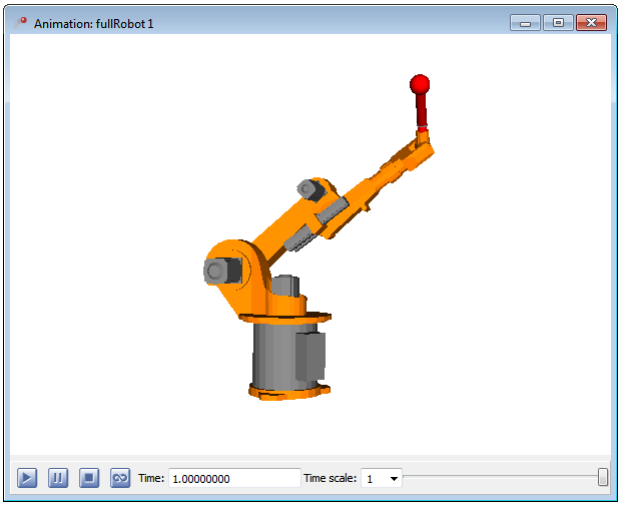

Figure 9-5: Animation view of `Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot`.

9.4 General Advice

Below is a collection of general tips that may prove useful to avoid some common problems.

- Debugging equation-based languages is a different task compared to algorithm-based languages, since it is not possible to add break points, solve step by step, etc. Therefore, try to build up your model step by step, i.e. make small tests of partial systems of your complete model. If an error occurs, it will be a lot easier to locate if the model is small.
- Use the **Validate Class** feature, described in Chapter 3 of the *SystemModeler* User Guide, when creating components. If a component validation is successful, it is more likely that the component will work together with other components. A complete component should always return an equal number of variables and equations from the validation. Note that a partial model naturally often returns a different number of equations than variables.
- If a simulation takes a long time, it could be due to a large number of events. The number of events can be found in the simulation log after a simulation is finished. If possible, try to reduce the number of events by using the `noEvent` (see Section 9.2) operator or try to change your model. Note that when modeling sampled systems, an event is generated at each sample time, which is normal.
- Try to avoid the automatic setting of output intervals when simulating a well-known system. Since data is written to file at each solver step, this could drastically reduce the simulation performance for larger systems. When the model behavior is known, it is recommended to switch to a fixed number of output intervals or use an interval length setting instead.

For further details, visit:

- **www.wolfram.com/system-modeler**
- **reference.wolfram.com/system-modeler**