

MathModelica®

Introductory Examples



Fifth edition, 2009

Published by MathCore Engineering AB.



MathCore Engineering AB	Support:	support@mathcore.com
Teknikringen 1F	URL:	www.mathcore.com
SE-583 30 LINKÖPING	Phone:	+46 13 328500
Sweden	Fax:	+46 13 312701

Copyright © MathCore Engineering AB.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without explicit written permission provided by the copyright holder.

MathCore Engineering AB is the holder of the copyright of the *MathModelica* software system described in this book, including without limitation such aspects of the system as its code, structure, sequence, organization, “look and feel”, and compilation of command names. Use of the system unless pursuant of the terms of a license agreement granted by MathCore Engineering AB or as otherwise authorized by law is an infringement of the copyright.

The author, MathCore Engineering AB, make no representations, express or limited, with respect to this documentation or the software it describes, including without limitations, any implied warranties of merchantability or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which MathCore Engineering AB is willing to license *MathModelica* is a provision that the author, MathCore Engineering AB, and their distribution licenses, distributors, and dealers shall in no event be liable for any indirect, incidental or consequential damages, and that liability for direct damages shall be limited to the amount of the purchase price paid for *MathModelica*.

In addition to the foregoing, users should recognize that all complex systems and their documentation contain errors and omissions. The author, MathCore Engineering AB, shall not be responsible under any circumstances for providing information on or corrections to errors and omissions discovered at any time in this book or software it describes, whether or not they are aware of the errors or omissions. The author, MathCore Engineering AB, do not recommend the use of the software described in this book

for applications in which errors or omissions could threaten life, injury, or significant loss.

MathCore and *MathModelica* are registered trademarks of MathCore Engineering AB. System Designer, and Simulation Center are trademarks of MathCore Engineering AB. *Modelica* is a registered trademark of the Modelica Association. All other product names are trademarks of their producers.

Printed in Sweden

The Modelica License

This book contains material from the Modelica Standard Library, reproduced with permission from the Modelica Association according to the License agreement below.

Version 1.1 of June 30, 2000

Redistribution and use in source and binary forms, with or without modification are permitted, provided that the following conditions are met:

1. The author and copyright notices in the source files, these license conditions and the disclaimer below are (a) retained and (b) reproduced in the documentation provided with the distribution.
2. Modifications of the original source files are allowed, provided that a prominent notice is inserted in each changed file and the accompanying documentation, stating how and when the file was modified, and provided that the conditions under (1) are met.
3. It is not allowed to charge a fee for the original version or a modified version of the software, besides a reasonable fee for distribution and support. Distribution in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution is permitted, provided that it is not advertised as a product of your own.

DISCLAIMER

The software (sources, binaries, etc.) in their original or in a modified form are provided “as is” and the copyright holders assume no responsibility for its contents what so ever. Any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are **disclaimed**. **In no event** shall the copyright holders, or any party who modify and/or redistribute the package, **be liable** for any direct, indirect, incidental, special, exemplary, or consequential damages, arising in any way out of the use of this software, even if advised of the possibility of such damage.

Contents

1	Introduction	1
2	Hello World	3
2.1	Hello World model	3
2.1.1	Exercise	9
3	Multi Domain - a servo mechanism	11
3.1	DC Motor	11
3.2	Stiff and weak axis	18
3.2.1	Exercise	20
3.3	Control System	21
3.4	Sensitivity Analysis	24
4	Component Based - Simple Circuit	29
4.1	Block-Based Circuit	29
4.2	Component-based circuit	32
4.2.1	Exercise	34
5	Custom Component - Chain Pendulum	35
5.1	Chain Link Component	35
5.2	Chain Pendulum Model	39
5.2.1	Exercise	41

6	External Functions - Chirp Signal	43
6.1	Chirp function	43
6.2	Modeling	44
6.2.1	Exercise	47
7	Tank System	49
7.1	Flat tank	49
7.2	Component-based tank	51
7.2.1	Interfaces	53
7.2.2	Tank components	54
7.2.3	Controllers	55
7.2.4	Small tank system	56
7.3	Tank with continuous PID controller	57
7.4	Three tanks system	58
8	Systems	61
8.1	Inverted Pendulum	61
A	MathModelica Professional	63

Chapter 1: Introduction

This document contains examples that are useful for getting started with *MathModelica*. The examples have a detailed step-by-step description of how to build and simulate the models. If you have the Professional edition of *MathModelica*, you can learn more from the *Mathematica* notebook examples presented in Appendix A.

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 library browser.

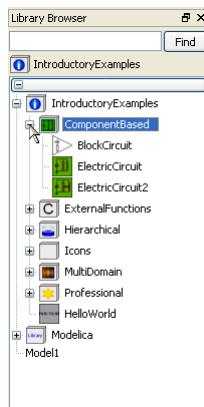


Figure 1-1: Browsing the package structure of the IntroductoryExamples Library using the library browser.

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

Additional information about the models in the Introductory Examples library is integrated into the packages and models and may be viewed by right clicking any package or model in

the Library Browser and choosing View Documentation from the pop-up menu.

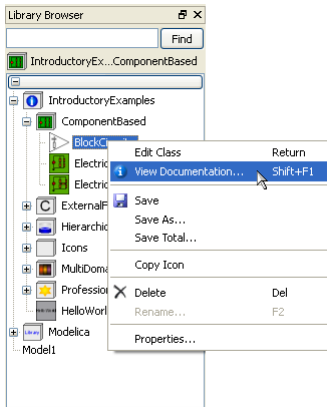


Figure 1-2: Viewing the documentation of a model.

The documentation of the classes will be shown in the Modelica help browser.

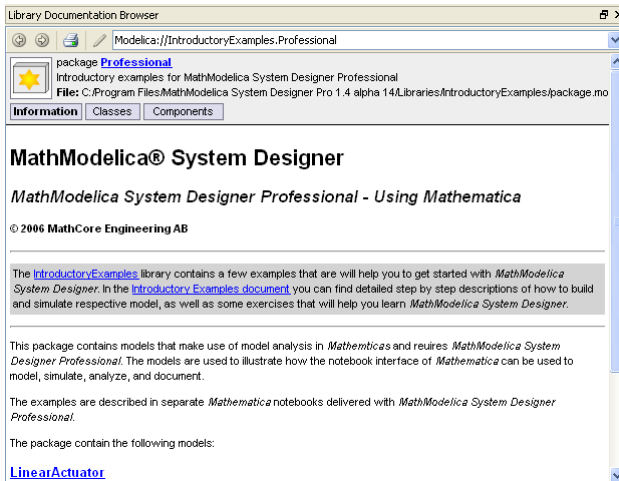


Figure 1-3: The documentation of the IntroductoryExamples.Professional package.

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 *MathModelica*, 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 $\text{der}(x)$ 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.

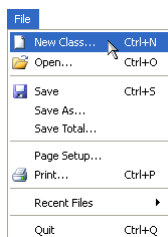


Figure 2-1: Choosing New Class from the File menu.

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".

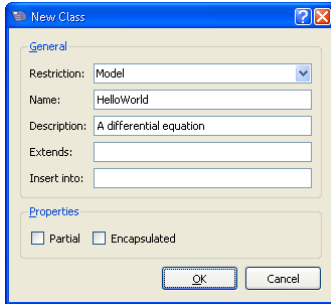


Figure 2-2: Specifying a name and description for a new model.

When clicking the OK button, the model will be created and become visible in the library 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 editor.

The textual representation of the model should look as follows:

```
model HelloWorld "A differential equation"
  annotation (...);
end HelloWorld;
```

The annotation in the second row contains 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"
  annotation (...);
  Real x(start=1);
equation
  der(x)=-x;
end HelloWorld;
```

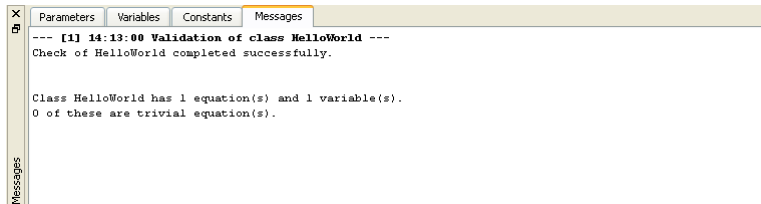
Note that when we declare the variable we also set its initial value to 1 by specifying a value for its parameter start.

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 editor.

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



If everything was typed in correctly you should find a report similar to the one above.

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



Figure 2-5: The Simulation Center button in the toolbar of the model editor.

Simulation Center will start and the HelloWorld model will automatically be translated into an executable. An experiment is created for the Hello World model in the experiment browser of Simulation Center.

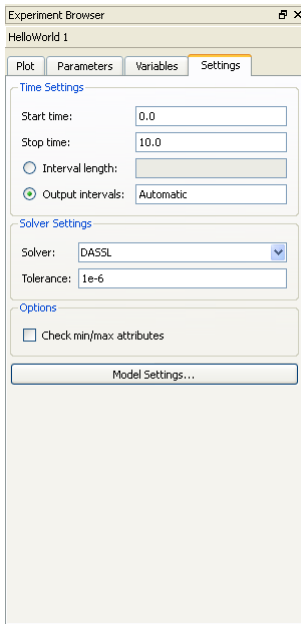


Figure 2-6: The settings view of the HelloWorld experiment in Simulation Center.

In the experiment browser you are able to 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 2-7: 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 check box in front of the variable x to plot the result.

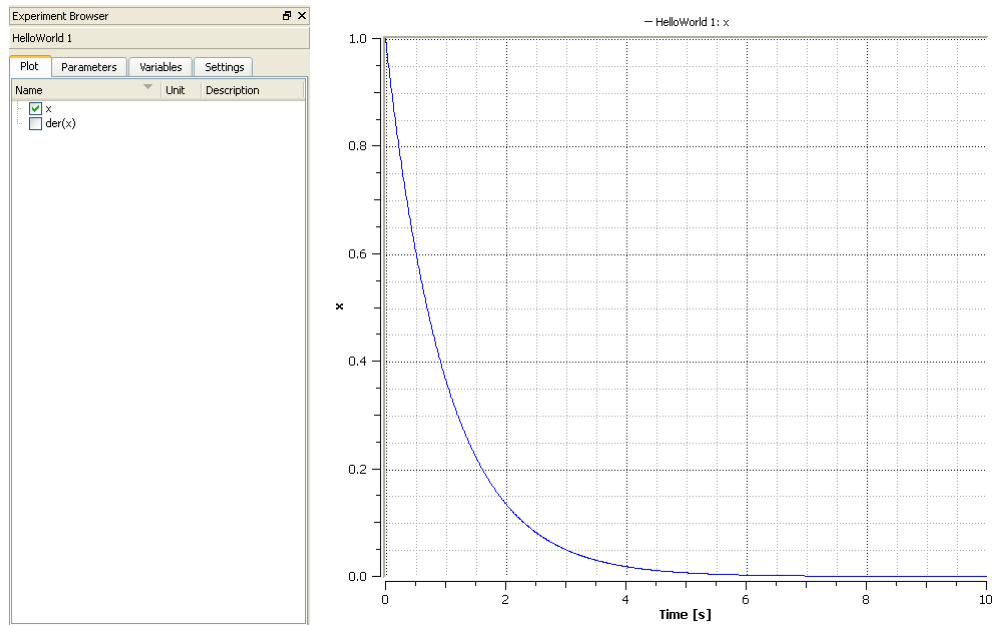


Figure 2-8: Plotting the variable x of the HelloWorld model in Simulation Center.

We will now return to the model editor 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 2-9: The icon view button in the toolbar of the model editor.

To create an icon we will use the drawing tools available in the toolbar of the model editor.

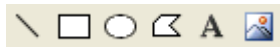


Figure 2-10: The drawing tools in the toolbar of the model editor.

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.

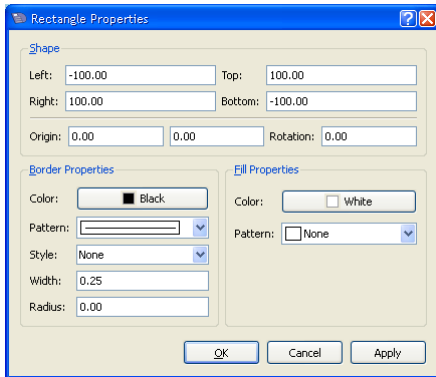


Figure 2-11: Editing the properties of a rectangle.

Change the fill color to grey and 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 item tool and draw a text item covering the entire rectangle, and change the text to "Hello World" in the 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.

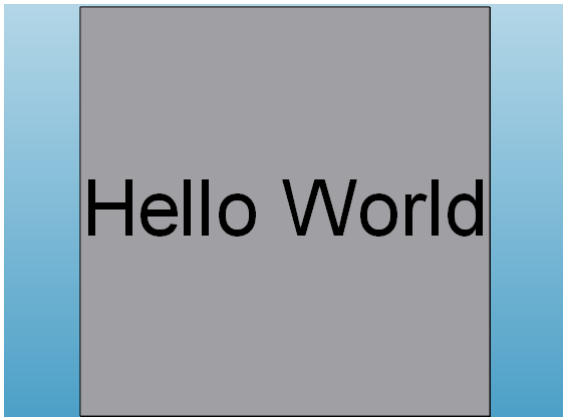


Figure 2-12: The icon of the HelloWorld model.

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 test the result.

Chapter 3: Multi Domain - a servo mechanism

This example shows how to develop a servo mechanism model step-by-step in *MathModelica*. It illustrates the multi-engineering capabilities and shows how you can use Simulation Center to analyze models created in the model editor, 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 electro-motric 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 in *MathModelica*. With the help of drag-and-drop they can be used to compose the model as illustrated in the figure below.

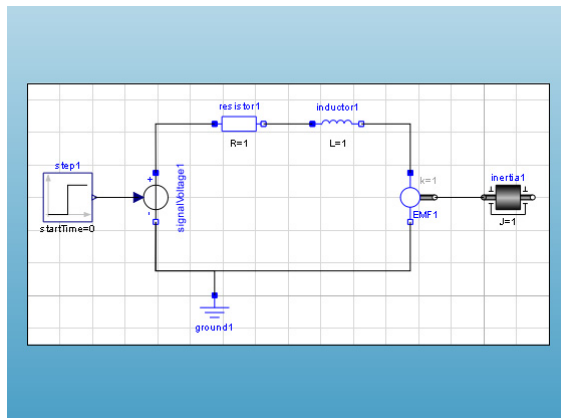


Figure 3-1: The diagram view of a DC Motor in the model editor

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 library browser. We will show how to do both.

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

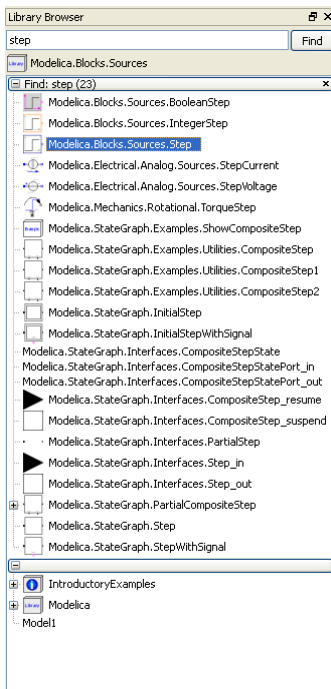


Figure 3-2: Searching for a step source component using the library browser in the model editor.

If everything went well, you should have at least 23 matches for "step" in the library 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 library 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 library browser and expand the branches of the tree all the way down to the branch which represents the package Sources in which the component is located in.

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

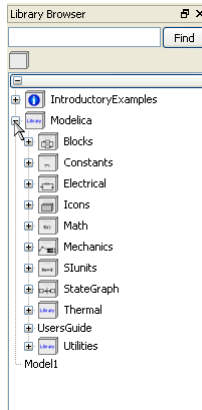


Figure 3-3: Expanding the Modelica package in the library browser.

As you can see the Modelica package has several packages within it. 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.

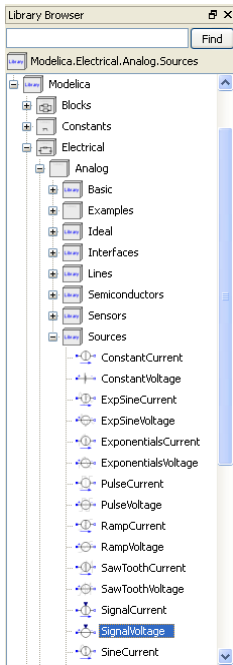


Figure 3-4: Expanding the Modelica.Electrical.Analog.Sources package.

Add the SignalVoltage component, highlighted in the figure above, to the DCMotor, by dragging it to the diagram view of the class window.

We have now added 2 of the 7 components. The 4 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.

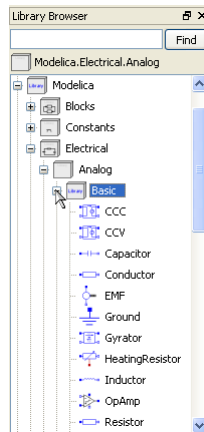


Figure 3-5: Expanding the Modelica.Electrical.Analog.Basic package

When you have added the electrical components to the DCMotor model, there is only 1 component left to add, the inertia. It is located in the Modelica.Mechanics.Rotational package. You can choose if you want to search for it or browse to it directly by expanding the Modelica, Mechanics, and Rotational 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 editor.

To connect, for instance 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, 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.

```

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.Inertia inertial;
  Modelica.Electrical.Analog.Basic.Ground ground1;

equation
  connect(EMF1.flange_b,inertial.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.

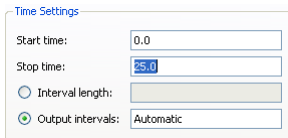


Figure 3-7: Setting the simulation time to 25 seconds for the DCMotor model.

Start the simulation and when completed, select the variables to plot in the experiment browser as illustrated in the figure below.

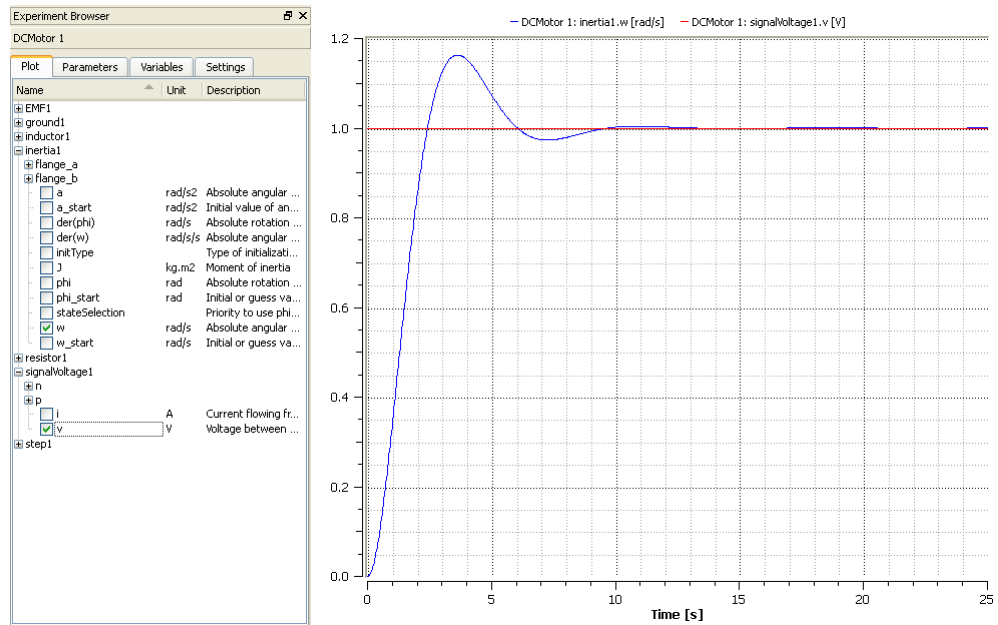


Figure 3-8: Plotting inertia1.w and signalVoltage1.v for the DCMotor model with default parameter values.

Finally, we get the result, with the plot of inertia.w vs. time and signalVoltage1.v vs. 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 parameter view in the experiment browser. To edit a parameter value in the parameter 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 kgm².

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

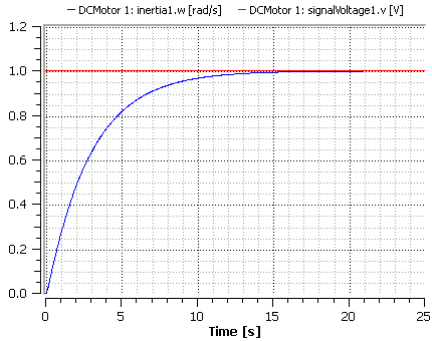


Figure 3-9: Plotting `inertia1.w` and `signalVoltage1.v` for the `DCMotor` model with customized parameter values.

3.2 Stiff and weak axis

In this section we will begin by develop 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 `Modelica.Blocks.Sources` and `Modelica.Mechanics.Rotational` packages in the library 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.

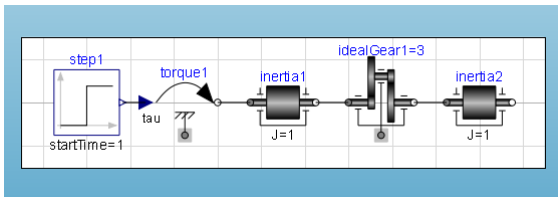


Figure 3-10: The diagram view of the `IntroductoryExamples.MultiDomain.StiffAxis` model.

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 model editor.



Figure 3-11: Editing the transmission ratio of a ideal gear component in the model editor.

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 s.

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.

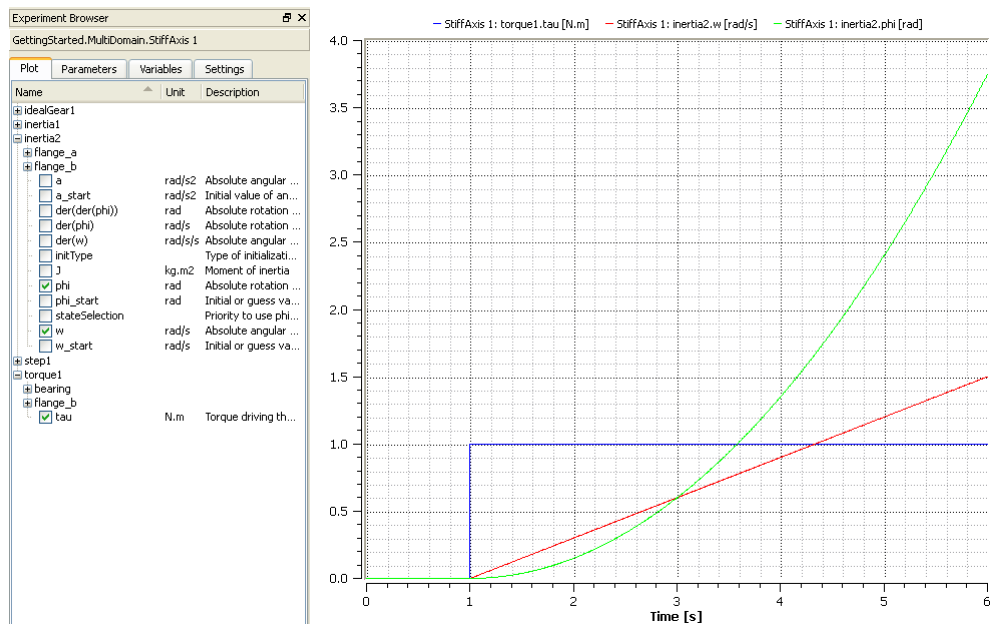


Figure 3-12: Plotting the torque, the angle of inertia2, and the angular velocity of inertia2 for the `IntroductoryExamples.MultiDomain.StiffAxis` model.

By including an additional weakness, the axis can be more accurately modeled. This is pos-

sible 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 package.

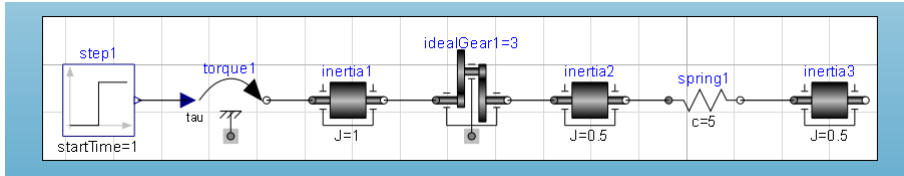


Figure 3-13: The diagram view of the IntroductoryExamples.MultiDomain.WeakAxis model.

Notice that inertia1 and inertia2 has been given a moment of 0.5 kgm^2 , and the spring constant of spring1 is set to 0.5 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.

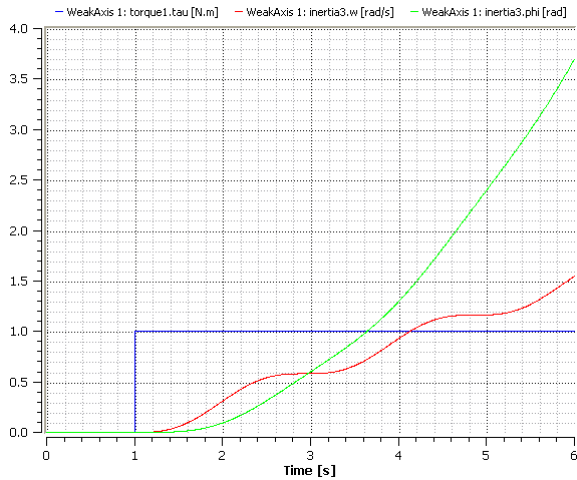


Figure 3-14: Plotting the torque, the angle of inertia3, and the angular velocity of inertia3 for the IntroductoryExamples.MultiDomain.WeakAxis model.

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, and 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.

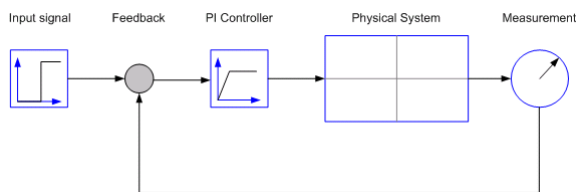


Figure 3-15: Simplified representation of a control system.

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.

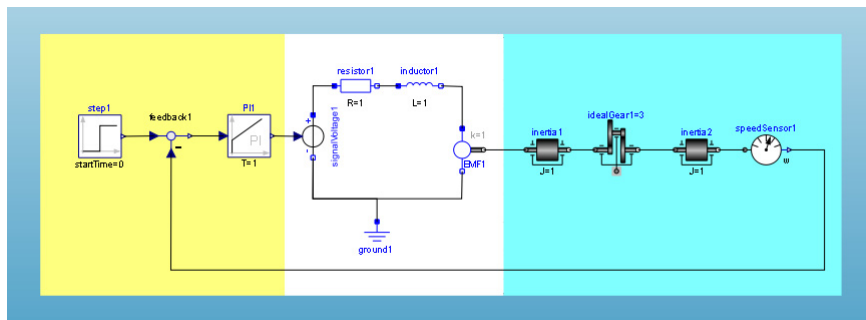


Figure 3-16: The diagram view of the IntroductoryExamples.MultiDomain.StiffServoMechanism

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.
- Finally, 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.

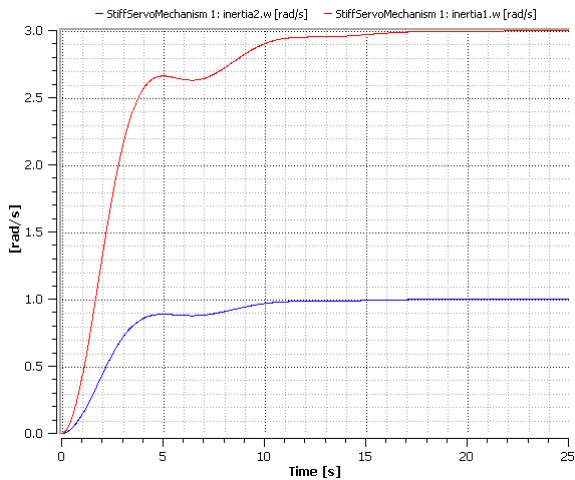


Figure 3-17: Plotting the angular velocity of inertia1 and inertia2 for the IntroductoryExamples.MultiDomain.StiffServoMechanism model.

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 plot 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.

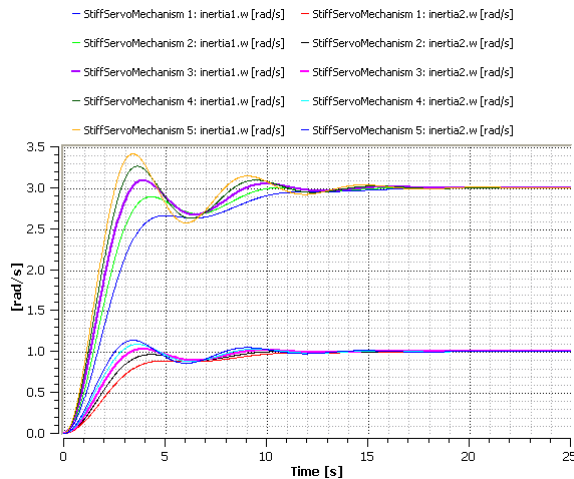


Figure 3-18: Plotting the angular velocity of inertia1 and inertia2 for the IntroductoryExamples.MultiDomain.StiffServoMechanism model with different controller gain.

By studying the angular velocity response for the motor and gear axes using different regulator gains we conclude that by choosing $k = 1.5$ (the plotted curves with a slightly thicker width in the figure above) we get a sufficiently fast response with few oscillations.

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

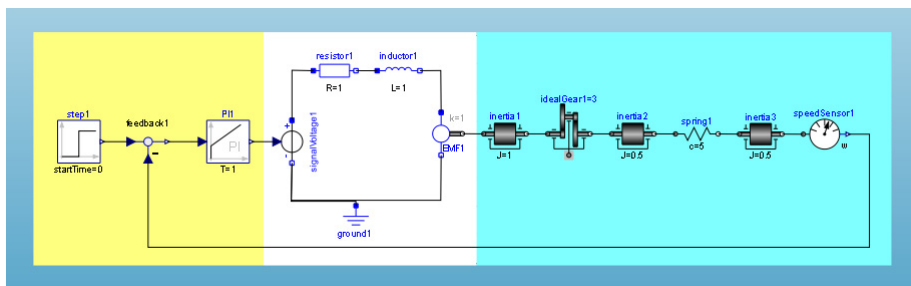


Figure 3-19: The diagram view of the IntroductoryExamples.MultiDomain.WeakServoMechanism model.

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

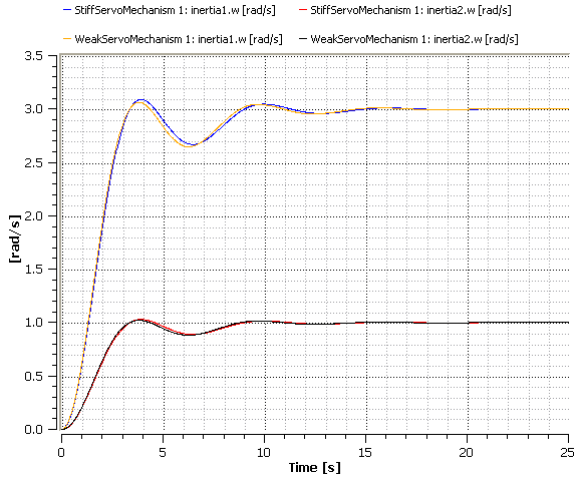


Figure 3-20: Comparison between the inertias of the StiffServoMechanism model and the WeakServoMechanism model with a regulator gain of $k = 1.5$.

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 check boxes in the parameters view for `interta1.J`, `interia2.J` and `internia3.J`.

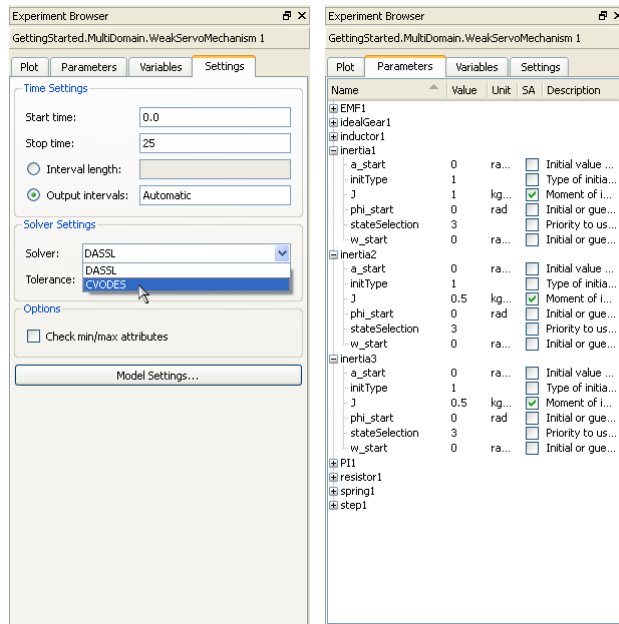


Figure 3-21: Selecting the CVODES solver and selecting inertia1.J, inertia2.J and inertia3.J for sensitive analysis.

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 solution 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. An impact that diminishes towards 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.

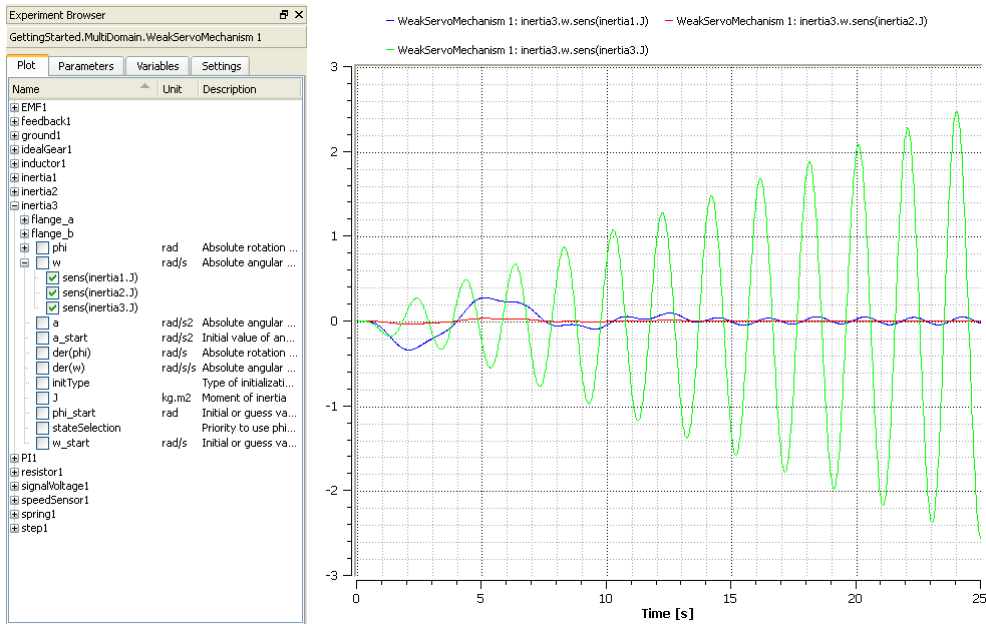


Figure 3-22: The solution sensitivity of inertia3.w with respect to inertia1.J , inertia2.J and inertia3.J .

To verify our results we perform the following simulations:

- WeakServoMechanism 1 - The original settings.
- WeakServoMechanism 2 - inertia1.J increased with 50%.
- WeakServoMechanism 3 - inertia2.J increased with 50%.
- WeakServoMechanism 4 - inertia3.J increased with 50%.

The result is shown in Figure 3-23 and there we can confirm our analysis:

- WeakServoMechanism 2 - Changing inertia1.J has some impact in the beginning of the simulation but it diminished 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.

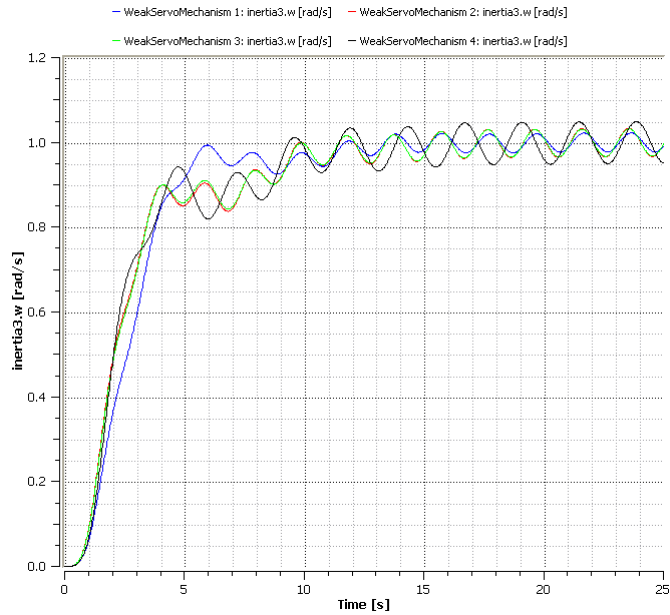


Figure 3-23: The result of changing inertia1.J, inertia2.J and inertia3.J respectively.

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 these types of signal-based systems is a control system. However in most cases the causality is not pre-defined, 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.

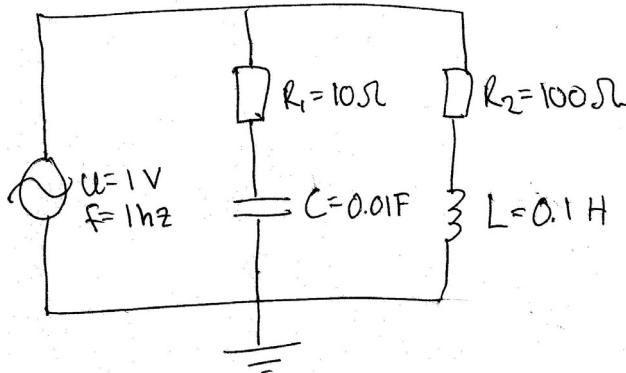


Figure 4-1: The draft schematics of an AC circuit model.

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

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:

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

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

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

Where i is the total current through the signal voltage, i_1 and i_2 are the currents running through resistor1 and resistor2 respectively. Using the Laplace-transform on the above equations resolves i as a function of u as seen in the following equations:

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

$$i_2(t) = \frac{1}{R_2} \left(U(s) - L \frac{di_2(t)}{dt} \right)$$

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

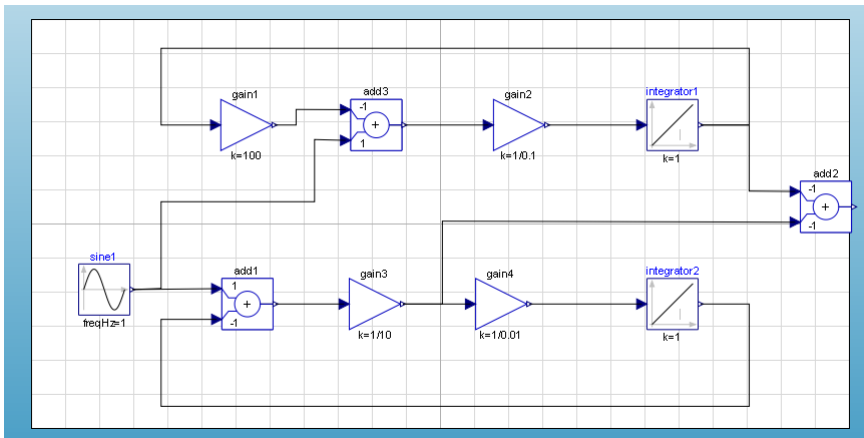


Figure 4-2: The diagram view of the IntroductoryExamples.ComponentBased.BlockCircuit model.

Create a new model, and locate the components in the library browser. All components re-

quired to implement the system with a block-based approach can be found in the following packages:

- Modelica.Blocks.Sources
- Modelica.Blocks.Math
- Modelica.Blocks.Continuous.

To view the components in the Modelica.Blocks.Sources package in the library browser, expand the Modelica package, followed by Blocks and Sources, by clicking the symbol to the left of each package icon and name.

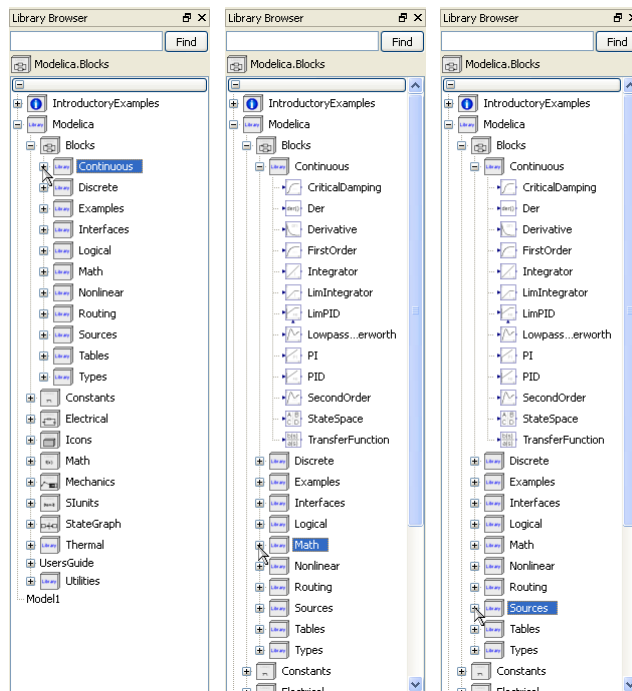


Figure 4-3: Expanding the Continuous, Math, and Sources packages within Modelica.Blocks.

Place the components in the diagram view of your model by dragging them from the library 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.

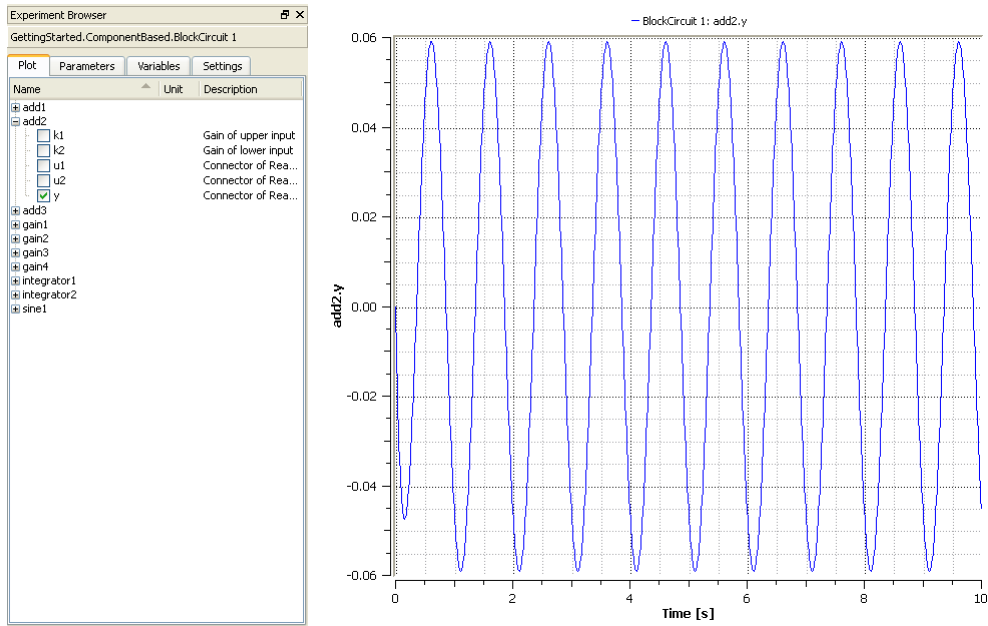


Figure 4-4: Plotting `add2.y` for the `IntroductoryExamples.ComponentBased.BlockCircuit` model with default parameters values.

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 with.

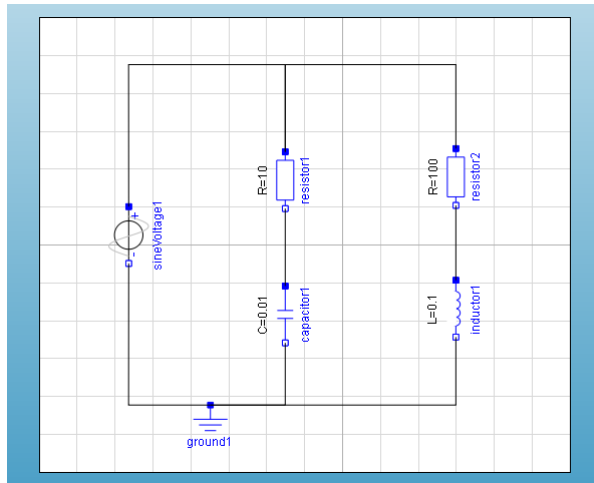


Figure 4-5: The diagram view of the IntroductoryExamples.ComponentBased.ElectricCircuit model.

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 differs 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.

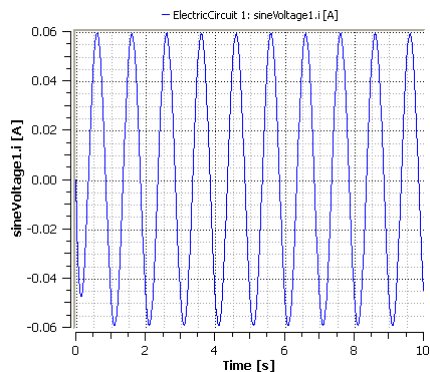


Figure 4-6: Plotting the current going through the source, for the

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.

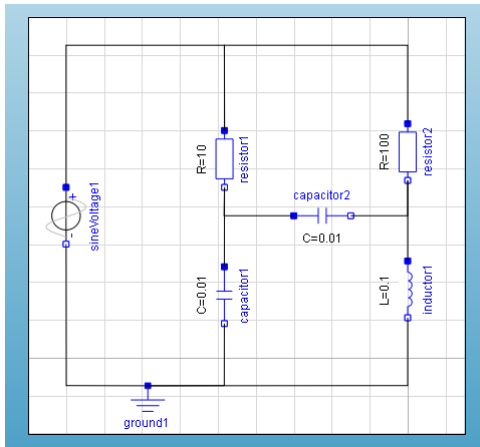


Figure 4-7: The diagram view of the of IntroductoryExamples.ComponentBased.ElectricCircuit2 model.

After simulation we compare the resulting currents with one another.

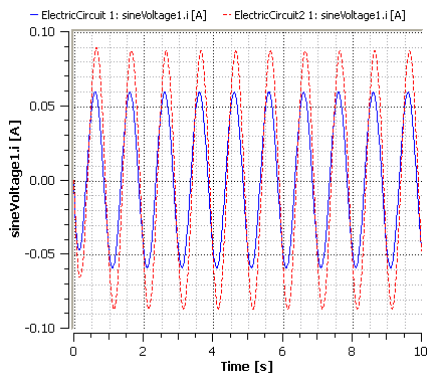


Figure 4-8: Comparison between the two currents going through the source in the ElectricCircuit and ElectricCircuit2 models.

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 MultiBody and Modelica standard libraries included in *MathModelica*. The chain link is inspired by the pendulum example in the 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* "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 details 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 added easily 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 MultiBody.Joints, a body "BoxBody" located in MultiBody.Parts and a rotational damper "Damper" in Modelica.Mechanics.Rotational. Accessing the components in the library browser is explained with more details in section 3.1.

To add a component to the ChainLink model, drag it from the library browser and drop it on the diagram view of the class window. The model is depicted in figure 5-1.

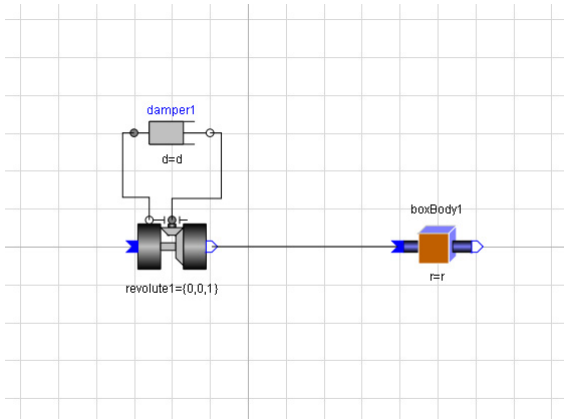


Figure 5-1: Diagram view of ChainLink model.

Once you have added the three components, you need to connect them between each other. Components are connected using the connection line tool.



Figure 5-2: The connection line tool in the toolbar of the model editor.

Only connectors with similar properties should 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 and a message error will appear.

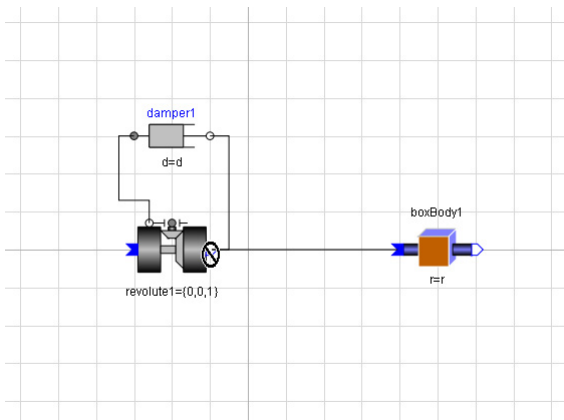


Figure 5-3: Example when the connection line is disabled between two incompatible components.

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_a1 of the boxBody1 to the frame_b1 of the revolute joint. The damper is connected to the revolute joint by using the 1 dimensional mechanical systems connectors, called "flanges".

To connect, for instance frame_a1 (blue connector) of boxBody1 to frame_b1 (white connector with blue border) of the revolute joint component, place the mouse cursor above frame_a1 of boxBody1, press the left mouse button and hold it down while moving the mouse cursor to frame_b1 of the revolute joint component. To make the connection, release the mouse button.

The two flanges of the damper, which are connectors for 1 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_b1 of boxBody1, 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_a1 of the revolute joint is created using the same method.

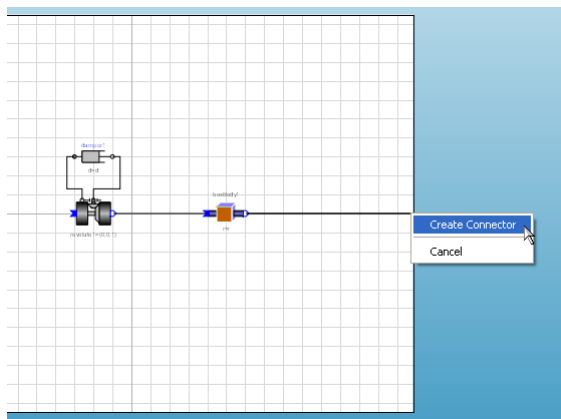


Figure 5-4: Adding a component connector with the connection line tool.

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 boxBody1 (length, width, height)

and the damping coefficient "d" as parameters.

```

model ChainLink
  MultiBody.Joints.Revolute revolutel;
  MultiBody.Parts.BoxBody boxBody1(r=r);
  Modelica.Mechanics.Rotational.Damper damper1(d=d);
  MultiBody.Interfaces.Frame_a frame_a1;
  MultiBody.Interfaces.Frame_b frame_b1;
  parameter Real r[3]={1,0.1,0.1};
  parameter Real d=1.0;

equation
  connect(boxBody1.frame_b,frame_b1);
  connect(revolutel.frame_b,boxBody1.frame_a);
  connect(revolutel.frame_a,frame_a1);
  connect(damper1.flange_b,revolutel.axis);
  connect(damper1.flange_a,revolutel.bearing);
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 in the Standard toolbar.

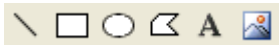


Figure 5-5: The Graphic tools in the Standard toolbar.

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 on 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 a "%" followed by the parameter name. We display the components parameters r and d by adding two text windows with text "%r" and "%d".

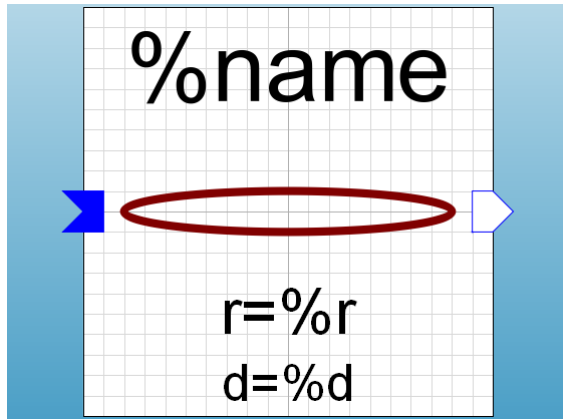


Figure 5-6: Icon view of the ChainLink component.

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 between each other and to the initial frame. The diagram view of the chain pendulum model is represented in figure 5-7.

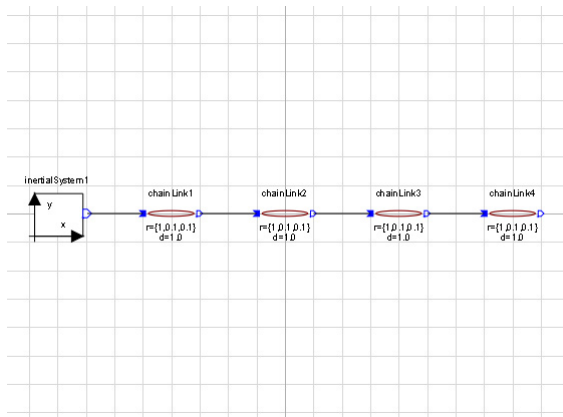


Figure 5-7: Diagram view of the chain pendulum model.

Switch to Simulation Center and simulate the model for 10 seconds. The pendulum animation can be visualized after simulation by choosing Animation in View menu.

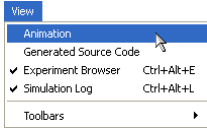


Figure 5-8: Animation is viewed by selecting Animation in View menu.

Figure 5-9 displays the animation view of the pendulum at time 4.52 seconds.

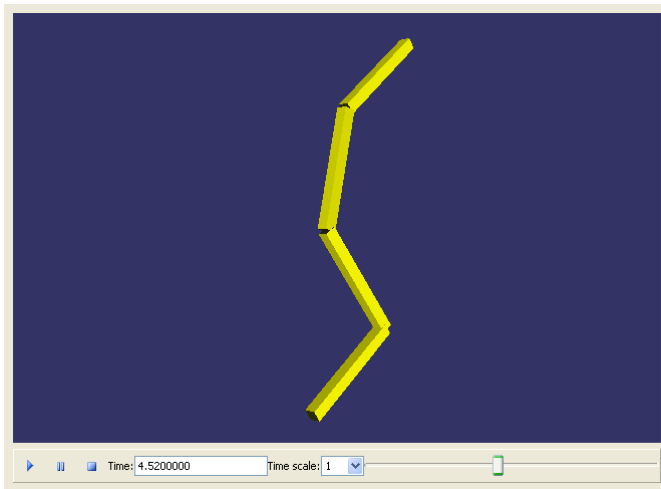


Figure 5-9: Animation view of the chain pendulum at time 4.52 seconds.

Position of the end of the pendulum in x direction (horizontal axis) and y direction (vertical axis) is displayed in figure 5-10.

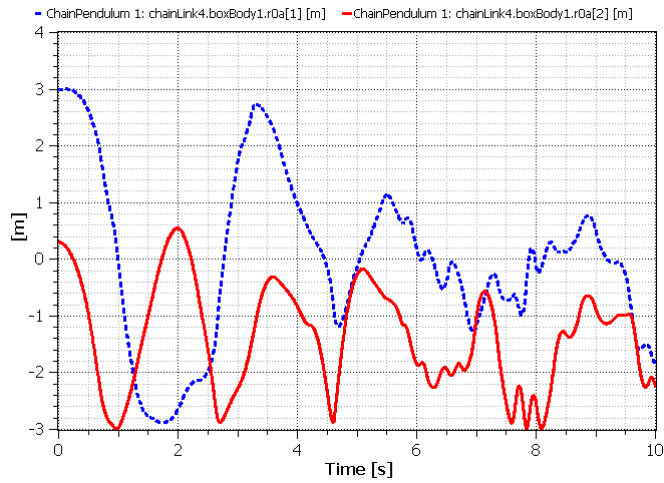


Figure 5-10: Plotting the horizontal position (dotted line) and vertical position of the end of the pendulum.

5.2.1 Exercise

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

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

Chapter 6: External Functions - Chirp Signal

While it is easy to write Modelica functions, it is sometimes convenient to call a subroutine written in C or FORTRAN. 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:

$$\Omega: \omega_1 \leq \omega \leq \omega_2$$

- a certain time period:

$$0 \leq t \leq M$$

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 example 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, if desired 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:

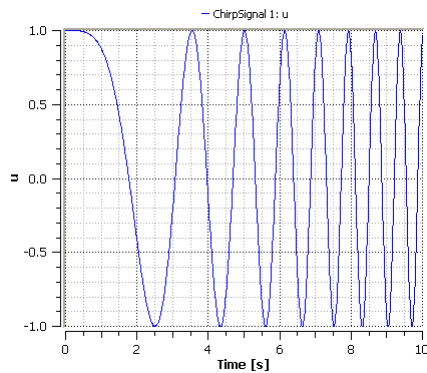


Figure 6-1: Plotting the chirp signal u of the `IntroductoryExamples.ExternalFunctions.SeriesCircuit` model.

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 resonant frequency of the following electrical circuit.

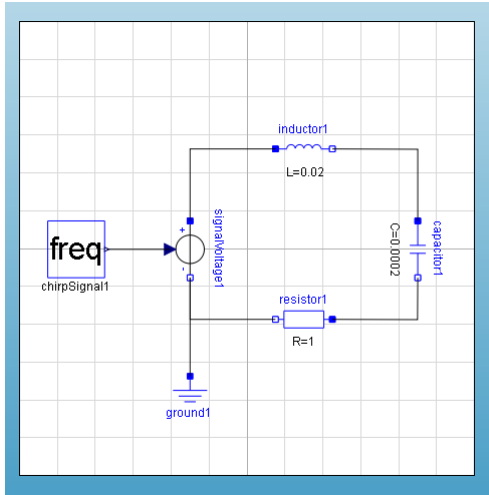


Figure 6-2: The diagram view of the IntroductoryExamples.ExternalFunctions.SeriesCircuit model.

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.

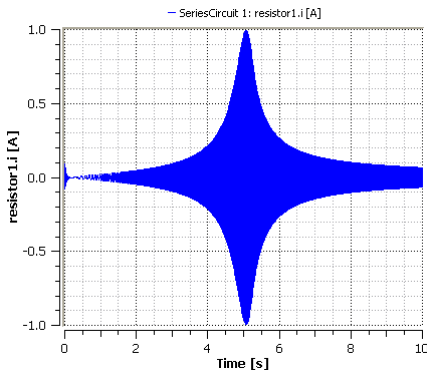


Figure 6-3: Plotting the current i of the IntroductoryExamples.ExternalFunctions.SeriesCircuit model.

As seen we get a top 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: Tank System

This example illustrates how you can build a hierarchical model using *MathModelica System Designer* 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.

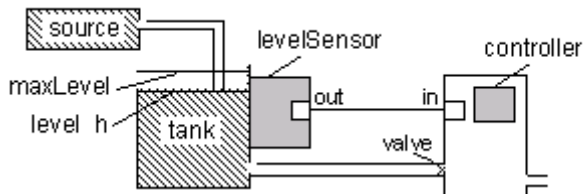


Figure 7-1: A graphical representation of a tank system.

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:

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

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:

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

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.

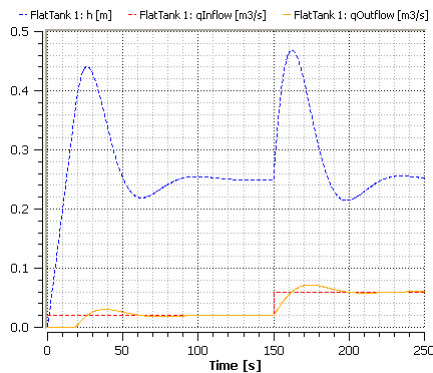


Figure 7-2: Plotting the tank level and the flows through in and out the flat tank with default parameters values.

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 Real variables 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 which 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.

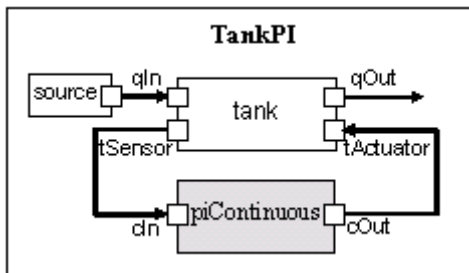


Figure 7-3: A graphical representation of an object-oriented component-based tank system.

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 sub packages. To create a package right-click on the tree root in the library browser and select New Class as shown in Figure 7-4. You can as well right-click on the package you want to add your package to and select New Class.

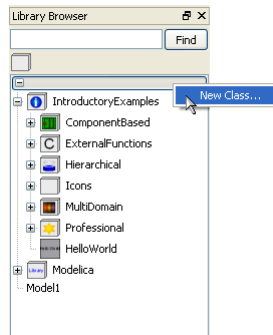


Figure 7-4: Menu to create a new class

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 main tree of the library 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 library 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 Hello World 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 LimitFunction, 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.lfFlow=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 `PIcontinuousController`, 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 Base-Controller, 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.

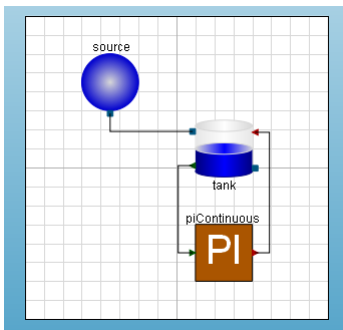


Figure 7-5: The diagram view of the IntroductoryExamples.Hierarchical.TankPI model.

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

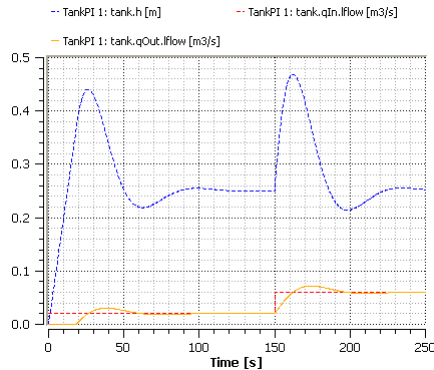


Figure 7-6: Plotting the tank level and the flows through in and out the PI tank with default parameters values.

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, derivative) 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";
  Real y "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.

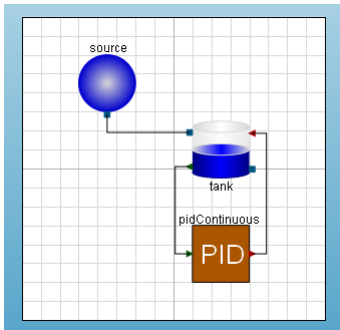


Figure 7-7: The diagram view of the IntroductoryExamples.Hierarchical.TankPID model.

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

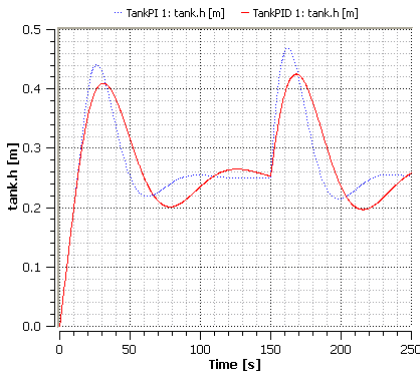


Figure 7-8: Comparison of tank levels between the TankPI model and a TankPID model.

7.4 Three tanks system

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

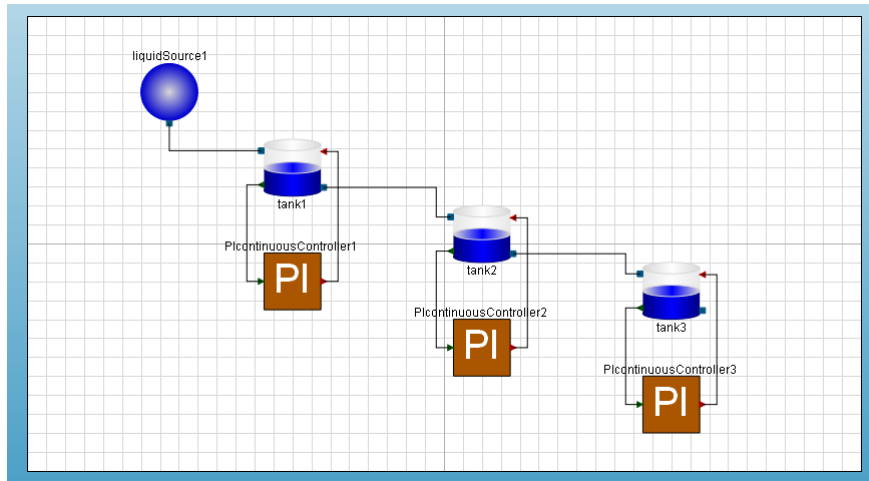


Figure 7-9: The diagram view of the IntroductoryExamples.Hierarachical.TankSystem model. Simulating this system, we can now study how the tank level of each tank is controlled.

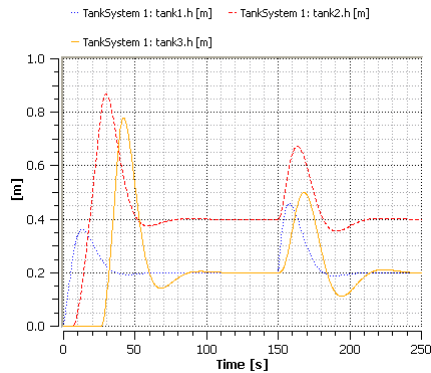


Figure 7-10: Evolution of the tank levels from the TankSystem model.

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

Chapter 8: Systems

This chapter describes complete system models of various applications. Currently the following examples are available.

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 upright position.

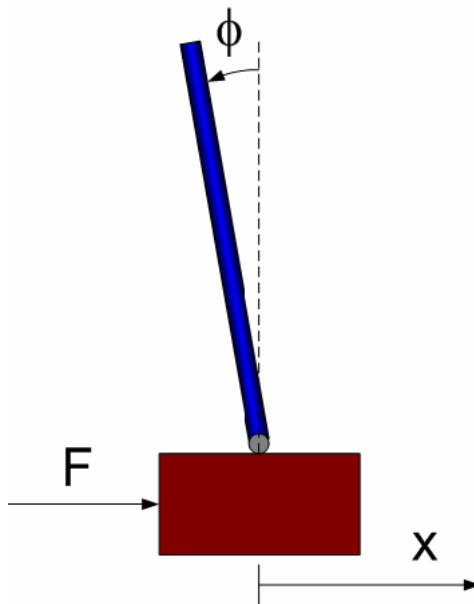


Figure 8-1: Control parameters for the pendulum.

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

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

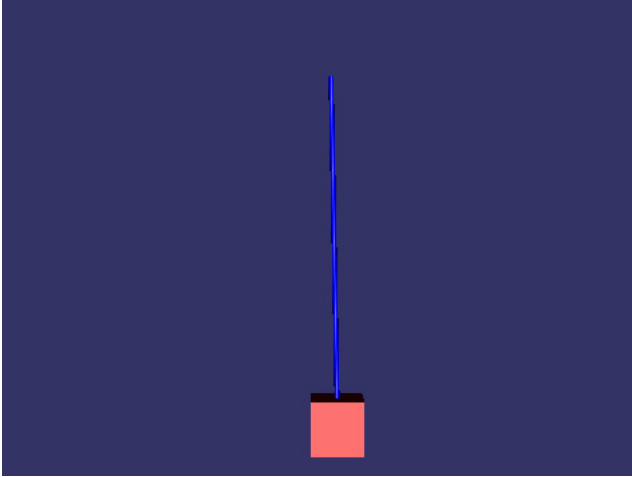


Figure 8-2: Animation of the pendulum system.

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`), length of pendulum (`pendulum.l_pendulum`) and see what happens.

Appendix A: *MathModelica Professional*

This appendix contains notebook examples that show how the notebook interface of *Mathematica* can be used to model, simulate, analyze, and document.

All these examples require *MathModelica Professional*. The examples are also available as separate *Mathematica* notebooks which are delivered with *MathModelica Professional*.

MathModelica® System Designer Professional

Hello World

© *MathCore Engineering AB*

1 Abstract

The most basic Modelica model is a differential equation. In this example a differential equation is implemented and simulated. Note that the Introductory Examples document shows how to develop the same model in the Model Editor and study plots in the Simulation Center.

2 Initialization

To be able to use *MathModelica System Designer* within *Mathematica* we need to load the *MathModelica* package.

```
Needs ["MathModelica`"]
```

3 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 MathModelica, is an equation-based language, printing a string does not make much sense. Instead our Hello World Modelica program solves a trivial differential equation:

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

The variable x in this equation is a dynamic variable (here also a state variable) that can change value over time. The time derivative is the derivative of x , represented as `der(x)` in Modelica.

All Modelica programs consist of class declaration (class, block, model, package, etc) and in this case we declare the program as a model. Note that when working with Modelica models in *Mathematica* it is recommended that you use the `MathModelica` style sheet (this can be found in the `Format>Style Sheet` menu) as this contains a special input cell called `ModelicaInput`. Even though you can type your Modelica models in normal Input cells, `ModelicaInput` cells are better suited for these purposes.

Just as any other *Mathematica* input `ModelicaInput` cells are evaluated by pressing `shift+enter`.

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

  equation
    der(x)=-x;
end HelloWorld;
```

4 Simulation

The *MathModelica System Designer Professional* command **Simulate** simulates the model given by the *Modelica* code shown above.


```
?? Simulate
```

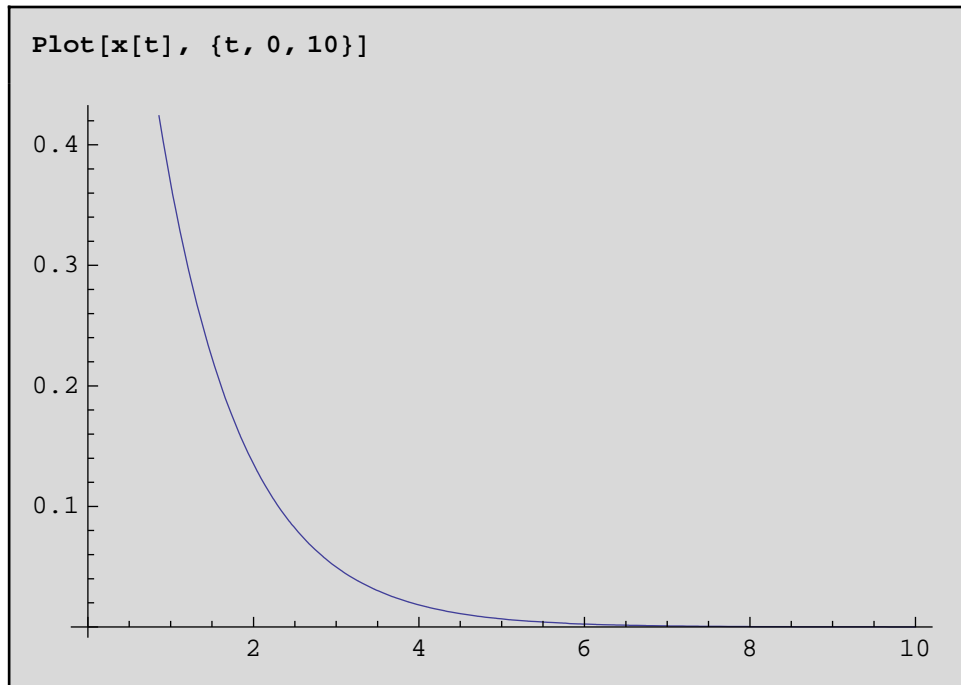
```
Simulate[model,{t,tmin,tmax}]  
    simulates the model in the range tmin to tmax.
```

```
Attributes[Simulate] = {ReadProtected}  
  
Options[Simulate] = {Tolerance -> 0.0001, Model -> ,  
    IntervalLength -> 0, NumberOfIntervals -> Automatic,  
    InitialValues -> {}, ParameterValues -> {},  
    InteractiveVariables -> True, ShowNotifications -> True,  
    Sensitivities -> {}, Method -> dassl}
```

We simulate the HelloWorld example.

```
Simulate[HelloWorld, {t, 0, 10}]  
  
<SimulationData: "HelloWorld" : : {0., 10.}  
    : 1007 data points : 0 events : 2 variables>  
{x', x}
```

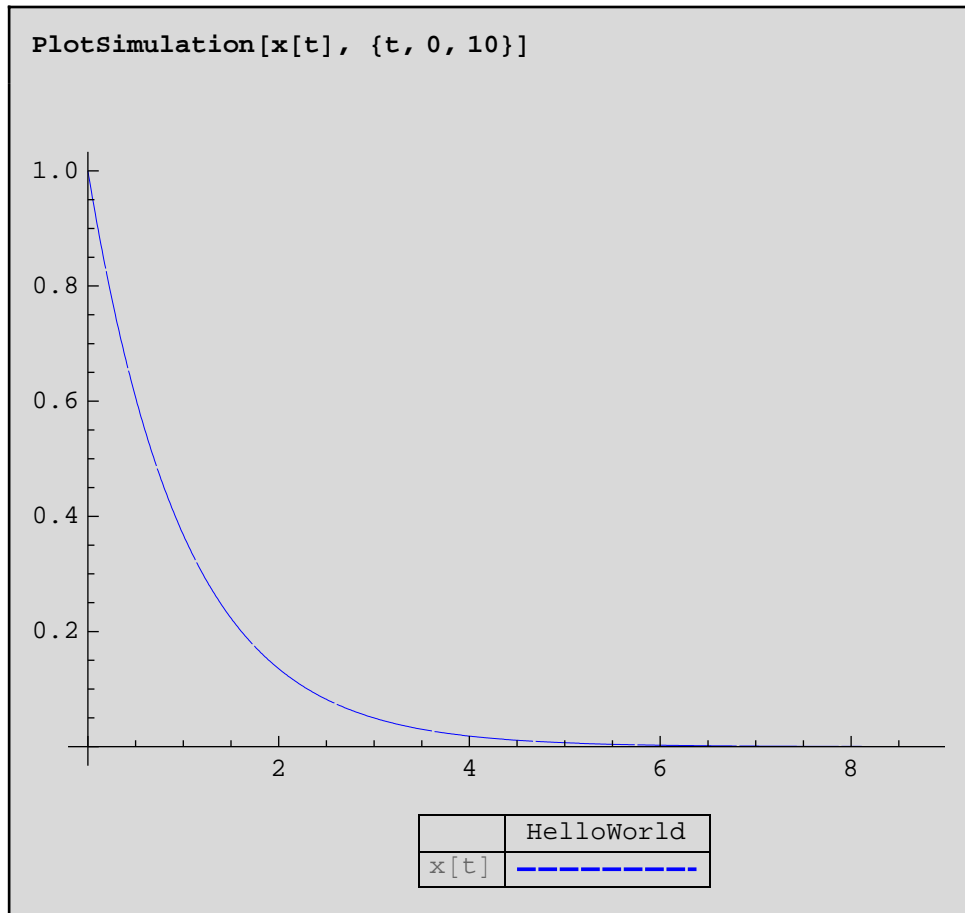
The Simulate command returns a SimulationData object containing all model parameters and variables in the form of interpolating functions. It is possible to use the *Mathematica* Plot command to plot simulation results.



However, it is better to use the `PlotSimulation` command as this is especially adapted to handle simulation results, and it also has the ability to plot simulation results from several different simulations in the same plot, using the `SimulationResult` option. It also has the ability to plot hierarchical variables, which are very often used in modeling.

?PlotSimulation

`PlotSimulation[f,{t,tmin,tmax}]` generates a plot of the signal `f` as a function of `t` from `tmin` to `tmax`.
`PlotSimulation[{f1,f2,...},{t,tmin,tmax}]` plots several functions `fi`. `PlotSimulation[{"var1","var2"},{t,tmin,tmax}]` plots variables having the Modelica names `var1` and `var2`.
`PlotSimulation[f,{t,tmin,tmax},SimulationResult->res]` generates a plot of `f` using simulation result `res`. Default value of `res` is the result from the latest simulation.



In many cases you want to compare results for different simulations, for instance you might want to test different initial or parameter values.

?? Simulate

`Simulate[model,{t,tmin,tmax}]`
simulates the model in the range tmin to tmax.

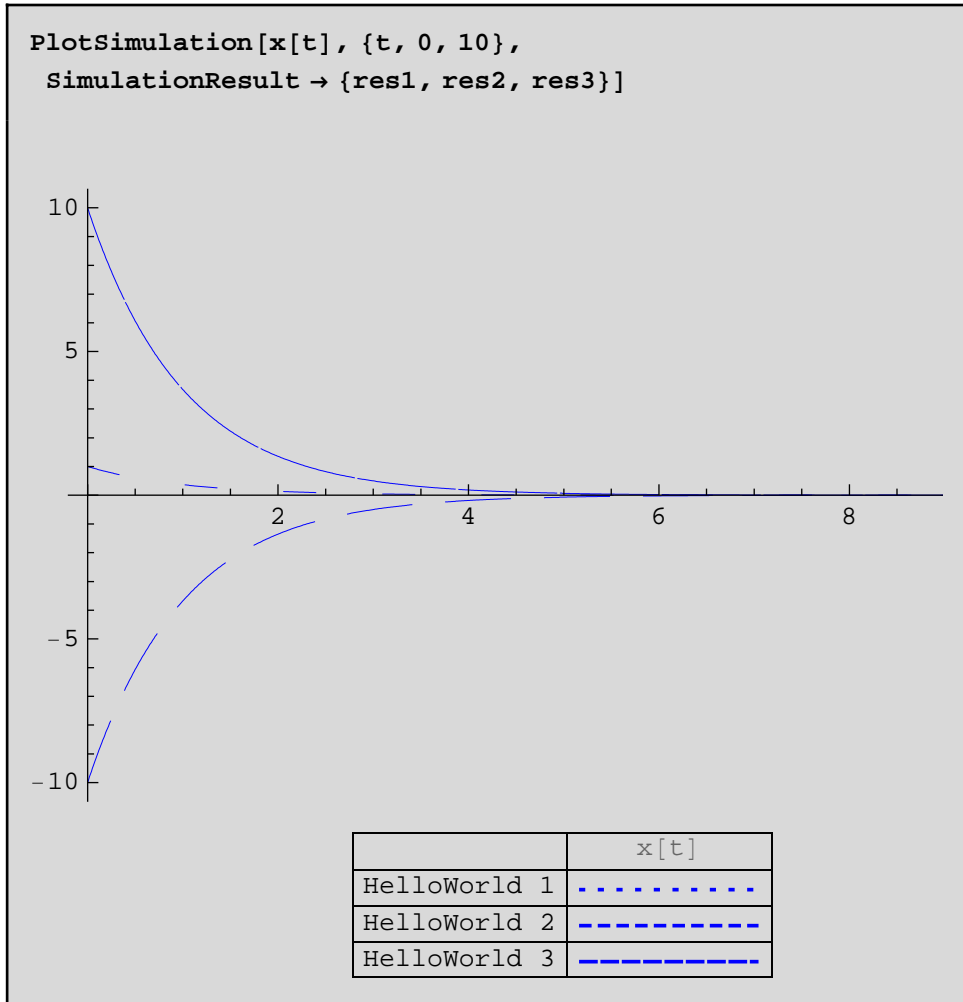
```
Attributes[Simulate] = {ReadProtected}
```

```
Options[Simulate] = {Tolerance -> 0.0001, Model -> ,  
  IntervalLength -> 0, NumberOfIntervals -> Automatic,  
  InitialValues -> {}, ParameterValues -> {},  
  InteractiveVariables -> True, ShowNotifications -> True,  
  Sensitivities -> {}, Method -> dassl}
```

We can make several simulations and store the results in different variables.

```
res1 = Simulate[HelloWorld, {t, 0, 10}];  
res2 = Simulate[HelloWorld,  
  {t, 0, 10}, InitialValues -> {x == -10}];  
res3 = Simulate[HelloWorld, {t, 0, 10},  
  InitialValues -> {x == 10}];
```

Then we can use the `SimulationResult` option to compare the results in one plot.



The option `PlotStyle` enables you to specify the colors, dashing and characteristics of the plots.

MathModelica® System Designer Professional

PlotSimulation

© *MathCore Engineering AB*

1 Abstract

This notebook gives examples on how to use PlotSimulation, ParametricPlotSimulation and ParametricPlotSimulation3D.

2 Initialization

To be able to use *MathModelica System Designer* within *Mathematica* we need to load the *MathModelica* package.

```
Needs["MathModelica`"];
```

3 PlotSimulation

First define a simple test model. In this case we define a model of a bouncing ball. Please refer to the Events.nb notebook for a more detailed model.

```
model BouncingBall "Simple model of a  
bouncing ball"
```

```
constant Real g=9.81;
parameter Real c=0.9;
parameter Real r=0.1;
Real x(start=1), y(start=0);

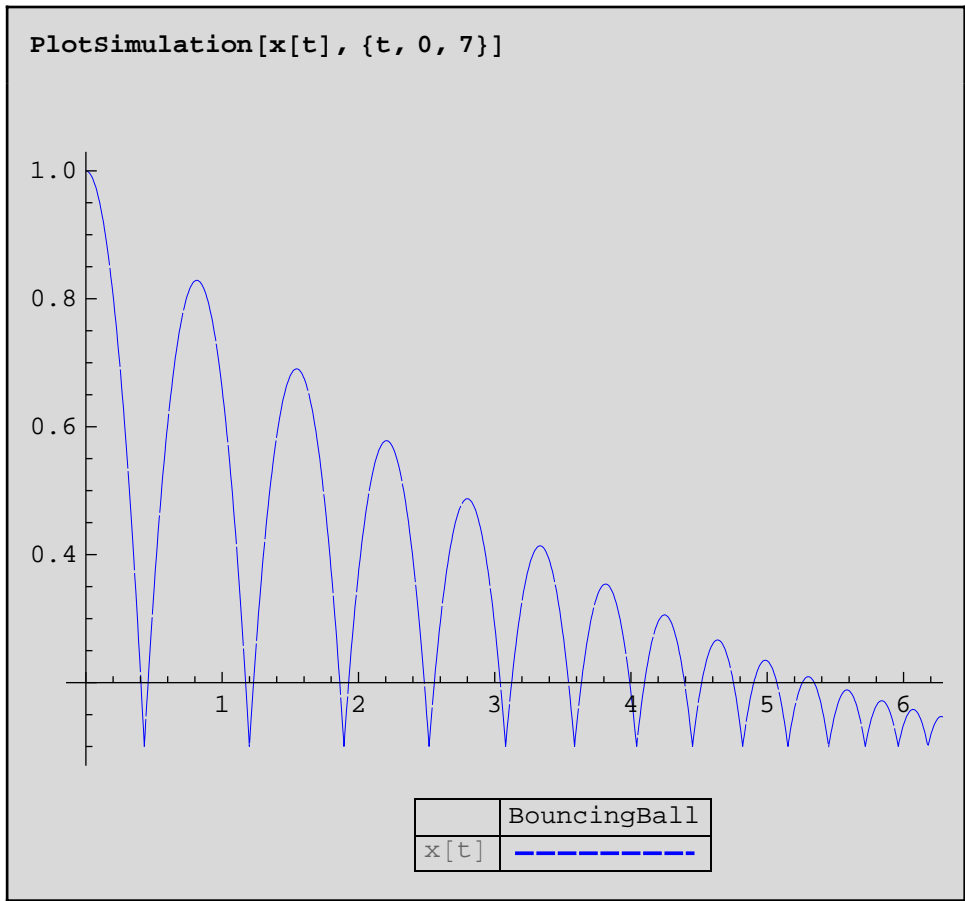
equation
  der(x) = y;
  der(y) = -g;
  when x < r then
    reinit(y, (-c)*pre(y));
  end when;
end BouncingBall;
```

Simulate the system and show the simulation log. The simulation data is returned as SimulationData objects.

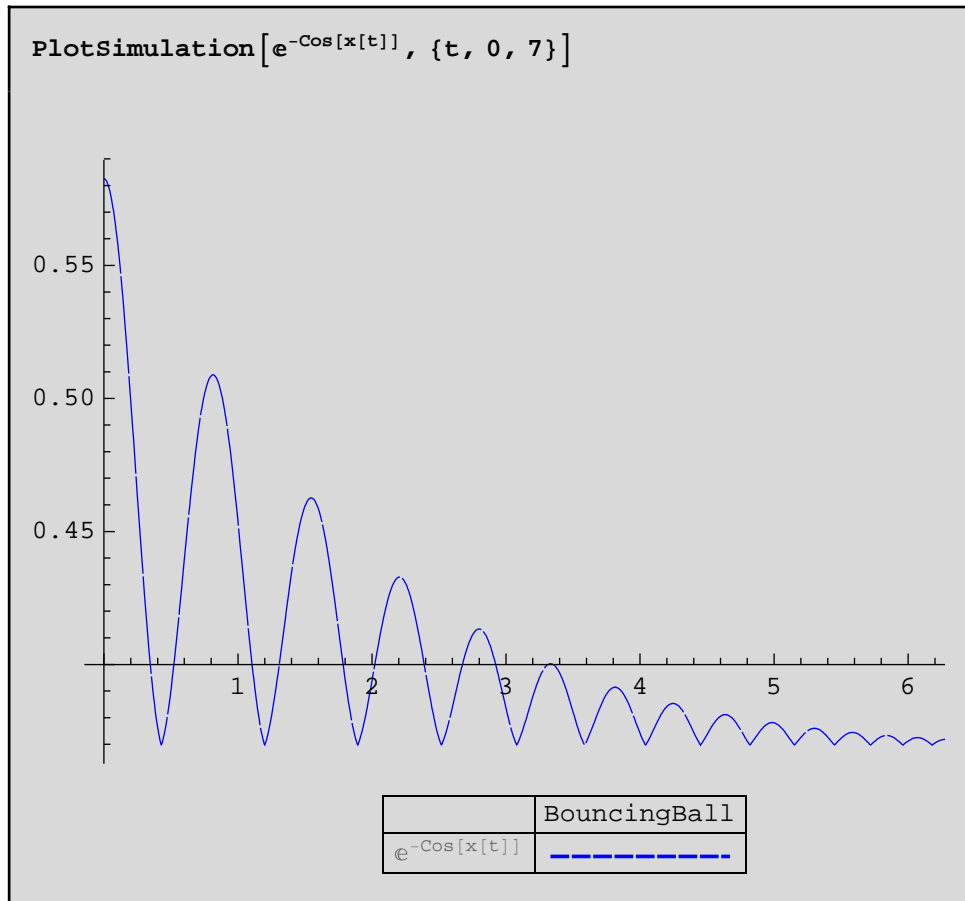
```
res1 = Simulate[BouncingBall, {t, 0, 7}]

<SimulationData: "BouncingBall" : : {0., 7.}
 : 1220 data points : 19 events : 6 variables>
{c, x', y', r, x, y}
```

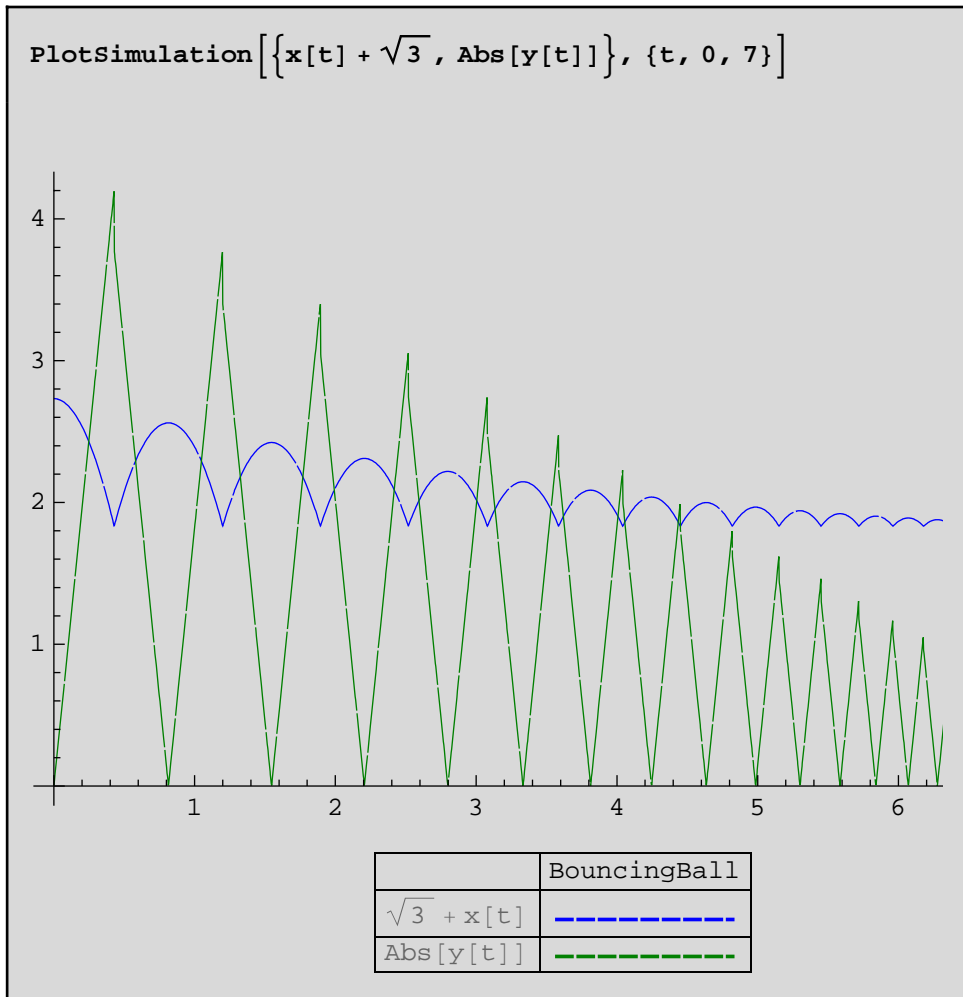
Plot simulation is used in the following way.



It is possible to use any expressions when plotting.



Plotting several functions can be done in the following way:



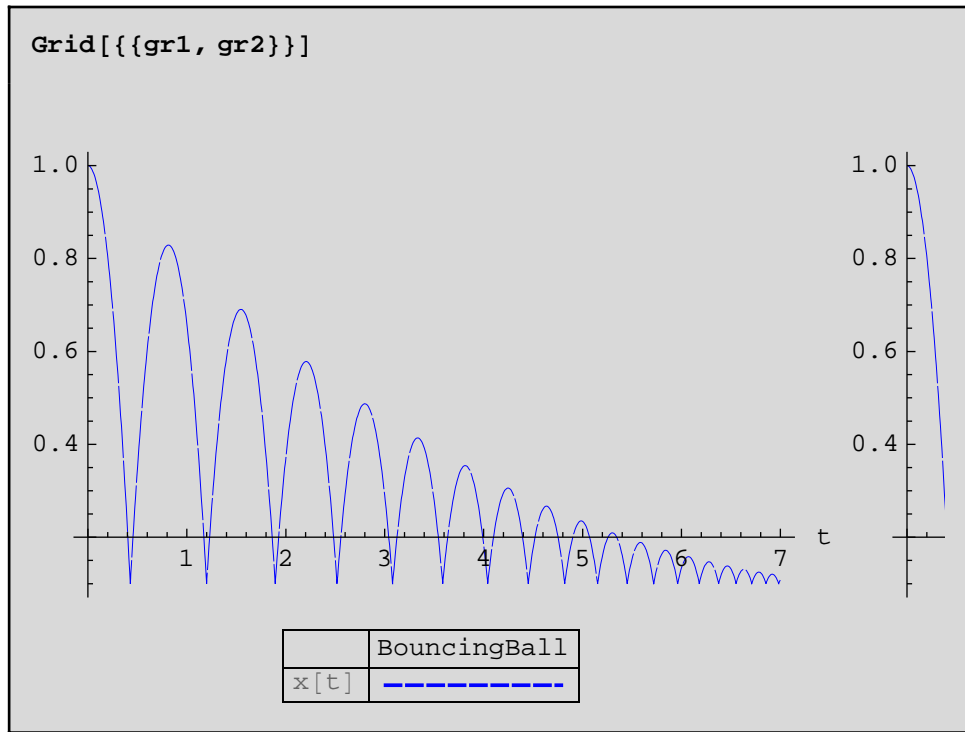
Generate an additional simulation and store the result in res2.

```
res2 = Simulate[BouncingBall,  
  {t, 0, 7}, ParameterValues → c == 0.95]  
  
<SimulationData: "BouncingBall" : : {0., 7.}  
  : 1129 data points : 11 events : 6 variables>  
{c, x', y', r, x, y}
```

The option **SimulationResult** specifies which simulation data to use.

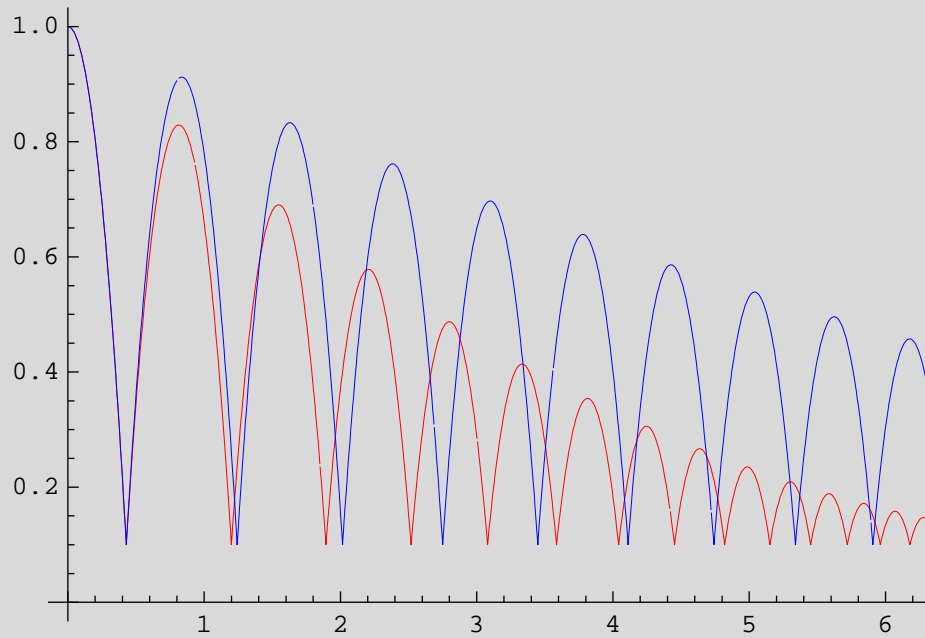
```
gr1 = PlotSimulation[x[t], {t, 0, 7},  
  SimulationResult → res1, ImageSize → 300];
```

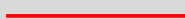

```
gr2 = PlotSimulation[x[t], {t, 0, 7},  
  SimulationResult → res2, ImageSize → 300];
```

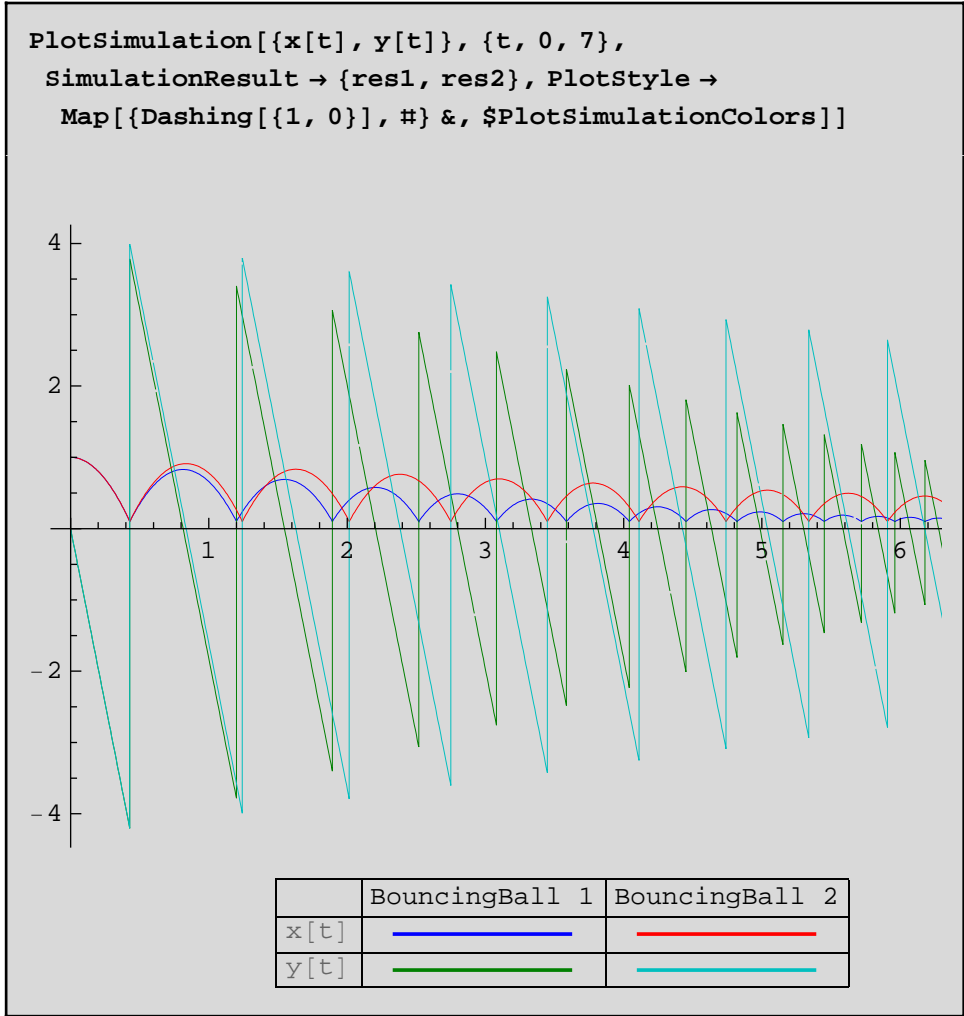


It is possible to plot functions using data from several simulations.

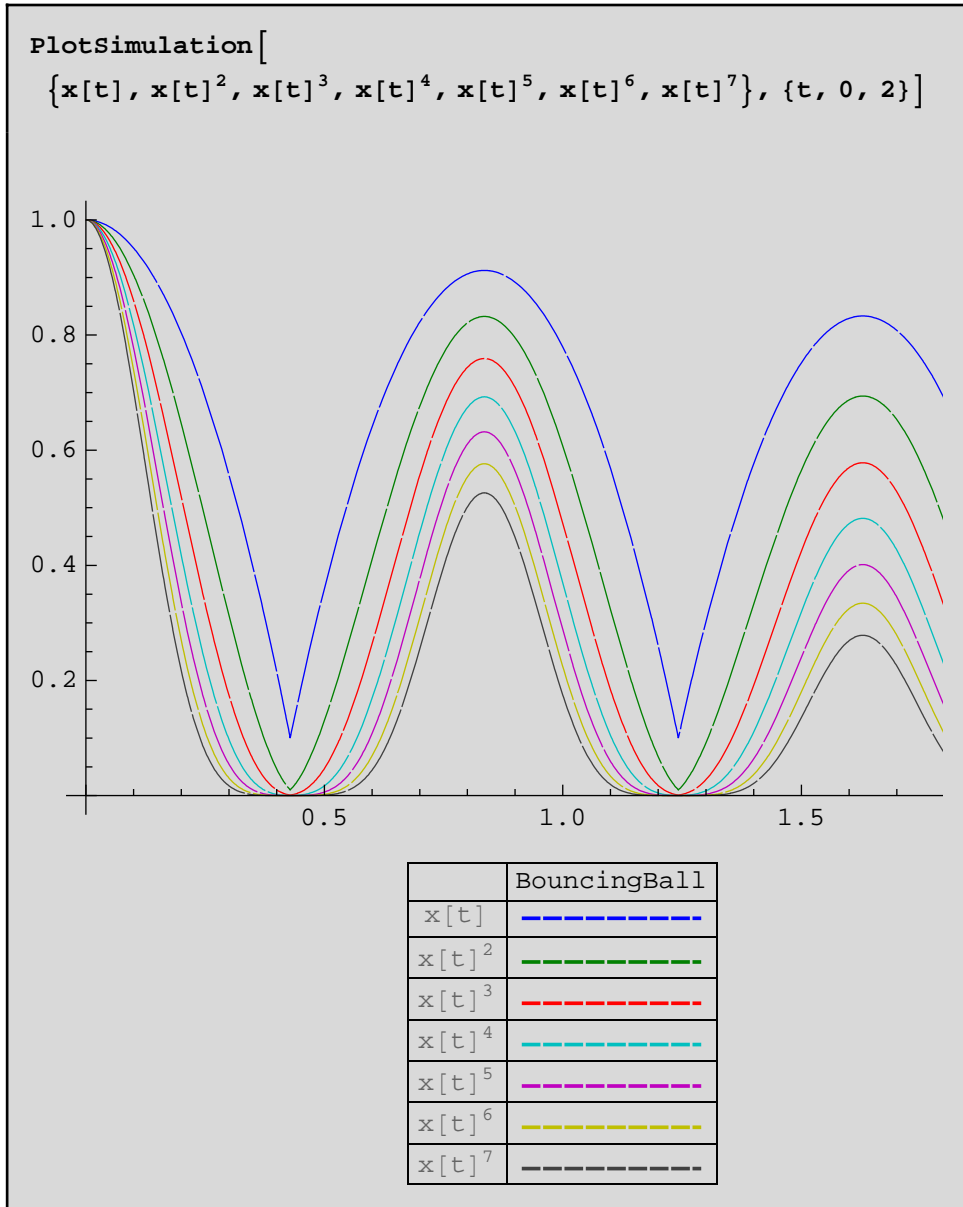
```
PlotSimulation[x[t], {t, 0, 7},  
SimulationResult → {res1, res2},  
PlotRange → Automatic, PlotStyle →  
{ {Dashing[{1, 0}], Red}, {Dashing[{1, 0}], Blue} }]
```



	$x[t]$
BouncingBall 1	
BouncingBall 2	

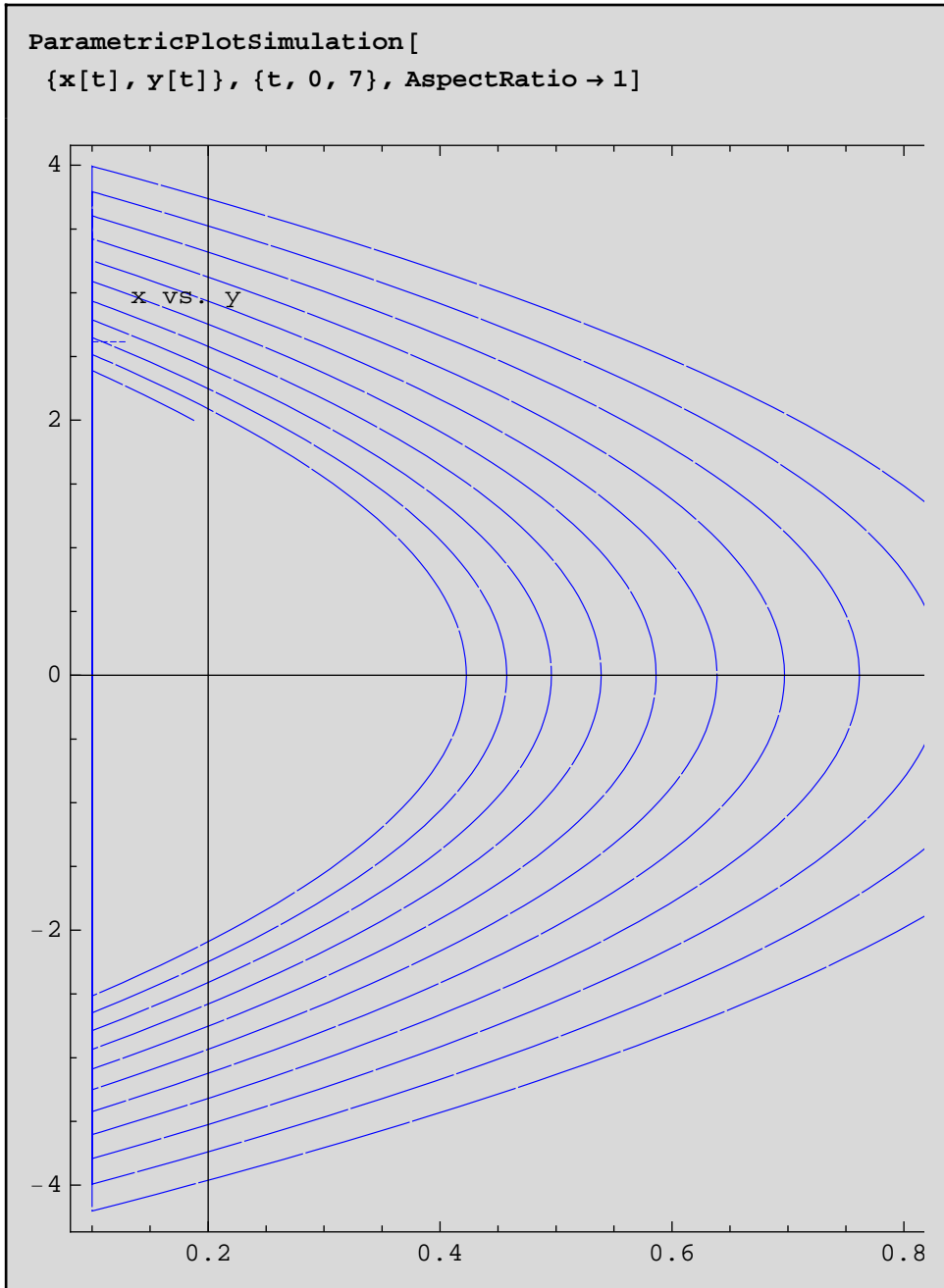


Checking color scheme



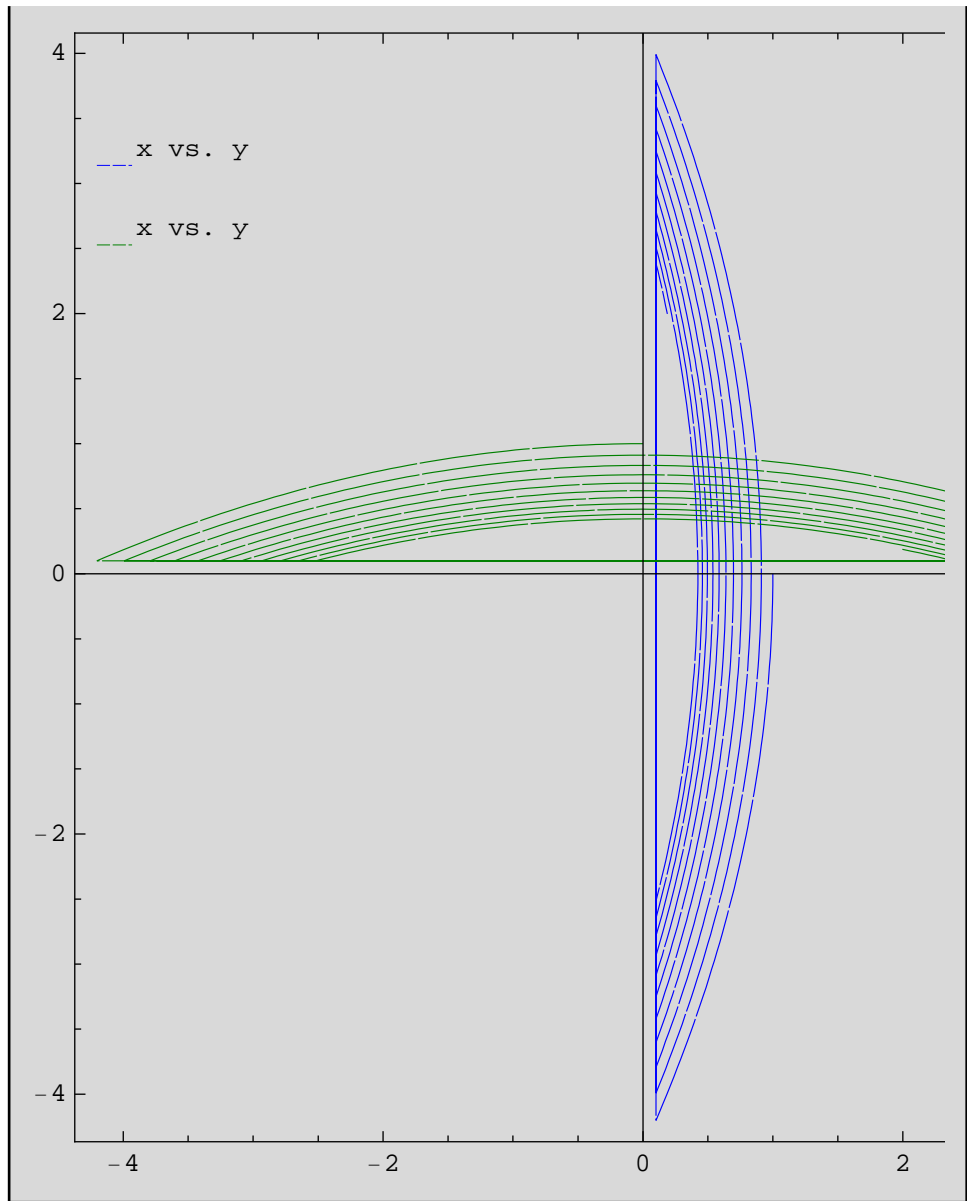
ParametricPlotSimulation

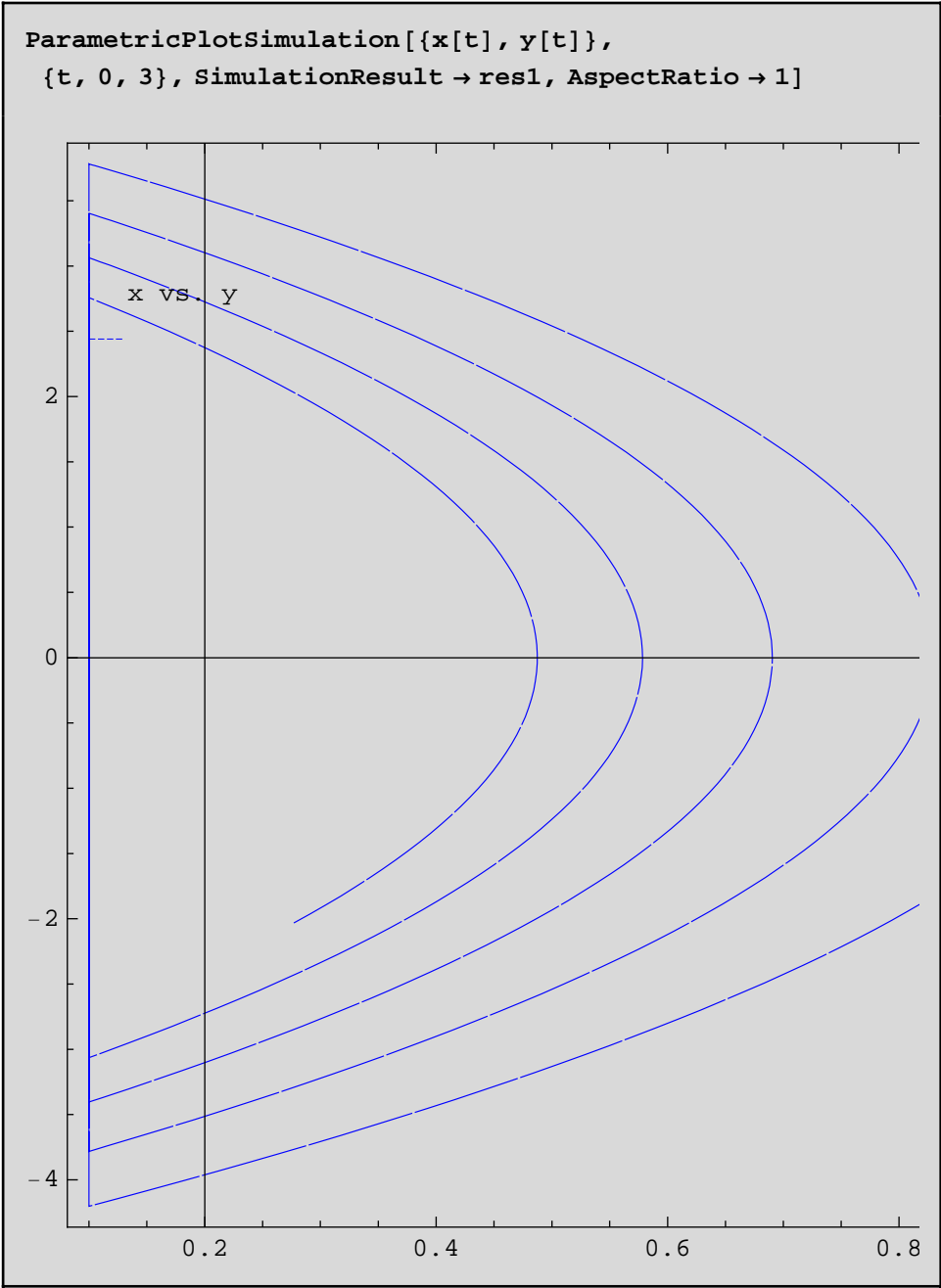
Parametric plots can be done using ParametricPlotSimulation.

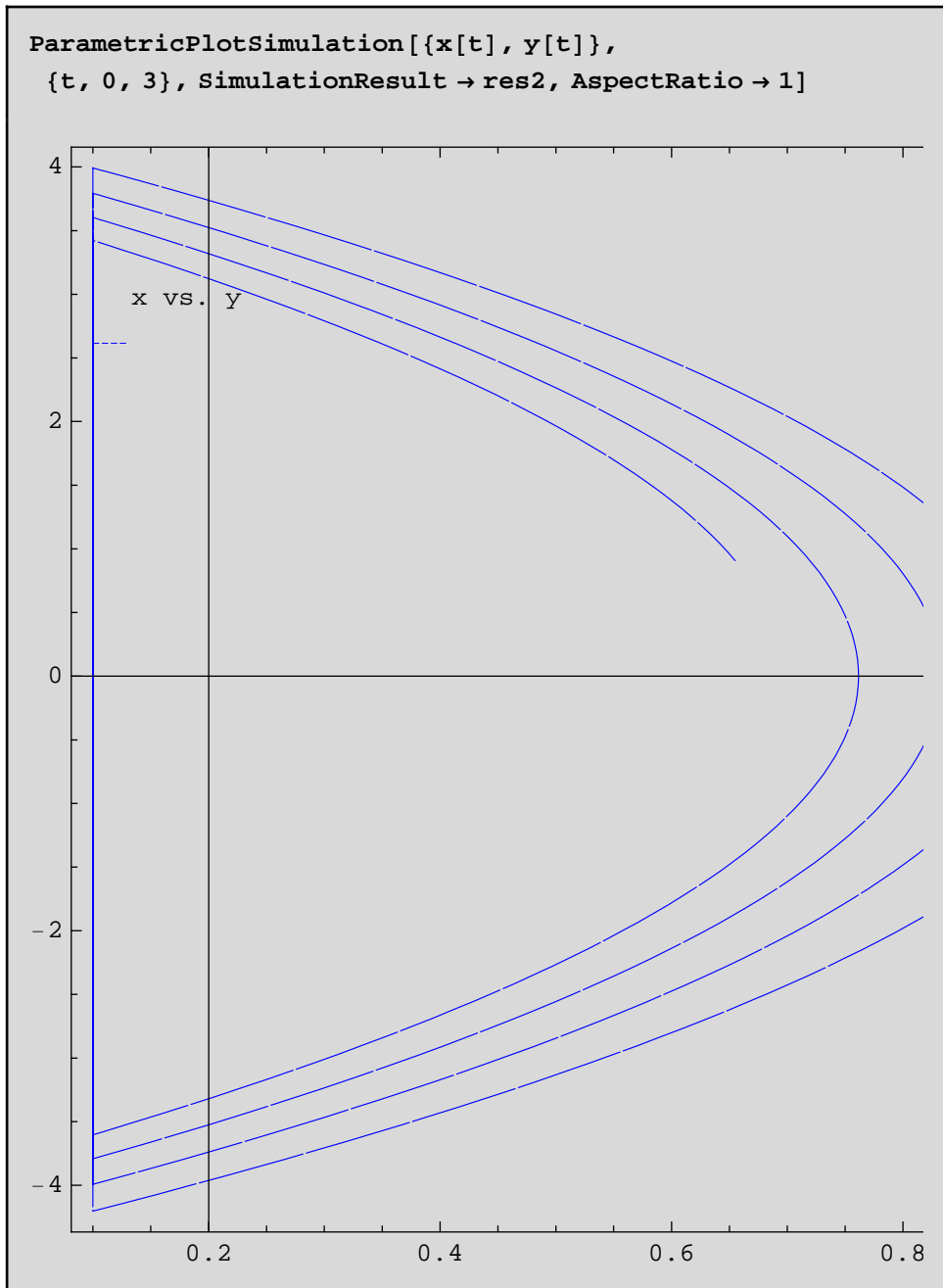


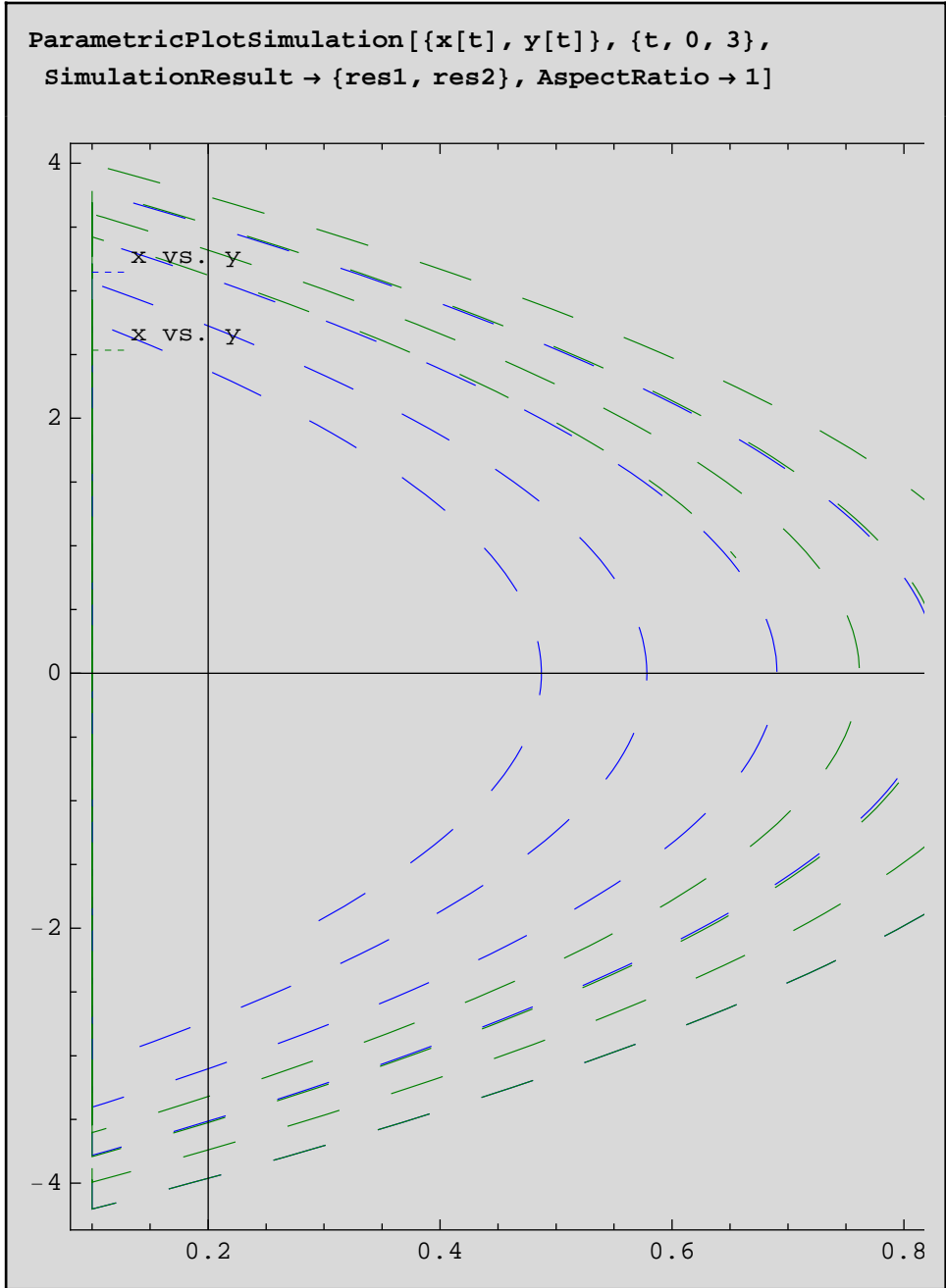
Several plots:

```
ParametricPlotSimulation[  
  {{x[t], y[t]}, {y[t], x[t]}}, {t, 0, 7}, AspectRatio -> 1]
```









ParametricPlotSimulation3D

Another simple model is the rossler attractor.

```

model Rossler "Rossler attractor"
  parameter Real alpha=0.2;
  parameter Real beta=0.2;
  parameter Real gamma=8;
  Real x(start=1);
  Real y(start=3);
  Real z(start=0);

equation
  der(x) = (-y)-z;
  der(y) = x+alpha*y;
  der(z) = (beta+x*z)-gamma*z;
end Rossler;

```

Simulate the model.

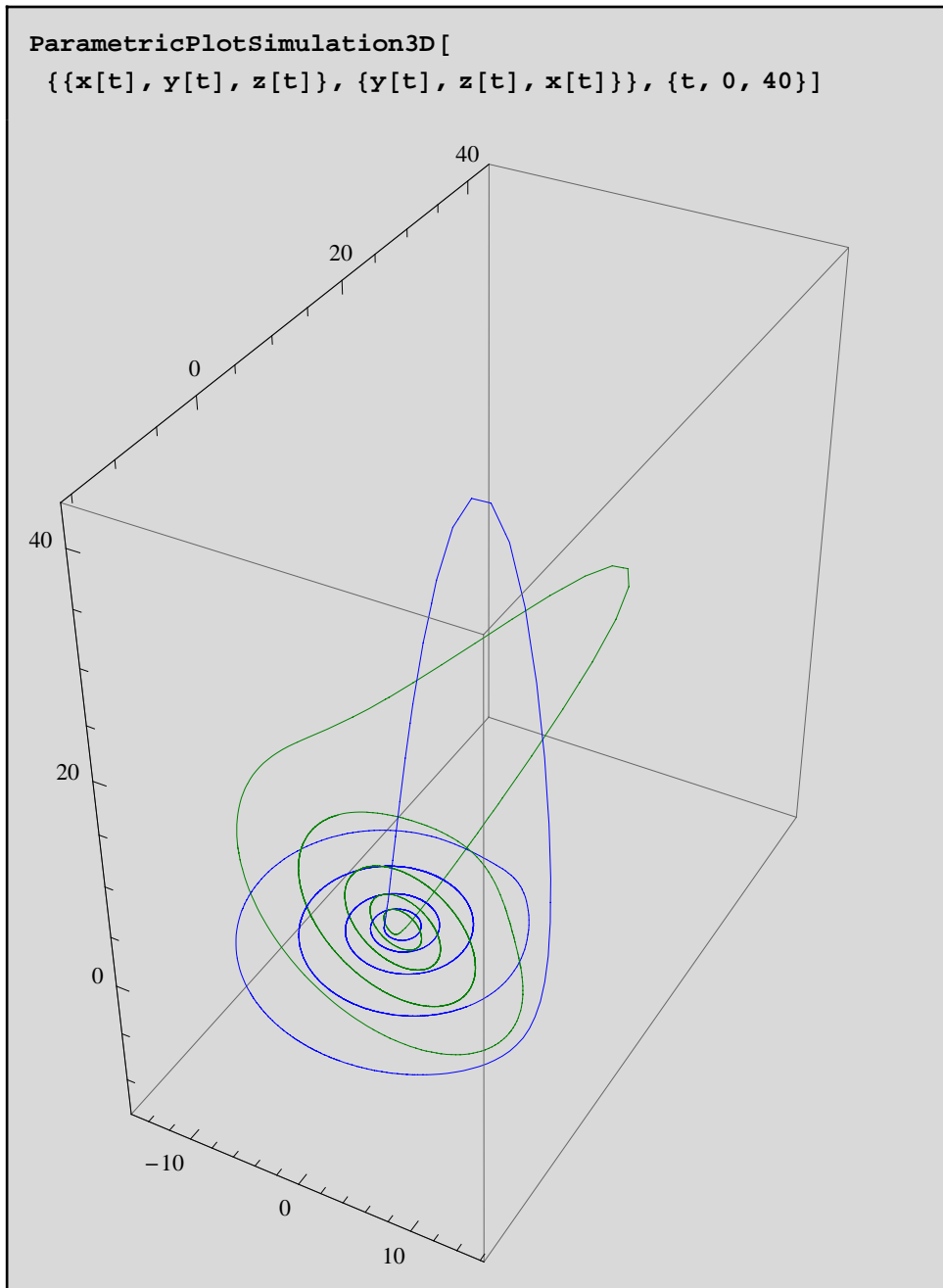
```

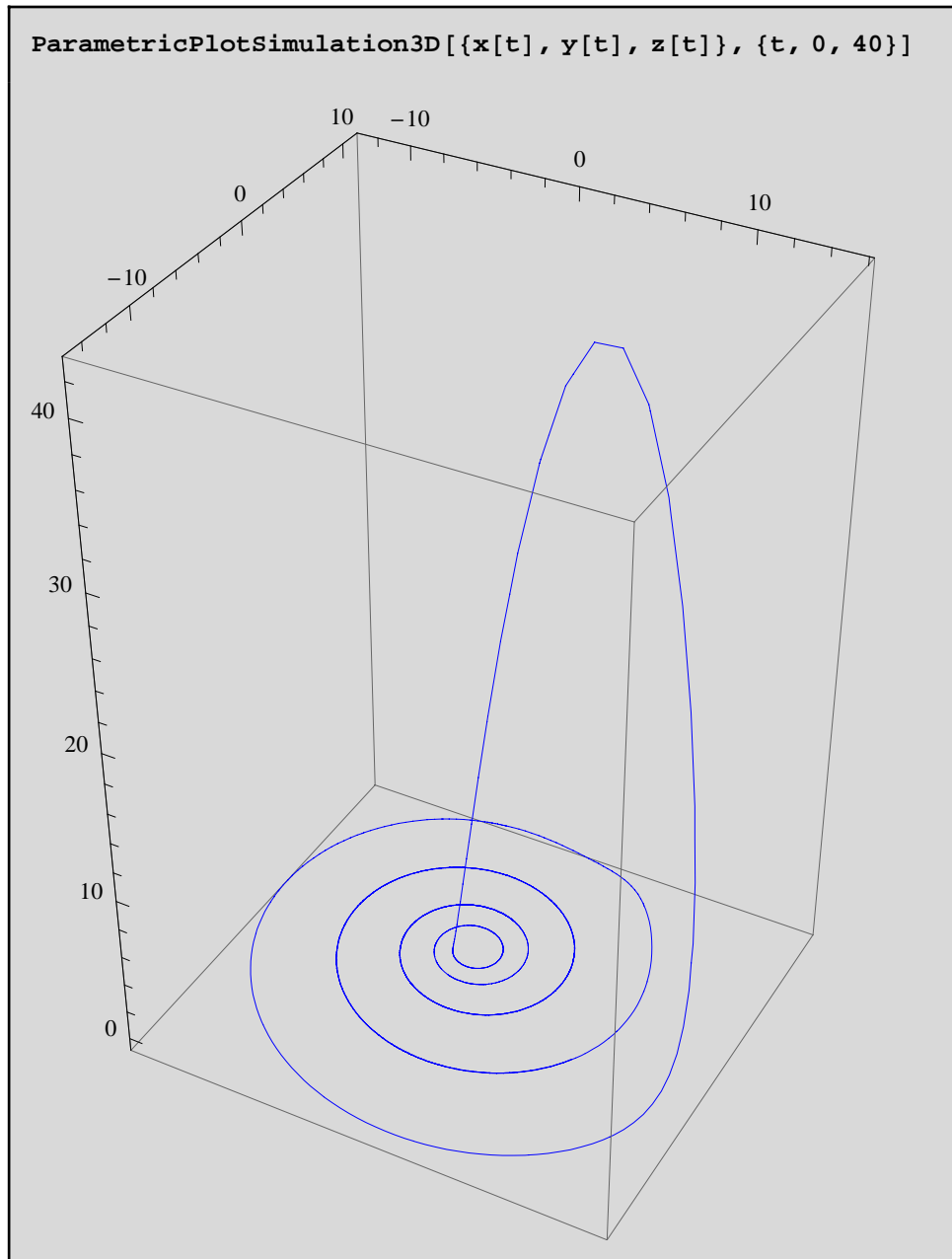
res = Simulate[Rossler, {t, 0, 40},
  InitialValues → {x == 2, y == 2.5, z == 0},
  NumberOfIntervals → 1000]

<SimulationData: "Rossler" : : {0., 40.}
 : 1002 data points : 0 events : 9 variables>
{α, β, x', y', z', γ, x, y, z}

```

We can make parametric plots in 3 dimensions, using ParametricPlot3D.





Options

Options [PlotSimulation]

```

{AlignmentPoint → Center, AspectRatio →  $\frac{1}{\text{GoldenRatio}}$ ,
  Axes → True, AxesLabel → Automatic,
  AxesOrigin → Automatic, AxesStyle → {},
  Background → None, BaselinePosition → Automatic,
  BaseStyle → {}, ClippingStyle → None,
  ColorFunction → Automatic, ColorFunctionScaling → True,
  ColorOutput → Automatic, ContentSelectable → Automatic,
  DisplayFunction := $DisplayFunction, Epilog → {},
  Evaluated → Automatic, EvaluationMonitor → None,
  Exclusions → Automatic, ExclusionsStyle → None,
  Filling → None, FillingStyle → Automatic,
  FormatType := TraditionalForm, Frame → False,
  FrameLabel → None, FrameStyle → {},
  FrameTicks → Automatic, FrameTicksStyle → {},
  GridLines → None, GridLinesStyle → {},
  ImageMargins → 0., ImagePadding → All,
  ImageSize → 400, LabelStyle → {}, Legend → True,
  LegendBackground → Automatic, LegendBorder → Automatic,
  LegendBorderSpace → Automatic, LegendLabel → ,
  LegendLabelSpace → 0, LegendOrientation → Vertical,
  LegendPosition → {-0.9, 0.57}, LegendShadow → None,
  LegendSize → {1, 0.3}, LegendSpacing → Automatic,
  LegendTextDirection → Automatic,
  LegendTextOffset → Automatic, LegendTextSpace → 15,
  MaxRecursion → Automatic, Mesh → None,
  MeshFunctions → {#1 &}, MeshShading → None,
  MeshStyle → Automatic, Method → Automatic,
  PerformanceGoal := $PerformanceGoal,
  PlotJoined → True, PlotLabel → None,
  PlotLegend → Automatic, PlotPoints → 500,

```

```
PlotRange → All, PlotRangeClipping → True,  
PlotRangePadding → Automatic, PlotRegion → Automatic,  
PlotStyle → Automatic, PreserveImageOptions → Automatic,  
Prolog → {}, RegionFunction → (True &),  
RotateLabel → True, ShadowBackground → GrayLevel[0],  
SimulationResult := $SimulationResult, Ticks → Automatic,  
TicksStyle → {}, WorkingPrecision → MachinePrecision}
```

The colors are set by default to the following, added to a dashing corresponding to each simulation:

\$PlotSimulationColors

```
{RGBColor[0, 0, 1], RGBColor[0, 0.5, 0], RGBColor[1, 0, 0],  
RGBColor[0, 0.75, 0.75], RGBColor[0.75, 0, 0.75],  
RGBColor[0.75, 0.75, 0], RGBColor[0.25, 0.25, 0.25]}
```

MathModelica® System Designer Professional

Hybrid DC Motor

© MathCore Engineering AB

1 Abstract

In this example the *Model Editor* has been used to build a DC motor and a weak axis with an applied torque. The torque is inactive for some time when the axis starts to spin. Note that some connectors do not have arrows indicating the direction of the signal. This means that the signal does not have a specified direction. These components contain equations (not assignments) making them more usable since the equations can be solved without taking the signal flow into account.

2 Initialization

To be able to use MathModelica within *Mathematica* we need to load the MathModelica package.

```
Needs["MathModelica`"];
```

3 Hybrid Motor

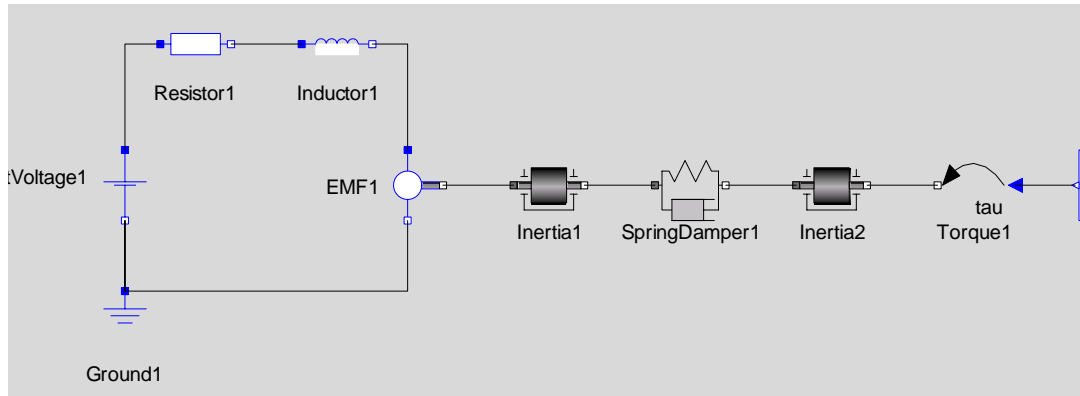


Diagram : HybridMotor

The corresponding model code is shown below.

```

model HybridMotor
  Modelica.Electrical.Analog.Basic.Ground
  Ground1;
  Modelica.Electrical.Analog.Basic.Resistor
  Resistor1;

  Modelica.Electrical.Analog.Sources.ConstantVol
  tage ConstantVoltage1;
  Modelica.Electrical.Analog.Basic.Inductor
  Inductor1;
  Modelica.Electrical.Analog.Basic.EMF EMF1;
  Modelica.Mechanics.Rotational.Inertia
  Inertia1;
  Modelica.Mechanics.Rotational.SpringDamper
  SpringDamper1(d=0.3,c=1);
  Modelica.Mechanics.Rotational.Inertia
  Inertia2;
  Modelica.Mechanics.Rotational.Torque

```

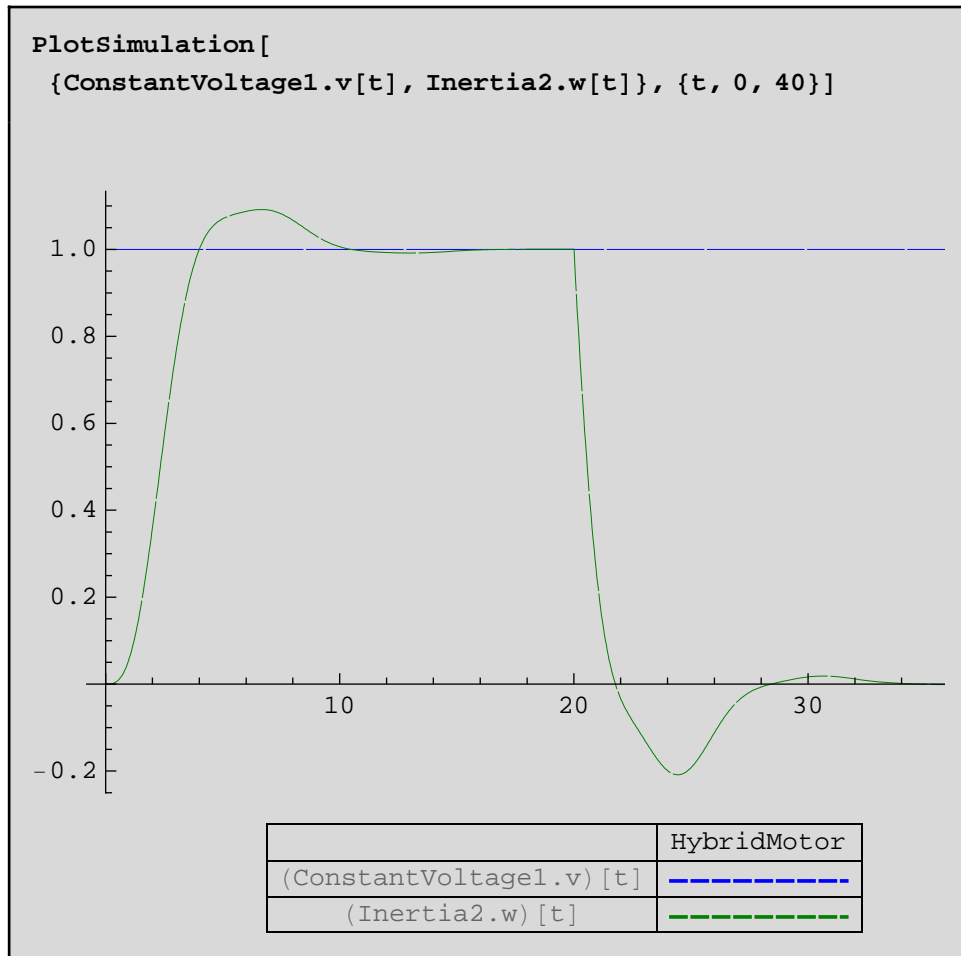
```
Torque1;  
  Modelica.Blocks.Sources.Step  
Step1(startTime=20,height=-1);  
  
equation  
  connect(Ground1.p, ConstantVoltage1.n);  
  connect(ConstantVoltage1.p, Resistor1.p);  
  connect(Inductor1.n, EMF1.p);  
  connect(ConstantVoltage1.n, EMF1.n);  
  connect(EMF1.flange_b, Inertial.flange_a);  
  connect(Inertial.flange_b,  
SpringDamper1.flange_a);  
  connect(SpringDamper1.flange_b,  
Inertia2.flange_a);  
  connect(Resistor1.n, Inductor1.p);  
  connect(Step1.y, Torque1.tau);  
  connect(Torque1.flange_b,  
Inertia2.flange_b);  
end HybridMotor;
```

4 Simulation

We simulate the model for 40 seconds: (Note that you can also perform simulation and plotting by using *Simulation Center* from the *Model Editor*)

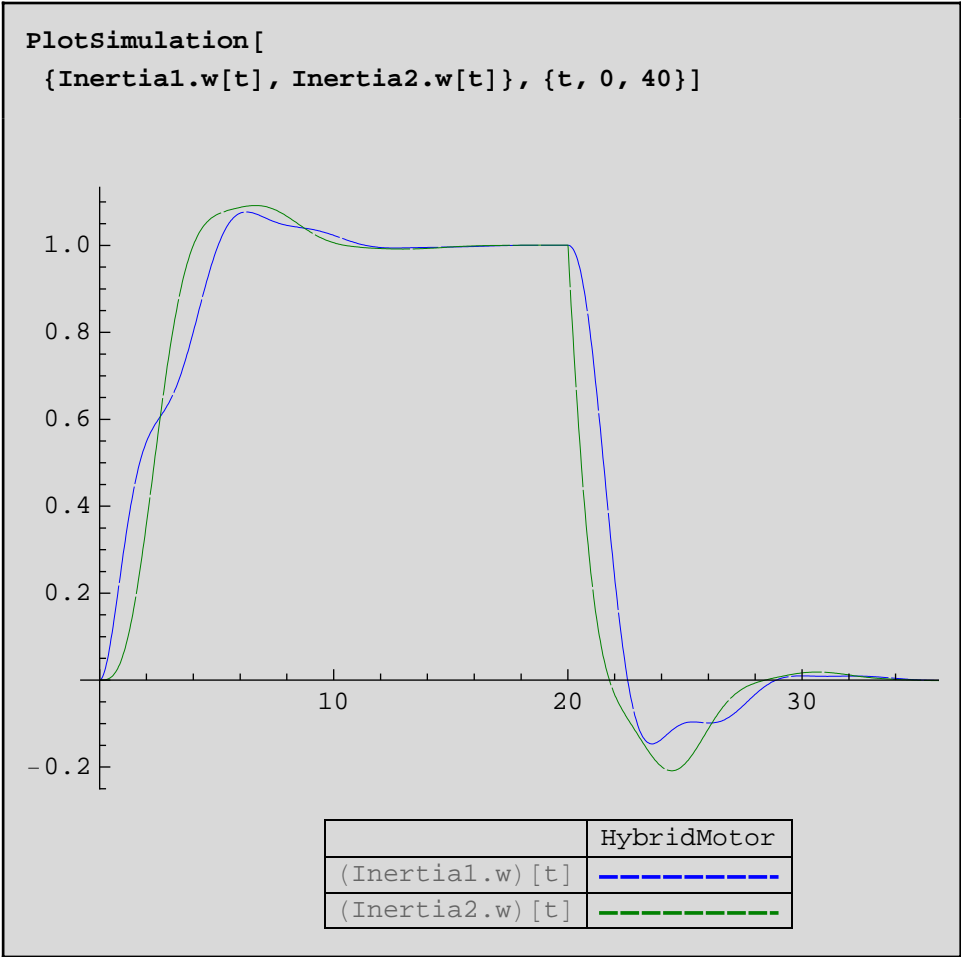
```
simulate[HybridMotor, {t, 0, 40}];
```

The signals are plotted using the `PlotSimulation` command. The constant voltage signal (`ConstantVoltage1.v`) and the angular velocity of the outgoing axis (`Inertia2.w`) are plotted below:

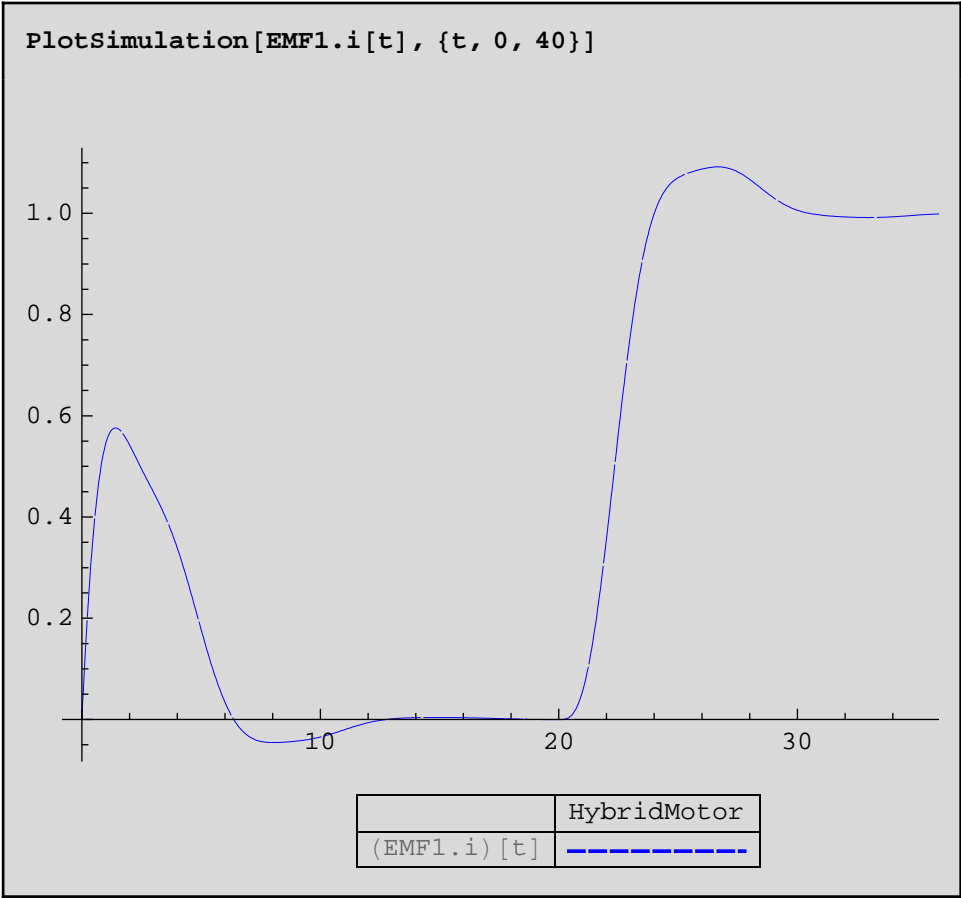


We can see that the outgoing axis spins up to about 1 rad/s. Then at 20 seconds the counteracting torque is applied on the outgoing axis and the axis start to spin down to zero.

We can also plot the two rotational speeds (Inertia1.w and Inertia2.w) in the same plot, showing the weakness of the axis:



Plotting the current in the electrical circuit shows that the current increases to about 1 Ampere when the torque is applied:



We have now illustrated how a model can be used in different ways without the need to change the model structure. The acausality can be useful when you want to know which input signal to use when a certain output signal is wanted.

MathModelica® System Designer Professional

Electric Circuit

© *MathCore Engineering AB*

1 Abstract

This notebook illustrates how a *MathModelica System Designer Professional* model may be developed directly within a *Mathematica* notebook. First a superclass of elements with two electrical pins is modeled. Then an ideal electrical resistor, a capacitor, an inductor, and a sine wave voltage source are modeled, all inheriting the superclass. Before the components are used to connect an entire electrical circuit a ground element is also modeled. Finally the circuit is simulated and several plots from the simulation are presented.

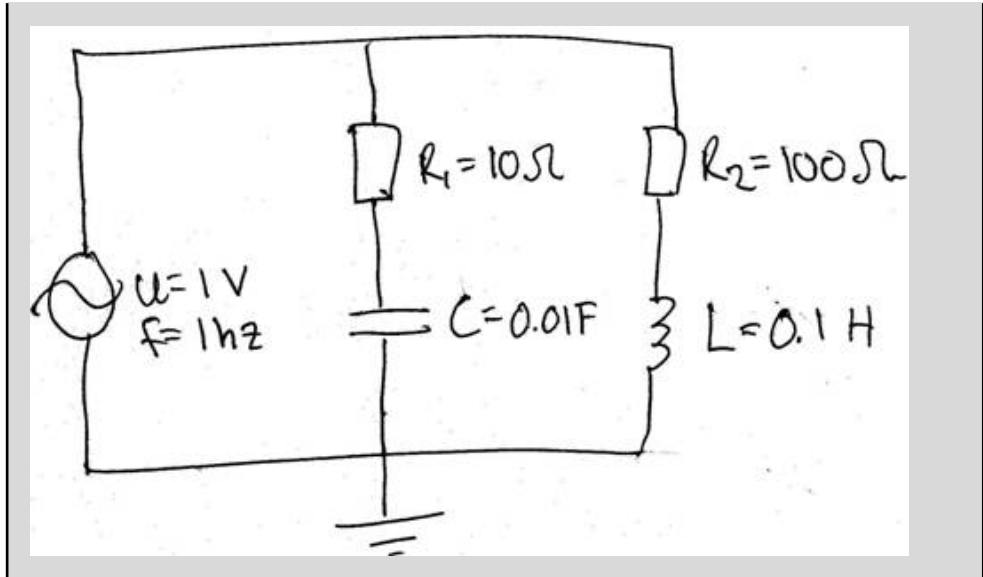
Note that models can also be developed in the model editor using drag-and-drop. See the Introductory Examples Examples of the model editor for more information on how to do this.

2 Initialization

To be able to use *MathModelica System Designer* within *Mathematica* we need to load the *MathModelica* package.

```
Needs["MathModelica`"];
```

Electric Circuit



We will develop the above model from scratch to illustrate how Modelica models can be built. The Introductory Examples document illustrates how this model can be modeled using drag-and-drop with ready-made components.

3.1 Modeling Electrical Components

First a superclass of elements with two electrical pins is modeled. Then an ideal electrical resistor, a capacitor, an inductor, and a sine wave voltage source are modeled, all inheriting the superclass.

3.1.1 Voltage, Current

The Modelica language allows us to define new types by extending already existing types. Here we will define a new type called Voltage, and another called Current. Using these instead of just declaring variables as Reals will make it easier for the user to interpret results.

```
type Voltage = Real(unit="V");
```

```
type Current = Real(unit="A");
```

Pin

Before we begin writing models for the electrical components, we must first identify the appropriate connector for these components. The connector, called Pin, identifies the two quantities associated with a connection point in electrical circuits, namely voltage and current.

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;
```

An important thing to note about this connector is the flow qualifier in front of the current i . The flow qualifier identifies quantities that sum up to zero in a connection point according to kirchhoff's first law. Variables that are declared without this qualifier are set equal at a connection point, i.e. they follow kirchhoffs second law.

3.1.3 TwoPin

Furthermore the electrical components we will define share a few other common properties, namely:

1. They have one positive, and one negative pin
2. The voltage level over the component equals the voltage difference between these pins
3. The current going in and out of the component sums up to zero

Therefore we define a superclass with these common properties and we also add a help variable for the current in order to make it easier to analyze results.

```
model TwoPin "Superclass of elements with two
electrical pins"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v-n.v;
  0 = p.i+n.i;
  i = p.i;
end TwoPin;
```

Resistor

A resistor is a TwoPin that obeys Ohm's law

$$v = R \cdot i$$

This component can be defined by extending the TwoPin superclass and adding the desired equation.

```
model Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real R(unit="ohm") "Resistance";
equation
  R*i = v;
end Resistor;
```

3.1.5 Capacitor

A capacitor can be defined in a similar way as we defined the resistor.

```
model Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real C(unit="F") "Capacitance";
equation
  der(v) = i/C;
end Capacitor;
```

3.1.6 Inductor

An inductor can also be defined.

```
model Inductor "Ideal electrical inductor"
  extends TwoPin;
  parameter Real L(unit="H") "Inductance";
equation
  L*der(i) = v;
end Inductor;
```

VsourceAC

In a similar fashion we define a voltage source with default amplitude 220 V and default frequency 50 Hz.

```
model VsourceAC "Sine-wave voltage source"
  extends TwoPin;
  parameter Voltage VA=220 "Amplitude [V]";
  parameter Real f=50 "Frequency [Hz]";
protected
  constant Real PI=Modelica.Constants.pi;
equation
  v = VA*sin(((2*PI)*f)*time);
end VsourceAC;
```

3.1.8 Ground

Finally all electrical circuits need a ground.

```
model Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;
```

3.2 Modeling the Electrical Circuit

Having defined all components we can now model the circuit. This is done by declaring the components and then connecting them using connect statements.

```
model Circuit
  Resistor resistor1(R=10);
  Capacitor capacitor1(C=0.01);
  Resistor resistor2(R=100);
  Inductor inductor1(L=0.1);
  VsourceAC sineVoltage1;
  Ground ground1;
equation
```

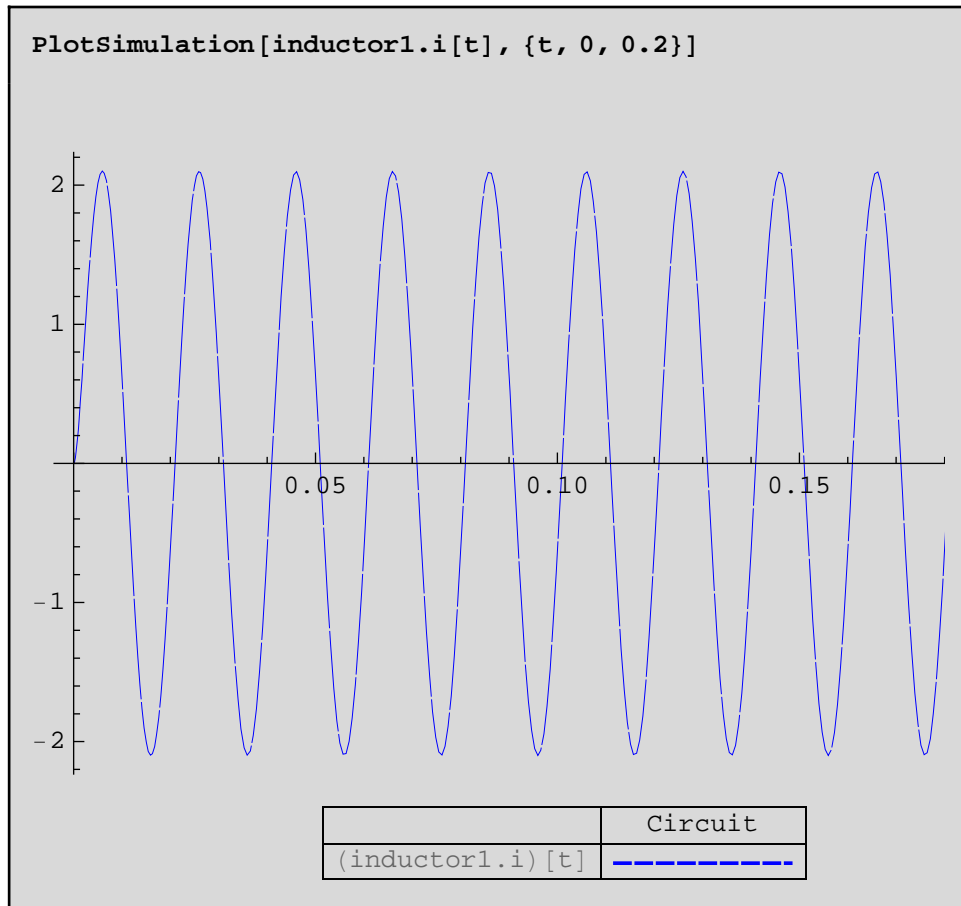
```
connect(sineVoltage1.p, resistor1.p);
connect(resistor1.n, capacitor1.p);
connect(capacitor1.n, sineVoltage1.n);
connect(resistor1.p, resistor2.p);
connect(resistor2.n, inductor1.p);
connect(inductor1.n, capacitor1.n);
connect(sineVoltage1.n, ground1.p);
end Circuit;
```

3.3 Simulating the Circuit

First simulate the model with the default initial values and parameter settings in the range $0 \leq t \leq 1$.

```
simulate[Circuit, {t, 0, 1}];
```

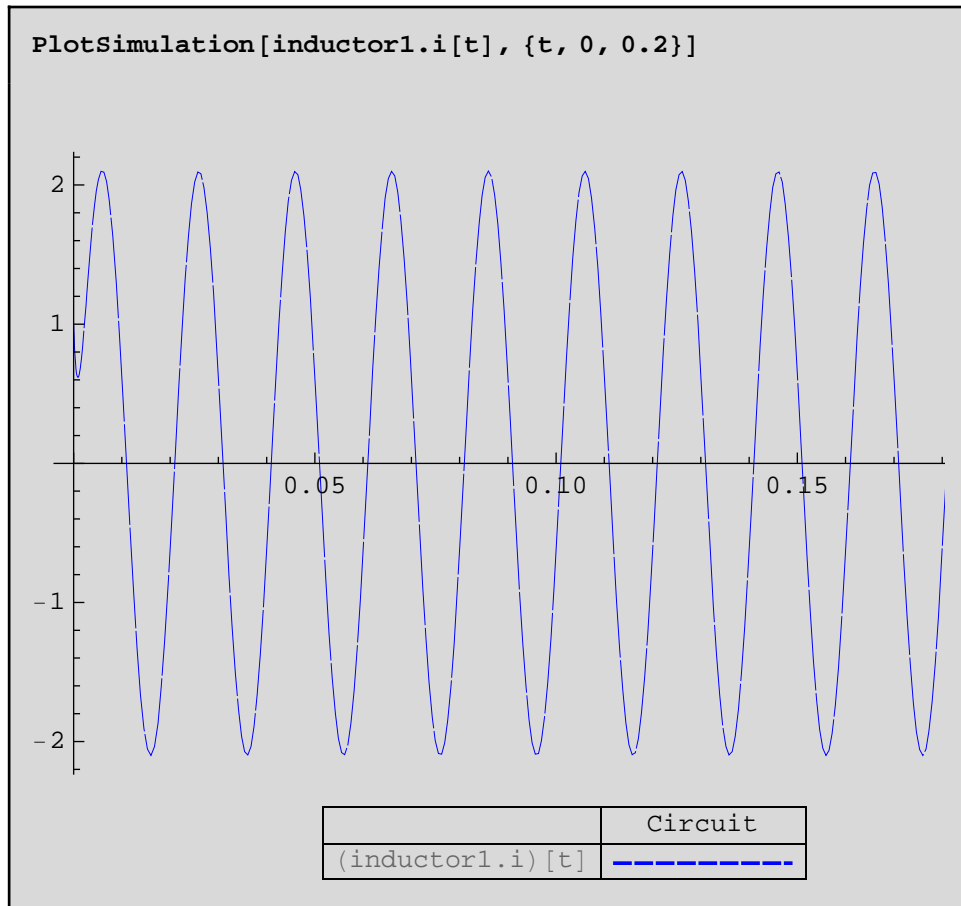
Let us plot the current in the inductor.



The default initial value for $L.i=0$. Setting another initial value is done by giving a list of equal statements for the option `InitialValues` in `Simulate`.

```
Simulate[Circuit, {t, 0, 1},
  InitialValues -> {inductor1.i == 1}];
```

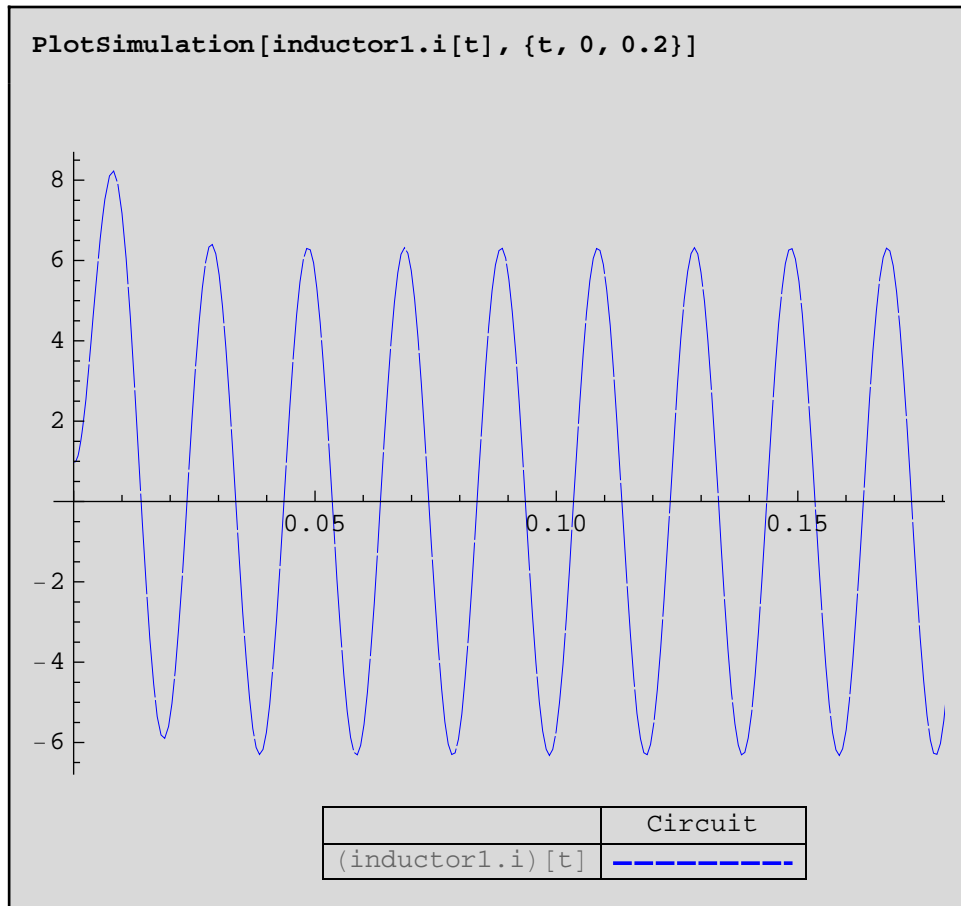
A plot shows the result.



Setting parameter values is done in the same way for the option `ParameterValues`.

```
Simulate[Circuit, {t, 0, 1},
  ParameterValues -> {resistor1.R == 15, resistor2.R == 15},
  InitialValues -> {inductor1.i == 1}];
```

Another plot shows the new result.



3.4 Comparing results

As mentioned in the example above the electrical circuit can be modeled with simple drag-and-drop using the model editor. Actually the model is available in the Introductory Examples package. We can print its definition by using the `GetDefinition` command.

```
GetDefinition[IntroductoryExamples.ComponentBased.  
ElectricCircuit, ShowAnnotations → False]
```

```
model ElectricCircuit  
  Modelica.Electrical.Analog.Sources.SineVoltage  
  sineVoltage1;  
  Modelica.Electrical.Analog.Basic.Ground ground1;  
  Modelica.Electrical.Analog.Basic.Resistor  
  resistor1(R=10);  
  Modelica.Electrical.Analog.Basic.Capacitor  
  capacitor1(C=0.01);  
  Modelica.Electrical.Analog.Basic.Inductor  
  inductor1(L=0.1);  
  Modelica.Electrical.Analog.Basic.Resistor  
  resistor2(R=100);  
equation  
  connect(sineVoltage1.n,ground1.p);  
  connect(resistor1.n,capacitor1.p);  
  connect(inductor1.n,ground1.p);  
  connect(resistor2.n,inductor1.p);  
  connect(capacitor1.n,ground1.p);  
  connect(sineVoltage1.p,resistor1.p);  
  connect(resistor2.p,resistor1.p);  
end ElectricCircuit;
```

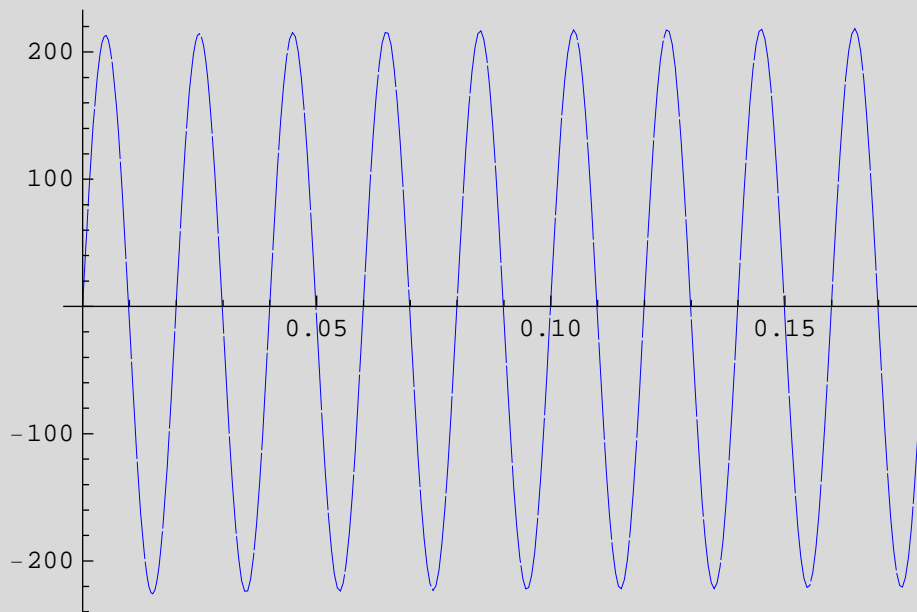
As seen the model also contains some html code used for documentation. The actual model is seen in the last lines and is similar to the model that we have created. The difference is that standard Modelica components are used instead of the components defined by us, and annotations are also added. These annotations contain the graphical information used and created by the model editor.

We can compare the results between the two models by storing simulation results in separate variables. Note that the default values for the voltage source differ between the models, and therefore we modify these in the simulation command, adding modifications for the parameter values.

```
nbCircuit = Simulate[Circuit, {t, 0, 1}];  
meCircuit = Simulate[IntroductoryExamples.  
  ComponentBased.ElectricCircuit, {t, 0, 1},  
  ParameterValues → {sineVoltage1.V == 220,  
    sineVoltage1.freqHz == 50}];
```

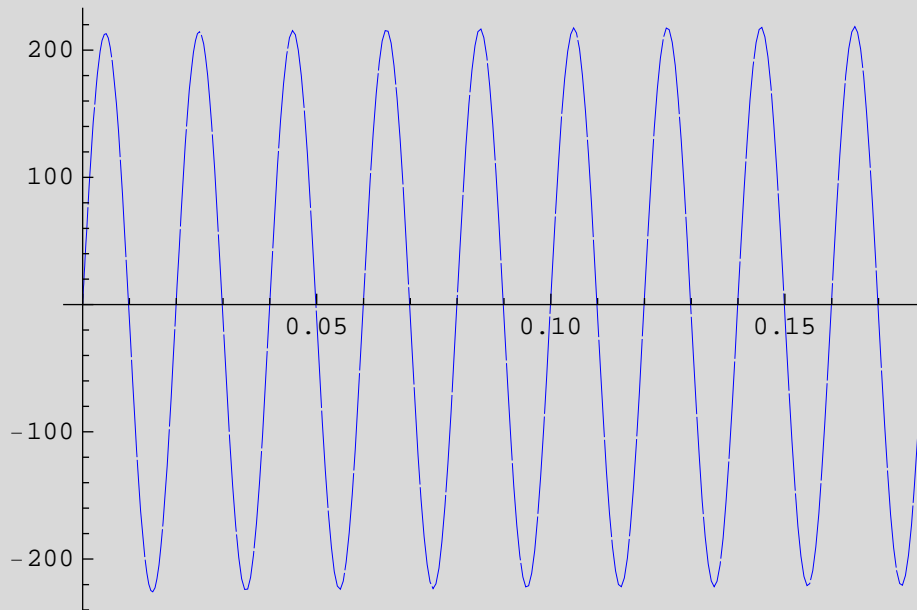
The plots below show that the results are identical just as expected.

```
PlotSimulation[resistor1.v[t],  
  {t, 0, 0.2}, SimulationResult → nbCircuit]
```



	Circuit
(resistor1.v) [t]	-----

```
PlotSimulation[resistor1.v[t],  
{t, 0, 0.2}, SimulationResult -> meCircuit]
```



IntroductoryExamples.ComponentBas
ElectricCircuit

(resistor1.v) [t]

MathModelica® System Designer Professional

MyPackage - a Modelica package

© *MathCore Engineering AB*

1 Abstract

This notebook intends to show some of the basic concepts of making packages in *MathModelica* and using `within` statements.

The most important thing to remember is that all cells which are intended to be in a package must be of `ModelicaInput` type. This cell type is included in the *MathModelica* style sheet. A Modelica cell can also be created using the *MathModelica* palette.

The `within` statement must be present in every class cell. By using `within` statements in this way it is possible to evaluate one cell at a time and make sure that the right scope is set.

2 Initialization

To be able to use *MathModelica System Designer* within *Mathematica* we need to load the *MathModelica* package.

```
Needs ["MathModelica`"] ;
```

Create Packages

3.1 MyPackage

We begin by defining the package structure.

```
package MyPackage
  package MySubPackage
  end MySubPackage;
  package SIunits = Modelica.SIunits;
end MyPackage;
```

3.2 MyPackage.MyModel_1

MyModel_1 is defined here. Note the initial `within` statement, stating that this model is within MyPackage. The same goes for all classes in this package. This makes it possible to evaluate a cell and still make sure that the right scope is set.

```
within MyPackage;
model MyModel_1
  Real x(start=1);
equation
  der(x) = -x;
end MyModel_1;
```

3.3 MyPackage.MyModel_2

MyModel_2 is defined here. Note the initial `within` statement, stating that this model is within MyPackage.


```
within MyPackage;
model MyModel_2
  Real y(start = 2);
equation
  der(y) = -y;
end MyModel_2;
```

3.4 MyPackage.MyModel_3

MyModel_3 is defined here. Note the initial within statement, stating that this model is within MyPackage.

```
within MyPackage;
model MyModel_3
  Real y(start = 3);
equation
  der(y) = -y;
end MyModel_3;
```

3.5 MyPackage.MyModel_4

MyModel_4 is defined here. Note the initial within statement, stating that this model is within MyPackage.

```
within MyPackage;
model MyModel_4
  Real y(start = 4);
equation
  der(y) = -y;
end MyModel_4;
```

3.6 MyPackage.MySubPackage.MySubModel_1

MySubModel_1 is defined here. Note the initial within statement, stating that this model is within MyPackage.MySubPackage.

```
within MyPackage.MySubPackage;  
model MySubModel_1  
  Real z(start = 3);  
  equation  
    der(z) = -z + 1;  
end MySubModel_1;
```

4 Using the Package

We can look at the definition of MyPackage using the GetDefinition command. This will show the entire package.

```
GetDefinition[MyPackage]
```

```
package MyPackage
  package MySubPackage
    model MySubModel_1
      Real z(start=3);
      equation
        der(z)=(-z) + 1;
      end MySubModel_1;
    end MySubPackage;

  package SIunits= Modelica.SIunits;
  model MyModel_1
    Real x(start=1);
    equation
      der(x)=-x;
    end MyModel_1;

  model MyModel_2
    Real y(start=2);
    equation
      der(y)=-y;
    end MyModel_2;

  model MyModel_3
    Real y(start=3);
    equation
      der(y)=-y;
    end MyModel_3;

  model MyModel_4
    Real y(start=4);
    equation
      der(y)=-y;
    end MyModel_4;

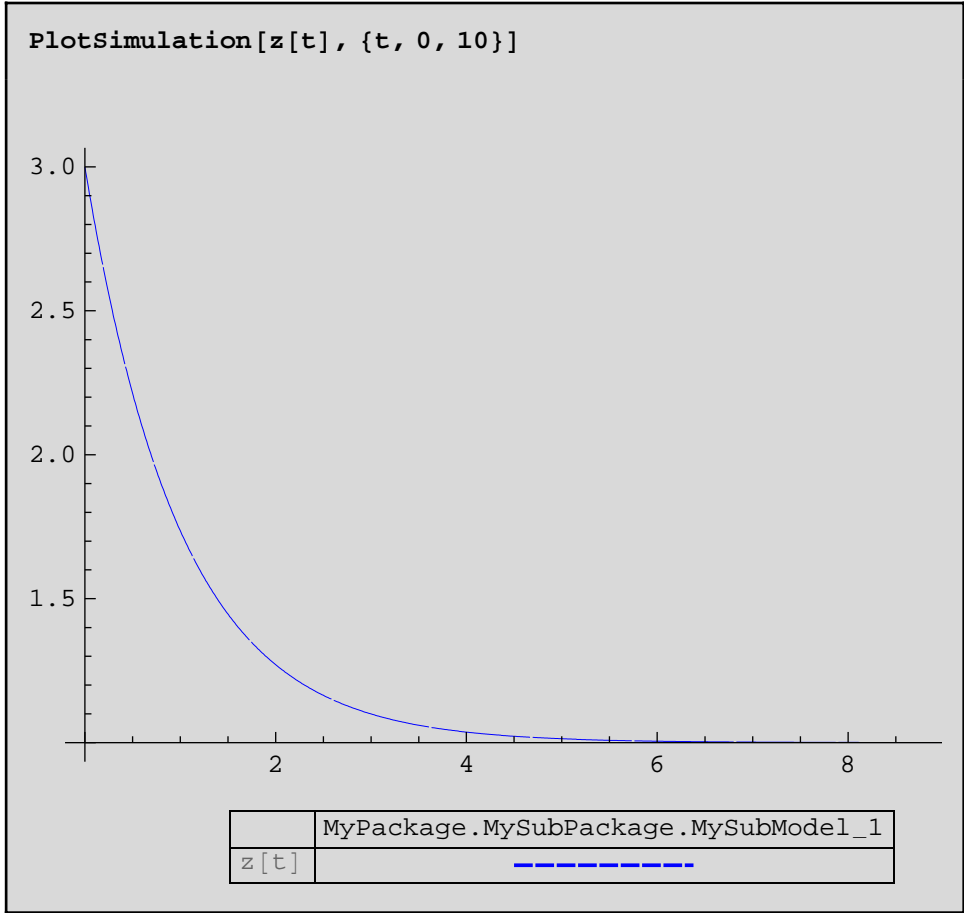
end MyPackage;
```

We can also simulate any model within the package and plot the result.

```
Simulate [MyPackage.MySubPackage.MySubModel_1, {t, 0, 10}]

<SimulationData:
  "MyPackage.MySubPackage.MySubModel_1" : : {0., 10.}
  : 1007 data points : 0 events : 2 variables>
{z', z}
```

The SimulationData object returned contains the variables z and \dot{z} .



MathModelica® System Designer Professional

A Sample Optimization Problem

© MathCore Engineering AB

1 Abstract

This notebook is an example of how the powerful scripting language of *Mathematica* can be utilized to solve non-trivial optimization problems that contain dynamic simulations. First we will define a Modelica model of a linear actuator with spring-damped stopping followed by a first order system. Using *MathModelica System Designer* scripting we will then find a damping for the translational spring-damper such that the step response is as "close" as possible to the step response from the first order system.

2 Initialization

2.1 Loading packages

To be able to use *MathModelica System Designer* within *Mathematica* we need to load the *MathModelica* package.

```
Needs["MathModelica`"]
```

Example

Consider the following model of a linear actuator with spring-damped stopping:

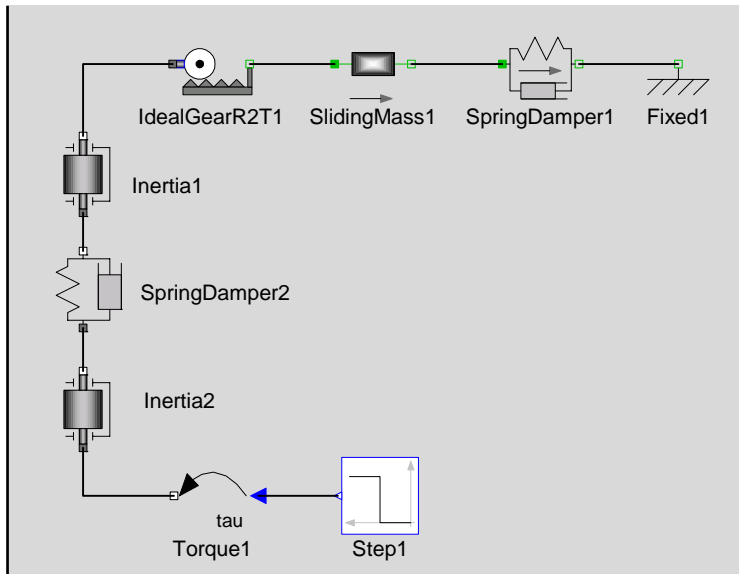


Diagram : LinearActuator

Below is the corresponding model code:

```
model LinearActuator
  Modelica.Blocks.Sources.Step step1;
  Modelica.Mechanics.Rotational.Torque torque1;
  Modelica.Mechanics.Rotational.Inertia inertial(J=0.1);
  Modelica.Mechanics.Rotational.Inertia inertia2(J=0.1);
  Modelica.Mechanics.Rotational.SpringDamper springDamper2(c=15,d=2);
  Modelica.Mechanics.Rotational.IdealGearR2T idealGearR2T1;
  Modelica.Mechanics.Translational.SlidingMass slidingMass1(m=0.5);
  Modelica.Mechanics.Translational.SpringDamper springDamper1(c=20,d=3);
  Modelica.Mechanics.Translational.Fixed fixed1;

equation
  connect(springDamper1.flange_b,fixed1.flange_b);
```

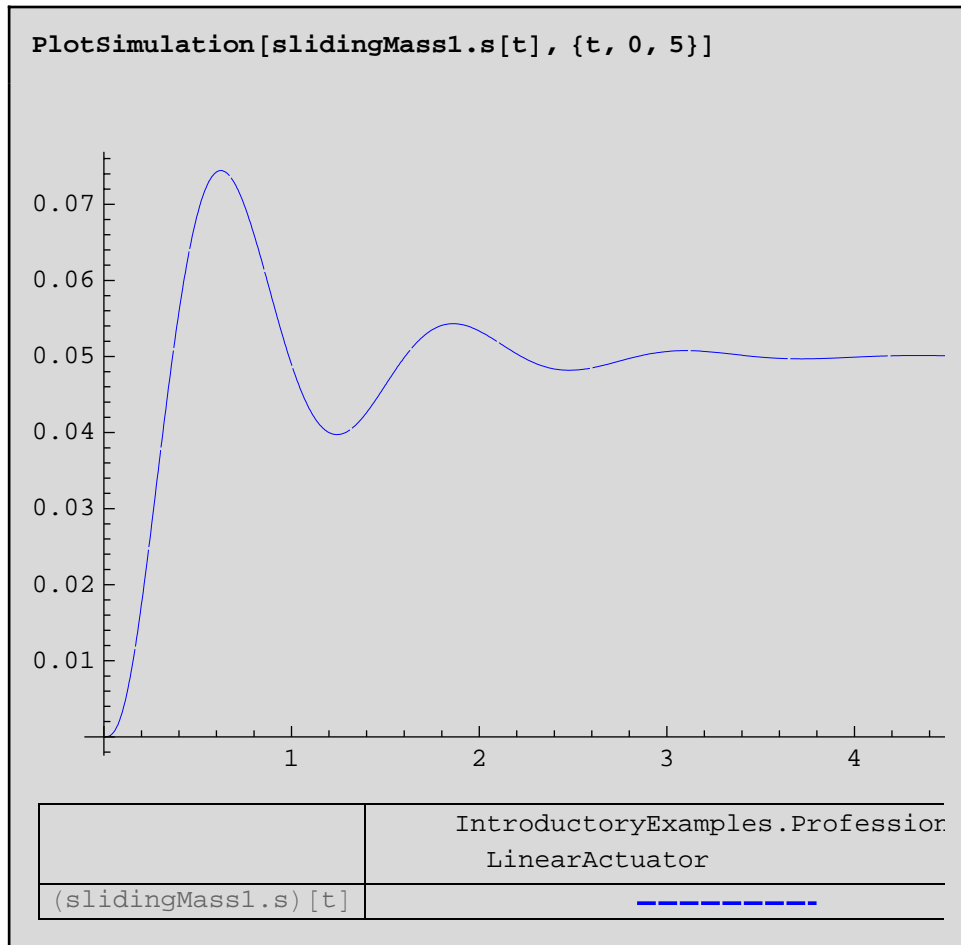
```
connect(slidingMass1.flange_b, springDamper1.flange_a);
connect(idealGearR2T1.flange_b, slidingMass1.flange_a);
connect(inertia2.flange_b, idealGearR2T1.flange_a);
connect(springDamper2.flange_b, inertia2.flange_a);
connect(inertia1.flange_b, springDamper2.flange_a);
connect(torque1.flange_b, inertia1.flange_a);
connect(step1.y, torque1.tau);
end LinearActuator;
```

This model is available from the `IntroductoryExamples.Professional` package delivered with *MathModelica System Designer* and we will use it in the rest of this example.

3.1 Optimization

The model can be simulated directly in *Mathematica* using the `Simulate` command of *MathModelica System Designer Professional*. We simulate a step response and store the result in the variable `res0`:

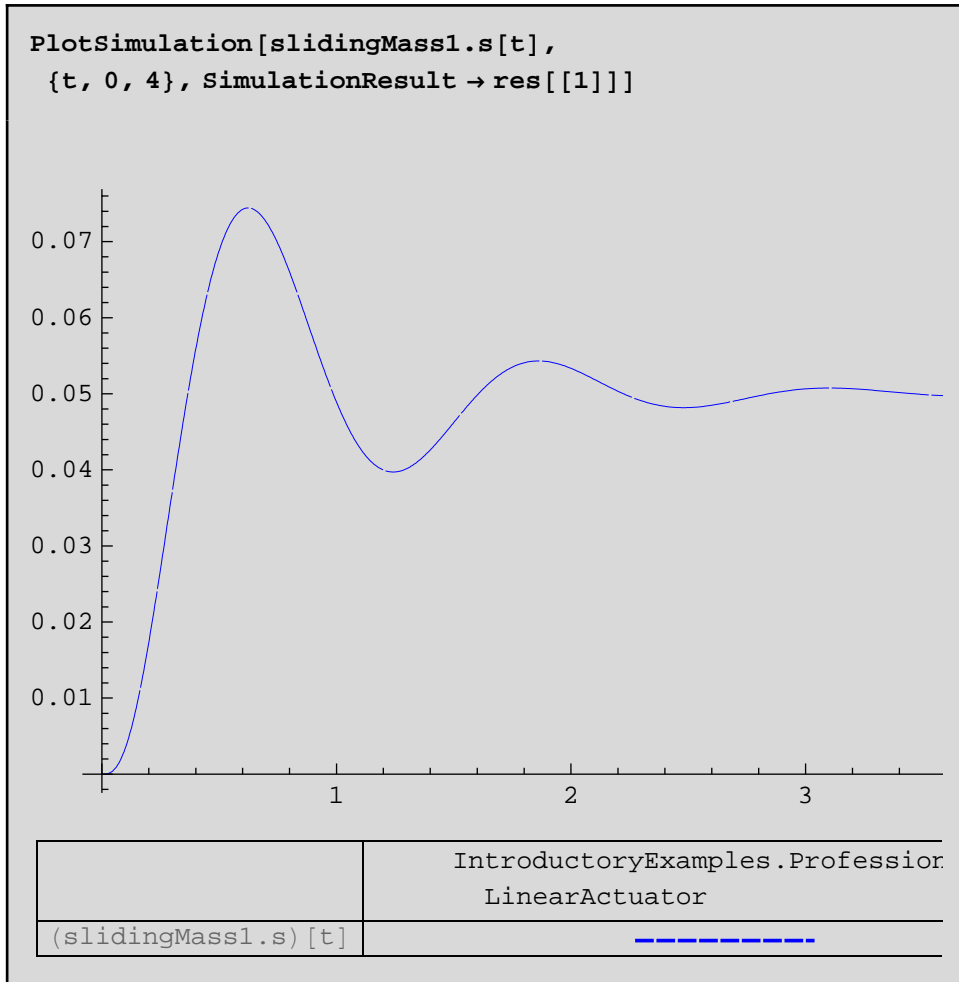
```
Simulate[
  IntroductoryExamples.Professional.LinearActuator,
  {t, 0, 5}, ParameterValues -> {springDamper1.d == 2}];
```

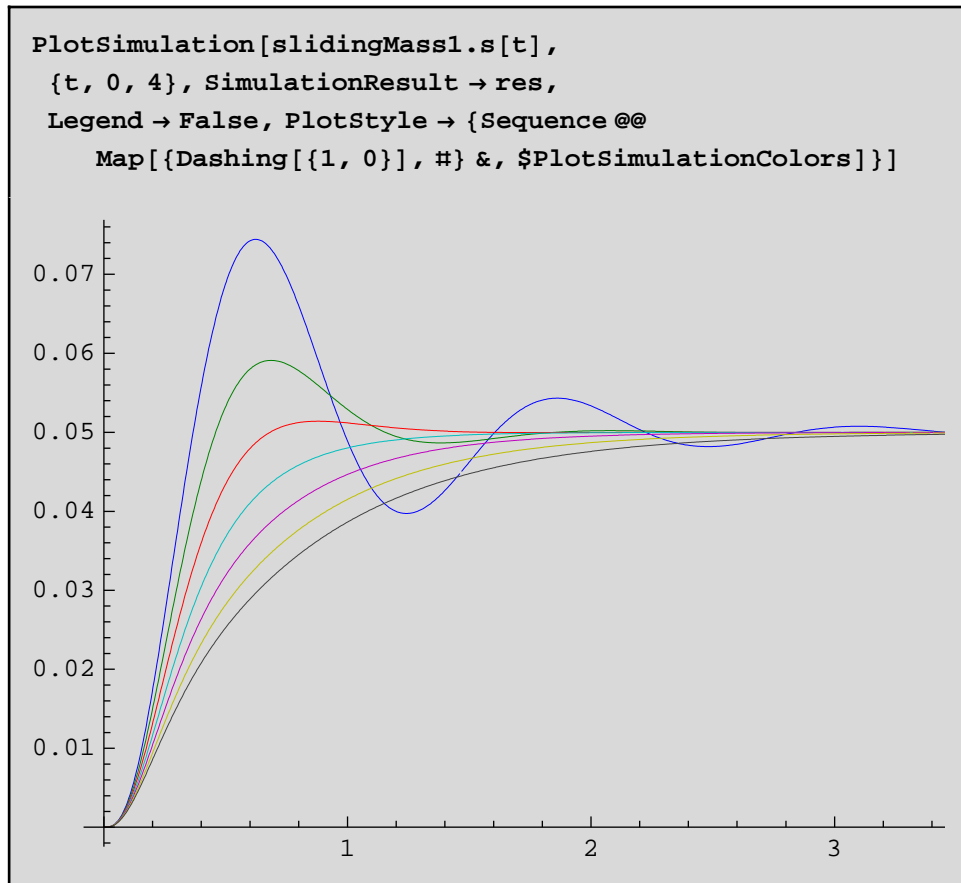
Suppose that we have some freedom in choosing the damping in the translational spring-damper. A number of simulation runs show what kind of behavior we have for different values of the damping d . Table [] is used in Simulate[] to make simulations for damping 2 to 14, with a step of 2, i.e. seven simulations are performed.

```
res = Table[Simulate[IntroductoryExamples.Professional.  
LinearActuator, {t, 0, 4}, ParameterValues →  
{springDamper1.d == q}], {q, 2, 14, 2}];
```

The simulation plot shows that the first simulation result from this batch is the same as the result we obtained previously, which is to be expected as all parameters are the same for these two simulations.



It is more interesting to show how the behavior differs depending on the damping.

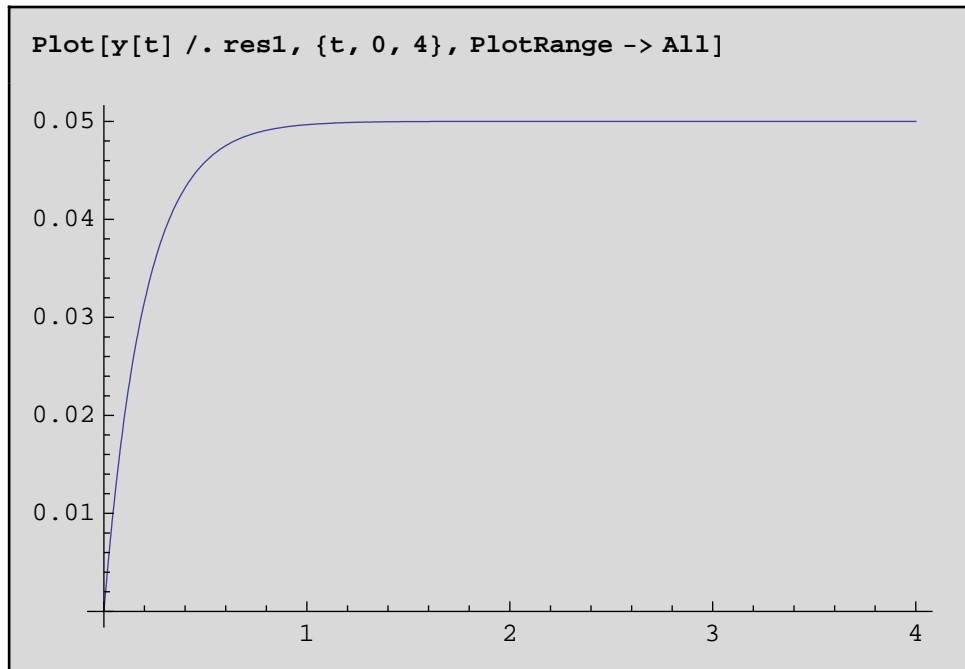


Now we want to get an optimal design for our system. We make sure that the variable y is not present in the global context in *Mathematica*:

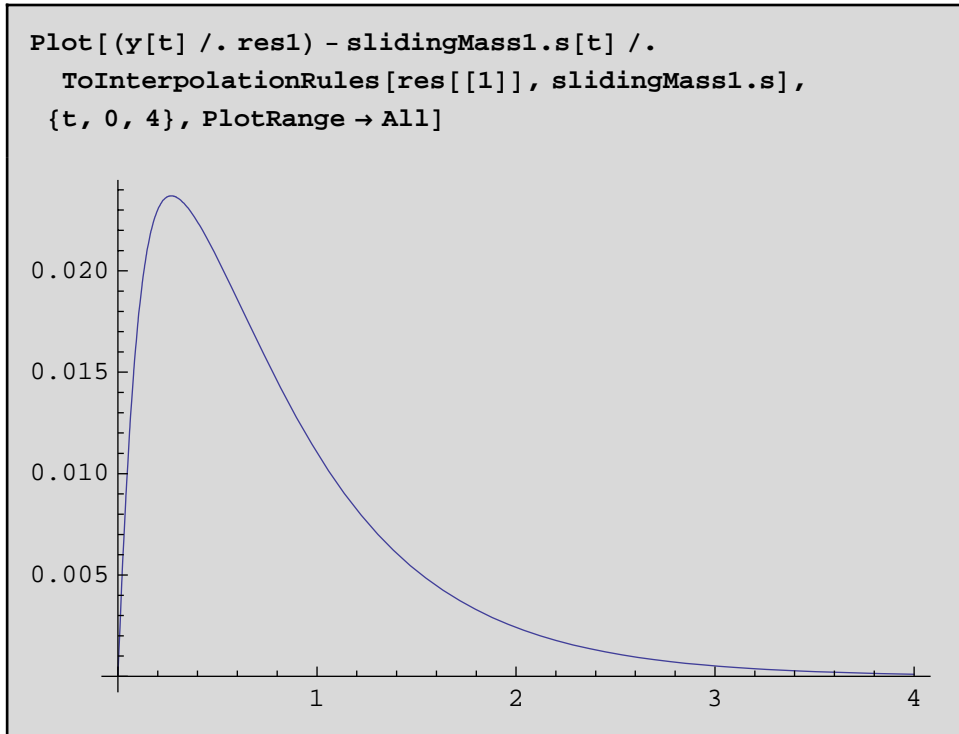
```
Clear[y]
```

Now assume that we would like to choose the damping d so that the resulting system behaves as closely as possible to the following first order system response:

```
res1 = NDSolve[
 {0.2 y'[t] + y[t] == 0.05, y[0] == 0}, {y}, {t, 0, 4}];
```



More interesting perhaps is to look at the difference between the desired signal and the step response.



Now, let us make things a little bit more automatic. Simulate and compute the integral of the square error from $t = 0$ to $t = 4$. We set the number of intervals to 1000 in order to get good enough sample rate for NIntegrate.

```
res = Simulate[IntroductoryExamples.Professional.
  LinearActuator, {t, 0, 4}, NumberOfIntervals -> 1000,
  ParameterValues -> {springDamper1.d == 3}];
```

```
NIntegrate[
  First[(y[t] /. res1) - slidingMass1.s[t]]2, {t, 0, 4}]

0.00016241
```

We define a function, $f(a)$, doing the same thing as above, but for different spring-damper parameters $d = a$,

```
f[a_] := Module[{res, t},
  res = Simulate[IntroductoryExamples.Professional.
    LinearActuator, {t, 0, 4}, NumberOfIntervals → 1000,
    ParameterValues → {springDamper1.d == a}];
  NIntegrate[First[(y[t] /. res1) - slidingMass1.s[t]]2,
    {t, 0, 4}]]
```

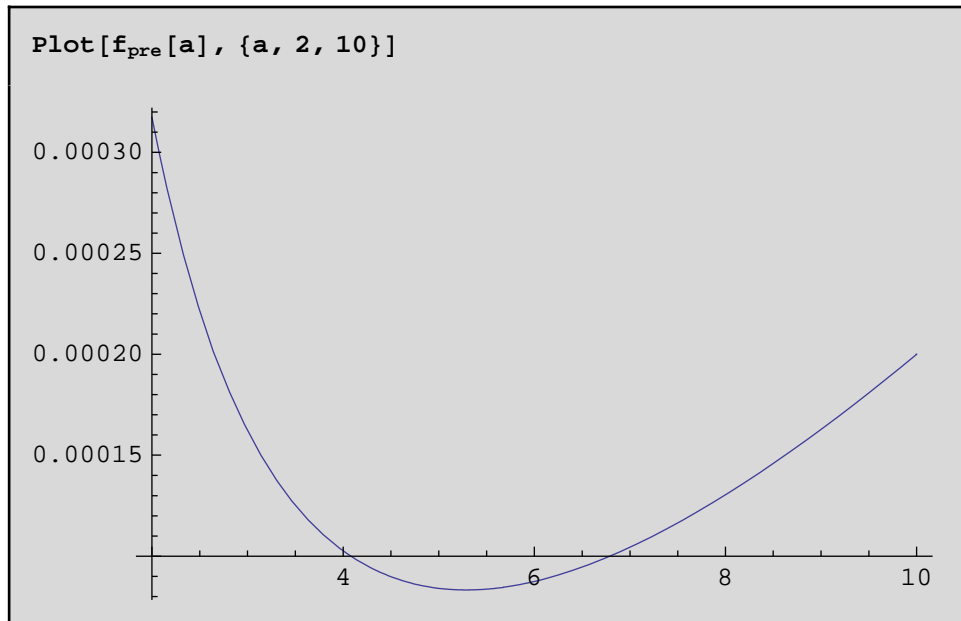
and tabulate several results for $2 \leq d = a \leq 10$.

```
Table[{a, f[a]}, {a, 2, 10, .5}]

{{2., 0.000317292}, {2.5, 0.000221338},
 {3., 0.00016241}, {3.5, 0.000125242},
 {4., 0.00010268}, {4.5, 0.0000898597},
 {5., 0.0000840732}, {5.5, 0.0000836745},
 {6., 0.0000874683}, {6.5, 0.0000945918},
 {7., 0.00010442}, {7.5, 0.000116484}, {8., 0.00013042},
 {8.5, 0.000145949}, {9., 0.000162822},
 {9.5, 0.000180893}, {10., 0.000200016}}
```

The tabulated values are interpolated using an interpolating function object. The default interpolation order is 3.

```
fpre = Interpolation[%];
```



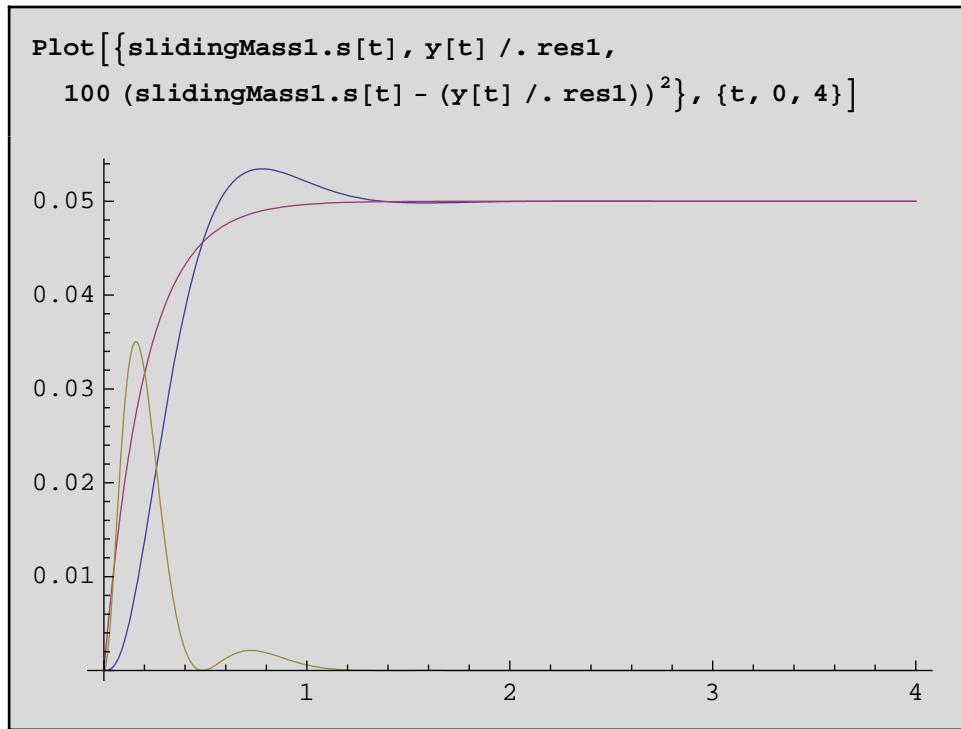
The minimum value of a can be computed using FindMinimum:

```
FindMinimum[f_pre[s], {s, 5}, PrecisionGoal -> 4]
{0.0000832623, {s -> 5.28674}}
```

Now let us simulate with the optimal parameter value

```
Simulate[IntroductoryExamples.
  Professional.LinearActuator, {t, 0, 4},
  ParameterValues -> {springDamper1.d == 5.28732}];
```

A plot comparing the first order system and linear actuator model response together with a plot of the squared error amplified with a factor 100.



MathModelica® System Designer Professional

Frequency Analysis of Simulation Data

© *MathCore Engineering AB*

1 Abstract

This notebook is an example of how *Mathematica* can be utilized for frequency analysis. First we will define a Modelica model of a weak axis, similar to what is described in the introductory examples of the model editor. Next we will develop a simple *Mathematica* program to perform fourier analysis.

2 Initialization

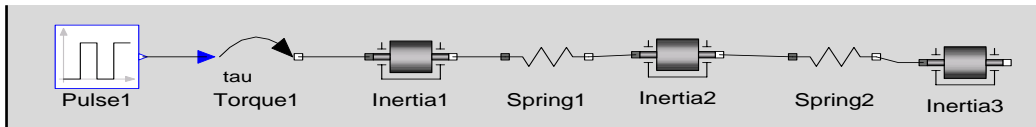
We begin by loading *MathModelica system Designer*. This is done by evaluating the following command:

```
Needs["MathModelica`"];
```

3 Frequency Analysis

The aim of this example is to take a model and show how *Mathematica* can be used to analyze simulation results. In this case we will do a frequency analysis on a weak axis model.

The Model



Consider a weak axis excited by a torque pulse train. The axis is modeled by three segments joined by two torsion springs. The model can easily be created in the model editor (see the Introductory Examples documentation, example 3, for a detailed explanation on how to do this). The model code is given below:

```

model WeakAxis
  Modelica.Mechanics.Rotational.Inertia
  inertia3;
  Modelica.Mechanics.Rotational.Inertia
  inertial1;
  Modelica.Blocks.Sources.Pulse
  pulse1(width=1, period=200);
  Modelica.Mechanics.Rotational.Inertia
  inertia2;
  Modelica.Mechanics.Rotational.Spring
  spring1(c=0.7);
  Modelica.Mechanics.Rotational.Torque
  torque1;
  Modelica.Mechanics.Rotational.Spring
  spring2(c=1);
equation
  connect(inertia2.flange_b, spring2.flange_a);
  connect(spring1.flange_b, inertia2.flange_a);
  connect(torque1.flange_b, inertial1.flange_a);
  connect(spring2.flange_b, inertia3.flange_a);
  connect(pulse1.y, torque1.tau);
  connect(inertial1.flange_b, spring1.flange_a);
end WeakAxis;

```

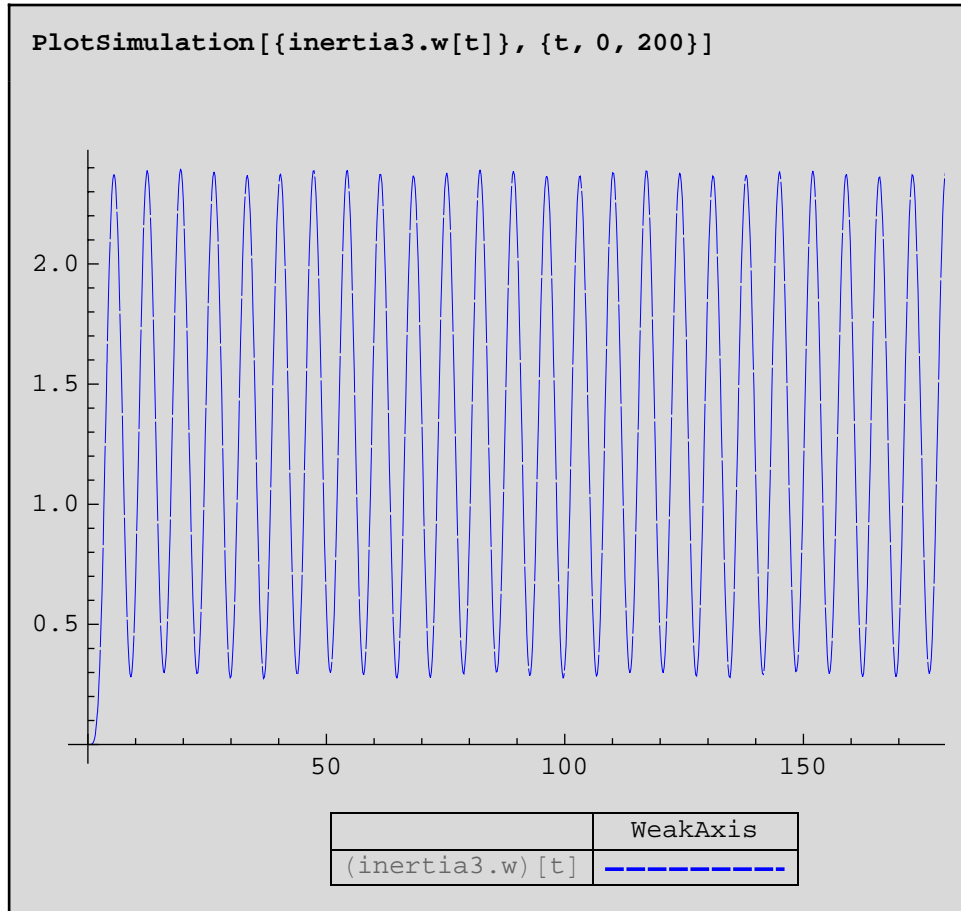
3.2 Simulation

When the model has been created and evaluated it can be simulated. We simulate the model for 200 seconds using the simulate command.

```
Simulate[WeakAxis, {t, 0, 200}]
```

```
<SimulationData: "WeakAxis" : : {0., 200.}
  : 1052 data points : 2 events : 74 variables>
{inertial.a, inertial.a_start, inertial.φ', inertial.w', in
  nertial.flange_a.φ, inertial.flange_a.τ, inertial.fl
  ange_b.φ, inertial.flange_b.τ, inertial.initType, in
  nertial.J, inertial.φ, inertial.phi_start, inertial.s
  tateSelection, inertial.w, inertial.w_start, inertia2
  ia2.flange_a.φ, inertia2.flange_a.τ, inertia2.flange
  _b.φ, inertia2.flange_b.τ, inertia2.initType, inert
  ia2.J, inertia2.φ, inertia2.phi_start, inertia2.state
  Selection, inertia2.w, inertia2.w_start, inertia3.a, i
  nertia3.a_start, inertia3.φ', inertia3.w', inertia3.fl
  ange_a.φ, inertia3.flange_a.τ, inertia3.flange_b.φ,
  nertia3.flange_b.τ, inertia3.initType, inertia3.J, in
  ertia3.φ, inertia3.phi_start, inertia3.stateSelector
  lsel.offset, pulsel.period, pulsel.startTime, pulsel.
  0, pulsel.T_width, pulsel.width, pulsel.y, spring1.c,
  pring1.flange_a.φ, spring1.flange_a.τ, spring1.flang
  e_b.φ, spring1.flange_b.τ, spring1.phi_rel, spring1.
  hi_rel0, spring1.τ, spring2.c, spring2.flange_a.φ, s
  pring2.flange_a.τ, spring2.flange_b.φ, spring2.flang
  e_b.τ, spring2.phi_rel, spring2.phi_rel0, spring2.τ,
  orquel.bearing.φ, torquel.bearing.τ, torquel.flange_k
```

A SimulationData object is returned. This object contains all model parameters and variables, and the results can be plotted using the PlotSimulation command. In this case we plot the angular velocity of the right-most axis segment, inertia3.w.



3.3 Analysis

We will use the built-in Fourier function to perform a frequency analysis. As this function takes a list rather than an interpolating function as argument, we sample `inertia3.w` with a sample frequency 4Hz and store the result in a variable called `data1`.

```
data1 = Table[inertia3.w[t], {t, 0, 200, .25}];
```

Then we remove the mean from `data1`, storing the result in `data2`.

```
mean = Apply[Plus, data1] / Length[data1];
```

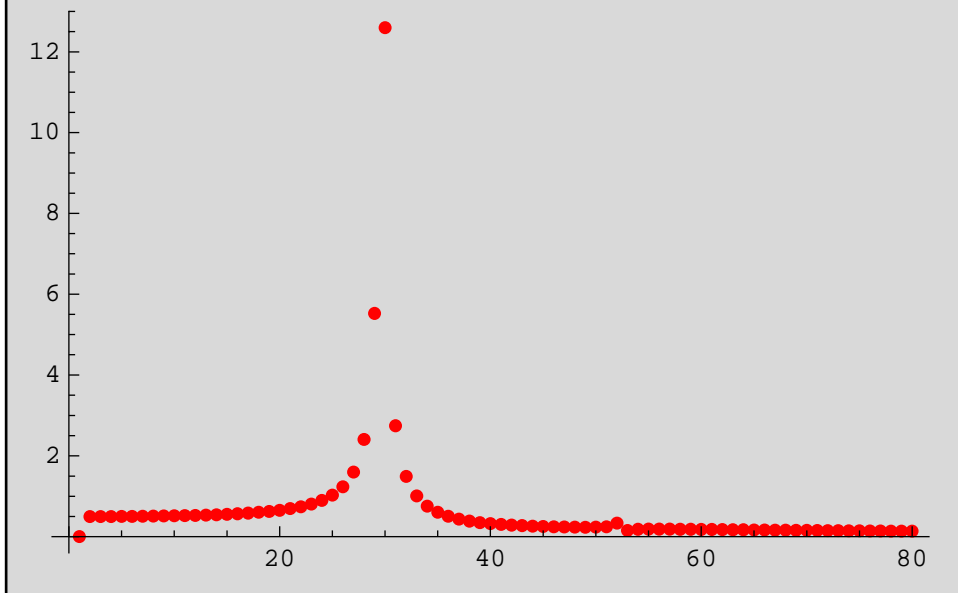
```
data2 = data1 - mean;
```

With this we can compute the absolute values of the discrete Fourier transform.

```
fdata1 = Abs[Fourier[data2]];
```

We can then plot the 80 first points of data.

```
ListPlot[fdata1[[Range[80]]], PlotRange -> All,  
PlotStyle -> {Red, PointSize[0.015]]}
```



The result shows two clear peaks at data point 30 and 52 respectively.

```
{fdata1[[30]], fdata1[[52]]}
{12.5953, 0.334874}
```

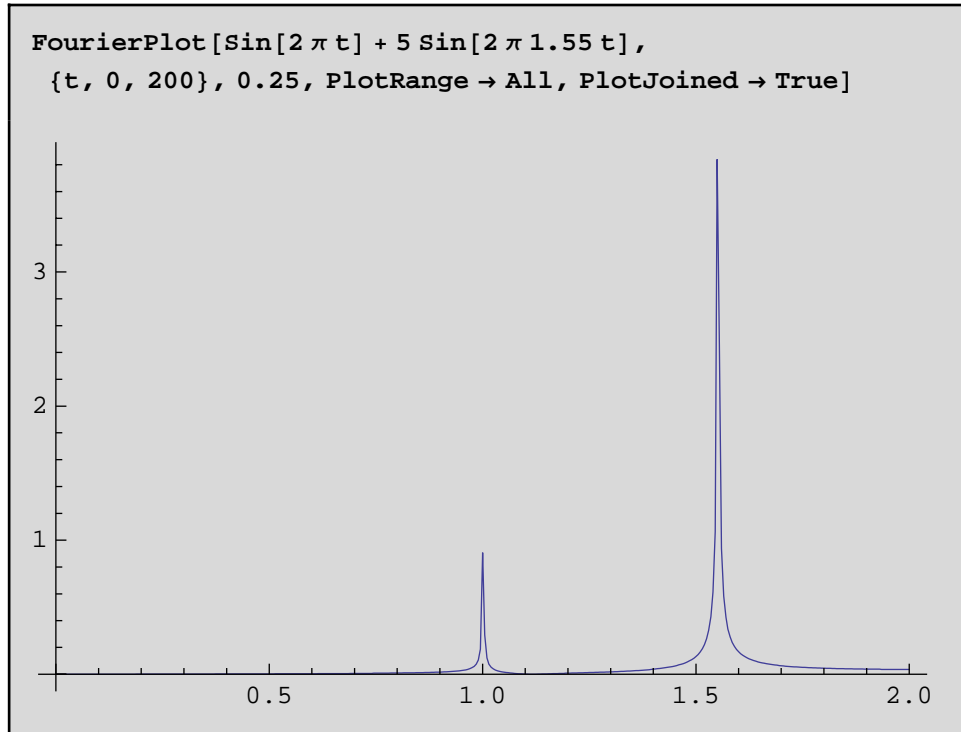
3.4 Defining a Fourier plot function

To make it easier to do the same analysis on other models we will define a *Mathematica* function for this. As the obtained result showed the peaks for respective data points rather than frequency, we will also define the function, `FourierPlot`, as it scales the axes such that amplitude of trigonometric components are plotted against frequency (Hz).

Now we define a function that takes a simulation variable as input, samples the signal from `tmin` to `tmax` with the sampling frequency $\frac{1}{T}$, and plots the result as a function of frequency.

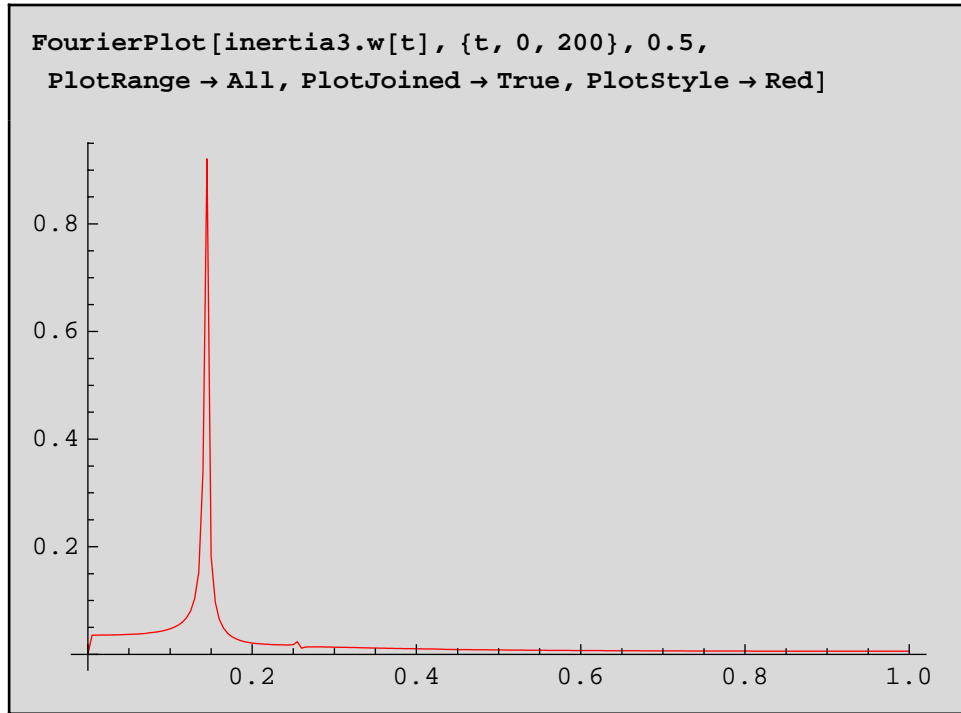
```
FourierPlot[signal_,
  {t_, tmin_, tmax_}, T_, options___] :=
Module[{data1, n, mean, data2, fdata1, fdata2, f},
  data1 = Table[signal, {t, tmin, tmax, T}];
  n = Length[data1];
  mean = Apply[Plus, data1] / n;
  data2 = data1 - mean;
  fdata1 = 2 / Sqrt[n] Abs[Fourier[data2]];
  fdata2 = Drop[fdata1, -Round[n / 2.]];
  f = Range[0, 1 / (2 * T), (1 / (2 T) - 0) / Round[n / 2.]];
  ListPlot[Transpose[{f, fdata2}], options]
]
```

We make a simple example to verify the function.



3.5 Using FourierPlot

We take the same variable as earlier, namely `inertia3.w`, and sample it with a sample frequency of $\frac{1}{0.5}$ Hz to be sure that we fulfill the sampling theorem, and use the newly defined function to plot the result.



We see that the frequency tops are at 0.14 Hz and 0.26 Hz respectively.

For this simple example it can actually be shown that the frequencies of the eigenmodes of the system are given by the imaginary parts of the eigenvalues of the following matrix (c_1 and c_2 are the spring constants).

$$\frac{1}{2\pi} \text{Eigenvalues} \left[\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -c_1 & 0 & -c_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -c_1 & 0 & -c_1 - c_2 & 0 & -c_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -c_2 & 0 & -c_2 & 0 \end{pmatrix} / \cdot \{c_1 \rightarrow 0.7, c_2 \rightarrow 1\} \right] // \text{Chop}$$

$$\{0.256077 i, -0.256077 i, 0.143344 i, -0.143344 i, 0, 0\}$$

Which fits very well with the peaks in the diagram above.

MathModelica® System Designer Professional

Systems with Events

© MathCore Engineering AB

1 Abstract

This notebook illustrates how events can be handled using When and If statements.

2 Initialization

To be able to use *MathModelica System Designer* within *Mathematica* we need to load the *MathModelica* package.

```
Needs["MathModelica`"];
```

3 Bouncing Ball

A simple example of event handling is the bouncing ball. We define a model where the ball is dropped from the height h at a velocity v . The ball bounces whenever the ball center is less than zero.

```
model BouncingBall
  parameter Real e=0.7 "coefficient of
```

```

restitution";
  parameter Real g=9.81 "gravity
acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball
is flying";
  Boolean impact;
  Real v_new;
  discrete Integer n_bounce(start=0);
equation
  impact = h <= 0.0;
  der(v) = if flying then -g else 0;
  der(h) = v;

  when {h <= 0.0 and v <= 0.0, impact} then
    v_new = if edge(impact) then -e*pre(v)
else 0;
    flying = v_new > 0;
    reinit(v, v_new);
    n_bounce=pre(n_bounce)+1;
  end when;

end BouncingBall;

```

Simulate the system. The simulation data is returned as a SimulationData object.

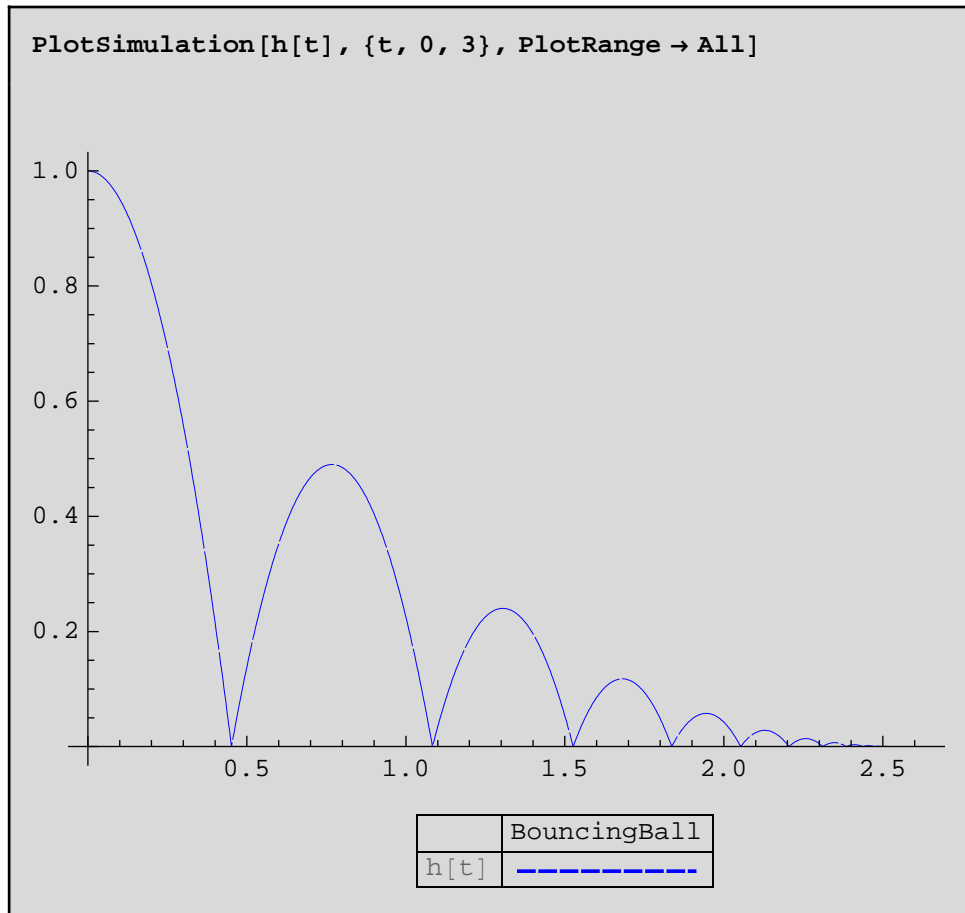
```
Simulate[BouncingBall, {t, 0, 3}]
```

```

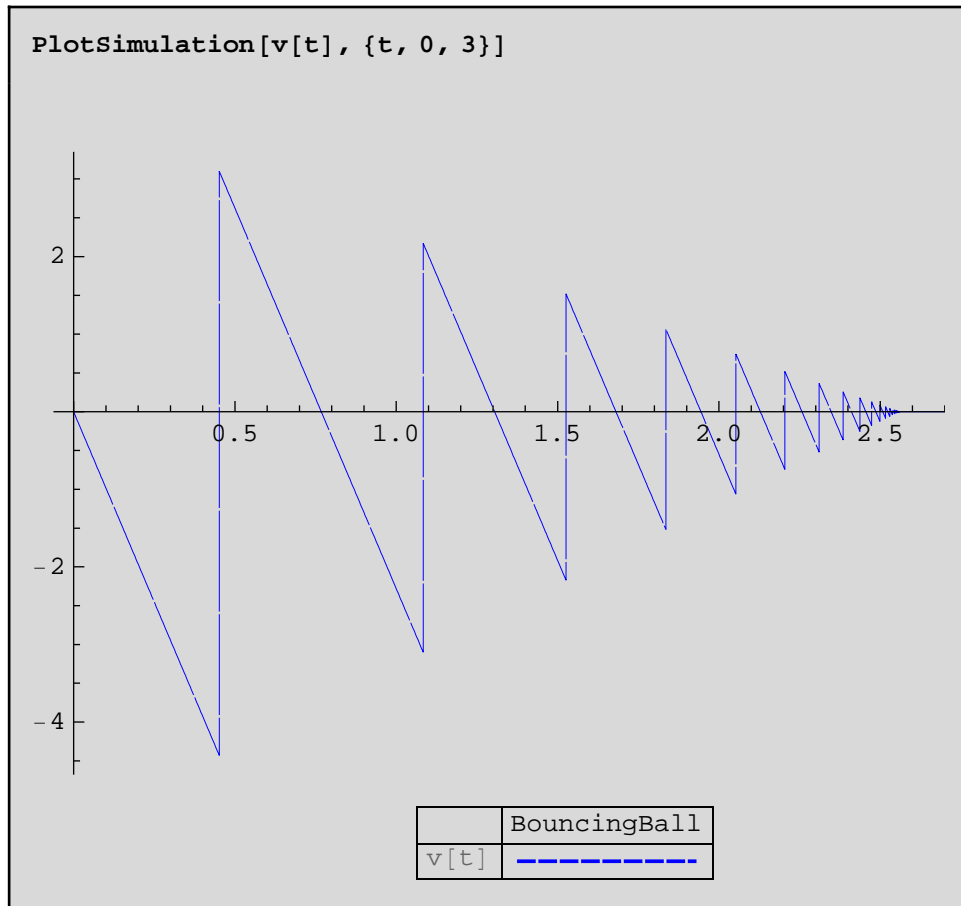
<SimulationData: "BouncingBall" : : {0., 3.} :
  1469 data points : 56 events : 10 variables>
{h', v', e, flying, g, h, impact, n_bounce, v, v_new}

```

A plot of the height looks like this:



A plot of the velocity looks like this.



The discrete variable `n_bounce` contains the number of bounces before the ball lies on the ground.

Note: Modelica variables containing underscores ('_') are in Mathematica represented using the special character `⏟` (underbracket). It can be entered using the command sequence

`⏟[⏟`

i.e. <Escape> + 'u' + '[' + <Escape>.

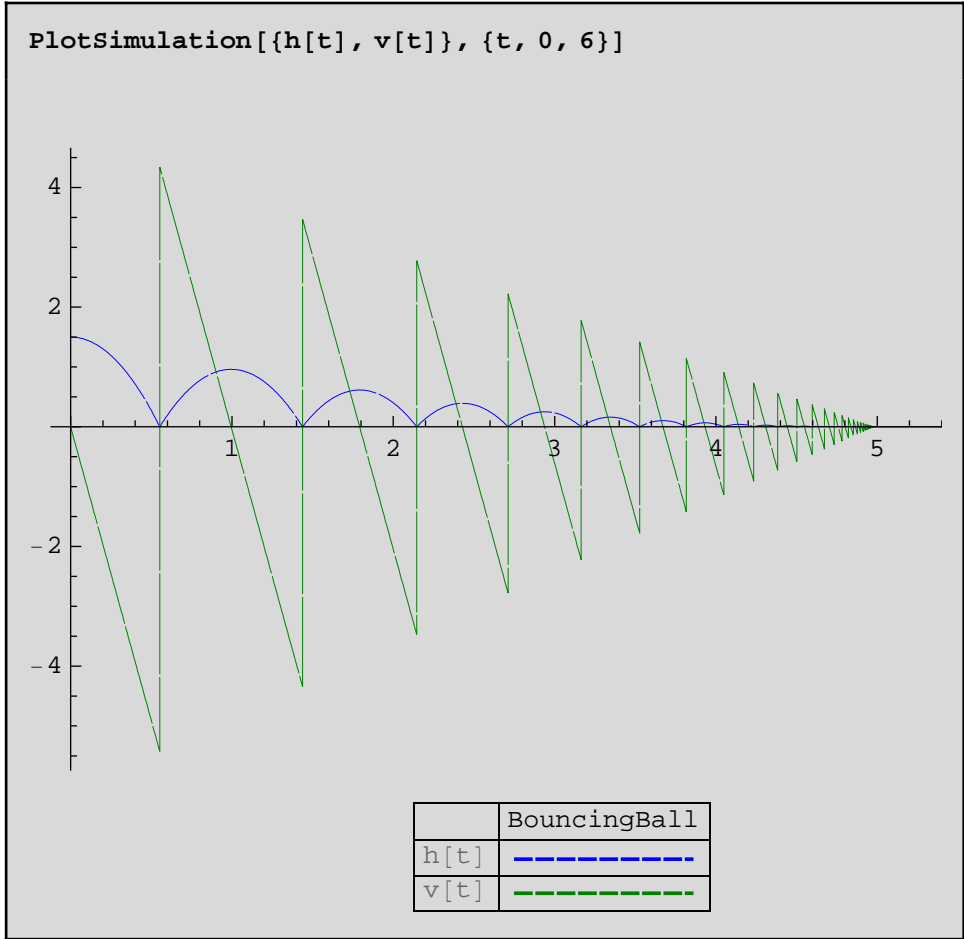
```
n_bounce [3]
```

```
29.
```

Change the coefficient of restitution, e , to 0.8 and the horizontal startvelocity, h , to 1.5. This is done by the `ParameterValues` and `InitialValues` option.

```
Simulate[BouncingBall, {t, 0, 6},  
ParameterValues → {e == 0.8}, InitialValues → {h ==  $\frac{3}{2}$ }] ;
```

Plot the result.



`n_bounce [6]`

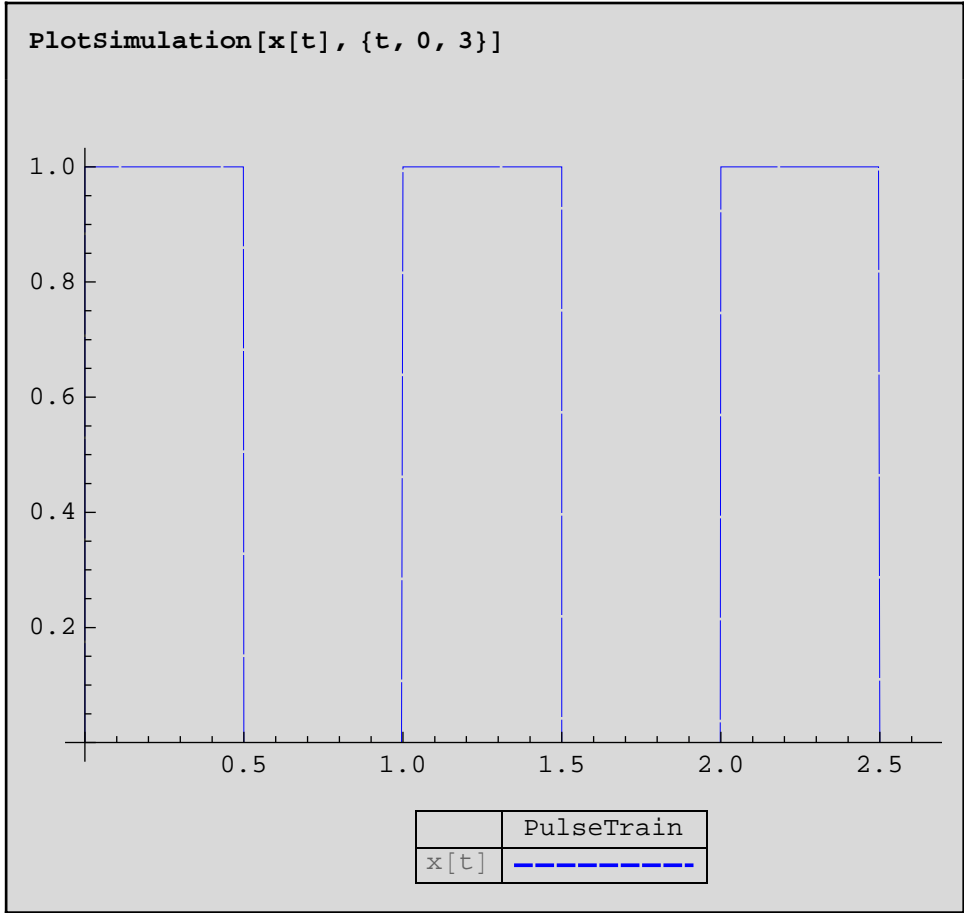
46.

4 Pulse Train

```
model PulseTrain "Model of a pulse train"
  Real x;
  Real pi=Modelica.Constants.pi;
equation
  x = if sin((2*pi)*time) > 0 then
    1
  else
    0;
end PulseTrain;
```

Simulate the system.

```
res = Simulate[PulseTrain,
  {t, 0, 3}, NumberOfIntervals → 500];
```

MathModelica® System Designer Professional

Steady State Linear Vibrations

© MathCore Engineering AB

1 Abstract

This Notebook examines the behavior of a coupled two-mass system with damping, using *Mathematica* and *MathModelica System Designer*. The behavior of the system is examined for different values of the parameters and the conditions for isolation of the first mass are derived.

Consider two masses coupled by a spring and a dashpot. The first mass is coupled to a wall by a second spring.

This example is based on and example by John M. Novak, Wolfram Research, Inc.

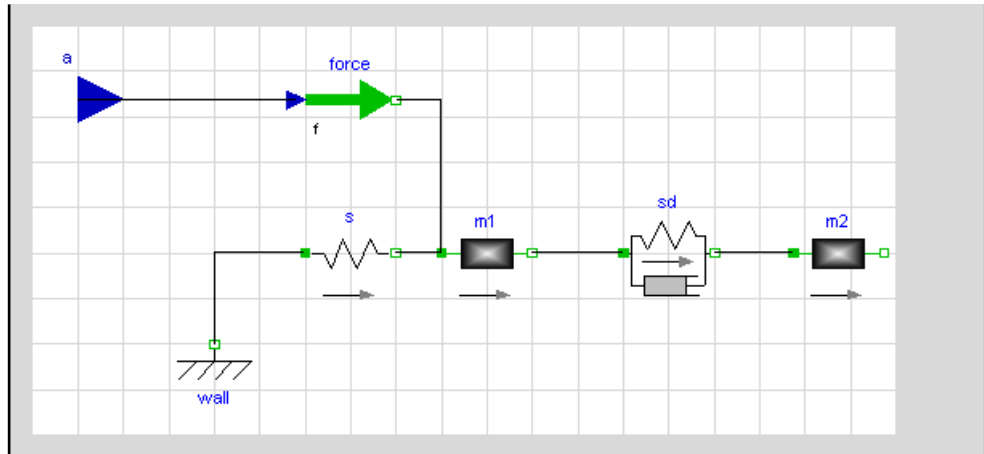
2 Initialization

To be able to use *MathModelica System Designer* within *Mathematica* we need to load the *MathModelica* package.

```
Needs["MathModelica`"]
```

3 Model

A model of two sliding masses with a string and dashpot between them and the first mass connected to a wall by a second spring can be made in the model editor by using components from the Modelica.Mechanics.Translational package. An external force acts on the first mass.



We also draw a suitable icon using the model editor.



The Modelica text for this model is (evaluate cell below to add the model to MathModelica)

```
model TwoMassSystem "A model of two sliding
masses and a string damper."
  import
  Modelica.Mechanics.Translational.SlidingMass;
  import
  Modelica.Mechanics.Translational.SpringDamper;
  import
  Modelica.Mechanics.Translational.Spring;
```

```

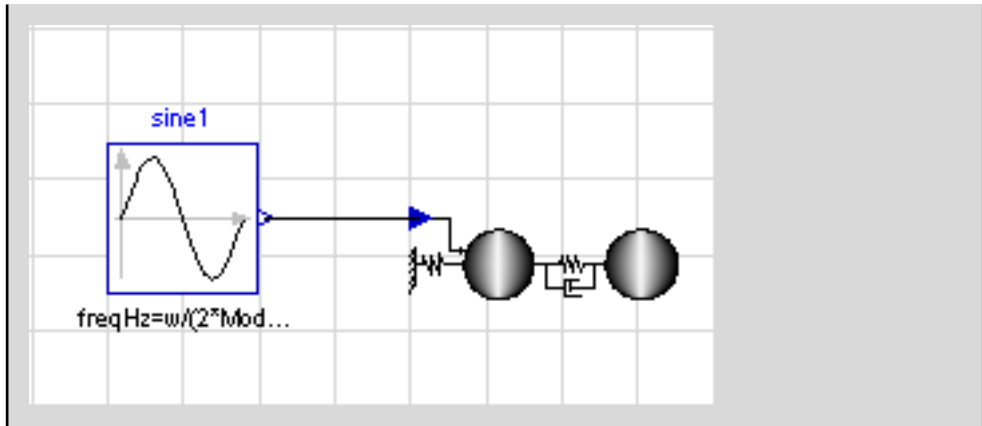
import
Modelica.Mechanics.Translational.Force;
import
Modelica.Mechanics.Translational.Fixed;
import Modelica.Blocks.Interfaces.RealInput;

SpringDamper sd;
SlidingMass m1;
SlidingMass m2;
Force force;
Fixed wall;
Spring s;
input RealInput a;

equation
connect(a,force.f);
connect(wall.flange_b,s.flange_a);
connect(s.flange_b,m1.flange_a);
connect(m1.flange_a,force.flange_b);
connect(sd.flange_b,m2.flange_a);
connect(m1.flange_b,sd.flange_a);
end TwoMassSystem;

```

To test it we create another model and connect the input force to a Sine curve:



```
model TwoMassSystemTest "A model of two
sliding masses and a string damper."
  parameter Modelica.SIunits.Frequency w=1;
  parameter Real A=1;
  Modelica.Blocks.Sources.Sine
sine1(amplitude=A,freqHz=w/(2*Modelica.Constants.pi));
  TwoMassSystem system;

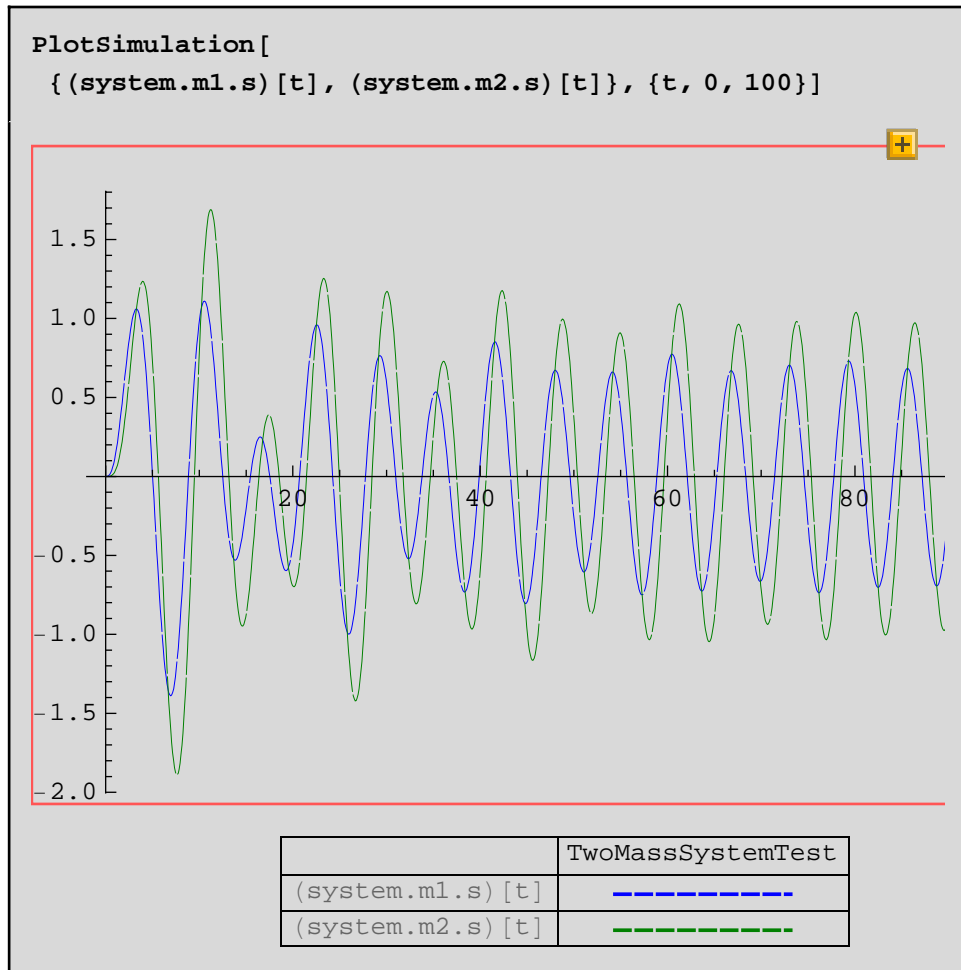
equation
  connect(sine1.y,system.a);
end TwoMassSystemTest;
```

We can now simulate the model using the Simulate command.

```
Simulate[TwoMassSystemTest, {t, 0, 100}]
```

```
<SimulationData: "TwoMassSystemTest" : : {0., 100.}
  : 1012 data points : 0 events : 60 variables>
{A, sine1.amplitude, sine1.freqHz, sine1.offset, sine1.pha
  se, sine1.startTime, sine1.y, system.a, system.forc
  e.f, system.force.flange_b.f, system.force.flange_b
  system.m1.flange_b.s', system.m1.flange_b.f, system.m
  1.flange_b.s, system.m1.L, system.m1.m, system.m1.s,
  ystem.m1.v, system.m2.a, system.m2.s', system.m2.v',
  system.m2.flange_a.s', system.m2.flange_a.f, system.m
  2.flange_a.s, system.m2.flange_b.f, system.m2.flang
  e_b.s, system.m2.L, system.m2.m, system.m2.s, system.
  2.v, system.s.c, system.sd.c, system.sd.d,
  system.sd.s_rel', system.sd.f, system.sd.flange_a.s', s
  ystem.sd.flange_a.f, system.sd.flange_a.s,
  system.sd.flange_b.s', system.sd.flange_b.f, system.s
  d.flange_b.s, system.sd.s_rel, system.sd.s_rel0, sy
  stem.sd.v_rel, system.s.f, system.s.flange_a.f, syst
  em.s.flange_a.s, system.s.flange_b.f, system.s.flang
  e_b.s, system.s.s_rel, system.s.s_rel0, system.wall.
  lange_b.f, system.wall.flange_b.s, system.wall.s0, w]
```

If we plot the position of the two masses we see that eventually the simulation reaches a steady state, with a periodic solution for the two positions.



4 Equations

To extract equations from Modelica models you can use the `ModelEquations` function. It will return the equations, variables and parameters of a Modelica model for further use in *Mathematica*. This can be used when performing steady state analysis, using *Mathematica's* `NDSolve` to solve the system, or as input to other application packages to perform control design or optimization, for example.

Extract the equations from the model

```
{eqns, ieqns, states, algvars, outputs, inputs, param} =  
  ModelEquations [TwoMassSystem];
```

Assume the system is driven by an external periodic driving force of frequency **ω** and magnitude **p_0** . We therefore replace the external force variable, $a[t]$ with a periodic signal using the euler formula.


```

eqns2 = eqns /. a[t] → p0 Exp[I omega t]

{
  eiωt p0 == (force.f) [t],
  (sd.f) [t] == sd.c (-sd.s_rel0 + (sd.s_rel) [t]) +
    sd.d (sd.der[s_rel]) [t],
  (sd.s_rel) [t] == (m2.flange_a.s) [t] - (sd.flange_a.s) [t],
  m1.m (m1.a) [t] == (sd.f) [t] + (m1.flange_a.f) [t],
  (force.flange_b.s) [t] == - $\frac{m1.L}{2}$  + (m1.s) [t],
  (sd.flange_a.s) [t] ==  $\frac{m1.L}{2}$  + (m1.s) [t],
  m2.m (m2.a) [t] == - (sd.f) [t],
  (m2.flange_a.s) [t] == - $\frac{m2.L}{2}$  + (m2.s) [t],
  (m2.flange_b.s) [t] ==  $\frac{m2.L}{2}$  + (m2.s) [t],
  (s.f) [t] == s.c (-s.s_rel0 + (s.s_rel) [t]),
  (s.s_rel) [t] == -wall.s0 + (force.flange_b.s) [t],
  - (force.f) [t] + (s.f) [t] + (m1.flange_a.f) [t] == 0,
  (m1.v) [t] == (m1.s)' [t], (m1.a) [t] == (m1.v)' [t],
  (m2.v) [t] == (m2.s)' [t], (m2.a) [t] == (m2.v)' [t],
  (m2.flange_a.der[s]) [t] == (m2.s)' [t],
  (sd.flange_a.der[s]) [t] == (m1.s)' [t],
  (sd.der[s_rel]) [t] ==
    (m2.flange_a.der[s]) [t] - (sd.flange_a.der[s]) [t]
}

```

Modelica does not allow second order derivatives and therefore requires additional state variables, thus the model has four states.

states

{ (m2.v) [t], (m2.s) [t], (m1.v) [t], (m1.s) [t] }

```
{(m2.v)[t], (m2.s)[t], (m1.v)[t], (m1.s)[t]}
{(m2.v)[t], (m2.s)[t], (m1.v)[t], (m1.s)[t]}
```

We eliminate all algebraic equations since they are not interesting for further analysis. We also observe that the states are coupled and we are only interested in the positions (m1.s and m2.s) in our analysis. We can therefore eliminate m1.v and m2.v by a simple transformation.

```
eqns3 = Eliminate[eqns2, algvars] /.
  {(m1.v) -> (m1.s)', (m2.v) -> (m2.s)'}

2 s.c wall.s0 == -2 ei ω t p0 - m1.L s.c - 2 s.c s.s_rel0 +
  m1.L s.d.c + m2.L s.d.c + 2 s.d.c s.d.s_rel0 +
  2 s.c (m1.s)[t] + 2 s.d.c (m1.s)[t] - 2 s.d.c (m2.s)[t] +
  2 s.d.d (m1.s)'[t] - 2 s.d.d (m2.s)'[t] + 2 m1.m (m1.s)''[t] &&
  2 m2.m (m2.s)''[t] == m1.L s.d.c + m2.L s.d.c +
  2 s.d.c s.d.s_rel0 + 2 s.d.c (m1.s)[t] -
  2 s.d.c (m2.s)[t] + 2 s.d.d (m1.s)'[t] - 2 s.d.d (m2.s)'[t]
```

The parameters from the model are given as a list of replacements

```
param
{sd.s_rel0 -> 0, sd.c -> 1, sd.d -> 1, m1.L -> 0, m1.m -> 1,
  m2.L -> 0, m2.m -> 1, wall.s0 -> 0, s.s_rel0 -> 0, s.c -> 1}
```

The parameters we are interested in are the damping, **d**, the first spring constant, **k1**, the second spring constant, **k2**, and the two masses **m1** and **m2**, therefore we build replacement rules for these to get more appropriate names.

```
param2 =
  {s.c -> k1, sd.d -> d, sd.c -> k2, m2.m -> m2, m1.m -> m1};
```

We simplify our equations by applying the parameter rules which results in an equation system with two equations and two states.

```
eqns4 = eqns3 /. param2 /. param
```

$$\begin{aligned}
 0 = & -2 e^{i \omega t} p_0 + 2 k_1 (m_1.s)[t] + \\
 & 2 k_2 (m_1.s)[t] - 2 k_2 (m_2.s)[t] + 2 d (m_1.s)'[t] - \\
 & 2 d (m_2.s)'[t] + 2 m_1 (m_1.s)''[t] \&\& \\
 2 m_2 (m_2.s)''[t] = & 2 k_2 (m_1.s)[t] - 2 k_2 (m_2.s)[t] + \\
 & 2 d (m_1.s)'[t] - 2 d (m_2.s)'[t]
 \end{aligned}$$

For a steady-state solution, **m1.s** and **m2.s** must have a periodic solution with the same frequency as the driving force, as we saw in the simulation above.

Therefore we replace m1.s and m2.s with a similar periodic solution as for a[t] previously.

```
eqns5 = eqns4 /. {m1.s -> (u1 Exp[I omega #] &),
  m2.s -> (u2 Exp[I omega #] &)}
```

$$\begin{aligned}
 0 = & -2 e^{i \omega t} p_0 + 2 e^{i \omega t} k_1 u_1 + 2 e^{i \omega t} k_2 u_1 + 2 i d e^{i \omega t} \omega u_1 - \\
 & 2 e^{i \omega t} m_1 \omega^2 u_1 - 2 e^{i \omega t} k_2 u_2 - 2 i d e^{i \omega t} \omega u_2 \&\& \\
 -2 e^{i \omega t} m_2 \omega^2 u_2 = & 2 e^{i \omega t} k_2 u_1 + 2 i d e^{i \omega t} \omega u_1 - \\
 & 2 e^{i \omega t} k_2 u_2 - 2 i d e^{i \omega t} \omega u_2
 \end{aligned}$$

where **u1** and **u2** are complex amplitudes.

As the amplitudes are independent of time, t, the equations can be further simplified by choosing t = 0.

```
eqns6 = eqns5 /. {t -> 0}
```

$$\begin{aligned}
 0 = & -2 p_0 + 2 k_1 u_1 + 2 k_2 u_1 + \\
 & 2 i d \omega u_1 - 2 m_1 \omega^2 u_1 - 2 k_2 u_2 - 2 i d \omega u_2 \&\& \\
 -2 m_2 \omega^2 u_2 = & 2 k_2 u_1 + 2 i d \omega u_1 - 2 k_2 u_2 - 2 i d \omega u_2
 \end{aligned}$$

The simplifications above make it possible to solve the equations for **u1** and **u2**

```
ans = First[Solve[eqns6, {u1, u2}]]

{u1 -> -((k2 + i d ω - m2 ω2) p0) / (-k1 k2 - i d k1 ω + k2 m1 ω2 +
      k1 m2 ω2 + k2 m2 ω2 + i d m1 ω3 + i d m2 ω3 - m1 m2 ω4),
  u2 -> -((k2 + i d ω) p0) / (-k1 k2 - i d k1 ω + k2 m1 ω2 +
      k1 m2 ω2 + k2 m2 ω2 + i d m1 ω3 + i d m2 ω3 - m1 m2 ω4)}
```

We have now reached an analytic solution for the amplitudes of the oscillating masses (u1 and u2) given the damping coefficient, d, the two masses, m1, m2 and the spring constant k2. This solution will in the next section be used for different kinds of analysis.

5 Behavior of the System

The symbolic solution encapsulates all the information about the steady state dynamics of the system. However, it is difficult to see immediately the consequences of these equations.

5.0.1 Normalized Displacement

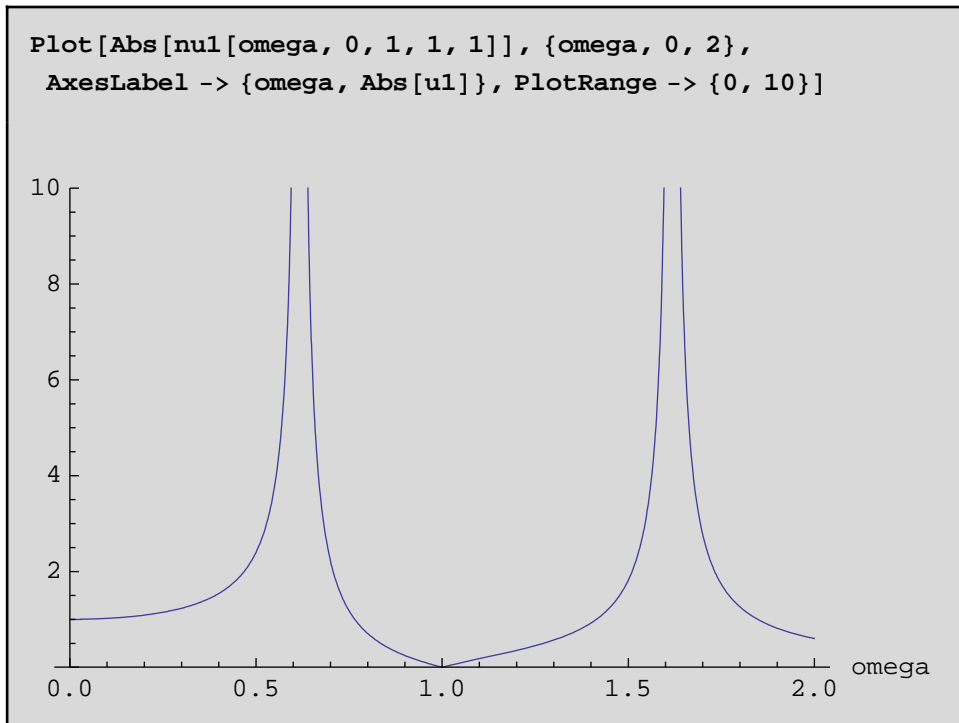
For use in the rest of this analysis, it is convenient to define a normalized displacement.

```
nu1[omega_, d_, m1_, m2_, k2_] =
  u1 /. ans /. {p0 -> 1, k1 -> 1} // Factor;

nu2[omega_, d_, m1_, m2_, k2_] =
  u2 /. ans /. {p0 -> 1, k1 -> 1} // Factor;
```

5.1 No Damping Force: Resonance

What is the behavior of the system when the damping term is zero? Plot the amplitude of the first displacement as a function of frequency. The remaining parameters are chosen to be unity, that is, all equal to one.



There are two resonances in this system. These occur when the denominator of \mathbf{u}_1 is zero.

```
Denominator[nu1[omega, 0, k2, m1, m2]] == 0
```

$$-m_2 + m_1 \omega^2 + k_2 m_2 \omega^2 + m_1 m_2 \omega^2 - k_2 m_1 \omega^4 = 0$$

This polynomial equation can be solved for the exact roots. For brevity, only the first root is shown.

```
Solve[%, omega] // First
```

$$\left\{ \omega \rightarrow - \frac{\sqrt{\frac{1}{k_2} + \frac{m_2}{k_2} + \frac{m_2}{m_1} - \frac{\sqrt{-4 k_2 m_1 m_2 + (-m_1 - k_2 m_2 - m_1 m_2)^2}}{k_2 m_1}}}{\sqrt{2}} \right\}$$

It can also be numerically approximated for specific values of the parameters.

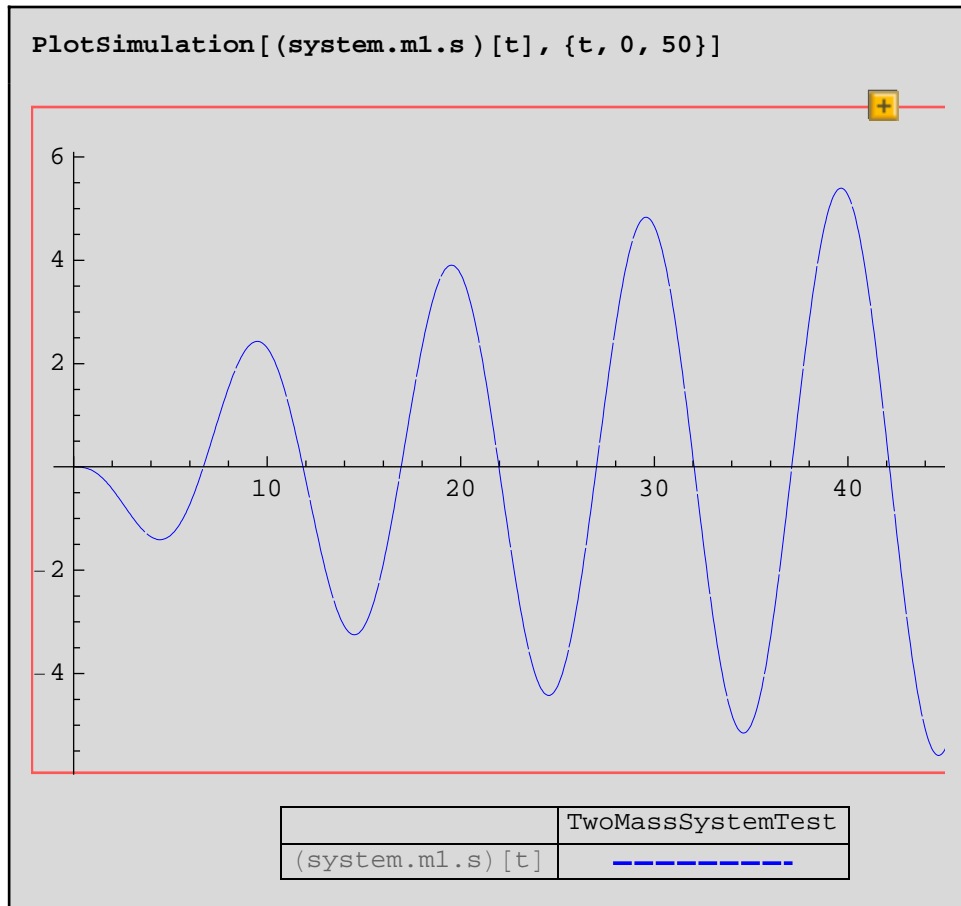
```
N[% /. {k2 -> 1, m1 -> 1, m2 -> 1}]
```

$$\{\omega \rightarrow -0.618034\}$$

The behaviour of the system at the resonance frequency can be illustrated by a simulation.

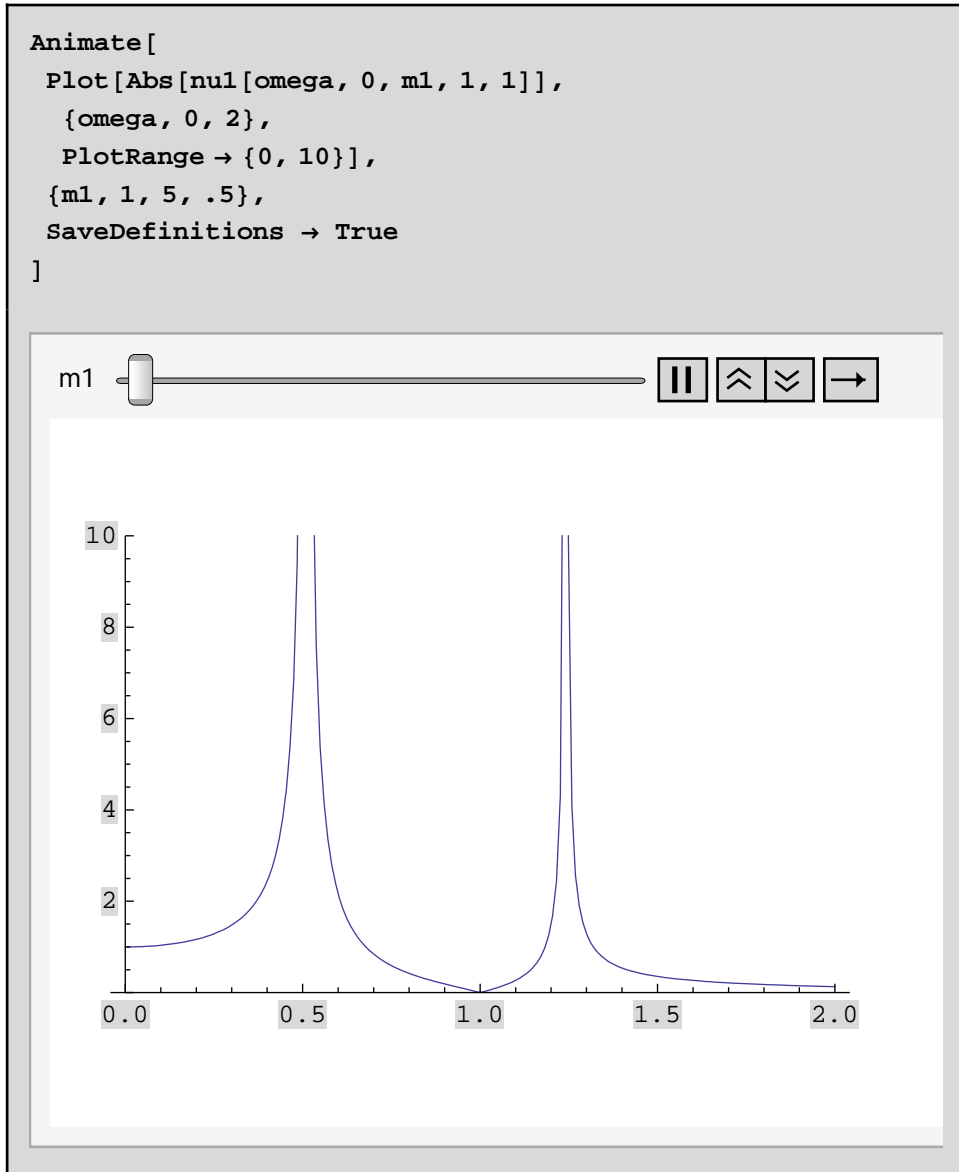
```
Simulate[TwoMassSystemTest, {t, 0, 50},  
ParameterValues -> {w == -0.6180339887498948`}];
```

We see that the amplitude of the first mass is increasing over time.



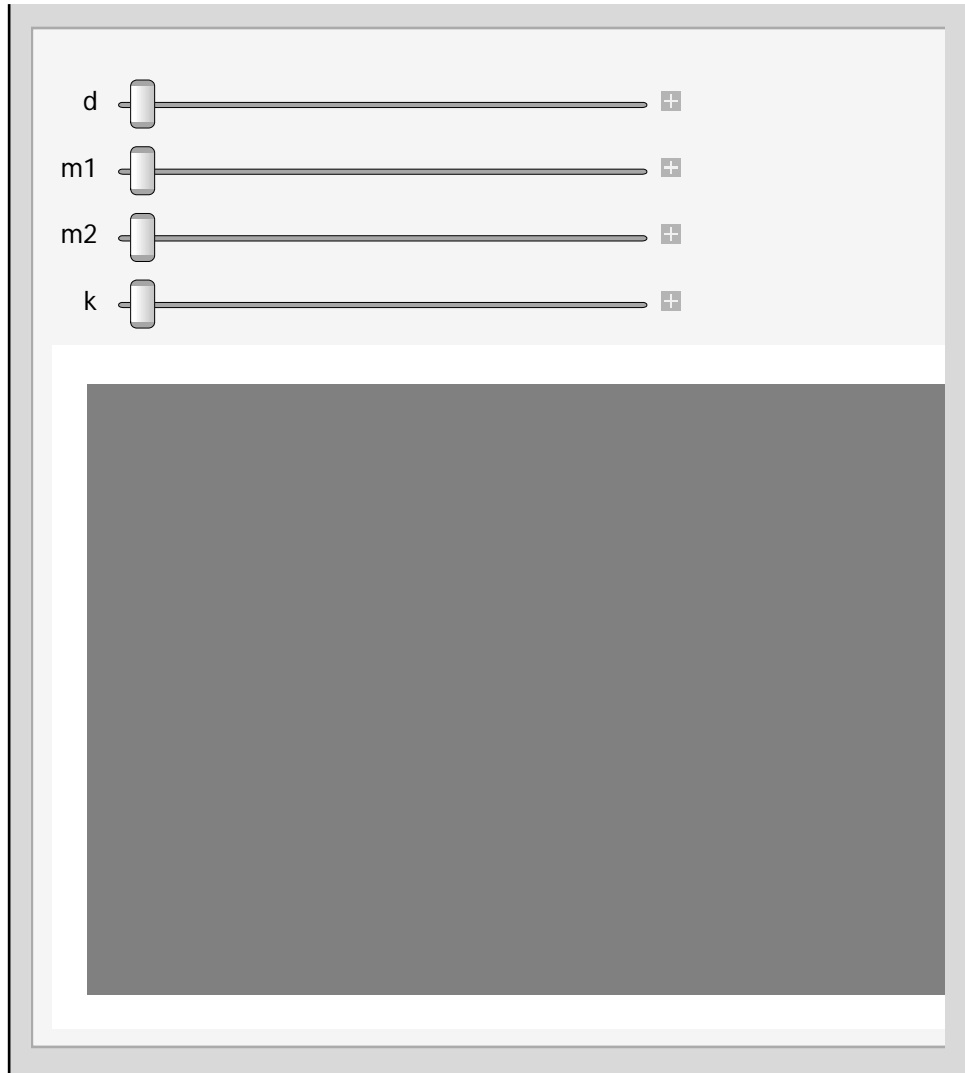
5.2 The Behavior as the Mass Changes

It is instructive to observe what happens as the mass is varied. A neat way to do this is to use Animate.



Using manipulate it is easy to create interactive plots to play around with. By pulling the sliders you can test how the behavior of the resonances changes when varying damping, d , mass, m_1 and m_2 , as well as the spring constant, k .


```
Manipulate[
  Plot[Abs[nu1[omega, d, m1, m2, k]],
    {omega, 0, 2},
    AxesLabel -> {omega, Abs[u1]},
    PlotRange -> {0, 10}],
  {d, 0, .4}, {m1, 1, 5}, {m2, 1, 5}, {k, 1, 5},
  SaveDefinitions -> True
]
```



5.2.1 Conditions for Isolation of the First Mass

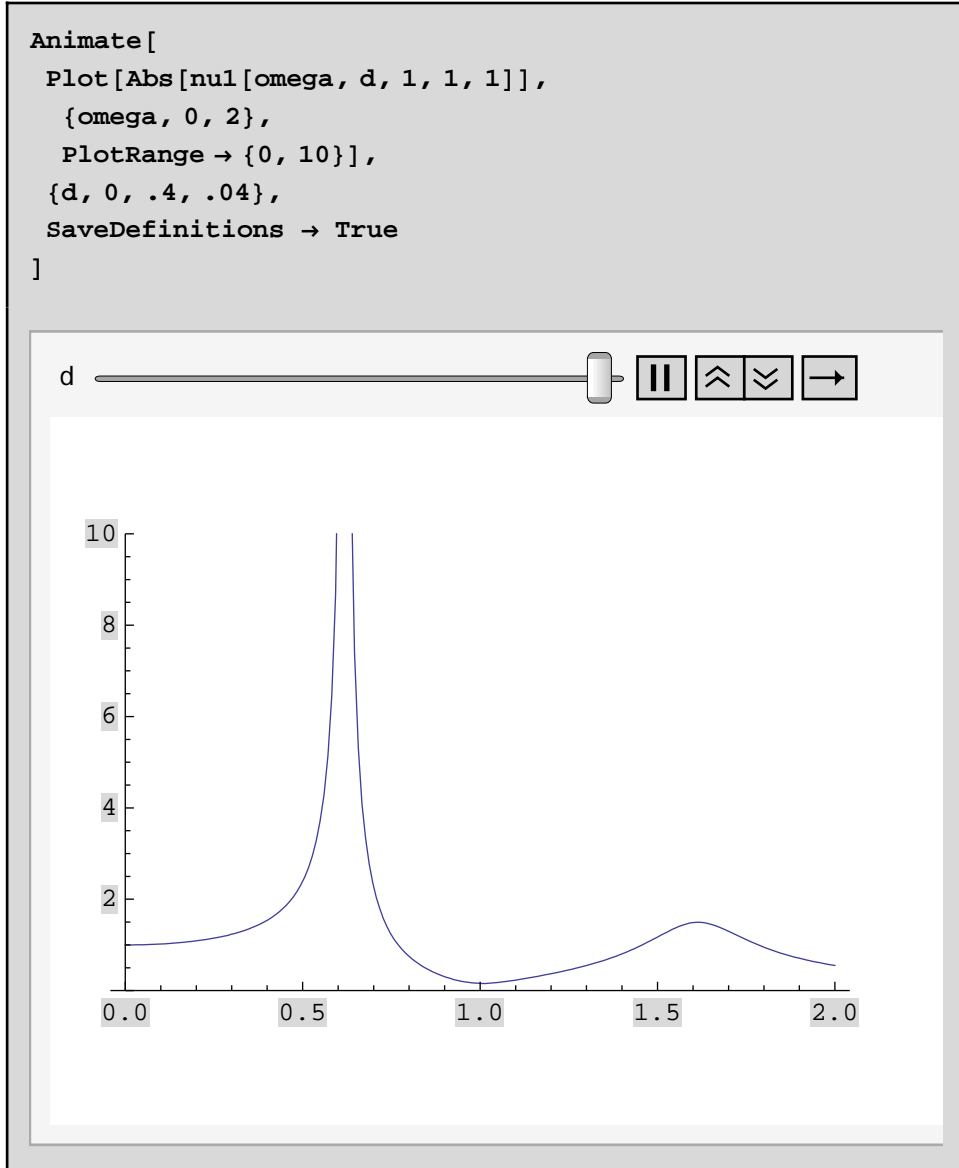
Notice that at $\omega=1$, u_1 is zero, for all values of m_1 . That is, the mass does not respond to the driving force and has been isolated. This occurs when the numerator of the expression is zero.

```
Solve[Numerator[nu1[omega, 0, m1, m2, k2]] == 0, omega]
```

$$\left\{ \left\{ \omega \rightarrow -\frac{\sqrt{k_2}}{\sqrt{m_2}} \right\}, \left\{ \omega \rightarrow \frac{\sqrt{k_2}}{\sqrt{m_2}} \right\} \right\}$$

5.3 The Behavior as a Damping Force Is Applied

A final point of interest comes in observing the effect of varying the applied damping. Again an animation can be used to show the effect. Note that the isolation of the first mass no longer occurs once a damping term is introduced.



An alternative is to use a three-dimensional plot to show the magnitude of the displacement as a function of the frequency and the damping term.

```
Manipulate[
  Plot3D[Abs[nu1[omega, d, m1, m2, k]],
    {omega, 0, 2},
    {d, 0, .4},
    PlotRange -> {0, 10},
    AxesLabel -> {"omega", "d"}],
  {m1, 1, 5}, {m2, 1, 5}, {k, 1, 5},
  SaveDefinitions -> True
]
```

