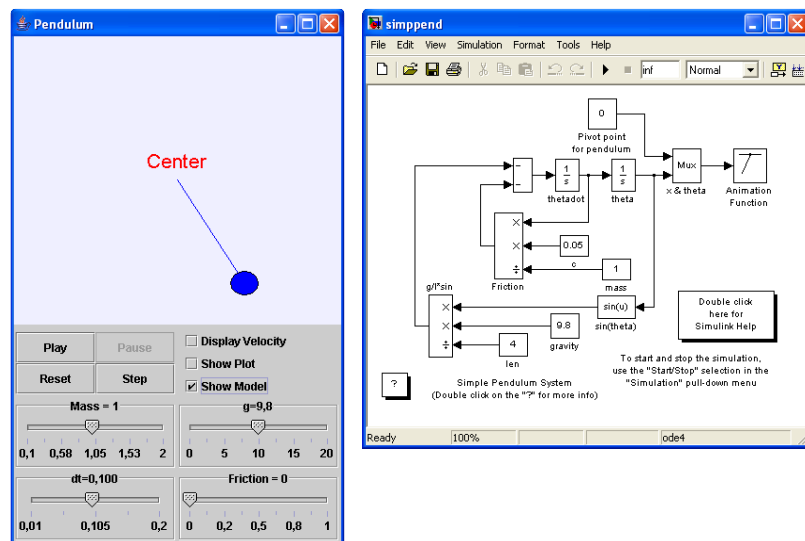




# Easy Java Simulations

Using Ejs to run Simulink models in  
an interactive way  
for version 3.4



**Francisco Esquembre**  
Universidad de Murcia. Spain

**José Sánchez-Moreno**  
Universidad Nacional de Educación a Distancia. Spain

**Ejs** uses *Open Source Physics* tools  
by **Wolfgang Christian**

January 2005  
<http://fem.um.es/Ejs>

## Contents

---

<b>1</b>	<b><i>Introduction .....</i></b>	<b><i>2</i></b>
<b>2</b>	<b><i>Calling Matlab functions .....</i></b>	<b><i>3</i></b>
2.1	A more sophisticated example .....	5
2.2	Using Matlab graphics.....	8
2.3	Using an M-file .....	9
2.4	Using more than one Matlab sessions.....	10
2.5	A first list of <i>_external</i> constructions .....	11
<b>3</b>	<b><i>Using Simulink models .....</i></b>	<b><i>13</i></b>
3.1	Creating Ejs variables for our simulation.....	14
3.2	Connecting Ejs variables to Simulink variables.....	15
3.3	Removing any visualization from the original model .....	20
3.4	Controlling the Simulink simulation .....	22
3.5	Creating a view for our simulation.....	23
3.6	Using more than one Simulink model .....	25
3.7	A second list of <i>_matlab</i> constructions .....	26

## 1 Introduction<sup>1</sup>

Simulink<sup>TM</sup> is a modeling tool, based in Matlab<sup>TM</sup>, that can be used to create dynamic system models in a graphical way (see [www.mathworks.com](http://www.mathworks.com)). However, models created with Simulink suffer from a certain lack of interactivity when they are to be used in pedagogical settings. To help improve the interactive and dynamic capabilities of these models, Ejs can be used together with Simulink to create new simulations based on existing Simulink models.

To be more precise, this means exactly that an author can create a simulation with Ejs that controls the execution of a Simulink model and that displays or modifies the values of parameters and variables of the model from the Ejs simulation. Additionally, authors can also call any Matlab function (either built-in or defined in an M-file) at any point in their Ejs models.

This document describes in detail the use of Matlab and Simulink with Ejs. The *\_examples/ExternalApps/Simulink* directory of the standard distribution of Ejs contains some illustrative examples of use of Matlab and Simulink with Ejs.

### Installation

Ejs can work with Matlab and Simulink (only under the Windows family of operating systems, at the moment of this writing), if you have them installed in your computer. This feature is already included in your distribution of Ejs. Hence, you don't need to install any other software. However, the possibility of using Matlab and Simulink from Ejs is not visually offered in the default distribution of Ejs, so that not to confuse unadverted users. To make this option visible, you need to edit Ejs' start-up file and modify it so that it includes the line:

```
set External=-externalApps
```

in the customization section of this file.

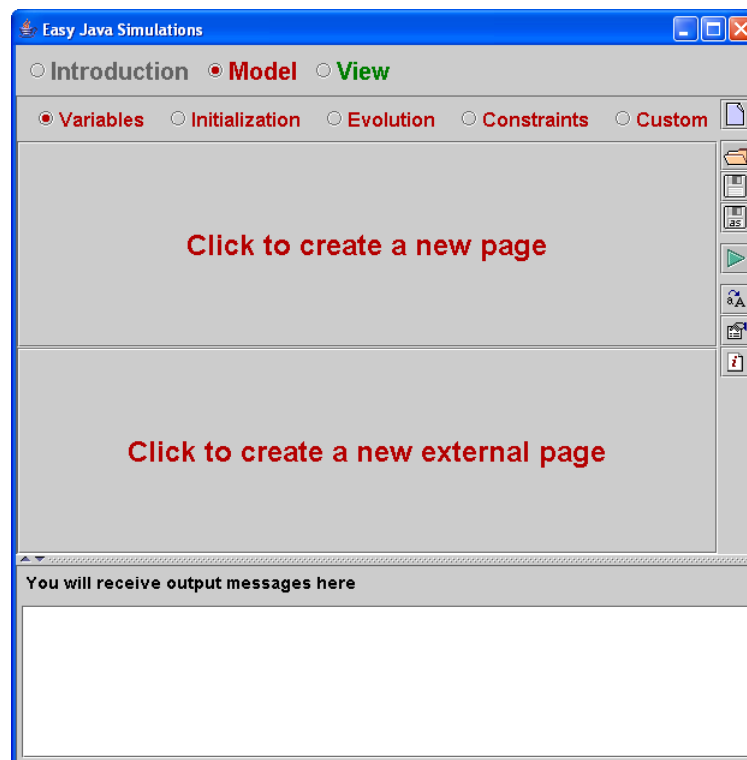
Ejs can always read a simulation which uses Matlab and Simulink, irrespective of whether you started Ejs with the *-externalApps* option or not. It will also correctly run the simulation (provided you have Matlab in your computer). However, if you didn't use this option, you may find difficulties to correctly edit the simulation file.

---

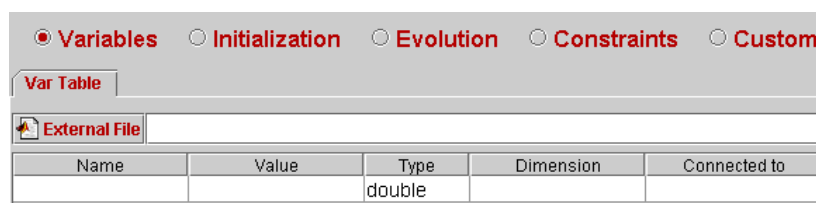
<sup>1</sup> **Important notice:** This document describes a feature new in release 3.4 that improves an existing capability of Ejs release 3.3. The old capability is kept for backwards compatibility, but this new document is intended to replace the previous document called "How to use Ejs with Matlab and Simulink", dated June 2004. **Except** in situations where the Simulink model can't be properly handled by the new feature. This document will try to warn you about these cases.

## 2 Calling Matlab functions

For simplicity, we start with the description of how to use Matlab functions from Ejs. The starting point is to create a special page of variables for your model called an *external page*. The possibility of creating one of these pages is only offered to you if you started Ejs with the *-externalApps* option (see *Installation* in Section 1). Then, a blank panel for variables will look as in the figure below.



Next figure shows in detail one of these pages right after creation.



The page looks very much like a standard variable page, except for the button and text field that allows us to provide an external (Matlab or Simulink) file, and for a new column in the table of variables labeled *Connected to*.

We don't really need to provide an external file to start working with Matlab, as well as we don't need to use the new column. This is only strictly necessary when we want to run Simulink models, as will be discussed in Section 3. For now, we just need to know that once we have created one of these external pages of variables, we have immediately Matlab at our perusal.

### How it works

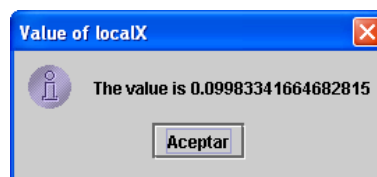
We can create and initialize variables the same way as we would do with any other table of variables. The gate to Matlab is open through the use of a new implicit object called `_external`. If you are not familiar with object-oriented programming, do not care very much about what an object is. For our purposes, `_external` can be considered as a new keyword that will allow us to construct sentences, in any of the code editors of Ejs, of the following type:

```
_external.setValue ("t",0.1);  
_external.eval ("x = sin(t)");  
double localX = _external.getDouble ("x");  
_alert (null,"Value of localX","The value is "+localX);
```

This is what these sentences do:

- The first of the sentences sets the value of the variable  $t$  in Matlab's workspace to  $0.1$ . Notice that  $t$  doesn't need to be defined as a variable in Ejs.
- The second of the sentences evaluates in Matlab the command " $x=\sin(t)$ ". Since  $t$  has been defined and set to  $0.1$  in Matlab's workspace by the previous sentence, this instruction evaluates the sine of  $0.1$  and defines the new Matlab variable  $x$ .
- The third sentence retrieves the value of  $x$ , just computed, from Matlab's workspace, and sets the value of a local variable in Ejs called *localX*.
- The last sentence is a call to the `_alert()` method that displays the value of the variable *localX* in a dialog window. This method is described in Ejs' manual and is not directly related to the use of Matlab.

If you write these sentences in the initialization section of an empty model and run this example, you will see that Ejs automatically starts a session of Matlab for you, and that the following dialog window appears on the screen.



This very simple example opens already a whole new world of possibilities, since it shows that you have all the power of Matlab accesible from Ejs.

But there is more. There are other constructions that can be used to access Matlab. The rest of this section describes them all.

### A first example

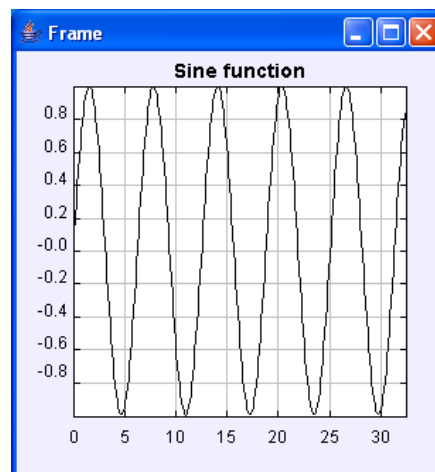
In order to give a full example with these simple sentences, we will plot the graph of the sine function using Ejs and Matlab. We start by declaring the following table of variables:

<input checked="" type="radio"/> Variables <input type="radio"/> Initialization <input type="radio"/> Evolution <input type="radio"/> Constraints <input type="radio"/> Custom				
Var Table				
External File				
Name	Value	Type	Dimension	Connected to
t	0	double		
x		double		

And then, creating a single evolution page as follows:

<input type="radio"/> Variables <input type="radio"/> Initialization <input checked="" type="radio"/> Evolution <input type="radio"/> Constraints <input type="radio"/> Custom	
Frames	Evol Page
per second MAX 20	<pre>t = t + 0.1; external.setValue ("t",t); external.eval ("x = sin(t)"); x = _external.getDouble ("x");</pre>

With this simple model we generate data for the graph of the sine function. We can now create a simple view to plot this function:



You will find this example in `_examples/ExternalApps/Simulink /MatlabExample1.xml`.

## 2.1 A more sophisticated example

We can now use all the powerful mathematical features of Matlab from Ejs. Instead of having to write the complicated Java code for many calculations, the model can simply make use of Matlab functions.

To illustrate this situation, let's imagine a simulation that needs to obtain the roots of the following fourth-order polynomial with user-defined coefficients  $a, b, c, d$  and  $e$ :

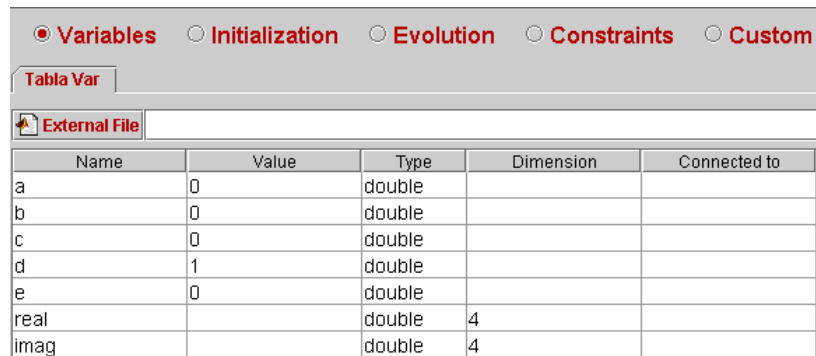
$$ax^4 + bx^3 + cx^2 + dx + e$$

We want to obtain the real and imaginary parts of the roots in two arrays: *real* and *imag*.

Finding the roots of a polynomial is a problem common to many disciplines. This problem is solved in Matlab using the function *roots*, and the real and imaginary parts of a complex number are obtained using the functions *real* and *imag*.

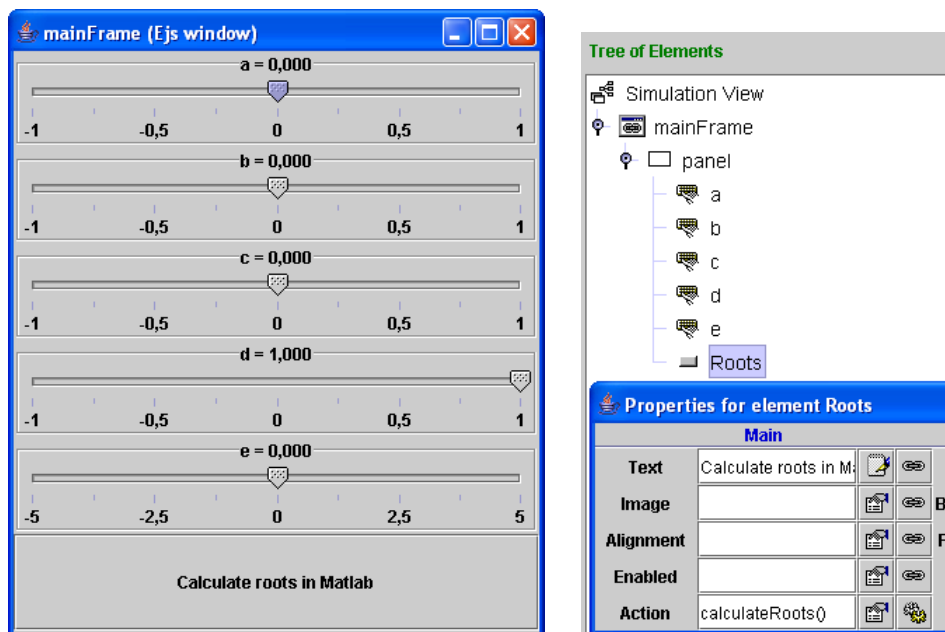
We provide only the code needed to make the necessary calculations using Matlab. We assume the reader can create the rest of the simulation.

First of all, we need to declare five variables to hold the coefficients of our polynomial. We also need two arrays to store the real and imaginary parts of the four roots:

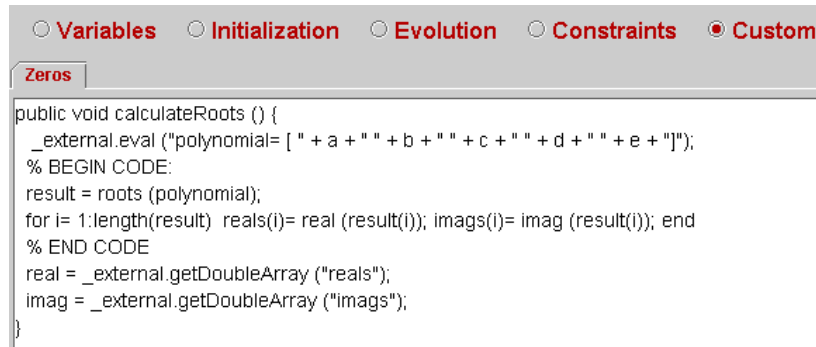


Name	Value	Type	Dimension	Connected to
a	0	double		
b	0	double		
c	0	double		
d	1	double		
e	0	double		
real		double	4	
imag		double	4	

To complete the example, a simple graphical user interface is designed to change the value of these five variables and thus change the coefficients of the polynomial. This view is just made of six elements: five sliders to modify each of the variables plus a button to ask Matlab to do the calculations and get the result back to Ejs. Next two figures show the view, the tree of elements, and a partial view of the properties of the button *Roots*.



We need to provide the Java code that will be executed when the button is clicked. We do this in a new page of the “Custom” subpanel of our model. The code necessary for this task is shown below:



Obviously, the name we have given to our Java method, *calculateRoots()*, must be the same we typed as the *Action* property of the button *Roots*.

Let's study the method *calculateRoots()*. The first sentence:

```
_external.eval ("polynomial= [ " + a + " " + b + " " + c + " " + d + " " + e + "]" );
```

sets the polynomial in Matlab's workspace according to Matlab's syntax for polynomials, that is, as a row vector of coefficients in descending order, including any zero term. Notice that these coefficients can be changed using the five sliders in the view.

Following this sentence, there is a block of sentences written exactly as we would write them directly in Matlab's prompt. This block is delimited by the special keywords:

```
% BEGIN CODE:
```

and

```
% END CODE
```

The use of this block construction is a simplified form of writing several consecutive calls to the *\_external.eval()* method. The keywords must be typed exactly as in the example.

In our case, the block between these keywords contains the sentences:

```
result = roots (polynomial);
for i= 1:length(result) reals(i)= real (result(i)); imags(i)= imag (result(i)); end
```

These sentences call the built-in Matlab function *roots* that calculates the roots of the polynomial, and separates the real and imaginary parts of the four roots, using Matlab's built-in functions: *real* and *imag*. Because the roots are stored as a column vector in the variable *result*, and since our two functions require a single value as input parameter, a loop must be coded to process the elements of the vector *result*.

After running this block, two new vectors have been created in Matlab's workspace: *reals* and *imags*. The last step needed to retrieve these values to the Ejs application is to use the *\_external* construction *\_getDoubleArray*. So, after running these two sentences:

```
real = _external.getDoubleArray ("reals");
imag = _external.getDoubleArray ("imags");
```

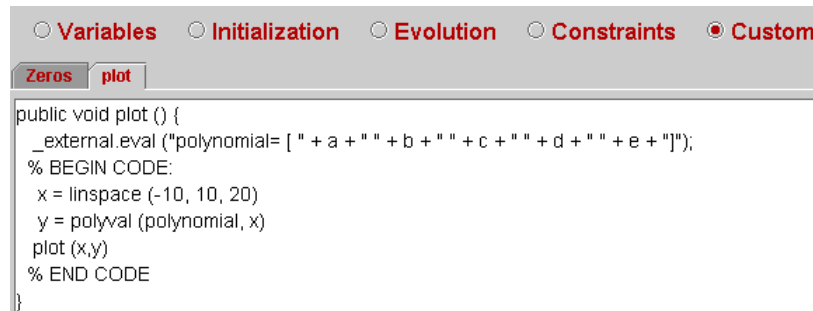
the Ejs arrays *real* and *imag* will hold the real and imaginary parts of the polynomial roots.



## 2.2 Using Matlab graphics

We can also make use of Matlab's graphical routines. As an example, let us plot the polynomial of the example above using the familiar Matlab function *plot* that can be used to plot 2-D data. The idea is to dynamically display the polynomial in a Matlab window, that is, every time a coefficient is changed in Ejs, Matlab will immediately update the graph of the polynomial.

First, we have to create a new Java method called, for example, *plot()*. The next figure shows this method:



The first sentence is already familiar to us. It creates the polynomial in Matlab's workspace. The rest of the sentences appear in a block of code. The line

```
x = linspace (-10, 10, 20)
```

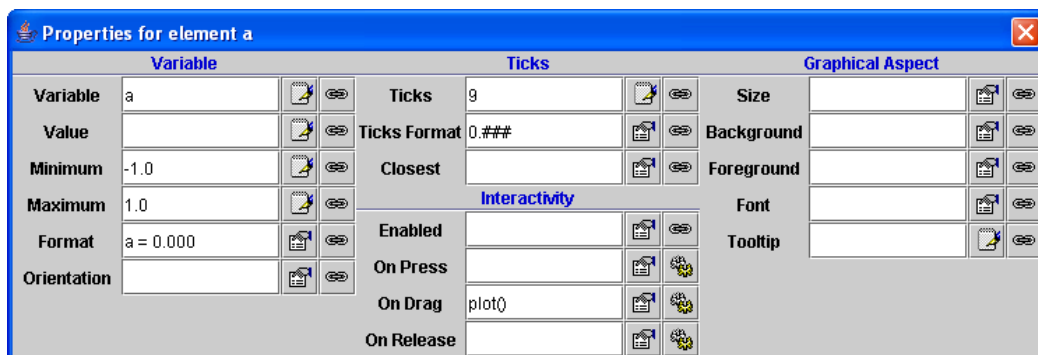
generates a row vector of 20 linearly equally spaced points between -10 and 10. We will evaluate the polynomial function in all these points. This is precisely what the next sentence does:

```
y = polyval (polynomial, x)
```

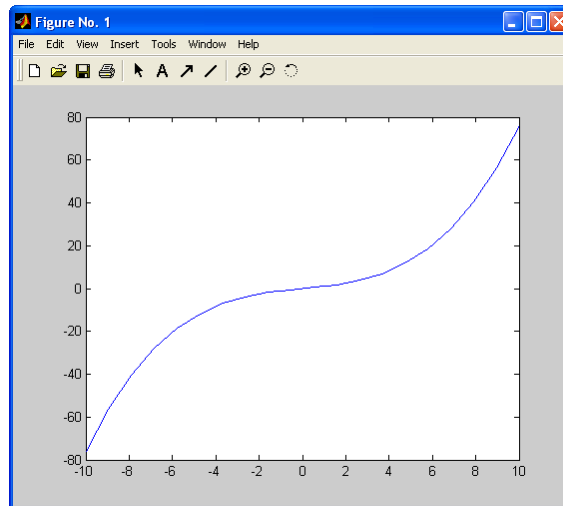
Finally, the plotting of the results is accomplished by means of the popular Matlab *plot* function:

```
y = plot(x,y)
```

The final step we need to plot the polynomial is to associate our *plot()* custom method to the action property *On drag* of each of the slider. We illustrate this only for the first of the sliders:



Now, each time the user moves a slider to change the value of a coefficient, the Matlab window shown in the next figure will be updated with the results of the evaluation of a new polynomial.



You can find this example in `_example/ExternalApps/Simulink/MatlabExample2.xml`.

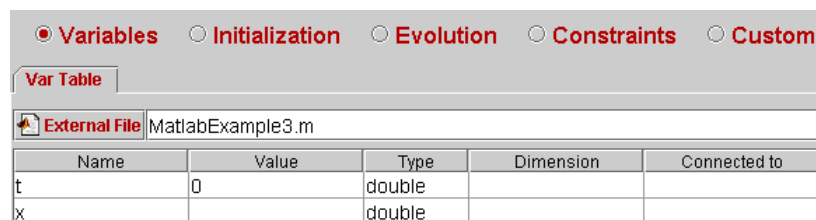
### 2.3 Using an M-file

We can also use a Matlab M-file if we need to initialize our Matlab's workspace, for instance to do some preliminary computations. For this, we would create an M-file in the directory in which the simulation will run (or in a subdirectory of it) and type the name of this file in the *External File* text field of the variable page.

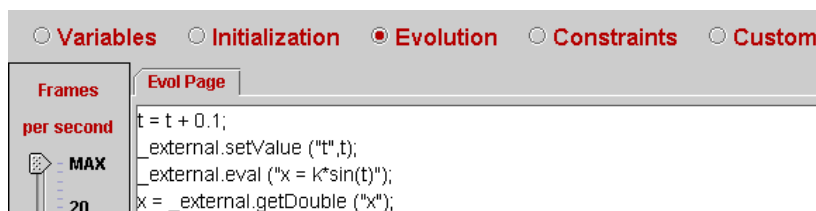
To illustrate this, let's change our first example slightly. We will first create an M-file in the *Simulations* directory called, say, *MatlabExample3.m* with the following single line in it.

`k = 2.0`

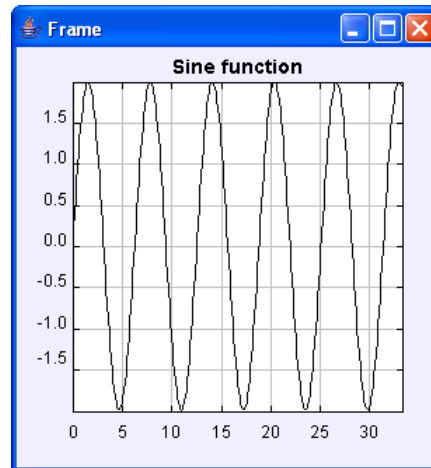
Now, we write this name in the *External File* text field, as shown in the picture:



The consequence is that, after start-up, the M-file *MatlabExample3.m* will be evaluated prior to playing our simulation. Therefore, the variable *k* is now accessible within Matlab's workspace. This means, for instance, that if we change our evolution page to read (notice the use of *k* in the *eval* construction):



we will get a plot two times higher (since  $k = 2.0$ ) than before.



You can find this example in `_example/ExternalApps/Simulink/MatlabExample3.xml`.

## 2.4 Using more than one Matlab sessions

In some cases, you may want to run more than one Matlab sessions at the same time. This can be useful if you want to do complicated computations and you want to keep both workspaces clearly separated. This is fairly possible and rather simple. You just need to create two variable pages, each of them with a different M-file in the corresponding *External File* textfield.

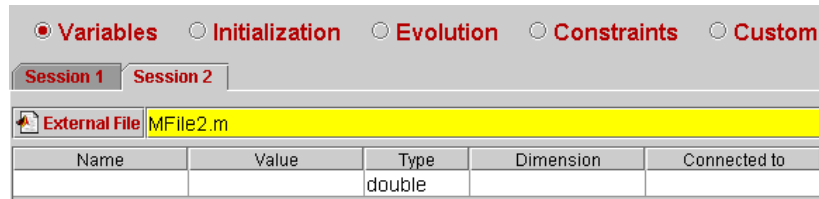
Notice that the M-files do not actually need to exist. This filenames can also be regarded as a way to name the Matlab's workspaces. In this case, that is, if the M-files do not exist, the text field will display a warning message and the field will display in red background. Despite this warning signals, the Matlab's workspaces will run just fine.

However, differently to the way we have been using *\_external* and blocks of code constructions until now, we now need to specify which workspace we want to address when we use any of the allowed methods.

For *\_external* constructions, this is simply done using a variation of the methods that accept a first String parameter. This parameter must be the name of one of the (existing or non-existing) M-files that you typed in the *External File* text fields of the external variable pages.

To illustrate this, assume that you create two Matlab variable pages such as these (we have changed the red background to yellow for readability):

<input checked="" type="radio"/> Variables <input type="radio"/> Initialization <input type="radio"/> Evolution <input type="radio"/> Constraints <input type="radio"/> Custom				
Session 1   Session 2				
External File MFile1.m				
Name	Value	Type	Dimension	Connected to
		double		



Now, if we use the sentences:

```
_external.setValue ("MFile1.m","k",1.0);
_external.setValue ("MFile2.m","k",2.0);
```

and run the simulations, we will get two warning messages with sentences like these:

```
Warning : the M-file MFile1.m does not exist!
Warning : the M-file MFile2.m does not exist!
```

However, two Matlab sessions will be created and, in them, the variable  $k$  will have the values  $1.0$  and  $2.0$ , respectively.

And this is all that is needed! Finally, please notice that the following rules apply:

- If only one Matlab session is started, the String parameter may be suppressed.
- If one page of variables leaves the *Matlab File* text field empty, the corresponding name for this session is the empty String "".
- If there are more than one sessions open and no String parameter for a session is indicated, then all the sessions will receive the *eval* and/or the *setValue* commands. The *getValue* command will however return the first value it finds.
- If two pages of variables specify the same M-file (which is perfectly legal, though rarely necessary) only one session with this name is started.

As for the `% BEGIN CODE`: construction, you must append to this keyword the name of the M-file which refers to the Matlab session in which you want to execute the code. As in:

```
% BEGIN CODE: MFile1.m
k = 1
% END CODE
```

## 2.5 A first list of *\_external* constructions

We are now ready to list all the methods that can be used to access Matlab from within Ejs. We leave until the next section the methods needed to run Simulink models.

Method	Description
void eval (String _command) void eval (String _mFile, String command)	Evaluates the command <i>_command</i> in the corresponding Matlab's workspace
void setValue (String _variable, (type) _value) void setValue (String _mFile, String _variable, (type)_value)	Sets the value of the variable <i>_variable</i> to the value indicated in the corresponding Matlab's workspace. <i>_value</i> can be either an integer, a double, a 1D array of doubles or a 2D array of doubles.
String getString (String _variable) String getString (String _mFile, String _variable)	Gets the value of the String variable <i>_variable</i> from the corresponding Matlab's workspace. The return type is String.

<code>double getDouble (String _variable)</code> <code>double getDouble (String _mFile, String _variable)</code>	Gets the value of the double or int variable <i>_variable</i> from the corresponding Matlab's workspace. The return type is double.
<code>double[] getDoubleArray (String _variable)</code> <code>double[] getDoubleArray (String _mFile, String _variable)</code>	Gets the value of the 1D array of doubles <i>_variable</i> from the corresponding Matlab's workspace.
<code>double[][] getDoubleArray2D (String _variable)</code> <code>double[][] getDoubleArray2D (String _mFile, String _variable)</code>	Gets the value of the 2D array of doubles <i>_variable</i> from the corresponding Matlab's workspace.
<code>% BEGIN CODE:</code>	Indicates the beginning of raw Matlab code.
<code>% BEGIN CODE: anMFile</code>	Indicates the beginning of raw Matlab code for the session indicated.
<code>% END CODE</code>	Indicates the end of raw Matlab code.

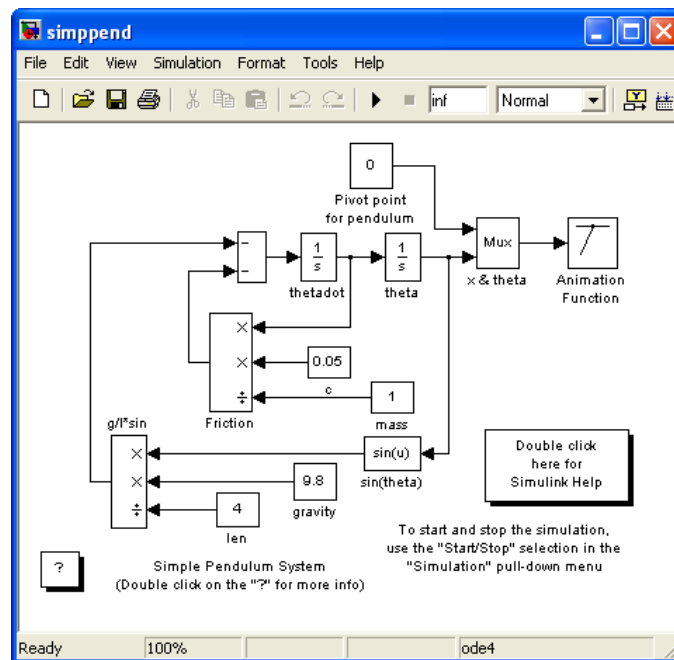
### 3 Using Simulink models

Using Simulink models from Ejs is possible and simple. The only necessary steps are the following:

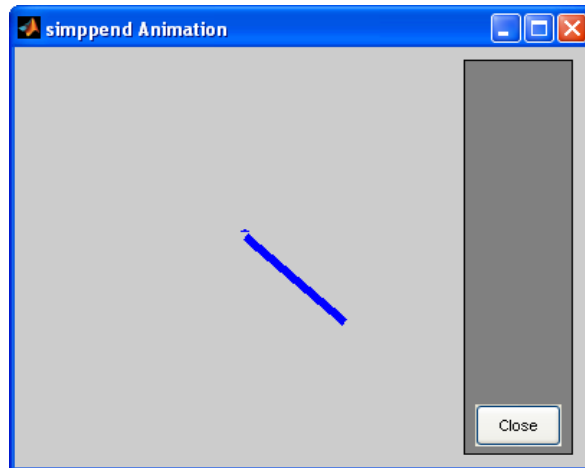
1. Linking Ejs variables to the corresponding variables or parameters in the Simulink model.
2. Including in Ejs model calls to methods that control the execution of the Simulink model.

#### A complete detailed example

We will illustrate these steps using an example of a Simulink model of a simple pendulum with friction. The example derives from one of Simulink's standard demos of Matlab 6.0 (although it will work well with later versions), which we have modified slightly so that the user can customize the different parameters of the simulation. The model is contained in the file `_examples/ExternalApps/Simulink/simppend.mdl` in the Simulations directory of your distribution of Ejs, and is displayed below.



As the picture shows, the model is a complete working simulation which can be run to display a pendulum of mass 1 and length 4, which oscillates with a friction coefficient of 0.05. The model includes a basic visualization of the oscillating pendulum, as shown below.



This Simulink model allows you to study very well the behaviour of a pendulum. But, if you want to modify any of its variables or parameters, you will need to stop the model, edit it in a manual form, and re-run it again.

Although this way of working is valid, you will most likely prefer a higher degree of interactivity. Also, you may want to have a better visualization of the phenomenon than what Simulink can offer.

This is precisely where Ejs can help you! With Ejs, you can provide a more dynamic and interactive interface to this model. And this, with minimal effort, as we will see.

### 3.1 Creating Ejs variables for our simulation

A simulation created with Ejs that uses a Simulink model is composed of the same parts as any other simulation created with Ejs. Thus, we need to fill the different panels of its model and to create a view for it. Obviously, the fact that Simulink already implements a model for our example will make our model in Ejs pretty small.

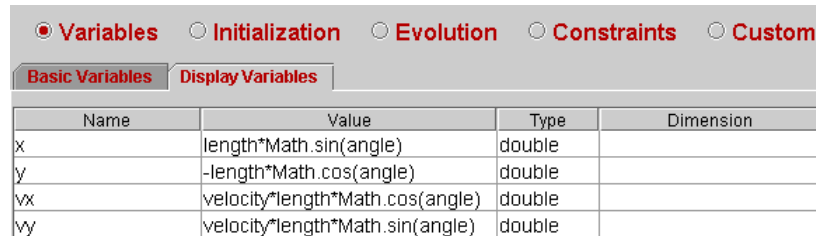
The first step in creating a model in Ejs requires that we define the variables that describe the phenomenon. This need to be done in this case, too. Inspired by the variables included in the Simulink model, we will create a table of variables as follows:

<input checked="" type="radio"/> Variables <input type="radio"/> Initialization <input type="radio"/> Evolution <input type="radio"/> Constraints <input type="radio"/> Custom				
Basic Variables				
External File: examples/ExternalApps/Simulink/simppend.mdl				
Name	Value	Type	Dimension	Connected to
time	0.0	double		
dt	0.1	double		
mass	1.0	double		
length	1.0	double		
gravity	9.8	double		
friction	0.0	double		
angle	Math.PI/6.0	double		
velocity	0.0	double		

Notice that this is one of the special pages of variables (a *external page*, Ejs calls it), which allows us to specify an external application. As you can see in the picture, we have selected this external file to be precisely our Simulink model for the pendulum.

The table of variables above contains all the variables that appear in the Simulink model, plus the simulation *time* and a parameter *dt*, which we will use to control the integration step of the Simulink model. Thus, the table of variables completely characterises the model.

But nothing prevents us from creating more tables of variables, if we need them. For instance, we can create a second page with Cartesian variables that will help us display the position and velocity of the pendulum.



The screenshot shows the 'Variables' tab in the Ejs software. It contains a table with the following data:

Name	Value	Type	Dimension
x	length*Math.sin(angle)	double	
y	-length*Math.cos(angle)	double	
vx	velocity*length*Math.cos(angle)	double	
vy	velocity*length*Math.sin(angle)	double	

### 3.2 Connecting Ejs variables to Simulink variables

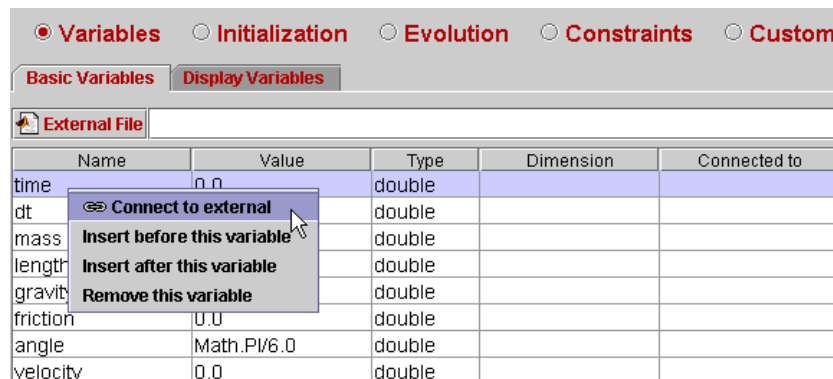
The key to using Simulink models from Ejs consists in associating or *connecting* the variables we have defined for the phenomenon in Ejs, with the variables and parameters of the existing Simulink model.

Connecting a connection means exactly:

- that the value of our Ejs variable will be automatically pushed to the variable of the model every time the Simulink model is going to be run, and
- that the value of the Simulink model variable will be automatically given back to our Ejs variable every time after running the Simulink model.

For instance, you may have already imagined that we want our *time* variable to reflect the simulation time of the Simulink model. Well, it's time to use the *Connected to* column of the table for this.

Right-click on the table row for the *time* variable and select from the pop-up menu that appears, the option *Connect to external*.



The screenshot shows the 'Variables' tab in the Ejs software. A context menu is open over the 'time' variable row. The menu options are:

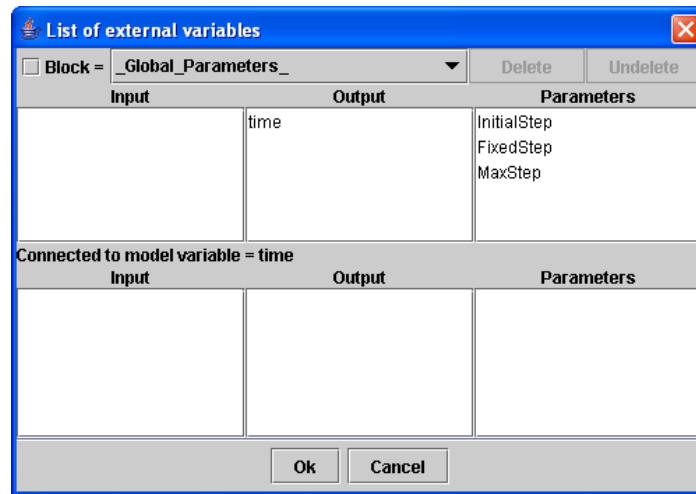
- Connect to external
- Insert before this variable
- Insert after this variable
- Remove this variable

The table data is as follows:

Name	Value	Type	Dimension	Connected to
time	n.n	double		
dt		double		
mass		double		
length		double		
gravity		double		
friction	0.0	double		
angle	Math.PI/6.0	double		
velocity	0.0	double		

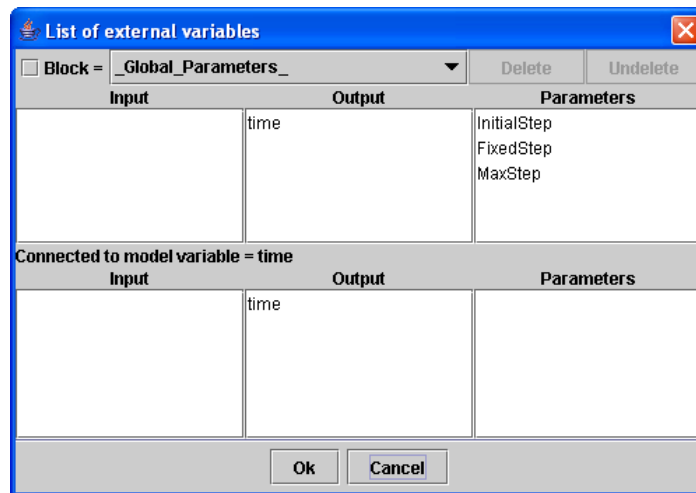


After a few seconds, which Ejs needs to launch Matlab in your computer, the following window will appear.<sup>2</sup>



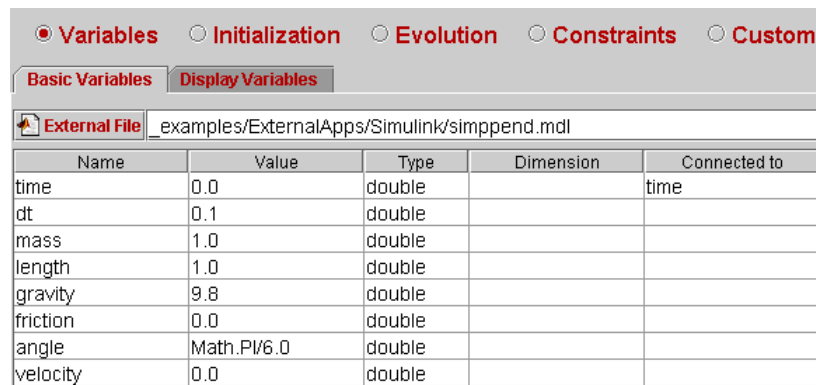
This window, which we will refer to as the *connection dialog*, helps us select which variable of the Simulink model will be associated to our variable *time*. In this case, the selection is straightforward, since the connection dialog is already offering us the global parameters of the simulation, among which we see the *time*.

We select the variable that will be connected to *time* by double-clicking our choice from the upper row of lists. The selected variable will then appear on the corresponding list in the lower row. See the figure below.



After doing this selection, we can click *Ok* and the association of both times will be reflected in the table of variables of Ejs.

<sup>2</sup> Obviously, this will only occur if you have Matlab installed in your computer.



Name	Value	Type	Dimension	Connected to
time	0.0	double		time
dt	0.1	double		
mass	1.0	double		
length	1.0	double		
gravity	9.8	double		
friction	0.0	double		
angle	Math.PI/6.0	double		
velocity	0.0	double		

That's all that is required to link the time of the Simulink model to Ejs' time.

Notice that there is no reason to give to our Ejs variables the same name they have in the original Simulink file (although this is rather reasonable, after all). Notice also that the type and dimension of your variable must match that of the variable in Simulink model. Actually, Ejs variables that are to be connected to Simulink variables are usually single doubles.

It's important to notice that the window we use to connect variables (the connection dialog) contains three different lists in each row, labeled *Input*, *Output* and *Parameters*, respectively. The category from which we select our variable has an effect on what we can do with it.

- Input variables can be freely changed from Ejs. That is, we can use any of Ejs' mechanisms to change them, at any time, either in the model or in the view. Any change we do to them will be immediately reflected in the model.
- Output variables can, on the contrary, only be read from Simulink. Hence, any change we do to their associated variables in Ejs won't affect their value in the Simulink model. This has an important exception, though, which consists in output variables of an integrator block. We will cover this case in detail a bit later in this section.
- Finally, parameters can also be changed from Ejs, but Simulink only guarantees using the new value if the model has not started running. That is, one should modify the value of parameters before running the model for the first time, or immediately after a reset of the simulation.


Although it seems like a complicated distinction, it is actually not such, since each category is typically used exactly as we would expect it to be. For instance, it is very unlikely that we want to change the time directly in our model. Instead, we expect Simulink to advance the time by iteratively running the model. And similarly for the other types of variables.

For example, our second variable in the table is *dt*. We want to associate this variable to the integration step of Simulink. We did not mention it, but our Simulink model is using a variable-step integration algorithm (more precisely an ode45 Dormand-Price integration method). Hence, we make again use of the connection dialog to connect the variable *dt* to two of Simulink's global parameters, *InitialStep* and *MaxStep*. This is what the table of variables will then look like:

☒ Variables
 ☐ Initialization
 ☐ Evolution
 ☐ Constraints
 ☐ Custom

Basic Variables

Display Variables


 External File

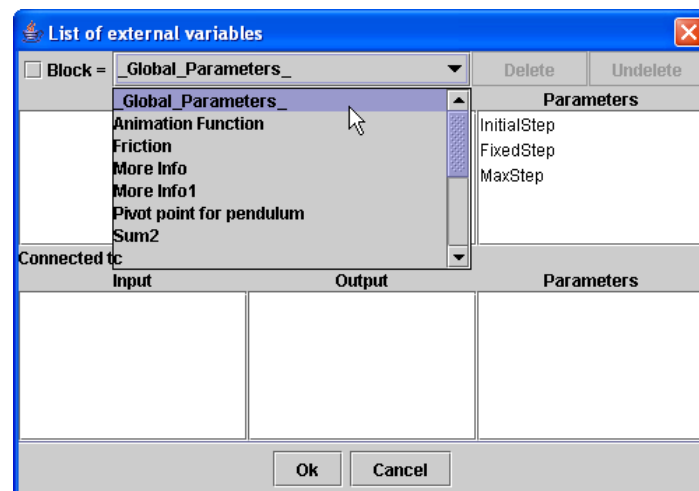
examples/ExternalApps/Simulink/simpend.mdl

Name	Value	Type	Dimension	Connected to
time	0.0	double		time
dt	0.1	double		InitialStep_ %_MaxStep
mass	1.0	double		
length	1.0	double		
gravity	9.8	double		
friction	0.0	double		
angle	Math.PI/6.0	double		
velocity	0.0	double		

*InitialStep* and *MaxStep* are both parameters, and, as mentioned above, Simulink will only accept changes to them before starting to run the model. Normally, this is what one would expect from it. We may want to run the model with different integration steps, but not in the same run, changing *dt* in the middle of a computation.

This is so important that we state it explicitly. **If you try to change a parameter during run-time, Simulink won't accept this change.** You may change *dt* in Ejs, but this change won't affect Simulink.

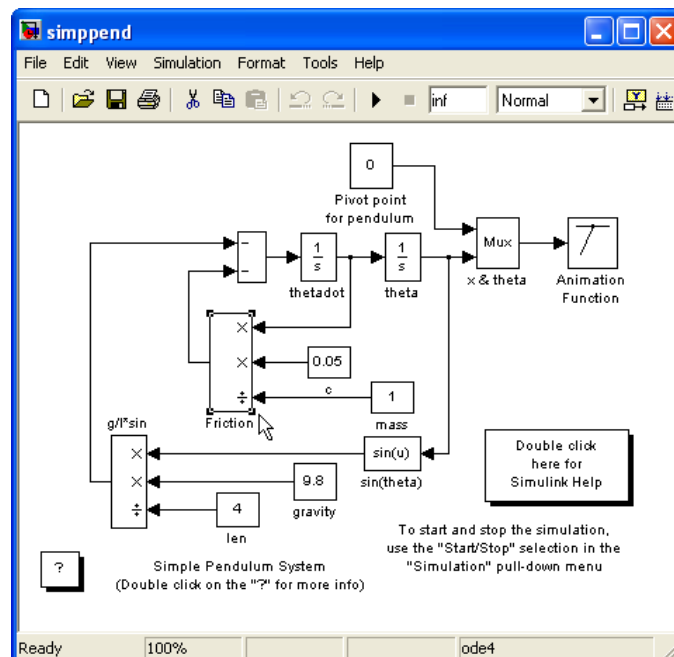
It's now time to consider an input variable. We want to associate the variable *mass* of our model to the mass of the Simulink model. The process required is similar. However, when we bring-in the connection dialog, the variable *mass* is not shown in the upper row of lists. The reason is that these lists only display the variables or parameters of one Simulink block at a time, and the block currently displayed is the one called *\_Global\_Parameters\_* (which is not a block displayed by Simulink's graphic display, by the way). To change the block you want to refer to, you can either click on the combo box which lists all the blocks in our model,



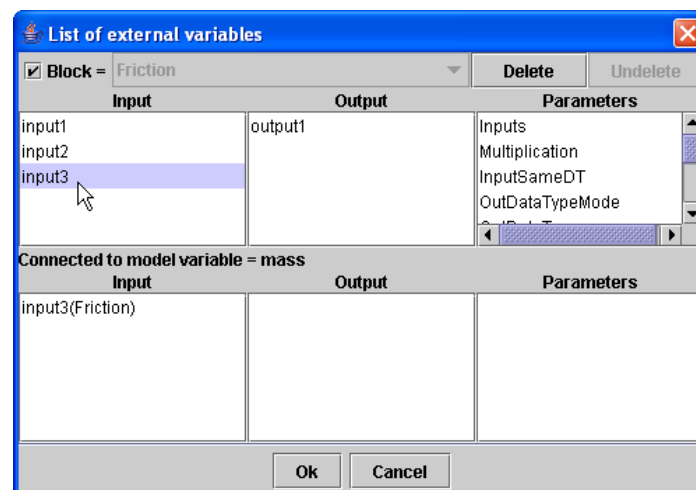
or, better yet, click on the *Block* check box on the upper-left corner of this window to bring-in a separate window with the Simulink model itself, and in this window click directly on the block that requires the value of the mass in the model.

In our case, this block is the mathematical block called *Friction* that appears selected in the picture below, right under the cursor. (The constant block called *mass* only serves to

provide a constant value for it, the value of the mass is really used in the *Friction* block to compute the frictional force that affects the motion of the pendulum.)

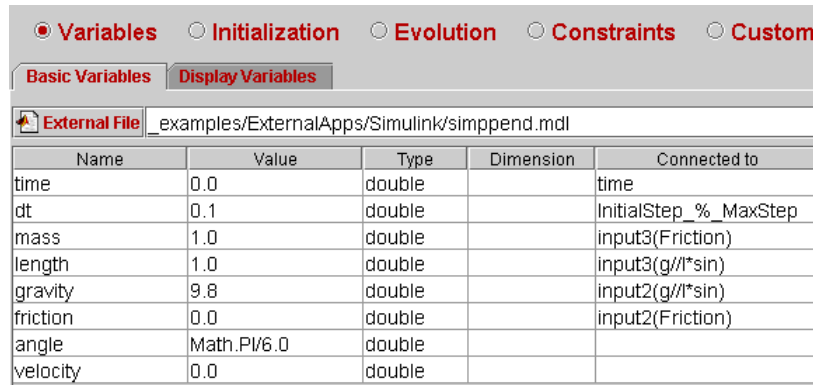


Once we have selected this block, we can return to the connection dialog and select the variable of the block that we are interested in. This variable is the third input variable of this block. When we double-click on it, the corresponding lower list displays its name and also the name of the block it belongs to, in parentheses.



This variable is an input variable, thus we can modify its value in Ejs whenever we want and the change will immediately have an effect in the Simulink model.

We leave to the reader to repeat this same process to connect the next three variables of Ejs, *length*, *gravity* and *friction*, to the corresponding input variables of the correct Simulink blocks. The table should look as in the picture below:



The screenshot shows the 'Variables' tab in the Ejs software. At the top, there are five radio buttons: 'Variables' (selected), 'Initialization', 'Evolution', 'Constraints', and 'Custom'. Below these are two sub-tabs: 'Basic Variables' and 'Display Variables'. The 'External File' field contains the path 'examples/ExternalApps/Simulink/simpend.mdl'. Below this is a table with the following data:

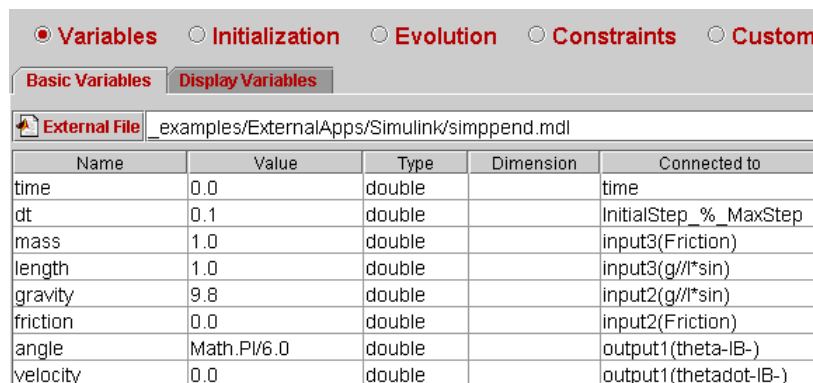
Name	Value	Type	Dimension	Connected to
time	0.0	double		time
dt	0.1	double		InitialStep_%_MaxStep
mass	1.0	double		input3(Friction)
length	1.0	double		input3(g/l*sin)
gravity	9.8	double		input2(g/l*sin)
friction	0.0	double		input2(Friction)
angle	Math.PI/6.0	double		
velocity	0.0	double		

We are almost done. We still need to connect the variables *angle* and (angular) *velocity* to the corresponding variables of the Simulink model. These variables are special because they are output variables of a type of Simulink block called *Integrator block* (because it integrates a differential equation).

Output variables of integrator blocks are in principle output variables, which means that (again, in principle) should only be read, not changed in Ejs. However, it is a common task, when exploring the behavior of a system described by a set of differential equations, to modify their initial conditions. For this reason, Ejs implements a special method that forces the Simulink model to read the values we set to Ejs variables even if they are connected to output variables of integrator blocks.

This method is called `_external.resetIC()`, and we will show when to use it later on. For the moment, it suffices to know that these variables are linked to Ejs variables the same way as any other output variable. The only visible change is that the name of the block will display an additional suffix, 'IB', that identifies the block as an integrator block. You should pay attention to this suffix, whenever it appears, in order to conveniently call the `_external.resetIC()` method.

The final table of variables is displayed below:



The screenshot shows the 'Variables' tab in the Ejs software, similar to the previous one but with updated connections for 'angle' and 'velocity'. The table data is as follows:

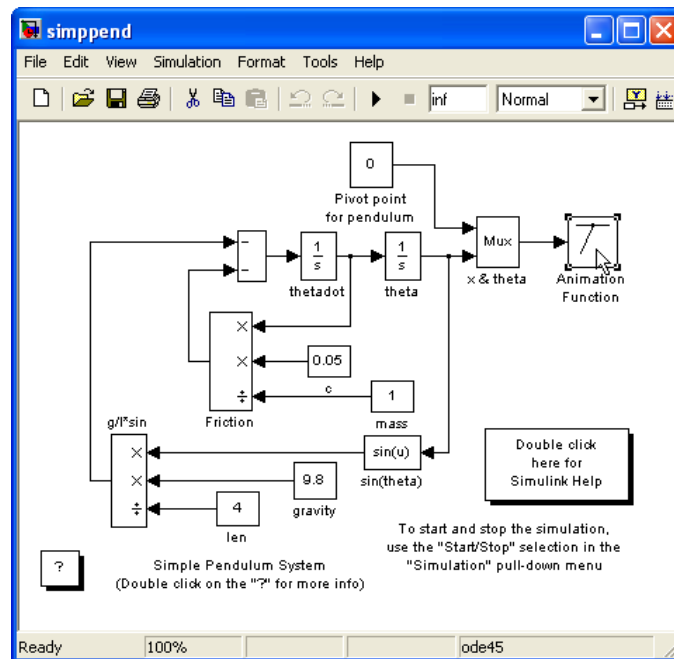
Name	Value	Type	Dimension	Connected to
time	0.0	double		time
dt	0.1	double		InitialStep_%_MaxStep
mass	1.0	double		input3(Friction)
length	1.0	double		input3(g/l*sin)
gravity	9.8	double		input2(g/l*sin)
friction	0.0	double		input2(Friction)
angle	Math.PI/6.0	double		output1(theta-IB-)
velocity	0.0	double		output1(thetadot-IB-)

### 3.3 Removing any visualization from the original model

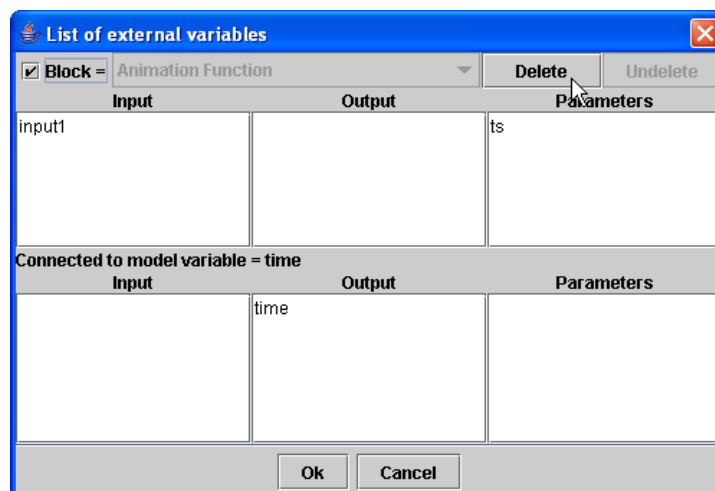
The final change, though this one is not mandatory, is to remove from the Simulink model any visualization of the phenomenon. Because we want to use Ejs for creating the view for the given model, it is usually unnecessary to keep the original visualization.

In the example we are working with, this equals to deleting the Simulink block called *Animation function* block. However, we don't need to remove it physically from the model. In fact, we haven't touched the original model at all until now, and we don't want to change it (we may still want to run it independently using Matlab). All we need to do is to tell Ejs that it should remove this block, **when it runs the model**.

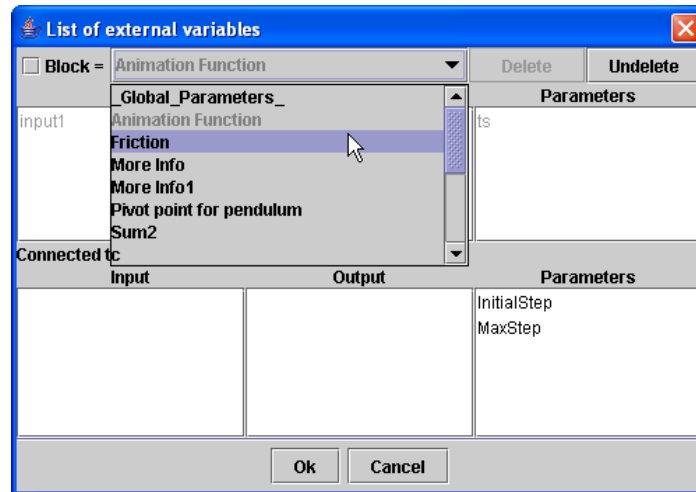
To do this, bring-in first the connection dialog by right-clicking on any variable, display the Simulink model, and select the block we want to remove by clicking in it. The figure shows this block selected.



Now, in the connection dialog window, click on the *Delete* button and Ejs will know that you want to remove this block from the final simulation.



Notice that deleting a block doesn't remove it from the model you are using within Ejs. Rather, Ejs will leave this original model unchanged. To show you that the block is selected for deletion, the connection dialog displays this block in greyed characters, as the figure below shows. Also, the *Undelete* button is now enabled, in case you want to undo the operation.



The information about which blocks need to be removed from the final simulation is remembered by Ejs when you save the simulation and reload it later. However, if you try to select a different external file for this page of variables, this information is lost (even if you select the same external file again).

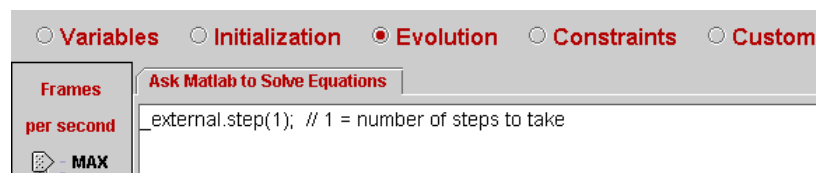
When editing the Simulink model, you may be tempted to change the name of an existing block. Don't do it, since this would confuse Ejs! Actually, if you try to change the name of an existing block, Ejs will complain and will ask you to undo the change. If you decide to go on with the change, you will need to save the Simulink model and reload the external file again (which would remove the information of deleted blocks!).

### 3.4 Controlling the Simulink simulation

After all our preparatory work, we can just run our Simulink model by using the special Java method:

```
_external.step(int steps);
```

in any suitable place in our Ejs model. A very appropriate place to include this sentence is an evolution page.



A call to this method has the following effects.

- 1) It first pushes the values of any Ejs variable which is connected to a Simulink input variable. Variables connected to Simulink parameters are also pushed, but they only have a real effect in the model if the simulation has not started running, that is, only when the time is still equal to 0. Ejs variables connected to Simulink output variables are not pushed, except those belonging to an integrator block (see the discussion below).
- 2) It runs the Simulink model as many steps as the parameter in parentheses states (1 in this case).

- 3) It finally retrieves the value of all Simulink output variables which are connected to Ejs variables.

This effectively causes the Simulink model to run one step and keeps a perfect synchronization of Ejs and Simulink variables.

### Changing initial conditions of integrator blocks

Output variables of integrators block play a particular role in our models. They are output variables, since they are computed as the result of an integration process, and we are surely interested in displaying their value in our Ejs view. However, a typical user will most likely be also interested in changing their value, which actually means changing the initial conditions of the differential equations involved.

Changing the initial conditions of a differential equation modeled with Simulink is not impossible, but is a bit traumatic, in the sense that Simulink needs to reset the integrator blocks and restart them. For this reason, Ejs provides a special Java method that does this task. A call to the method:

```
_external.resetIC( );
```

will restart all integrator blocks present in the Simulink model.

Ejs variables connected to Simulink output variables are always pushed to Simulink; however Simulink ignores these values until this method is called. Then, all integrator blocks will use whatever values the variables of Ejs have to restart the integration process. We will see below how to use this method to allow our user to interactively change the angular position and velocity of the pendulum.

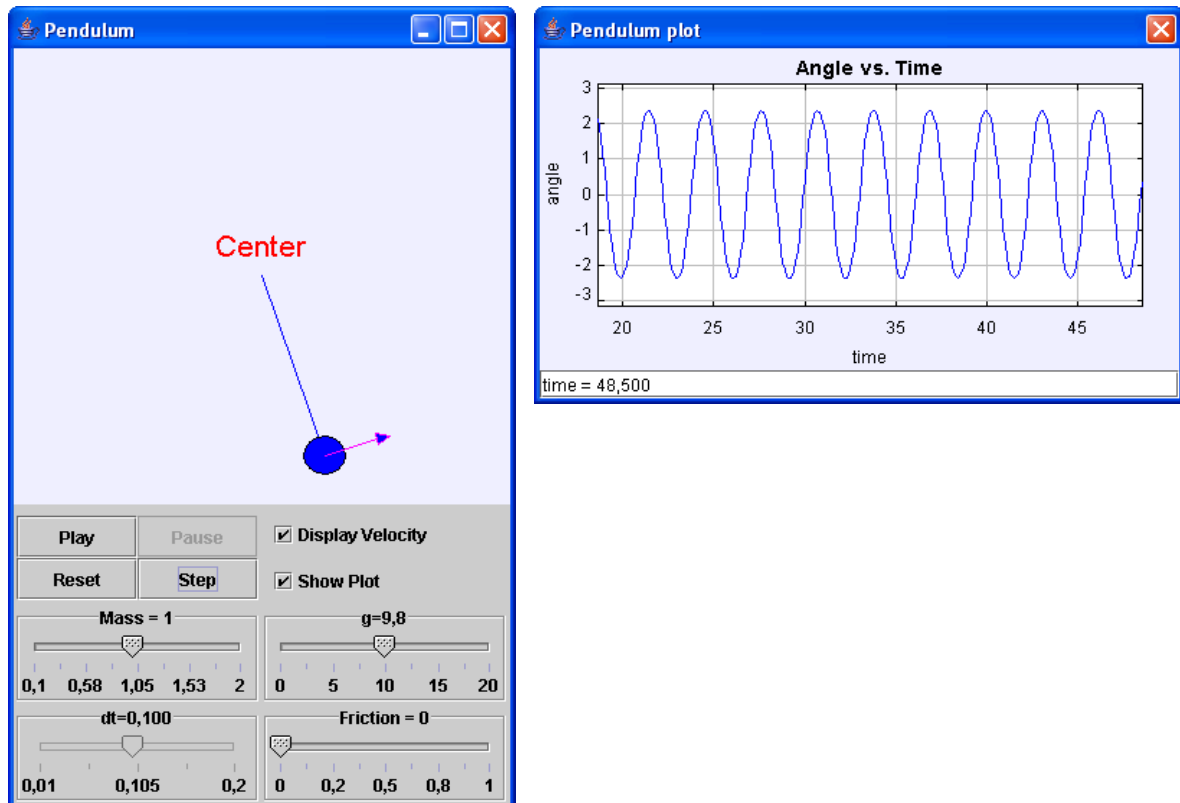
**Important exception.** In the current release of Ejs, integrator blocks that already use a reset condition are not treated correctly. Models which use these types of integrator blocks cannot therefore be used with this feature. Future releases of Ejs may solve this problem.

## 3.5 Creating a view for our simulation

Our Ejs model is ready to run the Simulink simulation. We now need to create a view for this simulation. This view can be constructed exactly as we would build a view for any other simulation created with Ejs, using the variables we declared in Ejs (using the names as declared in our table, not the names of the external Simulink file) and any of the available methods.

We will not describe how to create the view for this simulation in detail. We just show how it looks, once it is complete. See the figure below.





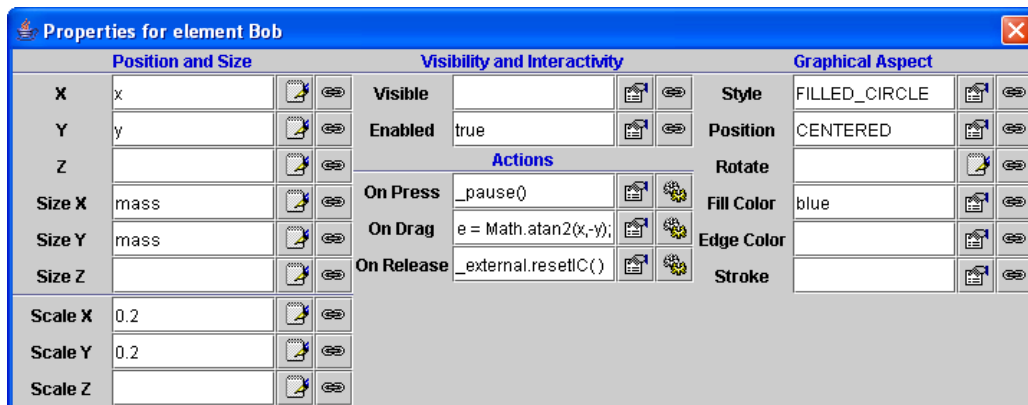
The display shows a first window that includes a visualization of the pendulum as it oscillates, together with some view elements that allow the user to modify some parameters of the model and to control the execution of the simulation. A second window to its right will display a graph of the angle versus the time as the simulation runs.

There are a couple of things to mention in this interface. The first one is the fact that the slider for  $dt$  must only be enabled when time is zero. This corresponds to the fact that  $dt$  is connected to a parameter of the model, hence we will not let the user modify it after the simulation has started to run, since it will have no real effect on the simulation. The way to achieve this is to set the *Enabled* property of the corresponding slider to the Boolean expression  $time==0$ , as the figure shows.

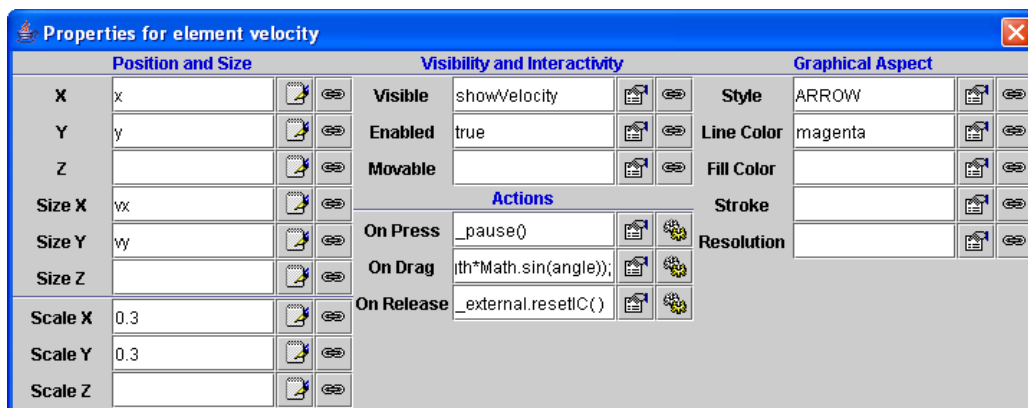


The second thing has to do with the interactivity associated to the bob of the pendulum and to its velocity vector. Changing the position of the bob in the plane corresponds to changing both the length of the pendulum and the initial angle of oscillation. Changing the length is not problematic, but changing the angle means changing the initial condition of the integrator block called *theta* of the Simulink model.

For this reason, as we said above, we need to set the *On Release* action property of the particle element that displays the bob to invoke the method `_external.resetIC()`, as the figure shows.



Similarly, changing the velocity vector implies changing the initial condition for the integrator block called *thetadot*. Again, the *On Release* action property of the arrow element that displays the velocity vector needs to invoke the same method.



If we forget to include these calls to `_external.resetIC()`, interacting with either the bob or the velocity vector of the pendulum might temporarily change the angle and angular velocity of the pendulum, **but the corresponding initial conditions of the Simulink model won't be affected.**

### 3.6 Using more than one Simulink model

It is possible to run more than one Simulink models from one single Ejs simulation. Although this is rarely needed, the process to do this is straightforward. You just need to create two or more separate pages of variables, each with its corresponding external Simulink file, and connect your Ejs variables to the corresponding variables of each of the models.

When you run the system, Ejs will take care of opening as many Matlab sessions as needed and will correctly manage all the connections.

When asking Ejs to play the Simulink models, you can still use the `_external.step()` method described above. This will play all the models at once in any order that the system sees fit. However, if you want to control the precise order in which the models are played, or if you

want to play only some particular models (but not all of them), you can use a new form of the *step* method. This form is used as

`_external.step (String externalFile, int n)`

where the specified string must match exactly one of the external Simulink files indicated in the *External File* text field of the variable pages.

As we said before, a call to the `_external.step()` method automatically takes care of updating all the connections among variables. However, the user can control when these connections are done individually by using the methods `setValues` and `getValues`. This is rarely needed, but the methods are provided for completeness. See the reference in the next section.

### 3.7 A second list of `_matlab` constructions

We can now complete the list of methods that can be used in `_matlab` constructions.

Method	Description
<code>void setValues (String _mFile)</code>	Sets the value of all connected Matlab variables to that of the corresponding Ejs variables.  Not to be used directly by users
<code>void getValues (String _mFile)</code>	Gets the values of all connected Matlab variables and gives them to the corresponding Ejs variables.  Not to be used directly by users
<code>void step (double _n)</code> <code>void step (String _mFile, double _n)</code>	Plays all the Simulink models <code>_n</code> times  Plays the given Simulink mode <code>_n</code> times.  Note: these methods take care of setting and getting the values before and after playing the simulation. There is, therefore, no need to use the <code>setValues</code> and <code>getValues</code> methods.
<code>void reset ()</code> <code>void reset (String _mFile)</code>	Resets all simulations to its initial state.  Resets the simulation to its initial state.  Both methods are used internally. Not to be used directly by users