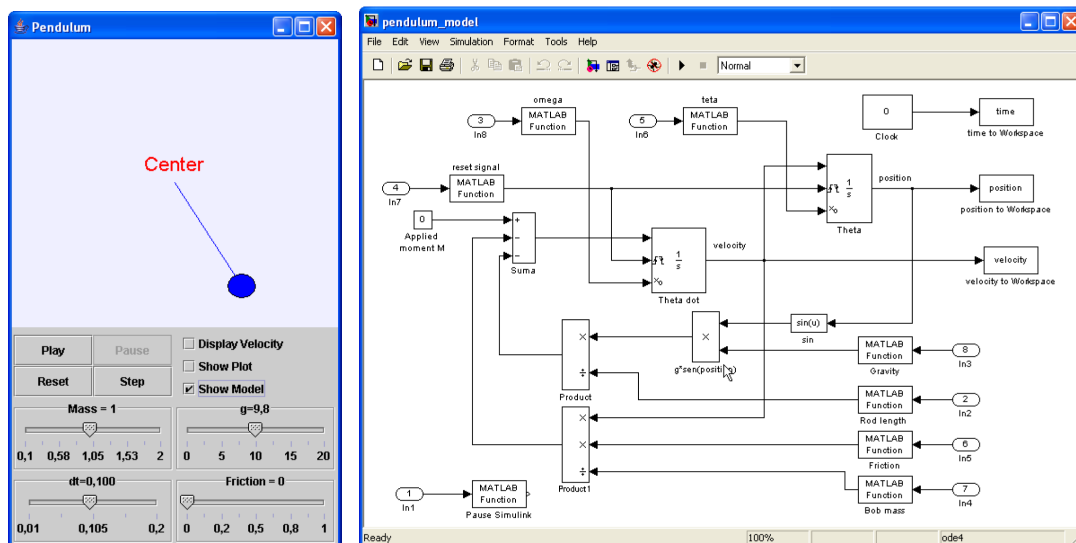
Easy Java Simulations

How to use Ejs with Matlab and Simulink

for version 3.3



Francisco Esquembre
Universidad de Murcia. Spain
José Sánchez
Universidad Nacional de Educación a Distancia. Spain

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

June 2004
http://fem.um.es/Ejs

Contents

1 Introduction 2

2 Calling Matlab functions 3

2.1 A first example 4

2.2 A second example: using Matlab to do calculations 5

2.3 Using Matlab graphics 8

2.4 Using an M-file 9

2.5 Using more than one Matlab sessions 10

2.6 A first list of *_external* constructions 12

3 Using Simulink models 13

3.1 Changes required to your Simulink model 13

3.2 Creation of a M-file to serve as interface 19

3.3 Connecting variables in Ejs with Matlab variables 21

3.4 Playing the Matlab simulation 22

3.5 Using more than one Simulink model 24

3.6 Modifying variables and parameters in run-time. 24

3.7 A second list of _matlab constructions 25

© Francisco Esquembre, April 2003

1

1 Introduction

Ejs can work with Matlab and Simulink if you have them installed in your computer. This feature is already included in your distribution of **Ejs**, although it may not be visible by default so that not to confuse unadverted users.

To make this option visible, you must run **Ejs** with the following option:

```
-externalApps
```

which you'll need to add to the *EjsOptions* line in the batch file with which you usually start **Ejs**. The standard distribution of **Ejs** includes a file called *Ejs_externalApps.bat* that runs Ejs (using an English interface) with this option turned on.

If **Ejs** is asked to read a simulation which uses Matlab, it will do it with no problem, irrespective of whether you started **Ejs** with the *-externalApps* option or not. You will also be able to run the simulation (provided you have Matlab in your computer, of course) with no problems. However, if you didn't use this option, you may find difficulties to correctly edit the simulation file.

Using **Ejs** in conjunction with Matlab means that users of **Ejs** can (provided that they have Matlab installed in their computer):

a. call any Matlab function (either built-in or defined in an M-file) at any point in their models,

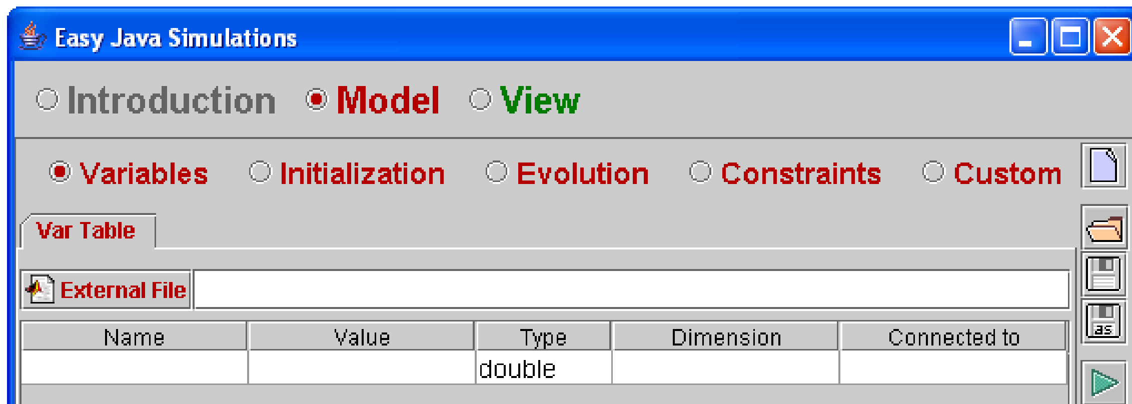
b. run Simulink models.

This document describes in detail, and provides examples of, the use of Matlab and Simulink with **Ejs**. The examples are not really very interesting in themselves, but are provided rather for illustration purposes and can certainly serve as a starting point for more meaningful uses of this feature.

The *_examples/externalApps/Matlab* directory of the standard distribution of **Ejs** contains a series of more interesting examples of use of Matlab and Simulink with **Ejs**.

2 Calling Matlab functions

The starting point to use Matlab with **Ejs** is to create a special page of variables for your model called an 'external page'. For this option to be accessible, you must have started **Ejs** with the *–externalApps* option (see Section 1). Next figure shows one of these pages right after creation:



The page looks very much like the standard variable page, except for the button and line that allows to provide an external Matlab M-file and for a new column labeled *Connected to*. To be more precise, we do not really need to provide an M-file to start working with Matlab, as well as we don't really need to use the new column, unless we want to *connect* our variables to Matlab variables. This connection makes more sense when working with Simulink examples and will be discussed in Section 0. For now, we just need to know that once we have created one of these external page of variables, we have immediately Matlab at our perusal.

We can create and initialize variables the same way as we do with a standard table of variables. The gate to Matlab is open through the use of a new 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 cosidered a keyword that will allow us to construct sentences of the form:

```
_ external.setValue ("t",0.1);
_ external.eval ("x = sin(t)");
_ external.getDouble ("x");
```

The first of these 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 one of our Ejs variables for this to work.

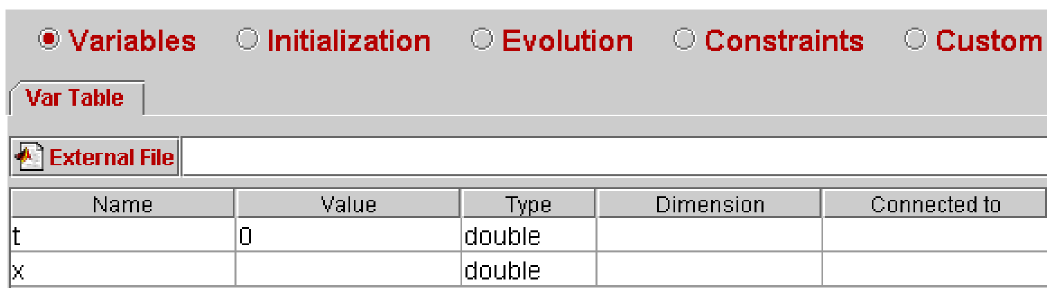
The second of the sentences evaluates in Matlab the command *"x=sin(t)"*. Since *t* has been defined and set to *0.1* in Matlab's workspace by the previous sentence, this evaluates the sine of *0.1* and defines the new Matlab variable *x*.

The last sentence retrieves the value of *x* as previously computed from Matlab's workspace. Obviously, we will need to give an use to this value for this example to be of interest.

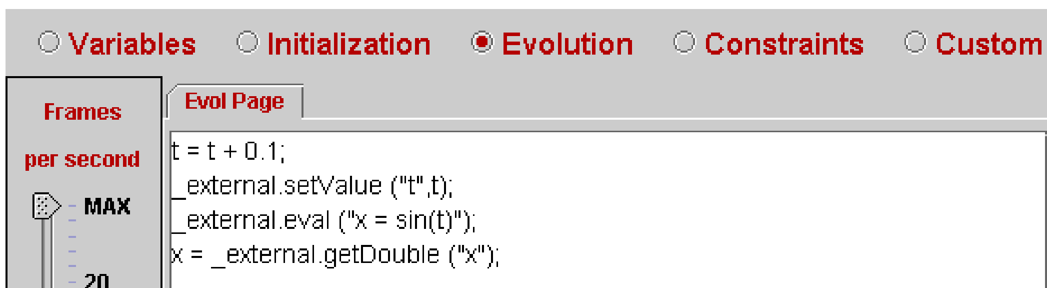
There are other constructions that can be used to access Matlab. We'll provide lists of all of them along this document.

2.1 A first example

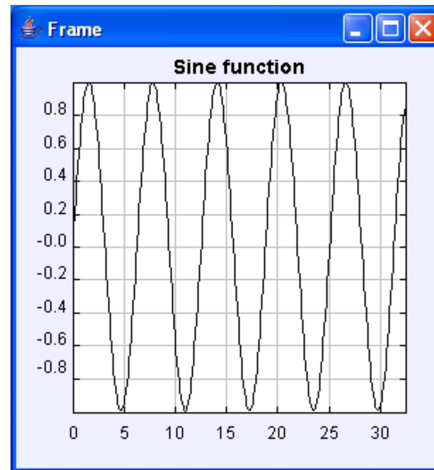
In order to give a full example with these simple sentences, we will plot the graph of the sine function using **Ejs** and Matlab. We start by declaring the following table of variables:



And then creating a single evolution page as follows:



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



2.2 A second example: using Matlab to do calculations

We can now use all the powerful mathematical features of Matlab from **Ejs**. Owners of Matlab can create advanced models very easily in **Ejs** thanks to the **Ejs**-Matlab connection. Instead of having to write the complicated Java code for many calculations, the model can simply make use of Matlab functions (either built-in or defined in an M-file).

To illustrate this situation, let's imagine a simulation that needs to obtain the roots of the following fourth-order polynomial whose coefficients are user defined:

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

And, to make it even more interesting (i.e. complicated ☺), we need to separate 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 by using the function *roots*, and the real and imaginary parts of a complex number are obtained using the functions *real* and *imag*.

We'll illustrate this example providing just the code needed to make the necessary calculations using Matlab and will assume that the rest of the simulation is ready. We don't care about the use the simulation will make of these roots, either.

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:

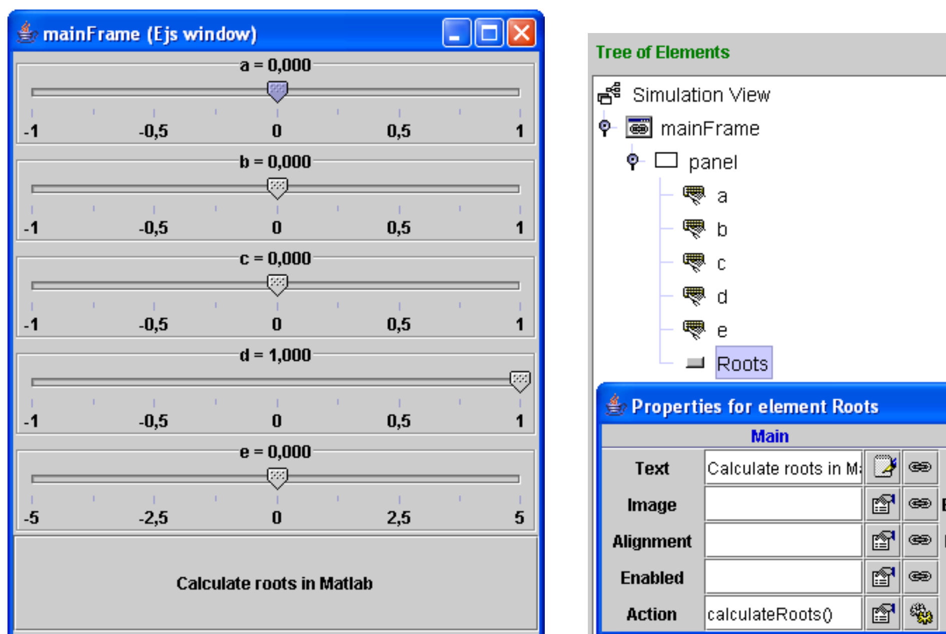
Variables | Initialization | Evolution | Constraints | Custom

Tabla Var

External File

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:

○ Variables ○ Initialization ○ Evolution ○ Constraints ◉ Custom

Zeros

```
public void calculateRoots () {
  _external.eval ("polynomial= [ " + a + " " + b + " " + c + " " + d + " " + e + "]");
  % BEGIN CODE:
  result = roots (polynomial);
  for i= 1:length(result)  reals(i)= real (result(i)); imags(i)= imag (result(i)); end
  % END CODE
  real = _external.getDoubleArray ("reals");
  imag = _external.getDoubleArray ("imags");
}
```

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 + "]");
```

sends the polymonial to 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 block contains the sentences:

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

that call the build-in Matlab function *roots* that calculates the roots of the polymonial, and separates the real and imaginary parts of the four roots, using Matlab's built-in function: *real* and *imag*. Because the roots are stored as a column vector in the variable *result*, and 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 take these values to the **Ejs** application is to use the *_external* construction *_getDoubleArray*. So, after running these two sentences:

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

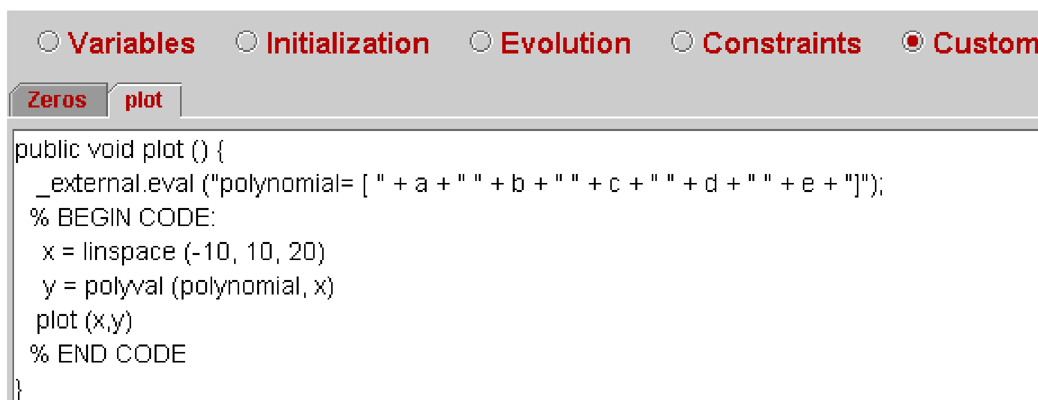
```
imag = _ external.getDoubleArray ("imags");
```

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

2.3 Using Matlab graphics

We can also use Matlab graphical features. As an example, let us plot the polynomial using the familiar Matlab function for plotting 2-D data: the *plot* function. 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)
```

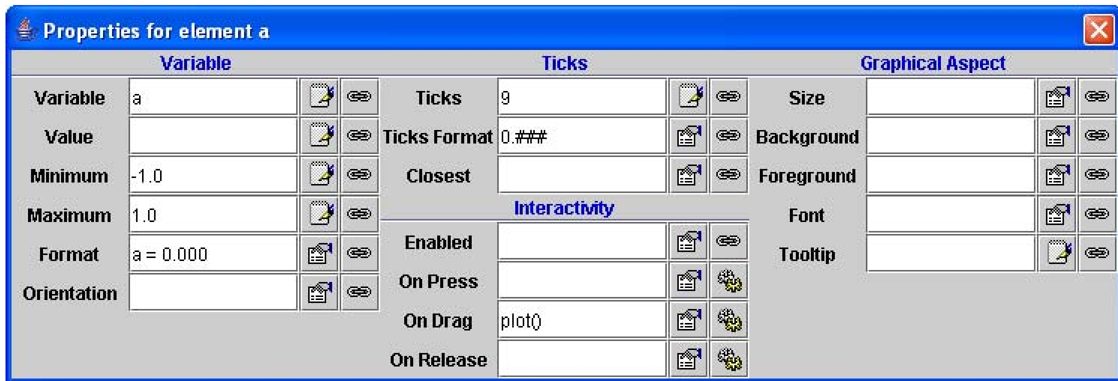
generates a row vector of 20 linearly equally spaced points between -10 and 10. We will evaluate the polynomial function in all these points. This is precisely what the next sentence does:

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

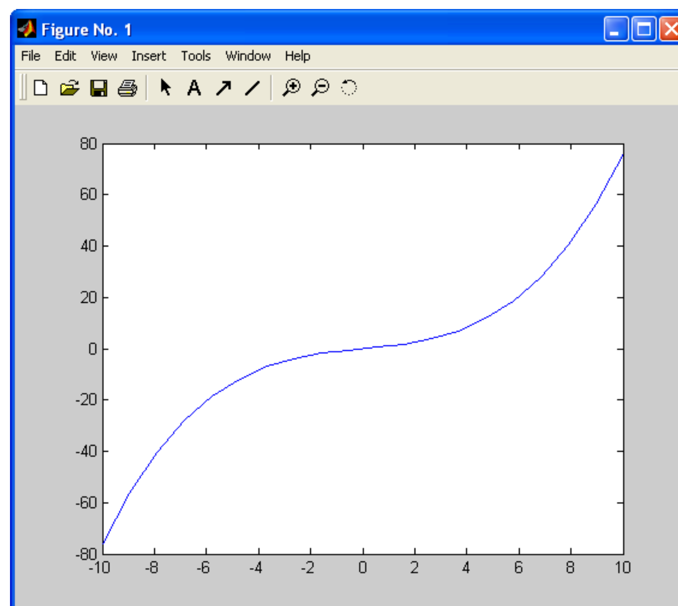
Finally, the plotting of the results is accomplished by means of the popular Matlab *plot* function:

```
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 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.



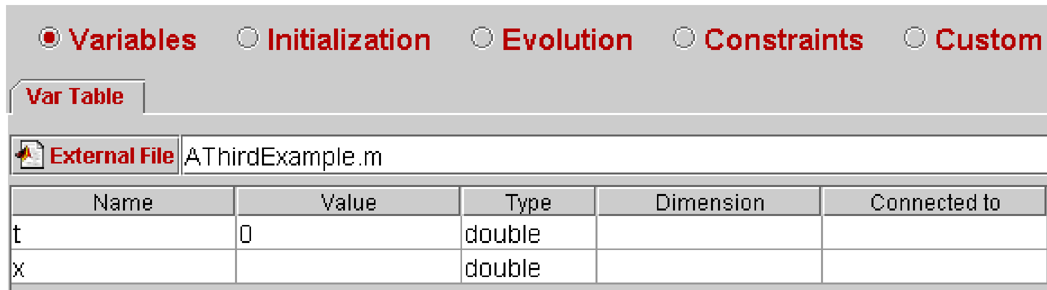
2.4 Using an M-file

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 will use this file in the *External File* textfield in the variable page.

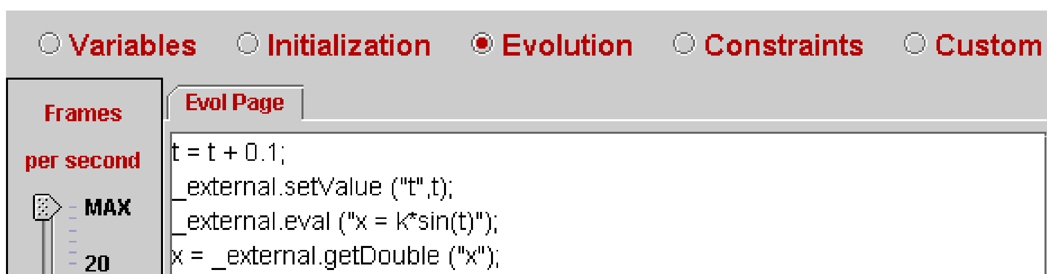
To illustrate this, let's change our example slightly. We will first create an M-file in the *Simulations* directory called, say, *AThirdExample.m* with the following single line in it.

```
k = 2.0
```

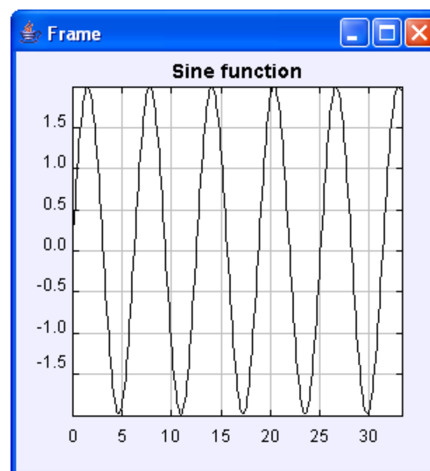
Now, we write this name in the *External File* textfield, as shown in the picture:



The consequence is that, after start-up, the M-file *AThirdExample.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.



2.5 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.

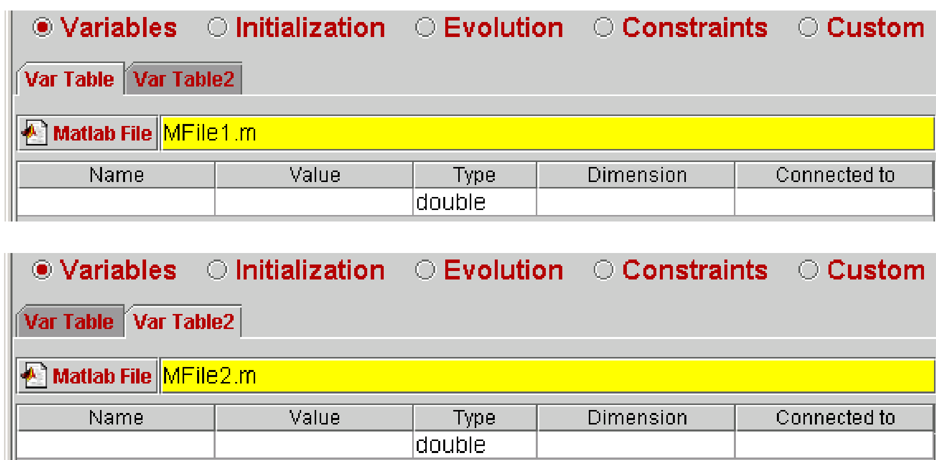
Notice that the M-files do not actually need to exist. This filenames can also be regarded as a way to name the Matlab's workspaces. In this case, that is, if the

M-files do not exist, the textfield 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 used *ExternalFile* text fields of the external variable pages.

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



Now, if we use the sentences

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

We will get two warning messages 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* textfield 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 the *setValue* commands. The *getValue* command will return however the first value it finds.

- If two pages of variables specify the same M-file (this is frequently the case!) 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 characterizes the Matlab session you want to execute the code.

2.6 A first list of *_external* constructions

We are now ready to list all the methods that can be used in *_external* and block of code constructions for the purposes we have seen so far. We leave for 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 *_command* 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 *_variable* to the value indicated in the corresponding Matlab's workspace. *_value* 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 *_variable* from the corresponding Matlab's workspace. The return type is String.
double getDouble (String _variable) double getDouble (String _mFile, String _variable)	Gets the value of the double or int variable *_variable* from the corresponding Matlab's workspace. The return type is double.
double[] getDoubleArray (String _variable) double[] getDoubleArray (String _mFile, String _variable)	Gets the value of the 1D array of doubles *_variable* from the corresponding Matlab's workspace.
double[][] getDoubleArray2D (String _variable) double[][] getDoubleArray2D (String _mFile, String _variable)	Gets the value of the 2D array of doubles *_variable* from the corresponding Matlab's workspace.
% BEGIN CODE: % BEGIN CODE: anMFile % END CODE	Indicates the beginning of raw Matlab code. Indicates the beginning of raw Matlab code for the session indicated. Indicates the end of raw Matlab code.

3 Using Simulink models

Using Simulink models from **Ejs** is possible and simple, although it requires some preliminary work. The steps needed to use Simulink models from **Ejs** are:

1. Make minor changes to the Simulink model so that it communicates properly with Matlab's workspace.

2. Create a simple Matlab M-file that informs **Ejs** about the model to be used and the variables and parameters that can be accessed in it, and also acts as communication mediator.

3. Link **Ejs** variables to the corresponding Matlab's variables.

4. Include in **Ejs** model the necessary calls to methods that control the execution of the Simulink model.

We will illustrate these steps using a simple example taken from the set of Simulink demos included in the distribution of Matlab. The example chosen is the *Simple pendulum simulation* included in the *General* section of Simulink's demos. The corresponding file is called *simppend.mdl*.

3.1 Changes required to your Simulink model

As we have just said, in most cases, small changes need to be done to a Simulink model before it can be used from Ejs. This is because Simulink models are (obviously) created to be run within Simulink, and are therefore what we could call *self-contained*.

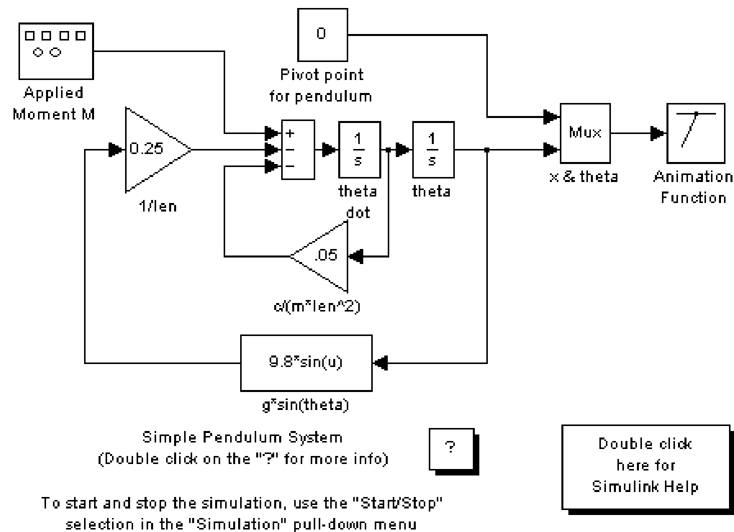
Since communication between Simulink and Ejs must go through Matlab's workspace, the first change we need to do to a model is to modify it so that it sends the value of some of its variables to Matlab's workspace. We may also want to change the model so that it receives the value of some of its variables or parameters from Matlab's workspace.

The second change comes from the fact that Simulink models are usually played and paused by the user through Simulink's user interface. We then need to change the model so that it will play exactly when Ejs tells it to play.

The final change, though this one is not mandatory, is to remove from the Simulink model any visualization of the state. Since we want to use Ejs for

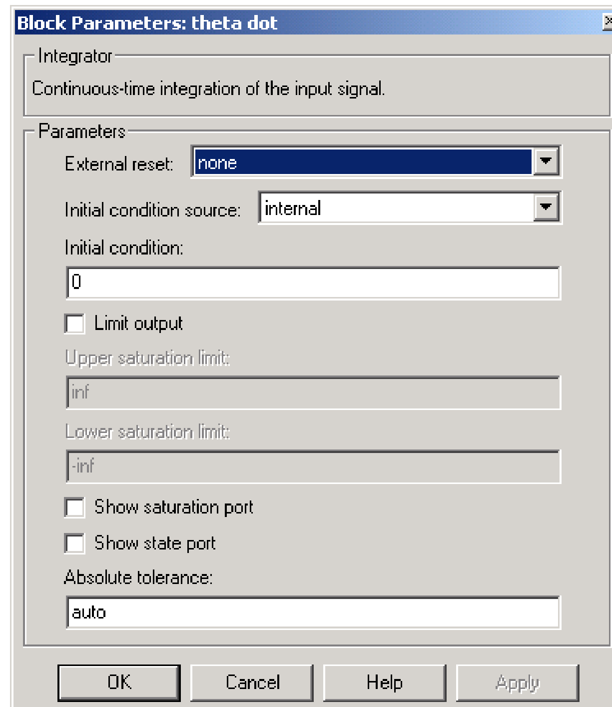
creating the view for the given model, it is usually unnecessary to keep the original visualization.

Let's do this changes to the example we have chosen, the pendulum. If we inspect the original demo model:



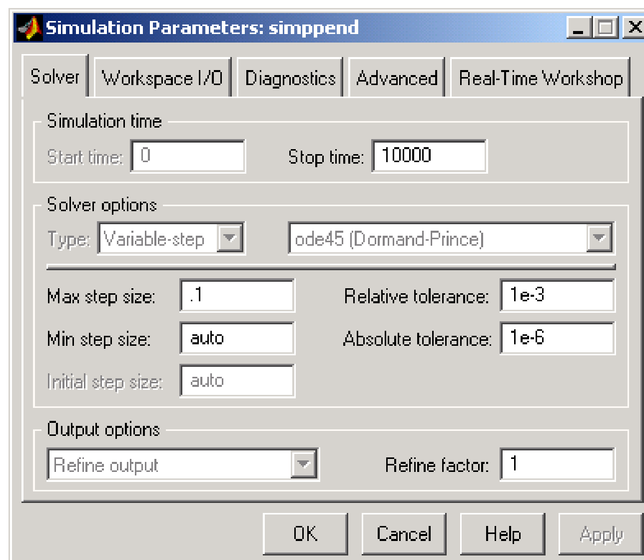
we can make more clear the changes needed. Note that:

a) the model contains the values for all variables needed to run. It is not possible to modify these values from Matlab's workspace since they have been hard coded in the model. A good example of variables that we may want to change are the initial conditions for the pendulum which are hard-coded inside the corresponding integrator blocks. The figure below shows the parameter dialog for the *theta dot* (angular velocity) integrator block:



Notice also that, the way the model is written right now, it is not possible to read the values of these variables from Matlab's workspace.

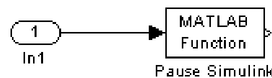
b) in Simulink, once the model is started, it plays until the prescribed *Stop time* as indicated in the Simulation parameter dialog, below. In our case, this is *10000* seconds.



c) the demo includes a visualization of the phenomenon using an animation function (which in turns requires a multiplexer *Mux*).

We will make our changes proceeding from last to first. To complete point c), we can just remove the animation function, the multiplexer and the *0* constant value, since we will visualize the phenomenon using a view created with Ejs.

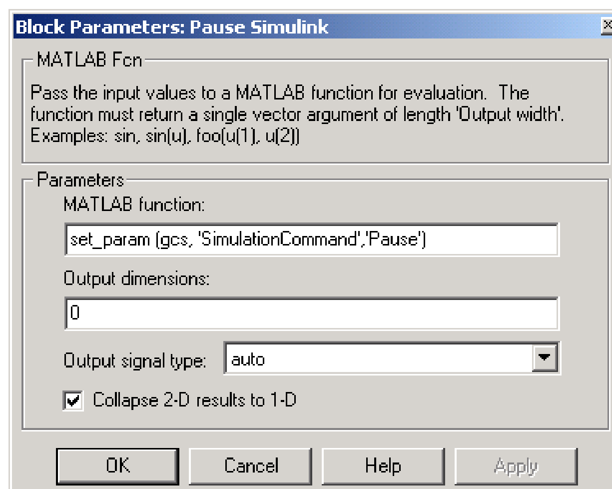
For point b), we will just add to our model the following construction:



where the first block is an *In1* input port block (category of *Signals and Systems* in Simulink's library) and the second is a *Matlab Fcn* block (category of *Functions & Tables*) with its function parameter set to is given by

```
set_param (gcs, 'SimulationCommand','Pause')
```

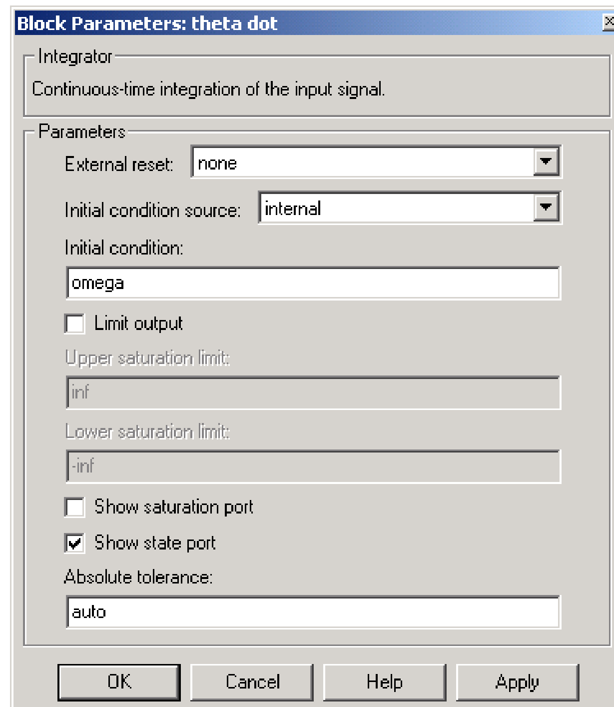
See the block parameter dialog below.



This construction has the effect of making the model to play only once every time it is instructed to run. Later on, we will control this execution from Ejs model.

Point a) can be more laborious, since we need to modify the model for each of the variables that we want to access from Ejs (through Matlab's workspace). For our demo, we will just concentrate on three variables, the time, the angle and the angular velocity.

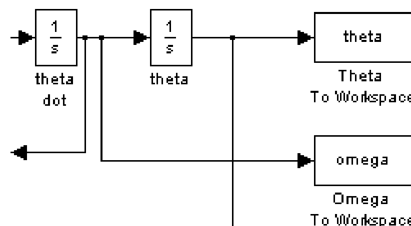
We first need to modify the parameters of the integrator blocks so that they use a given variable from Matlab's workspace as initial condition. We do this first for the *theta dot* integrator block. We edit the *Initial condition* text field and write there a new variable which we call *omega*. The dialog looks now as shown below.



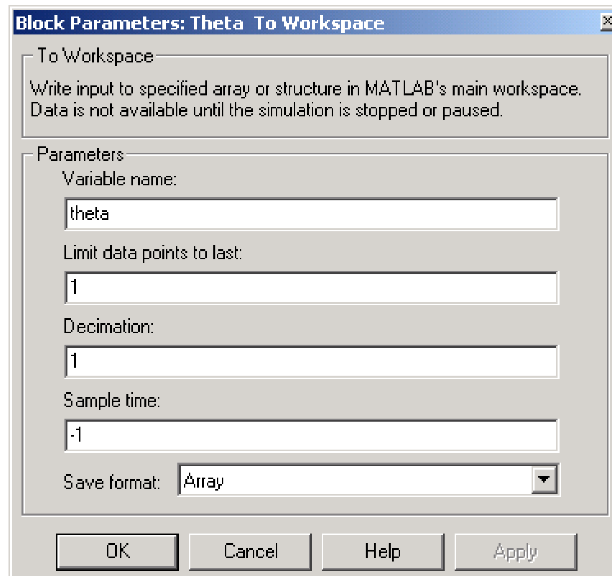
We must do the same thing with the *theta* integrator block. There we will edit the *Initial condition* text field to read a new variable called *theta*. This will provide the input to our model.

To be able to read the output, we need to create for each of these variables a construction that reads the value from wherever it is produced and takes it to Matlab's workspace.

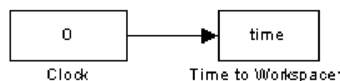
For the angle and the angular velocity we can do this reading the value from the output of the corresponding integrator block and taking it to the workspace using a *To Workspace* block (this block is found in the *Sinks* category in the Simulink library). The corresponding constructions can be seen in the following detail extracted from the whole final model.



The parameters for the *Theta to Wokspace* block can be seen below. Similar parameters do the work for the angular velocity *omega*.

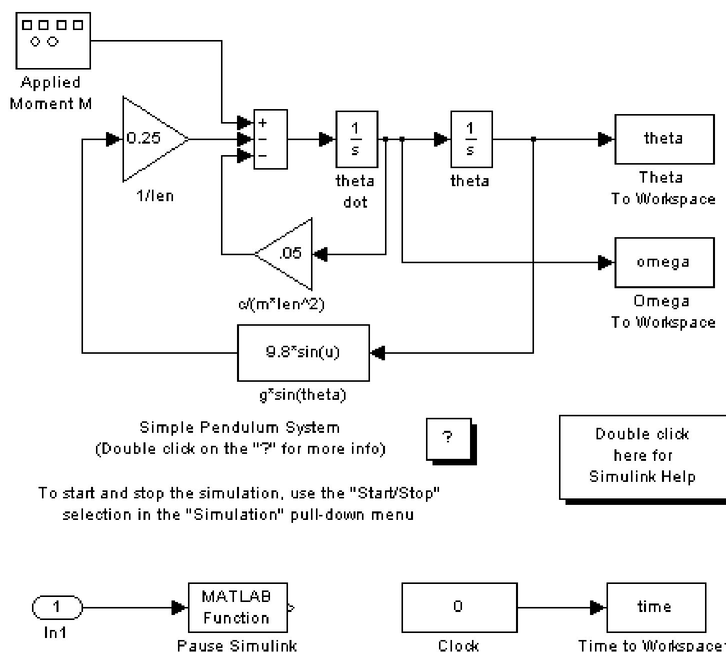


Finally, for time we need to add a construction that will allow us to read it from Matlab's workspace. This construction has the following form:



The first block reads the simulation time. The second is an output block similar to the two ones we have added above. Here we have introduced the new variable *time*.

The model of our example is ready to be used from Ejs. We can have a look now at the final model. You can find this model in the file *simppend.mdl* included with the examples bundled with Ejs.



3.2 Creation of a M-file to serve as interface

The next step in our process is to create an M-file that can be used by Ejs to extract the information it needs for its internal mechanisms. Creating such an M-file is rather straightforward. The file needs to have an entry for each of the variables we want to access plus one for the model itself.

The M-file can contain other lines. These will be executed before playing the Simulink model, hence extra lines can serve for other initialization purposes.

For our example the file reduces to the following lines:

```
theta = 0;                    %Ejs Variable
omega = 10;                   %Ejs Variable
time = 0;                     %Ejs Variable
model='simppend.mdl';  %Ejs Model
```

The syntax for this file is rather easy. You need to define and initialize the value for each of the variables, as if you were going to use this file as a Matlab command. In fact, this file will be executed before playing the Simulink model. But what enables Ejs to read and access this variable is a special comment at the end of the line. This comment must start with the word *Ejs* immediately after the begin-of-comment character '*%*'.

What follows the '*%Ejs*' keyword must start with one of the following words: *Model*, *Variable* or *Parameter*. The meaning of each of these keywords is explained below.

- *Model*. Exactly one line must include a variable of type String which is initialized to the name of the file which contains the Simulink model. This must be commented to be '*%Ejs model*'. The file must be specified relative to the location of the M-file. In our example, our M-file (which will be called *simppend.m*) and the Simulink file *simppend.mdl* are situated in the same directory. The name used for the model variable is irrelevant.

- *Variable*. This tells Ejs that this is a regular variable of the model. A variable can be read and modified, however the following optional modifiers apply:

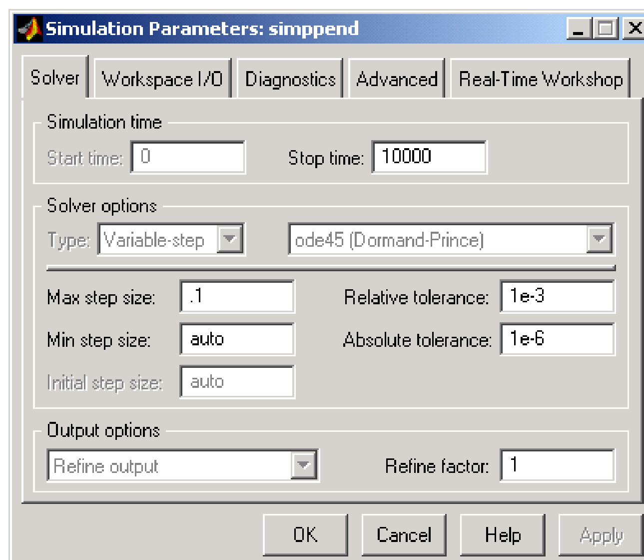
    - *InputOnly*. This tells Ejs that this variable can be set to any given value, but that the Simulink model will not modify it. That is, this is a constant for the model.

    - *OutputOnly*. This tells Ejs that this variable is a read-only variable. That is, any change we try to give to its value will be ignored.

These modifiers are not required at all. However, correctly including them can make the final simulation more efficient.

- *Parameter*. Paremeters are variables defined inside Simulink blocks. Ejs makes no distinction between variables and parameters for computation uses. However, for technical reasons, the comment for a parameter must include information about the block in which the parameter exists and the function of this parameter within the block.

  As an example, suppose that we want to change the increment of time for each simulation step in our example to be *0.01*. We must notice that this increment has been established to be *0.1* in the Simulation parameter dialog:

  We would need to modify the entry *Max step size* in this dialog to read, say *deltaTime*, and include in our M-file the following line:

  ```
  deltaTime= 0.01;      %Ejs Parameter=maxstep
  ```

  This works for parameters of the simulation itself. If the parameter belongs to a given block, the name of the block must be included separated by a colon ':' characters. As in

  ```
  MyParameter=1.0;     %Ejs Parameter=myBlock:parameterName
  ```

  Subblocks can be specified within the block field, separating them by a slash '/' character. As in

  ```
  MyParameter=1.0; %Ejs Parameter=block/subBlock1/subBlock2:paramName
  ```

  The internal names that blocks give to their parameters can be consulted reading the entries in the *mdl* file corresponding to this block.

  Parameters can also be qualified by the optional modifiers *InputOnly* and *OutputOnly*, similarly to Variables.

IMPORTANT NOTE: Parameters can only be changed before actually running a model. If we change a parameter when the model is running, the value of the parameter changes only in Matlab's workspace, but NOT in the Simulink model.

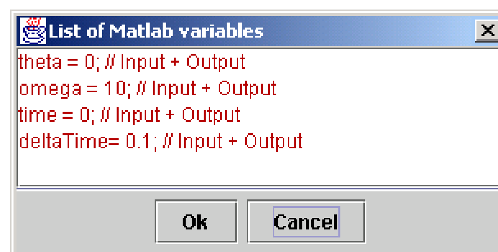
In order for this change to be properly reflected in the Simulink model, we must issue a call in Ejs to the predefined simulation method *_initialize()* whenever we change a parameter. This, of course, initializes the system as described in chapter 5 of the manual.

(Note: Read below about how to change the initial condition parameter of an integrator block during run-time.)

3.3 Connecting variables in Ejs with Matlab variables

Once we have created the M-file associated to our modified Simulink model, we are ready to connect the variables defined in Ejs with variables of Matlab workspace (that will in turn be associated to variables of the Simulink model).

Run Ejs with Matlab option turned on, create a page of Matlab variables and select in its *External File* text field the M-file that we have created for our Simulink model. Now, whenever you create a variable and right-click on it, you will be given a list of the suitable variables defined in the M file to which you can *connect* your variable. See the figure for the list in our example.

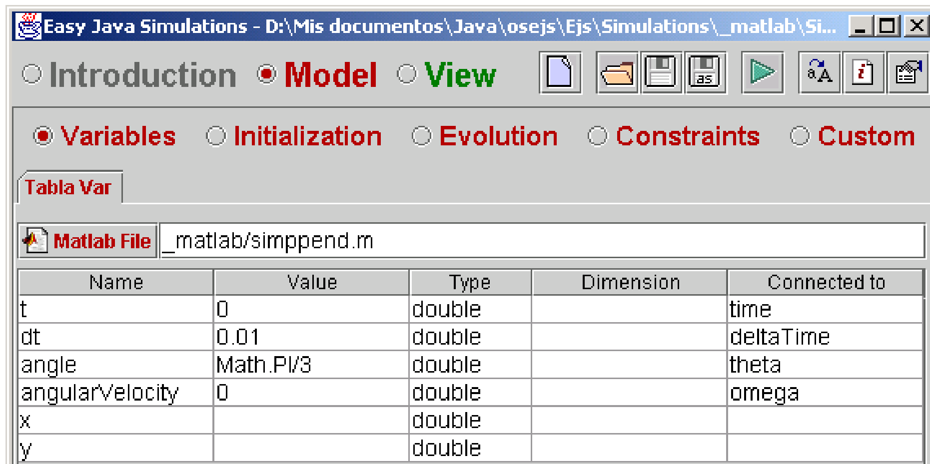


Click on any of these variables to establish a connection between your Ejs variable and the Simulink variable. Establishing a connection means exactly:

a) that the value of your Ejs variable will be pushed to the variable of the model before running the Simulink model and

b) that the value of the model variable will be given back to your Ejs variable after running the Simulink model.

For our example we need to creat the following table of variables:



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

A final remark about the initial values of the variables. In run-time, the variables are given initially the value specified in the corresponding entry of the Ejs table of variables. This means that whatever value is specified in the M-file for the connected variable will be superseeded by the value prescribed in Ejs table. This is to ensure that Ejs is always in control of the whole simulation.

3.4 Playing the Matlab simulation

After all our preparatory work, we can run our Simulink model by simply including the line

```
_external.step(1);
```

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 will have the following effects:

1) push the values of any Ejs variable which is connected to a Matlab variable (except for those which are declared to be *OutputOnly* in the M-file)

2) run the Simulink model once (due to the parameter 1)

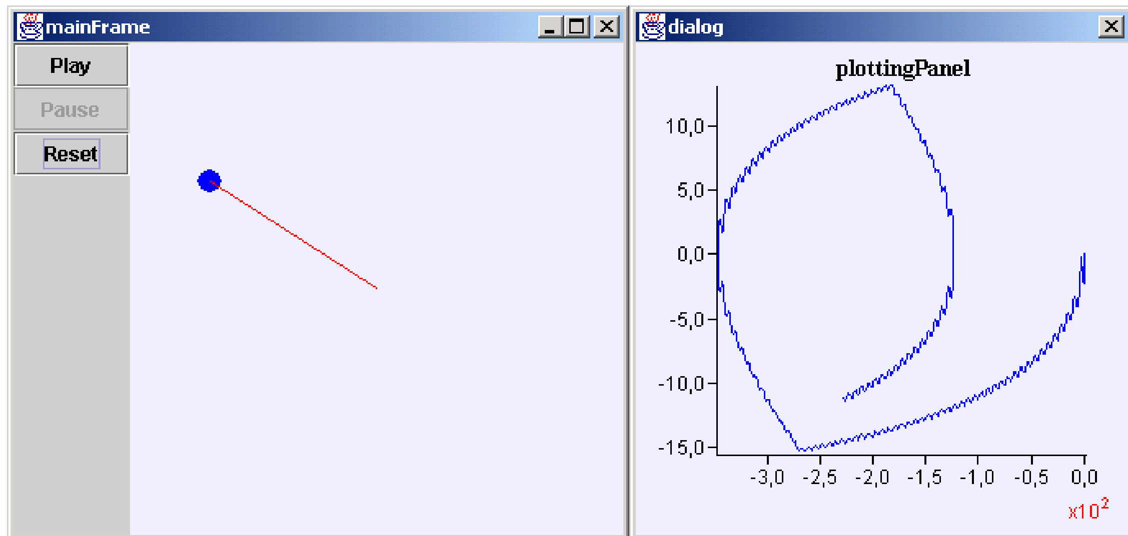
3) retrieve the value of the Matlab variables which are connected to Ejs variables (except for those which are declared to be *InputOnly* in the M-file).

A second form of the *step* method allows to play the Simulink model more than once. The correct way to use this method is

```
_external.step (n)
```

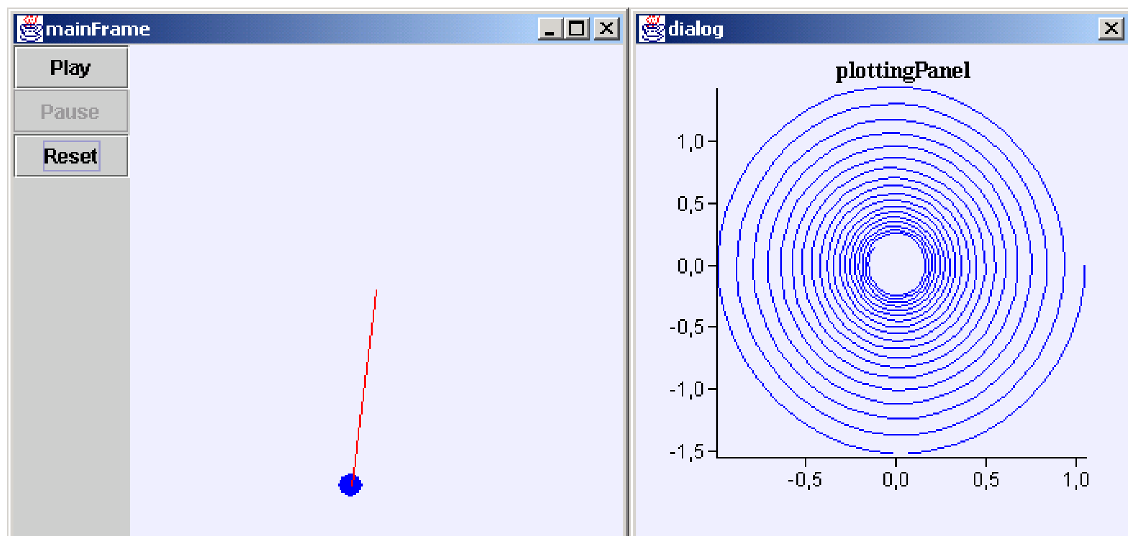
where *n* can be any positive integer. This has the same effects as before except that the Simulation plays exactly *n* times.

If we use this method and run our example, now complete, we would get the following output:



This example is included in the distribution of Ejs under the name *SimpPend.xml*.

Of course, this unfamiliar phase-space diagram is due to the fact that the pendulum is being excited by an external force. If we modify the Simulink model to delete this external force, we will get the more familiar phase-space diagram of a damped pendulum.



3.5 Using more than one Simulink model

It is possible to run more than one Simulink model 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 a suitable M-file, and connect your Ejs variables to the corresponding Matlab variables.

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

When asking Matlab to play the Simulink models, you can still use the *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 (“myMfile”,1)
```

where the specified string must match exactly one of the M-files indicated in the textfields of the Matlab variable pages.

Finally, a fourth form exists, given by

```
_external.step (String Mfile, double n)
```

which runs the model specified by *Mfile* exactly *n* times.

A final note: As we have said before, the call to any of the *step* methods 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*. These is rarely needed, but the methods are provided for completeness. These methods are referenced in the table below.

3.6 Modifying variables and parameters in run-time.

Write here how to set the model so that variables can be read from the workspace in run-time and not only before running the model…

Special Issue: how to change the initial condition parameter of an integrator block during run-time

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
void setValues (String _mFile)	Sets the value of all connected Matlab variables to that of the corresponding Ejs variables. Not to be used directly by users
void getValues (String _mFile)	Gets the values of all connected Matlab variables and gives them to the corresponding Ejs variables. Not to be used directly by users
void step (double _n) void step (String _mFile, double _n)	Plays all the Simulink models _n times Plays the given Simulink mode _n 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 setValues and getValues methods.
void reset () void reset (String _mFile)	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