

---

---

# Modeling Science: From Free Fall to Chaos

---

Wolfgang Christian  
and  
Francisco Esquembre





---

## Chapter Two

### Introduction to Easy Java Simulations

A good example is the best sermon. *Benjamin Franklin*

This chapter provides an overview of *Easy Java Simulations* (*EJS* for short), the high-level modeling and authoring tool that we will use to make our models explicit and to run these models to study their behavior. To provide a perspective of the modeling process, we first load, inspect, and run an existing simple harmonic oscillator simulation. We then modify the simulation to show how *EJS* engages the user in the modeling process and greatly reduces the amount of programming that is required.

#### 2.1 ABOUT Easy Java Simulations

Computer modeling is intimately tied to computer simulation. A model is a conceptual representation of a physical system and its properties and modeling is the process whereby we construct this representation. Computer modeling requires (1) a description and an analysis of the problem, (2) the identification of the variables and the algorithms, (3) the implementation on a specific hardware-software platform, (4) the execution of the implementation and analysis of the results, (5) refinement and generalization, and (6) the presentation of results. A computer simulation is an implementation of a model that allows us to test the model under different conditions with the objective of learning about the model's behavior. The applicability of the results of the simulation to those of the real (physical) system depends on how well the model describes reality. The process of devising more general and more accurate models is what science is about.

The implementation of a model and the visualization of its output requires that we program a computer. Programming can be fun, because it gives us complete control of every visual and numerical detail of the simulated world. But programming is also a technical task that can intimidate. This technical barrier can, however, be lowered if we use an appropriate tool. *Easy Java Simulations* is a modeling tool that has been designed to

allow scientists, not only computer scientists, to create simulations in Java. *EJS* simplifies this task, both from the technical and from the conceptual point of view.

*EJS* provides a simple yet powerful conceptual structure for building simulations. The tool offers a sequence of workpanels which we use to implement the model and its graphical user interface. *EJS* automates tasks such as numerically solving ordinary differential equations, and animation (using Java threads). The low-level communication between the program and the end-user that takes place at run-time, including handling of mouse actions within the simulation's graphical interface, is accomplished without low-level programming.

Obviously, part of the task still depends on us. You are responsible for providing a model for the phenomenon and for designing and selecting an output view that shows the model's main features. These high-level tasks are more related to science than to programming. You are encouraged to devote your time and energy studying the science, something that the computer cannot do. The purpose of this chapter is to demonstrate that this computer modeling is not only possible but can be relatively easy, with the help of *Easy Java Simulations*.

## 2.2 INSTALLING AND RUNNING THE SOFTWARE

Let us begin by installing *Easy Java Simulations* and running it. *EJS* is a Java program that can be run under any operating system that supports a Java Virtual Machine (VM). Because Java is designed to be platform independent, the *EJS* user interface on Mac OS X, Unix, and Linux is almost identical to the Windows interface shown in this book.

To install and run *EJS*, do the following:

1. **Install the Java Runtime Environment.** *EJS* requires the Java Runtime Environment (JRE), version 1.5 or later. The JRE may already be installed in your computer, but, if not, use the copy provided on the CD that comes with this book or, even better, visit the Java site at <http://java.sun.com> and follow the instructions there to download and install the latest version.
2. **Copy *EJS* to your hard disk.** You'll find *EJS* in a compressed ZIP file called something like **EJS\_X.x\_yymmdd.zip** on the CD of this book or, again better, download the latest release from *EJS* web site <http://www.um.es/fem/Ejs>. Here, the X.x characters stand for the actual version of the software, and yymmdd stands for the date this version was created. (For instance, you can get something like

**EJS\_4.2\_090901.**) Uncompress this file on your computer's hard disk to create a directory called **EJS\_X.x** (**EJS\_4.2** in the example). This directory contains everything that is needed to run *EJS*.<sup>1</sup>

3. **Run the *EJS* console.** Inside the newly-created **EJS\_X.x** directory, you will find a file called **EjsConsole.jar**. Double-click it to run the *EJS* console shown in Figure 2.1.

If double-clicking doesn't run the console, open a system terminal window, change to the **Ejs** directory, and type the command: `java -jar EjsConsole.jar`. You'll need to fully qualify the `java` command if it is not in your system's `PATH`.

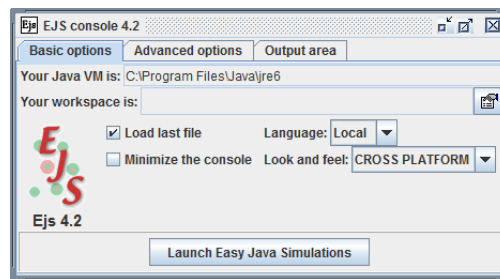


Figure 2.1: The *EJS* console.

You should see the console (Figure 2.1) and the file chooser dialog of Figure 2.2, that we will describe below, on your computer display.

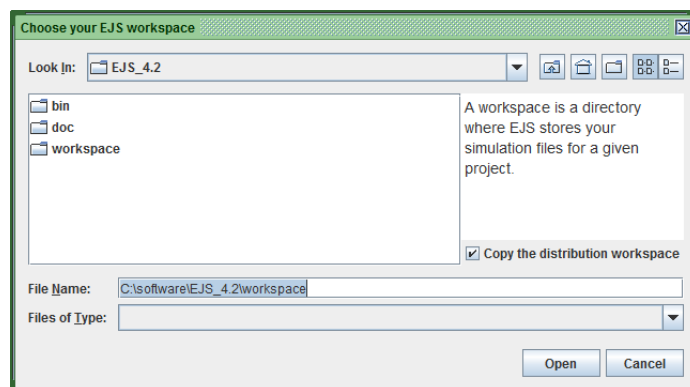


Figure 2.2: File chooser to select your workspace directory.

The *EJS* console is not part of *EJS*, but a utility used to launch one or several instances (copies) of *EJS* and to perform other *EJS*-related tasks. You can use the console to customize some aspects of how *EJS* looks and behaves at start up. (For instance, we changed the selection of the *Look*

<sup>1</sup>In Unix-like systems, the **EJS\_X.x** directory may be uncompressed as read-only. Enable write permissions for the **EJS\_X.x** directory and all its subdirectories.

and feel field to *Nimbus*, the latest look and feel for Windows platforms, and launched a new instance of *EJS* to apply the change. You will appreciate the new look and feel in subsequent figures.) The console also displays *EJS* program information and error messages on its *Output area* tab, and we will refer to it from time to time in this book. The console creates an instance of *EJS* at start-up and exits automatically when you close the last running instance of *EJS*. Other console features, such as its ability to process collections of *EJS* models, are described in the appendices.

However, before the console can run *EJS* right after installation, the file chooser displayed in Figure 2.2 will appear, letting you select the directory in the computer hard disk that you will use as your *workspace*. *EJS* uses the concept of a workspace to organize your work. A workspace is a directory in your hard disk where *EJS* stores your simulation files for a given project. A workspace can contain an unlimited number of simulations. Inside a workspace directory, *EJS* creates four subdirectories:

- **config** is the directory for user-defined configuration and options files.
- **export** is the proposed target directory when *EJS* generates files for distribution.
- **output** is the directory used by *EJS* to place temporary files generated when compiling a simulation.
- **source** is the directory under which all your simulation (source and auxiliary) files must be located.

When you first run *EJS*, the console asks you to choose a workspace directory. This must be a writable directory anywhere in your hard disk. You can choose to use the workspace included in the distribution, i.e. the **workspace** directory in the **EJS X.x** directory created when you unzipped the *EJS* bundle. But it is recommended to create a new directory in your usual personal directory. The file dialog that allows you to choose the workspace has a check box that, when checked, will copy all the examples files of the distribution to the new workspace. Leave this check box checked and you will find some subdirectories in the **source** directory of your workspace which contain sample simulations. In particular, the **ModelingScience** directory includes the *EJS* models described in this book.

You can create and use more than one workspace for different projects or tasks. The console provides a selector to let you change the workspace in use and *EJS* will remember the current workspace between sessions or even if you reinstall *EJS*. Appendix A describes how to configure and use *EJS* in a multiuser installation.

Finally, the first time you run *EJS*, the program will also ask you to

introduce your name and affiliation (Figure 2.3). This step is optional but recommended, since it will help you document your future simulations. You can choose to input or modify this information later using the options icon of *EJS*' task bar.

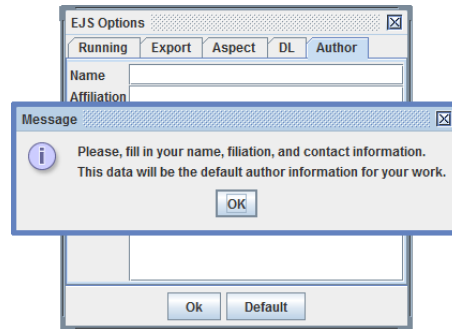


Figure 2.3: Optionally input your name and affiliation.

We are now ready to turn our attention to the *EJS* modeling tool, displayed with annotations in Figure 2.4 (our first image with the Nimbus look and feel). Despite its simple interface, *EJS* has all the tools needed for a complete modeling cycle.

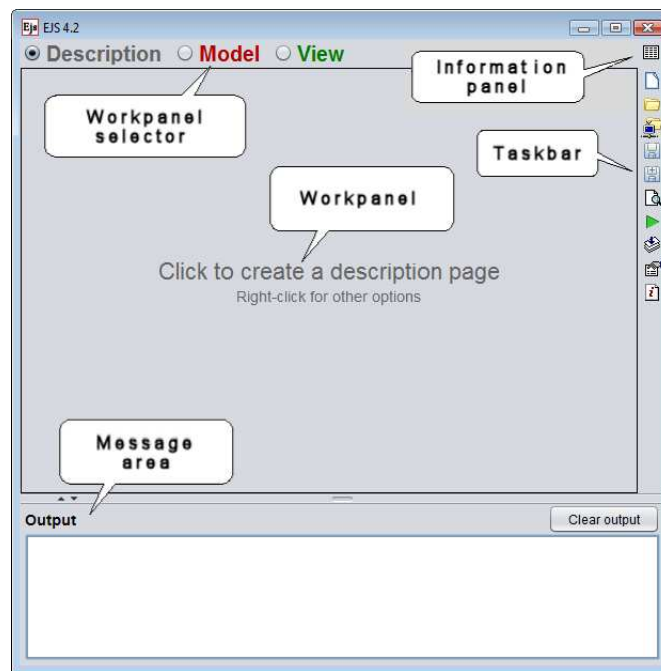


Figure 2.4: The *Easy Java Simulations* user interface with annotations.

The taskbar on the right provides a series of icons to clear, open,



search, and save a file, configure *EJS*, and display program information and help. It also provides icons to run a simulation and to package one or more simulations in a jar file. Right-clicking on taskbar icons invokes alternative (but related) actions that will be described as needed. The bottom part of the interface contains an output area where *EJS* displays informational messages. The central part of the interface contains the workpanels where the modeling is done.

*Easy Java Simulations* provides three workpanels for modeling. The first panel, *Description*, allows us to create and edit multimedia HTML-based narrative that describes the model. Each narrative page appears in a tabbed panel within the workpanel and right-clicking on the tab allows the user to edit the narrative or to import additional narrative. The second work panel, *Model*, is dedicated to the modeling process. We use this panel to create variables that describe the model, to initialize these variables, and to write algorithms that describe how this model changes in time. The third workpanel, *View*, is dedicated to the task of building the graphical user interface, which allows users to control the simulation and to display its output. We build the interface by selecting elements from palettes and adding them to the view's *Tree of elements*. For example, the *Interface* palette contains buttons, sliders, and input fields and the *2D Drawables* palette contains elements to plot 2D data.

## 2.3 INSPECTING THE SIMULATION

To understand how the *Description*, *Model*, and *View* workpanels work together, we inspect and run an already existing simulation. Screen shots are no substitute for a live demonstration, and you are encouraged to follow along on your computer as you read.

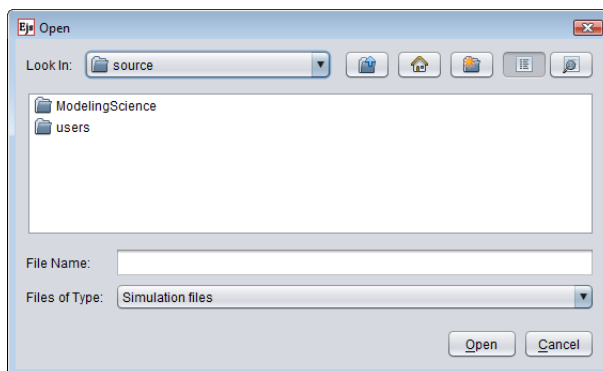



Figure 2.5: The open file dialog lets you browse your hard disk and load an existing simulation.

Click on the *Open* icon  on the *EJS* taskbar. A file dialog similar to that in Figure 2.5 appears showing the contents of your workspace's **source** directory. Go to the **ModelingScience** directory, and open the **Ch02\_Intro** subdirectory. You will find a file called **MassAndSpring.xml** inside this directory. Select this file and click on the *Open* button of the file dialog.

Now, things come to life! *EJS* reads the **MassAndSpring.xml** document which populates the workpanels and two new “Ejs windows” appear in your display as shown in Figure 2.6. A quick warning. You can drag objects within these mock-up windows but this will set the model's initial conditions. It is usually better to set initial conditions using a table of variables as described in Section 2.3.2.

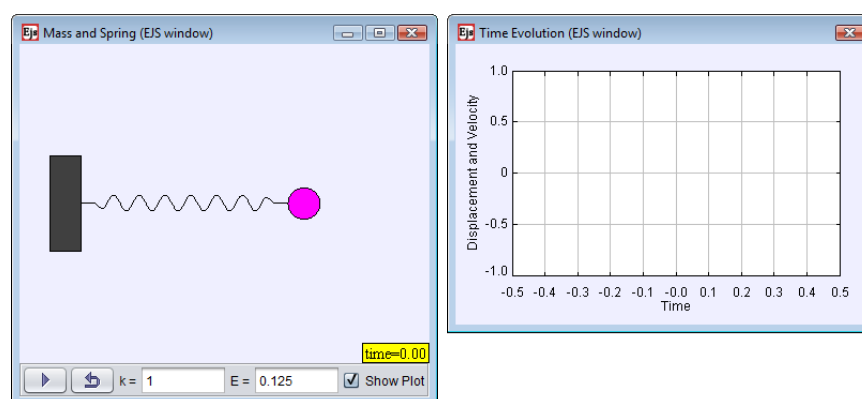




Figure 2.6: *EJS* mock-up windows of the **MassAndSpring** simulation. The title bar shows that they are Ejs windows and that the program is not running.

Impatient or precocious readers may be tempted to click on the green run icon  on the taskbar to execute our example before proceeding with this tutorial. Readers who do so will no longer be interacting with *EJS* but with a compiled and running Java program. Exit the running program by closing the *Mass and Spring* window or by right clicking on the (now) red run icon  on *EJS*' taskbar before proceeding.

### 2.3.1 The *Description* workpanel

Select the *Description* workpanel by clicking on the corresponding radio button at the top of *EJS*, and you will see two pages of narrative for this simulation. The first page, shown in Figure 2.7, contains a short discussion of the mass and spring model. Click on the *Activities* tab to view the second

page of narrative.

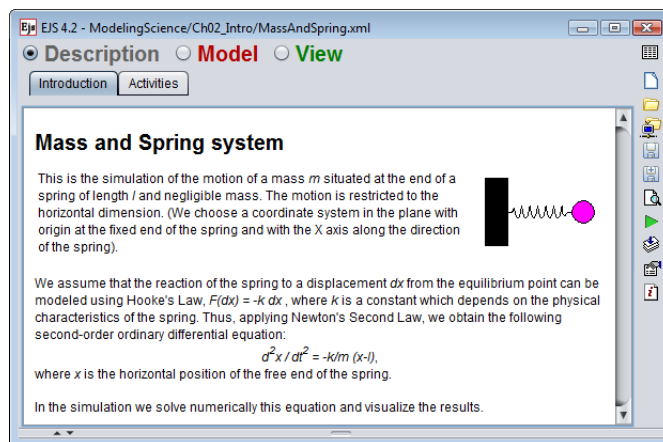


Figure 2.7: The description pages for the mass and spring simulation. Click on a tab to display the page. Right-click on a tab to edit the page.

A *Description* is HTML multimedia text that provides information and instructions about the simulation. HTML stands for HyperText Markup Language and is the most commonly used protocol for formatting and displaying documents on the Web. *EJS* provides a simple HTML editor that lets you create and modify pages within *EJS*. You can also import HTML pages into *EJS* by right clicking on a tab in the *Description* workpanel. (See Section 2.6.3.) Description pages are an essential part of the modeling process and these pages are distributed with the compiled model when the model is distributed as a Java application or posted on a Web server as an applet. These distribution options are described in Appendix A.

### 2.3.2 The *Model* workpanel

The *Model* workpanel is where the model is defined so that it can be converted into a program by *EJS*. In this simulation, we study the motion of a particle of mass  $m$  attached to one end of a massless spring of equilibrium length  $L$ . The spring is fixed to the wall at its other end and is restricted to move in the horizontal direction. Although the oscillating mass has a well known analytic solution, it is useful to start with a simple harmonic oscillator model so that our output can be compared with an exact analytic result.

Our model assumes small oscillations so that the spring responds to a given (horizontal) displacement  $\delta x$  from its equilibrium length  $L$  with a force given by Hooke's law,  $F_x = -k \delta x$ , where  $k$  is the elastic constant of

the spring, which depends on its physical characteristics. We use Newton's second law to obtain a second-order differential equation for the position of the particle:

$$\frac{d^2 x}{dt^2} = -\frac{k}{m} (x - L). \quad (2.3.1)$$

Notice that we use a system of coordinates with its  $x$ -axis along the spring and with its origin at the spring's fixed end. The particle is located at  $x$  and its displacement from equilibrium  $\delta x = x - L$  is zero when  $x = L$ . We solve this system numerically to study how the state evolves in time.

Let's examine how we implement the mass and spring model by selecting the *Model* radio button and examining each of its five panels.

### 2.3.2.1 Declaration of variables

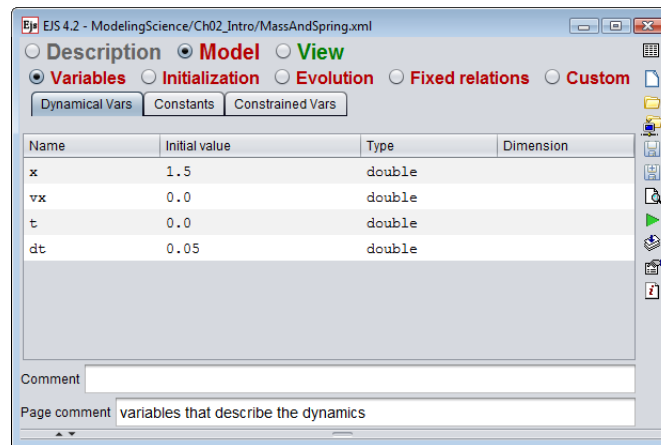


Figure 2.8: The *Model* workpanel contains five subpanels. The subpanel for the definition of mass and spring dynamical variables is displayed. Other tabs in this subpanel define additional variables, such as the natural length of the spring  $L$  and the energy  $E$ .

When implementing a model, a good first step is to identify, define, and initialize the variables that describe the system. The term *variable* is very general and refers to anything that can be given a name, including a physical constant and a graph. Figure 2.8 shows an *EJS* variable table. Each row defines a variable of the model by specifying the name of the variable, its type, its dimension, and its initial value.

Variables in computer programs can be of several types depending on the data they hold. The most frequently used types are `boolean` for true/false values, `int` for integers, `double` for high-precision ( $\approx 16$  significant

digits) numbers, and `String` for text. We will use all these variable types in this book, but the mass and spring model uses only variables of type `double` and `boolean`.

Variables can be used as parameters, state variables, or inputs and outputs of the model. The tables in Figure 2.8 define the variables used within our model. We have declared a variable for the time, `t`, for the  $x$ -position of the particle, `x`, and for its velocity in the  $x$ -direction, `vx`. We also define variables that do not appear in (2.3.1). The reason for auxiliary variables such as the kinetic, potential, and total energies will be made clear in what follows. The bottom part of the variables panel contains a comment field that provide a description of the role of each variable in the model. Clicking on a variable displays the corresponding comment.

### 2.3.2.2 Initialization of the model

Correctly setting initial conditions is important when implementing a model because the model must start in a physically realizable state. Our model is relatively simple, and we initialize it by entering values (or simple Java expressions such as `0.5*m*vx*vx`) in the *Initial value* column of the table of variables. *EJS* uses these values when it initializes the simulation.

Advanced models may require an initialization algorithm. For example, a molecular dynamics model may set particle velocities for an ensemble of particles. The *Initialization* panel allows us to define one or more pages of Java code that perform the required computation. *EJS* converts this code into a Java method<sup>2</sup> and calls this method at start-up and whenever the simulation is reset. The mass and spring *Initialization* panel is not shown here because it is empty. See Section 2.3.2.4 for an example of how Java code appears in *EJS*.

### 2.3.2.3 The evolution of the model

The *Evolution* panel allows us to write the Java code that determines how the mass and spring system evolves in time and we will use this option frequently for models not based on ordinary differential equations (ODEs). There is, however, a second option that allows us to enter ordinary differential equations, such as (2.3.1), without programming. *EJS* provides a dedicated editor that lets us specify differential equations in a format that resembles mathematical notation and automatically generates the correct Java code.

---

<sup>2</sup>A Java method is similar to a function or a subroutine in procedural computer languages.

Let's see how the differential equation editor works for the mass and spring model. Because ODE algorithms solve systems of first-order ordinary differential equations, a higher-order equation, such as (2.3.1), must be re-cast into a first-order system. We can do so by treating the velocity as an independent variable which obeys its own equation:

$$\frac{d x}{d t} = v_x \quad (2.3.2)$$

$$\frac{d v_x}{d t} = -\frac{k}{m} (x - L). \quad (2.3.3)$$

The need for an additional differential equation explains why we declared the `vx` variable in our table of variables.

Clicking on the *Evolution* panel displays the ODE editor shown in Figure 2.9. Notice that the ODE editor displays (2.3.2) and (2.3.3) (using

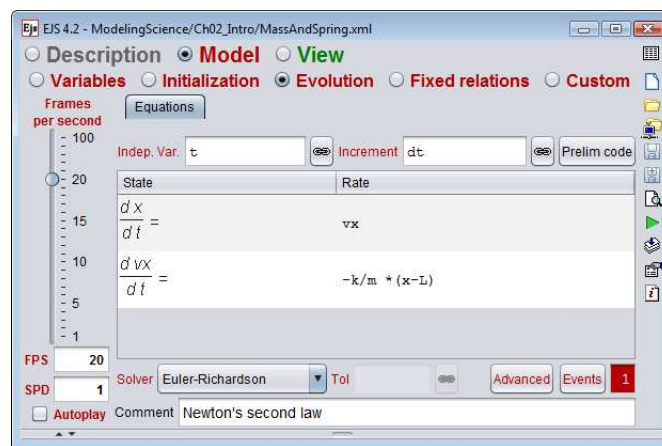


Figure 2.9: The ODE evolution panel showing the mass and spring differential equation and the numerical algorithm.

the `*` character to denote multiplication). Fields near the top of the editor specify the independent variable `t` and the variable increment `dt`. Numerical algorithms approximate the exact ODE solution by advancing the state in discrete steps and the increment determines this step size. The *Prelim* button at the top-right of the editor allows us to enter preliminary code, to perform computations prior to evaluating the equations (a circumstance required in more complex situations than the one we treat in this example). A dropdown menu at the bottom of the editor lets us select the ODE solver (numerical algorithm) that advances the solution from the current value of time, `t`, to the next value, `t + dt`. The tolerance field is greyed out because Euler-Richardson is a fixed-step method that requires no tolerance settings. The advanced button displays a dialog which allows us to fine-tune the execution of this solver, though default values are usually appropriated.

Finally, the events field at the bottom of the panel tells us that we have not defined any events for this differential equation. Examples with preliminary code and events can be found in Chapter 7. The different solver algorithms and its parameters are discussed in the *EJS* help.

The left-hand side of the evolution workpanel includes fields that determine how smoothly and how fast the simulation runs. The *frames per second (FPS)* option, which can be selected by using either a slider or an input field, specifies how many times per second we want our simulation to repaint the screen. The *steps per display (SPD)* input field specifies how many times we want to advance (step) the model before repainting. The current value of 20 frames per second produces a smooth animation that, together with the prescribed value of one step per display and 0.05 for  $\mathbf{dt}$ , results in a simulation which runs at (approximately) real time. We will almost always use the default setting of one step per display. However, there are situations where the model's graphical output consumes a significant amount of processing power and where we want to speed the numerical computations. In this case we can increase the value of the steps per display parameter so that the model is advanced multiple times before the visualization is redrawn. The *Autoplay* check box indicates whether the simulation should start when the program begins. This box is unchecked so that we can change the initial conditions before starting the evolution.

The evolution workpanel handles the technical aspects of the mass and spring ODE model without programming. The simulation advances the state of the system by numerically solving the model's differential equations using the midpoint algorithm. The algorithm steps from the current state at time  $\mathbf{t}$  to a new state at a new time  $\mathbf{t} + \mathbf{dt}$  before the visualization is redrawn. The simulation repeats this evolution step 20 times per second on computers with modest processing power. The simulation may run slower and not as smoothly on computers with insufficient processing power or if the computer is otherwise engaged, but it should not fail.

Although the mass and spring model can be solved with a simple ODE algorithm, our numerical methods library contains very sophisticated algorithms and *EJS* can apply these algorithms to large systems of vector differential equations with or without discontinuous events.

#### 2.3.2.4 Relations among variables

Not all variables within a model are computed using an algorithm on the Evolution workpanel. Variables can also be computed after the evolution has been applied. We refer to variables that are computed using the evolution

algorithm as state variables or dynamical variables, and we refer to variables that depend on these variables as auxiliary or output variables. In the mass and spring model the kinetic, potential, and total energies of the system are output variables because they are computed from state variables.

$$T = \frac{1}{2}mv_x^2, \quad (2.3.4)$$

$$V = \frac{1}{2}k(x - L)^2, \quad (2.3.5)$$

$$E = T + V. \quad (2.3.6)$$

We say that there exists *fixed relations* among the model's variables.

The *Fixed relations* panel shown in Figure 2.10 is used to write relations among variables. Notice how easy it is to convert (2.3.4) through (2.3.6) into Java syntax. Be sure to use the multiplication character `*` and to place a semicolon at the end of each Java statement.

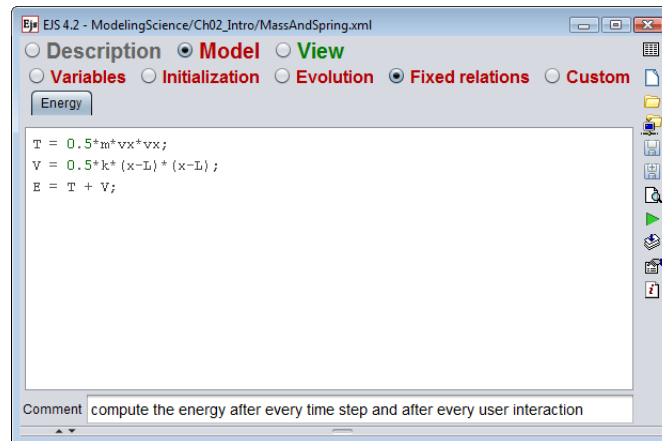


Figure 2.10: Fixed relations for the mass and spring model.

You may wonder why we do not write fixed relation expressions by adding a second code page after the ODE page in the *Evolution* panel. After all, evolution pages execute sequentially and a second evolution page would correctly update the output variables after every step. The reason that the *Evolution* panel should not be used is that relations must *always* hold and there are other ways, such as mouse actions, to affect state variables. For example, dragging the mass changes the  $x$  variable and this change affects the energy. *EJS* automatically evaluates the relations after initialization, after every evolution step, and whenever there is any user interaction with the simulation's interface. For this reason, it is important that fixed relations among variables be written in the *Fixed relations* workpanel.



### 2.3.2.5 Custom pages

There is a fifth panel in the *Model* workpanel labeled *Custom*. This panel can be used to define methods (functions) that can be used throughout the model. This panel is empty because our model currently doesn't require additional methods, but we will make use of this panel when we modify our mass and spring example in Section 2.6. A custom method is not used unless it is explicitly invoked from another workpanel.

### 2.3.3 The View workpanel

The third *Easy Java Simulations* workpanel is the *View*. This workpanel allows us to create a graphical interface that includes visualization, user interaction, and program control with minimum programming. Figure 2.6 shows the view for the mass and spring model. Select the *View* radio button to examine how this view is created.

The right frame of the view workpanel of *EJS*, shown in Figure 2.11, contains a collection of *view elements*, grouped by functionality. View elements are building blocks that can be combined to form a complete user interface, and each view element is a specialized object with an on-screen representation. To display information about a given element, click on its icon and press the *F1* key or right-click and select the *Help* menu item. To create a user interface, we create a frame (window) and add elements, such as buttons and graphs, using “drag and drop” as described in Section 2.6.

The *Tree of elements* shown on the left side of Figure 2.11 displays the structure of the mass and spring user interface. Notice that the simulation has two windows, a **Frame** and a **Dialog**, that appear on your computer screen. These elements belong to the class of *container* elements whose primary purpose is to visually group (organize) other elements within the user interface. The tree displays descriptive names and icons for these elements. Right-click on an element of the tree to obtain a menu that helps the user change this structure.

Each view element has a set of internal parameters, called *properties*, which configure the element's appearance and behavior. We can edit these properties by double clicking on the element in the tree to display a table known as a *properties inspector*. Appearance properties, such as color, are often set to a constant value, such as **RED**. We can also use a variable from the model to set an element's property. This ability to connect (bind) a property to a variable without programming is the key to turning our view

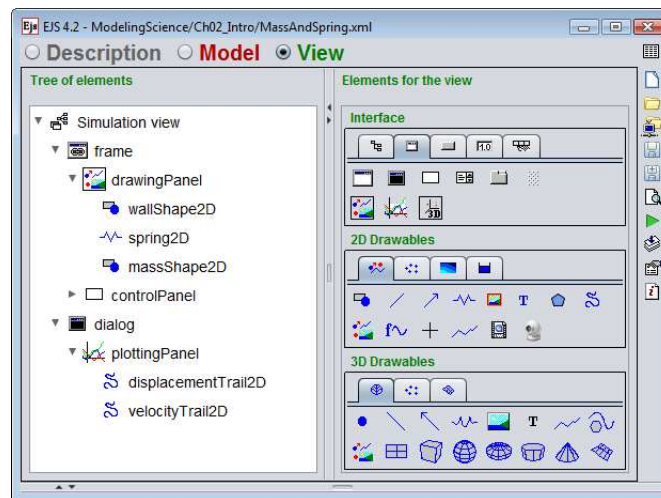


Figure 2.11: The View workpanel showing the *Tree of elements* for the mass and spring user interface.

into a dynamic and interactive visualization.

Let's see how this procedure works in practice. Double-click on the `massShape2D` element (the 'Shape2D' suffix we added to the element's name helps you know the type of the element) in the tree to display the element's properties inspector. This element is the mass that is attached at the free end of the spring. The `massShape2D`'s table of properties appears as shown in Figure 2.12.

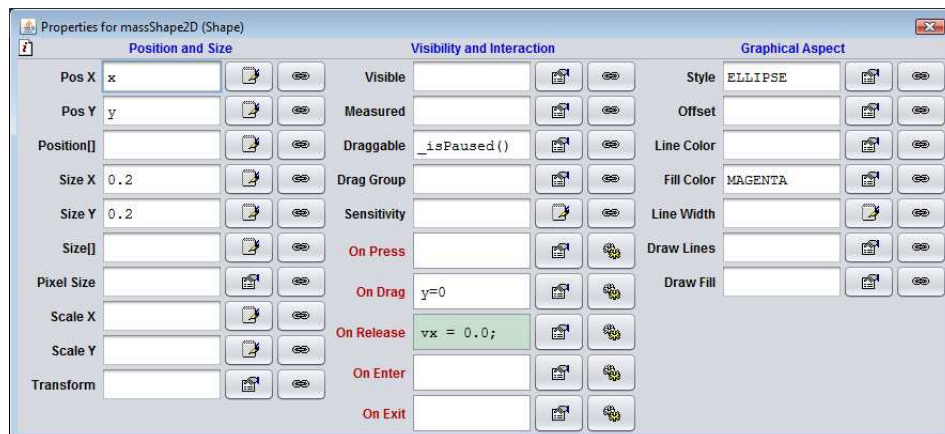


Figure 2.12: The table of properties of the `massShape2D` element.

Notice the properties that are given constant values. The **Style**, **Size**

`X`, `Size Y`, and `Fill Color` properties produce an ellipse of size `(0.2,0.2)` units (which makes a circle) filled with the color magenta. More importantly, the `Pos X` and `Pos Y` properties of the shape are bound to the `x` and `y` variables of the model. This simple assignment establishes a bidirectional connection between model and view. These variables change as the model evolves and the shape follows the `x` and `y` values. If the user drags the shape to a new location, the `x` and `y` variables in the model change accordingly. Note that the `Draggable` property is only enabled when the animation is paused.

Elements can also have *action properties* which can be associated with code. (Action properties have their labels displayed in red.) User actions, such as dragging or clicking, invoke their corresponding action property, thus providing a simple way to control the simulation. As the user drags the mass, the code on the `On Drag` property restricts the motion of the shape to the horizontal direction by setting the `y` variable to 0. Finally, when the mouse button is released, the following code is executed:

```
vx = 0.0;           // sets the velocity to zero
_view.resetTraces(); // clears all plots
```

Clicking on the icon next to the field displays a small editor that shows this code.

Because the `On Release` action code spans more than one line, the property field in the inspector shows a darker (green) background. Other data types, such as boolean properties, have different editors. Clicking the second icon displays a dialog window with a listing of variables and methods that can be used to set the property value.

### Exercise 2.1. Element inspectors

The mass' inspector displays different types of properties and their possible values. Explore the properties of other elements of the view. For instance, the `displacementTrail2D` and `velocityTrail2D` elements correspond to the displacement and velocity time plots in the second window of the view, respectively. What is the maximum number of points that can be added to each trail? □


### 2.3.4 The completed simulation



We have seen that *Easy Java Simulations* is a powerful tool that lets us express our knowledge of a model at a very high level of abstraction. When

modeling the mass and spring, we first created a table of variables that describes the model and initialized these variables using a column in the table. We then used an evolution panel with a high-level editor for systems of first-order ordinary differential equations to specify how the state advances in time. We then wrote relations to compute the auxiliary or output variables that can be expressed using expressions involving state variables. Finally, the program's graphical user interface and high-level visualizations were created by dragging objects from the *Elements* palette into the *Tree of elements*. Element properties were set using a properties editor and some properties were associated with variables from the model.

It is important to note that the three lines of code on the Fixed relations workpanel (Figure 2.10) and the two lines of code in the particle's action method are the only explicit Java code needed to implement the model. *Easy Java Simulations* creates a complete Java program by processing the information in the workpanels when the run icon is pressed as described in Section 2.4.

## 2.4 RUNNING THE SIMULATION

It is time to run the simulation by clicking on the *Run* icon of the taskbar, . *EJS* generates the Java code and compiles it, collects auxiliary and library files, and executes the compiled program. All at a single mouse click.

Running a simulation initializes its variables and executes the fixed relations to insure that the model is in a consistent state. The model's time evolution starts when the play/pause button in the user interface is pressed. (The play/pause button displays the  icon when the simulation is paused and  when it is running.) In our current example, the program executes a numerical method to advance the harmonic oscillator differential equation by 0.05 time units and then executes the relations code. Data are then passed to the graph and the graph is repainted. This process is repeated 20 times per second.

When running a simulation, *EJS* changes its *Run* icon to red and prints informational messages saying that the simulation has been successfully generated and that it is running. Notice that the two *EJS* windows disappear and are replaced by new but similar windows without the (Ejs window) suffix in their titles. These views respond to user actions. Click and drag the particle to a desired initial horizontal position and then click on the play/pause button. The particle oscillates about its equilibrium point and the plot displays the displacement and velocity data as shown in Figure 2.13.

Stop the simulation and right-click the mouse over any of the draw-

ing areas of the simulation. In the popup menu that appears, select the **Elements options->plottingPanel->Data Tool** entry to display and analyze the data generated by the model. The same popup menu offers other run-time options, such as screen capture. To exit the program, close the simulation's main window.

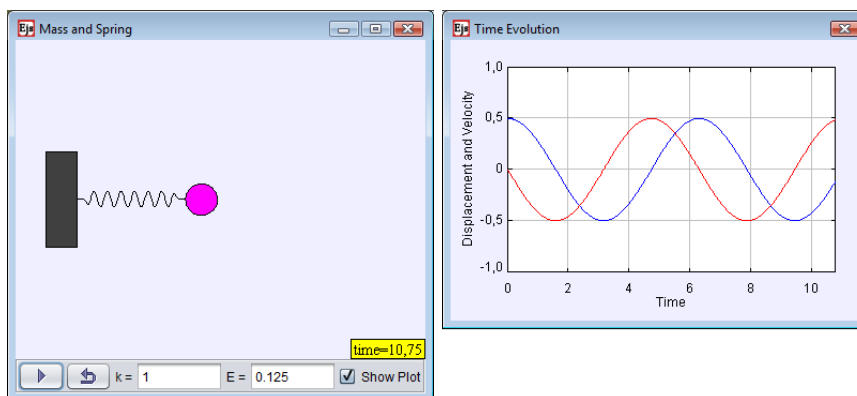



Figure 2.13: The mass and spring simulation displays an interactive drawing of the model and a graph with displacement and velocity data.

## 2.5 DISTRIBUTING THE SIMULATION

Simulations created with *EJS* are stand-alone Java programs that can be distributed without *EJS* for other people to use. The easiest way to do this is to package the simulation in a single executable jar file by clicking on the **Package** icon, . A file browser appears that lets you choose a name for the self-contained jar package. The default target directory to hold this package file is the **export** directory of your workspace, but you can choose any directory and package name. The stand-alone jar file is ready to be distributed on a CD or via the Internet. Other distribution mechanisms are available by right-clicking on the icon as described in Appendix A.

### Exercise 2.2. Distribution of a model

Click on the **Package** icon on the taskbar to create a stand alone jar archive of the mass and spring simulation. Copy this jar file into a working directory separate from your *EJS* installation. Close *EJS* and verify that the simulation runs as a stand-alone application.  $\square$

Although the mass and spring jar file is a ready to use and to distribute Java application, an important pedagogic feature is that this jar file is created in such a way that users can return to *EJS* at any time to examine, modify, and adapt the model. (*EJS* must, of course, be installed.) The jar file contains a small *Extensible Markup Language* (XML) description of

each model and right clicking on a drawing panel within the model brings in a popup menu with an option to copy this file into *EJS*. This action will extract the required files from the jar, search for the *EJS* installation in the user's hard disk, copy the files into the correct location, and run *EJS* with this simulation loaded. If a model with the same name already exists, it can be replaced. The user can then inspect, run, and modify the model just as we are doing in this chapter. A student can, for example, obtain an example or a template from an instructor and can later repackage the modified model into a new jar file for submission as a completed exercise.

### Exercise 2.3. Extracting a model

Run the stand-alone jar file containing the mass and spring model created in Exercise 2.2. Right click on the model's plot or drawing and select the *Open Ejs Model* item from the popup menu to copy the packaged model back into *EJS*.  $\square$

*EJS* is designed to be both a modeling and an authoring tool, and we suggest that you now experiment with it to learn how you can create and distribute your own models. As a start, we recommend that you run the mass and spring simulation and go through the activities in the second page of the *Description* workpanel. We modify this simulation in the next section.

## 2.6 MODIFYING THE SIMULATION

As we have seen, a prominent and distinctive feature of *Easy Java Simulations* is that it allows us to create and study a simulation at a high level of abstraction. We inspected an existing mass and spring model and its user interface in the previous section. We now illustrate additional capabilities of *Easy Java Simulations* by adding friction and a driving force and by adding a visualization of the system's phase space.

### 2.6.1 Extending the model

We can add damping in our model by introducing a viscous (Stoke's law) force that is proportional to the negative of the velocity  $F_f = -b v_x$  where  $b$  is the damping coefficient. We also add an external time-dependent driving force which takes the form of a sinusoidal function  $F_e(t) = A \sin(\omega t)$ . The introduction of these two forces changes the second-order differential equation (2.3.1) to

$$\frac{d^2 x}{dt^2} = -\frac{k}{m} (x - L) - \frac{b}{m} \frac{dx}{dt} + \frac{1}{m} F_e(t), \quad (2.6.1)$$

or, as in equations (2.3.2) and (2.3.3):

$$\frac{d x}{d t} = v_x, \quad (2.6.2)$$

$$\frac{d v_x}{d t} = -\frac{k}{m} (x - L) - \frac{b}{m} v_x + \frac{1}{m} F_e(t). \quad (2.6.3)$$

### 2.6.1.1 Adding variables

The introduction of new force terms requires that we add variables for the coefficient of dynamic friction and for the amplitude and frequency of the sinusoidal driving force. Return to the *Model* workpanel of *EJS* and select its *Variables* panel. Right-click on the tab of the existing page of variables to see its popup menu, as in Figure 2.14. Select the *Add a new page* entry as shown in Figure 2.14. Enter **Damping and Driving Vars** for the new table name in the dialog and an empty table will appear.

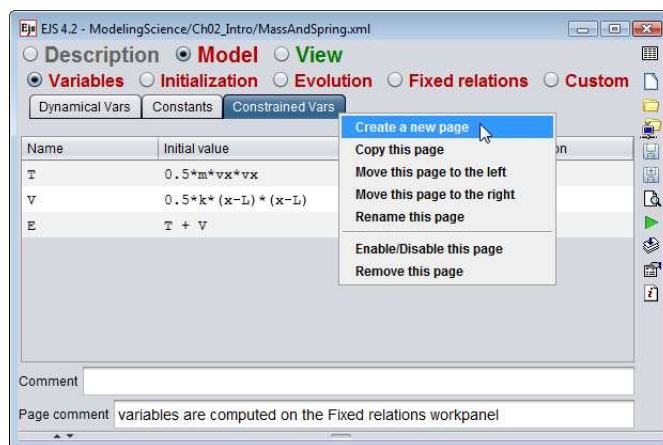


Figure 2.14: The popup menu for a page of variables.

We now use the new table to declare the needed variables. We could have used the already existing tables, but declaring multiple pages helps us organize the variables by category. Double-click on a table cell to make it editable and navigate through the table using the arrows or tab keys. Type **b** in the *Name* cell of the first row, and enter the value **0.1** in the *Initial value* cell to its right. We don't need to do anything else because the **double** type selected is already correct. *EJS* checks the syntax of the value entered and evaluates it. If we enter a wrong value, the background of the value cell will display a pink background. Notice that when you fill in a variable name, a new row appears automatically. Proceed similarly to declare a new variable for the driving force's **amp** with value **0.2** and for its **freq** with value

2.0. Document the meaning of these variables by typing a short comment for each at the bottom of the table. Our final table of variables is shown in Figure 2.15. You can ignore the empty row at the end of the table or remove it by right-clicking on that row and selecting *Delete* from the popup menu that appears.

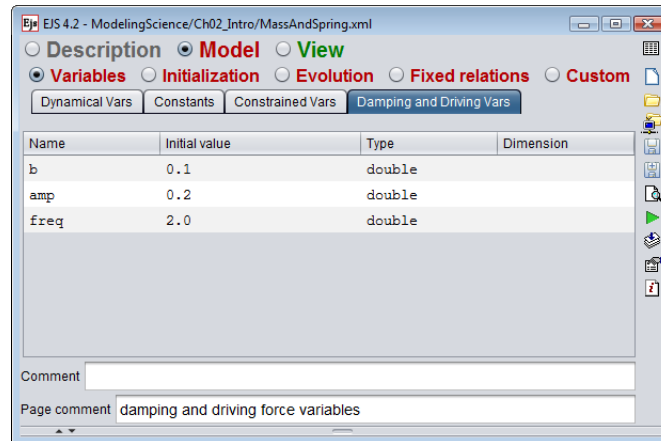


Figure 2.15: The new table of variables for the damping and forcing terms.

### 2.6.1.2 Modifying the evolution

We now modify the differential equations on the evolution page by adding expressions for the new terms in equation (2.6.3). Go to the evolution panel, double-click on the *Rate* cell of the second equation, and edit it to read:

$$-k/m * (x-L) - b*vx/m + force(t)/m$$

Notice that we are using a method (function) named **force** that has not yet been defined. We could have written an explicit expression for the sinusoidal function. However, defining a **force** method promotes cleaner and more readable code and allows us to introduce custom methods.

### 2.6.1.3 Adding custom code

The **force** method is defined using the *Custom* panel of the *Model*. Go to this panel and click on the empty central area to create a new page of custom code. Name this page *force*. You will notice that the page is created with a code template that defines the method. Edit this code to read:



```
public double force (double time) {
    return amp*Math.sin(freq*time); // sinusoidal driving force
}
```

Type this code exactly as shown including capitalization. Compilers complain if there is any syntax error.

Notice that we pass the time at which we want to compute the driving force to the `force` method as an input parameter. Passing the time value is very important. It would be incorrect to ask the method to use the value of the variable `t`, as in:



```
public double force () { // incorrect implementation of the force method
    return amp*Math.sin(freq*t);
}
```


The reason that time must be passed to the method is that time changes throughout the evolution step. In order for the ODE solver to correctly compute the time-dependent force throughout the evolution step, the time must be passed into the method that computes the rate.

Variables that change (evolve) must be passed to methods that are used to compute the rate because numerical solvers evaluate the *Rate* column in the ODE workpanel at intermediate values between  $t$  and  $t + dt$ . (See Chapter 5.) In other words, the independent variable and any other dynamic variable which is differentiated in the *State* column of the ODE editor must be passed to any method that is called in the *Rate* column. Variables which remain constant during an evolution step may be used without being passed as input parameters because the value of the variable at the beginning of the evolution step can be used.

### 2.6.2 Improving the view

We now add a visualization of the phase space (displacement versus velocity) of the system's evolution to the *View*. We also add new input fields to display and modify the value of the damping, amplitude, and frequency parameters.

Go to the *View* workpanel and notice that the *Interface* palette contains many subpanels. Click on the tab with the  icon to display the *Windows, containers, and drawing panels* palette of view elements. Click on the icon for a plotting panel, , in this palette. You can rest (hover) the mouse cursor over an icon to display a hint that describes the element if

you have difficulty recognizing the icon. Selecting an element sets a colored border around its icon on the palette and changes the cursor to a magic wand, . These changes indicate that *EJS* is ready to create an element of the selected type.

Click on the `dialog` element in the *Tree of elements* as shown in Figure 2.16 to add the plotting panel to the view.

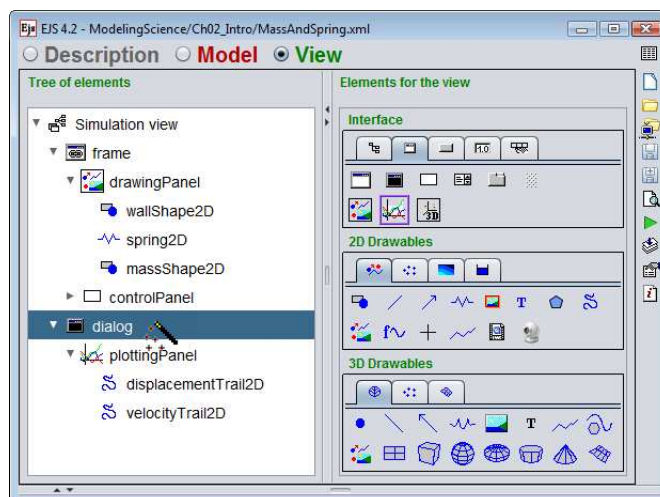



Figure 2.16: Creation of a plotting panel as a child of the `dialog` element of the view.

*EJS* asks for the name of the new element and then creates the element as a child within the existing `dialog`. A new plot appears but the dialog is too small. Return to the design mode (get rid of the magic wand) by clicking on any blank area within the *Tree of elements* or hitting the *Esc* key. Resize the dialog box by dragging its corner. You can also resize the dialog box by double-clicking on the `dialog` element in the tree to show its properties table and changing its *Size* property to "385,524", thus doubling its height. Finally, edit the properties table of the newly created plotting panel element to set the *Title* property to *Phase Space*, the *Title X* property to *Displacement*, and the *Title Y* property to *Velocity*. (*EJS* will add leading and trailing quotes to these strings to conform to the correct Java syntax.) Set the minima and maxima for both X and Y scales to -1 and 1, respectively, and leave the other properties untouched.

The plotting panel is, as its name suggests, the container for the phase-space plot. Phase space data are drawn in this panel using an element of type `Trail2D`, . Find the `Trail2D` element in the 2D Drawables palette and follow the same procedure as before. Select the `Trail2D` element and create an element of this type by clicking with the magic wand on the phase

space panel. Finally, edit the properties of the new trail element to set its **Input X** property to  $x - L$  and its **Input Y** property to  $v_x$ . This assignment causes the simulation to add a new  $(x - L, v_x)$  point to the trace after each evolution step, thus drawing the phase-space plot shown in Figure 2.17.

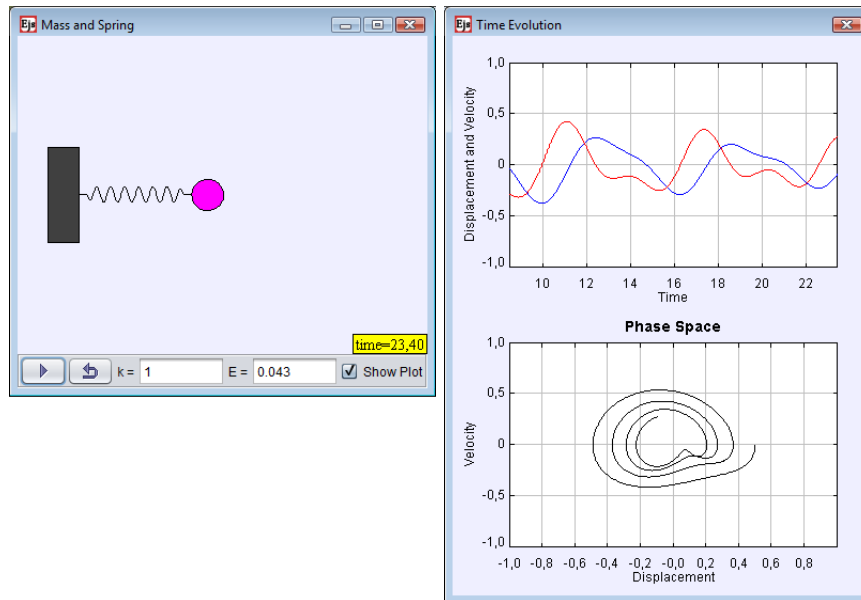






Figure 2.17: The modified simulation. The dialog includes now both a time and a phase-space plot.

To finish the modifications, we add a new panel to the top of the drawing frame that shows the sinusoidal driving force parameters.

- Select the **Panel** element icon, , on the *Windows, containers, and drawing panels* subgroup of the *Interface* palette. Click with the magic wand on the element named **frame** within the *Tree of elements* to create a new panel named **forceParamPanel** in the frame's top location. Use the properties inspector to set this panel's layout to *FLOW:center,0,0* and its border type to *LOWERED\_ETCHED*.
- Select the **Label** element icon, , on the *Input and output* subgroup of the *Interface* palette and create a new element of that type in the force parameter panel. Set the label's text property to `'frequency ='`.
- Select the **Field** element icon, , and create a new element named **freqField** in the force parameter panel. Edit the **freqField** properties table as shown in Figure 2.18. The connection to the **freq** variable is established using the **Variable** property. Click on the second icon to the right of the property field, , and choose the appropriate variable. The variable list shows all the model variables that can be used to

set the property field. The **Format** property indicates the number of decimal digits with which to display the value of the variable.

- Repeat this process to add the **amp** variable to the user interface.

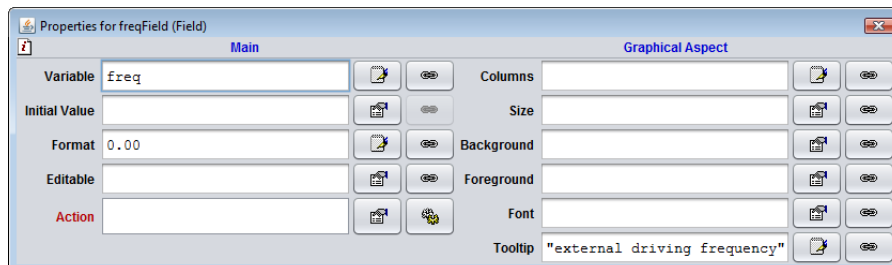




Figure 2.18: The table of properties of the `freqField` element.

### 2.6.3 Changing the description

Now that we have changed the model and the view, we should modify the description pages of our simulation. Go to the *Description* workpanel and right-click on the tab of the first page, the one labeled **Introduction**, to display the popup menu for this page. Select the *Edit/View this page* option. The description page will change to edit mode, as shown in Figure 2.19, and a simple editor will appear that provides direct access to common HTML features.

If you prefer to use your own editor, you can copy and paste HTML fragments from your editor into the *EJS* editor. If you know HTML syntax, you can enter tagged (markup) text directly by clicking the source icon, , in the tool bar. You can even import entire HTML pages into *EJS* by right-clicking on a tab in the workpanel.

Edit the description pages as you find convenient. At least change the discussion of the model to include the damping and driving forces. When you are done, save the new simulation with a different name by clicking the *Save as* icon of *EJS*' taskbar, . When prompted, enter a new name for your simulation's XML file. The modified simulation is stored in the **MasAndSpringComplete.xml** file in the **source** directory for this chapter.

## 2.7 FINDING MODELS

Now that we have covered the basics of *EJS* and you know how to load, inspect, run, and even modify an example, you may be interested in finding more examples to see what other users have done with *EJS*. Maybe, you can

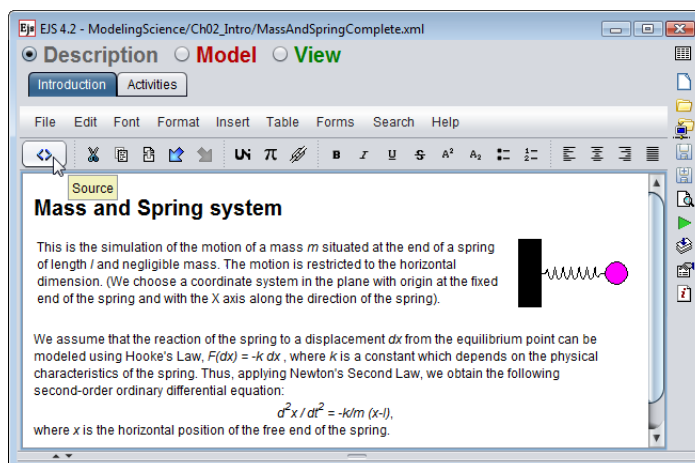



Figure 2.19: The HTML editor of *EJS*. The cursor points to the icon that switches the editor into source code edition mode.

find a model that already fits your needs or that you can easily modify to be ready for classroom use.

There are two places you can look at to find more models. The first place to look at is the source sample directory that came with your distribution of *EJS*. In the source directory of the distribution's workspace you will find some directories with sample simulations. These sample directories were also copied to your own workspace (unless you unselected this option) when you first run *EJS*.

The second, and perhaps more interesting, place (actually places) to look for new models are available through the Internet. The *EJS* digital libraries icon in the taskbar, , opens a window which allows you to connect to repositories of *EJS* models available through the Internet. This window, displayed in Figure 2.20, contains a combo box at its top that lists the available digital libraries. Select one of these libraries or click the *Get catalog* button to get the list of *EJS* models in it. All these libraries work in a similar way, and we use the **comPADRE** digital library repository to illustrate how they are accessed from within *EJS*.

The **comPADRE Pathway**, a part of the (USA) National Science Digital Library, is a growing network of educational resource collections supporting teachers and students in Physics and Astronomy. Of special relevance for our interests is the Open Source Physics comPADRE collection available at <http://www.compadre.org/OSP>. This collection contains computational resources for teaching in the form of executable simulations and curriculum resources that engage students in physics, computa-



Figure 2.20: The Digital Libraries window of *EJS*. Select one of the available repositories using the combo box at the top of the window, or click the *Get catalog* button to retrieve the list of models available.

tion, and computer modeling. In particular, it contains *EJS* models whose source (XML) code can be accessed directly from *EJS* using the digital libraries icon.

If you are connected to the Internet, select the *OSP collection on the comPADRE digital library* entry of the top combo box and *EJS* will connect to the library to obtain the very latest catalog of *EJS* models in the library. At the moment of this writing, there are some 160 models organized in different categories and subcategories, and the collection is expected to grow. As the left frame of Figure 2.21 shows, the collection is organized in categories and subcategories. When the name of a subcategory appears in red, double-click it to expand the node with the list of models of the subcategory. Because many models have primary and secondary classifications, the check box at the top, right below the library combo box, allows you to decide whether you want the models to be listed uniquely under their primary classification, or appear in all matching categories (thus appearing more than once).

When you click a model node, the right frame shows information about the model obtained instantly from the library. The information describes the model, and includes a direct link to the comPADRE library for further information. Double-clicking the model entry, or clicking the *Download* button, will retrieve the model and auxiliary files from the library, ask you for a place in your source directory of your workspace to download them,

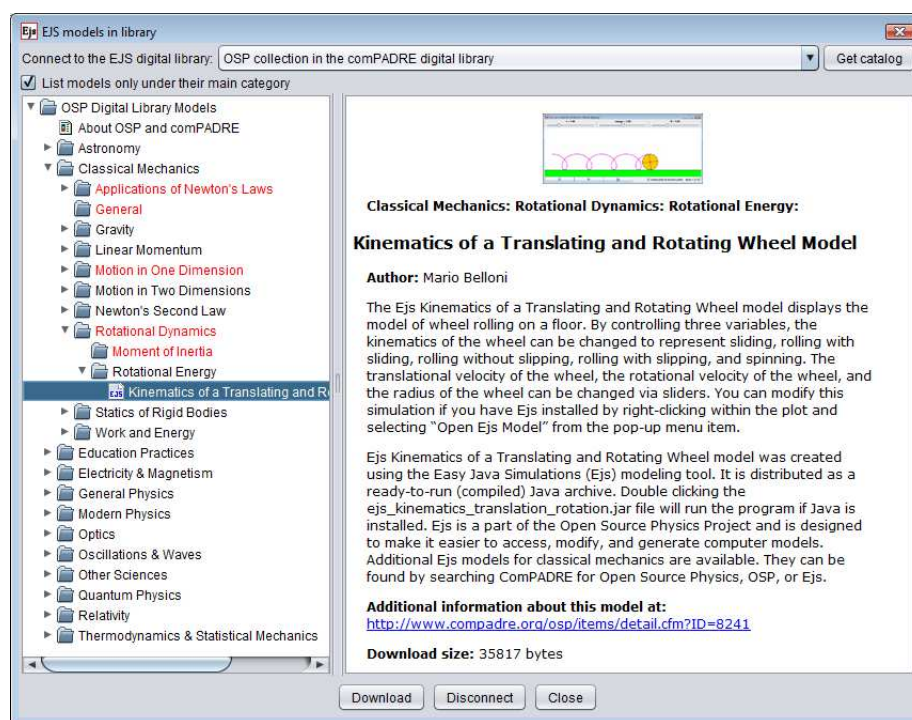


Figure 2.21: The OSP collection on the comPADRE digital library. The collection is organized in categories and subcategories. The entry for a model provides information about the model.

and open the model in *EJS* when the download is complete. Because source files are usually small, the download takes place almost instantly. Now, you can inspect, run, or modify the model as we did, earlier in this chapter, for the mass and spring model.

The OSP collection on the comPADRE digital library is a highly recommended place to look for *EJS* models and accompanying curricular material. We will often include references to models in the comPADRE digital library in this book, whenever they relate to the narrative.

## 2.8 SUMMARY

This book is about modeling and using these models to study and display a wide range of phenomena ranging the simple to the complex. An appropriate way to conduct these studies is to use computer simulations, that is, to use a computer to obtain numerical data from our models as they advance in time, and to display this data in a form humans can understand.



*Easy Java Simulations* is a modeling and authoring tool expressly devoted to this task. It has been designed to let us work at a high conceptual level, concentrating most of our time on the scientific aspects of our simulation, and asking the computer to automatically perform all the other necessary but easily automated tasks. Every tool, including *Easy Java Simulations*, has a learning curve. The first part of the book contains a series of detailed examples that will familiarize you with the modeling capabilities of *EJS* and with the most frequently used view elements. The second part of the book is devoted to advanced examples, and emphasizes the scientific content of the models and their behavior. The appendices cover additional features such as a review of Java and guidelines that will help you through the unavoidable moment when you make your first programming mistakes.

Modeling is both a science and an art. This book gives you a solid starting point in the science, a coverage of the techniques required by the art, and examples that are useful in practice.

## 2.9 PROBLEMS AND PROJECTS

**Problem 2.1** (Energy). Add a third plotting panel to the dialog window of the **MassAndSpringComplete.xml** simulation that will display the evolution of the kinetic, potential, and total energies.

**Problem 2.2** (Function plotter). The analytic solution for the undriven simple harmonic oscillator is

$$x(t) = A \sin(w_0 t + \phi) \quad (2.9.1)$$

where  $A$  is the amplitude (maximum displacement),  $w_0 = \sqrt{k/m}$  is the natural frequency of oscillation, and  $\phi$  is the phase angle. Consult a mechanics textbook to determine the relationship between the amplitude and phase angle and the initial displacement and velocity. Use the **Function-Plotter.xml** simulation in the Chapter 2 directory to compare the analytic solution to the numerical solution generated by the **MassAndSpringComplete.xml** model.

**Project 2.1** (Two-dimensional oscillator). Modify the model of the mass and spring simulation to consider motion that is not restricted to the horizontal direction. Assume that a second spring with spring constant  $k'$  produces a vertical restoring force  $F_y(\delta y) = -k' \delta y$ . Modify the simulation to allow the user to specify the Hooke's law constants as well as the initial conditions in both directions. Describe the motion produced without a driving force but under different initial conditions and with different spring constants. (Try  $k = 1$  and  $k' = 9$ .) Show that it is possible to obtain circular motion if  $k = k'$ .



**Project 2.2** (Simple pendulum). Create a similar simulation as the one described in this chapter for a simple pendulum whose second-order differential equation of motion is

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L} \sin(\theta), \quad (2.9.2)$$

where  $\theta$  is the angle of the pendulum with the vertical,  $g$  is the acceleration due to gravity, and  $L$  is the arms's length. Use fixed relations to compute the  $x$  and  $y$  position of the pendulum bob using the equations:

$$\begin{aligned} x &= L \sin(\theta) \\ y &= -L \cos(\theta). \end{aligned}$$