
Chapter Three

Exploring the Javascript flavor of Easy Java(script) Simulations


Change is the law of life. And those who look only to the past or present are certain to miss the future. *John F. Kennedy*

To provide a perspective of the modeling process, in this chapter we first load, inspect, and run an existing simple harmonic oscillator simulation. We then modify the simulation to show how *EjsS* engages the user in the modeling process and greatly reduces the amount of programming that is required. This chapter uses Javascript as the programming language for the modeling and is a twin chapter for Chapter 2 (where Java is used).

3.1 INSPECTING THE SIMULATION

As mentioned in Chapter 1, *Easy Java(script) Simulations* provides three workpanels for modeling. The first panel, *Description*, allows us to create and edit multimedia HTML-based narrative that describes the model. 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, *HtmlView*, is dedicated to the task of building the graphical user interface, which allows users to control the simulation and to display its output.

To understand how the *Description*, *Model*, and *HtmlView* 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.

Click on the *Open* icon  on the *EjsS* taskbar. A file dialog similar to that in Figure 3.1 appears showing the contents of your workspace's **source** directory. Go to the **JavascriptExamples** directory, where you will find

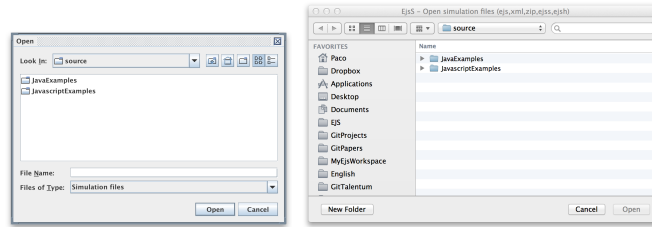


Figure 3.1: The open file dialog lets you browse your hard disk and load an existing simulation. The appearance of the dialog (shown here using two different look and feels) may vary, depending on your operating system and the selected *look and feel*.

a file called **MassAndSpring.ejss**. Select this file and click on the *Open* button of the file dialog.

Now, things come to life! *EjsS* reads the **MassAndSpring.ejss** document which populates the workpanels and a new “EjsS Emulator” appears in your display as shown in Figure 3.2. A quick warning. You can drag objects or click buttons within this mock-up window but the model will not exhibit its full behavior. You need to run the simulation for that.

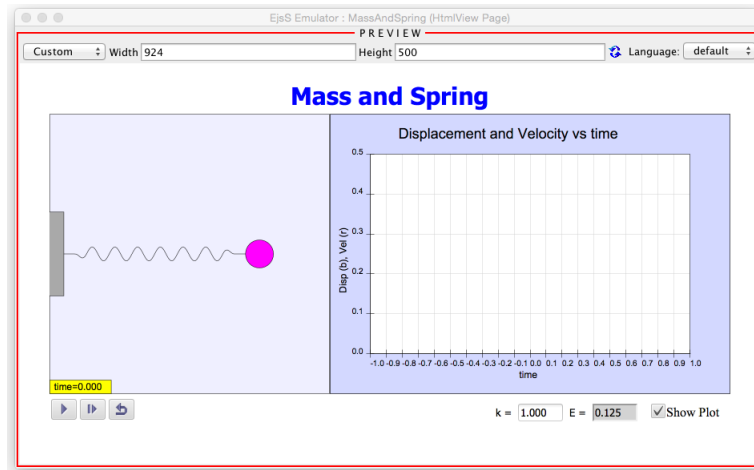




Figure 3.2: *EjsS* mock-up windows of the **MassAndSpring** simulation. The title bar and the red border show that this an HTML Emulator window within *EjsS* and that the program is not running.

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 *EjsS* but with a compiled and running Javascript program on an HTML page. Exit the running program by closing the *Mass and Spring* window or by right clicking on the (now) red-squared stop icon  on *EjsS*' taskbar before proceeding.

3.1.1 The *Description* workpanel

Select the *Description* workpanel by clicking on the corresponding radio button at the top of *EjsS*, and you will see two pages of narrative for this simulation. The first page, shown in Figure 3.3, contains a short discussion of the mass and spring model. Click on the *Activities* tab to view the second page of narrative.

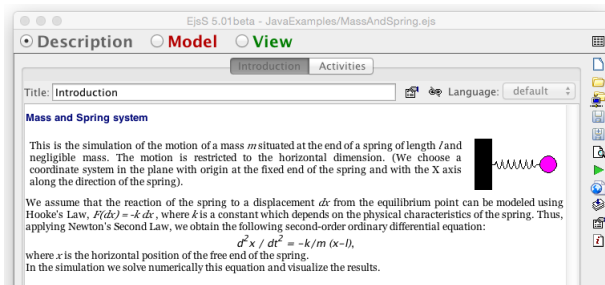


Figure 3.3: The description pages for the mass and spring simulation. Click on a tab to display the page. Right-click on a tab to edit the page.

A *Description* is HTML or XHTML 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. The X in XHTML stands for eXtensible. XHTML is basically HTML expressed as valid XML or, in simpler words, perfectly formatted HTML.

EjsS provides a simple HTML editor that lets you create and modify pages within *EjsS*. You can also import HTML or (preferably) XHTML pages into *EjsS* by right clicking on a tab in the *Description* workpanel. (See Section 3.4.3.) Description pages are an essential part of the modeling process and these pages are included with the compiled model when the model is exported for distribution.

3.1.2 The *Model* workpanel

The *Model* workpanel is where the model is defined so that it can be converted into a program by *EjsS*. 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 δx 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:

$$\frac{d^2 x}{dt^2} = -\frac{k}{m} (x - L). \quad (3.1.1)$$

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 six panels.

3.1.2.1 Declaration of variables

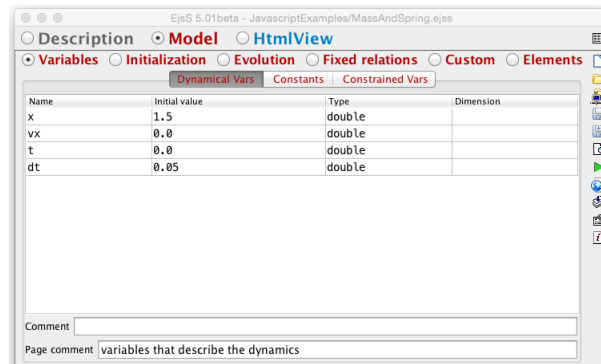


Figure 3.4: The *Model* workpanel contains six subpanels. The subpanel for the definition of mass and spring dynamical variables is displayed. Other tabs in this subpanel define additional variables, such as the natural length of the spring L and the energy E .

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 3.4 shows an *EjsS* 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 (≈ 16 significant digits) numbers, and `String` for text. We will use all these variable types in this document, but the mass and spring model uses only variables of type `double` and `boolean`.

If you have already learnt a bit of Javascript, you will probably know that Javascript has actually no types for variables. All variables are declared with a `var` keyword. This means that, in principle, Javascript makes no difference among integers, doubles, Strings, etc. . . But it does! For instance, you should not use a double variable as index for an array. For this reason, and also because it helps clarify the use of variables in your model (what values they can have and where you can or cannot use them), we ask you to assign a type to each variable in your model.

Variables can be used as parameters, state variables, or inputs and outputs of the model. The tables in Figure 3.4 define the variables used within our model. We have declared a variable for the x -position of the particle, `x`, for its velocity in the x -direction, `vx`, for the time, `t`, and for the increment of time at each simulation step, `dt`. We define some variables, in this and other tabs, that do not appear in Equation(3.1.1). The reason for auxiliary variables, such as `vx` or 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.

3.1.2.2 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 Javascript expressions such as `0.5*m*vx*vx`) in the *Initial value* column of the table of variables. *EjsS* 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 Javascript code that perform the required computation. *EjsS* converts this code into a Javascript function 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 Subsection 3.1.2.4 for an example of how Javascript code appears in *EjsS*.

3.1.2.3 The evolution of the model

The *Evolution* panel allows us to write the Javascript 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 (3.1.1), without programming. *EjsS* provides a dedicated editor that lets us specify differential equations in a format that resembles mathematical notation and automatically generates the correct Javascript code.

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 (3.1.1), must be recast into a first-order system. We can do so by treating the velocity as an independent variable which obeys its own equation:

$$\frac{d x}{d t} = v_x \quad (3.1.2)$$

$$\frac{d v_x}{d t} = -\frac{k}{m} (x - L). \quad (3.1.3)$$

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 3.5. Notice that the ODE editor displays (3.1.2) and (3.1.3) (using

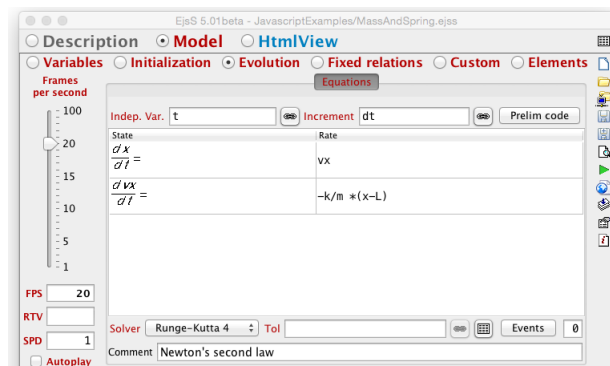


Figure 3.5: The ODE evolution panel showing the mass and spring differential equation and the numerical algorithm.

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 code*

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 (*Tol*) is greyed out and empty because Runge–Kutta 4 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 further on in this document. The different solver algorithms and its parameters are discussed in the *EjsS* 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 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 t to a new state at a new time $t + dt$ before the visualization is redrawn. The simulation repeats this evolution step 20 times per second on computers or devices with modest processing power. The simulation may run slower and not as smoothly on computers or devices 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 *EjsS* can apply these algorithms to large systems of vector differential equations with or without discontinuous events.

3.1.2.4 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.

$$T = \frac{1}{2}mv_x^2, \quad (3.1.4)$$

$$V = \frac{1}{2}k(x - L)^2, \quad (3.1.5)$$

$$E = T + V. \quad (3.1.6)$$

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

The *Fixed relations* panel shown in Figure 3.6 is used to write relations among variables. Notice how easy it is to convert (3.1.4) through (3.1.6) into Javascript syntax. Be sure to use the multiplication character `*` and to place a semicolon at the end of each Javascript statement.

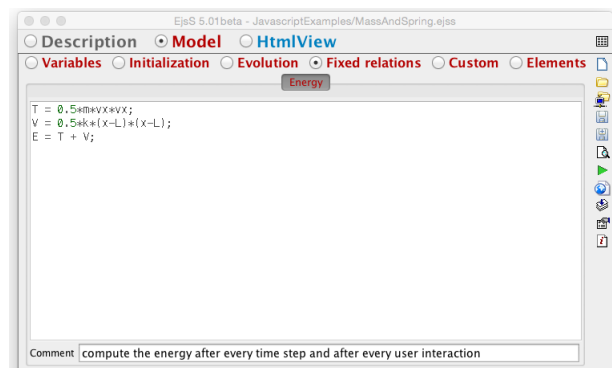


Figure 3.6: Fixed relations for the mass and spring model.

Here goes an important remark. 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. *EjsS* 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.

3.1.2.5 Custom pages

There is a fifth panel in the *Model* workpanel labeled *Custom*. This panel can be used to define Javascript 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 3.4. A custom method is not used unless it is explicitly invoked from another workpanel.

3.1.2.6 Model elements

The final, sixth panel in the *Model* workpanel is labeled *Elements* and provides access to third-party Javascript libraries in the form of drag and drop icons. You add these libraries to your program by dragging the corresponding icon to the list of model elements to use for this model. This creates Javascript objects you can then use in your model code. This panel is also empty for this model because our mass and spring doesn't require additional Javascript libraries.

3.1.3 The *HtmlView* workpanel

The third *Easy Java(script) Simulations* workpanel is the *HtmlView*. This workpanel allows us to create a graphical HTML-based interface that includes visualization, user interaction, and program control with minimum programming. Figure 3.2 shows the HTML-based view for the mass and spring model. Select the *HtmlView* radio button to examine how this HTML-based view is created.

The right frame of the *HtmlView* workpanel of *EjsS*, shown in Figure 3.7, contains a collection of HTML-based *view elements*, grouped by

functionality. View elements are building blocks that can be combined to form a complete user HTML-based 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 an empty HTML view (think of it as a blank HTML page) and add elements, such as panel, buttons and graphs, using “drag and drop” as described in Section 3.4.

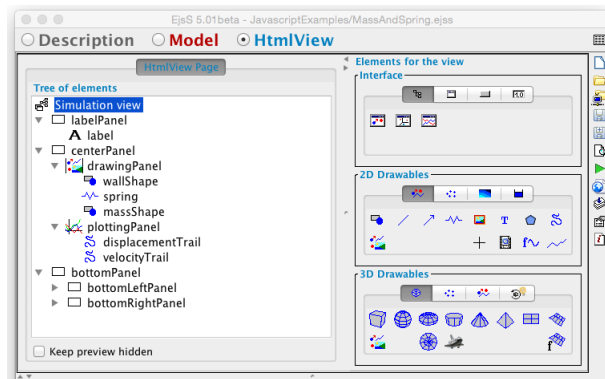


Figure 3.7: The *HtmlView* workpanel showing the *Tree of elements* for the mass and spring user interface.

The *Tree of elements* shown on the left side of Figure 3.7 displays the structure of the mass and spring user interface. Notice that the simulation has three main panels, `labelPanel`, `centerPanel` and `bottomPanel`, that appear tiled vertically on the HTML page in the Emulator. (The *EjsS* Emulator emulates an HTML browser.) These panels 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. Alternatively, you can drag and drop elements from one container to another to change the parent-child relationship, or within a container to change the child order. (There are conditions for a container to accept a given element as child. For instance, a two-dimensional drawing panel can only accept 2D drawable elements.)

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

into a dynamic and interactive visualization.

Let's see how this procedure works in practice. Double-click on the `massShape` element (the 'Shape' 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 `massShape`'s table of properties appears as shown in Figure 3.8.

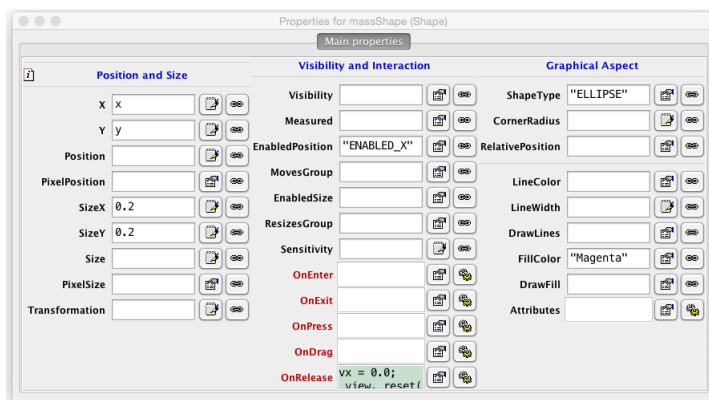


Figure 3.8: The table of properties of the `massShape` element.

Notice the properties that are given constant values. The `ShapeType`, `SizeX`, `SizeY`, and `FillColor` properties produce an ellipse of size $(0.2, 0.2)$ units (which makes a circle) filled with the color Magenta. More importantly, the `X` and `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, however, that the `Draggable` property is set to allow you only to drag horizontally the mass.

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. When the user releases the mass after dragging it, the following code (specified on the `OnRelease` action property) is executed:

```
vx = 0.0;           // sets the velocity to zero
_view.reset();    // 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 3.1. 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 `displacementTrail` and `velocityTrail` elements correspond to the displacement and velocity time plots in the rightmost big plotting panel of the view, respectively. What is the maximum number of points that can be added to each trail? □

3.1.4 The completed simulation

We have seen that *Easy Java(script) 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 3.6) and the two lines of code in the particle's action method are the only explicit Javascript code needed to implement the model. *Easy Java(script) Simulations* creates a complete Javascript program by processing the information in the workpanels when the run icon is pressed as described in Section 3.2.

3.2 RUNNING THE SIMULATION

It is time to run the simulation by clicking on the *Run* icon of the taskbar, ►. *EjsS* generates the Javascript code, collects auxiliary and library files,

and generates and opens an HTML page with the complete 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 fixed 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, *EjsS* changes its *Run* triangle icon to a red *Kill* square and prints informational messages saying that the simulation has been successfully generated and that it is running. Notice that the *EjsS* Emulator mock-up window disappears and is replaced by a new but similar Emulator window without the red border in their title. (Alternatively, you can set *EjsS*' options to run the simulation in your system's default HTML browser.) 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 3.9. To exit the program, close the simulation's main window.

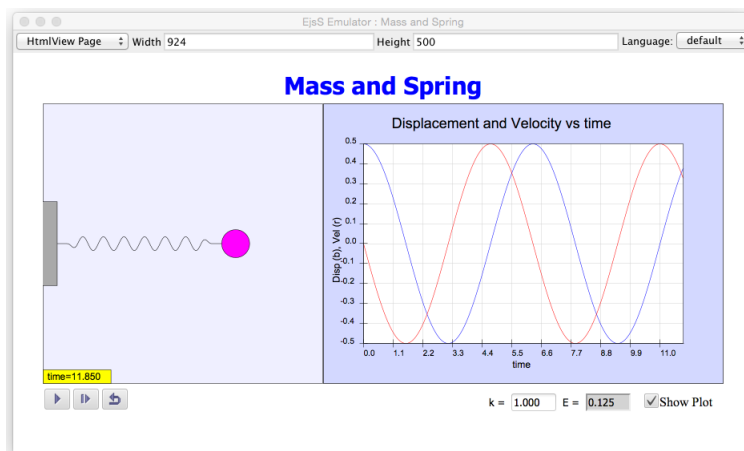



Figure 3.9: The mass and spring simulation displays an interactive drawing of the model and a graph with displacement and velocity data.

3.3 DISTRIBUTING THE SIMULATION

Simulations created with *EjsS* are stand-alone Javascript programs that can be distributed without *EjsS* for other people to use. The easiest way to do this is to package the simulation in a single executable zip 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 zip file is ready to be distributed on a CD or via the Internet. Other distribution mechanisms are available by right-clicking on the icon.

Exercise 3.2. Distribution of a model

Click on the *Package* icon on the taskbar to create a stand alone zip archive of the mass and spring simulation. Copy this zip file into a working directory separate from your *EjsS* installation. Close *EjsS* and verify that the simulation runs as a stand-alone application by unzipping the file and double-clicking the **MassAndSpring.xhtml** file that is extracted from the zip file. □

An important pedagogic feature is that is very easy to distribute your simulation source code so that other people can use it with *EjsS* at any time to examine, modify, and adapt the model. (*EjsS* must, of course, be installed.) *EjsS* writes all the information in its workpanels into a small *Extensible Markup Language* (XML) description file. And it can also create a single, compressed ZIP file with that XML information and all the resource files (such as images) that you used in your simulation. This ZIP file can then be distributed, and trying to open it with *EjsS* will extract the required files from the ZIP, copy the files into a folder of your workspace, and load *EjsS* with this simulation. 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 sources into a new ZIP file for submission as a completed exercise.

Exercise 3.3. Packaging and extracting a model source files

Right click the *Package* icon and select *ZIP the simulation source files* from the menu that appears. Reopen the created ZIP file with *EjsS* and select a destination folder (different from **JavascriptExamples**) in your workspace for *EjsS* to unzip the sources. Notice that *EjsS* copies all the files in the original example for you and opens the new **MassAndSpring.ejss** XML file. □

EjsS 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.

3.4 MODIFYING THE SIMULATION

As we have seen, a prominent and distinctive feature of *Easy Java(script) 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(script) Simulations* by adding friction and a driving force and by adding a visualization of the system's phase space.

3.4.1 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 (3.1.1) to

$$\frac{d^2 x}{dt^2} = -\frac{k}{m}(x - L) - \frac{b}{m} \frac{dx}{dt} + \frac{1}{m} F_e(t), \quad (3.4.1)$$

or, as in equations (3.1.2) and (3.1.3):

$$\frac{dx}{dt} = v_x, \quad (3.4.2)$$

$$\frac{dv_x}{dt} = -\frac{k}{m}(x - L) - \frac{b}{m} v_x + \frac{1}{m} F_e(t). \quad (3.4.3)$$

3.4.1.1 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 *EjsS* and select its *Variables* panel. Right-click on the tab of the existing page of variables to see its popup menu, as in Figure 3.10. Select the *Add a new page* entry as

shown in Figure 3.10. Enter **Damping and Driving Vars** for the new table name in the dialog and an empty table will appear.

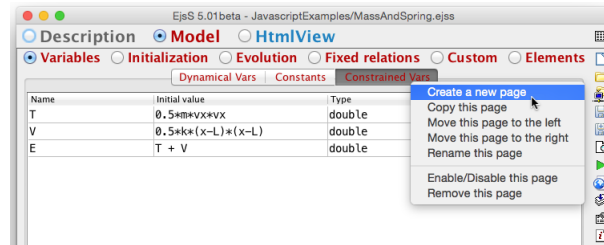


Figure 3.10: The popup menu for a page of variables.

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. *EjsS* 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 3.11. 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.

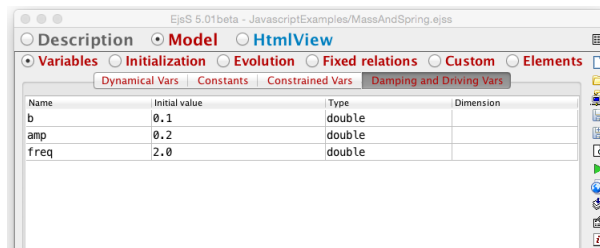


Figure 3.11: The new table of variables for the damping and forcing terms.

3.4.1.2 Modifying the evolution

We now modify the differential equations on the evolution page by adding expressions for the new terms in equation (3.4.3). 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.

3.4.1.3 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:

```
function force (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:

```
function 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 $t + dt$. 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.

3.4.2 Improving the view

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

Go to the *HtmlView* 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 *EjsS* is ready to create an element of the selected type. (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.)

Click on the *Simulation view* tree node in the *Tree of elements* as shown in Figure 3.12 to add the plotting panel to the view.

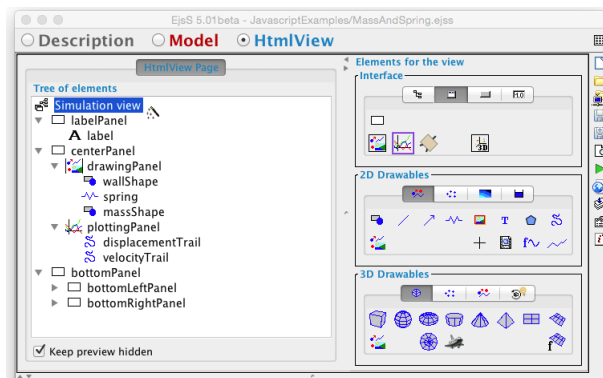



Figure 3.12: Creation of a plotting panel as a new (and last) child of the simulation view.

EjsS asks for the name of the new element and then creates the element as a new child of the simulation view (in the last position). A new plot appears but the Emulator window is too small. Resize the Emulator window

by editing the *Width* and *Height* fields at its top. This window size is useful to orient you about how the final simulation will look in devices with fixed screen resolution (such as tablets). Finally, edit the properties table of the newly created plotting panel element to set the **Title** property to **Phase Space**, the **TitleX** property to **Displacement**, and the **TitleY** property to **Velocity**. (*EjsS* will add leading and trailing quotes to these strings to conform to the correct Javascript 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 a 2D element of type **Trail**, . Find the **Trail2D** element in the **2D Drawables** palette and follow the same procedure as before. Select the **Trail** element and create an element of this type by clicking with the magic wand on the new phase space plotting panel. Finally, edit the properties of the new trail element to set its **InputX** property to $x - L$ and its **InputY** 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 3.13.

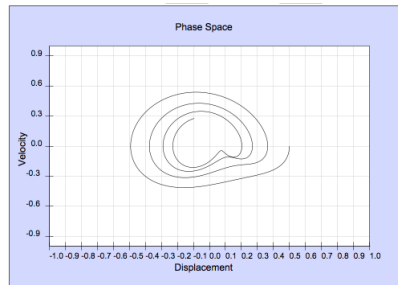






Figure 3.13: The new phase-space plot added to the simulation view.

To finish the modifications, we will add a new panel before the new plotting 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 **Simulation** view root node within the *Tree of elements* to create a new panel named **forceParamPanel** as its last child. Right-click and use the *Move up* option to locate it before the phase space plotting panel. (You can also drag-and-drop it to the new position in the tree.)
- Select the **Label** element icon, , on the *Buttons and decorations* subgroup of the *Interface* palette and create a new element of that type in the **forceParamPanel** 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 3.14. The connection to the `freq` variable is established using the **Value** 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.

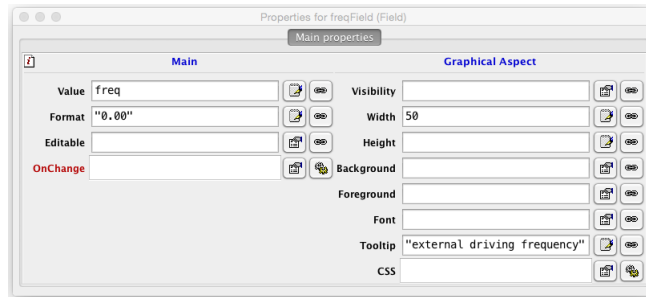







Figure 3.14: The table of properties of the `freqField` element.

3.4.3 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 click on the tab of the first page, the one labeled *Introduction*. Once you see this page, click the *Click to modify the page* icon, . The description page will change to edit mode, as shown in Figure 3.15, 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 *EjsS* 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 *EjsS* by clicking the *Link/Unlink page to external file* icons, , .

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 *EjsS*' taskbar, . When prompted, enter a new name for your simulation's XML file. The modified simulation is stored in the **MassAndSpringComplete.ejss** file in the **source** directory for this chapter.

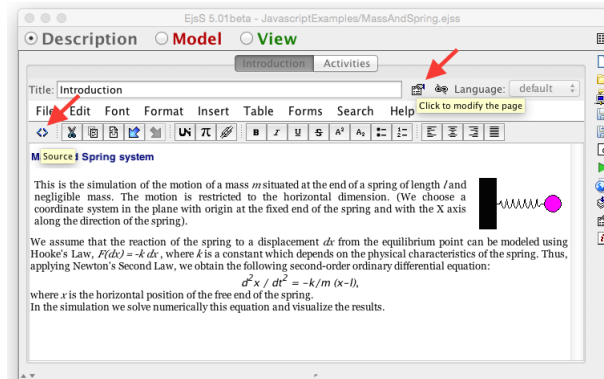


Figure 3.15: The HTML editor of *EjsS*. The added red arrows point to the edit and source code edition mode icons.

3.5 PROBLEMS AND PROJECTS

Problem 3.1 (Energy). Add a third plotting panel to the dialog window of the **MassAndSpringComplete.ejss** simulation that will display the evolution of the kinetic, potential, and total energies.

Problem 3.2 (Function plotter). The analytic solution for the undriven simple harmonic oscillator is

$$x(t) = A \sin(\omega_0 t + \phi) \quad (3.5.1)$$

where A is the amplitude (maximum displacement), $\omega_0 = \sqrt{k/m}$ is the natural frequency of oscillation, and ϕ 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.ejss** simulation in the examples directory to compare the analytic solution to the numerical solution generated by the **MassAndSpringComplete.ejss** model.

Project 3.1 (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 3.2 (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

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L} \sin(\theta), \quad (3.5.2)$$

where θ 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:

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

Bibliography