

Chapter 1

Introduction

©2012 by Wolfgang Christian
13 January 2012

Adapted from *An Introduction to Computer Simulation Methods* by Harvey Gould, Jan Tobochnik, and Wolfgang Christian

The importance of computers in physics and the nature of computer simulation is discussed. The nature of object-oriented programming and various computer languages also is considered.

1.1 Importance of computers in physics

Computation is now an integral part of contemporary science, and is having a profound effect on the way we do physics, on the nature of the important questions, and on the physical systems we choose to study. Developments in computer technology are leading to new ways of thinking about physical systems. Asking “How can I formulate this problem on a computer?” has led to the understanding that it is practical and natural to formulate physical laws as rules for a computer rather than only in terms of differential equations.

For the purposes of discussion we will divide the use of computers in physics into the following categories: numerical analysis, symbolic manipulation, visualization, simulation, and the collection and analysis of data. *Numerical analysis* refers to the solution of well-defined mathematical problems to produce numerical (in contrast to symbolic) solutions. For example, we know that the solution of many problems in physics can be reduced to the solution of a set of simultaneous linear equations. Consider the equations

$$\begin{aligned}2x + 3y &= 18 \\ x - y &= 4.\end{aligned}$$

It is easy to find the analytical solution $x = 6$, $y = 2$ using the method of substitution. Suppose we wish to solve a set of four simultaneous equations. We again can find an analytical solution, perhaps using a more sophisticated method. However, if the number of simultaneous equations becomes much larger, we would need to use a computer to find a solution. In this mode the computer is

a tool of numerical analysis. Because it often is necessary to compute multidimensional integrals, manipulate large matrices, or solve nonlinear differential equations, this use of the computer is important in physics.

One of the strengths of mathematics is its ability to use the power of abstraction, which allows us to solve many similar problems simultaneously by using symbols. Computers can be used to do much of the *symbolic manipulation*. As an example, suppose we want to know the solution of the quadratic equation, $ax^2 + bx + c = 0$. A symbolic manipulation program can give the solution as $x = [-b \pm \sqrt{b^2 - 4ac}]/2a$. In addition, such a program can give numerical solutions for specific values of a , b , and c . Mathematical operations such as differentiation, integration, matrix inversion, and power series expansion can be performed using symbolic manipulation programs. The calculation of Feynman diagrams, which represent multi-dimensional integrals of importance in quantum electrodynamics, has been a major impetus to the development of computer algebra software that can manipulate and simplify symbolic expressions. Maxima, Maple, Mathematica, and Sage are examples of software packages that have symbolic manipulation capabilities as well as many tools for numerical analysis. Matlab and Octave are examples of software packages that are convenient for computations involving matrices and related tasks.

As the computer plays an increasing role in our understanding of physical phenomena, the *visual representation* of complex numerical results is becoming even more important. The human eye in conjunction with the visual processing capacity of the brain is a very sophisticated device. Our eyes can determine patterns and trends that might not be evident from tables of data and can observe changes with time that can lead to insight into the important mechanisms underlying a system's behavior. The use of graphics also can increase our understanding of the nature of analytical solutions. For example, what does a sine function mean to you? We suspect that your answer is not the series, $\sin x = x - x^3/3! + x^5/5! + \dots$, but rather a periodic, constant amplitude curve (see Figure 1.1). What is most important is the mental image gained from a visualization of the form of the function.

Traditional modes of presenting data include two- and three-dimensional plots including contour and field line plots. Frequently, more than three variables are needed to understand the behavior of a system, and new methods of using color and texture are being developed to help researchers gain greater insight about their data.

An essential role of science is to develop models of nature. To know whether a model is consistent with observation, we have to understand the behavior of the model and its predictions. One way to do so is to implement the model on a computer. We call such an implementation a *computer simulation* or simulation for short. For example, suppose a teacher gives \$10 to each student in a class of 100. The teacher, who also begins with \$10 in her pocket, chooses a student at random and flips a coin. If the coin is heads, the teacher gives \$1 to the student; otherwise, the student gives \$1 to the teacher. If either the teacher or the student would go into debt by this transaction, the transaction is not allowed. After many exchanges, what is the probability that a student has s dollars? What is the probability that the teacher has t dollars? Are these two probabilities the same? Although these particular questions can be answered by analytical methods, many problems of this nature cannot be solved in this way (see Problem 1.1).

One way to determine the answers to these questions is to do a classroom experiment. However, such an experiment would be difficult to arrange, and it would be tedious to do a sufficient number of transactions.

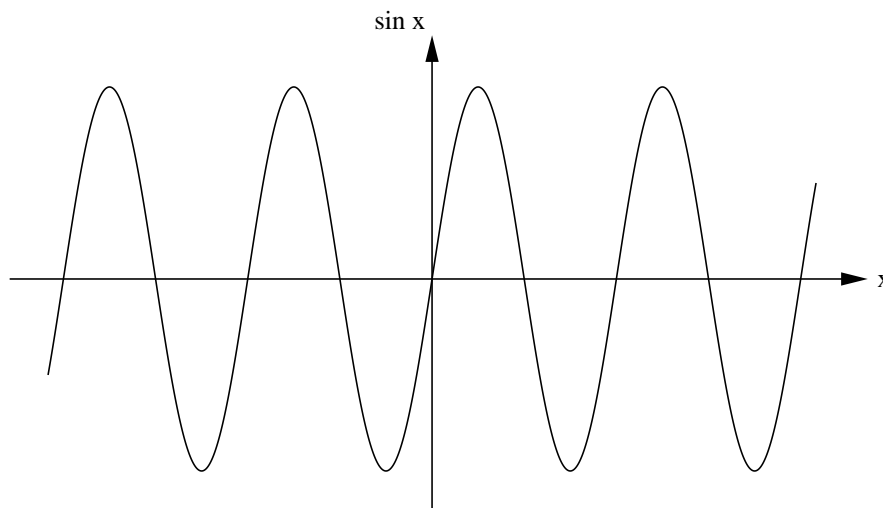


Figure 1.1: Plot of the sine function. What is the meaning of the sine function?

A more practical way to proceed is to convert the rules of the model into a computer program, simulate many exchanges, and estimate the quantities of interest. Knowing the results might help us gain more insight into the nature of an analytical solution if one exists. We also can modify the rules and ask “what if?” questions. For example, would the probabilities change if the students could exchange money with one another? What would happen if the teacher were allowed to go into debt?

Simulations frequently use the computational tools of numerical analysis and visualization, and occasionally symbolic manipulation. The difference is one of emphasis. Simulations are usually done with a minimum of analysis. Because simulation emphasizes an exploratory mode of learning, we will stress this approach.

Computers also are involved in all phases of a laboratory experiment, from the design of the apparatus to the *collection and analysis of data*. LabView is an example of a data acquisition program. Some of the roles of the computer in laboratory experiments, such as the varying of parameters and the analysis of data, are similar to those encountered in simulations. However, the tasks involved in *real-time control* and interactive data analysis are qualitatively different and involve the interfacing of computer hardware to various types of instrumentation. We will not discuss this use of the computer.

1.2 The importance of computer simulation

Why is computation becoming so important in physics? One reason is that most of our analytical tools such as differential calculus are best suited to the analysis of *linear* problems. For example, you probably have analyzed the motion of a particle attached to a spring by assuming a linear restoring force and solving Newton’s second law of motion. In this case, a small change in the

Laboratory experiment	Computer simulation
sample	model
physical apparatus	computer program
calibration	testing of program
measurement	computation
data analysis	data analysis

Table 1.1: Analogies between a computer simulation and a laboratory experiment.

force on the particle leads to a small change in its velocity. However, many natural phenomena are *nonlinear*, and a small change in a variable might produce a large non-proportional change in another. Because few nonlinear problems can be solved by analytical methods, the computer gives us a new tool to explore nonlinear phenomena.

Another reason for the importance of computation is the growing interest in systems with many variables or with many degrees of freedom. The money exchange model described in Section 1.1 is a simple example of a system with many variables.

Computer simulations are sometimes referred to as *computer experiments* because they share much in common with laboratory experiments. Some of the analogies are shown in Table 1.1. The starting point of a simulation is the development of an idealized model of a physical system of interest. We then need to specify a procedure or *algorithm* for implementing the model on a computer and decide what quantities to measure. The results of a simulation can serve as a bridge between laboratory experiments and theoretical calculations. In some cases we can obtain accurateresults by simulating an idealized model that has no laboratory counterpart. The results of the idealized model can serve as a stimulus to the development of the theory. In other cases we can do simulations of a more realistic model than can be done theoretically, and hence make a more direct comparison with laboratory experiments. Computation has become a third way of doing physics and complements both theory and experiment.

Computer simulations, like laboratory experiments, are not substitutes for thinking, but are tools that we use to understand natural phenomena. The goal of all our investigations of fundamental phenomena is to seek explanations of natural phenomena that can be stated concisely.

1.3 Programming languages

There is no single best programming language any more than there is a best natural language. Fortran is the oldest of the more popular scientific programming languages and was developed by John Backus and his colleagues at IBM between 1954 and 1957. Fortran is commonly used in scientific applications and continues to evolve. Fortran 90/95/2000 has many modern features that are similar to C/C++.

The Basic programming language was developed in 1965 by John Kemeny and Thomas Kurtz at Dartmouth College as a language for introductory courses in computer science. In 1983 Kemeny and Kurtz extended the language to include platform independent graphics and advanced control structures necessary for structured programming. The programs in the first two editions of our

computer simulation textbook were written in this version of Basic, known as True Basic.

C was developed by Dennis Ritchie at Bell Laboratories around 1972 in parallel with the Unix operating system. C++ is an extension of C designed by Bjarne Stroustrup at Bell laboratories in the mid-eighties. C++ is considerably more complex than C, has object oriented features, and other extensions. In general, programs written in C/C++ have high performance, but can be difficult to debug. C and C++ are popular choices for developing operating systems and software applications because they provide direct access to memory and other system resources.

Python, like Basic, was designed to be easy to learn and use. Python enthusiasts like to say that C and C++ were written to make life easier for the computer, but Python was designed to be easier for the programmer. Guido van Rossum created Python in the late 80's and early 90's. It is an interpreted, object-oriented, general-purpose programming language that also is good for prototyping. Because Python is interpreted, its performance is significantly less than optimized languages like C or Fortran.

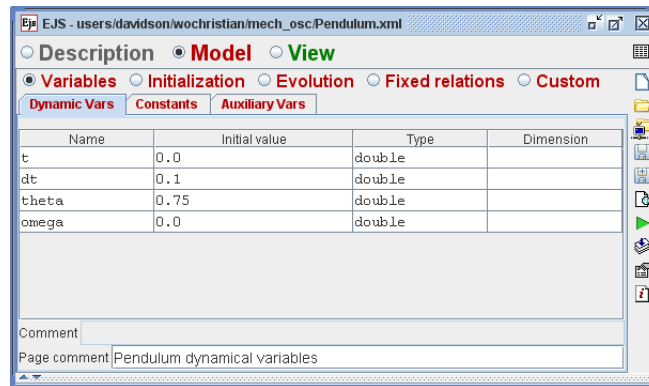
Java is an object-oriented language that was created by James Gosling and others at Sun Microsystems. Since Java was introduced in late 1995, it has rapidly evolved, and has become very popular and is the language of choice in most introductory computer science courses. Java borrows much of its syntax from C++, but has a simpler structure. Although the language contains only fifty keywords, the Java *platform* adds a rich library that enables a Java program to connect to the internet, render images, and perform other high-level tasks.

Java can be thought of as a platform in itself, similar to the Macintosh and Windows, because it has an application programming interface (API) that enables cross-platform graphics and user interfaces. Java programs are compiled to a platform neutral byte code so that they can run on any computer that has a Java Virtual Machine (VM). Despite the high level of abstraction and platform independence, the performance of Java is comparable with native languages because the Java VM compiles and optimizes the byte code for the host platform when the program is run.

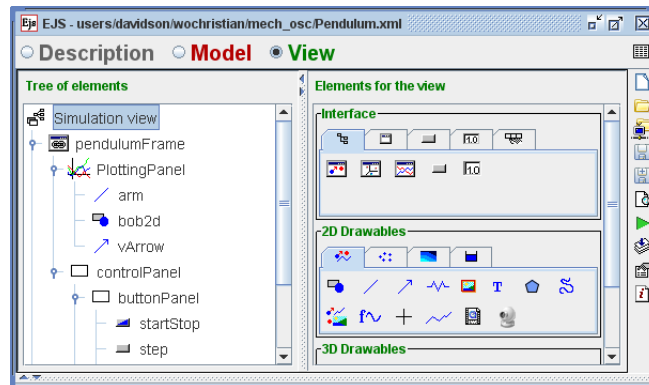
1.4 *Easy Java Simulations*

Every programming language requires tools, such as editors and compilers, to produce and distribute ready to run programs. These tools are often integrated into development environments with graphical user interfaces, such as *Eclipse*, that provide easy access to the underlying tools. We have chosen the *Easy Java Simulations (EJS)* tool shown in Figure 1.2 because its dynamic and highly interactive user interface greatly reduces the amount of programming required to implement a model and an algorithm. *Easy Java Simulations* enables experienced programmers and novices to quickly and easily prototype, test, and distribute packages of Java simulations. *EJS* gently introduces novices to Java syntax, but even experienced programmers find it useful because it is faster and easier to:

- develop a prototype of an application to test an idea or algorithm;
- create user interfaces without programming;
- create models whose structure and algorithms can be inspected and understood by non-programmers;



(a) Model.



(b) View.

Figure 1.2: The Model workpanel provides access to the data and methods that define the physics. The View workpanel shows the graphical user interface that connects to the model.

- encourage students or colleagues to create their own simulations;
- quickly prepare simulations to be distributed as applets or as stand alone programs;
- and create a package containing multiple programs and the associated curricular material.

Easy Java Simulations simplifies the modeling process by breaking the process into activities that are selected using radio buttons: (1) documentation, (2) modeling, and (3) interface design. The model's html-based documentation is accessed by selecting Description. The physics is accessed by selecting Model, which provides access to the data and the methods (Java code) by which the model can evolve. A plot is a visual representation of the data and is an example of a View and *EJS* provides a graphical drag-and-drop editor for the view which eliminates much coding. It is possible to have multiple views of the same data but there is only one model. The views contains graphical user interface controls, such as buttons and input fields, that allows a user to interact with model and these controls are also created using the drag and drop editor.

This model-view-controller analysis of a program is a well known computer science paradigm and provides a solid foundation for software development.

The advantage of *EJS* for teaching computational physics is that it forces the user to separate the model into logical parts and to separate the model from the view. Students learn the logic of computer modeling using loops and control structures and study algorithms used in professional practice when building models. Students are also introduced to object-oriented programming concepts by using object properties and methods when they create user interfaces. However, little user-interface coding is required because the user interface code is created automatically by *EJS*. A button click is all that is required to produce a stand-alone ready-to-run computer program (see Figure 1.3) that implements the model.

Our choice of *Easy Java Simulations* for this text is motivated in part by its platform independence, flexible standard graphics libraries, good performance, and its availability at no cost. The popularity of the Java language and its open source nature ensures that *EJS* will continue to evolve. *EJS* users can leverage a vast collection of third-party Java libraries, including those for numerical calculations and visualization. Java also is relatively simple to learn, especially the subset of Java that we will need to simulate physical systems.

Easy Java Simulations can be downloaded from the *EJS* website and installed (unzipped) into a directory as explained in Appendix 1A. Readers who wish to use another programming language should find the algorithmic components of the Java listings in the text to be easily converted into a language of their choice.

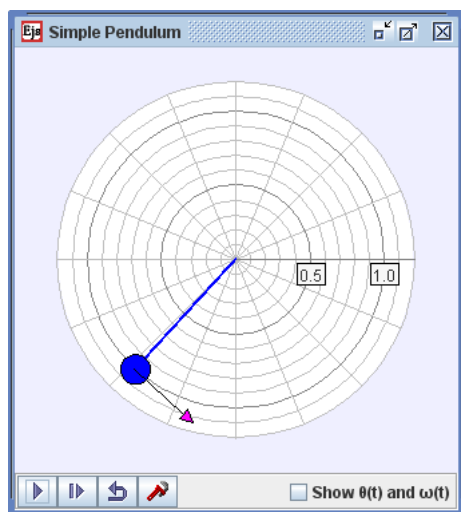



Figure 1.3: *EJS* produces ready-to-run computer programs as specified in the Model and View workpanels.



Figure 1.4: The Digital Libraries window of *EJS*. Select one of the available repositories using the combo box at the top of the window and click the *Get catalog* button to retrieve the list of available models.

1.5 How to obtain models

Although a small selection of models is copied into the source directory of your workspace (unless you unselected this option when you first ran *EJS*), a much larger selection is available on the Internet. The easiest way to access these models is through the Digital Libraries (DL) icon in the *EJS* taskbar . The taskbar icon opens a window that connects to several online digital collections. The DL connection window displayed in Figure 1.4 contains a combo box that lists these repositories. Select the Davidson College library and click the *Get catalog* button to browse the models in the DL. The left frame of the DL connection dialog shows the library catalog tree and the right frame shows information about the selected model as shown in Figure 1.5. Double-clicking a leaf (end) node in the catalog, or clicking the *Download* button, will retrieve the model and its auxiliary files from the library after asking for a storage location in the source directory in your *EJS* workspace. Because source files are usually small, the download takes place quickly. *EJS* opens the model when the download is complete and you can now inspect, run, or modify the downloaded model as you work through the problems and exercises in this text.

The **ComPADRE Pathway**, a part of the National Science Digital Library (U.S.), hosts another *EJS* DL. ComPADRE is a network of educational resources supporting teachers and students in Physics and Astronomy and the Open Source Physics (OSP) collection in ComPADRE (<http://www.compadre.org/OSP>) contains executable simulations and curriculum of interest in

physics, computation, and computer modeling. This collection is organized by subject categories and subcategories and delivers ready to run *EJS* models and source code that can be accessed from a web browser as well as from the *EJS* DL taskbar icon. We will sometimes include references to models in the ComPADRE digital library.

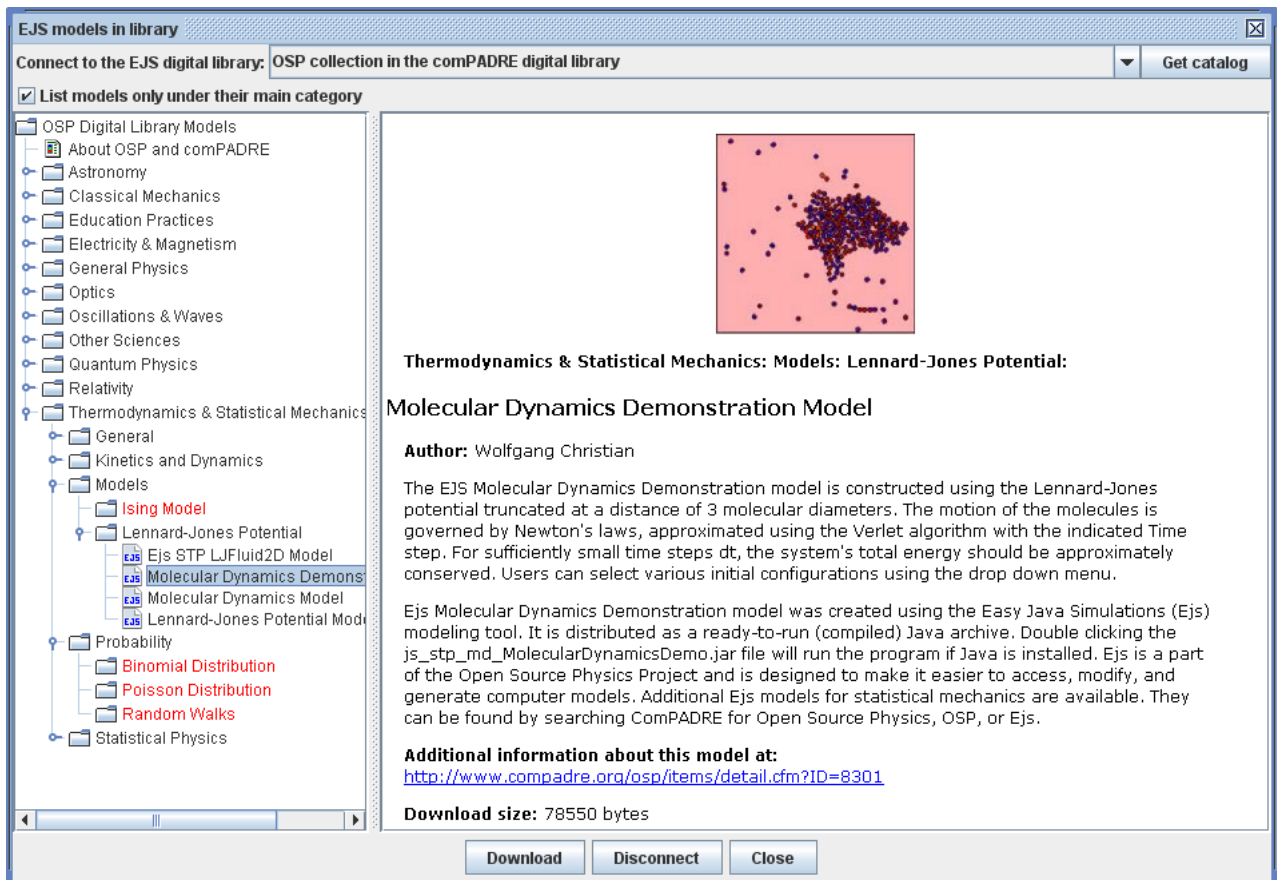


Figure 1.5: The OSP collection within the ComPADRE digital library. The collection is organized in categories and subcategories. The entry for a model provides information about the model. The collection can be accessed using *EJS* or a web browser.

Exercise 1.1. ComPADRE digital library

- Use your web browser to visit the ComPADRE website (<http://www.compadre.org/OSP>) and use the site's online search tools to find the model shown in Figure 1.5. Download the jar file for this model and run it.
- Use the DL icon in *EJS* to connect to the Davidson Digital Library and download the models for Chapter 2.



1.6 How to use this book

Science students have a rich context in which to learn programming. The past several decades of doing physics with computers has given us numerous examples that we can use to learn physics, programming, and data analysis. Unlike many programming manuals, the emphasis of this book is on learning by example. We will not discuss all aspects of Java, and this text is not a substitute for a text on Java. Think of how you learned your native language. First you learned by example, and then you learned more systematically.

Most chapters in this text begin with a brief background summary of the nature of a system and the important questions. We then introduce the computer algorithms, new syntax as needed, and discuss a sample program. The programs are meant to be read as text on an equal basis with the discussions and are interspersed throughout the text. It is strongly recommended that all the problems be read, because many concepts are introduced after you have had a chance to think about the result of a simulation.

It is a good idea to maintain a computer-based notebook to record your programs, results, graphical output, and analysis of the data. This practice will help you develop good habits for future research projects, prevent duplication, organize your thoughts, and save you time. After a while, you will find that most of your new programs will use parts of your earlier programs. Ideally, you will use your files to write a laboratory report or a paper on your work. Guidelines for writing a laboratory report are given in [Appendix 1B](#).

Many of the problems in the text are open ended and do not lend themselves to simple “back of the book” answers. So how will you know if your results are correct? How will you know when you have done enough? There are no simple answers to either question, but we can give some guidelines. First, you should compare the results of your program to known results whenever possible. The known results might come from an analytical solution that exists in certain limits or from published results. You also should look at your numbers and graphs, and determine if they make sense. Do the numbers have the right sign? Are they the right order of magnitude? Do the trends make sense as you change the parameters? What is the statistical error in the data? What is the systematic error? Some of the problems explicitly ask you to do these checks, but you should make it a habit to do as many as you can whenever possible.

How do you know when you are finished? The main guideline is whether you can tell a coherent story about your system of interest. If you have only a few numbers and do not know their significance, then you need to do more. Let your curiosity lead you to more explorations. Do not let the questions asked in the problems limit what you do. The questions are only starting points, and frequently you will be able to think of your own questions.

The following problem is an example of the kind of problems that will be posed in the following chapters. Note its similarity to the questions posed on [page 2](#). Although most of the simulations that we will do will be on the kind of physical systems that you will encounter in other physics courses, we will consider simulations in related areas, ranging from traffic flow, small world networks, and economics. Of course, unless you already know how to do simulations, you will have to study the following chapters so that you will be able to do problems like the following.

Problem 1.1. Distribution of money

The distribution of income in a society, $f(m)$, behaves as $f(m) \propto m^{-1-\alpha}$, where m is the income (money) and the exponent α is between 1 and 2. The quantity $f(m)$ can be taken to be the number of people who have an amount of money between m and $m + \Delta m$. This power law behavior of the income distribution often is referred to as Pareto's law or the 80/20 rule (20% of the people have 80% of the income), and was proposed by Vilfredo Pareto, an economist and sociologist, in the late 1800's. In the following we consider some simple models of a closed economy to determine the relation between the microdynamics and the resulting macroscopic distribution of money.

- a. Suppose that N agents (people) can exchange money in pairs. For simplicity, we assume that all the agents are initially assigned the same amount of money m_0 , and the agents are then allowed to interact. At each time step, a pair of agents i and j with money m_i and m_j is randomly chosen and a transaction takes place. Again for simplicity, let us assume that $m_i \rightarrow m'_i$ and $m_j \rightarrow m'_j$ by a random reassignment of their total amount of money, $m_i + m_j$, such that

$$m'_i = \epsilon(m_i + m_j) \quad (1.2a)$$

$$m'_j = (1 - \epsilon)(m_i + m_j), \quad (1.2b)$$

where ϵ is a random number between 0 and 1. Note that this reassignment ensures that the agents have no debt after the transaction, that is, they always are left with an amount $m \geq 0$. Simulate this model and determine the distribution of money among the agents after the system has relaxed to an equilibrium state. Choose $N = 100$ and $m_0 = 1000$.

- b. Now let us ask what happens if the agents save a fraction, λ , of their money before the transaction. We write

$$m'_i = m_i + \delta m \quad (1.3a)$$

$$m'_j = m_j - \delta m \quad (1.3b)$$

$$\delta m = (1 - \lambda)[\epsilon m_j - (1 - \epsilon)m_i]. \quad (1.3c)$$

Modify your program so that this savings model is implemented. Consider $\lambda = 0.25, 0.50, 0.75$, and 0.9 . For some of the values of λ , as many as 10^7 transactions will need to be considered. Does the form of $f(m)$ change for $\lambda > 0$?

The form of $f(m)$ for the model in Problem 1.1a can be found analytically and is known to students who have had a course in statistical mechanics. However, the analytical form of $f(m)$ in Problem 1.1b is not known. More information about this model can be found in the article by Patriarca, Chakraborti, and Kaski (see the references at the end of this chapter).

Problem 1.1 illustrates some of the characteristics of simulations that we will consider in the following chapters. Implementing this model on a computer would help you to gain insight into its behavior and might encourage you to explore variations of the model. Note that the model lends itself to asking a relatively simple “what if” question, which in this case leads to qualitatively different behavior. Asking similar questions might require modifying only a few lines of code. However, such a change might convert an analytically tractable problem into one for which the solution is unknown. ■

Problem 1.2. Questions to consider

- a. You are familiar with the fall of various objects near the earth's surface. Suppose that a ball is in the earth's atmosphere long enough for air resistance to be important. How would you simulate the motion of the ball?
- b. Suppose that you wish to model a simple liquid such as liquid Argon. Why is such a liquid simpler to simulate than water? What is the maximum number of atoms that can be simulated in a reasonable amount of time using present computer technology? What is the maximum real time that is possible to simulate? That is, if we run our program for a week of computer time, what would be the equivalent time that the liquid has evolved?
- c. Discuss some examples of systems that would be interesting to you to simulate. Can these systems be analyzed by analytical methods? Can they be investigated experimentally?
- d. An article by Post and Votta (see references) claims that "... (computers) have largely replaced pencil and paper as the theorist's main tool." Do you agree with this statement? Ask some of the theoretical physicists that you know for their opinions.

■

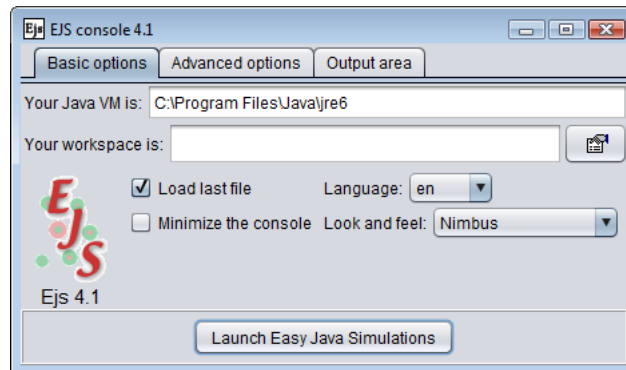
Appendix 1A: Using *Easy Java Simulations*.

EJS is a Java program that can be run under any operating system that supports a Java Virtual Machine (VM). Because Java is platform independent, the *EJS* user interface on Macintosh 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, version 1.5 or later. The Java Runtime Environment might be already installed on 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 for Linux, Unix, or Windows.
2. **Copy *EJS* to your hard disk.** You'll find *EJS* in a compressed ZIP file named **EJS_X.x_yymmdd.zip** on the *EJS* website (<http://www.um.es/fem/Ejs/>). 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 **EJS_4.1_081110**.) Uncompress this file on your computer's hard disk to create a directory called **EJS_X.x** (**EJS_4.1** in the example). This directory contains everything that is needed to run *EJS*.¹
3. **Run the *EJS* console.** Inside the newly-created **EJS_X.x** directory you will find the file **EjsConsole.jar**. Double-click it to run the *EJS* console shown in Figure 1.6.

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 1.6: The *EJS* console.

You should see the console (Figure 1.6) on your display. 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. The console displays *EJS* program information and error messages. 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 *EJS* Wiki (<http://www.um.es/fem/Ejs/>).

Before the console can run *EJS* after installation, the file chooser displayed in Figure 1.7 will appear, letting you choose the directory in your hard disk that you will use as a *workspace*.

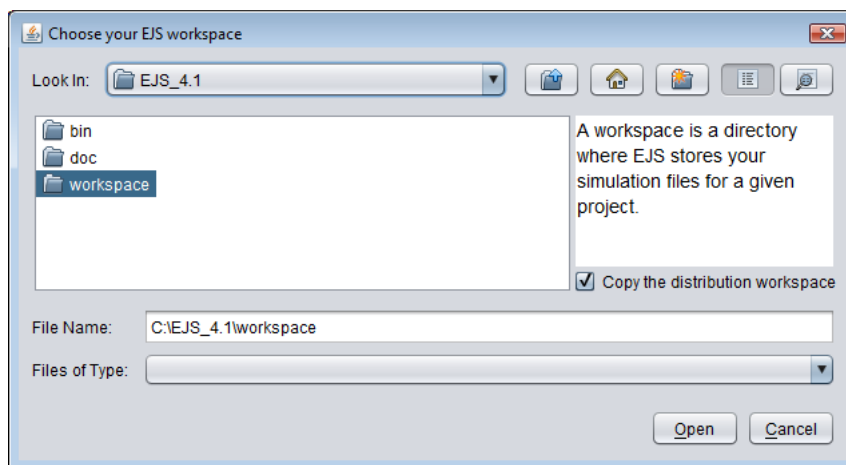


Figure 1.7: Use File chooser to select your workspace directory.

EJS uses a workspace to organize your work. A workspace is a directory in your hard disk where *EJS* stores a collection of user files such as models and their documentation. Inside a

¹In 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.

workspace directory *EJS* creates four subdirectories:

- **config** is the directory for user-defined configuration and options files.
- **export** is the target directory where *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 directory must be writable and may be anywhere on 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 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 contains 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*. Because the workspace location is stored in a user's profile, *EJS* is well suited for a multi-user computer classroom.

The first time you run *EJS* the program will also ask you your name and affiliation. This step is optional but recommended, because 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.

We are now ready to discuss the *EJS* modeling tool, displayed with annotations in Figure 1.8. Despite its simple interface, *EJS* has all the tools needed for a complete modeling cycle.

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 *Description* allows the user to create and edit a 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 *Model* workpanel 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. *View* workpanel is dedicated to 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.

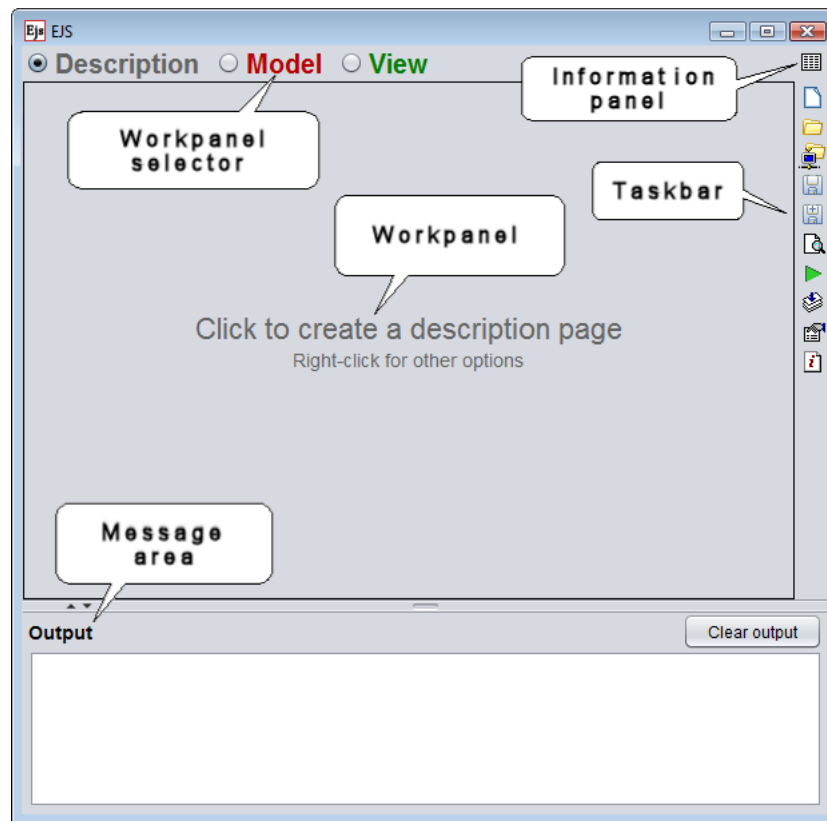




Figure 1.8: The *Easy Java Simulations* user interface with annotations.

Inspecting the Simulation

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

Click on the *Open* icon  on the *EJS* taskbar. A file dialog similar to that in Figure 1.9 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** file and initializes the workpanels. Two new “Ejs windows” appear in your display as shown in Figure 1.10. A quick warning. You can drag objects within these mock-up windows but this action will set the model’s initial conditions. It is usually better to set initial conditions using a table of variables as described in Section 1.6.

Impatient or precocious readers may be tempted to click on the green run icon  on the

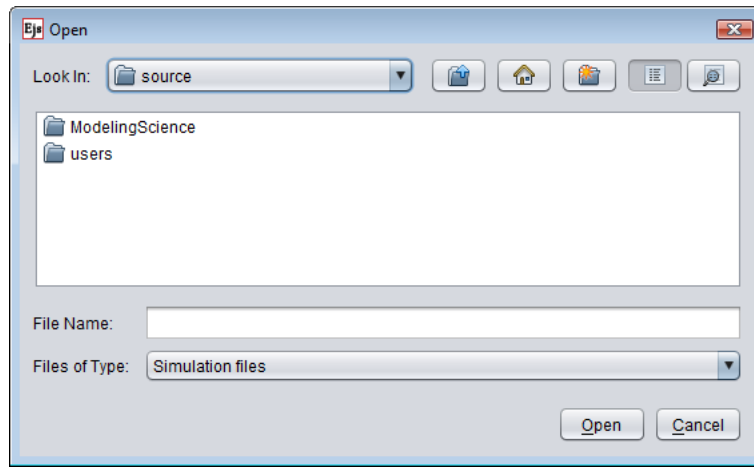


Figure 1.9: The open file dialog lets you browse your hard disk and load an existing simulation.

taskbar to execute (run) 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. In other words, the Mass and Spring model is running as a separate program independent of *EJS*. Exit the running program by closing the *Mass and Spring* window or by right clicking on the (now) red run icon ► on the *EJS* taskbar before proceeding.

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 1.11, contains a short discussion of the mass and spring model. Click on the *Activities* tab to view the second page of narrative.

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. Description pages are an essential part of the modeling process. These pages are distributed with the compiled model when the model is distributed as a Java application or as an applet.

The *Model* workpanel

The *Model* workpanel is where the model is defined. 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 the exact analytic result.

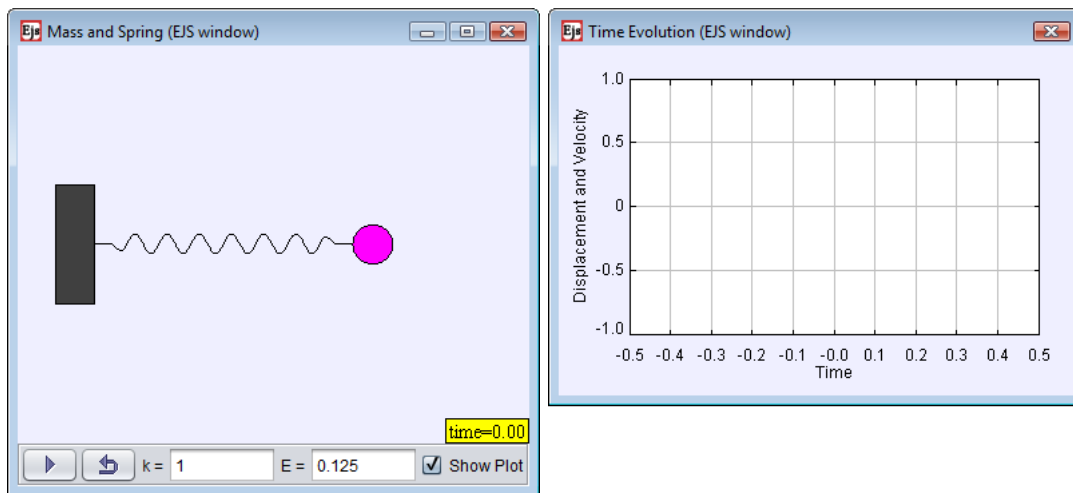


Figure 1.10: *EJS* mock-up windows of the **MassAndSpring** simulation. The title bar shows that they are *Ejs* windows and that the program is not running.

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

$$\frac{d^2 x}{dt^2} = -\frac{k}{m} (x - L). \quad (1.4)$$

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 $x - L$ 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.

Declaration of variables

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 1.12 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 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 numbers (≈ 16 significant digits), 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**.

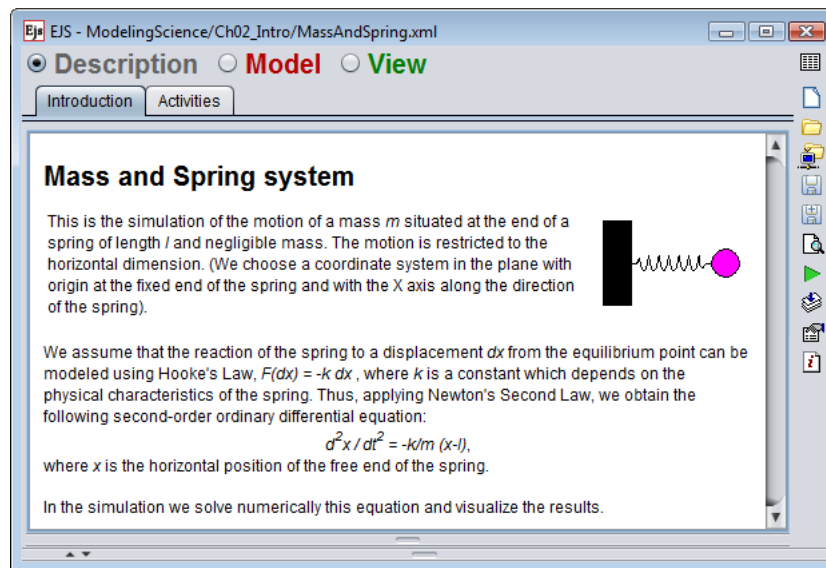


Figure 1.11: The description pages for the mass and spring simulation. Click on a tab to display the page. Right-click on a tab to edit the page.

Variables can be used as parameters, state variables, or inputs and outputs of the model. The tables in Figure 1.12 define the variables used within our model. We have declared a variable for the time, τ , for the x -position of the particle, x , and for its velocity in the x -direction, v_x . We also define variables that do not appear in (1.4). 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 provides a description of the role of each variable in the model. Clicking on a variable displays the corresponding comment.

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 $0.5*m*v_x*v_x$) 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 Java code that performs the required computation. *EJS* converts this code into a Java method² and calls this method at start-up and whenever the simulation is reset. The mass and spring does not require special initialization and this panel is empty.

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

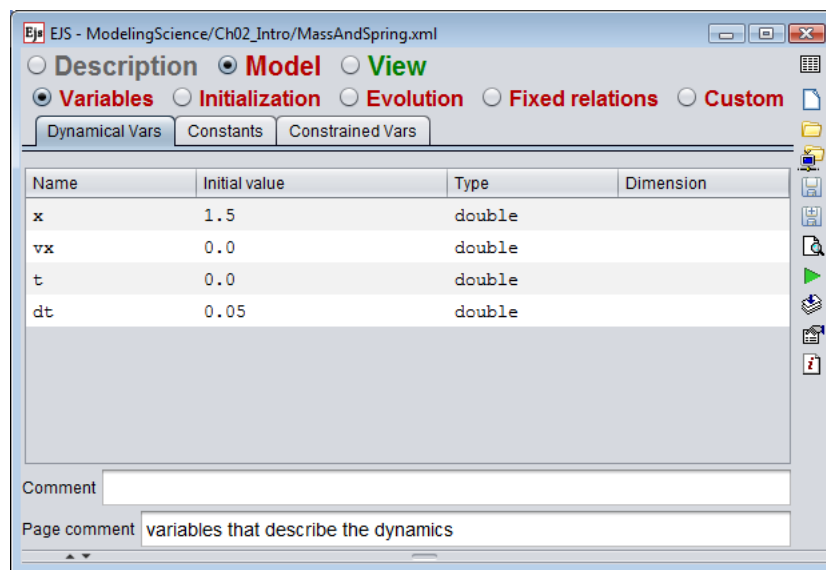


Figure 1.12: The *Model* workpanel contains five subpanels. The subpanel for the definition of mass and spring dynamical variables is displayed. Other tabs in this subpanel define additional variables, such as the natural length of the spring L and the energy E .

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. We will use this option frequently for models not based on ordinary differential equations. There is, however, a second option that allows us to enter ordinary differential equations (ODEs) such as (1.4), 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.

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 (1.4), 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:

$$\frac{d x}{d t} = v_x \quad (1.5)$$

$$\frac{d v_x}{d t} = -\frac{k}{m}(x - L). \quad (1.6)$$

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 1.13. Notice that the ODE editor displays (1.5) and (1.6) (using the $*$ character to denote multiplication). Fields near the top of the editor specify the independent variable t and the variable increment dt . Numerical

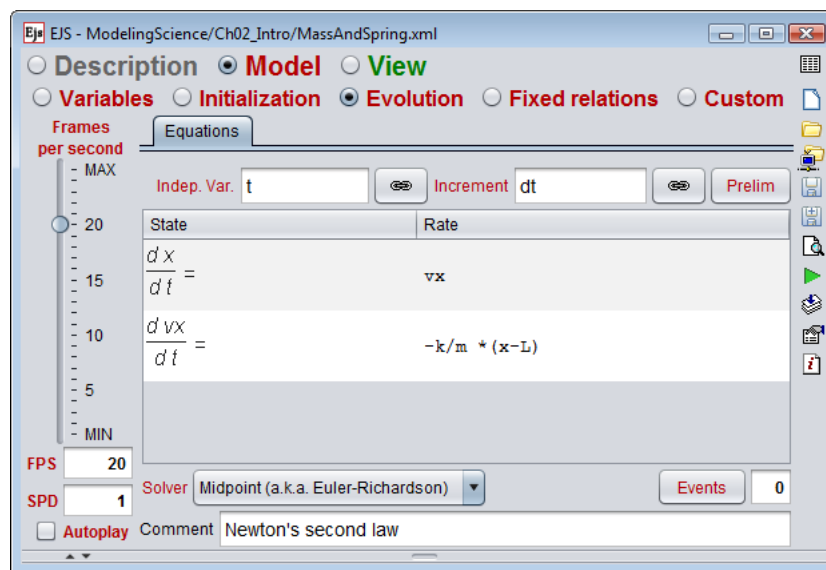


Figure 1.13: The ODE evolution panel showing the mass and spring differential equation and the numerical algorithm.

algorithms approximate the exact ODE solution by advancing the state in discrete steps and the increment determines this step size. A dropdown menu at the bottom of the editor lets us select numerical algorithm (the ODE solver) that advances the solution from the current value of the time, t , to the next value, $t + dt$. The events field at the bottom of the panel tells us that we have not defined any events for this differential equation.

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 the 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 solves the mass and spring ODE model without our having to explicitly program the numerical solution algorithm. 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 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.³

Relations among variables

Not all variables within a model are computed using an algorithm in 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.

$$T = \frac{1}{2}mv_x^2, \quad (1.7)$$

$$V = \frac{1}{2}k(x - L)^2, \quad (1.8)$$

$$E = T + V. \quad (1.9)$$

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

The *Fixed relations* panel shown in Figure 1.14 is used to write relations among variables. Notice how easy it is to convert (1.7) through (1.9) into Java syntax. Be sure to use the multiplication character `*` and to place a semicolon at the end of each Java statement.

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, such as energy conservation, must *always* hold and there are other ways, such as mouse actions and keyboard input, to affect state variables. For example, dragging the mass changes the model's x variable and this change affects the energy. *EJS* automatically evaluates the fixed 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.

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 doesn't require additional methods.

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

³Although the mass and spring model can be solved with a simple ODE algorithm, our numerical methods library contains very sophisticated algorithms that can be applied to large systems of differential equations. See Chapter 4.

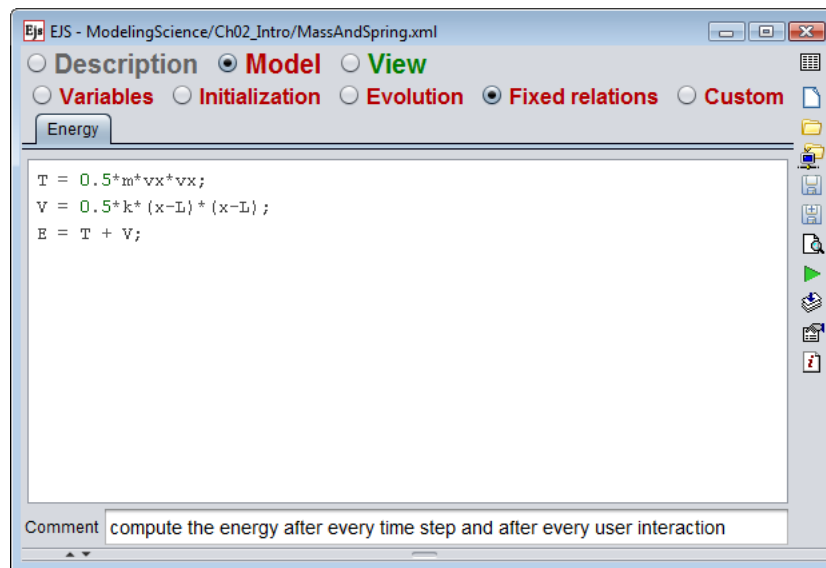


Figure 1.14: Energy relations for the mass and spring model. Statements in a fixed relations panel are evaluated when a user interacts with the program in any way and after every evolution step.

programming. Figure 1.10 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 1.15, 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 Chapter 2.

The *Tree of elements* shown on the left side of Figure 1.15 displays the structure of the mass and spring model’s user interface. Notice that the simulation has two windows, a **Frame** and a **Dialog**, which 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 determine the element’s appearance and behavior. We can edit these properties by double clicking on an 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 `massShape2D` element (the

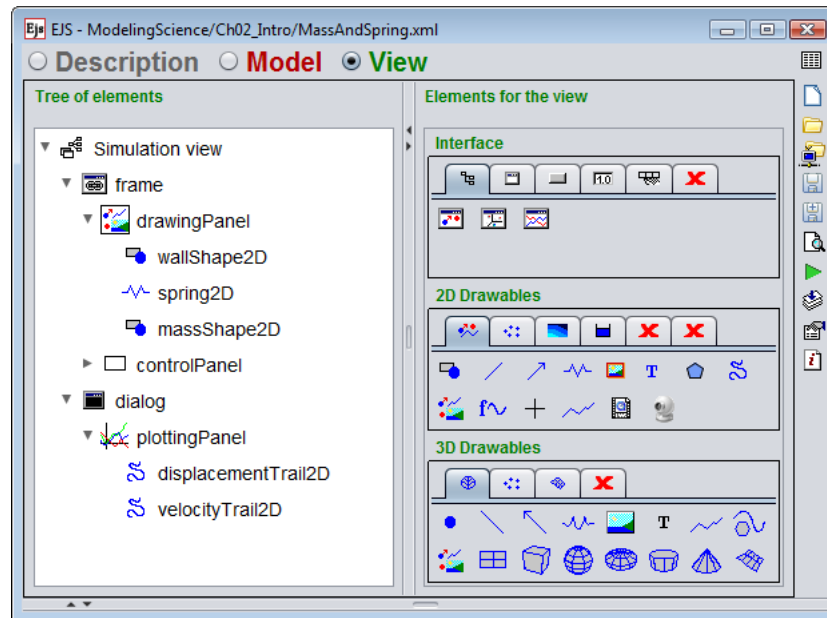


Figure 1.15: The *View* workpanel showing the *Tree of elements* for the mass and spring user interface.

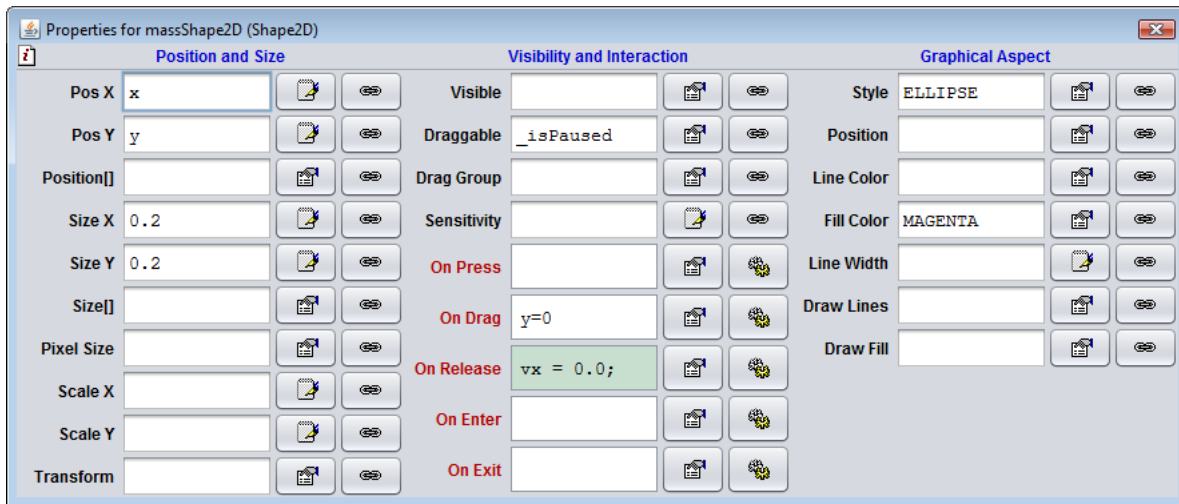
“Shape2D” suffix added to the element’s name identifies the type of element) in the tree to display the element’s properties inspector. This element is the mass that is attached at the free end of the spring. The massShape2D’s table of properties appears as shown in Figure 1.16.

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 their 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.

Figure 1.16: The table of properties of the `massShape2D` element.

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 1.2. 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? ■



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 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 variables that depend on the solution of the differential equations. 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 1.14) 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 1.6.

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 click of a mouse.

Running a simulation initializes its variables and executes the fixed relations to insure that the model is in a consistent state. The model's 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 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 1.17.

Stop the simulation and right-click the mouse over any of the drawing 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.

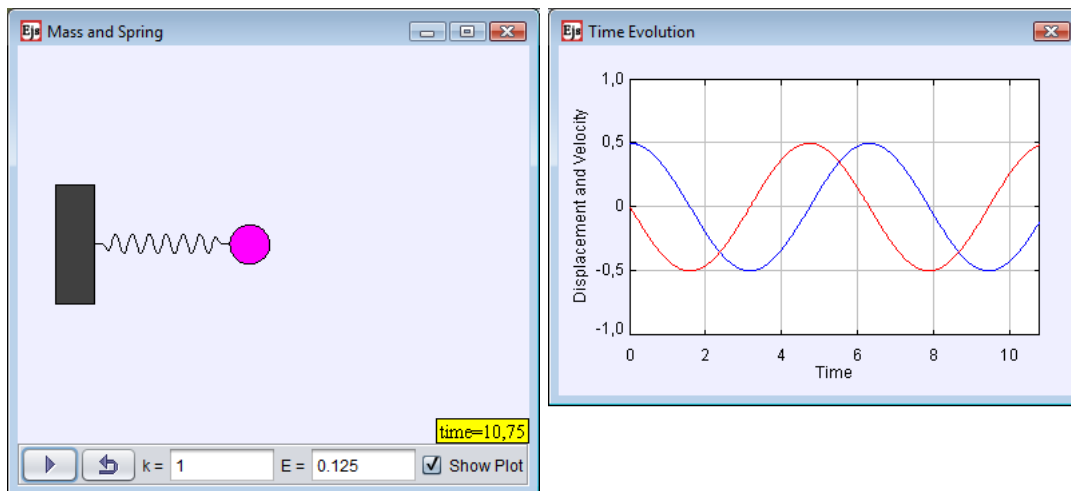



Figure 1.17: The mass and spring simulation displays an interactive drawing of the model and a graph with displacement and velocity data.

Distributing the Simulation

Simulations created with *EJS* are stand-alone Java programs that can be distributed without *EJS* for others to use. The easiest way to distribute a program is to package it 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 is the **export** directory at the root of your workspace, but you can choose any directory and any package name. The stand-alone jar file is ready to be distributed. Other distribution mechanisms are available by right-clicking on the icon.

Exercise 1.3. 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. ■

Although the mass and spring jar file is a ready to use and ready 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, a dialog asks if it should 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 problem.

Exercise 1.4. Extracting a model Run the stand-alone jar file containing the mass and spring model created in Exercise 1.3. 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*. ■

EJS is designed to be both a modeling and an authoring tool. We suggest that you experiment with it to learn how you can create and distribute your own models as you work through the exercises, problems, and projects in this book.

Appendix 1B: Documentation and Reports

Documentation and laboratory reports should reflect clear writing style and obey proper rules of grammar and correct spelling. Write in a manner that can be understood by another person who has not researched the model. Documentation is packed with the model and is often duplicated in the laboratory report. In the following, we give a suggested format for your documentation and reports.

Title, authorship, and credits. Give your model a meaningful name and enter bibliographic information into the *EJS* information dialog (top button on the *EJS* toolbar) and on the first documentation page.

Introduction. Briefly summarize the nature of the physical system, the basic numerical method or algorithm, and the interesting or relevant questions. This summary should be included in Documentation workpanel so that it is packaged with the compiled model.

Model. Describe the algorithm and how it is implemented. In some cases this explanation can be given in the *EJS* workpanels. Variables defined in tables should be given meaningful names and their purpose should be stated in the comment field. Code should be annotated in a way that is as self-explanatory as possible. Be sure to discuss any important features of your implementation.

Verification of program. Confirm that your model is not incorrect by considering special cases and by giving at least one comparison to a hand calculation or known result.

Data. Show the results of some typical runs in graphical or tabular form. Additional runs can be included in an appendix. All runs should be labeled, and all tables and figures must be referred to in the body of the text. Each figure and table should have a caption with complete information, for example, the value of the time step.

Analysis. In general, the analysis of your results will include a determination of qualitative and quantitative relationships between variables, and an estimation of numerical accuracy.

Interpretation. Summarize your results and explain them in simple physical terms whenever possible. Specific questions that were raised in the assignment should be addressed here. Also give suggestions for future work or possible extensions. It is not necessary to answer every part of each question in the text.

Critique. Summarize the important physical concepts for which you gained a better understanding and discuss the numerical or computer techniques you learned. Make specific comments on the assignment and your suggestions for improvements or alternatives.

Log. Keep a log of the time spent on each assignment and include it with your report.

References and suggestions for further reading

General References on Physics and Computers

Richard E. Crandall, *Projects in Scientific Computation*, Springer-Verlag (1994).

Paul L. DeVries, *A First Course in Computational Physics*, John Wiley & Sons (1994).

Alejandro L. Garcia, *Numerical Methods for Physics*, second edition, Prentice Hall (2000). Matlab, C++, and Fortran are used.

Neil Gershenfeld, *The Nature of Mathematical Modeling*, Cambridge University Press (1998).

Nicholas J. Giordano and Hisao Nakanishi, second edition, *Computational Physics*, Prentice Hall (2005).

Dieter W. Heermann, *Computer Simulation Methods in Theoretical Physics*, second edition, Springer-Verlag (1990). A discussion of molecular dynamics and Monte Carlo methods directed toward advanced undergraduate and beginning graduate students.

David Landau and Kurt Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, Cambridge University Press (2001). The authors emphasize the complementary nature of simulation to theory and experiment.

Rubin H. Landau, *A First Course in Scientific Computing*, Princeton University Press (2005).

P. Kevin MacKeown, *Stochastic Simulation in Physics*, Springer (1997).

Tao Pang, *Computational Physics*, Cambridge (1997).

Franz J. Vesely, *Computational Physics*, second edition, Plenum Press (2002).

Michael M. Woolfson and Geoffrey J. Perl, *Introduction to Computer Simulation*, Oxford University Press (1999).

Other Articles

H. Gould, “Computational physics and the undergraduate curriculum,” *Computer Physics Communications* **127** (1), 6–10 (2000).

Brian Hayes, “g-OLGY,” *Am. Scientist* **92** (3), 212–216 (2004) discusses the g -factor of the electron and the importance of algebraic and numerical calculations.

Problem 1.1 is based on a paper by Marco Patriarca, Anirban Chakraborti, and Kimmo Kaski, “Gibbs versus non-Gibbs distributions in money dynamics,” *Physica A* **340**, 334–339 (2004), or cond-mat/0312167.

An interesting article on the future of computational science by Douglass E. Post and Lawrence G. Votta, “Computational science demands a new paradigm,” *Physics Today* **58** (1), 35–41 (2005) raises many interesting questions.

Ross L. Spencer, “Teaching computational physics as a laboratory sequence,” *Am. J. Phys.* **73**, 151–153 (2005).