Chapter 1

Introduction

14 September 2005

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 the usual 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, and Mathematica 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, $\sin x = x - x^3/3! + x^5/5! + \ldots$, 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.

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

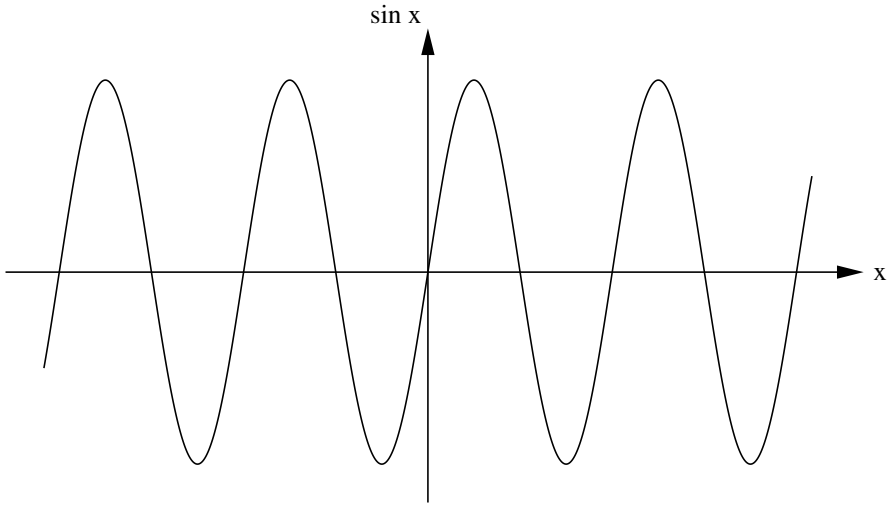

Figure 1.1: What is the meaning of the sine function?

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 displacement of the particle leads to a small change in the force. However, many natural phenomena are *nonlinear*, and a small change in a variable might produce a large change in another. Because

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.

relatively 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. A similar problem is given at the end of this chapter.

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 computer 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 computer simulation can serve as a bridge between laboratory experiments and theoretical calculations. In some cases we can obtain essentially exact results 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. On the other hand, we sometimes 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 can 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 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.

Most modern languages incorporate *object-oriented* features. The idea of object-oriented programming is that functions and data are grouped together in an *object*, rather than treated separately. A program is a structured collection of objects that communicate with each other causing the internal state within a given object to change. A fundamental goal of object-oriented design is to increase the understandability and reusability of program code by focusing on what an object does and how it is used, rather than how an object is implemented.

Our choice of Java for this text is motivated in part by its platform independence, flexible standard graphics libraries, good performance, and its no cost availability. The popularity of Java ensures that the language will continue to evolve, and that programming experience in Java is a valuable and marketable skill. The Java programmer can leverage a vast collection of third-party 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.

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. Despite the high level of abstraction and platform independence, the performance of Java is becoming comparable with native languages. If a project requires more speed, the computationally demanding parts of the program can be converted to C/C++ or Fortran.

Readers who wish to use another programming language should find the algorithmic components of the Java program listings in the text to be easily converted into a language of their choice.

1.4 Object oriented techniques

If you already know how to program, try reading a program that you wrote several years, or even several weeks, ago. Many of us would not be able to follow the logic of our own program and would have to rewrite it. And your program would probably be of little use to a friend who needs to solve a similar problem. If you are learning programming for the first time, it is important to learn good programming habits to minimize this problem. One way is to employ object-oriented techniques such as encapsulation, inheritance, and polymorphism.

Encapsulation refers to the way that an object's essential information is exposed through a well-documented interface, but unnecessary details of the code are hidden. For example, we can model a particle as an object. Whenever a particle moves, it calculates its acceleration from the total force on it. Someone who wishes to use the trajectory of the particle, for example to animate the particle's trajectory, needs to refer only to the interface and does not need to know how the trajectory is calculated.

Inheritance allows a programmer to add capabilities to existing code without having to rewrite it or even know the details of how the code works. For example, you will write programs that show the evolution of planetary systems, quantum mechanical wave functions, and molecular models. Many of these programs will use (extend) code in the Open Source Physics library known as an `AbstractSimulation`. This code has a timer that periodically executes code in your program and then refreshes the on-screen animation. Using the Open Source Physics library will let you focus your efforts on programming the physics, because it is not necessary to write the code to produce the timer or to refresh the screen. Similarly, we have designed a general purpose graphical user interface (GUI) by extending code written by Sun Microsystems known as a `JFrame`. Our GUI has the features of a standard user interface such as a menu bar, minimize button, and title, even though we did not write the code to implement these features.

Polymorphism helps us to write reusable code. For example, it is easy to imagine many types of objects that are able to evolve over time. In Chapter 15 we will simulate a system of particles using random numbers rather than forces to move the particles. By using polymorphism, we can write general purpose code to do animations with both types of systems.

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.

Although using an object oriented language makes it easier to write well structured programs, it does not guarantee that your programs will be well written or even correct. The single most important criterion of program quality is readability. If your program is easy to read and follow, it is probably a good program. There are many analogies between a good program and a well-written paper. Few papers and programs come out perfectly on their first draft, regardless of the techniques and rules we use to write them. Rewriting is an important part of programming.

1.5 How to use this book

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 1A.

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 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 $m_i \to m_i'$ and $m_j \to m_j'$ by a random reassignment of their total amount of money, $m_i + m_j$, such that

$$m_i' = \epsilon(m_i + m_j) \tag{1.1a}$$
$$m_j' = (1-\epsilon)(m_i + m_j), \tag{1.1b}$$

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

$$m_i' = m_i + \delta m \tag{1.2a}$$
$$m_j' = m_j - \delta m \tag{1.2b}$$
$$\delta m = (1-\lambda)[\epsilon m_j - (1-\epsilon)m_i]. \tag{1.2c}$$

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: Laboratory reports

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 done the research. In the following, we give a suggested format for your reports.

Introduction. Briefly summarize the nature of the physical system, the basic numerical method or algorithm, and the interesting or relevant questions.

Method. Describe the algorithm and how it is implemented in the program. In some cases this explanation can be given in the program itself. Give a typical listing of your program. Simple modifications of the program can be included in an appendix if necessary. The program should include your name and date, and be annotated in a way that is as self-explanatory as possible. Be sure to discuss any important features of your program.

Verification of program. Confirm that your program 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

Programming

We list some of our favorite Java programming books here. The online Java documentation provided by Sun at <java.sun.com/docs/> is essential (look for API specifications), and the tutorial, <java.sun.com/docs/books/tutorial/>, is very helpful. There are many other useful tutorials on the Web.

Joshua Bloch, *Effective Java*, Addison-Wesley (2001). This excellent book is for advanced Java programmers and should be read after you have become familiar with Java.

Rogers Cadenhead and Laura Lemay *Teach Yourself Java in 21 Days*, fourth edition, Sams (2004), An inexpensive self-study guide that uses a step by step tutorial approach to cover the basics.

Stephen J. Chapman, *Java for Engineers and Scientists*, second edition, Prentice Hall (2004).

Wolfgang Christian, *Open Source Physics: A User's Guide with Examples*, Addison-Wesley (2006). This guide is a useful supplement to our text.

Bruce Eckel, *Thinking in Java*, third edition, Prentice Hall (2003). This text discusses the finer points of object-oriented programming and is recommended after you have become familiar with Java. See also <www.mindview.net/Books/>.

David Flanagan, *Java in a Nutshell*, fifth edition, O'Reilly (2005) and *Java Examples in a Nutshell*, third edition, O'Reilly (2004). A fast-paced Java tutorial for those who already know another programming language.

Brian D. Hahn and Katherine M. Malan, *Essential Java for Scientists and Engineers*, Butterworth-Heinemann (2002).

Cay S. Horstmann and Gary Cornell, *Core Java 2: Fundamentals* and *Core Java 2: Advanced Features*, both in seventh edition, Prentice Hall (2005). A two-volume set that covers all aspects of Java programming.

Patrick Niemeyer and Jonathan Knudsen, *Learning Java*, second edition, O'Reilly (2002). A comprehensive introduction to Java that starts with `HelloWorld` and ends with a discussion of XML. The book contains many examples showing how the core Java API is used. This book is one of our favorites for beginning Java programmers. However, it might be intimidating to someone who does not have some familiarity with computers.

Sherry Shavor, Jim D'Anjou, Pat McCarthy, John Kellerman, and Scott Fairbrothe, *The Java Developer's Guide to Eclipse*, Addison-Wesley Professional (2003). A good reference for the open source Eclipse development environment. Check for new editions because Eclipse is evolving rapidly.

General References on Physics and Computers

A listing of textbooks on computational physics is available at <www.opensourcephysics.org/sip/books/>.

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*, second edition, Cambridge University Press (2005). 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 University Press (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 References

Ruth Chabay and Bruce Sherwood, Matter & Interactions (John Wiley & Sons, 2002). This two-volume text uses computer models written in VPython to present topics not typically discussed in introductory physics courses.

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

Brian Hayes, "g-OLOGY," 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).