Teaching Computational Physics to Undergraduates

Jan Tobochnik
Kalamazoo College, Department of Physics, Kalamazoo, MI 49006
jant@kzoo.edu

Harvey Gould
Clark University, Department of Physics, Worcester, MA 01610
hgould@clarku.edu

Abstract

We discuss the advantages of incorporating computational physics into the undergraduate curriculum and the special advantages of a separate course that emphasizes computer simulations.

1 Introduction

Computational physics is now recognized as an independent way of doing physics. Until recently, the more familiar use was as a complement to doing experiment and analytical theory. Almost all analytical theories require the help of a computer to complete the calculations. For example, many theoretical problems require numerical calculations such as the evaluation of an integral, obtaining the roots of an equation, and matrix manipulations. In some cases, such as diagrammatic calculations, symbolic manipulation programs are essential for keeping track of the different classes of diagrams. On the experimental side, computers are essential for the control of experiments and the collection and analysis of data. Computers are being used in this context to implement statistical procedures for extracting a signal from noise, do Fourier transforms to obtain power spectra, fit data to functional forms, and plot data in a form that helps us to interpret its meaning.

These uses of the computer in scientific research can be viewed as specific tools, much like an oscilloscope is used to make voltage measurements or a mathematical transformation is used to simplify a mathematical expression. However, computational physics also includes a fundamentally different way of doing physics that goes beyond using the computer as a specific tool. We have in mind the part of computational physics that is called computer simulations [1], in contrast to many of the tasks listed above which we classify as numerical analysis. One distinction between numerical analysis and computer simulation is that in the former, the user does as much as possible analytically before the problem is given to a computer. In some ways, the numerical analysis mode can be thought of as an extension of techniques used before computers were invented. For example, in the past a human calculator would generate tables of sines and cosines; now a computer can generate these values very quickly. However, there are ways of doing numerical analysis that were inconceivable before the advent of computers. For example, the generation of high-order Feynman diagrams and the evaluation of the corresponding integrals using computational graph theory and Monte Carlo methods. The distinction between numerical analysis and simulation is often one of emphasis.
Computer simulations are a way of doing physics that is distinct from the way physics was done before the advent of computers. In a computer simulation we might use various numerical methods and even some symbolic manipulations. However, the style and motivation is quite different and is analogous to a laboratory experiment. For this reason computer simulations are sometimes referred to as *computer experiments*. Simulations begin with the development of a model that can be represented by an algorithm. The model plays a role analogous to the sample or physical system of interest in an experimental system. Sometimes these models are formulated in ways that are related to traditional ways of doing theoretical physics. For example, a molecular dynamics simulation involves solving Newton’s equations of motion for many particles. In other cases, the models are designed with the capabilities of the computer in mind. Examples include cellular automata models of fluids, traffic flow, and granular material.

In analogy to building the physical apparatus for an experiment, we write programs that implement the algorithms. Just as we calibrate a measurement apparatus, we have to test our program and compare its outcome with known results in limiting cases. Then data is collected as the system evolves, and finally, the data is analyzed and displayed. In an ideal world, we would be finished, but in the real world, our initial results frequently lead to further improvements in the program and more data collection.

Computer simulations also have some features that are distinct from laboratory experiments. For example, each new kind of laboratory measurement requires a new piece of equipment and may preclude other measurements. In a computer simulation a new measurement requires only some additional code within the existing program. Simulations also allow us to probe the microscopics of a system, which may or may not be possible in experimental systems.

One advantage of analytical solutions is that a solution frequently can be written in terms of a parameter so that more than one case is readily available. In contrast, separate simulations frequently must be done for each value of the parameters of interest. However, simulations have the advantage that many modifications of the model require simple changes in the program (for example changing the force law in a molecular dynamics simulation), whereas even minor changes in a theoretical model can make an analytical calculation intractable. Frequently, analytical calculations require approximations whose consequences are not known. In contrast, simulations frequently use numerical procedures that are exact in principle, and the results are only approximate due to statistical errors and limitations due to the effects of finite time and size.

The above discussion describes in general terms the features of simulations. As an example consider the simulation of a liquid with spherically symmetric intermolecular forces. A simple model of such a liquid would consist of $N$ particles enclosed in a region $V$ interacting with a two-body potential such as the Lennard-Jones 6-12 potential. This model can be implemented by two general methods: molecular dynamics [2] and Monte Carlo [3]. There are advantages to both methods. In the molecular dynamics approach, Newton’s dynamical equations of motion are integrated
numerically. External constraints such as a connection to a heat bath at constant temperature can be implemented using a generalization of the usual molecular dynamics algorithm. All static and dynamical quantities can be measured including the temperature, pressure, radial distribution function, static structure function, diffusion constants, relaxation times, viscosity, and elastic constants. Measurement methods also are available for calculating free energies and entropies. The Monte Carlo approach samples from the appropriate distribution function, and thus the dynamics of the particles is not realistic. However, all static quantities can be computed and some dynamical quantities such as diffusion constants can be computed up to an overall scale factor. Monte Carlo calculations sometimes require less computational effort than molecular dynamics calculations, and are frequently easier to program [3].

In either method we have access to the positions of the particles at all times, and hence we can compute quantities that have no direct analog in experiments but might help us understand the behavior of the system. Finally, animations of the trajectories can provide insight into the structure and dynamics that are not obvious from measurements of macroscopic quantities. For example, there are theoretical arguments [4] that suggest the presence of clusters or solid-like regions of all length scales in supercooled liquids near the glass transition. Simulations done by our group have indicated the existence of these clusters and recently have shown how these clusters can be determined indirectly in laboratory experiments [5].

The most serious limitation of this type of simulation is that the number of particles is limited to a few thousand for educational purposes and in the range $10^6$ – $10^9$ for research purposes. Away from phase transitions, this limitation is not serious, because the correlation length is very small. Near phase transitions special techniques such as finite size scaling are necessary to obtain important quantities such as the critical exponents near a continuous phase transition. Similarly, a typical simulation is run for only a few nanoseconds in real time, which may be insufficient in the presence of critical slowing down or near the glass transition. This limitation makes a molecular dynamics simulation of protein folding impractical for the foreseeable future.

Information at many different levels can be obtained by using computer simulations in conjunction with theory and experiment. For example, we can test model force laws by comparing the thermodynamics of a simulation with results from experiment. We can do simulations and measure quantities that would be expensive to measure in the laboratory. If we find novel effects or universal quantities in the simulation, then they can be used as a guide to setting up an experiment. One of the earliest examples of this synergy was the discovery in simulations of long-time tails in the velocity autocorrelation function of a liquid [6]. Sometimes experiments find novel behavior and a simulation of simplified models can help determine whether the behavior is peculiar to the specific experimental system or has a more universal character. For example, one of us has been doing simulations of inelastic steel balls moving on a vibrating inclined plane with the goal of gaining a better understanding of granular particle dynamics. Various patterns and an unusual spatial dependence of the density has been found in experiments, but it is not clear at present
whether these patterns are peculiar to steel balls or a general phenomena found in granular systems [7].

Simulations are also useful for testing theoretical ideas. For example, the Kosterlitz-Thouless theory of the planar model predicts that a phase transition occurs due to the unbinding of vortex-antivortex pairs. This prediction and others were tested and confirmed by Monte Carlo simulations [8]. A more difficult problem has been to test the validity of extensions of this model to two-dimensional melting. The results from both experiment and simulation are still inconclusive.

Another theoretical idea that has been tested is the ability of the standard model of quarks to have both asymptotic freedom (the quarks appear to be noninteracting at short distances) and confinement (individual quarks are never seen). When implemented as a lattice gauge theory, these requirements translate into a nonabelian lattice gauge model that has no phase transition at any nonzero temperature [9].

Simulations can also test theoretical approximation methods. For example, many years ago analytical calculations were performed on the two-dimensional, three-component Heisenberg spin model, and an estimate for the susceptibility at all temperatures was made by ensuring that the high and low temperature expansions matched at a certain temperature. The results were off by many orders of magnitude, as determined by Monte Carlo calculations that were able to determine definitively where the high and low temperature expansions were valid [10].

The above brief discussion, mostly in our areas of interest, should make it clear that computer simulations play an important role in physics and other sciences [11], and are likely to become even more important as the capability of computers increases. However, at the undergraduate level computer simulations are almost exclusively used as a teaching aide and not as a part of the undergraduate physics curriculum in the United States [12]. There are a variety of simulations such as Interactive Physics [13], the CUPS programs published by Wiley, and the programs available from Physics Academic Software [14]. Although these programs are useful for illustrating a variety of physical ideas, they do not help students learn how to create such programs and their educational value is not known. Without actually being involved in program development, it is difficult to discern whether an unusual result is due to some interesting physical behavior, the implementation of the model, a limitation of the model, the accuracy and/or stability of an algorithm, truncation error, a programming error, or a syntax error in the computer code. In addition, writing programs can be very helpful in making many theoretical relations more meaningful. Implementing them on a computer is one of the most direct and enlightening ways of using results from theory.

The undergraduate physics curriculum at most universities is only slightly different than it was before the widespread availability of computers. Most courses are theoretically oriented, and even much of the laboratory work has changed only to the extent that more sophisticated electronic equipment is used and the student is further removed from direct measurements than in the past. For example, instead of a student taking measurements, the data is collected directly by a computer. This use eliminates much tedium, but it also eliminates at least some of the
value of many laboratory experiments. Just as we stated that a student not involved in program development misses much of the learning that can come from a simulation, the same is true of an experiment where the student has very little control over the setup of the experiment and is only vaguely aware of how the measurements are actually made.

Most of the traditional topics such as mechanics, electromagnetism, waves, quantum mechanics, statistical mechanics, and thermodynamics are taught more or less the same as they were in the past. In many cases these courses are taught as mathematical physics courses with only occasional references to actual experimental data. One of the reasons for this emphasis is that the only systems that can be handled analytically using the mathematical tools available to students are idealized systems for which there is not much experimental data. The use of simulations provides an opportunity to broaden the range of models that can be discussed and allows us to compare theoretical results with a wider range of experimental data.

Computational physics with an emphasis on computer simulations should be incorporated into the physics curriculum as early as possible so that it can help modernize and rejuvenate the rest of the physics curriculum. In the rest of this article we will describe some ways of doing so.

2 Where in the curriculum should computational physics be taught?

Unlike other scientific disciplines, physics is defined less by what part of nature is studied than by how nature is studied. For example, biologists study living things, geologists study the earth, and chemists study molecules. However, physicists study all of these areas and others, but in different ways. Physicists are primarily interested in the universal properties of nature and the causes of this universality. Other disciplines usually are more concerned with specific details about the systems they study, and for that reason courses in other disciplines typically appear more modern than courses in physics. Biology students may learn about some of the newest developments in genetics or molecular biology, whereas physics undergraduates rarely discuss, except very descriptively, the newest developments in physics. Learning physics is about learning skills not about accumulating facts.

This focus on skills in physics education has led in part to inertia in the physics curriculum. Most physicists believe that there is a sequential series of steps that students must go through in the development of increasingly sophisticated tools of analysis. At the introductory level algebra is used, then single variable calculus, then vector calculus, and finally abstract linear algebra methods. Partial differential equations and complex analysis are introduced near the end of the undergraduate curriculum.

In addition to increasing the level of mathematical sophistication as students move through the curriculum, there is an increase in the level of abstraction. For example, in mechanics students first learn about force,
then about energy, and then about the Lagrangian which has no direct physical interpretation.

This focus on skills is what makes a physics education so valuable. One can always learn more information, but learning a new skill usually requires expert instruction. However, the downside of the present focus on analytical techniques is that many experimental and computational skills have been neglected. Even basic physical reasoning skills have been neglected, and many students have much difficulty making qualitative arguments or back of the envelope calculations. It is common for students to have had so little exposure to experimental data that they have no feel for numerical values of physical quantities. In addition, the focus on analytical skills has weighted the choice of problems to idealized situations in mechanics, electromagnetism, and quantum mechanics.

The justification for this focus is that the fundamental ideas are best illuminated by looking at simple situations. But the justification gets turned on its head when students are asked to solve complicated and tricky problems where no new physics is learned. We argue that the same level of mathematical sophistication can be maintained by reducing the amount of time spent on solving complicated but still unrealistic problems. Sophisticated mathematics is needed mostly to provide the general formalism for our discipline. However, when it is time to apply the formalism, it is best to move quickly to numerical solutions. An extreme example is the solution of the two-dimensional Ising model. The Onsager solution clearly does not belong in the undergraduate curriculum (and it is unclear how useful it is at the graduate level). However, learning how to do a Monte Carlo simulation is not only useful for extracting information about the Ising model, but also for obtaining a deeper understanding of statistical mechanics. Simulations of nonlinear dynamical systems can provide more insight and physical understanding than complicated analytical approximation schemes, and many of the series solutions in electromagnetism could be replaced by learning numerical techniques for boundary value problems.

We suggest that increasing mathematical sophistication should remain in the undergraduate physics curriculum, but should be used primarily for providing the formal structure of the theory. The primary problem solving tools should involve computer simulations and numerical analysis. These tools would be used in conjunction with qualitative physical reasoning, mathematical analysis in the context of illustrative problems, and the availability of more experimental data. This approach more closely approximates what is done in research, and thus provides a natural model for education.

How should this change be implemented? The ideal time to introduce computational methods is in the first-year in the same spirit as we introduce students to calculus and laboratory work. One way would be for students to do several computer labs in their introductory courses and learn how to convert Newton’s second law to a simple numerical algorithm. Later in the year they could use simple Monte Carlo methods to model the approach to equilibrium of many body systems. However, the reality at many institutions is that because the introductory course is overloaded with too many other demands and has such a diverse audience,
even asking students to write simple programs or use spreadsheets is too ambitious. And we are doubtful that the students would learn very much from these exercises. At present, some institutions are using simulations such as Interactive Physics in various contexts in the introductory course, and it is likely that such use is the most that can be done at many places.

An alternative is to introduce computer-related problems throughout the upper level curriculum. Such a approach might be suitable if it were possible to enlist the majority of the faculty teaching undergraduate physics to use computational methods in their courses. However, we know of only a few institutions that have been successful using this approach. One difficulty is that students must learn how to program somewhere, usually in an introductory programming course taught by computer science faculty. Although physics majors should be encouraged to take more advanced computer science courses, introductory programming courses usually serve a very diverse group. Our experience is that most students learn only the syntax of a language in this context, but none of the algorithms relevant to physics. Another obstacle is that the time needed to introduce meaningful change into the curriculum is very time consuming, even if the instructor has the knowledge and interest.

The approach that we claim is the most desirable way at most institutions is to offer a separate laboratory-based course that explicitly teaches students how to write and use computer simulations. Ideally, such a course should be available for first-year students, who might be taking introductory physics as a co-requisite. For example, Davidson College [15] requires all physics majors to take a course in Computational Physics and incorporates computer-related work into most of its curriculum. Such coherence is easier at a small department than at a large department. At our institutions, such a course is not required, but the vast majority of physics majors take it. Once students have the necessary background, computational methods can be more effectively introduced into upper level courses. In fact, even if the faculty does not take the initiative, students will start using a variety of computer tools on their own if the tools are available to them. If the students have the tools, they will help reform the curriculum. In the next section we discuss ways of teaching such a course.

3 Laboratory-based computer simulation course

For the past fifteen years we have been teaching a laboratory-based computer simulation course to about 10–15 students per semester. The formal prerequisites for this course are one college level course in physics and one in calculus, but we are willing to waive the prerequisites if a student has had programming experience. No previous programming background is required for those with the prerequisites. First-year students through graduate students are enrolled in the course. In addition, the course attracts students from other disciplines including computer science, chemistry, mathematics, biology, and economics. We have written...
a textbook for the course that is now in its second edition [16]. The first edition was translated into Russian, and the second into Japanese [17]. Our Web site, http://sip.clarku.edu, provides additional information about the course, the text, and related material.

A distinctive aspect of our approach is that the physics drives the course rather than programming. There are many topics that can be discussed, and from year to year they vary depending on our interests and those of the students. Particular programming constructs and numerical algorithms are introduced as needed. Because most physics programs require the use of conditionals, loops, arrays, input and output routines, and functions, these topics are covered in depth. Other aspects of programming such as pointers, linked lists, bit manipulation, and recursion usually receive less attention. In addition, we are stressing visualization and animation more and more as a way of understanding the behavior of a system.

An important goal of our approach is to instill good programming habits. Thus, we encourage modular programs with reusable components. We also encourage students to avoid tricky programming constructs that might lead to slightly faster, but less readable code. We make the point that the programmer’s time is more valuable than the computer’s. Thus, we strongly discourage the “spaghetti” code that is a hallmark of old style Fortran programming. In Section 4 we discuss the issue of programming languages.

The course is project oriented, which provides much flexibility in the choice of topics. Students work at their own pace and do not all work on the same assignments at the same time. Those with stronger programming skills are encouraged to explore the physics deeper, while novices might spend most of their time working on coding. The mix of abilities and levels encourages students to teach each other. In addition to the instructor, we typically have a teaching assistant to help during the scheduled laboratory times.

We divide up the course into two themes. We first discuss those problems that can be solved using deterministic algorithms. For example, these might involve the numerical solution of differential equations to simulate the classical motion of particles. Applications include the Kepler problem, chaotic systems, and systems of many particles. We have students compare different algorithms and determine the level of accuracy one can expect from such algorithms. We also discuss chaotic maps because they are simple to program and are of much interest to students.

The second broad theme is the Monte Carlo approach. We usually discuss random walks, percolation, fractals, and depending on the student, algorithms useful for statistical mechanics. Many of these topics require very little formal preparation, and are also rarely discussed in other physics courses. Thus, they provide a natural means of showing students that there is more to physics than what is discussed in introductory physics. Also, many of these topics can include examples of systems traditionally considered by other scientists such as polymers, earthquakes, and population dynamics. Thus, the student can see how their skills can be used in a much broader context.

The project based structure of the course leads to a number of im-
important differences between our course and more traditional ones. For example, the instructors do not know all the answers and hence their role is different than in a traditional lecture course. Because some students know certain aspects about computer languages and programming better than us, these students have an opportunity to contribute to the course, and thus become more committed to it. Occasionally, students discover behavior that was not expected, and can tell the class about their results. This possibility exists because most of the assignments have some open-ended aspects in addition to direct questions. The direct questions help students see what the relevant issues are, and the open-ended suggestions allow students to explore a topic in more depth if they have time and the interest. The changed role of the instructor means that it is not necessary or even possible for the instructor to be expert in all the topics that are discussed.

A key component of our course is teaching students how to communicate scientific information. Students write laboratory reports, which are the basis for their grade in the course in addition to class participation and initiative. These reports include a summary of the system being modeled, a discussion of the method used to simulate the system, verification of the program, data presented in terms of tables and figures, an analysis and interpretation of the data, and a critique discussing the key physical concepts and techniques learned by doing the project. In addition, students are expected to keep a computer-based laboratory notebook that includes a log of the time spent on their work. The reports are an excellent vehicle for teaching writing skills.

A project begins with a discussion of a physical situation. Next we discuss what can be learned about the system using analytical methods. For about the first-half of the course, students are given a simple program that they can use as a starting point for their own program or as pseudocode if they are using a different language. They first run the program and collect data that they can use to test its validity and their understanding of what the program is doing. Next they are asked questions that require extending the program, collecting more data and doing more data analysis. Finally, the students report their results either formally in a written report or more informally during class discussions or by an interview with the instructor. Typically students work on four to six projects of this nature in a semester-long course and do a two to three week project of their choice at the end. The final project is written up as laboratory report and presented as a poster.

The above process follows the typical method of doing research in science. Students learn the importance of testing and how easy it is to make mistakes. They learn how difficult it is to distinguish between interesting behavior and various errors. They also learn the importance of having some analytical results and physical insight to guide their investigations. They also learn to share information. All of these lessons are important to the researcher and rarely incorporated in a meaningful way in traditional courses.

The format for the course consists of lecture/discussion periods plus a scheduled laboratory session for a total of six hours of scheduled time per week. The instructor and a teaching assistant are available during
the laboratory sessions to answer and ask questions. The discussions are usually general and emphasize general themes rather than particular problems. Specific programming questions are discussed in the laboratory. Undergraduate students are asked to work a total of 12–15 hours per week on the course, a total that is often more than some of their other courses. Students are discouraged from spending more than 15 hours per week on the course.

There are many obstacles that must be overcome to achieve an effective course. One example, is that students frequently have difficulty analyzing their data. They will fit their result to a straight line even if their plot shows curvature. They will accept the numerical output of a computer program even if it is off by many orders of magnitude from what is physically possible. They will record a number with eight digits, even though the number is only accurate to three digits. They will spend too much time making the graphical output of their programs and their reports look good, while spending not enough time on the physics. They will include five figures in their report and not discuss them in context.

Having undergraduate and graduate students in the same class at Clark University is a necessity because of the nature of the institution, but this mix has turned out to be good for both types of students. Although the course usually begins with traditional topics that are well known to the graduate students, they enjoy improving their programming skills in a traditional context. It also is not a good assumption that the graduate students will do better than the undergraduates. In fact, we have observed little correlation between the background of the students and their grades in other courses and their success in the computer simulation course. It is good to be reminded that there are many kinds of intelligence. On the other hand, the progress of graduate students in the computer simulation course is a much better predictor of how students will do in research than their grades in the traditional graduate courses. Some students do better in open-ended environments than others.

Grades in the course are determined by students’ progress since they started the course, rather than on their absolute knowledge. This procedure can cause some confusion and uncertainty among students. For example, two students might receive the same numerical score on their laboratory reports, but receive different final grades because of differences in their background. However, students generally appreciate the fact that their grade is usually proportional to their effort within the constraint that they can work no more than 15 hours per week (for undergraduates).

One of the most important lessons we have learned is that students are primarily interested in topics that they are able to understand well rather than topics that might seem interesting to us, but for which the student does not have sufficient background or judgment to appreciate. For example, we usually begin our courses with a simple example of one body motion in an external potential. In this way we can introduce various programming concepts and simple numerical integration algorithms in a well known context. One of the first problems that we ask students to do is to compute the two-dimensional trajectory of a particle that is thrown at some initial angle with respect to the ground. Of course, all the students know that the trajectory is a parabola. When we first gave this
assignment, we thought that the more advanced students, including the
physics graduate students, would be bored. But to our surprise, they have
just as much fun obtaining some positive feedback from their programming
efforts. Moreover, it is easy to ask the more advanced students more
difficult questions, such as the effects of including frictional forces and the
possible coupling of the motion in the vertical and horizontal directions.

There are occasionally some students who do not acquire many pro-
gramming skills even by the end of the course, and cannot write a program
from scratch that is significantly different from one in the text. These stu-
dents are able only to modify already existing programs. However, these
students could easily pass a quiz on the core syntax of a programming
language and can do simple exercises. Clearly, one course is not enough
for all students to become expert programmers. The same complaint is
usually justified in traditional courses as well, since most students cannot
solve problems significantly different from the ones that they have already
seen, even though it would seem like they have the appropriate knowledge
to do so. There is a mistaken notion that learning programming, physics,
mathematics and many other skills is an all or nothing kind of activity.
Instead these skills require years to become proficient.

There are a number of resources available to support the course dis-
cussed above. A listing of undergraduate and graduate level texts on com-
putational physics, including those that emphasize simulation, numerical
methods, and symbolic manipulation such as Maple, Mathematica, and
Matlab and can be found at our Web site [22]. In addition, the reader
can find links to other sites including links to Web sites of other courses
on computational physics.

The introduction of a laboratory-based project-oriented course into
the curriculum is a significant faculty investment of time. In addition,
one course can only begin to provide the skills needed to do computa-
tional physics. Either an additional course, or better, the integration of
computational physics into other courses, is needed to realize the greatest
benefit of a computer simulation course. We argue that because of its
open-ended nature, such a course is the most effective way of introducing
students to research and to the burgeoning new field of computational
physics.

4 Choice of programming language

One of the questions that we are frequently asked is, “What is the appro-
priate choice of programming language?” Our students have used True
Basic, Fortran 77/90, Pascal, C, C++, and Java. Our text [16] uses True
Basic [18], which can serve as a template for novice programmers or as
pseudocode for more advanced programmers or instructors who wish to
use a different language. No language is ideal for all purposes, and it
is important for students to appreciate the desirability of choosing the
language and computer platform that is most appropriate for the desired
goal. For our purposes there are a number of features of a programming
language that we believe are important. These features include:

- Platform independence, inexpensive, and easy to learn.
• Intrinsic graphics statements.
• Libraries for doing numerical calculations.
• Modular and preferably object-oriented.
• Event-based programming capability.
• Useful outside of physics so that the language will be maintained and improved and provide a marketable skill for students.
• Bit manipulation and parallel programming capability.

Given these considerations, we made the choice of using True Basic many years ago when most computer science departments were using Pascal in their beginning courses and most physicists were using Fortran 77. True Basic has a clean, easy to learn syntax, and makes it possible to do simple graphics in a platform independent way. We have learned how to teach it in the context of learning physics so that students have very little trouble with the syntax. Most students have no difficulty making the transition from True Basic to C or F (a subset of Fortran 90) [19]. However, True Basic lacks bit manipulation, does not allow the user to distinguish between real and integer variables, and is not used much outside of educational institutions. In addition, True Basic, like Fortran and C, is a procedural language, and it is now generally accepted that the object-oriented approach is the most desirable paradigm. In particular, an object-oriented approach is particularly appropriate for graphics and event based programming. Most importantly, using True Basic and Fortran 90 no longer excites most students who are more interested in Java and C.

For these reasons, our present language of choice is Java, especially beginning with version 1.1. This language has all of the desired features that we listed in the above. The main disadvantage is that at present, a Java program runs about a factor of ten more slowly in comparison to compiled languages such as C and Fortran. However, this disadvantage is expected to be reduced as efficient Java compilers are developed. This difference is usually not a significant limitation in the undergraduate classroom, especially with the availability of inexpensive Pentium III and G3/G4 computers. If a student project requires more speed to obtain reasonable results, the project would most likely be done by an advanced student who would have little trouble converting the program to C++ or Fortran 90.

The two currently popular object-oriented languages are C++ and Java. Because C++ is faster, why not choose it? Our main reasons are its lack of intrinsic graphics statements and its hybrid nature that allows the user to write a procedural program. In contrast, Java forces the user to write an object-oriented program, but of course, does not force the user to program well.

If one chooses an object-oriented language, then the challenge becomes to convert our thinking from a procedural approach to an object-oriented approach. There is little difficulty using an object-oriented approach to handle programming tasks such as a graphical user interface that allows the user to interact with the program while it is running. Nor is it difficult to write a program that uses graphics and animation to visualize results.
These tasks are very natural in Java and are relatively easy to learn. The difficulty comes in deciding whether it is worthwhile to take the numerical algorithms we have for doing physics and rethink them in terms of objects.

In traditional scientific programming there are a variety of numerical tasks that are performed by libraries of numerical algorithms and procedurally oriented pieces of code. The procedures or functions and the data that the functions act on are treated separately. For example, we might decide to code a matrix as a two-dimensional array. Then we might pass this array to a function that finds the eigenvalues and eigenvectors. In an object-oriented approach the data and the functions (called methods) are in one object. All the operations that one would do on the data are part of the object. To avoid duplication when creating objects, we use inheritance so that methods of one object are inherited directly from another object, and thus are not rewritten. An advantage of this approach is that we can use the same method names for different kinds of objects. Thus, we can build a multiplication method into an object and use the same method name for different types of data. For example, we could have data organized as vectors, complex numbers, and matrices. For each of these we would write a method, called multiply, and place it in the appropriate object.

Object-oriented programming can help emphasize the unity of physics. For example, we can create an object called \texttt{Particle} that could be used for a ball falling through air, a planet circulating the sun, or a molecule in a molecular dynamics program. One object can contain the properties of a particle such as position, velocity, mass, and charge as well as the methods acting on the object such as a method to update its position and velocity. Another example is given by Coppersmith \cite{20}, who discusses how the object-oriented approach can be used to emphasize universality in chaotic systems. As a start in teaching Java in the context of a simulation course, we have included in the Appendix a tutorial similar to what we gave our students in our computer simulation course in the Fall of 1999 \cite{21}. We expect to continue to develop this tutorial and if there is sufficient interest, write a Java edition of our text.

5 Curriculum Development

The introduction of a course on computer simulation will not have a major impact on the overall physics curriculum unless other courses are developed that require students to use their computational skills. A number of recent texts covering traditional physics topics incorporate well defined computer exercises. These texts will help make numerical work a part of the physics student’s toolbox, but they will not help change the curriculum so that students are better able to tackle modern research and engineering problems. As we discussed earlier, we suggest that the upper level courses should move away from asking students to solve many idealized problems and move toward asking students to work on bigger projects that encourage students to bring together skills and knowledge from different areas.

One area in which simulation plays a natural role is statistical and
6 Conclusion

During the next decade, physics faces the challenge of remaining the central discipline of the natural sciences, at least in the eyes of students. There is even a danger that physics will suffer the same fate as Latin, which is the basis of the Romance languages. One problem is that physics no longer has the glamour once associated with relativity and quantum theory and many of the better science students are choosing computer science and molecular biology where the excitement appears to be.

At the same time we know that the skills associated with a background in physics are even more important and that physicists are playing leading roles at the edge of our discipline, whether it be computational biology, materials science, or finance to name just a few areas. However, it is not enough just to tell our students that physicists can do anything. We need to organize our courses and our departments so that these connections are obvious and it is clear that physics remains the central discipline of the sciences. Computational physics with its clear links to computation in the other sciences as well as in other areas can play a leading role in keeping the discipline of physics healthy and active.

Acknowledgements

We acknowledge the National Science Foundation, grant # PHY-9801878, for their support of our work. We also thank Shubha Tewari, Wolfgang Christian and Macneil Shonle for useful comments and Dietrich Stauffer for inviting us to contribute to this volume.

A Java tutorial

In the following, we give a brief introduction to the nature of Java. Our main purpose is to help scientists who have not used Java to get over the barrier presented by most of the books on Java. Some of our favorite books on Java are listed in the references [25, 26, 27, 28, 29].
Java is an object-oriented programming language with a built-in application programming interface (API) that can handle graphics and user interfaces. Because of its rich set of API's, similar to Macintosh and Windows, and its platform independence, Java can also be thought of as a platform in itself. Java also has standard libraries for doing mathematics.

Much of the syntax of Java is the same as C and C++. One major difference is that Java does not have pointers. However, the biggest difference is that the user must write object oriented code. Procedural pieces of code can only be embedded in objects. In the following, we assume that the reader has some familiarity with a programming language. In particular, some familiarity with the syntax of C/C++ is useful.

In Java we distinguish between applications, which are programs that perform the same functions as those in other programming languages, and applets, which are programs that can be embedded in a Web page and accessed over the Internet. Our initial focus is on writing applications. When a program is compiled, a byte code is produced that can be read and executed by any platform that can run Java.

The structure of a Java program consists of various objects exchanging messages. An object consists of data and functions known as methods which use or change the data. (Methods are similar to procedures or functions in other languages.) Objects of the same kind are said to have the same type or be in the same class. A class defines what data can be in an object, and what operations are performed by the methods. One or more objects can be created or “instantiated” from a class. In our first example we define the class Particle.

```java
public class Particle
{
    double x, y, vx, vy, mass;
}
```

This class does not do anything (because it has no methods), but it can be used to describe a particle by its position, velocity, and mass. For simplicity, we will consider only two dimensions.

**Constructors.** Every class has at least one constructor, a method that has the same name as the class. A constructor initializes a new object belonging to the class.

```java
public class Particle
{
    // these variables can be used by any method in the class
    double x, y, vx, vy, mass;

    // example of constructor method
    public Particle(double x, double y, double vx, double vy, double mass)
    {
        // Use this to make explicit that method accesses its own variables
        // set instance variable x equal to value of x in parameter list
        this.x = x;
        this.y = y;
        this.vx = vx;
        this.vy = vy;
        this.mass = mass;
    }
}
```
The constructor `Particle` creates an object of the `Particle` class by specifying five parameters: the initial position and velocity of a particle and the value of its mass. We say that `Particle` is the constructor for the `Particle` class.

We have used the following properties of Java:

- The types of all variables must be declared. The primitive types are `byte`, `short`, `int`, `long` (8, 16, 32, and 64 bit integer variables, respectively), `float` and `double` (32 and 64-bit floating point variables), `boolean` (true or false), and `char`. Boolean is a distinct type rather than just another way of using integers. Strings are not a primitive type, but are instances of the `String` class. Because they are so common, string literals may appear in quotes as in other languages.

- Java distinguishes between upper and lower case variables. The convention is to capitalize the first letter of a class name. If the class name consists of several words, they are run together with successive words capitalized within the name (instead of using underscores to separate the names). The name of the constructor is the same as the name of the class. All keywords (words that are part of the language and cannot be redefined) are written in lower case.

- Instance variables and methods can be accessed from any method within the class. The `x` in the argument list of the above constructor refers to the local value of the parameter which is set when `Particle` is called. We use the `this` keyword to refer to those variables defined for the entire class in contrast to those defined locally within a method and those that are arguments to a method. In the above example, `this.x` refers to the variable `x` which is defined just after the first line of the class definition. After we have introduced multiple constructors, we show a more illustrative example of the `this` keyword.

- Classes are effectively new programmer defined types; each class defines data (fields) and methods to manipulate the data. Fields in the class are templates for the instance variables that are created when objects are instantiated (created) from that class. A new set of instance variables is created each time that an object is instantiated from the class.

- The members of a class (variables and methods) are accessed by referring to an object created from the class using the `dot operator`. For example, suppose that a class `Particle` contains an instance variable `x` and a method `step()`. If an object of
this class is named p, then the instance variable in p would be
accessed as p.x and the method accessed as p.step().

• A semicolon is used to terminate individual statements.
• There are three comment styles in Java. A single line comment
starts with //</> and can be included anywhere in the program.
Multiple line comments begin with /** and end with */. Text
enclosed within /** @*/ serves to generate documentation
using the javadoc command.

Multiple Constructors. The arguments of a constructor specify the
parameters for the initialization of an object. Multiple constructors
provide the flexibility of initializing objects in the same class with
different sets of arguments, as in the following example.

```java
public class Particle
{
    double x, y, vx, vy, mass;

    // examples of multiple constructors

    public Particle()
    {
        this(0.0,0.0);
    }

    public Particle(double x, double y)
    {
        this(x,y,1.0);
    }

    public Particle(double x, double y, double m)
    {
        this(x,y,0.0,0.0,m);
    }

    public Particle(double x, double y, double vx, double vy)
    {
        this(x,y,vx,vy,1.0);
    }

    public Particle(double x, double y, double vx, double vy, double mass)
    {
        this.x = x;
        this.y = y;
        this.vx = vx;
        this.vy = vy;
        this.mass = mass;
    }
}
```

• The multiple constructors (all named Particle) are distinguished
only by the number of arguments. The constructors can be
defined in any order. Hence, the keyword this in the first con-
structor refers to the next in the sequence because the latter
has two arguments. The first constructor has no arguments and creates a particle of unit mass at the origin; the next is defined with two arguments: the spatial coordinates of the particle. The second constructor in turn references the third constructor which uses the spatial coordinates and the mass. The third and fourth constructors each refer to the final constructor which uses all five arguments. Once the Particle class with its multiple constructors is defined, any class can call the constructor Particle using the number of arguments appropriate to that application. The advantage of having multiple constructors is that applications that use a particular constructor are unaffected by later additions made to the class Particle, whether variables or methods. For example, adding acceleration as an argument does not affect applications that rely only on the definitions given above.

- Using multiple constructors is called method overloading — the method name is used to specify more than one method. The rule for overloading is that the argument lists for all of the different methods must be unique, including the number of arguments and/or the types of the arguments.

- All classes have at least one implicit constructor method. If no constructor is defined explicitly, the compiler creates one with no arguments.

We next give an example of a Particle class with methods for computing the weight of a particle and its distance from the origin. We omit the velocity components for now.

```java
public class Particle
{
    double x, y, mass;

    public Particle()
    {
    }

    public Particle(double x, double y)
    {
        this(x,y,1);
    }

    public Particle(double x, double y, double mass)
    {
        this.x = x; this.y = y; this.mass = mass;
    }

    public double getWeight()
    {
        return 9.8*mass;
    }
}
```
public double distanceFromOrigin()
{
    return Math.sqrt(x*x + y*y);
}

- The fundamental parts of a method are its name, arguments, return type, and the body. Methods can be created only as part of a class. By convention, variable and method names begin with a lower case letter.

- The keyword `return` followed by an expression returns the value of the expression to the caller of the method. The type of the expression must be the same as specified by the method. (Note that a constructor does not return a value and hence no type can be specified.) A method can return only the value of a single expression.

- The method `distanceFromOrigin` in the class `Particle` computes the distance of a particle from the origin using the particle coordinates `x` and `y`. The square root function `Math.sqrt` in this calculation is part of the `Math` class, a library of mathematical functions which consists of a collection of static methods. Their use is clear from the context.

A program consists of one or more class definitions, each of which should be in separate files. The name of the file should be the same as the name of the class, for example, `MyApplication.java`. One of these classes must define a method `main`.

The following program creates two objects of type `Particle`.

```java
public class MyApplication
{
    public static void main(String[] args)
    {
        /* Need an instance of class Particle - a particle object;
        assign it to the variable a which is of type Particle. */
        Particle a = new Particle(1.0,1.0);
        System.out.println("Distance of a from origin = " + a.distanceFromOrigin());
        System.out.println("Mass of particle a = " + a.mass);
        Particle b;
        b = new Particle();
        b.x = 2.0;
        b.y = 3.0;
        b.mass = 3;
        System.out.println("Distance of b from origin = " + b.distanceFromOrigin());
        System.out.println("Mass of particle b = " + b.mass);
        System.out.println("Weight of particle b = " + b.getWeight());
    }
}
```

- The first statement in `main` declares `a` to be a `Particle` and uses the `new` keyword to instantiate an object. These two operations could have been done separately in two lines as illustrated.
for Particle b. Note that Java already defines primitive data types such as int and double.

- If a program has more than one class, one and only one of the classes must define main().
- The statement System.out.println() allows the user to print the argument to the standard output with a call to the System class. System.out is an object used for sending output to the screen and println is a method that this object invokes.

Public and Private Variables. Java uses three explicit keywords and one implied keyword to set the boundaries in a class: public, private, and protected. The default access specifier for the names of variables and methods is “package visibility” or “friendly,” which means that all the other classes in the current package have access to them. (Packages are Java’s way of grouping classes to make libraries.) The access specifier public means that the variables and methods are available from any package; private implies that the variables and methods can only be accessed inside methods of the same class. The keyword protected indicates additional access to variables and methods in subclasses. We will clarify its meaning when we discuss inheritance.

One reason to make a variable private is to restrict access to it. Access becomes an issue for threading which refers to the sequence of execution of the program code. For example, we would want to avoid changing the value of a variable while another portion of the code is trying to read it. Make x private and see what happens when you run MyApplication.

If we declare x, y, and mass as private variables, we have to include explicit methods in Particle to allow another class to access the variable information in Particle. For simplicity, we will consider only the variables x and mass. Our particle class becomes:

```java
public class Particle {
    private double x;
    private double mass;

    public Particle(double x) {
        this(x, 1.0);
    }

    public Particle(double x, double mass) {
        this.x = x;
        this.mass = mass;
    }

    public double getX()
```


```java
{ 
    return x;
}

public void setX(double newX) 
{ 
    x = newX;
}

public double getWeight() 
{ 
    return 9.8*mass;
}

public double distanceFromOrigin() 
{ 
    return Math.abs(x);
}

}

Note the new methods getX and setX. They are used in the following.

public class MyApplication
{
    public static void main(String[] args)
    {
        Particle p = new Particle(10.0, 2.0);
        System.out.println("Distance of p from origin = " + p.distanceFromOrigin()); // would have written p.x if x were public
        System.out.println("x-coordinate = " + p.getX());
        System.out.println("weight = " + p.getWeight());
        p.setX(3.0); // change value of x
        System.out.println("new x-coordinate = " + p.getX());
        System.out.println("new distance from origin = " + p.distanceFromOrigin());
    }
}

In the following, we will not make variables such as x and y private, because they will be used in so many classes.

Using the Particle class. We now redefine the class Particle to include a method to compute the distance between two particles. The application that follows creates two particles and gives their mutual separation.

public class Particle
{
    double x, y, mass;

    public Particle(double x, double y)
    {
        this(x, y, 1.0);
    }

    public Particle(double x, double y, double mass)
    {
        this.x = x;
        this.y = y;
    }
```
this.mass = mass;
}

public double distanceFrom(Particle a)
{
    double r2 = Math.pow(this.x - a.x, 2) + Math.pow(this.y - a.y, 2);
    return Math.sqrt(r2);
}

public double distanceFromOrigin()
{
    return Math.sqrt(x*x + y*y);
}
}

public class MyApplication
{
    public static void main(String[] args)
    {
        Particle a = new Particle(5.0, 7.0);
        Particle b = new Particle(1.0, 2.0);
        System.out.println("Distance of a from b = " + a.distanceFrom(b));
        System.out.println("Distance of b from a = " + b.distanceFrom(a));
    }
}

Extending a class. Object oriented programming allows the user to reuse existing code rather than rewrite it. Classes have a hierarchical relationship, allowing the user to extend or modify the behavior of classes derived from base classes using inheritance. For example, we can expand the Particle class to include charged particles. The new class ChargedParticle will combine all the behavior and functionality of Particle with the ability to have a charge. We can implement ChargedParticle as an extension of Particle as shown below.

public class ChargedParticle extends Particle
{
    // magnitude of electron charge in Coulombs
    public static final double ELQ = 1.602e-19;
    private int charge;

    public ChargedParticle(double x, double y, double mass, int charge)
    {
        super(x, y, mass);          // constructor for Particle
        this.charge = charge;
    }

    public int getCharge()
    {
        return charge*ELQ;
    }

    public void setCharge(int newCharge)
    {
charge = newCharge;
}

public static int netCharge(ChargedParticle a, ChargedParticle b)
{
    return a.charge + b.charge;
}

An example of the use of this new class is given below.

public class MyApplication
{
    public static void main(String[] args)
    {

        // particle charge is expressed in units of electron charge
        ChargedParticle a = new ChargedParticle(10.0,0,0,1);
        System.out.println("distance of a from origin = " + a.distanceFromOrigin());
        System.out.println("charge of a = " + a.getCharge());
        System.out.println("charge of a in coulombs = " + ChargedParticle.ELQ*a.getCharge());
        ChargedParticle b = new ChargedParticle(-5.0,0,0,-1);
        System.out.println("distance of b from origin: " + b.distanceFromOrigin());
        System.out.println("charge of b = " + b.getCharge());
        System.out.println("net charge of a and b = " + ChargedParticle.netCharge(a,b));
        b.setCharge(3);
        System.out.println("new charge of b = " + b.getCharge());
        System.out.println("net charge of a and b = " + ChargedParticle.netCharge(a,b));
    }
}

Note how the (named) constant ELQ, the charge of an electron, has been introduced.

- The keyword super refers to the constructor of the base class, in this case Particle. An inherited subclass contains all the members of the superclass except the private ones. Thus a derived class can receive the same messages as a base class.
- A variable declared final is a constant. Its value must be specified when it is declared and this value cannot be changed. The static keyword means that there is only one copy of this variable, regardless of the number of instances of the class that are created. (A better name is class variable.) The C convention of using upper case letters for constants will be used whenever it will not cause confusion.
- The method setCharge defines the behavior of a particular object. In contrast, the static method netCharge belongs to the class and not to a particular instantiation of the class. To call a static method, we use the class name rather than an instance name. Thus, to call the method netCharge, we use ChargedParticle.netCharge(a,b) not a.netCharge(a,b). A static method is equivalent to a global function in C.
Static methods (class methods) and static variables, belong to a class and not a given instance of the class. The following example illustrates the nature of static variables:

```java
public class StaticTest
{
    public static int x = 0;
    public int y = 0;

    public String getCoordinates()
    {
        return "x = " + x + ", y = " + y;
    }

    public static void main(String[] args)
    {
        StaticTest a = new StaticTest();
        StaticTest b = new StaticTest();
        a.x = 5; // static variable
        a.y = 12; // instance variable
        System.out.println(a.getCoordinates()); // outputs 5, 12
        b.x = 7;
        b.y = 13;
        System.out.println(a.getCoordinates()); // outputs 7, 12
        System.out.println(b.getCoordinates()); // outputs 7, 13
        StaticTest.x = 2;
        System.out.println(a.getCoordinates()); // outputs 2, 12
        System.out.println(b.getCoordinates()); // outputs 2, 13
    }
}
```

The (static) x member belongs to class StaticTest and hence belongs to a and b. There is only one variable x and thus the value of x is always the one most recently specified. In contrast, y belongs to the instances a and b of the class StaticTest and hence a.y and b.y are distinct variables.

The String type is unique among all classes because it is the only class to support an operator: the + operator can be used to concatenate strings. When a number is concatenated with a string, the number is converted to a string. Because a String variable is an object, it has methods.

Arrays. An array is a special object containing a group of contiguous memory locations that have the same name and the same type and a separate variable containing an integer constant equal to the number of array elements. The elements of Java arrays are numbered starting from 0.

An array must be created before it can be used. We first declare a reference or “handle” to an array that permits Java to locate the object in memory when it is needed. Then we create an array object to assign to the reference using the new operator. For example, we can write
double x[];  // create an array reference
x = new double[5];  // create array object

Or we can create an array reference and an array object on a single line:
double x[] = new double[5];

The number of elements in the array x is \( x.length \). The elements of the array are written as \( x[0] \), \( x[1] \), \( x[2] \), \( x[3] \), \( x[4] \).

An array object may be created and initialized when its reference is declared. For example,

double x[] = {1.0, 1.4, 1.6, 1.8, 3.0};

It is a good idea to declare array sizes using named constants (final variables) so that the length of the arrays can be easily changed.

final int ARRAY_SIZE = 1000;
double x[] = new double[ARRAY_SIZE];

A two-dimensional array is implemented by creating a one-dimensional array each of whose elements is also an array. We first declare a reference to an array of arrays and then create the individual arrays associated with each element. For example,

double x[][];  // create reference to an array of arrays

Then we can write,

\( x = \text{new double}[3][5]; \) create array objects

\textit{Methods and Scope.} The general form of a method definition is

\[
\text{return-value-type method-name(\text{parameter list})} \\
\{ \\
\quad \text{declarations and statements} \\
\quad \text{(return statement)} \\
\}
\]

- The \textit{return-value-type} is the data type of the result returned by the method to the caller. If no value is returned, the return-value-type is \textit{void}. If the return-value-type is not void, then there must be a \textit{return} statement.
- The \textit{parameter-list} is a comma-separated list containing the parameters received by the method whenever it is called. A type must be declared for every parameter in the list. If the parameter-list is empty, the parentheses is still required.
- Variables that are used within a method and that are not accessible by calling methods are \textit{local variables}.
- Java programs communicate with their methods using \textit{pass-by-value}. The method works with a copy of the original argument, not the argument itself which means that the method cannot modify the original argument even if it modifies the parameter. A program that illustrates pass-by-value is shown below.
However, if a parameter is an object, the value of the object is its location in memory. In this case the values of any instance variables (or array elements for arrays) can be modified within the method.

```java
public class TestPassByValue {
    public static void main(String[] args) {
        // instantiate a TestPassByValue object
        TestPassByValue t = new TestPassByValue();
        int i = 1; // local variable
        System.out.println("value of i before test = " + i);
        // pass value of local variable
        int j = t.test(i);
        System.out.println("value of i after test = " + i);
        System.out.println("value of j = " + j);
        j = t.test2(j);
        System.out.println("value of i after test2 = " + i);
        System.out.println("value of j after test2 = " + j);
    }

    public int test (int i) {
        i = ++i; // same as i = i + 1;
        System.out.println("value of i in test = " + i);
        return i;
    }

    public int test2 (int k) {
        int i = k; // i refers to instance variable
        System.out.println("value of k = " + k);
        i = ++i;
        System.out.println("value of i in test2 = " + i);
        return i;
    }
}
```

Variables have either class scope or block scope. Methods and the instance variables of a class have class scope. Class scope begins at the opening left brace and ends at the closing right brace of the class definition. Class scope allows any method in the class to directly invoke any other method in the class and to directly access any instance variable of the class. In effect, instance variables are global variables within a class. Instance variables can be used to communicate between methods in a class or to retain information between calls of a given method in the class.

A block is a compound statement and consists of all the statements between an opening and closing brace. Variables defined within a block have block scope are visible within the block; they are not visible outside the block.

A variable can have the same name as an instance variable or method in the class in which the method is defined. In this case the
instance variable is hidden from the method by the local variable. Example:

```java
public class Point {
    // define instance data
    public double x, y; // instance variables

    // define constructors
    public Point() {
        x = 1; // same variables as instance variables
        y = 1;
    }

    public Point(double x, double y) {
        this.x = x; // this.x refers to instance variable
        this.y = y;
        this.x++;
        System.out.println("this.x = " + this.x);
        System.out.println("x = " + x);  // parameter variable
    }

    public Point(double x, double y, boolean dummy) {
        x++;
        System.out.println("x = " + x);
    }
}
```

```java
public class TestPoint {
    public static void main(String[] args) {
        /* Need an instance of class Particle - a particle object; assign it to the variable a which is of type Particle. */

        Point a = new Point(2.0, 1.0);
        System.out.println("x = " + a.x);
        Point b = new Point();
        b.x = 3.0;
        System.out.println("xb = " + b.x);
    }
}
```

**Simple Graphics.** A powerful feature of Java is its ability to do graphics relatively simply. We next introduce several methods in the `Particle` class to draw a representation of the particle.

```java
import java.awt.*;
public class Particle {
    double x, y;
    Color color; // part of AWT package
```
public Particle(double x, double y)
{
    // constructor with color variable
    this.x = x;
    this.y = y;
    color = Color.blue; // define default color
}

// Draw representation of particle in given graphics context
public void draw(Graphics g)
{
    Color oldColor = g.getColor();
    g.setColor(color);
    g.fillOval((int)x, (int)y, 12, 12);
    g.setColor(oldColor);
}

public Color getColor()
{
    return color;
}

public void setColor(Color newColor)
{
    color = newColor;
}

public void move(double dx, double dy)
{
    x += dx; // same as x = x + dx;
    y += dy;
}

• The import statement allows us to use the Java libraries. The notation java.awt.* means that all the classes in the java.awt package can be used. This package implements the Java Abstract Window Toolkit (AWT) and contains all the classes and interfaces necessary for creating a user interface. Packages are a way of grouping a collection of related classes [24].

• We have used the java.awt.Graphics class which provides drawing and painting methods. The method draw has an argument g of type Graphics. Think of g as a part of the screen (a bit map) where painting will take place and the methods that are associated with it.

• The four principal elements need to create a GUI include:

  – A component is a visual object containing text or graphics that can respond to keyboard or mouse inputs. Examples of components include buttons, labels, text boxes, check boxes, and lists. A blank component is known as a canvas, which can be used as a drawing area for text or graphics.
All components inherit a common set of methods, the most common of which is \texttt{paint}.

- A \texttt{container} is a graphical object that can hold components or other containers. The most important type of container is a \texttt{Frame}.

- A \texttt{layout manager} is automatically associated with each container when it is created, but the layout manager can be changed. Examples include BorderLayout, BoxLayout, GridLayout.

- Events, such as the click of a mouse, are handled by creating \texttt{listener classes} which implement \texttt{listener interfaces}. The standard listener interfaces are in \texttt{java.awt.event}.

The following simple class draws the particle:

```java
import java.awt.*;
public class ParticleDrawer extends Frame
{
    public static void main(String[] args)
    {
        // Create new frame
        ParticleDrawer frame = new ParticleDrawer();
    }

    // Construct ParticleDrawer frame
    public ParticleDrawer()
    {
        // predefined methods
        setSize(512, 342); // units of coordinate system in pixels
        setVisible(true);
    }

    // Paint particle
    public void paint(Graphics g)
    {
        Particle p = new Particle(3.1, 4.1);
        p.draw(g);
        p.move(26.5, 35.8);
        p.setColor(Color.red);
        p.draw(g);
    }
}
```

- A \texttt{frame} is a very simple window, with a border, a place for a title, and a close window button.

- Note how the \texttt{paint} method is used. It is not called directly. Instead, whenever the frame is shown, the paint method is called. In addition, if one covers the frame with another window, the paint method will be called again when the frame is uncovered. To directly call the paint method, use the \texttt{repaint()} method.

As an example, we draw the trajectory of a projectile. To do so we add to \texttt{Particle} the method \texttt{step} to integrate the equations of
motion for one time step. The arguments of \texttt{step} are the time step and the \texttt{Force} on the particle, which is defined in another class. We also add other useful methods.

```java
public class Particle {
    private double x, y, vx, vy, ax, ay;
    private double mass = 1.0;
    private boolean firststep = true;

    public Particle(double x, double y, double vx, double vy) {
        this.x = x;
        this.y = y;
        this.vx = vx;
        this.vy = vy;
    }

    public double getMass() {
        return mass;
    }

    public void setMass(double mass) {
        this.mass = mass;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getVx() {
        return vx;
    }

    public double getVy() {
        return vy;
    }

    public double getAx() {
        return ax;
    }

    public double getAy() {
        return ay;
    }
}
```
public void setX(double x) {
    this.x = x;
}

public void setY(double y) {
    this.y = y;
}

public void setVx(double vx) {
    this.vx = vx;
}

public void setVy(double vy) {
    this.vy = vy;
}

public void step(double dt, Force f) {
    if (firststep) {
        ax = f.getfx(x, y, vx, vy, this)/mass; // acceleration at beginning of interval
        ay = f.getfy(x, y, vx, vy, this)/mass;
        firststep = false;
    }
    // Euler-Richardson algorithm
    double vxm = vx + 0.5*ax*dt; // velocity at middle of interval
    double vym = vy + 0.5*ay*dt;
    double xm = x + 0.5.vx*dt; // position at middle of interval
    double ym = y + 0.5.vy*dt;
    double axm = f.getfx(xm, ym, vxm, vym, this)/mass;
    double aym = f.getfy(xm, ym, vxm, vym, this)/mass;
    vx += axm*dt; // velocity at end of interval
    vy += aym*dt;
    x += vxm*dt; // position at end of interval
    y += vym*dt;
    ax = f.getfx(x, y, vx, vy, this)/mass; // acceleration at end of interval
    ay = f.getfy(x, y, vx, vy, this)/mass;
}

public class Force {
    private final static double g = 9.8; // not convention
    double b = 0; // used in drag force

    public void setb(double b) {
        this.b = b;
    }

    public double getfx(double x, double y, double vx, double vy, Particle p) {
        return -b*vx;
    }
public double getfx(double x, double y, double vx, double vy, Particle p) {
    return -b*vx - g*p.getMass();
}

Note that if statements are the same as in C/C++ except that Java uses a boolean value to condition the execution.

The step method implements the Euler-Richardson integration algorithm. We need to compute the acceleration at the beginning of the interval the first time the algorithm is used. Then we can use the acceleration computed at the end of the previous interval. Note how this is used in the step method to refer to the Particle itself; in Force we use the argument Particle to get the particle's mass.

The class that draws the trajectory is given below.

```java
import java.awt.*;
public class Simulation extends Frame {
    public static void main(String[] args) {
        Simulation sim = new Simulation(); // set up window for application
    }

    public Simulation() {
        setSize(512,342);
        setVisible(true);
    }

    public void paint(Graphics g) {
        setBackground(Color.white);
        calculateTrajectory(g);
    }

    private void calculateTrajectory(Graphics g) {
        final double tmax = 10.0;
        final double dt = 0.5;
        Particle p = new Particle(0.0, 200.0, 40.0, 25.0);
        Force f = new Force();
        g.setColor(Color.blue);
        double time = 0.0;
        while (time < tmax) {
            // draw circle of diameter 10 pixels and note use of casting
            g.drawOval((int)p.getX(), getSize().height - (int)p.getY(), 10, 10);
            p.step(dt,f);
            time += dt;
        }
    }
}
```
• The Simulation class extends the Frame class. The paint method is called when the Frame is first shown and anytime Frame is brought to the front.

• Java uses a coordinate system whose origin is at the upper left-hand corner with positive x values to the right and positive y values down. Because of this choice of screen coordinate system, we need to convert world coordinates to screen coordinates when drawing an oval to represent the particle. Because the last two arguments of drawOval are the same, the oval is a circle. (Note that drawOval requires integers.)

• Note the use of the while statement which allows all the statements within the braces of the while statement to be executed again and again as long as the argument of the while statement (time < tmax) is true.

• Type casting changes the type of a value from its normal type to some other type. For example:

```java
double distance;
distance = 10.0;
int x;
x = (int)distance;  // example of a type cast
```

In the above example x = 9 if distance = 9.9.

• An alternative to the while loop is the for loop which looks like

```java
for (int i = 0; i < 10; i++)
    statement;
```
or

```java
for (int i = 0; i < 10; i++)
{
    statement 1;
    ...
    statement n;
}
```

Semicolons are used to separate the fields inside the parenthesis. The first field initializes a variable. Usually, the variable is also declared here. The second field determines under what condition the loop will continue. The third field lists how the variable in the first field changes at the end of each pass through the loop.

Offscreen buffers. When the window of another application covers the frame, Java automatically calls paint when the user returns to the original program. This action allows us to see the contents of the original window. However, in the above program it means that the trajectory must be recomputed. A better approach is to first draw
the trajectory to an offscreen image buffer and then blast the latter to the screen. As long as the buffer is saved, no new calculations need to be done.

First we will show how this is done for the `Simulation` class, and then we will show an example where it is more useful.

```java
import java.awt.*;
public class Simulation extends Frame
{
  Image offscreen;

  public static void main(String[] args)
  {
    Simulation sim = new Simulation(); // set up window for application
  }

  public Simulation()
  {
    setSize(512, 342);
    setVisible(true);
    offscreen = createImage(getSize().width, getSize().height);
    calculateTrajectory();
  }

  public void paint(Graphics g)
  {
    setBackground(Color.white);
    g.drawImage(offscreen, 0, 0, this); // draw image onto screen
  }

  public void calculateTrajectory()
  {
    final double tmax = 10.0;
    final double dt = 0.5;
    Graphics g = offscreen.getGraphics();
    g.setColor(Color.blue);
    Particle p = new Particle(0.0, 200.0, 40.0, 25.0);
    Force f = new Force();
    double time = 0.0;
    while (time < tmax)
      {  
        g.drawOval((int)p.getX(), getSize().height - (int)p.getY(), 10, 10);
        p.step(dt, f);
        time += dt;
      }
  }
}
```

- The `Graphics` class is used here to provide drawing methods for `offscreen`, which is an instantiation of the `Image` class. Both `Graphics` and `Image` cannot be instantiated directly with a constructor; instead we use the `getGraphics` and `createImage` methods, respectively. The `Image` class represents a displayable image.
- Note how we declare `offscreen` outside of any method, and
instantiate it with the `createImage` method in the constructor. This technique is common for making objects accessible to all the methods of a class, and to insure that they continue to exist after a method returns.

**User Interaction.** Offscreen buffers are useful when we want to be able to draw more than one trajectory without running the program over again. To do so we want the user to interact with the program. This feature is very easy to accomplish using components such as Buttons, TextFields, and Menus. Below we show how to use a TextField to receive a number from the user. We put the TextField on a separate frame and use an `actionListener` to detect whether the user has entered a number in the TextField. The number entered will be the friction coefficient, $b$, used in a drag force of the form $- bv$.

```java
import java.awt.*;
import java.awt.event.*; // needed for actionListener

class FrictionInput extends Frame
{
    TextField tf;
    Label lb;
    Simulation sim;

    public FrictionInput (Simulation sim)
    {
        this.sim = sim;
        setUpFrame();
    }

    public void setUpFrame()
    {
        setTitle("Friction Input");
        setSize(200,100);
        setLocation(400,50); // not using any layout manager
        lb = new Label("Friction Coefficient"); // new label for textfield
        lb.setSize(150,20);
        lb.setLocation(30,70);
        add(lb); // add label
        tf = new TextField(); // new textfield
        tf.addActionListener(new actionTF()); // add listener
        tf.setSize(50,20);
        tf.setLocation(30,40);
        add(tf); // add textfield
        setVisible(true);
    }

    class actionTF implements ActionListener
    {
        // internal class

        public void actionPerformed(ActionEvent e)
        {
            // your code here
        }
    }
}
```
Double R = new Double(tf.getText().trim());
    sim.calculateTrajectory(R.doubleValue());
}

- There are two objects attached to the Frame. Label lb represents some text that will be placed under the TextField tf. The size and location of each of these components is specified. There are layout managers that automatically decide these specifications, but they frequently do not look good.

- One of the features of Java is the use of the actionListener which is implemented by using the addActionListener method and an internal class which implements the ActionListener interface. An interface is like a class except none of the methods have implementations. Instead the user must supply these as we have done above for the method actionPerformed(ActionEvent e). The beauty of the above event model is that the user does not have to write code to check for the occurrence of an event in other parts of the program. Whenever a number is entered into the TextField, the actionPerformed method will be executed. Here we extract the number by converting the string in the TextField to a number using the Double class, and then call the calculateTrajectory method of the object sim.

The following modification of Simulation uses FrictionInput.

```java
import java.awt.*;

public class Simulation extends Frame {
    private Image offscreen;
    private int counter = 0; // used to change color for each trajectory

    public static void main(String[] args) {
        Simulation sim = new Simulation(); // set up window for application
        FrictionInput fi = new FrictionInput(sim); // set up second window
    }

    public Simulation() // constructor
    {
        setSize(512, 342);
        setVisible(true);
        offscreen = createImage(getSize().width, getSize().height);
    }

    public void paint(Graphics g) {
        setBackground(Color.white);
        g.drawImage(offscreen, 0, 0, this); // draw image on screen
    }
}
```
public void calculateTrajectory(double b)
{
    final double tmax = 10.0;
    final double dt = 0.5;
    Graphics g = offscreen.getGraphics(); // create buffer
    changeColor(g);
    Particle p = new Particle(0.0, 200.0, 40.0, 25.0);
    Force f = new Force();
    f.setb(b);
    double time = 0.0;
    while (time < tmax)
    {
        g.drawOval((int)p.getX(), getSize().height - (int)p.getY(), 10, 10);
        p.step(dt, f);
        time += dt;
    }
    repaint();
}

public void changeColor(Graphics g)
{
    switch(counter++)
    {
        case 0:
            g.setColor(Color.red);
            break;
        case 1:
            g.setColor(Color.blue);
            break;
        case 2:
            g.setColor(Color.green);
            break;
        case 3:
            g.setColor(Color.yellow);
            break;
        case 4:
            g.setColor(Color.magenta);
            break;
        case 5:
            g.setColor(Color.orange);
            break;
        case 6:
            g.setColor(Color.cyan);
            break;
        case 7:
            g.setColor(Color.pink);
            break;
        default:
            g.setColor(Color.black);
    }
}

• Note the use of the method repaint which clears the Frame and calls paint. However the trajectories are still on the offscreen buffer so that they will all appear on the screen.
• We have added a method, changeColor, which uses a different
color each time a new trajectory is drawn, up to 8 trajectories. Any more trajectories are colored black. Each time a trajectory is drawn, the value of counter is checked and then incremented (using counter++;). (Note that counter++; means that the current value of counter is used in the switch statement before it is incremented by 1. If we used ++counter; instead, counter would be incremented by 1 first.) The value checked then determines which statement to execute. Also note that without the use of the break statement, every case would be executed.

- The switch statement is useful if there are many possible choices. If there are only a few choices or if the choices cannot be encoded as an integer, then the if statement is used. Examples of the latter are as follows:

```java
if (x < 0)
{
  y = 20;
  x = 10;
}

if (x < 0)
  y = 10;  // braces not needed for single statement
else
  y = 20;

if (x < 0)
  y = 10;
else if (x > 0)
  y = 20;
else
  y = 30;
```

**Animation.** We now return to a single trajectory and discuss animation. Instead of showing the trajectory, we will show a ball as it moves. We have reduced the time step so that the motion of the ball will be slowed down.

```java
import java.awt.*;

public class Simulation extends Frame
{
    Image offscreen;

    public static void main(String[] args) {
        // set up window for application
        Simulation sim = new Simulation();
        calculateTrajectory();
    }

    public Simulation()
    { // constructor
        setSize(512, 342);
    }
```
setVisible(true);
offscreen = createImage(getSize().width, getSize().height);
}

public void paint(Graphics g)
{
setBackground(Color.white);
g.drawImage(offscreen,0,0,this);
}

public void calculateTrajectory()
{
final double tmax = 10.0;
final double dt = 0.005;
Graphics b = offscreen.getGraphics();
Particle p = new Particle(0.0, 200.0, 40.0, 25.0);
Force f = new Force();
double time = 0.0;
while (time < tmax)
{
Rectangle oldRect = new Rectangle((int)p.getX(), getSize().height - (int)p.getY(),11,11);
p.step(dt,f);
time += dt;
// new region of ball
Rectangle newRect = new Rectangle((int)p.getX(), getSize().height - (int)p.getY(),11,11);
Rectangle r = newRect.union(oldRect); // new plus old region
// clear new plus old region on buffer
b.clearRect(r.x,r.y,r.width,r.height);
b.fillOval((int)p.getX(), getSize().height - (int)p.getY(), 10, 10);
Graphics g = getGraphics();
g.drawImage(offscreen,0,0,this);
}
}

The basic strategy used above is to define three rectangles. One containing the old position of the ball, one containing the new position, and one which is the union of the two. Then we clear the buffer of this latter rectangle, and draw the new ball. We then draw the image to the screen. Note how we grab the Graphics object for the screen within calculateTrajectory instead of using repaint. This use avoids some of the flickering that would occur from clearing the Frame.

Threads. Often we want the user to be able to interrupt a simulation while it is running, and then start it again. For example, we might want to know the coordinates of the ball at some instant while we are watching it. We can use Threads to accomplish this. Basically, a Thread is a process that is running in parallel with another process. Below we set up a Thread and use a MouseListener to interrupt the program. The interface Runnable contains the methods we need to start, stop, and run the Thread.

```java
import java.awt.*;
import java.awt.event.*;
```
public class Simulation extends Frame implements MouseListener, Runnable {
    Image offscreen;
    Graphics gb; // graphics buffer
    Particle p;
    Force f;
    boolean running = false;
    Thread runner;

    public static void main(String[] args) {
        Simulation sim = new Simulation(); // set up window for application
    }

    public Simulation() {
        // constructor
        setSize(512, 342);
        setVisible(true);
        offscreen = createImage(getSize().width, getSize().height);
        p = new Particle(0.0, 200.0, 40.0, 25.0);
        f = new Force();
        gb = offscreen.getGraphics();
        start();
        addMouseListener(this);
    }

    public void paint(Graphics g) {
        setBackground(Color.white);
        g.drawImage(offscreen, 0, 0, this);
    }

    public void start() {
        runner = new Thread(this);
        runner.start();
    }

    public void stop() {
        runner.stop();
    }

    public void run() {
        while (true) {
            try {Thread.sleep(5);} // delay 5 msec between updates
            catch (InterruptedException e) {
                if (running) calculateTrajectory();
            }
        }
    }

    public void mouseClicked(MouseEvent e) {
        Graphics g = getGraphics();
        g.clearRect(0, 0, getSize().width, getSize().height);
    }
}
We have modified calculateTrajectory so that it only computes and draws one step. We have eliminated the `while` statement. Instead control of the trajectory is given to the `run` method of the `Thread runner`. Note the code within this method. The `try catch` statement is used to interrupt the Thread for 5 milliseconds to check to see if any event occurred. The `MouseListener` checks for events. In this case we can stop the motion by moving the mouse outside the Frame, and start it again by moving it into the Frame. If the mouse is clicked, we write the \( x \) and \( y \) coordinates of the ball at the location of the mouseclick, redraw the ball, and stop the motion. Note that the interface methods `mousePressed` and `mouseReleased` must be defined even though they do not do anything in this example.

References


[11] Recent developments in computer simulation are discussed in the Computer Simulation column, edited by the authors, that appears in the bimonthly publication Computing in Science and Engineering.
[12] An undergraduate physics major in the United States is a four-year program leading to a Bachelor of Arts or Bachelor of Science degree in physics. The age of students is typically 18 to 22 years.
[14] Information about Physics Academic Software can be found at http://webassign.net/pasnew/.
[15] Information about the Computational Physics course at Davidson College can be found at http://webphysics.davidson.edu/Course_Material/Py200/.
[17] Information about the Japanese of Ref. 16 can be found at ftp://spdg1.sci.shizuoka.ac.jp/pub/sip/english/index.html.
[23] Information about the Statistical and Thermal Physics (STP) curriculum development project is available at http://stp.clarku.edu/.