Computational physics in the introductory calculus-based course

Ruth Chabay[a] and Bruce Sherwood[b]
Department of Physics, North Carolina State University, Raleigh, North Carolina 27695

(Received 28 September 2007; accepted 21 December 2007)

The integration of computation into the introductory calculus-based physics course can potentially provide significant support for the development of conceptual understanding. Computation can support three-dimensional visualizations of abstract quantities, offer opportunities to construct symbolic rather than numeric solutions to problems, and provide experience with the use of vectors as coordinate-free entities. Computation can also allow students to explore models in a way not possible using the analytical tools available to first-year students. We describe how we have incorporated computer programming into an introductory calculus-based course taken by science and engineering students. © *2008 American Association of Physics Teachers.*
[DOI: 10.1119/1.2835054]

I. INTRODUCTION

In the past it was reasonable to divide the scientific (and engineering) disciplines into theory and experiment. The ubiquity of computers and computational tools has changed this dichotomy significantly. Computation is a central tool in both theory and experiment, has altered the nature of both in fundamental ways, and has become a third pillar of physics. In some cases it has blurred the distinction between theory and experiment; for example, theorists perform Monte Carlo calculations and high-energy experimentalists construct neural networks to identify particular events in a mass of data. Just as beginning students have traditionally been introduced to both theory and experiment in introductory courses, students in these courses ought to be introduced to the role of computation in science.

The traditional introductory physics course has changed little in the past fifty years and, in most institutions, the course runs smoothly. It is inevitable that the introduction of a significant computational component into the course will rock the boat, simply because we do not have fifty years of experience to rely on in this area. Without this change, however, the course will become increasingly out of date, failing to make contact with the practice of science, and will become increasingly disconnected from the world in which students live. The introductory biology course is now moving toward the integration of a computational component.[1] The National Science Foundation with its new CPATH program[2] is trying to stimulate a much needed expansion of computational science in undergraduate science curricula by forging alliances between computer scientists and natural scientists.

The potential benefits of the integration of computation, including programming, into the course are high, and sufficient experience has been gained to support departments and instructors who wish to make this change. This article includes the details of one particular approach that has been developed over the past ten years in the context of the Matter & Interactions curriculum,[3–7] and which has now been adopted and adapted at other institutions. The particular features of the computational environment in use (VPython,[8] a 3D programming language) are described in an Appendix. VPython, which is based on Python, is an open source project and is free and available on all platforms.

II. GOALS OF COMPUTATION IN THE INTRODUCTORY COURSE

The introduction of computation into an introductory course makes sense only to the extent that it supports the goals of the course. Conceptual understanding of fundamental physics principles, and skill at solving a variety of problems, are typical desired outcomes of an introductory physics course. The introduction of appropriate computational activities has the potential to contribute significantly to learning in both of these areas.

A. Conceptual understanding of fundamental principles

The difficulty of building a firm conceptual understanding of principles such as the momentum principle (Newton's second law) or the nature of a field is widely documented.[9] One factor that contributes to this difficulty is that the traditional approach, limited to the use of algebra and simple calculus, deals only with a small number of situations in which a closed form solution is accessible. In mechanics, these situations include constant acceleration, constant speed circular motion, and harmonic oscillations. In electricity and magnetism, expressions for the fields of various charge distributions are typically calculated only at special locations, such as along the axes of dipoles, rods, and disks. Cognitive science research has established that reasoning using mental models which support step-by-step thinking about processes is more natural and easier for novices than reasoning from closed form constraint-based solutions.[10] To understand the dynamic nature of the momentum principle, for example, a student might need the experience of applying it to predict the motion of interacting objects, step by step, in an open-ended manner. To understand the global nature of the superposition principle, students might need the experience of dividing charge distributions into pieces and adding up their vector contributions one at a time, watching the result accumulate. Although these ideas may be central in instructors' minds, for many introductory level students the analytical use of calculus and a focus on closed-form solutions does not connect to these procedures.

These insights may be difficult to obtain any other way. Ganiel[11] reported that in developing a unit on chaos, he and his colleagues found that students who had completed the traditional introductory physics course did not have a deter-

ministic view of classical mechanics. The group found it necessary to create a module on determinism to precede the material on chaos.

Our central goal of introducing computation into the introductory course is to provide experiences that can support the students' development of conceptual understanding of fundamental principles. Computational activities can also help students make the connections between the formalism of integral calculus and the procedure of adding up discrete quantities, a connection that is often not clear even to students who have taken one or more semesters of calculus.

B. Visualization

Many of the quantities central to physics are abstract three-dimensional vectors such as momentum, angular momentum, and electric and magnetic fields. These quantities can change dynamically in time and space. Few students in the introductory course have had previous practice imagining a momentum vector changing in 3D as an object moves, or imagining the varying magnetic field of a moving charge at many locations throughout space. Although canned images or movies can be useful, the experience of constructing dynamic 3D visualizations by using the basic principles of physics can add concreteness to entities such as observation locations and relative position vectors.

Interpreting 2D representations of 3D situations can be difficult for novice students, even if the situation is one which, to an expert, is adequately represented in 2D. For example, although the trajectory of a particle in a cyclotron is planar, we have observed even well-prepared students displaying confusion about the process of accelerating a proton when solving problems on paper or when writing computer programs to model the process in 2D. However, in a 3D programming environment, the addition of a 3D wire frame model of the cyclotron dees and 3D arrows representing the magnetic and electric fields clarified the situation for many students. The interactivity of the 3D display, which the user can zoom and rotate, was also a helpful feature.[12]

C. Modeling complex situations

Almost all of the problems solved by students in the traditional introductory course are highly idealized. This idealization contributes to the view held by many students that physics has little or no relevance in the real world. Computational modeling provides a venue in which students can begin with a simplified, idealized model of a situation, and then add features which make the model more realistic. By employing only the simplest numerical integration technique, students can analyze situations that are not accessible analytically at the introductory level (or, in some cases, at any level), such as elliptical or parabolic orbits, three-body interactions, 3D spring-mass oscillations, or motion damped by air resistance, sliding friction, or viscous friction.

Electric and magnetic fields can be calculated and displayed at many observation locations, including locations not on an axis of symmetry. Students can explore the range of validity of approximations. For example, how far from the perpendicular axis of a uniformly charged rod must an observer be before the electric field differs significantly from the field on the axis? How close to a dipole can an observation location be before the actual field differs from the approximate $1/r^3$ field by a certain amount?

D. Rethinking the curriculum

The addition of a computational component to a course necessarily requires that some existing content or component be reduced. Thinking about how to make such a change in an established curriculum offers a useful opportunity to reflect on the underlying goals of the course. We should not assume that the existing course content and pedagogy are necessarily ideal; the traditional course was shaped in part by the limitations of the mathematical tools available to beginning students more than fifty years ago. For example, although much work has been devoted to teaching the three solutions to the equations of motion accessible with elementary calculus (constant acceleration, constant-speed circular motion, and harmonic oscillations), restricting our discourse to these special cases may not be adequate to allow students to see the generality of the underlying principles. The considerations leading to the particular set of choices we have made are too involved to describe here; they are documented in Refs. 3–7.

III. PROS AND CONS OF PROGRAMMING

The ability to write a program to solve a problem is a useful skill. In addition, programming offers practice in algorithmic thinking, which is a powerful intellectual tool. Students who are introduced to computational physics in the introductory level course have a good foundation on which to build in later physics courses and other computationally oriented courses in science or engineering.

Programming is a powerful modern metaphor which has had a major impact on thinking in all scientific disciplines. An in-service high school physics teacher enrolled in a distance education version of Matter & Interactions reported that when he read a popular science article about the modeling of weather and of climate, for the first time he had the necessary background to understand the nature of such computer modeling, and why weather prediction required smaller cell sizes for accuracy than did climate modeling. Doing computational physics in the form of programming, even at the introductory level, can be an important component of a general education for living in today's world. Without this experience, even people with a background in science may not appreciate the strengths and limitations inherent in a result obtained from "computer modeling."

There are many possible computational modalities that might be productively incorporated into the calculus-based introductory physics course. For example, students might specify differential equations to be solved by an equation solver, or vary parameters in a simulation. The goal of this paper is not to survey all such modalities and tools, but to discuss in some detail the rationale for having students write programs and how we have integrated programming into the course.

For the past decade we have incorporated programming into introductory courses at Carnegie Mellon and North Carolina State. The Matter & Interactions curriculum is also being used in large courses at Purdue and Georgia Institute of Technology and in small courses at institutions such as Carleton. In these courses students write computer programs as an integral part of the introductory course. The environment used, VPython, which is discussed in the Appendix, has a number of features which make it particularly suitable for this purpose. In the following, we discuss the positive and negative aspects of integrating computer programming into

the introductory course. Although some of these aspects apply to any computational environment, some are specific to programming, and some of these are specific to VPython.

A. Positive aspects of programming

Perhaps the most significant advantage of writing programs from scratch is that there are no "black boxes": students write all of the computational statements to model the physical system and to visualize the abstract quantities. In doing so, students must bring together various components of their physics knowledge; for example, identify all interactions, describe them mathematically, and correctly write and apply fundamental principles such as the momentum principle. All the physics is nakedly exposed in the program.

Creating a program gives students the opportunity to link multiple representations: algebraic equations; the same equations translated into similar program syntax; an animation of physical objects moving on the screen; a dynamic graph of a quantity such as kinetic or potential energy that is generated as objects move. Experts use multiple representations in problem solving; novices need to practice connecting different representations such as equations, graphs, and diagrams. Writing program statements may be the only practice students obtain in solving problems symbolically, because it is increasingly the case in big introductory courses that daily homework is done through a web-based homework system which emphasizes randomized numerical answers. The observed behavior of objects in a running program can support connections between representations; for example, if the sign of a gravitational force is reversed, the student may observe a planet repelling a moon—a dramatic consequence of an error in a symbolic statement.

Writing computer programs can vivify the universality of fundamental physics principles. Students who write a program to simulate Rutherford scattering sometimes spontaneously comment that their computation is essentially the same as their earlier modeling of a binary star, despite the quantitative difference between the gravitational and electric force, and a difference of scale of 10^{23}.

In VPython, coordinate-free vector operations can be done directly in programming statements. For example, a relative position vector can be calculated in a single vector subtraction statement involving 3D position vectors such as

```
r=Moon.pos–Earth.pos
```

Programming in this environment offers the opportunity to conceptualize vectors as powerful single entities rather than as trigonometric manipulations of components.

A computer program can be open-ended, allowing students to observe the behavior of interacting objects indefinitely into the future. This behavior can be interesting in situations such as three-body motion, where the nature of the trajectories is not obvious. It is easy to change a time step or spatial increment in a program and observe the effects of this change on the predicted behavior of the objects or predicted pattern of fields.

Programming offers an opportunity for stimulating creativity in physics assignments, something that is more difficult to do with traditional homework. With a relatively small investment, a student can go beyond what is required in an assignment and explore modifications to a program which can produce unusual, surprising, and beautiful results.

A practical consideration is that scientific programming can be a highly useful skill for students in their later careers in scientific or technical fields.

B. Negative aspects of programming

The most significant barrier to the integration of programming into the introductory course is the fact that most students have never before written a program. Current students are very knowledgeable about all aspects of computers except programming. At least half of the engineering and science students at North Carolina State have never written any kind of computer program before entering the introductory mechanics course, and most of the other students have had minimal experience. Time spent teaching programming concepts and program syntax can decrease the time spent on learning physics.

In an already full introductory physics curriculum, there is little or no time to teach major programming skills. Students who are new to programming are also new to the process of debugging; teaching debugging strategies requires even more time. (Many of the strategies used to debug a program are the same strategies students should be using in checking and correcting solutions to pencil and paper problems. However, program bugs can provoke more frustration because the failure of a buggy program to run is much more salient than the failure of a buggy written solution to produce a reasonable answer.)

Working on programming activities only once a week in a lab or recitation section may not be adequate to keep knowledge of syntax and program structure fresh in the students' minds. Especially early in a semester, there may be a startup transient for each activity, in which the students need to call up and refer to their previous programs to remind themselves how to do something.

IV. HOW CAN PROGRAMMING IN AN INTRODUCTORY COURSE BE FEASIBLE?

To integrate computation, especially programming, into an introductory course, it is necessary to minimize the amount of non-physics related material that must be taught. To do so it is necessary to teach a minimal subset of programming constructs; employ an environment and language that are easy to learn and use; ensure that program constructs match key physics constructs; provide a structured set of scaffolded activities that introduce students to programming in the context of solving physics problems; and provide a supervised setting in which the students can work on these activities in the presence of an instructor who can offer help with debugging.

The minimum programming concepts required to implement and visualize simple physics models are the creation of objects such as spheres and arrows to represent physical or abstract entities; the specification of initial values and attributes (including vector values such as velocity); exactly one way to iterate; and one way to update the value of a variable or an attribute.

This minimal set of concepts will allow students to construct programs that, although sometimes inelegant by professional coding standards, will adequately embody physical models. It is important to note some things that are intentionally not included in this list, including how to write user interfaces, how to do graphics coding, and how to implement

more sophisticated numerical algorithms. These omissions impose stringent conditions on the environment; in particular, graphical output and a basic user interface must be provided automatically, as effortless consequences of physics computations.

A. An example of a student program

The following VPython program is an example of a moderately complex program which models a restricted three-body interaction involving a fixed Earth, a fixed Moon, and a spacecraft. In the introductory course at North Carolina State, students write a similar program during two one-hour sessions (part of the lab activities for two successive weeks), at about the seventh and eighth weeks in the semester. Later they add graphs of the kinetic energy and gravitational potential energy versus the time to the program. After writing the program, students explore initial conditions to see what kinds of trajectories they can produce. The complex behavior emerging from simple interactions illustrates the predictive power of fundamental physics principles and is a graphic example of the nature of the momentum principle. Although the trajectory is an example of classical determinism, it is extremely sensitive to the initial conditions, which hints at one of the important aspects of chaos.

Note that the vector algebraic statements in VPython are a direct conversion of the algebraic statements students write on paper. Also note that in the loop there are no graphics statements. The 3D display automatically updates many times per second, using the updated object positions.

In Python a pound sign (#) introduces a comment. Here is the complete program:

```
from __future__ import division # always do floating point arithmetic: 1/2 = 0.5
from visual import * # import 3D graphics module

#CONSTANTS
G = 6.7e-11
mEarth = 6e24
mcraft = 150e3
mMoon = 7e22
deltat = 100

#OBJECTS AND INITIAL VALUES
Earth = sphere(pos=vector(0, 0, 0), radius = 6.4e6, color = color.cyan)
craft = sphere(pos=vector(-10*Earth.radius, 0, 0), color = color.magenta)
vcraft = vector(0, 3.2703e3, 0)
pcraft = mcraft*vcraft
Moon = sphere(pos=vector(4e8, 0, 0),radius = 1.75e6)
pa = arrow(color=color.green) # arrow for representing momentum
sf = 0.1 # scale factor for displaying momentum arrow pa
trail = curve(color=craft.color) # craft trail: starts with no points
t = 0
scene.center = (Moon.pos.x/2, 0, 0) # instead of center of scene at origin

#CALCULATIONS

while t < 10*365*24*60*60: # continue plotting for 10 Earth years
    rate(200) # slow down loop to make animation look nicer
    re = craft.pos - Earth.pos
    rehat = re/mag(re)
    Fge = -rehat*G*mEarth*mcraft/mag(re)**2
    rm = craft.pos-Moon.pos
    rmhat = rm/mag(rm)
    Fgm = -rmhat*G*mMoon*mcraft/mag(rm)**2
    Fnet = Fge + Fgm
    pcraft = pcraft + Fnet*deltat
    craft.pos = craft.pos + (pcraft/mcraft)*deltat
    pa.pos = craft.pos # position tail of momentum arrow on spacecraft
    pa.axis = pcraft*sf # scale the arrow representing momentum
    if mag(re) < Earth.radius:
        break # leave loop if crashed on Earth
    if mag(rm) < Moon.radius:
        break # leave loop if crashed on Moon
    trail.append(pos=craft.pos) # add new position of the craft to the trail
    t = t + deltat
print 'Calculations finished after ', t, 'seconds'
```

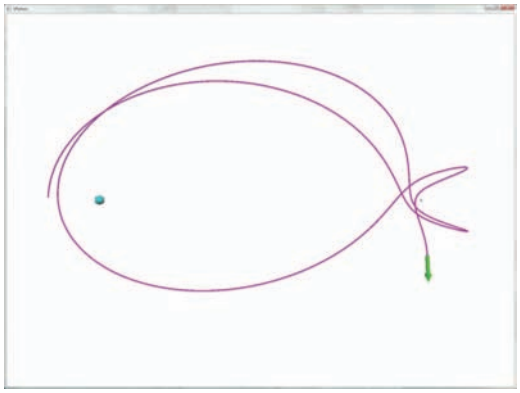

Fig. 1. Restricted three-body program written by a student: a spacecraft coasting near a stationary Earth and stationary Moon. An arrow represents the current momentum of the spacecraft. (The sphere representing the Moon is very small because the Earth and Moon are drawn to scale.)

A snapshot of the screen display produced by the program is shown in Fig. 1. Note that the length of an arrow representing the momentum (in kg·m/s) must be scaled to fit on the display, whose width is in (virtual) meters. Such scaling is an issue of representation in all physics diagrams, but it arises explicitly only in this computational environment, which presents an opportunity to distinguish between abstract vectors (here, momentum) and their representation as arrow objects.

One reason the students' code need not be elegant is that the programs they write are very short. In most problems appropriate to the introductory course, sophisticated algorithms are not needed because computers are now fast enough to make it possible to increase the accuracy of an Euler integration adequately simply by reducing the step size.[13]

Figure 2 represents the 3D motion of a mass hanging from a spring. Students are challenged to find initial conditions that produce oscillations in 3D; they are often intrigued by the patterns their programs can produce.

B. Choosing a computational tool

The choice of a computational tool is often unnecessarily contentious. Ideally, students should have the opportunity during their undergraduate years to use several different kinds of tools including an algorithmic programming language such as C++, Java, or Python; a symbolic processor such as Maple or Mathematica; an environment such as Matlab; and a spreadsheet. Each tool has particular advantages for certain uses. It is important to keep in mind that no matter what tools a student uses today, different tools will be popular tomorrow. Learning a programming language can provide a foundation for learning new tools in the future.

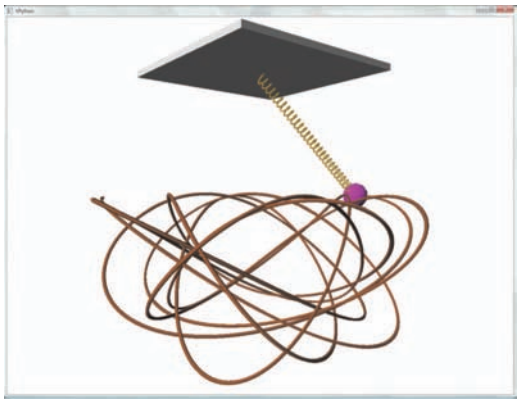

Fig. 2. Program written by a student to model the 3D motion of a mass hanging from a spring leaving a trail. Students did experiments with a similar apparatus.

V. COMPUTATIONAL ACTIVITIES

After a decade of work on developing computational activities for the introductory course, we cannot claim that our instructional sequence realizes the full educational potential of computation that we have outlined in previous sections. Much research and development still remains to be done in this area. However, we now have a sufficient collection of activities to make it possible for other instructors to build on what we have developed.[14]

The following lists detail the sequence of computing activities in recent offerings of the introductory course at North Carolina State, using Matter & Interactions. The specific activities vary somewhat from one semester to the next. Recent versions of each of these assignments are available.[15]

A. Mechanics

In dynamics problems involving computer modeling, the momentum principle is invoked to predict the behavior of objects or systems of objects ($\Delta \boldsymbol{p} = \boldsymbol{F}_{\text{net}} \Delta t$ for sufficiently small values of Δt). Given the initial positions of the interacting objects and a force law, we can calculate the forces the objects exert on each other. These forces can be applied for a short time to update the momenta, and the new momenta can be used to update the positions of the objects. The student writes a simple computer program to repeat this process many times. In these problems, kinematics is united with dynamics: changes in motion are clearly caused by interactions. The total energy of the system can be plotted as a check on the accuracy of the calculations. Simultaneous consideration of momentum and energy also helps students distinguish between these concepts.

The following computational activities are part of the first semester introductory course.

(1) Introduction to VPython and 3D vectors. Create sphere objects, display 3D arrows to represent position vectors, including relative position vectors. Learn to use symbolic names to perform vector operations.
(2) After experimenting with a fan-driven cart, write a program to model the motion of such a cart on a track by writing an iterative loop containing updates of the momentum and position. Empirically determine the initial conditions that produce certain behavior; experiment with (unphysical) nonzero y or z component of velocity.
(3) A sequence on gravitational interactions.
    (a) Compute and display the 3D gravitational force exerted by a planet on a spacecraft at different static locations.
    (b) Launch a spacecraft near Earth. Model the motion of the craft under the influence of the changing gravitational force. Observe the effect of initial conditions on the shape of the orbit.

(c) Consider a spacecraft, fixed Earth, and fixed Moon (see Fig. 1). Explore the effect of the initial velocity on the trajectories.
(d) Add graphs of the kinetic energy, gravitational potential energy, and the sum for the spacecraft, fixed Earth, and fixed Moon. Graphs are generated in real time as the motion occurs.
(e) Alternatively model the motion of a binary star system in a frame of reference in which the total momentum of the system is nonzero.

(4) A sequence on spring-mass oscillations.
(a) After experimenting with real spring-mass oscillators, write a program to model the motion of 1D and 3D vertical oscillators, using experimental data for mass, spring length, and spring stiffness. Compare the predicted and observed behavior. For 1D oscillations, plot y versus t. Find initial conditions that produce 3D oscillations (Fig. 2)
(b) Add energy plots to the spring-mass program.

(5) Rutherford scattering (two moving particles): compute and display the motion of an alpha particle approaching a gold nucleus, and construct graphs of the horizontal and vertical components of the momentum for each particle. Vary the impact parameter and observe the effects.
(6) A sequence on entropy, temperature, and specific heat capacity of a solid.[16]
(a) Plot the number of ways to distribute a fixed amount of energy between two nanoparticles modeled as Einstein solids (independent quantized oscillators). Vary the sizes of the two nanoparticles to observe that equipartition of energy is most probable.
(b) Graph the entropy in terms of the natural logarithm of the results from the previous program.
(c) Compute the temperature from the inverse rate of change of the entropy with energy added to a nanoparticle.
(d) Compute and graph the specific heat (from second differences) as a function of temperature for aluminum and lead. Experimental data are provided. Vary the effective stiffness of the interatomic bond to fit the curves to data. Good fits are observed with a stiffness that is consistent with measurements of Young's modulus. (Students had previously measured Young's modulus for another metal in the lab.)

B. Electricity and magnetism

Calculating and visualizing electric fields involves the same vector operations used to calculate gravitational forces in the mechanics course and the same issues with scaling arrows arise. One goal of this activity is to give students a sense of the 3D character of the fields. VPython automatically allows the user to rotate and zoom using the mouse, which enhances 3D perception.

The following computational activities are part of the second semester introductory course. The *Advanced* items have been used successfully with advanced or honors students.

(1) Introduction to VPython and 3D vectors. (For those students who previously took the mechanics course, this introduction is a review.)
(2) A sequence on electric fields, distributed charges, and superposition.
(a) Compute and display with arrows the electric field at various 3D locations near a positive or negative point charge.
(b) Compute and display the electric field of a dipole. First use of the superposition principle. Display the field at various 3D locations, including off-axis locations.
(c) Compute and display the electric field of a uniformly charged rod or ring. Model the object as a line (or circle) of point charges. Experiment with the effect of increasing or decreasing the number of charges used to model the object. Calculate and display the field at locations where an analytical solution is not available.

(3) A sequence on magnetic fields and forces.
(a) Use the Biot–Savart law to calculate and display the magnetic field of a moving proton at several locations as the proton moves in a straight line. Note that the field at a particular location varies as the source moves.
(b) Compute and display the helical motion of a proton in a uniform magnetic field.
(c) *Advanced*: Model the motion of a proton in a cyclotron showing the electric and magnetic fields affecting its motion.
(d) *Advanced*: Calculate the magnetic field throughout a solenoid of finite length using the Biot–Savart law.

(4) *Advanced*: Compute the motion of a positron in a sinusoidal electromagnetic wave showing the dynamically varying electric and magnetic field vectors along a line. Start a positron at rest and model its motion (relativistically) under the influence of the electric and magnetic fields (ignoring radiation by the accelerated positron).

ACKNOWLEDGMENTS

These developments were supported in part by the National Science Foundation through Grant Nos. DUE-0320608, DUE-0237132, and DUE-0618504.

APPENDIX: THE VPYTHON PROGRAMMING ENVIRONMENT

Our choice of a programming environment for a first exposure to scientific computation is based on the desire to ease students into programming and to give them the opportunity to develop a conceptual model of what a program is and what it does.

Among algorithmic programming languages, Python[17] is increasingly used in a variety of environments. Python has a good reputation among computer scientists because of its clean design and object-oriented nature, and it is sometimes the preferred first language for introducing programming to students. There is a comprehensive library of modules for tasks as diverse as manipulating images and running web servers. There is a large community of scientific users.[18] Much of the programming done by Google is in Python, and programmers knowledgeable in Python are in demand.

Python is maintained and extended by a large community of users. It is open source, multi-platform, and free. A recent

special issue of *Computing in Science & Engineering* was devoted to the use of Python.[19]

VPython is the name given to the combination of Python plus a module called Visual (and the numpy module, which is the basis for Visual's vector capabilities). The Visual module and numpy are also open source, multiplatform, and free. Visual adds to Python the easy creation of navigable 3D animations and the ability to manipulate 3D vectors mathematically, instead of dealing only with separate components of vectors.

VPython was created in 2000 by David Scherer,[20] then an undergraduate student at Carnegie Mellon, who, after taking the *Matter & Interactions* course, had the original idea of making navigable 3D animations be a side effect of physics computations. While the calculations are being done, a parallel thread periodically (many times per second) creates a 3D image in OpenGL corresponding to the current attributes of objects declared by the program. The effect is that, without any explicit graphics statements in the computational loop, a window appears with a 3D animation of the motion of the objects created by the program, and the user can navigate in the scene by rotating and zooming with the mouse.

Because VPython supports standard vector computations, students are encouraged to view vectors as powerful tools for analysis rather than as unpleasant trigonometry. After establishing initial conditions within a standard 3D Cartesian coordinate system, all of the iterative computations are written as coordinate-free vector statements. In the example program reproduced in Sec. IV A, the relative position vector between two bodies is calculated by vector subtraction, the unit vector is calculated by dividing the relative position vector by its magnitude, and the gravitational force is calculated as a product of the magnitude and the unit vector, which is calculated by dividing the relative position vector by its magnitude. (There exists a norm() function to calculate a unit vector directly, but beginning students often find it clearer to replicate in code the calculation they do on paper.) The net force (a vector) is used to update the momentum (a vector), and the position updates are also written as vector statements.

[a]Electronic address: ruth_chabay@ncsu.edu
[b]Electronic address: bruce_sherwood@ncsu.edu

[1]"BIO2010: Transforming undergraduate education for future research biologists," ⟨www.nap.edu/openbook.php?isbn=0309085357⟩.
[2]"CISE pathways to revitalized undergraduate computing education (CPATH)," ⟨www.nsf.gov/pubs/2006/nsf06608/nsf06608.htm⟩.
[3]R. Chabay and B. Sherwood, *Matter & Interactions I: Modern Mechanics and Matter & Interactions II: Electric & Magnetic Interactions* (Wiley, New York, 2007), 2nd ed.; also see ⟨www4.ncsu.edu/~rwchabay/mi⟩.
[4]R. Chabay and B. Sherwood, "Bringing atoms into first-year physics," Am. J. Phys. **67**(12), 1045–1050 (1999).
[5]R. Chabay and B. Sherwood, "Modern mechanics," Am. J. Phys. **72**(4), 439–445 (2004).
[6]R. Chabay and B. Sherwood, "Restructuring the introductory electricity and magnetism course," Am. J. Phys. **74**(4), 329–336 (2006).
[7]R. Chabay and B. Sherwood, "Matter & interactions," in *Research-Based Reform of University Physics*, edited by E. F. Redish and P. J. Cooney ⟨www.compadre.org/per/per_reviews/volume1.cfm⟩.
[8]VPython, ⟨vpython.org⟩.
[9]L. C. McDermott and E. F. Redish, "Resource Letter: PER-1: Physics Education Research," Am. J. Phys. **67**(9), 755–767 (1999).
[10]P. N. Johnson-Laird, *Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness* (Harvard U. P., Cambridge, MA, 1986).
[11]Uri Ganiel, private communication (January 2005).
[12]B. Tversky, J. B. Morrison, and M. Betrancourt, "Animation: Can it facilitate?," Int. J. Hum.-Comput. Stud. **57**, 247–262 (2002).
[13]A notable previous effort to incorporate computational physics in the form of programming into an introductory course is discussed in E. F. Redish and J. M. Wilson, "Student programming in the introductory physics course: M.U.P.P.E.T.," Am. J. Phys. **61**, 222–232 (1993) and ⟨www.physics.umd.edu/ripe/muppet/papers.html⟩. This project involved a small course for physics majors, many of whom had extensive prior programming experience. Because the computers available at the time were much slower than current computers, it was necessary to employ sophisticated Runge-Kutta algorithms in the programs. Interactive 3D graphics were not a possibility (the only output was graphs). Many lines of setup code were necessary for even a simple program. The major increases in the capabilities of both hardware and software since that period have led to a qualitative change in what is feasible in an ordinary instructional setting.
[14]Some of the activities, particularly some of the carefully scaffolded sequences, were developed by Matthew Kohlmyer as part of his Ph.D. dissertation, in which he studied some of the difficulties encountered by introductory students in programming assignments. M. Kohlmyer, Ph.D. thesis, Carnegie Mellon University, 2005.
[15]Matter & Interactions computational labs: ⟨www.compadre.org/psrc/items/detail.cfm?ID=5692⟩.
[16]The statistical treatment of entropy, temperature, and specific heat capacity in *Matter & Interactions* (Ref. 3) is based on T. Moore and D. Schroeder, "A different approach to introducing statistical mechanics," Am. J. Phys. **65**, 26–36 (1997).
[17]Python ⟨python.org⟩.
[18]See ⟨www.scipy.org⟩.
[19]Comput. Sci. Eng. **9**(5), (2007), Special Issue.
[20]D. Scherer, P. Dubois, and B. Sherwood, "VPython: 3D interactive scientific graphics for students," Comput. Sci. Eng. **2**(5), 56–62 (2000).