Editor: Denis Donnelly, donnelly@siena.edu

NUMERICAL PROBLEM SOLVING FOR UNDERGRADUATE CORE COURSES

By Michael E. Peskin

A TYPICAL UPPER-DIVISION UNDERGRADUATE PHYSICS COURSE—FOR EXAMPLE, A COURSE IN CLASSICAL ELECTRODYNAMICS AT THE LEVEL OF GRIFFITHS' TEXT[1]—INVOLVES EXTENSIVE ANALYTIC PROBLEM SOLVING. STUDENTS LEARN TO solve Laplace's equation in rectangular and cylindrical coordinates, with Bessel functions and Legendre polynomials. These methods are important in their own right and provide examples to illustrate central physical concepts.

But students would benefit more if we discussed numerical problem-solving methods at a similar level. Today, students often have computers in their dorm rooms and backpacks that have the power of a typical 1970s university mainframe. It would be helpful if we could use computing power for explicit problem-solving and to develop students' intuition.

In this article, I explain a method that uses the Java programming language's built-in class structure to do this. I also supply a Java class library that can assist instructors in writing programs of this type.

Current Approaches

Including numerical calculations in a physics or engineering curriculum is important because students typically go on to careers in research or industrial settings in which their basic tasks are to model physical systems. Analytic methods are useful for estimates or for working out the dependence on parameters, but understanding a realistic system in detail typically requires a computer simulation. So, we should make it clear to our students that it's straight forward to put the equations that appear in their classes onto their computers to obtain sensible physical results.

We currently address this curriculum issue in two ways. First, we encode specific problems into computer programs, which students can run as black boxes, changing the parameters and seeing what consequences develop. Davidson College's scripted applets provide noteworthy examples of this approach.[2]

Black-box programs are useful in introductory courses but, for upper-division courses, they do not teach all the skills we would like students to develop. These programs also are time-consuming to write. We often provide them under the philosophy that students should not modify the code. (The Davidson applets do not even allow professors to modify the code.) But I would like students to write some code, and to understand how simple sets of instructions can iterate to the patterns that solve interesting equations of nature.

In undergraduate mechanics courses, instructors often give students differential equations to integrate into commercial packages such as Maple or Mathematica. Such projects have a similar black-box approach.

In the other common approach, professors teach computational physics courses modeled, for example, on the textbooks by Gould and Tobochnik[3] or Koonin.[4] Such courses often revolve around major projects as students spend a large part of a semester learning a computer environment and constructing elaborate code for one particular application. These courses are important for giving students experience with large-scale computer applications and beginning to study sophisticated numerical methods. But it also would be advantageous to let students do simpler numerical problems that tie in directly to their core courses.

Ideally, a weekly problem set in mechanics or electrodynamics should consist of several analytical and one numerical problem. The reason we do not usually see this is that it is difficult to have students master the task's purely computer-programming aspects.

A Java Solution

Java is an object-oriented (OO) computer language that lets programs have hierarchical structures. Using Java, you can assign students to write small pieces of code, perhaps a subroutine that carries out a numerical computation. They will then compile their code together with a larger program that implements a GUI for visualizing the program's results. You can assemble the user-interface code yourself, so that students do not have to confront its complexity. Finally, you can simplify writing this larger program by drawing various user-interface elements from a preassembled class library.

This is a version of the familiar strategy of asking students to write subroutines that tie to a larger code package. Java assists this strategy in two ways. First, it lets you neatly encapsulate the student's code, hiding the details of the graphical elements and user interface. Second, it provides you with a programming library that makes it simpler to write interesting and pleasing graphical elements.

We also could use this strategy in other programming languages that allow easy access to GUI components—for example, Visual Basic. The Consortium for Upper-Level Physics Software (CUPS; www.wiley.com/college/math/phys/cg/sales/CUPS.html) project has created a range of effective simulations in Pascal[5] that are extensible at the code level.

An advantage of Java is that the same code runs on Unix, Windows, or Macintosh systems, so students can put together their assignments on whatever operating system (OS) is most convenient. Additionally, Java belongs to a family of computer languages, including C, C++, and Pascal, in which numerical computations have a common syntax. The part of the code that students must write is almost indistinguishable among these four languages, so students need no prior Java experience.

As a part of their work for the second edition of the textbook,[3] Christian, Gould, and Tobochnik are putting together an extensive set of software resources for creating educational simulations in Java (www.opensourcephysics.org/). In contrast to their work, I provide here a minimal set of Java resources to provide exercises of the type I describe.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Laplace extends LaplaceGUI{
  double criterion = 1.0e-2;

  void solve(){
    double maxdiff = 1.0;
    int iteration = 0;
    while ( maxdiff > criterion){
      for (int n = 1; n <= 20; n++){
        maxdiff = 0.0;
        for (int i = 1; i < Nx; i++){
          for (int j =1 ; j < Ny; j++){
     /* check whether (i,j) is a cathode or ground point */
            if (normal(i,j) == false) continue;
              /* update the phi array  */
            double oldphi = phi[i][j];
              /*  put something more sensible here  : */
            double newphi = 33.0;
            phi[i][j] = newphi;
           /*  compute the criterion for stopping */
            double delta = Math.abs(newphi-oldphi);
            if (delta > maxdiff) maxdiff = delta;
          }
        }
        iteration++;
      }
      refreshPicture();
      Legend.write(“max. diff : “+maxdiff+
                                  ”    “+iteration);
      if (timetostop) break;
    }
  }
```

Figure 1. The Laplace.java program. This program gives a simple realization of the solution of Laplace's equation by the relaxation method. The potential is carried in a preassigned lattice `phi[i][j]`. Note that students must supply the one line of code that updates phi correctly.

You can find the Java code for the class library and example programs in a tar file at http://arXiv.org/ps/physics/0302044, as well as more complete software documentation and example programs.

Laplace Applet Example

Textbook discussions of Laplace's equation say that a solution $\phi(x)$ is the average of the solution at neighboring points. This fact can be the basis for a numerical solution of Laplace's equation.[1] Here, I present a homework problem that lets students implement this observation in a numerical program and see it work. Figure 1 shows the program.

The program sweeps through an array `phi[i][j]`, updating successive values. The program tests for the maximum change across the array and quits when the change is sufficiently small.

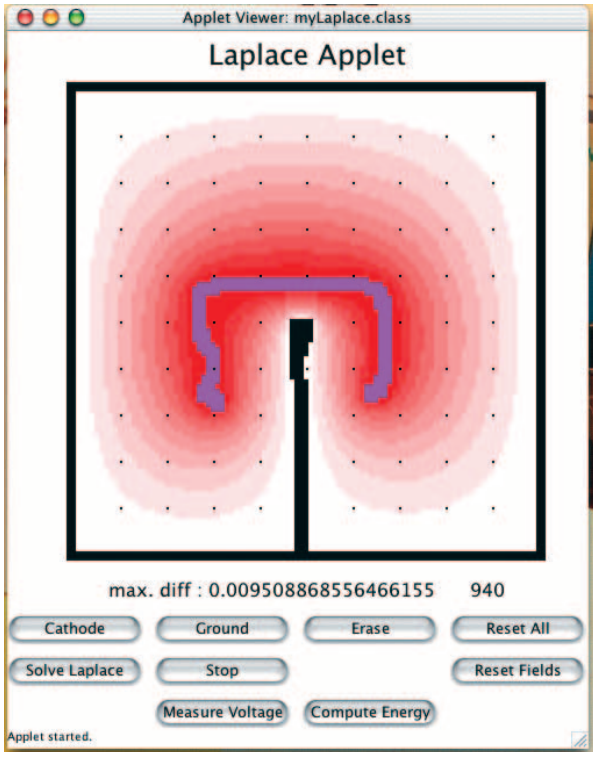


Figure 2. A working version of the Laplace applet. The program implements the method given in Figure 1 with a graphical user interface provided by the instructor.

The criterion for stopping is the variable `criterion`. The algorithm for updating the array is missing.

In the problem set, you would tell your students that the missing algorithm successively sets each array element equal to the average of its neighbors, and that the equilibration of this process yields a solution to the Laplace equation. To implement this algorithm, students must modify only one line of the code, replacing the assignment to `newphi` by

```
double newphi = (0.25)*
(phi[i+1][j]+phi[i][j+1]
+phi[i-1][j]+phi[i][j-1]);.
```

This statement has the same form in C, C++, Pascal, or Java. It should be at least recognizable by students whose only programming experience is in Basic or Fortran.

The `class Laplace extends LaplaceGUI` statement indicates that the simple program Laplace.java in Figure 1 works with functions and data structures defined in a parent program LaplaceGUI.java. This program, its parent PhysicsApplet.java, and a small file, Laplace.html, are given the directory `Laplace` and are available at the URL previously mentioned. You could make these files available for download on the course's Web page.

When you compile these programs together, you get a working simulation toy. Program linking depends on the OS, but under Unix or Macintosh OS X, you simply put the four files in the same directory and type `javac Laplace.java`. You run the compiled program, or applet, by viewing the file Laplace.html with a Web browser. You would not expect students to modify, or even open, any of these files except for the original Laplace.java. This file contains all the physics; the others simply supply the computer interface.

The outcome of this process is a working application (see Figure 2). The applet shows a figure with a box that displays the values of the array `phi` in gray scale. In the example shown, phi is a 100 × 100 array. Fiducial dots mark each 10th grid point to facilitate specific numerical computations.

By clicking on the Cathode and Ground buttons, you can paint a set of boundary conditions with the mouse. Clicking on the Solve Laplace button calls the method `solve()` in the Laplace.java program. As the `phi [i] [j]` array updates, the values of `phi` display on the screen in gray scale. Clicking on the Measure Voltage button and then clicking on the screen causes `phi`'s value at that point to appear as a label under the box.

Once the applet is programmed and working correctly, you can use it for many illuminating exercises. Some of these are qualitative problems, such as illustrating the Faraday cage principle by placing small, grounded conductors around a cathode. Others are quantitative, such as determining the size of edge effects on the capacitance of a finite-sized capacitor.

To aid in the latter calculation, the Compute Energy button computes the electrostatic energy stored in the configuration shown from a discrete approximation to the expression $\int d^2x 1/2\varepsilon_0 E^2$. The problem set that contained this applet asks students to implement an `Energy()` function, called by this button, and returns the result. The GUI then writes the result to the screen.

You can construct many other computational exercises along these lines. Let's look at two of these. (The online software distribution has nine applets, including those I present here.)

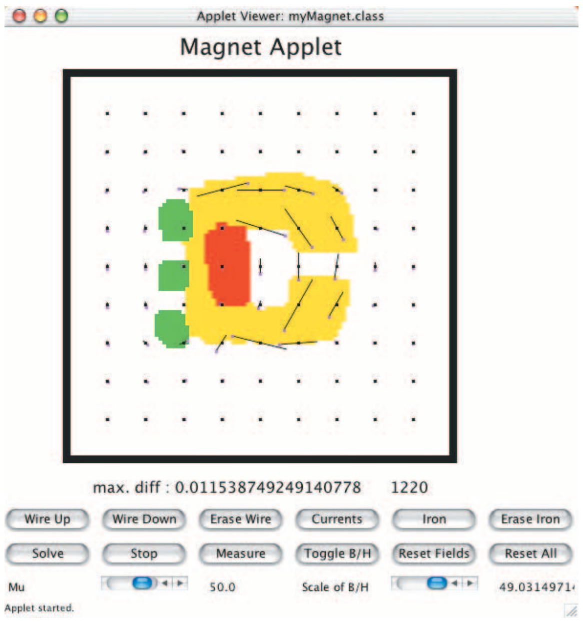
Further Examples

Figure 3a shows an applet that computes the magnetic fields in an array of wires and magnetic material that you draw on the screen. The situation is uniform in the third dimension, with current flowing up or down through the screen. You can assign a fixed current, up or down, to green or red squares, respectively. Students can paint the squares via a mouse. They also can color in regions to be filled with a linear magnetic material (iron) with adjustable permeability μ.

The applet generates magnetic fields from a vector potential $\mathbf{A} = (0, 0, \mathcal{A})$, where $\mathcal{A}$ solves the Poisson equation: $-\nabla^2\mathcal{A} = \mu J$ in which J is the current density in the wires. We can solve this equation with the relaxation method used for the Laplace equation. The problem set containing this applet explains the strategy, then asks students to work out the details using their experience with the Laplace applet. Next, the problem set asks them to use this Magnet applet to solve qualitative examples in the theory of magnetism and quantitative problems of magnet design.

Figure 3b shows an applet that illustrates basic signal-analysis principles. The applet displays three boxes on the screen. In the upper box, students can enter a waveform, either from the computer program as a mathematical function or by drawing it with a mouse. The center box shows the modulus of the waveform Fourier transform. We can multiply this by a filter function supplied by the program or drawn on the screen. The bottom box shows the filtered waveform in real space.

To present this applet in a problem set, I supply my students with five files: FourierLab.java, FourierTransform.java, FourierLabGUI.java, PhysicsApplet.java, and FourierLab.html. The FourierLab.java file contains the functions defining the initial waveform and filter. The file FourierTransform.java contains the methods called by the Transform button. The next two Java programs define the GUI; they do not contain any of the computation's physics and cannot be modified by students. The last file displays the compiled program.

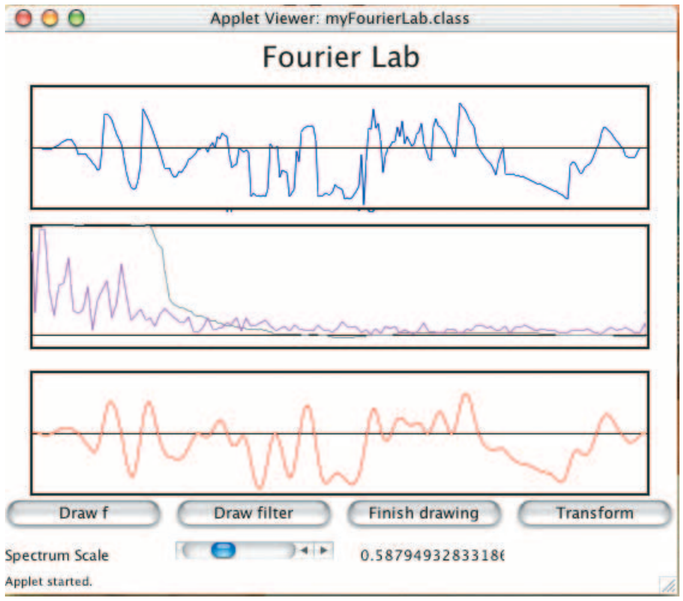


max. diff : 0.0115387492491407781220

50.0

49.0314971

(a)



0.58794932833186

(b)

Figure 3. Working versions of the (a) Magnet and (b) FourierLab applets. These programs illustrate concepts of magnetostatics and signal analysis, respectively.

Figure 4 shows the contents of the FourierTransform.java file I give to my students. The program's structure uses OO programming concepts and is self-explanatory. The real-space function is a real-valued function $f(x)$, represented as an array $f[n]$, $n = 0, \ldots, N - 1$. The real and imaginary parts of the Fourier transform are represented as arrays `fcos`$[m]$ and `fsin`$[m]$, $m = 0, \ldots, N/2$. Using their basic understanding of array treatments in any familiar programming language, students can complete the Fourier transform algorithms so that they work correctly. (You do need to tell students that Java has the peculiar feature that sin x, cos x, and π are written `Math.Sin(x)`, `Math.Cos(x)`, and `Math.PI`.) You could build different versions of FourierTransform.java from this basic struc-

```
public class FourierTransform{

    int Nx, N2;
    double[] f, fcos, fsin;

    FourierTransform(int N){
        Nx = N;
        N2 = Nx/2;
        f = new double[Nx];
        fcos = new double[N2+1];
        fsin = new double[N2+1];
    }

    void Transform(){
        for (int i = 0; i <= N2 ; i++){
            double Tc = 0.0;
            double Ts = 0.0;
             for (int j = 0; j < Nx ; j++){
                /*  do something intelligent here   */
             }
             fcos[i] = Tc;
             fsin[i] = Ts;
        }
    }

    void InverseTransform(){
        for (int i = 0; i < Nx ; i++){
            double T = fcos[0] + fcos[N2];
             for (int j = 1; j < N2 ; j++){
                /*  do something intelligent here   */
             }
             f[i] = 0.0;
        }
    }
}
```

Figure 4. The class FourierTransform.java presented to students. This program is a shell illustrating calls to a predefined array. Students should fill in the Fourier transform algorithms. Figure 3b shows a working version of this program with its GUI.

ture, incorporating different algorithms for computing the Fourier transform.

I constructed the set of programs we have just discussed from a simple, student-accessible physics program and a more complex user-interface program whose contents are unimportant to the students. I have not yet discussed any aspect of the Java programming language, but its OO nature is key to these programs' structure.

Constructing Example Applets

In OO programming, the basic element of a program is a *class*—a set of data variables together with functions (methods) that act on these variables. You can create one class from another in a parent-child relation. The child class is said to inherit the parent's variables and methods; you can add new variables and methods intrinsic to the child. In this way, you can build a complex structure in stages.

You can define an *abstract class* in which one or more methods are defined in principle but are not implemented. You cannot create (instantiate) an abstract class in a computer program, but you can create and use a child of the abstract class that defines the required methods. In Java, each individual class A is defined in a separate file A.java.

The Java language defines an applet—a mini-program accessible through a Web browser—as a predefined parent class. This Applet class manages the window in which the program appears and provides methods to draw in this window.

The definition of the Laplace applet, and of the other examples discussed here, starts from a parent class PhysicsApplet, which is a child class of Applet. This class defines various user-interface elements useful in constructing problem-set applets of the type that I have already presented. The Java language already contains many hooks to graphical elements, making it straightforward to construct basic user interfaces. Many excellent books describe user-interface programming in Java,[6,7] but I hope that the collection of specialized elements contained in PhysicsApplet will make it one step easier for instructors to build their own programs.

Figure 1 shows the Laplace applet constructed as a daughter class of a class LaplaceGUI, which is constructed as a child class of PhysicsApplet. The LaplaceGUI class puts into the applet the specific displays and buttons described earlier. LaplaceGUI still is an abstract class. It defines almost all of the methods of PhysicsApplet but leaves undefined the methods `solve()`, which actually carries out the solution of Laplace's equation, and `Energy()`, which computes the electrostatic energy.

When the class Laplace adds a definition of `solve()` to this structure (as in

Figure 1), and a definition of `Energy()`, all needed methods are defined and the class can be instantiated. The construction hides the definitions of all the other methods of LaplaceGUI.

I present and describe in detail the code for class LaplaceGUI in the online version (http://arXiv.org/ps/physics/0302044). The program is about 150 lines of code, but it is all straightforward bookkeeping, written by calling out the graphical elements one by one and specifying their appearance and function.

You should note two peculiar features of the programming style. First, all the graphic elements rely on modes of operation specified by integer labels. Many GUI books (*Macintosh Human Interface Guidelines*,[8] for example) ask that a user interface be modeless, so a user will have as many options as possible at any given time. The price of this feature is a complex programming style. My philosophy is just the reverse. I want to simplify an instructor's programming task, even if it costs users some flexibility.

Second, I defined the graphical elements, such as buttons and drawing areas, as internal classes of the class PhysicsApplet. This approach goes against Java programming's usual conventions, but it lets you pack the class library into a single file, which students easily can copy without the need for a development environment.

Online Appendix A documents the class PhysicsApplet in detail, including a description of all internal classes.

With the system I described, students can get a taste of numerical problem solving, including programming, as an integrated part of their core courses. I hope that this system will give instructors the opportunity to present more robust and vivid examples to their classes, and that it will motivate students to learn more about computation as applied to problems in physical science.

Acknowledgments
I thank Patricia Burchat, Blas Cabrera, Norman Graf, and Tony Johnson for discussions of the subjects presented here, and to the students in Physics 120–122 at Stanford University, whose help and feedback was essential in developing these materials.

References
1. D.J. Griffiths, *Introduction to Electrodynamics*, 3rd ed., Prentice-Hall, 1999.
2. W. Christian and M. Belloni, *Physlets: Teaching Physics with Interactive Curricular Material*, Prentice-Hall, 2001.
3. H. Gould and J. Tobochnik, *Introduction to Computer Simulation Methods*, Addison-Wesley, 1988.
4. S. Koonin, *Computational Physics*, Addison-Wesley, 1986.
5. R. Ehrlich et al., *Electricity and Magnetism Simulations*, J. Wiley & Sons, 1995.
6. C.S. Horstman and G. Cornell, *Core Java*, 2 vols., Sun Microsystems Press, 1999.
7. D. Flanagan, *Java Examples in a Nutshell*, O'Reilly, 2000.
8. Apple Computer, *Macintosh Human Interface Guidelines*, Addison-Wesley, 1992.

Michael E. Peskin is a professor of physics at the Stanford Linear Accelerator Center, Stanford University. He is interested in how to test exotic ideas about fundamental physics, including superstring theory and extra space dimensions, in high-energy physics experiments. Contact him at mpeskin@slac.stanford.edu; www.slac.stanford.edu/~mpeskin/.

Submissions: Send one PDF copy of articles and/or proposals to Francis Sullivan, Editor in Chief, fran@super.org. Submissions should not exceed 6,000 words and 15 references. All submissions are subject to editing for clarity, style, and space.

Editorial: Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *CiSE* does not necessarily constitute endorsement by the IEEE, the AIP, or the IEEE Computer Society.

Circulation: *Computing in Science & Engineering* (ISSN 1521-9615) is published bimonthly by the AIP and the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314, phone +1 714 821 8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903; AIP Circulation and Fulfillment Department, 1NO1, 2 Huntington Quadrangle, Melville, NY 11747-4502. Annual subscription rates for 2003: $42 for Computer Society members (print only) and $48 for AIP society members (print plus online). For more information on other subscription prices, see http://computer.org/subscribe or http://ojps.aip.org/cise/subscrib.html. Back issues cost $10 for members, $20 for nonmembers.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *Computing in Science & Engineering*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 0605298. Printed in the USA.