

# Teaching Computation to Undergraduate Scientists

Daniel T. Kaplan  
Department of Mathematics and Computer Science  
Macalester College  
St. Paul, Minnesota  
kaplan@macalester.edu

## ABSTRACT

This paper describes the motivation and design of an introductory computational course for natural, physical, and social scientists.

**Categories and Subject Descriptors:** K.3.2 Computers and Education: Computer and Information Science Education — *Curriculum*

**General Terms:** Algorithms, Languages

**Keywords:** Scientific programming, databases, education, image processing, signal processing

## 1. INTRODUCTION

How should computation be included in the education of an undergraduate science student? The need is there. Advanced scientific computing is now a feature of almost every field of science and the large majority of practicing scientists use computation in their technical work, whether for solving partial differential equations or conducting homology searches. The manner in which established scientists have learned about computing is typically haphazard: they pick up a package that colleagues use or learn programming on their own; they rely on friends in their laboratory who are adept with computers; they hire computer technicians to provide programming services.

Surprisingly, only rarely do science students learn about computing via a formal course. Consider, for example, the curricula at liberal arts colleges, whose graduates get science Ph.D.s at a high rate compared to other undergraduate institutions.[1] Liberal arts colleges have traditionally emphasized interdisciplinarity. Requirements for a science major often contain almost as many science courses outside the department as inside it: calculus at several levels, statistics, chemistry, physics. But computation is rarely on the course list.

To quantify the extent to which undergraduate science training includes computation, I examined the graduation requirements for several majors at national liberal arts col-

**Table 1: Major Requirements for Computational Courses.**

Department	Required	Allowed	No Mention
Biology	0	0	10
Chemistry	0	1	9
Economics	0	0	10
Mathematics	1	3	6
Physics	0	4	6

leges. The sample I used was the top ten such colleges according to the US News rankings[2]: Amherst, Bowdoin, Carleton, Davidson, Haverford, Middlebury, Swarthmore, Pomona, Wellesley, and Williams. At each school, I tallied the requirements and recommendations for the major in Chemistry, Economics, Mathematics, and Physics. Each set of requirements for a major was placed into one of three categories:

1. Requires a course in computation. This might be either an introductory programming course or a higher-level course that explicitly requires programming skills.
2. Allows an introductory computation course to fulfill a requirement for the major. Typically there is a list of courses from varied departments of which the student must select one or more as part of the major.
3. Makes no reference to a computation course or computational skills.

The results are given in Table 1. The numbers show how many out of the sampled 10 liberal arts colleges require, allow, or fail to mention a computational course for their major.

Why this seeming indifference to computation? One possibility is that science educators believe that students can pick up sufficient computing skills without formal study. In my years working with economists, physicists, physiologists, physicians, biologists, and geneticists, I find that most scientists interpret difficulty working with computers as a sign of the intrinsic difficulty of the problem rather than a symptom of inadequate training. This is so even when the problem at hand is comparatively simple, say, reading an ASCII file containing a matrix of numbers.

Another possibility is the traditional resistance to change of undergraduate education. For example, since no programming skills are required of students, courses in the major

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGSCE '04 March 3–7, 2004, Norfolk, Virginia, USA  
Copyright 2004 ACM 1-58113-798-2/04/0003 ...\$5.00.

cannot include programming components. Without such components, there is no need for students to have computer skills, hence no requirement.

It must be pointed out that computation is already integrated, to some extent, into mainstream courses for the major. For example, it's common for calculus courses to use software such as Mathematica or Maple or for linear algebra courses to use Matlab. Statistics courses nowadays use a variety of packages, ranging from Excel to S-Plus. However, while students may successfully use software to integrate a mathematical function or conduct a t-test, they learn few if any basic concepts of computing. Students who have completed such courses typically do not have even the simple notion of the application of a function to arguments, or values returned by the function, or of assignment to variables.

Yet another reason for the lack of computational requirements in undergraduate science programs is that the introductory course offerings in computer science are typically of no visible relevance to a natural, physical, or social scientist. Consider the following descriptions of courses listed as required or recommended for a non-computer science major from some of the liberal arts colleges included in the above table:

*The course will introduce many important abstract data types (ADTs) such as stacks, queues, and trees, as well as the object-oriented programming (OOP) paradigm. It will then focus on various data structures used to implement these ADTs, contrasting the competing structures in terms of their space and time efficiency and other considerations that vary from application to application. — or*

*Introduction to computer science for students planning to major in computer science or a related field. Topics include iteration and recursion, arrays, linked lists, sorting and searching, binary search trees, elementary analysis of algorithms, and a thorough introduction to object-oriented programming in Java. — or*

*An introduction to object-oriented programming and the Java programming language. Topics include the use of classes to support encapsulation of object behavior and state, class inheritance, and polymorphism.*

Such courses may provide an excellent introduction to the programming aspects of computer science, but they do not relate clearly to the needs of natural, physical or social scientists. Neither do they appear to have relevance through the applications or examples listed: “networks and robotics,” “artificial intelligence,” “comparison and evaluation of programming languages.” One course description includes a scientifically promising but vague “Non-numerical and numerical applications.”

The usual follow-up course in data structures similarly offers no apparent relevance to a practicing scientist. For example:

*Abstract data types and efficient data structures, including priority queues, dynamic dictionaries, and disjoint sets. Analysis of data structures, including worst-case, average-case, and amortized*

*analysis. Storage reclamation. Extensive practice in implementing these data structures.*

It would be extraordinary for a chemist or economist or physicist not to have a basic familiarity with the contents of a required calculus course, but it is commonplace for them to have no idea of the meaning of words used in the description of introductory computer science courses. If science faculty cannot see the relevance of the course contents, they will have little reason to require or even encourage students to enroll in such courses.

## 2. HOW SCIENTISTS DIFFER

In developing a computation course that might reasonably be adopted as a requirement for an undergraduate science major, we need an objective that is of clear relevance to scientists: to give students an adequate background in computation so that they can work effectively with teachers and researchers in their area of interest and to be able to determine what areas of mathematics and computer science will contribute usefully to their own future development.

It is important to keep in mind the ways that the needs of scientists differ from computer scientists:

- Scientists work with entities such as signals, images, systems of equations, tables of data, etc. To be directly useful, a scientific computation course must show students how to represent and operate on such entities. Structures such as priority queues and B-trees are of no interest.
- Computation is not their primary interest or career goal. They think of computers as a tool rather than an object of intrinsic interest. It is comparatively easy to alienate such students. Students already have a good exposure to computers and perform sophisticated computational operations in image processing, music composition, and so on. When a student has used his or her home computer easily to add graphical effects to web pages, but learns with difficulty in a college computer science course how to print out the numbers from 1 to 10, the student may conclude that the college course is irrelevant, out of date, and provides skills of little practical use.
- The students come from diverse backgrounds and are heading in diverse directions: physics, chemistry, economics, mathematics, neuroscience, biology, and even fine arts. This makes it difficult to motivate examples used in a course by future courses or future work; the examples have to be accessible at an intuitive level by different kinds of students.
- Almost all of a science student's mentoring will come from people who do not have a strong background or interest in computing per se. These people will not help them refine their CS skills.
- Scientists have very limited time to devote to the formal study of computation. A science student taking one required course will have roughly 40 hours of class time and 100 hours of out-of-class work to study the topic. In contrast, a computer science student will spend about 10-fold as much time in formal study.
- It is impossible to know in advance what software science students are likely to need in their scientific work.

There is a huge variety of scientific packages in use. In contrast, for a computer science student, it is reasonable to assume that they will make extensive use of a small and very well defined set of languages such as Java and C++.

- Scientists use graphics extensively, even at an introductory level. Most of these are in standardized formats: line plots, histograms, scatter plots, and contour and perspective plots.
- Scientists generally do not write packages, but use them. They produce software for their own specialized use, rather than for others to use.
- Scientists do not generally design binary file layouts, but use established formats such as spreadsheets and image and sound files. They have a strong need to know how to organize and access data stored in tables and databases. The relevant concepts are rooted in computer science but wholly absent from introductory courses. One of the liberal arts course descriptions does start with the scientifically enticing statement, “A fundamental problem in computer science is that of organizing data so that it can be used effectively.” But the data structures used in the course are the computer scientist’s “lists, stacks, queues, trees, sets and graphs,” rather than the selection, projection, and join operations of databases.

### 3. COMPONENTS OF A COURSE

Keeping in mind the ways in which science students differ from computer science students helps to inform the ways we answer basic questions about the structure of a course in computation.

#### 3.1 What aspects of programming?

Practicing scientists tend to use application packages rather than programming systems. This might suggest — and does, to many scientists — that it is not necessary to learn general programming skills. I believe this view is seriously flawed. The most important computational skill for a scientist is learning to use a new package. The packages they use will continue to change every few years and knowledge of any specific package quickly becomes outdated. We want to free students from the tyranny of the mouse, the degeneration of computer understanding into hand-movement skills and sequences of gestures.

A knowledge of general programming concepts greatly facilitates the use of packages. Basic concepts are some of the most useful: the difference between a number and a character string; how strings (and case) are represented; indexing; the idea of a function, of arguments, and of applying a function to arguments to produce a return value; assignment and change of state; conditionals and iteration. It is less clear how the concepts of object-oriented programming, which tend to be emphasized in modern introductory CS courses, enhance a scientist’s ability to learn a new package.

Increasingly, scientific packages include scripting languages or even full-fledged programming languages (e.g., Matlab, Mathematica, Maple). The acquisition of general programming skills can help directly in the use of such a package.

A secondary benefit of teaching a programming language is that it prepares those students who are interested for future work in computer science.

#### 3.2 Therefore, What Language?

Some simple criteria can direct us to an appropriate language for teaching general programming skills:

- The language must be simple to learn, because we have only about 40 hours of classroom time and we want to spend a good fraction of this working with data structures, operators, and applications of scientific interest.
- The language must make clear the general programming concepts we want to teach. For example, one of the most important definitions of a computation is “a transformation from inputs to outputs.” Consider one of the most famous programs of all time:[3]

```
#include <stdio.h>
main(){
    printf("hello, world\n");
}
```

What is the input, and what is the output? The point here isn’t to claim that C doesn’t support explicit identification of inputs and outputs; obviously it does. But there is so much overhead to producing a program — compilation, linking, running, specifying arguments on the command line, needing to read files to handle most inputs — that the idea of a transformation from input to output is lost. Things get even more complicated with JAVA. For instance, from [4] we have an “introductory” example:

```
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

The very notion of a function is lost here, buried in more complicated ideas of objects, classes, and methods.

- The language must offer basic operators relevant to scientists. One of the most common tasks for a scientist is to make a scientific graph, for example a scatter plot or the plot of a function over a specified interval. Languages such as C or Java do not have standard operators for producing such graphs. What’s needed is a package where the programming language is integrated with graphics; where a student can learn to make a plot after one or two classroom sessions.
- The language must be general enough that topics of importance in computer science can be illustrated. For example, the language should have a natural, simple way to represent a tree and should support recursion. It should enable functions to be passed as arguments and returned as values.

Fortunately, several packages exist that satisfy these criteria, among them Mathematica, Maple, Matlab, and R.

It’s desirable that the language be interpreted — or, at least, that students be able to avoid dealing with compilation and linking. An interactive debugger is also very helpful to students. For these reasons, the course I have developed at Macalester College is based on Matlab.[5]

Some educators argue that it is important to expose students to a variety of languages. There is an excellent intermediate text that directs students to computational tools

from multiple sources.[6] However, given the time available in a single introductory course, I think the priority must be to establish useful programming skills in a single language.

In my course, for example, rather than covering symbolic computation by introducing a system such as Mathematica or Maple, I focus on how one can represent a function in symbolic form in the language that students have already learned. We then construct operators on this symbolic form to carry out symbolic differentiation. Those students who need to carry out production-level symbolic computations will be well positioned to learn one of the appropriate systems.

### 3.3 What Applications and Examples?

Programming applications and examples have to be chosen carefully, to demonstrate to the student the relevance of the computational topics and to guide the students to techniques appropriate for their future scientific work. The applications also need to be accessible and motivational to a general scientific audience. They have to display a level of sophistication that matches the capabilities the students are familiar with from their home computer.

The following are some of the examples that I use, typically in the last half of the course.

#### 3.3.1 Sounds

Music and speech sounds are, obviously, familiar to almost all students. They are used to transmitting, storing, and playing sound files. However, they generally have much less experience recording sounds on the computer and have little notion about basic principles of sounds and signals: amplitude and frequency. There are simple programming exercises that emphasize these basic structures of sound and allow students to implement reasonably sophisticated effects. Some of the sound-based examples used in my course[5] are:

- Music synthesis. Given as inputs a musical score (in spreadsheet format) and a recording of a single note from a real instrument, students synthesize other notes and put them together into a complete musical piece.
- Noise reduction. Using recordings of speech from old radio broadcasts, students implement simple noise reduction techniques. These do not require any advanced knowledge of signal processing. For example, truncating to zero any samples of low amplitude will effectively reduce noise between words.
- Speed change. With a simple stuttering technique, students can alter the speed of playback of a sound without changing its pitch or tone quality.

#### 3.3.2 Images

Students have ready access to digital images through digital cameras and scanners, but they often have little understanding of the basic principles of representing an image, for example the use of three color planes. Even simple operations such as cropping can be motivating to students. Some examples used in my course [5]:

- Color adjustment. Given an image of a painting with faded colors, and some information about the original colors, restore the painting toward its original form.

- Image segmentation. Using color information on a pixel-by-pixel basis, divide an image into different regions. In one application, students create a false-color land-use map based on Landsat images of the local region.
- Edge detection. A simple differentiation operator can be used to detect edges in an image.

#### 3.3.3 Mathematical Relationships

Students' capabilities with mathematical relationships are almost always limited to symbolic manipulation of equations to find solutions. For example, they know how to solve  $ax^2 + bx + c = 0$  for  $x$  but have no way to extend their knowledge to functions of other forms. A computational course can make a contribution to a student's mathematical understanding by showing how to represent a mathematical relationship numerically and introducing the various operations that can be performed on that representation. Most students, for example, are surprised to learn that they can easily write a computer program that approximates the solution of a mathematical function of just about any form. Other, easily understood operations are differentiation, interpolation and optimization. The emphasis does not need to be on the mathematical concepts (e.g., the theory of a cubic spline), but on how the inputs and outputs are to be represented and manipulated (e.g, the tabulated points of the function to be interpolated).

Students vary widely in their mathematical capabilities, so I have found it useful to divide the subject into two levels: mathematical relationships with one unknown versus relationships with several unknowns. Simple graph-reading skills are all that is needed to understand the operations and algorithms on relationships with one unknown. Students who have stronger mathematical backgrounds — mathematics, physics, and engineering majors, for example — have the ability to understand the numerical approaches to relationships with several unknowns.

#### 3.3.4 Data

To a computer scientist, the word "data" may suggest queues and stacks, trees and graphs. But to a natural, physical, or social scientist, "data" brings to mind a lab notebook or the tabulated results of an experiment or survey. The basic organizing concept of scientific data is that of the table, which most scientists now access using spreadsheet software. Despite the sophistication of such software, the basic concepts of rows and columns, cases and variables<sup>1</sup> are often not explicitly present in the user's mind. Operations of selection and projection are intuitive, but most users lack the appropriate vocabulary.

Computer scientists obviously have important things to say about sorting and searching. But practicing scientists treat these operations as self-evident and care little about the algorithms (so long as they are efficient).

Computer scientists can have a tremendously positive impact on other scientists by emphasizing an important area that is systematically misunderstood and ignored by science: relational databases. Even though the fundamental operations on a single table are intuitive, very few working scientists are aware that databases generally consist of multiple

---

<sup>1</sup>Or, in database notation: tuples and fields.

tables and that database queries involve many operations on pairs of tables.

### 3.3.5 Computer Science

Computer Science is a science too, and it's appropriate to include examples from it. In my course I focus on recursion and trees. In addition to the standard examples of factorials, Fibonacci numbers, and the Towers of Hanoi, I try to work in topics of engineering or scientific interest: optimal matching, and clustering of data using dendrograms. For a moderately mathematical audience I also cover the basics of symbolic differentiation of mathematical functions; this not only shows the power of recursion but raises important issues of how to represent a function symbolically. Algorithms drawn from bioinformatics — e.g., string alignment, partial pattern matching, and comparison — are also potentially very useful.

### 3.3.6 User Interfaces

User interfaces can be very complicated to build, and introductory computer science courses understandably emphasize ways to deal with the complexity imposed by interfaces. The time constraints in an introductory course make it difficult to cover both scientifically relevant topics and user interfaces. On the other hand, almost all students (and their science faculty mentors) demand that they learn some “modern” computing: GUIs rather than text interfaces.

As a compromise, I spend one week on GUIs. The programming concepts emphasized are not objects but scope and ways to communicate between environments. I try to focus on GUI examples that emphasize acquiring information that cannot practically be gotten via a text interface: e.g., identifying points on an image or recording the timing of fast paced events.

## 4. PACE OF THE COURSE

I spend roughly half of my course (6-7 weeks) introducing Matlab programming: basic data types, applying functions to arguments, indexing, file operators for reading standard file types (e.g., text files, spreadsheets), constructing functions, conditionals and loops. The remainder of the course (7-8 weeks) covers examples and applications drawn from a wide range of scientific disciplines. These serve both to introduce ideas of representing real world and theoretical entities (images, sounds, mathematical relationships) and to provide reinforcement of the programming skills.

There is some emphasis on how to design software effectively. This does not refer to object-oriented programming but classical ideas such as functions and scope, and decomposing tasks functionally. Students must complete an individual or small group (2 person) term project. I also assign a team project. The team of 10-15 students takes a broad task, such as identifying which language a text is written in or identifying the author of unattributed text. The team then splits into groups to implement and document individual components of the task while maintaining enough contact with the other groups to integrate the components into a functioning whole.

## 5. ASSESSMENT

The course I describe has been offered at Macalester (*Scientific Programming*, CS 121) in roughly its present form for

3 years. The alternative introductory computation course, designed primarily for prospective computer science majors, called CS I, is an Abelson-and-Sussman style Scheme-based course. The follow-up to this is CS II, a Java-based course.

From the very beginning, Scientific Programming has been effective in attracting students from various disciplines including chemistry, economics, math, physics, biology, and neuroscience. A small number of students from the fine arts and humanities take the course as well. Both the chemistry and the physics departments specifically recommend the course; the course also fulfills a computing requirement of the math major. Before the course was introduced 3 years ago, only the math department required or allowed an introductory computer science course.

Initially, the Scientific Programming attracted almost entirely juniors and seniors. Some of these were students seeking to fulfill a computational requirement in mathematics or engineering, or a perceived requirement for admission to graduate school. Only 5 to 10% of the students were sophomores and there were very few freshman. There has been a dramatic growth in underclass enrollments: one third of the students are now sophomores and 15% are freshman. CS 121 now has the largest enrollment of all introductory computer science classes; this is due to a small but steady growth in enrollments for CS 121 and a dramatic decline — seen nationwide — in the enrollments for the introductory computer science courses oriented toward majors.

The course has helped to recruit students to computer science. Several computer science majors and minors had this course as their first contact with computer science. The computer science department accepts the scientific computation course as fulfilling the pre-requisites for the second-level computer science course.

The availability of a course such as CS 121 allows us to set a meaningful pre-requisite for the junior-level “numerical methods” mathematics course (now called “scientific computation”). Whereas previously that course had a theoretical orientation suited to mathematics students with little computational background, now the course (taught using [6]) is oriented around projects and applications. Enrollments in the course have more than doubled and it attracts physics and economics students.

Overall, the course appears to be fulfilling the goal of providing a formal introduction to computation in a manner that is attractive and useful to undergraduate scientists.

## 6. REFERENCES

- [1] T.R. Cech, “Science at liberal arts colleges: a better education?” in S. Koblik and S.R. Graubard, eds, *Distinctively American: the Residential Liberal Arts Colleges*, Transaction Publishers (2003)
- [2] US News and World Report. See [www.usnews.com/usnews/edu/college/corank.htm](http://www.usnews.com/usnews/edu/college/corank.htm)
- [3] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, (1988), p. 6
- [4] M Campione, K. Walrath, A. Huml, *The Java Tutorial: A Short Course on the Basics.*, 3rd ed., Addison Wesley, (2001), p. 32
- [5] D.T. Kaplan, *An Introduction to Scientific Programming and Computation*, Brooks/Cole (2004)
- [6] M.T. Heath, *Scientific Computing: An Introductory Survey*, McGraw Hill (2002)