What programming languages should we teach our undergraduates?

JOHN SCALES and HANS ECKE, Colorado School of Mines, Golden, Colorado, U.S.

Computer science is no more about computers than astronomy is about telescopes.

—E. W. DIJKSTRA

We begin by investigating how we teach programming languages now (and what's wrong with it). Consider the following two situations:

> Student X has 200 data files nnn.dat with nnn ranging from 1 to 200. X wants to transform them to plain text (ASCII) with a program dat2asc which should be called thus for each file: dat2asc nnn.dat nnn.asc. X doesn't know how to accomplish that! Two solutions present themselves: generate an Excel spreadsheet with the correct commands—200 nearly identical lines—and write them into a batch file, or type the commands 200 times.
>
> Student Y inherited a program, written in C, from student Z to perform a certain computation. The code is clean and well structured. Still, after listening to Y talk, it becomes clear that Y thinks the program is an impenetrable black box. Y understands the theory behind the calculation but goes to great pains to avoid having to change or extend the implementation.

Our point is that students need to learn how to read programs in much the same way they need to learn how to read scientific papers. They need to know how to get the gist of an argument, what to focus on, and what to ignore.

Frequently students are unable to test and play with ideas interactively by coding. They thus resort to impractical implementations. Here is an example from Joe Dellinger of BP: "I remember seeing a student give a talk about a (large-scale processing) program he had written in Matlab that he was quite proud of. Most of the talk, and I gathered most of his time, was not on the theory or science behind his program but on all the clever things he did to get it to run in any reasonable time. I estimated that his code was approximately 20 000 times less efficient than ours."

This phenomenon is becoming increasingly widespread as students gravitate to integrated packages that seem to free them from learning how to write programs. We have no one to blame but ourselves. We don't insist that our students learn more about the art and science of computing, but we force them to deal with languages that are cumbersome and error-prone. As stated by Kurt Marfurt (formerly a researcher at Amoco and now at the University of Houston): "I no longer believe in APL, Algol, FPS assembler language, Star 100 assembly language, Cray assembly language, and CM2 assembly language as the languages of the future. I learned all these and now I wish I had learned Aramaic instead. At least I could learn how to save my soul."

Our curriculum is still based on the assumption that we should use programming languages that result in the fastest execution. However, on today's fast hardware, the main problem is often development time, not execution time.

At Mines, in trying to resolve this problem, we require a one-semester crash course in C++. But we can't expect students to learn computer languages if faculty do not integrate them into course work. This is a problem because most professors don't have time to learn a huge language like C++. This means a replacement language must be found that is easy to learn and powerful enough to be useful.

At the end of our crash course, few students could write a functioning C++ program without a great deal of pain and suffering. And it's not clear that languages such as C++ are the best choice. The majority of bugs in C++ and C are associated with memory management and pointers, so why not eliminate memory management and pointers in the first place?

As we will try to show in this article, it is possible to switch to more modern languages with nearly no loss in performance if the right strategy is used. The increase in the ease of development is so remarkable that it can be called a quantum leap.

The computational tool chest. The computer tool chest of experienced scientists and engineers is varied, but there are four types of tools that the authors use all the time:

1) a high-level code development/editing environment (e.g., emacs, Visual Studio)
2) an integrated numerical/graphical environment for quick calculations and prototyping (e.g., scilab, Matlab)
3) a scripting language for manipulating files and gluing programs together (e.g., Perl, Python, or emacs/Lisp)
4) a high-level programming language (e.g., C++, Java, Fortran 90) for large-scale applications

We think students should also have the advantage of such a rich environment. Our department does a pretty good job of teaching and exploiting tools such as Matlab. We introduce Matlab within several geophysics courses and many exercises are built around it, but this is not enough. As we have seen, our efforts to encourage programming in C++ have been less than successful. Further, our students don't learn a scripting language, which would simplify most routine programming. We don't teach anything like emacs, which would give students a powerful environment for editing, compiling and debugging programs, typesetting documents, and interacting with interpreters. As a result, students

What MIT does

MIT's entry-level computing (course) emphasizes controlling the complexity of software systems through general techniques common to all engineering design: building abstractions to hide details and to separate specification from implementation, establishing conventional interfaces to allow the creation of standard modules, and shifting modes of linguistic description ... its focus is not on particular programming-language constructs, data structures, or algorithms—these are regarded as details.

—HAROLD ABELSON, MIT

gravitate to simple cut-and-paste editors and never learn about regular expressions or other higher-level text manipulation techniques inside a more sophisticated environment.

This situation has forced the department to re-examine what we want our students to know about computers and how we're going to teach them. In this article, we consider what programming languages we should teach geophysics undergraduates. A host of important issues related to computer science and numerical analysis are outside the scope of this discussion. Here we focus only on programming languages.

The only thing absolutely clear is that there is no absolutely clear choice of "best" starting language for students. We believe that the majority of our students will eventually need to learn significant parts of several languages, including Java, C++, Perl, and perhaps Fortran, not to mention the ancillary tools that go hand-in-hand with programming (editors, debuggers, plotting tools, make, etc.).

But as Abelson says (see box on previous page): The focus should not be on programming-language constructs, data structures or algorithms—these are regarded as details. Instead, students are brought to appreciate a diversity of ways to look at programming problems and solve them. Consider this example. Word and Latex both allow you to format a document. But Word is a WYSIWYG (what you see is what you get) text processing system and Latex is a typesetting language. With Word you do the typesetting. With Latex, typesetting is done by the author of the language, an expert on typesetting. So these are two different paradigms for formatting text.

Major programming paradigms include data abstraction, rule-based object-oriented programming, functional programming, logic programming, and constructing embedded interpreters. We think it important that students understand these paradigms and know when to use them.

It is wrong to assume that a university education should simply impart facts. An education should teach ways to think—to organize thought, detect mistakes in reasoning, and analyze assumptions. Programming computers is an important way to acquaint oneself with the scientific way of thinking. If we fail to teach programming, we make it harder for our students to succeed in science.

Fast and easy—interpreted languages. In the course that Harold Abelson and Gerald Sussman developed at MIT, taken by half to two-thirds of all MIT undergraduates, the language used is Scheme, a dialect of Lisp. Scheme is an example of a modern scripting language that supports a variety of programming paradigms.

In the beginning, scripting languages were intended largely as simple tools for manipulating files and running programs. These languages are interpreted—meaning that when you type code into the interpreter, you get immediate feedback as to its correctness. (An interpreter for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.) We use *interpreted language* as being equivalent to *scripting language*.

The very speed and ease of code development in an interpreted environment can fundamentally change the way we think of programming. In his book *ANSI Common Lisp*, Paul Graham makes an intriguing analogy between these changes in programming with changes in painting that resulted from the use of oil-based paints: "Strange as it sounds, less planning can mean better design. The history of technology is full of parallel cases. A similar change took place in painting during the 15th century. Before oil paint became popular, painters used a medium called *tempera* that cannot be blended or over-painted. The cost of mistakes was high, and this tended to make painters conservative. Then came oil paint, and with it a great change in style. Oil "allows for second thoughts." This proved a decisive advantage in dealing with difficult subjects like the human figure ... The medium did not just make painters' lives easier. It made possible a new and more ambitious kind of painting."

In fact, it is possible to perform nearly any programming task with interpreted languages, apart from large-scale numerical calculations. Our own experience is that nearly always we attempt a first solution in an interpreted language and then either rewrite the application in C++ when it gets too big, or rewrite the compute-intensive kernels in C++ and link them into the interpreted language.

The ease of learning such languages, and their widespread utility, make interpreted languages extremely appealing as first languages for novice programmers.

What are we trying to accomplish? But we're putting the cart before the horse unless we first address the fundamental question: What we are trying to accomplish through computer programming education? Our answer includes four main objectives.

1) The language should be useful. For immediate utility the language needs to be integrated into course work and useful in solving problems important to students (e.g., homework). Thus, first languages should have strong built-in graphics and numerical libraries.

For long-term utility, knowledge of a programming language should help students find a job. The languages required for jobs changes with time and varies with industry. Students should be able to pick up new languages quickly and still write in good style.

Of the interpreted languages, Lisp-Stat has excellent built-in graphics and numerical libraries and is immediately useful. Python also has many built-in and contributed libraries for numerical and scientific computing.

Languages currently in demand with a variety of employers include C, C++, Java, and Perl. The demand for Fortran in the geophysics community appears limited primarily to large-scale numerical calculations.

Of course, many other special-purpose systems such as ProMAX, MathCAD, and Excel are useful in getting jobs, but these are beyond the scope of our discussion and don't address our objectives.

2) Programming computers should be fun. Admittedly it is often more difficult to communicate with a computer than a person; computers require a precision of expression that humans do not. But writing computer programs should not be drudgery—a seemingly endless cycle of design, code, debug.

A big part of the problem of computer illiteracy is attitude: If you believe that computers are strange things that only the initiated understand, you won't even try to master them. A language that helps change that attitude will allow you to solve problems even with a small subset of the syntax. It will also allow you to immediately tackle problems of everyday life: writing a small GUI, solving a homework problem, manipulating text files or processes, etc.

This is a big reason we prefer interpreted languages such as Perl, Python, and various flavors of Lisp (Scheme, Lisp-Stat, etc.) for a first programming experience. These languages encourage rapid, interactive code development. This turnaround speed is partly due to certain features of these languages (e.g., garbage collection and run-time typing) that free programmers from much nitty gritty associated with C and Fortran.

The speed is also in part due to the interpreted nature of the language—when you type code into the interpreter, you get immediate feedback as to its correctness. And the speed is partly the result of certain features of functional languages that help isolate the effects of bugs (e.g., using local variables only for definitions and not assignments, whenever possible).

Languages with a high fun quotient include Java, Python, Perl, and Lisp-Stat.

3) The language should teach good style. Programming in good style is the subject of many excellent books by programmers far better than we can ever hope to be. (See "Suggested reading.") It is more art than exact science. However, some elements can be singled out. Good style is mastery of all major programming paradigms and knowledge of when to use them: procedural, object-oriented (OO), generic and functional programming (see box on OO Programming.) We cannot hope to teach each in depth. But, knowledge of OOP is definitively necessary.

Ultimately, good programming style comes only with experience. But we can reduce the time required to develop good style by choosing a good teaching language. Among the easiest, cleanest languages that meet this criterion are Java, Python, Scheme, Lisp-Stat, and Eiffel.

4) Students should be able to take their computing environment with them. We believe strongly that the language should be available and used on all major platforms (including Macintosh, Windows, and Unix) under a free software license. CSM has a mixture of Linux and Windows machines, and many students have Macs at home, so it is essential that applications run on all these platforms. Further, we want students to be able to take their computing environment with them when they leave school. By focusing on free tools, students can feel that their investment in learning a tool or language will not be lost because they or their future employer doesn't have a particular software license.

The lack of an implementation distributed under a free software license currently rules out Fortran 90, C# and Visual Basic.

Language overview. Here is a summary of our impressions of some popular modern languages.

Perl is powerful, easy, fun to learn, but it is not a good choice for a first language because it doesn't discourage bad style. On the other hand, it contains strong OO and functional elements.

Python is less powerful than Perl but has a cleaner syntax that enforces good style. It also has a powerful interface to Java and a huge number of built-in and contributed modules that make it attractive for scientific programming.

Lisp-based languages (such as Scheme or Lisp-Stat) make excellent teaching tools. They have lots of functions, but you can write nice code with only a small subset of these. As you learn more, your code becomes progressively simpler. Lisp-Stat is certainly one of the easiest OO languages to learn. Our experience is that programming novices can be writing clean, interesting OO code with Lisp-Stat in less than a semester. Lisp-Stat also has an extensive library of numerical and statistical tools.

C++, a superset of C, is a vast and complicated language. Students need several semesters to assimilate the full details. In fact, knowledge of C++ presupposes a working knowledge of C. In practice, subsets of both C and C++ are used to teach programming. We will use the term C+- to denote the minimal indispensable core of C++.

Of the C family, only C+- should be considered as a teaching language for beginners. C frustrates users unfamiliar with it because it leads easily to syntax errors, is difficult to debug, and is prone to subtle bugs. C++ is far too big and

A program should be light and agile, its subroutines connected like a string of pearls. The spirit and intent of the program should be retained throughout. There should be neither too little nor too much. Neither needless loops nor useless variables; neither lack of structure nor overwhelming rigidity.

A program should follow the "Law of Least Astonishment." What is this law? It is simply that the program should always respond to users in the way that least astonishes them.

A program, no matter how complex, should act as a single unit. The program should be directed by the logic within rather than by outward appearances.

—The Tao of Programming

Good style

Good style requires exercising the ability to write extensible, understandable, portable, modular, robust, and friendly code.

- Extensible code is easily changed to support wider or changed objectives.
- Understandable code can be read, and still make sense, even years after the writing or by somebody other than the original developer.
- Portable code is easily transferred to another computer system that may run a different operating system.
- Modular code is split into logical pieces that can be reused for different projects.
- Robust code anticipates error conditions and either recovers or dies gracefully, with an indication of the exact reason for the error.
- Friendly code knows the demands of the user and goes out of its way to fulfill them in an unsurprising way. Meaningful defaults are a good example.

None of these attributes of good code will usually appear in isolation.

Object oriented programming

Objects are pieces of software that respond to messages. We are used to static data abstractions—seismic traces for instance, with associated headers and samples. An object is really a dynamic data abstraction. Suppose the seismic trace could respond to messages (such as plot yourself, or describe yourself, or resample yourself, etc.). Then the data abstraction becomes a much more interesting thing, with the ability to perform tasks. While it's possible to write OO programs in almost any language, some languages have special features that make such programming easy—allowing one to write programs that are readily extensible, less error prone, and more understandable than the conventional procedural approach.

complex to learn in class. Teaching C+-, on the other hand, means that the students may use C++ features that they don't really understand, for example when using a vector<float> class but having no idea how to write it themselves.

On the whole, we think that Java offers the best compromise presently available. It's widely used, clean, easy to learn, and operates across platforms. Java was designed for objects, whereas C++ is C with objects grafted onto it. A recent study by Evans Data shows that more than half of North American software developers use Java (and that number is expected to rise in the near future).

Neither of us currently knows Java very well, though. (JS uses Lisp as a scripting language, Lisp-Stat and scilab as a computing environment, and C/C++ for numerical/data acquisition work. HE uses a mixture of Perl, C++, and Matlab and is especially happy with Perl.)

The difficulty in finding a good compromise reflects the diversity of programming languages: Each has its own problems and jobs for which it is suited ideally and a large set it can solve only with difficulty.

Geophysics programming survey. In order to get a broader perspective on these issues, we surveyed several dozen geophysicists and applied mathematicians working in exploration, software development, processing, and academia concerning the question in the title. Although there is widespread recognition that much Fortran legacy code exists in the geophysical industry, there was equally wide recognition of the need to teach modern programming principles. Here are some responses.

Bill Harlan, Landmark: If I had to pick one language, I'd recommend Java. Second would be C with some exposure to C++. But if students know Java, they can pick up C/C++ on their own.

If I were teaching programming to students that had never programmed, I'd use Python. You won't waste time explaining tricky syntax. Python supports both procedural and object-oriented approaches.

Do not require Fortran. I have not written Fortran on the job for 10 years. If faculty do not consider programming languages worth learning, then students will probably follow their example.

I now consider formal computer science courses as essential training for any scientists who expect to program in their careers.

Students need courses in style, data structures, algorithms, encapsulation, object-oriented programming, functional/generic programming, and software patterns. Students need concepts more than the mechanics of any single language.

Teaching C++ is tricky because you must identify a safe minimal subset. The best way to improve one's style in C++ is to write more Java.

Java is so easy to debug that I almost never use a debugger. Memory management is the source of most bugs in C/C++. Java prevents you from writing outside the bounds of an array, prevents you from casting a reference to the wrong type, prevents mischief with pointer arithmetic, prevents you from reading uninitialized variables, prevents you from allocating too little space for an object, prevents you from leaking unfreed memory, prevents bad assumptions about the size and byte order of primitive types, and so on. Those are the kind of problems that require a debugger and cost you days.

Performance should be for advanced students, not novices. Two thirds of our new processing system is in Java.

Jon Claerbout, Stanford University: Undergraduates definitely should not use the two languages that defined my career, Fortran 77 and C. I suggest Matlab and Java. I also like the idea of starting students out on Python.

Graduate students are another matter. We use almost all Fortran 90. It's the gold standard for numeric intensive computation. I also like it because, like Matlab (and unlike Java), it looks more like mathematics.

Joe Dellinger, BP: Fortran 77 is still king when it comes to real hard and dirty large-scale number crunching. C or C++ is used for interactive graphical applications, and for I/O packages. Knowledge of HTML is useful for Web applications. Java looks to become more and more useful, but not for anything requiring heavy lifting. Some knowledge of scripting languages such as sh, csh, awk, Perl, or Python certainly comes in handy. At least minimal knowledge of make is also handy. However, students should learn C++ or Java or some modern language while in school.

Wences Gouveia, ExxonMobil: Students should be exposed to structured and procedural and, later, to OO programming. They should know C well, followed by C++. I would teach Matlab in a special, elective class.

Fernando Moraes, Universidade Estadual do Norte Fluminense: Although we may agree on appropriate computer languages for R&D, another important issue is related to how many students will actually write software after graduating. Depending on the objectives of the undergraduate program, it can be very difficult to introduce programming classes into the main curriculum. In my opinion, C provides the best platform for developing programming skills while keeping complexity relatively low. An introduction to C and Unix tools with the basics of numerical computations could be given in one course. Further developments could come from special courses and projects for interested students.

Matthias Schwab, The Boston Consulting Company: I would teach:

1) Java. Fun because it is cleanly defined, leads beginners to a clean object-oriented programming style, empowers its programmers by covering a huge variety of programming problems (graphics, I/O, interactive Web pages, data structures) on a fairly high level, and is portable.
2) Perl. Fun because it is quick and dirty (especially for hackers, less fun for purists), empowers its programmers, and is portable.
3) Fortran 77 or C. Neither is particularly fun but they are workhorses. In a perfect world, I would use them to accelerate my number-crunching code after I had programmed it in Java.
4) make. Not much fun to program but fun to use (reproducibility!). Serious programmers will not be able to avoid make.

Bill Symes, Rice University: I made my choices manifest in a scientific programming course at Rice—C, C++, Fortran 77, make, shell programming, debugging, graphics, documentation tools, hints at other things.

OO should be in the curriculum because students will encounter it in the "real" world if they do any programming. Practically speaking this means C++ for the moment.

Nevertheless it seems plausible that eventually Java execution could incorporate optimizations utterly out of reach for statically compiled languages. My sense is that some interesting demonstrations have occurred, but that this sort of thing is not yet available in any routine way and won't be for a few years. But "in a few years" is what we should be thinking of in designing courses.

Meanwhile Java is indisputably the finest widely used platform for OO design and Web apps.

Kurt Marfurt, University of Houston: There are three basic programming tasks:

1) Weenie tasks. HTML Web pages to sell real estate. Oracle database searches, writing javascripts, etc. This is where 95% of our UH computer science majors get jobs. Only about 5% do any kind of scientific (computational) programming.
2) Graphics/GUI tasks. This appears to be where the bulk of geophysical application programs fall. Lots of neat stuff to do here. C, C++, and others here.
3) Computational tasks. Modeling, multidimensional signal analysis, imaging, and inversion. This is where I work. I like Fortran 90. It increased my productivity by a factor of 3 over Fortran 77. C and C++ are also quite good for computational tasks, but the macho libraries are missing (Sparspak, state of the art eigen-analysis, ...).

Interpreters or human/machine oriented students need to learn C/C++ along with a better high-end package like INT widgets.

Dave Hale of Landmark on Java versus C++: Recommends a "90% Pure Java" strategy.

1) Java has features that speed software development such as array index bounds checking and garbage collection.
2) Imagine you are writing a scientific program. Often you will have infrastructure code, maybe a GUI, parameter retrieval, in addition to a numeric kernel. There are three alternatives:

   - 100% C++ (fast but difficult, takes longer to write);
   - 90% Java with selected numeric kernel in C++ (as fast, but easier to write). As long as you do all the object-oriented stuff in Java (what I really mean by "90%"), then the Java-to-C++ glue is easy.
   - 100% Java (1.5-3 times slower, for many numeric kernels, than the two other cases). For more object-oriented code, the difference is less; and I have even seen Java code beat the C++ equivalent. Most importantly, the Java code is easiest (takes the least time) to write.

   For large commercial software projects, a factor of two in development and maintenance time is more important than a factor of two in execution time. This is especially true if one writes Java so that inner loops may be ported to C++, as necessary.
3) Java was designed for OOP. C++ is C with OOP added later. The backward compatibility with C shows and it hurts; e.g., in exception handling. A more obvious example is the lack, until recently, of even a standard class for strings. This has led to creation of incompatible (but otherwise useful) C++ class libraries. Even today, with ANSI standard C++, would you use string or wstring in your C++ code? In contrast, a Java String is legal currency.
4) Java comes with standard documentation syntax and tools (javadoc). C++ does not.
5) Java comes with lots of high quality packages from Sun, with source code. These provide coding examples and a de-facto standard coding style.
6) The Jython interpreter offers seamless integration of Java with Python.
7) The standard GUI toolkit, Swing, works well. With C++, would you use Qt, Gtk+, wxWindows, Motif, ...?
8) Examples of (100% Pure Java) packages for scientific computing: Jama, Colt from CERN, VisAD.

What we are proposing. Taking into account our goals and feedback from our survey, we propose a thorough overhaul of the way we currently teach (or, more precisely, do not teach) computing skills to undergraduates.

Rethinking CS101

Over the last few years, computation has undergone dramatic transformations. For example, computations are about concurrent interactions with users, networks, and environments. The World Wide Web, embedded microchip controllers in every gadget you can buy, and large-scale, interconnected office-suite applications are all part of this new style of computation... The Java programming language was designed for this new challenge. Too often, the Java language is taught as just a variant of C++. In fact, Java technology can represent a whole new way of thinking about computation.

—Lynn Andrea Stein, MIT

(From an interview at http://java.sun.com/features/1998/06/ lynnstein.html. *More information on Stein's "Rethinking CS101" is at* http://www.ai.mit.edu/projects/cs101/.

We propose that students have a minimum of two semesters of computational science. The first semester would focus on foundation issues and tools. Students would learn emacs as an integrated environment for text manipulation, composing email, and interacting with various interpreters. They would learn how computers work: files and devices, the kernel, processes, shells, and shell programming. They would learn the nature of geophysical computing (balancing computation versus data acquisition and processing). They would learn how to download and install free software, how to use make. They would also learn some differences between Unix and Windows. And they would get a solid introduction to programming via a modern, interpreted language such as Python. Python is fun, free, runs on a broad range of platforms and has a large library of sophisticated modules, including numerical. It meets all our criteria for a first language. It can be learned quickly, a necessary requirement if we want faculty involved.

In the second semester, students would learn Java and large-scale geophysical programming. The transition should be greatly eased by the Jython interpreter. It is a Python interpreter written in Java that basically offers Python scripting for Java. With Jython, you have interactive access to all Java classes and methods and data structures, without glue. Java is a clean OO language that encourages good style. It would also give students a highly marketable skill. As we have seen, Java is now a serious language for large-scale geophysical computations. This would therefore be a good point to introduce key ideas from numerical analysis. Java also offers a good starting point if other languages (C, Fortran) are required later in a career.

It is essential that students get substantial programming practice in later courses. This means that professors need to learn right along with students and be willing to integrate new software tools into course work. If faculty do not lead, students will not follow.

For more information, resources and discussion see *http://acoustics.mines.edu/tpl*. E

Corresponding author: