Numerical Recipes: Does This Paradigm Have a Future?

William H. Press
Harvard-Smithsonian Center for Astrophysics
and
Saul A. Teukolsky
Department of Physics, Cornell University

June 6, 1997

Long-time readers of *Computers in Physics* may remember us as the editors/authors of the Numerical Recipes column that ran from 1988 through 1992. At that time, with the publication of the Second Edition of our *Numerical Recipes* (NR) books in C and Fortran[1, 2], we took a sabbatical leave from column writing, a leave that became inadvertently permanent. Now, as a part of *CIP*'s Tenth Anniversary celebration, we have been invited back to offer some observations about the past of scientific computing, including the educational niche occupied by our books, and to make some prognostications about where the field is going.

CIP's first decade of publication closely overlaps *Numerical Recipes'* first decade in print, NR's original Fortran edition having been published in early 1986. This is not entirely a coincidence. In the *preceding* 15 years (the 1970s through mid 80s) the constituency of scientific computing had undergone an extraordinary broadening, from a few specialists in only certain subfields, to virtually every researcher in broad areas of science. Available technology had moved from the single campus computer, jobs submitted via punch-card decks handed to a functionary behind a little window, to widely available "time sharing systems", minicomputers, and the nascent PC, whose revolution has been the story of the last decade. Nevertheless, certain scientific institutions are notoriously slow to react to structural revolutions in science, with journals and textbooks among the slowest and stodgiest of all. By the 80s, there were obvious needs to which both *CIP* (as a journal) and NR (as a textbook) were inevitable reactions.

Inside the Black Boxes

Through the 1970s, at least, almost all physicists – and we include ourselves – enjoyed a blissful, almost touching, ignorance of numerical methods. To give a personal example, we recall an early collaboration, when we were both graduate students at Caltech. We were having difficulty in integrating a second-order ODE numerically, and our mutual thesis advisor suggested that we consult Herb Keller, the distinguished professor of applied mathematics. "What method are you using?" Keller asked us. We looked at each other blankly – you mean there's more than one numerical method?

"Runge-Kutta," of course we answered.

"Runge-Kutta! Runge-Kutta!" he exclaimed, banging his head with his fist. "That's all you physicists know!" He suggested that we look into Bulirsch-Stoer, which we did, to our eventual benefit.

Eventual, but not immediate, because there is a twist to this story: Bulirsch-Stoer turned out to be a *poor* choice of method for our problem, while Runge-Kutta, when we eventually learned to integrate away from – not into – singularities, and to make a pre-integration change of variables in the equations, worked splendidly. So this story illustrates not only our lack of knowledge about numerical methods, but also that when physicists consult professional numerical analysts, either in person or through a book or journal article, they not infrequently will be disappointed. This and similar early experiences firmly convinced us of the necessity for an algorithm's user (the physicist) to understand "what is inside the black box."

This ideal became, later, one of the defining features of our Numerical Recipes project. It was then, and remains now, exceedingly controversial. The physicist-reader may be astonished, because physicists are all tinkerers and black-box disassemblers at heart. However, there is an opposite, and much better established, dogma from the community of numerical analysts, roughly: "Good numerical methods are sophisticated, highly tuned, and based on theorems, not tinkering. Users should interact with such algorithms through defined interfaces, and should be prevented from modifying their internals – for the users' own good."

This "central dogma of the mathematical software community" informed the large scientific libraries that dominated the 1970s and 1980s, the NAG library (developed in the U.K.) and its American cousin, IMSL. It continues to be a dominant influence today in such useful computational environments as MATLAB and Mathematica. We don't subscribe to this religion, but it is worth pointing out that it is the *dominant* religion (except perhaps among physicists and astronomers) in mathematical software.

This controversy will continue, and NR's viewpoint is by no means assured of survival! An early book review of NR, by Iserles[3], gives an interesting and balanced perspective on the issue.

Origins of Numerical Recipes

Not infrequently we are asked questions like, "How did NR come about?" and "How long did it take to write?" A brief history:

Although Brian Flannery and Bill Vetterling have pursued highly successful careers in industry (at Exxon Research Labs, and Polaroid, respectively), we four NR authors were all in academia at the project's inception. Three of us (Flannery, Press, and Teukolsky) had taught courses on scientific computing in our respective departments, and we had all noticed – one could hardly *not* notice – the almost complete lack of any usable text for a physics-oriented course. Flannery, who taught a course at Harvard to astronomers in 1979, was the first to suggest that a book should be written, and his course notes were the original scaffolding for the project. Press took over the course, and leadership of the project, when Flannery left Harvard for Exxon at the beginning of 1981.

Just as there was a "fifth Beatle," there was also a fifth NR author! Paul Horowitz, co-author of the well known book on electronics[4], was an NR author from September, 1982, through the first part of 1983; so he was in fact the third, not fifth, Beatle. When he left the collaboration (parting on friendly terms, but having written nothing), he contributed his no-doubt favorite phrase, "the art of" to the book's subtitle. Press spent the month of April, 1983, at the Institute for Advanced Study in Princeton, writing as fast as he could. The sheer volume of the output (or rather, lack thereof) convinced Press and Flannery that more recruitment was in order. Saul Teukolsky joined the collaboration in August; and Bill Vetterling, in September. Vetterling, an experimental physicist, replaced Horowitz's useful laboratory and instrumentation perspective in the project.

With the increased number of collaborators, the book was more than half done by early 1984, at which time we began to talk to publishers. Several had expressed early interest, including Addison-Wesley's Carl Harris (who was probably the first to suggest the idea of a book based on Flannery's course), and Ed Tenner of Princeton University Press. Tenner sent a big piece of our manuscript out for review, and agreed to share the reviews with us. The reviewers were Forman Acton (himself the author of a wonderful book on computing[5]) and Stephen Wolfram (long before the first release of Mathematica). Acton's review was laudatory. Wolfram wrote, "The book's style is chatty, and in parts entertaining to read. But in achieving its practical ends, much of it is quite sterile." One would never say such a thing about Wolfram's new bodice-ripper[6], of course!

David Tranah ultimately convinced us to sign with Cambridge University Press, a choice that we don't regret. C.U.P. is big enough to do a good job on worldwide distribution, but small enough to be responsive to our needs as authors. As a not-for-profit organization, C.U.P. also has the luxury of taking a longer view, where a purely commercial publisher might optimize only for the short run. For many years now, Lauren Cowles has been our capable and indulgent editor at C.U.P.'s North American branch in New York.

The beginning was somewhat rockier, however. We delivered the completed manuscript, in the original version of TeX (now called TeX78 [7]), on November 24, 1984, in the form of a VAX/VMS half-inch backup tape, as created on a MicroVAX II. (We include these details for their quaint ring.) Ours was the first TeX manuscript ever received by C.U.P. We wanted C.U.P. to have our crude formatting macros redefined by a professional book designer, and we wanted to have the book copy-edited by a professional copy editor.

In the event, we got the copy editing, but not the book design. At that time, a mere decade ago, there was simply no interface between "book designers" and "people who can use TeX". Indeed, it took C.U.P. more than 6 months to find a firm that could even read our tape and produce camera-ready output in the same format that we originally delivered. (C.U.P. suggested seriously that the book could be published faster – and cheaper –

if they reset it all in type, by hand, in the U.K. We strenuously declined.) Fourteen months thus elapsed between submission and publication.

Original Goals and Subsequent Directions

The first edition of NR had several goals, all of them fairly modest. First, we wanted to fill the obvious gap in the textbook literature, already described. Our attitude was that *any* such attempt was better than none. Second, if we were successful as a textbook, we wanted to decrease – but surely not close – the huge gap then present between "best practice" (as exemplified by the numerical analysis and mathematical software professional communities) and "average practice" (as exemplified by most scientists and graduate students that we knew). Third, we wanted to expand the content of then-standard numerical courses to include some then-less-standard topics (FFT methods, e.g.). Fourth, we wanted to promote a "language neutral" approach in which scientific programming could be accomplished in any available computer language.

Some additional comments should be made about the second and fourth of these goals. "Average practice" for numerical work by faculty and graduate students up to the early 1980s, in the physics and astronomy departments that we knew best (Caltech, Harvard, Cornell) was truly abominable, and perhaps corresponded, barely, to "best practice" in the 1920s. It was not difficult to find practitioners who thought that all determinants should be computed by the expansion of minors (an $N!$ process – Gauss knew better than this at the beginning of the 19th Century), all integrals should be done by Simpson's rule, and, yes, all differential equations integrated by the Runge-Kutta method (or perhaps even forward Euler). Computational astrophysicists still tell the story of the graduate student [name withheld] who exhausted his professor's entire year's computing budget by repeated runs, extending over months, of a certain Monte Carlo program – each run using exactly the same "random" numbers and giving exactly the same results.

Our 1980s goal for NR, overtly discussed by the authors, was to provide a means for bringing average practice into the 1960s – or 1970s in a few areas (FFTs). This is the old problem of, "do you teach to the top of the class, or to the middle?" We were conscious of trading the last factor in 2 in performance, or a more advanced (but less well-tested) method, for pedagogical clarity, or for the use of a better-known method. We still think that these decisions were, in the main, correct; we like to think that NR *has* advanced average practice noticeably. We have learned, however, the necessity of developing thick skins against the barbs of critics who feel that one should teach to the top of the class, and that only the best practice of today – right now – should be allowed in books.

In the Second Edition of NR we did shift the balance slightly, towards more modern (and inevitably less pedagogically accessible) methods in some areas. But we are still not willing to give up on that "middle of the class", whom we view as our primary constituency. Many people do sincerely believe that a book can be both completely accessible and completely state-of-the-art. We say to them, with every good wish, *"Write that book!"*

Our attitude towards computer languages has always been somewhat promiscuous. Fortran, before the very recent introduction of Fortran 90 (on which more below), had become an antique language, if not truly decrepit. The problem has always been no clear candidate to replace it. Our goal was, and still is, to help separate scientific computing, as a body of expertise, from being tied to (specifically) Fortran programming. We try to help level the playing field in which other languages compete for acceptance by scientists. At the same time, we don't feel omniscient enough to pick the winner of this competition a priori.

The original printing of NR, in 1986, had all the Fortran code repeated in Pascal as an Appendix. Pascal, still going strong, got its own full edition in 1989; but by 1992 it had already lost so much ground with respect to the new contender, C, that we decided not to continue it into our Second Edition. Our C version of NR was first published in 1988. C and Fortran Second Editions came out in 1992. We have authorized others to do limited versions of NR in other languages (BASIC [8], e.g.), but we have never viewed these as influential.

In recent years, the C version of NR has outsold the Fortran version about 2 to 1. However, we have no way of distinguishing in sales between scientists who are actively computing in C and other computer professionals who just want to own one book on scientific programming. Our guess would be that a majority of physical scientists still use primarily Fortran, with a large (and mostly younger) minority using primarily C. Smaller minorities make *primary* use of Mathematica, MATLAB, IDL, and similar integrated "total environments," about which more below.

Fortran 90, Parallel Programming, and Programmer Productivity

The newest version of NR, published as Volume 2 of the Fortran book, is *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing* [9]. The word "parallel" added in the subtitle is highly significant, but perhaps not in the way that you might first imagine.

Michael Metcalf, a distinguished computer professional at CERN and co-author of one of the best manuals on Fortran 90 [10], introduced us to Fortran 90 through a set of lectures given in a very beautiful setting at the International Centre for Theoretical Physics (ICTP) in Trieste in May, 1993. Soon after that we apprenticed ourselves for a time to Gyan Bhanot, a computational scientist at Thinking Machines Corporation and expert programmer in CM Fortran, a Fortran 90 precursor developed for TMC's Connection Machines.

It was, for us, as if the sky had opened up and all heaven stood revealéd.

Well, perhaps not quite. We did (and do) see in Fortran 90, however, something *much more important* than its superficial traits of being (i) a much needed updating of Fortran to a modern computer language, and (ii) a language designed to produce code that can be parallelized on computers with multiple processors. To explain what that something is, we must digress briefly on the subject of programmer productivity.

It is a commonplace (see, e.g., [11]) that the average productivity of a professional programmer (that is, lines of final, debugged and documented code divided by total programmer effort) is on the order of four lines per day. This is sometimes rendered as "two lines of code before lunch, two lines after." In research science our standards for documentation and debugging are *much* lower, and our graduate students work harder, so our average productivity, in lines, may be as much as two or three times higher.

Another commonplace, as emphasized by Mary Shaw [12], is that a single individual, without specialized software engineering tools or training, can write and master a program of length about 3000 lines, but not longer. Since we physicists offer no such specialized training or tools to our graduate students, this sets a lifetime limit on the complexity of any single scientific project that they can undertake.

We are convinced by the evidence that these limits on productivity are fundamental. However, it seems to be a fact [11, 12] that a programmer has about the same productivity (in lines per day, or "mastered" lines) independent of the level of the language, over a range as wide as (e.g.) assembly language to (e.g.) Mathematica. It follows, and the thought is not original to us, that the only way projects in computational science of larger scope can be done by individual researchers (as opposed to software engineering teams), is by the use of intrinsically higher level languages.

This, for us, was the revelation of parallel programming in Fortran 90. The use of parallel and higher-level constructions – wholly independently of whether they are executed on tomorrow's parallel machines or today's ordinary workstations – expresses *more science per line of code* and per programming workday. Based on our own experience, we think that productivity, or achievable complexity of project, is increased a factor of 2 or 3 in going from Fortran 77 to Fortran 90 – *if* one makes the investment of mastering Fortran 90's higher level constructs.

We give a simple example: Suppose we have two matching arrays, one with a list of galaxy velocities, the other with the corresponding magnitudes. We want to know the upper-quartile value of the magnitude for galaxies whose velocity is in the range $100 < v \leq 200$. (Astronomical magnitudes decrease as objects get brighter, so this corresponds to finding the lower-quartile magnitude numerically.) In Fortran 77 the code is something like the first example shown in Table 1. While not difficult code, it does contain various "gotchas" that are painful to get right, mostly having to do with array index arithmetic.

The second example in Table 1 shows the same task accomplished in two lines, using Fortran 90 language intrinsics, plus routines from our NR in Fortran 90 book. While the lines are each fairly dense, they are completely free of index fussiness; indeed, they are much closer to the underlying conceptual task. In this example, `pack` and `ceiling` are Fortran 90 language intrinsics, `array_copy` and `select` are Numerical Recipes procedures.

The same task, accomplished in Mathematica [6], is shown as the third example in Table 3; in IDL [13] it is shown as the fourth example. (Note that `Select` in Mathematica has a completely different meaning from `select` as a Numerical Recipes routine name!) Mathematica lacks the "data parallel" constructs of Fortran 90, but has powerful list handling intrinsics instead. In this example, Mathematica's main weakness is the awkwardness of its component addressing. IDL's formulation is almost crystalline in its clarity.

(To understand it, note that the `where` and `sort` functions return arrays of indices, not of values.)

Programming Languages versus Total Environments

In the preceding example, IDL, Mathematica and "Fortran 90 plus Numerical Recipes" emerged as comparably high-level languages. Indeed, syntactically and semantically, all are of comparable complexity. In particular, all are "large" languages, requiring a serious investment in learning. In almost every other respect, however, Fortran 90, as a direct programming language, is really quite a different beast from the other two. This brings us finally to the question posed by the title of this article.

Just as the "central dogma" of the mathematical software community turned most program libraries of the 1970s and 80s into black boxes with defined interfaces, there is a similar emerging dogma of the 1990s, that scientific programmers should move to high-level "total environments" (here called "TEs") such as Mathematica, MATLAB, IDL, and similar. As already discussed, we agree completely with the necessity of moving to higher level languages. But we strongly disagree with the new dogma.

The key problem in working with any TE, we find, is the user's lack of control over *scalability* in the TE's internals. For example, one can easily write a piece of code that works splendidly for 10 data points, but fails for 10^6 data points. Sometimes the problem is simply that the TE is just too slow. Other times, its memory requirement goes through the roof.

But isn't the problem the same for a programming language such as Fortran 90? Sometimes yes, if the scaling with size of data is truly intrinsic to the underlying numerical algorithm, as inverting a matrix for example. But very often, the answer is no, the runaway scaling in the TE is *not* fundamental to the *user's* problem, but is rather a "feature" of the generality of the data structures used in the TE's internals, or the internal algorithms used to manipulate those structures. In a programming language like Fortran 90, when you encounter such a problem, you fix it – often by a bit of messy, lower-level programming, in which you create a better kind of specialized data structure, or more highly optimized "inner loop". These options are

Fortran 77, with external sort:

```
n = 0
do j=1,ndat
     if (vels(j).gt.100..and.vels(j).le.200.) then
          n = n+1
          temp(n) = mags(j)
     endif
enddo
call sort(n,temp)
answer = temp((n+3)/4)
```

Fortran 90, with Numerical Recipes procedures:

```
call array_copy(pack(mags,(vels>100..and.vels<=200.)),temp,n,nn)
answer = select(ceiling(n/4.),temp(1:n))
```

Mathematica:

```
Select[ Transpose[{vels,mags}], (#[[1]] > 100. && #[[1]] <= 200.)& ]
Sort[%, ( #2[[2]] > #1[[2]] )& ] [[Ceiling[Length[%]/4]]] [[2]]
```

IDL:

```
temp = mags(where(vels le 200. and vels gt 100., n))
answer=temp((sort(temp))(ceil(n/4)))
```

Table 1: Coding of the same example in Fortran 77 and in three "higher level" languages, Fortran 90, Mathematica, and IDL.

not available within a TE.

Imagine the different "levels" of programming spread out vertically on some kind of logarithmic scale. In Fortran 77 or C, you spend all your time at the bottom of the scale, down in the mud. In Mathematica, MATLAB, or IDL, you spend almost all of your time (generally quite productively!) at the top of the scale; but it is practically impossible to dig down when you need to. Neither of these paradigms is optimal.

We conjecture that an optimal programming model is one in which the programmer has approximately equal access to each logarithmic level, and we think that a skilled programmer will spend roughly equal programming time in each logarithmic level, laying out bold strokes in the top levels, clever optimizations in the bottom ones. Fortran 90 is by no means a perfect language. But augmented by a good set of utility routines and an accessible source-code library (you can guess which we favor), it seems to us to be closer to the ideal than any other choice available right now.

There is a possible rejoinder that our objections to TEs simply reflect today's technological limitations, and that they will get much better in the future. No doubt true. However, TEs, because of their very generality, will always be much slower than the execution of native arithmetic operations on simple data structures that are "close to the machine". The latter capability is exactly what a modern compiled language – one that contains a broad mixture of high level and lower level constructs – provides.

If there is a single sentence in the Numerical Recipes books that has annoyed more people than any other, it is this one: "The practical scientist is trying to solve tomorrow's problem on yesterday's computer; the computer scientist, we think, often has it the other way around." We stand by this statement.

And What About C++?

Indeed, what about C++? This language would seem to meet all our requirements: It allows arbitrary high level constructions, through the mechanism of a class library, yet its underlying C syntax is even more primitive, and closer to the machine, than old Fortran 77.

We have spent a lot of time in the last five years scratching our heads over C++ (and over Java in the last couple of years). Probably a *Numerical*

Recipes in C++ would have value. There are several reasons, however, that we have not yet produced such a version.

First, the original "democratic" dream of object-oriented programming, that every programmer would accumulate an idiosyncratic collection of useful object classes whose reusability would allow moving to ever higher levels of programming abstraction in the course of a career – this dream seems dead. Instead, today's trend is toward a fixed, universal object class library (Microsoft's MFC, more or less), and is discouraging of more than a minimal amount of idiosyncratic programming at the object class definition level. The result is that C++ has become essentially a large, but fixed, programming language, very much oriented towards programming Microsoft Windows software. (Fortran 90, on the other hand, is strongly biased towards scientific computing – it is a poor language in which to write native Windows applications!)

Second, there is a genuinely unresolved debate regarding what should be the fundamental structure of a scientific programming library in C++. Should it be true to the language's object-oriented philosophy that makes methods (i.e., algorithms) subsidiary to the data structures that they act on? If so, then there is a danger of ending up with a large number of classes for highly specific data structures and types, quite complicated and difficult to learn, but really all just wrappers for a set of methods that can act on multiple data types or structures. There exist some ambitious class libraries for scientific computing (e.g., [14]) that suffer, to one extent or another, from this problem.

Confronting just this issue, a competing viewpoint, called "generic programming with the Standard Template Library (STL)" [15, 16, 17] has emerged. Here algorithms and data structures ("containers") have more equal claims to primacy, with the two being connected by "iterators" that tell the algorithm how to extract data from the container. STL is implemented as C++ template classes that naturally allow for multiple data types (single versus double precision, e.g.).

We don't feel ready to choose one of these C++ methodologies, and we have only just begun thinking about what we might conceivably propose as an alternative. One possibility would be to define a generic, very high level, interface that encapsulates a set of objects and methods comparable to everything in Fortran 90, but not in itself dictating any particular template or class inheritance structure for its implementation. Then, a variety of compatible implementations could be written, optimized quite differently for today's serial or tomorrow's parallel machines. Our preliminary efforts along these lines are at `http://nr.harvard.edu/nr/cpp`, and we would be grateful for thoughts and comments from readers.

Where We End Up

We think that compiled languages, giving the user direct access to machine-level operations acting on simple data structures ("close to the machine") continue to have an important future in scientific programming. The Numerical Recipes paradigm is not, we think, in danger of extinction. Total environments like IDL or Mathematica are additional great tools to have. There is therefore room for improved interfaces between the two methodologies.

Nevertheless, to remain competitive, compiled languages (and their users) must move to higher-level constructs and to parallelization, both in the computer and in the programmer's mind. The programming languages of the future, whether Fortran 90, C++ with STL, or some new variant in the Fortran or C families, will be "large" languages. The "Pascal experiment" (of a small, tightly structured language) is in the dustbin of history. Indeed, it seems unlikely to us that any other language family will overtake the predominant positions of Fortran and C.

It takes about ten years for the power of the supercomputers of any given era to migrate to the desktop. By the time we celebrate the twentieth anniversary of *Computers in Physics*, we can expect machines with hundreds of processors to be routinely available for scientists' personal use. Making effective use of all this horsepower will require contributions from many different fields: from the computer science and numerical analysis communities, libraries of software to distribute problems over multiple processors; from computer vendors, optimizing compilers for a multiprocessor environment; and from users, a willingness to change our way of doing business. We need to move away from a coding style suited for serial machines, where every microstep of an algorithm needs to be thought about and explicitly coded, to a higher-level style, where the compiler and library tools take care of the details. And the remarkable thing is, if we adopt this higher-level approach right now, even on today's machines, we will see immediate benefits in our productivity.

References

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Second Edition (Cambridge University Press, Cambridge, 1992).

[2] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in Fortran: The Art of Scientific Computing*, Second Edition (Cambridge University Press, Cambridge, 1992).

[3] A. Iserles, *Mathematical Gazette*, **73**, No. 464 (June, 1989).

[4] P. Horowitz and W. Hill, *The Art of Electronics*, Second Edition (Cambridge University Press, Cambridge, 1989).

[5] F.S. Acton, *Numerical Methods that Work*, 1970; reprinted edition (Mathematical Association of America, Washington, 1990)

[6] S. Wolfram *The Mathematica Book*, Third Edition (Cambridge University Press, Cambridge, 1996).

[7] D.E. Knuth, *TeX and Metafont: New Directions in Typesetting*, (Digital Press, Bedford, MA, 1979); see also the Preface in D.E. Knuth, *The TeXbook*, (Addison-Wesley, Reading, MA, 1984).

[8] J.C. Sprott, *Numerical Recipes Routines and Examples in BASIC*, in association with Numerical Recipes Software (Cambridge University Press, Cambridge, 1991).

[9] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing*, Volume 2 of Fortran Numerical Recipes (Cambridge University Press, Cambridge, 1996).

[10] M. Metcalf and J. Reid, *Fortran 90/95 Explained* (Oxford University Press, Oxford, 1996).

[11] F.P. Brooks, *The Mythical Man-Month*, revised edition (Addison-Wesley, Reading, MA, 1995).

[12] M. Shaw, *Journal of Computer Science Education*, **7**, 4 (1993).

[13] *Interactive Data Language*, Version 4 (Research Systems, Inc., Boulder, CO, 1995).

[14] J.J. Barton and L.R. Nackman, *Scientific and Engineering C++* (Addison-Wesley, Reading, MA, 1994).

[15] A. Stepanov, *Byte*, October, 1995; available at http://www.byte.com/art/9510/sec12/art3.htm

[16] M. Nelson, *C++ Programmer's Guide to the Standard Template Library* (IDG Books, Foster City, CA, 1995).

[17] D.R. Musser and A. Saini, *C++ Programming with the Standard Template Library* (Addison-Wesley, Reading, MA, 1996).