Science, Computational Science and Computer Science: At a Crossroads

D. E. Stevenson
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
(803) 656-5880 or steve@cs.clemson.edu

August 16, 1993

Abstract

We describe computational science as an interdisciplinary approach to doing science on computers. Our purpose is to introduce computational science as a legitimate interest of computer scientists.

We present a foundation for computational science based on the need to incorporate computation at the scientific level; *i.e.,* computational aspects must be considered when a model is formulated. We next present some obstacles to computer scientists' participation in computational science, including a cultural bias in computer science that inhibits participation. Finally, we look at some areas of conventional computer science and indicate areas of mutual interest between computational science and computer science.

Keywords: education, computational science.

1 What is Computational Science?

In December, 1991, the U. S. Congress passed the High Performance Computing and Communications Act, commonly known as the *HPCC*. This act focuses on several aspects of computing technology, but two have received the most attention: *(i)* computational science as embodied in the Grand Challenges (Table 1) and *(ii)* the National Research and Educational Network (*NREN*). The *NREN* is to be a network of extremely high speed, capable of transmitting in the terabit per second range—approximately ten times faster than we can currently transmit data. The Grand Challenges are engineering and scientific problems considered vital to the economic well-being of the United States. Many of these problems, such as drug design and global climate modeling, have worldwide impact. The exact goals of the *HPCC* are published in a pamphlet and updated yearly[14].

The science and engineering components of the *HPCC* require an interdisciplinary approach to solving very difficult problems. The solutions require the concerted actions of physical scientists, engineers, mathematical scientists, and computer scientists. Computational science embraces this collaborative effort among many diverse disciplines. Even in the final analysis, the "answer" may have to be pieced together from the many viewpoints.

Our purpose is to ask whether today's computer scientists are able to take up the challenge of computational science. Some might argue that computational science is not an interest of computer science; that current areas of interest comprise the total domain. Indeed, it is strange that one has to argue for scientific applications as a part of computer science, since, after all, modern computing's roots are in scientific and engineering applications.

An exact definition of *computational science* is open to debate. There are many programs in the

Astronomy	Semiconductor Design
Human Genome Mapping	Structural Biology
High T_c Semiconductors	Superconductivity
Molecular Design of Drugs	Underwater Acoustics
Naval Architecture	Weather, Climate, and Global Change Modeling
Quantum Chromodynamics	Vision

Table 1: Grand Challenges

United States and elsewhere that use the term and each program probably has its own view of computational science. We outline the Clemson view to computational science as one possible approach. That view recognizes three components to computational science: applications, algorithms, and architectures. We visualize this as a pyramid supporting the science and engineering. Applications need not be restricted to the traditional science and engineering applications; for example, complex econometric models can also benefit from computational science.

The conduct of computational science, in the Clemson view, is interdisciplinary. This interdisciplinary thinking demands that the constituent disciplines (physical sciences, engineering, mathematics, computer science) maintain their autonomy. Within computational science, a computer scientist retains her/his expertise in computer science, but emphasizes applications in science or engineering.

We argue that although computational science is not for every computer scientist, computational science is an idea whose time has come—again. Our premises:

1. Computational science is addressing problems that have important implications for humankind. These problems are complex and their solutions desirable.

2. Computational science is unlikely to succeed in the near term without further advances in software and hardware. Without computer science involvement, the solutions to these problems will take much more time.

3. Computer science is generally not participating in science and engineering applications, nor is it preparing students to do so in the future.

We present evidence for point 3 and we propose some remedies. We hasten to add that all the constituent disciplines may be in similar situations; see, for example, comments adapted from Robert Pike in *Computing the Future*[30, page 126]. We further point out the obvious changes at the foundations of the scientific method as evidence for these intra-disciplinary changes.

In Section 2 we present the Clemson view of computational science and describe the role of various disciplines. This section primarily addresses issues in the philosophies of science and of mathematics. Section 3 presents partial evidence that computer science is not participating in, nor preparing students for, computational science. Section 4 we propose some actions that computer science can take to prepare for computational science: both in education and in research.

2 The Challenge of Computational Science

This section is primarily philosophical in nature. We discuss four principal areas. The first area is the environment of computational science, with emphasis on the general method of investigation. The second area focuses on methods, in which we outline our view of modeling. The third area relates to the relationship among the scientific application, algorithms, and the architectures. Finally, there is a question of the veracity of a computation.

This section is intended to address a broad-based audience—computer scientists, primarily, but physical scientists, engineers, and mathematicians as well. We do not assume that the reader is currently active in computational science.

2.1 The Environment of Computational Science

Computational science is an emerging discipline characterized by the use of computers to provide detailed insight into the behavior of complex physical systems. Computational science uses computational methods to conduct experiments which are either too expensive or, in fact, impossible to conduct in the real world. A simple perusal of the scientific literature clearly shows that computer simulation is enormously fruitful in most fields. The interplay of experiment, traditional theory, and computational modeling has strong, symbiotic results. The simulations can be used to provide unique insight into physical processes. In order to improve this capability, the full power of computing technology must be available to the scientist and engineer. There are many aspects to computing technology and we emphasize that *computational science* is not synonymous with *supercomputing*. Much scientific and engineering work takes place on workstations; it is as important to have correct answers from a workstation as from a supercomputer. The proper subject of computational science is proper modeling and correct computation.

The modern view of science recognizes an interplay between theory and experiment. This view was first presented in a polished form by Bacon in the *Novum Organum*[1] in the 17th century. Indepen-

[1]The *Organum* is Aristotle's work on reasoning and the scientific method. Bacon's book is oriented towards changing the attitudes of his day which Bacon attributed to slavish following

dently, Kepler and Galileo emphasized mathematics as the language of science. These two thoughts have been merged into the foundations of modern science. Modern science and engineering arose through the interplay of theory and experiment: theories are proposed and the role of experiment is to sort the theories out. Mathematics has not been a bone of contention[2]. However, there are problems with differences between some areas of classical mathematics and mathematics needed for computation[21].

The standard model of scientific enquiry must be altered to include computer models. A simplified version of the new (proposed) process:

1. A model $\mathcal{M}$ is derived from physical or engineering principles. $\mathcal{M}$ may contain submodels previously developed.

2. $\mathcal{M}$ is further developed using numerical techniques into perhaps many computational models $\mathcal{C}_i$.

3. The computational models $\mathcal{C}_i$ serve as a basis for experiments, using visualization techniques or perhaps automated tools, to explore and validate the model $\mathcal{M}$. Experiments or data from real examples of the system are processed as the modeler attempts to validate the model $\mathcal{M}$.

4. At some point, the computational models $\mathcal{C}_i$ provide insights into the physical behavior of the system under study. $\mathcal{M}$ will continue development through refinement based on the results of these computational experiments.

In our view, computational aspects must be considered during model formulation. The computer is too often seen as capable of very fast computation, but rarely are finite arithmetic, numerical algorithms, architecture, and program construction taken into account in scientific formulations. The scientist or engineer who avoids these considerations is at a grave disadvantage. In the same way that sloppy experimental technique cannot be tolerated, so too the inappropriate marrying of applications, algorithms, and architectures cannot be tolerated in computer modeling. It is important to realize that computer technology can be applied inappropriately. On the positive side, the computer allows scientists and engineers to have unprecedented control over their models.

The computer now allows the use of non-linear methods where non-physical assumptions were required before (see Section 2.2). A simple example in every sophomore physics book is the pendulum: if we do not make the "small angle assumption", the resultant differential equation usually makes use of elliptic functions for its solution. Instead of having a nice analytic function to investigate, we must instead run many "numerical experiments" before we can understand the behavior of the pendulum. Such experiments must be carefully performed and documented and are always subject to both computer and human error. Thus, *computation* becomes part of the philosophy of science.

Technical innovation is not without its consequences for the computational scientist. Computational power is often not accessible due to the exotic nature of some of the newer architectures (*e.g.,* hypercubes) or the admitted difficulty of programming and debugging the models. Older, validated models often are difficult to port to the newer architectures. Algorithms that work on one architecture are often inappropriate on another. Looking to the future, we see even more exotic hardware that must be integrated into an already complex environment. *Heterogeneous computing environments* are currently available to large corporations and national laboratories. Computational science is thus involved in delivering technology, directly to the scientist and engineer, while at the same time actually enhancing fundamental scientific models.

Computational science focuses upon with the development of computationally feasible models for physical systems, developing algorithms for solving issues arising in the modeling process, and matching algorithms to computer architectures. This should be accomplished in an environment that frees the scientist and engineer from the confines of low-level programming. The role of computational science is to provide the scientist tools and computational environments which allows fruitful exploitation of available resources without having to resort to non-physical approximations simply to reduce the model to mathematically tractable form. Scientists should not have to be concerned that the computing engine is scalar, vector, and/or parallel with shared or distributed memory. Rather, with an appropriate environment in which to describe the model and to specify the spatial configuration and interactions, the details of the solution within a class of algorithms should be rather transparent. The lack of a cohesive programming model is perhaps the biggest obstacle to computational science. Who better to address this lack of a programming model than computer science?

Our view of "computational science" emphasizes interdisciplinary involvement in the scientific process. The *Clemson Program* has three goals:

of Aristotle. Science in the 17th century was oriented towards "reasoning" but not "experimental verification."

[2] Except for some areas like quantum logic. See [5].

> *Goal 1.* To find and eliminate unwarranted assumptions and approximations in models;
>
> *Goal 2.* To correctly marry the appropriate algorithms to the appropriate architectures given a model and its parameter space; and
>
> *Goal 3.* To deal with the complexity and veracity of the programming process.

The primary impact of computational science will be the development of a new view of science. In order to understand the role of this proposal in the development of computational science, we describe our vision of how computational science will evolve. Sections 2.2 through 2.4 address these goals.

2.2 New Foundations—the New *Novum Organum*

Computational science places science and engineering first and makes sound scientific modeling the basis. The model reflects the scientist's or engineer's understanding of the physical system. Almost inevitably, the models incorporate assumptions about how a system operates. These physical assumptions require the use of mathematical approximations. Such assumptions we call *physical* since they are open to validation procedures within the science. A model with only physical assumptions we call *(physically) exact.* By contrast, assumptions introduced into the model for mathematical convenience lead to a *physically inexact* model; such assumptions we term *non-physical.*

"Classically derived" models are rarely physically exact. That is, such models include non-physical assumptions needed to produce closed-form solutions. We contend that such models are *mathematically exact but physically approximate.* One is therefore left with a nearly exact solution of approximated—and perhaps unrealistic—models. We discuss an example below. Anecdotal evidence shows that often these models may give unsatisfactory results when used in a computational setting. An alternative is to reformulate the models to be more physically exact and therefore more realistic. Unfortunately, these newer models have no closed form and generally are very hard to solve numerically.

The lines between physically exact and inexact may be blurred, but the distinction is useful. For example, consider the model of a pendulum such as one might find in an undergraduate physics book[16]. Figure 1 shows a diagram of a simple pendulum. There is a point mass m at the end of a rigid, massless bar of length L. The pendulum swings in an arc measured by the angle θ. At θ, the restoring force

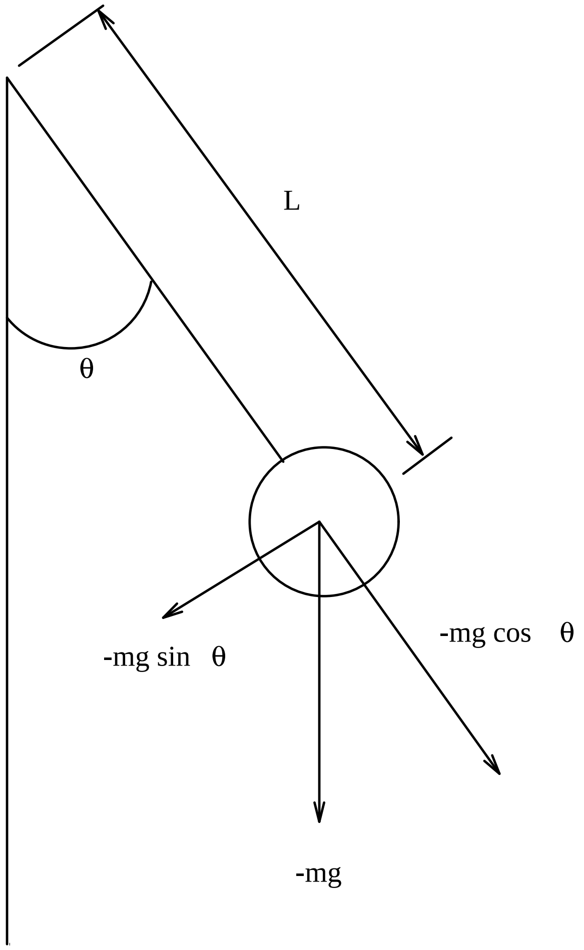


Figure 1: Pendulum Diagram.

is $-mg\sin\theta$, ignoring friction. In this model, the assumptions "point mass", "massless bar", and "frictionless" are physical assumptions since they may be validated. The equations of motion are given by

$$mL\frac{d^2\theta}{dt^2} = -mg\sin\theta. \tag{1}$$

This differential equation does not have an analytic solution (although it does have an elliptic solution[12]).

The next assumption is non-physical: since for small θ, $\sin\theta \approx \theta$, we can rewrite the equation of motion to

$$mL\frac{d^2\theta}{dt^2} = -mg\theta. \tag{2}$$

This latter equation is solvable by analytic methods, leading to the well-known sinusoidal solution. Actually, the assumption that $\sin\theta \approx \theta$ is not, in itself, all that bad as long as we stay in the region for which that assumption is true. However, "small" is a difficult quantity to determine. For example, if the smallest relative error of perception of an angle is 10^{-6} radians, the maximum angle would be about 2×10^{-2} radians; that would make the maximum swing of a twenty foot chain about four inches. Galileo might not have even seen such a pendulum move.

It is important to note that the solution to (2) lets one talk about all sorts of silly things. For example, in this linearized pendulum, one can "wind up" the pendulum, say to 4π (two times around) and the equations of motion will "unwind" between -4π and 4π. No real pendulum[3] does this and hence we would call the model non-physical. Therefore, there is an important distinction between physical and non-physical models. In our terminology, we would say that Equation (2) is "physically approximate but mathematically exact" while Equation (1) is more "physically exact but mathematically approximate". It is mathematically approximate because elliptic integrals are solvable only by computation of series[12].

Since we see computational science as an interdisciplinary endeavor, there is a need to merge the methods and viewpoints from the individual disciplines involved. Under the current methodologies of science, mathematics plays a role as a tool. For the outsider, certain questions about the basis of mathematics are ignored[21]. The most important question for the present discussion is the question of computability. Ordinary calculus, as taught to freshmen and sophomores, assumes certain things about existence, leading to impredicative assumptions that are inherently non-computational in nature[21]. The reliance on computation in computational science opens a very important question: How much of ordinary mathematics is usable in the computational world? This question has been addressed[9, 11, 18, 22, 24, 28, 27] but the results are not generally practiced. As an illustration as to why this question is important, take the recent "discovery" of chaos. Chaos came to light from computational solutions to problems, but one must be sure that chaos is a physical artifact and not a computational one. Some models were known, such as the logistics equation, which were chaotic but it was not until the computer got involved was this attribute seen.

The Clemson Program proposes that modeling proceed by the following principles:

- *Physical Exactness.* We strive to identify nonphysical (mathematically convenient) assumptions and eliminate them.
- *Computability.* We must identify non-computable[4] relationships. No mathematical relationship is exact unless it follows directly from the development of an exact model and is computable. In this sense, most mathematical relationships turn out to be *approximate.*
- *Bounded Errors.* No formulation is acceptable without *a priori* error estimates or *a posteriori* error results. Because the computation is approximate, we must be able to tell "how good" the answers are.

These new models must meet the computational science criteria of no unwarranted approximations and suitability for solution on state-of-the-art computers. We emphasize the rederivation of models for their exactness to physical principles. This should not be taken to mean that we consider *only* computer solutions to these models.

To complete this subsection, consider our pendulum example in light of the paradigm of science given earlier. The model $\mathcal{M}$ is that of the non-linear pendulum of Figure 1. Two computational models come to mind: $\mathcal{C}_1$ as the numerical solution of the elliptic integral or $\mathcal{C}_2$ as a numerical solution to the differential equation defined by Equation (1). In either case, g and L are parameters. We would have to explore the behavior of the pendulum by "solving" the equations repeatedly for different values the parameters. Each run of the computational models is an experiment.

2.3 Applications, Algorithms, and Architectures.

Assuming that models have been properly formulated does not guarantee that the appropriate numerical method(s) or the optimal choice of architecture are chosen for the computational models. Architectural advances have made new and specialized machines available. The scientific computer center of the future will have a network of diverse machines. Compilers and operating systems will have the difficult task of managing these dispatchable machines. The scientists and engineers will want to use these advanced architectures, but the task of knowing what machines are suitable for which algorithms and data ranges will become mind-boggling. If one makes scientists deal with the intricacies of distributed processing, it is more likely that productivity will likely go down rather than up.

The optimal algorithms for these as-yet unknown systems are most likely not the ones that are

[3] In fact, the "pendulum" is now a torsional spring.

[4] Impredicative relations are the basis of non-computability.[21].

optimal on a von Neumann architecture. Our experience with distributed algorithms for hypercubes, for example, would indicate that the old algorithms will not suffice for the new architectures. The problems of designing, documenting, debugging, and supporting a large library of scientific routines have been hinted at in the literature. There is also a problem with an exploding number of versions: often differing only in architectural details. For example, consider the development of the so-called Level 3 BLAS[13]. When *LINPACK* was originally conceived, the only model of computation was the von Neumann model. The Basic Linear Algebra Subprograms—*BLAS* as they came to be known—were motivated by vector operations. The *BLAS*, which were originally considered absolutely primitive, have been redesigned several times as , first, vector processors and, then, distributed processors became available. Designing and tuning such a project as *LINPACK*, or its follow-on *LAPACK*[32], for a large number of incompatible architectures will be daunting, to say the least.

LINPACK also points out the difference between mathematics as practiced by the computationalist and the non-computationalist. For the formal mathematician, it is enough to know that one can invert a matrix using something like Gaussian elimination. That algorithm is probably familiar to any undergraduate in science (including computer science) and engineering. Gaussian elimination, however, may not be the best way to compute the inverse on a computer. *LAPACK*, in fact, has found that certain computers had to be excluded from consideration if optimal error characteristics were to be obtained for the remaining architectures.

2.4 Development and Verification Support for Computational Science

The modeling environment will provide for visualization of results and tools for developing models in the computational science paradigm.

One major goal must be to extend the concept of model derivation to include the numerical and programming aspects. Programming must be considered an integral part of the modeling process. The scientist must believe and be able to verify that the output of the computer model faithfully reflects the intended model. Too often, the programming aspect is considered as an independent activity separate from the rigorous rules of science and mathematics. For a discussion of these areas, see [15].

3 Is Computer Science Out of Step?

In this section, we focus on computer science and its place in computational science. At first blush, one would think that computer science is well-positioned to make important contributions to computational science. Certainly computer scientists have the exposure to programming and current architectures. Surely we should be able to take the specifications of a model and turn it into code. How hard can that be?

The reality, however, is quite the opposite. For example, in a recent workshop, the following problem specification was presented:

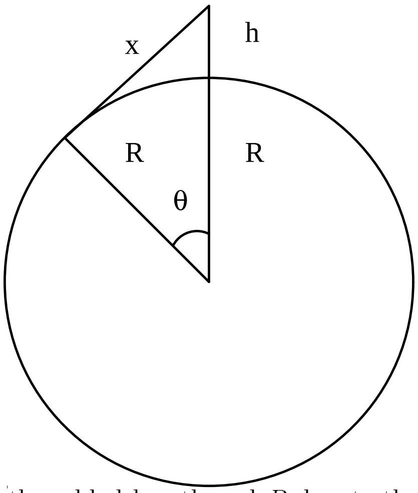
> Take a string and tie it around the equator of the (spherical) Earth. Add l feet to the string. How high a tower must be built to pull the string taut? Find the answer to the best precision you can and defend the number of digits you claim to have found.

The algebraic solution, which uses college algebra and trigonometry concepts, can be found very quickly and is shown in Figure 2. The answer requires solving an implicit trigonometric equation and then solving a quadratic equation. The symbolic system is one form of the answer; it is perfectly acceptable until the contractor asks how much steel she should order.

Computing the *numbers* is very difficult due to the relative sizes of l and R. The solution is made difficult by several cancellations in the computations which must be removed in order to obtain the desired accuracy. It turns out that one can get about 21 digits of accuracy out of 28 digits of precision on a Cray[5]. To find these 21 digits takes a significant amount of work, involving many test programs and a good bit of experimentation and testing. The naive solution coded in double precision does not work well. This type of exercise is very common in computational science. Are computer science students prepared to deal with such problems? Certainly they should be if they expect to participate in computational science.

There are some computer scientists who say that computational science is a subdiscipline of computer science. There are more radical computational scientists who have suggested that computer science should be abolished. Both extremes seem to miss the mark. We argue that computer science is not currently well-positioned to take on the challenges of computational science due to own attitudes toward

[5]We have since done better.



Let l be the added length and R denote the radius of the "perfectly round" Earth. Then

$$R^2 + x^2 = (R + h)^2$$

$$x = R \tan \theta$$

$$\theta - \tan \theta + \frac{l}{2R} = 0$$

Figure 2: The World on a String.

engineering and science and the attitudes transmitted to their students. But we also contend that there is much computer science to be done in computational science and some computer scientists would do well to seek out these opportunities. Let us begin by outlining some reasons why computer science out of step with computational science.

3.1 Lack of Foundations

In any mature discipline, there is a basic set of principles. These principles are the "rules of the game" that can be called the *philosophy* of that science. These principles are known by the workers in the field, if only informally. Section 2.2 sets out a possible philosophy for computational science. For example, the "scientific method" [20] arises from the combined experience and criticism of scientists: how they work, what they will accept as good work and what they reject. Interestingly enough, there may be several philosophies in use at any given time.

What, then, is a philosophy of computer science? Where is the critical analysis of methods? Where do we see the skeptical, reasoned approach to the discipline? Computer science stands in danger of falling into the "meaning" trap. Students can easily see computer science as devoid of meaning and programming devoid of empirical import. We teach "problem solving" devoid of problems: little "sound bites" of ideas without a cohesive whole. Artificial intelligence seeks to emulate human intelligence by formal token systems devoid of meaning. We develop a theory of computational complexity that deals with asymptotic behaviors in regimes far beyond what algorithms are called upon to support in practice.

All around us the ground rules are changing. Computer scientists are ill-prepared to critically analyze their own positions. They cannot determine what is new and what is old; what has worked and what has not; and why. That does not exempt us, however, from improving our foundations. There should indeed be a philosophy of computer science that addresses the questions of the various positions taken on various issues. Our students should be made to understand what is opinion and what is empirical fact and what the "rules of the game" are. In the sciences, engineering and mathematics there are rules of the game and these rules must be followed. More importantly, we must be able to explain to others what we stand for. Here are some areas that need further exploration that are of direct interest to computational science:

> *Basic Questions.* The philosophies of mathematics and science explore two issues: *(i)* what objects exist (*metaphysics*) and *(ii)* how do we come to know about these objects (*epistemology*). Algorithms would seem to be one of computer science's objects, yet textbooks—and the field as well—continue to eschew definitions of algorithms. What is *computational knowledge* and how do we achieve it?
>
> *Literature.* What is the literature of computer science? Programs? Algorithms? Journal articles? If it is programs, are these programs to run on all possible machines? And what are the requirements for veracity? Should a program in a journal article be expected to run as is?
>
> *Formal Methods of Program Specifications.* Should not a program be proven to work and have the behavior described formally? When are formal methods appropriate? Are they required to be validated in the sense of a physical model? What is the empirical import of formalisms? How do formal methods (a formalism) convey meaning (an empirical concept)?

> *Numerical computation.* Numerical processes and floating point applications are virtually ignored in the current programming, compiler, and data structure texts. When addressed, these issues are addressed without application and without any concept of correctness.

Another pervasive foundational problem is the lack of scientific rigor. Most basic to science is a consensual vocabulary and notation. Science and mathematics have struggled ever since Kepler to develop just such a vocabulary. In computer science, though, we have a hodge-podge of definitions with no agreed upon foundation. No wonder the scientists and mathematicians are frustrated when working with computer scientists. Likewise, computer scientists are mystified by the strict notational and definitional framework of the sciences as well as the harsh requirements for proof.

3.2 Lack of Integration of Science and Mathematics

The current *ACM, CSAB* and *IEEE* recommendations for the computer science curriculum include a significant exposure to the sciences and mathematics. The Clemson curriculum for a B. S. in computer science is probably typical:

1. One year of calculus—but no multivariate calculus and no differential equations.
2. One semester each of discrete mathematics, statistics, linear algebra and "decision science[6]."
3. One year of natural science—usually biology or chemistry.
4. One year of physics.

Most of these courses are completed early in the training of the computer scientist. What is missing? For one thing, numerical analysis is conspicuously absent! The contents of these courses are rarely used in computer science courses! On the one hand, we might argue because these things have no *apparent* relevance to computer science, we should not waste our students' time.

However, even within the current curriculum there are problems. Checking my bookshelf for texts used in the data structures-algorithm courses, I find not one of the five uses the word "optimal"; it does not appear in the index of any of the five. I then looked for the word "average". Two did not use the term at all. Two have a subsection on average case analysis. One actually did some derivations. None suggested any empirical validation. The concept of optimal is central to many scientific and engineering disciplines.

As another example, I computed the amount of scientific and engineering literature indexed in "ACM Guide to Computing Literature"[2]. There were 377 pages used to list the literature by *CR* category. Only 17 pages (about 4.5 percent) was needed for the *J.2 Physical Sciences and Engineering* category but 35 pages were devoted to "information processing" applications. Also interesting is the fact that only two pages were devoted to numerical linear algebra. Why is so much attention paid to business applications? And why so little attention paid to engineering and scientific applications?

We would argue the following: we should use scientific, engineering and mathematical contexts precisely because such contexts represent natural subject areas that the student already understands. After all, we live in a physical world. For example:

- A natural—and perhaps the simplest—way to approach parallelism is through simple numerical models. Nature is inherently parallel and most students have personally experienced the phenomena which are being modeled.
- Natural questions of correctness of computation are usually evident in simple numerical problems.
- The vagaries of finiteness can be easily demonstrated in small, easy to understand programs.
- The validation of computer models gives empirical import to programs and is a natural development ground for software testing concepts.
- Simulations of physical systems are far easier to justify and explain than simulations of non-physical systems.
- Some algorithms—simulated annealing and genetic algorithms, for example—are derived from physical principles. If the underlying physics or biology is understood, the algorithm is understood intuitively.

3.3 Lack of Emphasis by Faculty

The above points could be easily overcome if faculty put emphasis on the use of scientific principles and proper mathematics. But how many times have we

[6] Statistics, probability, linear programming.

sloughed off a difficult mathematical point as "useless" when it really is "too hard" to teach or because it is hard to understand? The message is clear to the student: science and mathematics are neither interesting nor important or just too hard. More fundamental, difficult details can be sloughed off as insignificant, leading the students into a false sense of security. If you ignore the hard parts, they cannot hurt you.

With the possible exception of visualization, computer science has been at odds with science and engineering interests. While there are occasional calls for "more mathematics" in the computer science curricula, there are just as many who lament the inclusion of mathematics. Really, now: what is the relationship of mathematics and computer science? Perhaps we would like it to be that "Real computer scientists don't do math—or databases, either."

There does seems to be bad blood between the groups. We have all heard pronouncements on the programming language issue. At a recent conference, the author participated on a panel on computational science. One computer scientist put out the suggestion that *Fortran* should be abolished—without regard to the fact that the community has many well-tested, well-understood programs in *Fortran*, and that most scientists and engineers program only in *Fortran*. The argument was that programs in this newer language would be, oh, so much better because of the work in vectorizing. The scientists counter—and I am afraid that we are not hearing this argument well[26]— is that those old, empirically validated programs are the purpose of *programming*. Calculating the wrong answer quickly is not any help. Programs are not the object of science, *knowledge* is. Those old, antiquated programs are well-tested and jive with the empirical relations observed in the real world. We, computer science, are forgetting the Hamming dictum[19]:

> The purpose of programming is insight, not numbers.

The language debate, if indeed it is a debate, just will not go away. But are we asking the right questions[26]? If we continue with an attitude[26] that the world is waiting with bated breath for the next program—or programming language—we will not endure as a discipline. If we continue imbuing our students with this attitude, we will continue to see declining enrollments as the sciences and engineering disciplines draw the best and the brightest. We also run the risk that the application disciplines will alter their own curricula to embrace the useful parts of computer science.

3.4 The Results

The result of these and other factors is that computer science (or even perhaps computer engineering) students do not understand science and are ill-equipped to deal with scientific and engineering software. However, computer science students are not irretrievably lost to science. The author has been involved along with mathematics and physics faculty in developing courses for computational science. We have had a broad mix of students, including computer science, mathematical science, physics and engineering students, who have taken the courses. The computer science students, after being given the instruction needed to make up their prerequisite deficits, perform very well. Since they already understand programming, they can concentrate on the algorithms. The non-computer science students find programming hard and often rely on the computer scientists to deal with algorithm complexities. The students in this class respond enthusiastically when presented hard problems involving higher mathematics. One student, who is a co-op student, summed it up best: "I'm not sure I'd like to do this for a living, but it's been the most realistic use of my training."

Computer science is not the only loser: scientific and engineering codes are being written using inappropriate, ineffective, and inefficient algorithms because the scientists and engineers are forced to "go it alone." The experience of computational science teams, in theory and in practice, is that no one has to go it alone and that everyone benefits from the interdisciplinary team approach. The problems facing science and engineering are no longer solved by a single person but by a team. The nature of computational science is inherently interdisciplinary.

4 Where Should Computer Science Put Its Effort?

Computational science is an interdisciplinary area and thus does not properly contain any one of its subdisciplines; we do not think of it as an independent discipline. All the constituent disciplines must make adjustments and concentrate efforts. There are several different areas wherein computer science can put out effort, at the *K-12* level as well as the undergraduate and graduate level. We argue below that the high school student is well-equipped to enter the computational science pipeline. At the graduate level, we can offer programs of study which familiarize the students with scientific and engineering problems and their computational solution. Finally, there are research programs which advance both computer science and computational science.

4.1 Education

Ultimately, we see the academic involvement in computational science as spanning high school, undergraduate and graduate studies. Research programs by computational scientists will continue to absorb the well-trained researcher for many years to come. There is an immediate problem of publicizing the Grand Challenges and justifying to high school and undergraduate students the excitement and importance of these and other problems. This can most fruitfully be done by developing a sense of curiosity in the physical world and an appreciation of mathematical and computer modeling. This perhaps includes changing some cherished teaching modules along the line. We must develop in the students a curiosity relating to observations of what can and cannot be done with the computer. As has been seen in competitions such as *SuperQuest* at Cornell University and other state programs[7], high school students respond enthusiastically to real problems in science or engineering. Even video games, with their goal of realism, make use of physical principles. Movies, such as *Star Wars*, use enormous amounts of supercomputing time to generate their effects. How many computer science graduates are able to step into any of these endeavors?

Currently, there are too few trained computational scientists to form a critical mass on any one problem. We need to provide a program that serves the secondary school student as well as the post-doctoral fellow. It is necessary to increase interest in numerical analysis, scientific software engineering, languages, algorithms and architectures as disciplines and as requisite knowledge of all computational scientists. Under the current situation, the expertise for computational science comes from the constituent disciplines.

4.1.1 Goals

The major educational goals of computational science at all levels are

1. To appreciate the role of computation in science and to stimulate interest in computational science.

2. To create a healthy sense of what computation can and cannot do with respect to scientific models.

3. To instill understanding of the application-algorithm-architecture nature of computational science.

4. To expose the students to the consequences of not following proper computational practices. the correct ones.

The objective is to develop a cohesive, comprehensive foundation for dealing with numerical methods and software. We must also be careful not to identify computational science as the traditional numerical analysis course. Numerical textbooks are largely independent of applications, counter to the computational science viewpoint. Too often, students are not introduced to pathologies in computation until they are out of school and the results count for real. It is also true that we do not hold scientific programs to the same rigorous standard that the rest of science must meet. This latter situation is unacceptable. Such rigorous standards would be called—at first blush—*software engineering of scientific software* to differentiate it from software engineering in its more usual setting.

4.1.2 High School and Undergraduate Programs

We need a comprehensive curriculum in computational science. Our view is that there need not be a separate administrative unit to develop a viable curriculum. Our initial curriculum is below.

1. Each scientific or engineering department which is participating in the computational science program would make available a course with the approximate title, "Computational Models in X." The purpose of these courses is to give the students as wide a spectrum of subjects as possible to choose from.

2. Mathematics requirements are kept to a minimum. At the high school level, one can deal very effectively at the intuitive level. Significant problems can be dealt with using only precomputer concepts such as elementary finite difference techniques.

3. Most of the disciplines at the undergradate level already have significant exposure to mathematical science courses. For numerical work, how-

[7] There are several state programs. One is put on by the North Carolina Supercomputing Center at Research Triangle. The program involves high school students from around the state in a problem chosen by the students that uses supercomputing in the solution. Several other states—*e.g.,* Alabama and New Mexico—have similar programs. Clemson is inaugurating a program for South Carolina.

ever, there are three basic requirements: (i) sequences, convergence, and error; (ii) differential equations; and (iii) linear algebra.

4. Computational requirements are likewise part of most technical subject areas. There are four subject areas that should be studied:
   (a) data structures specifically oriented towards the problems in computational science. We have developed a list of some sixty specialized structures.
   (b) design of graphical user interfaces. This includes graphics, human-computer interaction, and even compiler design.
   (c) introduction to computability theory, emphasizing recursion and recursive functions, to understand what is computable and how to think about computation.
   (d) software engineering of scientific software.
5. Two computational science modeling courses: one emphasizing techniques for discrete models and one emphasizing continuous models.

These courses must be developed around modules which emphasize the interaction of the application (problem); the analysis of numerical and non-numerical algorithms; and the appropriateness of the architecture(s) available. This can be done by organizing around three units.

1. The first unit introduces a problem and should be discussed by someone in the relevant field.
2. The second discusses possible solution techniques. Various approaches should be tested in their order of intuitive appeal.
3. The third unit—given after the students have programmed and studied their solutions—discusses the teaching points. Each unit is accompanied by written material. The students will prepare a report—much like a laboratory report—on their solutions and observations.

The core problem for most computer science curricula is the lack of mathematics—most notably, the lack of differential equations and a solid linear algebra course. Most curricula now have positive involvement and reinforcement in the traditional sciences: physics or chemistry. Since there is not a mandated curriculum in computer science, one needs to work within the framework of the *CSAB* checklist and the proposed *ACM-IEEE* proposal. For example, the Clemson program is accredited by *CSAB* and the needed changes can be accomplished within the current BS curriculum: the student takes a mathematical science or numerical analysis applications emphasis[8] and two senior-level modeling courses.

We have tested this concept through a special topics course which included seniors and post-graduates. The course we taught had engineering, physics, mathematics, and computer science students. In some cases, the problem presented was a word problem form to make the problem focused. In other cases, we have taken a problem directly from the experience of the student or some important problem from the application-oriented students. The trick is to make the problem easily understood.

Contrary to the opinion of some, many students react very favorably to hard problems that are presented in an intuitive way. Also contrary to opinion, many students can deal with higher-level mathematical concepts, particularly when developed in the context of a real problem. In a tightly controlled classroom situation, students can explore issues in:

1. Floating point arithmetic
2. Numerical error and conditioning
3. Functional approximation and interpolation
4. Linear and non-linear differential equations
5. Quadrature
6. Optimization
7. Experimental data techniques
8. Tables and interpolation

It is worth pointing out that traditional undergraduate mathematics courses are open to much criticism because the courses are taught with an emphasis on formulas and theorems but independent of meaning. The richness of calculus, for example, is in its applications. Even with the current *reform* underway in undergraduate mathematics, we are unlikely to see Bishop's criticisms[8] answered. The upshot is that undergraduate mathematics is not computationally oriented and hence inappropriate for computational science.

We are also exploring the possibilities of including aspects in high school mathematics and science. In this case, just simply asking the question of how good the built-in trigonometric functions are might be sufficient to keep a high school class busy all semester. Simply taking away the student's calculators and making them deal with the tables of values

[8] This is the equivalent of a minor

is a valuable exercise in error analysis and interpolation without high-powered mathematics being required. For example, when asked for the value of π, the value most often given is 3.14. How good is that value?

The guidelines for development individual problems are:

1. The problems should be easy to grasp and capable of full analysis.
2. The solutions should be intuitive at the outset so that the student can propose better solutions as they understand more.
3. The students should address several small problems extensively rather than one or two large, complicated problems.
4. The course should expose common failures caused by commonly used techniques when applied inappropriately.

There is a large number of quite simple but important problems which fall into these guidelines. For example, one can trace the history of the computing of π from Archimedes to the current supercomputing efforts which have recently been so widely touted. In the process, the students learn about series, acceleration methods, finite difference algebra, limits, and coding techniques not to mention a healthy dose of round-off error and conditioning analysis. All of this can be done with little or no reference to anything above an intuitive grasp of limits—or it can be done with the most advanced concepts. The point is that one can use this one problem across a broad spectrum of students—high school to Ph.D.

When working actual physical problems, such as we have done with our class, we have found that the following rules make life easier:

1. The working groups must be small and multidisciplinary.
2. The homework should emphasize graphic/visualization techniques over printing out and poring over lists of numbers.
3. Course materials should emphasize literate explanations of the methods employed and the programs written.

The syllabus developed is being expanded and developed into a series of teaching modules. When completed, these modules will be available from the North Carolina Supercomputing Center[9].

[9]Contact Curtis Edge, Director of Education, North Carolina Supercomputing Center, Research Triangle Park, NC, 27709. e-mail: edge@ncsc.org

A particular aspect of programming needs to be dealt with: the tendency to think of programs as something beyond explanation. In our syllabus development work, we are employing literate programming techniques pioneered by D. E. Knuth[33]. We are using the *FWEB* program written by John Krommes at Princeton[23]. This approach has proven viable.

4.1.3 The Graduate Program

For the graduate student who does not have a background commensurate with the outline above, most schools would be able to add sufficient courses to fill the gap, assuming that the students a sufficient science background. Clemson offers the usual fare of theoretical and applied courses of interest to the computational scientists. Some are advanced architectures, compiling, computability, computational complexity, operating systems, and parallel and distributed processing. These are—directly usable, subject to the criticisms given earlier, as are many of the topics in software engineering, database management, and graphics.

While many of the scientific questions posed by the Grand Challenges are not directly related to computer science research, some are: *e.g.*, the Human Genome project. The history of genome decoding as a coding theory/formal language problem is quite long. Visualization, by its very nature, is tightly tied to current graphics research.

However, there are many topics which have been hinted at in this paper which perhaps need to be explicated. We list three obvious and active areas of computer science research which have direct applicability to computational science: foundational issues, software engineering, and programming languages.

4.2 Research Issues

Research topics for computer science in computational science are many and varied. In this section, we touch on only the three most obvious. Firstly, there are several foundational issues; indeed, there are several deep philosophical issues. Secondly, there is the problem of developing software; we present a case that the current efforts in software engineering are not applicable to scientific software development. Finally, there are several issues about programming languages.

We propose that the program for computer science's contribution in the computational sciences is the sound basis of programming scientific applications and should concentrate on the issues below.

4.2.1 Foundational Issues

One of the problems for computer scientists who are not also mathematicians is the the role of mathematics in computer science. For those not familiar with the history of mathematics, Kline[21] is heartily recommended, if not required, reading. The basic point, however, is that most computer scientists are introduced to formalistic mathematics and not constructive mathematics. The latter, with its emphasis on objects is, much more likely to appeal to an algorithmic view[1, 7, 17, 18, 22, 24, 28].

There are many intriguing questions which are of the mathematical/computational nature. If we pick up on Bishop's program[9], we might say that Bishop did not go far enough for computational science purposes. While we can have large numbers of digits (say in a multiprecision package[3]), the numbers are still finite and bounded. We propose the following program: to develop a sound theoretical basis for deriving computer programs by taking the *computational real* formulation as the specification. Such a program would replace the "finite but not *a priori* bounded" numbers of the computational reals by the "finite and *a priori* bounded" numbers of the machine. The development of a sound understanding of the number systems starts with Wilkinson[36, 37]. The concept of the *Wilkinson set* fits very nicely with the ideas of denotational semantics[4, 6, 25, 29, 31]. This development should be primarily algebraic in nature, adding a level to the traditional algebraic hierarchy. The constructive program might also shift in emphasis in development of numerical mathematics. For example, we can achieve some results by replacing limits with extrapolations. In this program, we might well shed some light on the age-old question of the semantics of a mathematical expression. We might propose that the semantics of the expression is the appropriate numerical programs that compute the expression to a certain accuracy. Here, we use "appropriate" to mean "appropriate to the region of the parameter space under investigation." Seldom does one method suffice for *all* possible subregions of the parameter space.

The last foundational issue to touch on is that of complexity. While asymptotic complexity continues to be important for computer science, there are other issues to address. Asymptotic analysis has been mostly successful in delineating worst case performance. The comparisons are only valid for large inputs, something meaningless in computational science. However, a more important criticism can be leveled: the current scheme does not address *how fast* the algorithms approach their asymptotic speeds. This criticism can also be leveled against the development of numerical codes. New methods and ideas are available and should be explored[35, 10].

4.2.2 Practical Development Support

While foundations have a place in supporting computational science, computer science can address issues in the development tools and techniques for the implementation of models in the heterogeneous environment. In this section, we allude to some concrete suggestions for research. This material is a very short version of [34].

Some areas, such as architecture, operating systems, and graphics, have applications to computer science as well as computational science. We have alluded to the need for problem-solving environments[15] that make use of areas such as computational geometry and artificial intelligence. Even an area such as database management—which we associate more with business systems than scientific systems—has important applications in managing the large volume of data generated in many types of scientific experiments. Two areas should receive special mention: software engineering and programming languages.

The software engineering of scientific systems can be quite different from other kinds of systems. While the concerns are much the same, the method may be different. Scientific models evolve over time; hence, the management of change assumes special importance. The role of the specification is evolutionary and based on analysis. It is also the case that the specification is not open to negotiation. Testing assumes a different dimension since it is often hard to determine what the "right answer" is.

Programming languages are an important part of the development of computational science. We are not just thinking of the eternal "Fortran *versus* C" discussion. The basis of design for most scientific systems is matrix-theoretic. Even problems that are only in a single variable may employ matrices—it is impossible to talk about quadrature without talking about "grids" and matrices. Primitives in computational matrix algebra probably look more like the *BLAS* than one might conclude from a linear algebra text. There are also many special matrix shapes which need to be supported. With regards to arithmetic, there is the ongoing problem of dealing with interrupts and the proper support for *IEEE* arithmetic.

5 Summary

Computational science is an emerging discipline offering opportunities for computer scientists. Computational science is an *interdisciplinary* approach to addressing the *Grand Challenges*, whose solutions are deemed vital to the economic health of the United States. The opportunities for participation in computational science range from the traditional areas of computer science—such as language development, system design, and (non-numerical) algorithms—to decidedly new areas such as software engineering related the the development and justification of scientific programs.

The excitement of computational science is in renovating the scientific research paradigm. There are three goals:

1. To find and eliminate unwarranted assumptions and approximations in models;
2. To correctly marry the appropriate algorithms to the appropriate architecture given a model and its parameter space; and
3. To deal with the complexity and veracity of the programming process.

The computational science program proposes to develop a new approach to science by the principles: physical exactness, guaranteed computability, and bounded errors.

The organizational paradigm is an integrated, interdisciplinary focus on *applications-algorithms-architectures*—that is, the focus is on solving a class of problems rather than generating new pieces which might be fit together into a solution.

The goals for computational science courses are:

1. To create a healthy sense of what computation can and cannot do with respect to scientific models.
2. To instill appreciation of the application-algorithm-architecture nature of computational science.
3. To expose the students to the consequences of not following proper computational principles.

The conduct of the courses reflects the philosophy of the professors as well as the subjects themselves. Our emphasis reflects experiences gained in industry: it is imperative that the students work in *interdisciplinary* groups. It is important that the groups understand that each member has a special contribution based on her/his background. We also emphasize the requirement for successive refinements to the original model—something students tend not to understand until they see the answer unravel before them.

The concept of "laboratory" has meaning only when teaching "laboratory techniques." These techniques range from using electronic mail and programs such as *ftp* to benchmarking and running numerical experiments. We emphasize throughout the course, not just in the laboratory exercises, that self-criticism and self-analysis is an indispensable part of computational work.

While computational science is not for every student and researcher, there are plenty of exciting problems to be addressed. It is time to make computational science part of computer science—and *vice versa*.

Acknowledgement. The author would like to acknowledge the roles of Dr. Robert M. Panoff of the North Carolina Supercomputing Center and Dr. Daniel D. Warner of the Department of Mathematical Sciences, Clemson University, in developing many of the ideas in this paper. David A. Sykes, one of the author's Ph.D. students, made many major contributions to this paper both as a reader and a sounding board. Finally, the many fine comments from the referees have led to many major improvements and a much clearer perspective.

References

[1] O. Aberth. *Constructible Analysis.* McGraw-Hill, 1980.

[2] ACM. *ACM Guide to Computing Literature.* ACM, 1990.

[3] David H. Bailey. Automatic translation of fortran programs to multiprecision. Technical report, NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035, April 17, 1992.

[4] H. P. Barendregt. *Lambda Calculus: Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North Holland, 1981.

[5] Enrico G. Beltrametti and Bas C. van Fraassen. *Current Issues in Quantum Logic.* Plenum Press, 1981.

[6] E. W. Beth. Semantic construction of intuitionistic logic,. *Kon. Ned. Ak. Wet.*, 19:257–388, 1956.

[7] Errett A. Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[8] Errett E. Bishop. The crisis in contemporary mathematics. *Historia Mathematica*, 2:507–517, 1975.

[9] Errett J. Bishop and Douglas Bridges. *Constructive Analysis.* Springer-Verlag, 1985.

[10] G. J. Chaitin. *Algorithmic Information Theory.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[11] J. P. Cleave. The primitive recursive analysis of ordinary differential equations and the complexity of their solutions. *J. Comp. Sys. Sci.*, 3:447–455, 1969.

[12] Harold T. Davis. *Introduction to Nonlinear Differential and Integeral Equations.* Dover, 1960.

[13] J. J. Dongarra *et al. LINPACK: users' guide.* Soc. for Indust. and Appl. Math., 1979.

[14] Federal Coordinating Council for Science, Engineering and Technology. *Grand Challenges 1993: High Performance Computing and Communications.* NSF, National Science Foundation, Computer and Information Science and Engineering Directorate, 1800 G Street N.W., Washington, DC 20550, 1993. May be obtained via e-mail through *pubs@note.nsf.gov.*

[15] E. Gallopoulos, E. Houstis, and J. R. Rice. Future research directions in problem solving environments for computational science. Technical report, Center for Supercomputing Research and Development, University of Illinois, 1991. Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Aritificial Intelligence for Computational Science, partially supported by NSF Grant CCR-90-24549.

[16] W. Edward Gettys, Frederick J. Keller, and Malcolm J. Skove. *Classical and Modern Physics.* McGraw-Hill, 1988.

[17] R. L. Goodstein. *Recursive Analysis.* North Holland, 1961.

[18] A. Grzegorczyk. Some approaches to constructive analysis. In A. Heyting, editor, *Constructivity in Math.*, Stud. Logic Found. Math., pages 43–61. North Holland, 1959.

[19] Richard H. Hamming. *Numerical Methods for Scientists and Engineers.* McGraw-Hill, 1962.

[20] Rom Harré. *The Principles of Scientific Thinking.* The University of Chicago Press, 1970.

[21] Morris Kline. *Mathematics: The Loss of Certainty.* Oxford University Press, 1980.

[22] C. Kreitz and K. Weihrauch. A unified approach to constructive and recursive analysis. In E. Börger, W. Oberschelp, M. M. Richter, B. Schinzel, and W. Thomas, editors, *Computation and Proof Theory, Logic Colloq. Aachen 2, 1983.*, LNM 1104. Springer-Verlag, 1983.

[23] John Krommes. *FWEB User's Guide.* Princeton University, 1991.

[24] S. Mazur. *Computable Analysis.* PWN, 1963.

[25] P. D. Mosses. Abstract semantic algebras. In D. Bjorner, editor, *Formal Descriptions of Programming Concepts II.* North-Holland, 1983.

[26] C. M. Pancake. Software support for parallel computing: Where are we headed? *CACM*, 34(11):53–64, 1991.

[27] Marian B. Pour-El and J. Ian Richards. Noncomputability in analysis and physics: a complete determination of the class of noncomputable linear operators. *Adv. Math.*, 48:44–74, 1983.

[28] Marian B. Pour-El and J. Ian Richards. *Computability in Analysis and Physics.* Ω Perspectives in Mathematical Logic. Springer-Verlag, 1989.

[29] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[30] Computer Science and National Research Council Telecommunications Board. *Computing the Future: A broader agenda for computer science and engineering.* National Academy Press, Washington, DC, 1992.

[31] D. S. Scott. Domains for denotational semantics. *Proc. 9th ICALP*, 140:577–613, 1982.

[32] SIAM. *LAPACK Users' Guide*, May 1992.

[33] Lisa M. C. Smith and Mansur H. Samadzadeh. An annotated bibliography of literate programming. *ACM SIGPLAN Notices*, 26(1), January 1991.

[34] D. E. Stevenson and R. M. Panoff. Experiences in building the Clemson computational science program. In *Proceedings of Supercomputing '90.* ACM, 1990.

[35] J. F. Traub, G. W. Wasilkowski, and H. Woźniakowski. *Information Based Complexity*. Academic Press, 1988. LC QA267 .T73 1988.

[36] J. H. Wilkinson. Error analysis of floating-point computation. *Numer. Math*, 2:219–340, 1960.

[37] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Wiley, 1963.