Software Engineering Frontiers in Computational Science and Engineering

D. E. Stevenson
Department of Computer Science
Clemson University
Clemson, SC USA 29634-1906
steve@cs.clemson.edu

Abstract-In 1991, the US Congress passed the *High Performance Computing and Communications* bill, commonly known as the HPCC bill, enshrining the Grand Challenges as national priorities. The very nature of these problems require the multidisciplinary teamwork of engineers plus computer, mathematical and physical scientists. But many important scientific and engineering problems are solved daily on workstations—these were dubbed the "petty challenges". Both classes of problem are demanding computational systems although quite different from non-scientific systems.

We review a philosophical background for CSE, using this development to point out how seemingly innocuous decisions made by engineers and scientists can have disastrous results. Hence, software engineers should see CSE as a professional challenge. Our program is based on studying *applications*, the *algorithms* to solve problems arising in those applications, and the mapping of those algorithms to *architectures*. Using *Computing Reviews* categories, we outline the subjects required for understanding CSE systems.

Keywords- Computational Science, Software Engineering, Scientific Programming.

1 Introduction

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 and engineering (*CSE*) as embodied in the Grand Challenges and *(ii)* the National Research and Educational Network (*NREN*)—the so-called "Information Superhighway". 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 world-wide impact. The current goals of the *HPCC* are published yearly[2]. The Grand Challenges acknowledge what computational practitioners have long known: problems that are analytically intractable are approachable by computational means. The focus on computation opens an old argument: why do we want to do things numerically, anyway? By extension, what are the ramifications for software development methodologies?

We review the Clemson view of computational science and engineering foundations. A more definitive treatment is to be found in [8]. The Clemson approach (Section 2) recognizes three components to computational science and engineering: applications, algorithms, and architectures. The conduct of computational science and engineering is interdisciplinary and team-oriented. Information concerning the educational aspects of computational science and engineering can be found in [8, 7, 6]. In Section 3, we analyze the conduct of computational science and engineering with an eye to determining where software engineering fits. Section 4 looks at system development directly. Finally, Section 5 considers current efforts to educate all members of the computational science and engineering team.

2 The Challenge of Computational Science and Engineering

This section reviews foundational and philosophical issues. It is meant to address broad-based concerns and changes in paradigms. A fuller explanation can be found in [8].

2.1 The Environment of Computational Science and Engineering

Computational science and engineering is characterized by the use of computation to provide detailed insight into the behavior of complex systems. Computational science and engineering uses computational methods to conduct experiments which are either too expensive or, in fact, impossible to conduct in the real world. Computation is necessary due to the nature of the models: in [8], we justify this need. 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 ubiquitous nature of high performance computing engines coupled with the need to explore complicated nonlinear systems makes computation immensely important in modern science and engineering. The simulations can be used to provide unique insight into physical or hypothetical processes. In order to improve this capability, the full power of computing technology must be available to the scientist and engineer.

The proper foci of computational science and engineering are proper modeling and correct computation. The modern view of science recognizes an interplay between theory and experiment. However, the mathematical foundations for modeling are not generally taught engineers and scientists. Computational science and engineering is, by its very nature, constructive, whereas most students only learn formalistic mathematics. It is also the case that few scientists and engineers have a broad background in numerical methods.

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 properly taken into account in scientific formulations[8]. 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 computational science and engineering. 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. On the other hand, the computational methods employed are far harder to validate in the context of the problem.

Perhaps the greatest impact of computational science and engineering to date is that it 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 typically requires 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.

In summary, we need to realize that much of the engineering, mathematics, and science we were taught is suspect in the conduct of computational science and engineering. We must be clear that it might not be appropriate to use computers on this old science.

2.2 Physical and Non-Physical Models

Computational science and engineering places science and engineering first and makes sound modeling the basis. The model reflects the scientist's or engineer's understanding of the system. Almost inevitably, the models incorporate assumptions about how a system operates. These assumptions often require the use of mathematical approximations. We make two useful distinctions to classify these assumptions.

Some assumptions about the behavior of the system are required. This is caused by incomplete knowledge of the system. Some systems studied by scientific research are actually *hypothetical*—we are attempting to discover the real mechanisms. Engineering hypotheses usually arise when a system is in the design stage. In either case, we call such assumptions *physical* since they are attempts understand the physical structures. We would expect all assumptions open to validation procedures. On the other hand, we often see assumptions introduced solely for mathematical tractability. We term such assumptions *non-physical*. In [8] we give an example of the insidious nature of non-physical assumptions in the context of the simple pendulum. The lines between *physical* and *non-physical* may be blurred, but the distinction is useful.

"Classically derived" models are rarely free of non-physical assumptions. Such models include non-physical assumptions to produce closed-form solutions. One may therefore be left with a nearly exact solution of approximated—and perhaps unrealistic—models. In [8] we actually give an example of how non-physical assumptions completely change the model: in that case, the pendulum is changed to a torsional spring. Anecdotal evidence shows that often these models may give unsatisfactory results when used in a computational setting. An alternative is to reformulate the models by removing these non-physical assumptions. Unfortunately, these newer models generally have no closed form and generally are very hard to solve numerically.

Since we see computational science and engineering as an interdisciplinary endeavor, there is a need to merge the methods and viewpoints from the individual disciplines involved. 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[5]. The reliance on computation in computational science and engineering opens a very important question: How much of ordinary mathematics is usable in the computational world? The constructivity question has been addressed, for example[1]. Unfortunately, constructivity is not the same thing as "compute on a workstation." 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. However, computation did point the way in this case as there are many models that exhibit chaos analytically; it was not found earlier because there was no tool to make us look.

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. The following problem scenario is very common in computational science and engineering.

> Take a string and tie it around the equator of the Earth with radius R. Add l feet to the string. How high (x) 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, using college algebra and trigonometry, can be found very quickly

$$\begin{aligned} R^2 + x^2 &= (R+h)^2, \\ x &= R\tan\theta, \\ \tan\theta - \theta &= \frac{l}{2R}, \end{aligned}$$

where θ is the angle between the Earth's radii at the point of tangency and the tower. Unfortunately, the answer can only be found by solving an implicit trigonometric equation and then solving a quadratic equation. This 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. It turns out that one can get about 21 digits of accuracy out of 28 digits[1] of precision on a *Cray.* The naive solution coded in double precision is not how to obtain those 21 digits. To find these 21 digits takes a significant amount of work, involving many test programs and a goodly bit of experimentation and testing. Then, the problem of *validating* these digits is equally challenging.

[1] We have since done much better.

Assuming that models (*application*) have been properly formulated does not guarantee that the appropriate numerical method(s) (*algorithms*) or the optimal choice of architecture are chosen for the computational models. We use the term *architectures* to encompass both hardware and software. Software architecture include compilers, libraries, editors, graphics programs and the like. *Heterogeneous hardware systems* complicate the picture even more. 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 optimal on a von Neumann architecture. Experience with distributed algorithms, 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 for many different architectures have been hinted at in the literature; *e.g.,* the evolution of `LINPACK`. There is also a problem with an exploding number of versions: often differing only in architectural details. Is this a hardware, a software, or an algotecture problem? An *algotecture problem* is a problem arising from the attempt to put a particular algorithm on a particular (hardware-software) architecture.

In summary, new concerns arise for the practioner. Writing down the mathematics and perhaps gaining an analytical expression is not the same thing as being able to compute the numbers. And computing numbers is not the same as justifying the number of correct digits.

3 Initial Analysis of Software Engineering Concepts for Computational Science and Engineering

We now turn to several natural question that arise: (i) How are programming and program development different in computational science and engineering? and (ii) How do these differences change our thinking about software engineering? It is most natural to start our analysis with the new "life cycle" typified by applications-algorithms-architectures.

3.1 The Application

For the most part, we assume the application is specified in analytic (continuous) terms. This may be in the form of an ordinary or partial differential equation, integral equation, some sort of optimization criteria, *etc.* This may not always the case: there are many interesting problems that arise that are discrete.

Experience shows that such model specifications may already be in trouble. If the model has had too much detail abstracted away—usually with with classical mathematics in mind—the resulting formulation may be incorrect according to computational science and engineering standards. Even if care has been taken, any subsequent manipulations must be done with some care. Many models are of low asymptotic complexity, say N^5, but N is very large, say 10^9 or more. Over-abstracting the model before such things are determined may eliminate a desirable hardware platform unnecessarily.

Despite these caveats, it is clear that the concept of *specification* is fulfilled by this formulation. What is interesting is that this specification becomes embedded in the subsequent models—the models that are derived based on this initial computational model. We are then faced with a real problem: the understanding of the model ultimately depends on our ability to understand the resultant computational model and programs.

3.2 The Algorithms

The most obvious things about numerical algorithms are the vast number of them and the huge literature surrounding them. On the whole, though, what one needs to know about a particular algorithm is probably not in the literature: we most want to know when to use what. There are several aspects of deciding what algorithm to use when. Firstly, most algorithms are used *on* something. For example, ordinary differential equations of the most garden variety can be solved with something like Runge-Kutta-Fehlberg algorithms. However, there are classes of problems—stiff systems, for example—that require special algorithms. Whether or not a system is stiff, for example, depends on the *application*.

A second area of difficulty in numerical algorithms can be illustrated by the problem associated with summation: the problem is called *cancellation of significant digits*. While most operations on finite arithmetic are well-behaved, addition has a problem with losing significant digits when summing over a list of numbers. This can be alleviated by sorting the numbers low to high and adding. But one need not always do this and to do so would waste time. A simple trick—called "Gill's Device"—removes the need for the sort. There are many "tricks" that represent correct coding because the trick is backed up by a theorem—and practical experience. The problem is that these tricks are either learned through experience by word-of-mouth; they almost never appear in a text.

3.3 The Architecture

The development of exotic hardware has been driven more by computational science and engineering concerns than any other single concern. There are many hardware paradigms available to the scientific researcher, some more exotic than others. There are specialized machines that are used for specific computations. One of the original Grand Challenges—quantum chromodynamics—has led to many specialized, *ad hoc* architectures. The designer of computational science and engineering systems must anticipate that such systems must run on *any* available architecture.

On the software side, there is the issue of libraries: the necessary but unwanted proliferation of libraries due to architectural details. In general, though, there is a pool of well-developed libraries available from around the world. What is different about many of these libraries is that a "library" is often a solution mechanism for only one problem or special cases of a particular problem.

To see the depth of the problem, consider spend a month monitoring `sci.math.numeric-anal`, the news group for numerical computation. The vast majority of the queries are of the form "I need a `C` routine that runs on a x using y's algorithm."

4 Understanding the Development Cycle

As far as program development is concerned, the development of scientific software goes through the same phases as other software. However, the activities and concerns are far different.

4.1 The Specification Phase

For this part of the discussion, there is no real distinction between production and research models. The first distinction is that there is no negotiated specification: Nature does not have to negotiate with anyone. Whether modeling a gear train or looking for the top quark, the modeling question is to develop something by which we can correctly predict the behavior of the physical system. Over time, we see the program and the model changing as the underlying phenomena are understood better. The problem with this is that the program encapsulates what we know about the phenomena. Therefore, the specification becomes part of ("embedded") in the system in very intricate ways. Indeed there are programs that are themselves the only knowledge about certain phenomena. Before examining this exact point, we need to digress and decide what *correct* means.

Because we are dealing with physically exact but mathematically inexact models, we must deal with *errors*. Of course, this term in this context is not pejorative. In principle, we want to make the difference between what we observe and the actual values as small as possible. Now, these values are usually start out as numbers, but may be filtered through graphics. For example, the program computes a wave length that is then displayed using a color map. The color is then compared with the actual running of the system. Such a procedure is probably not all that exact. Converting the color to a frequency and comparing that with the computed number may not be all that accurate either. For most numerical work, we define the *relative error* between a number x and its computed value $\hat{x}$ as

$$\frac{|x-\hat{x}|}{x}.$$

unless $x = 0$, in which case we use the *absolute error* $|x-\hat{x}|$. There are also other errors that arise from infinite computations made finite, or *truncation error*.

Now, to understand how complex the embedding of the specification is, consider the following. Suppose we have a model of a pendulum as described by

$$mL\frac{d^2\theta}{dt^2} = -mg\sin\theta,$$

where m is fob mass, L is the length of the massless arm, g is the acceleration due to gravity, and θ is the angle from the vertical. Suppose we see a difference between observations of the real period and the model predicted values. What are we to conclude? The error could be from any or all of the following:

- m was measured wrongly,
- g is not known accurately enough for the model,
- The time period was incorrectly measured,
- The mass of the arm is too great to be ignored,
- There is too much friction to be ignored,
- The value of $\sin\theta$ is computed imprecisely or wrongly,
- Not enough precision is being used,
- A wrong method for solving the differential equation is used, or
- An "okay" method is used but improperly programmed

to name but a few. Any changes to the model caused by any of the above become incorporated in the model. Hence, the errors are embedded. Without sufficient care, the model soon becomes worse than useless—it is totally misleading. It is also clear that "testing" is a different process since it is harder to define what the software is supposed to due independent of the model validation.

To illustrate how tricky these things are, we turn to a war story. In a particle beam experiment, the statistics are used to find the mean of the energy. Properly, the energies were accumulated in floating point (say *energy*) and the count of events (say *counts*) was accumulated in an integer. The value of *energy* was saturated[2] before the program had processed all inputs. This made the average energy too low and completely changed the outcome of the experiment.

4.2 The Design Phase

We can now turn our attention to the design of the algorithms. Unlike non-numerical problems, the tendency in computational science and engineering is to use reliable software obtained from reliable sources and modify it as needed. Here is an exercise. Use *netlib* via its email interface or through the *xnetlib* server and ask for the term "zero finder." There are two listed: one for Bessel functions and one for Brent. Where are all the rest? The ones we usually give our students are not there since Brent is regarded as far superior to any others. Checking numerical texts does not turn up any references to a zero finder by Brent—until you run into a text Forsythe, Malcolm and Moler[3]. Routine like Brent's *zeroin* are often buried in the literature and there is no particular guarantee that a "classically-trained" mathematician or numerical analyst is aware of such routines. So, one measure of design complexity is whether or not reliable software is available from sources like *netlib*, *IMSL* (a commercial package), or vendor-supplied. (Let's not forget the *Pentium*).

Reliable computation has many aspects. For example, the simple question of finding the midpoint between two numbers is difficult to design in general without introducing "software bloat"[4]. For example, if we want $(a+b)/2$ we are faced with the problem of the relative size of a and b. Analytically,

$$\frac{a+b}{2} = \frac{a}{2}+\frac{b}{2} = b-\frac{b-a}{2} = a+\frac{b-a}{2}.$$

These are not the same computationally. For example, if a and b are very large, $a+b$ can overflow but $a/2+b/2$ cannot. On the other hand, if $a \gg b$, then $a+b \approx a$ and if not done properly the relative error in the answer is quite large. The natural reaction is to have decision code

[2]Saturated values lose small increments due to roundoff.

to check for the cases. Thus, a thorough understanding of the operating resolution of values is required.

Another aspect of design is the problem of terminating *while* loops. Most analytically derived conditions specify the termination of loops by taking the difference of two successive iterates and determining if these are within tolerance—something of the order of a few machine epsilons. What's wrong with that? The problem, of course, is the arithmetic. The difference of two nearly equal numbers is the very definition of *catastrophic cancellation of digits.* The answer, then, is that some other method should be found.

The last design bugaboo is computational complexity. Here the problem is insidious: what may work well at one stage of the model's life may not work later. This manifests itself in many of the Grand Challenges. Small incarnations of the problem may be slow and large incarnations completely impractical due to computation times. The trade-off here is often between accuracy and speed. But computational science and engineering policy is to understand this situation and still be able to say how accurate the solution is.

4.3 The Implementation Phase

The implementation of the code for the model should be fairly straightforward—but may not be so. Of course, there are many decisions made during coding that have grave consequences on the actual answers produced by the computer model. There are also the unnecessary arguments of language and operating system. Unlike many non-numerical programs, numerical programs are expected to run on all currently available architectures. This is, of course, almost impossible to do if one considers just the three usual architectures: SISD (standard von Neumann), SIMD(vector processor), and MIMD (distributed memory processors). Even von Neumann architectures have been converted to MIMD by use of systems like PVM. The implementation issues, in order, are (i) correctness, (ii) performance, and (iii) portability. It is important to emphasize their point about correctness and performance: correctness is first. This has not always been kept in mind: the literature often emphasizes speed only. Because of the correctness issue, we may expect a tighter relationship between implementation and testing. We would also hope, in the long run, that program verification can play a role in implementation; surely, though, this is a research issue.

There are many implementation issues that are unique to numerical methods. Many of these issues turn on the fact that two algebraically equivalent statements are not computationally equivalent. Another class of problems is related to slowly converging series. A classical illustration of both problems is Leibniz' series:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \pm \ldots.$$

This is a conditionally convergent series. If one writes a naive program that sums the series forwards (as written above) for 10,000 iterations, the relative error ($\pi/4$-sum) is 10^{-5}. This indicates that there are four correct digits in the answer. If one sums backwards, it only takes 5,000 iterations to get the same accuracy.

4.4 The Testing Phase

Testing in computational science and engineering is more than running test cases on the software to determine whether or not it produces an apparently correct answer. We distinguish two requirements: software verification and model validation. Model validation is those activities that attempt to determine the degree of coherence between prediction of the model (as embodied in the program) and the observed behavior of the physical system. It is not always possible to test conditions directly against the physical system: we certainly do not want to create a nuclear winter to test that model. Now we again see the interplay discussed in the specification phase in Section 4.1. Therefore, testing is highly unlikely to be automated easily.

Software verification plays a different role in CSE systems; non-numerical systems tend to have different types of errors. For example, integer arithmetic is exact within the machine and can be counted on to overflow with unsensible arguments. IEEE floating point, which is used by most vendors today, normally does not interrupt a program on overflow or underflow, although one might want to enable interrupts on some but not all statements. IEEE can also be made to round in one of four ways on each individual statement. We have seen an example of how order can drastically effect outcome. Finally, recall the earlier discussion about outputs: one must justify the number of digits, not just read them.

4.5 The Maintenance Phase

At last, something that should be as we expect—Well, almost. There are two interesting scenarios. Firstly, suppose that one of the computer manufacturers on which the current model runs changes the computational environment in some way so as to disturb the arithmetic. This could be changes to the algorithms making them more (or less) accurate or altering the interrupt mechanisms. In effect, the model must be re-validated since the programs now potentially produce different results.

The second interesting possibility is the introduction of a new computer paradigm: for example, consider the next Kendall Square machine or a new MasPar. If this is a research model, the scientists often will want to port the code to this machine. In the worst case, the arithmetic algorithms on the new machine will not mimic the arithmetic on the older ones. Since we normally would have to re-implement the code on this new machine, we will be faced with trying to justify the new implementation against all the others *and* the physical validation as well. This task is generally more complicated than just porting and finding annoying changes of program names.

5 Finally...

Much of the material discussed in this paper has been used by the author in various computational science and engineering courses both at Clemson and at other places. The material is also used in NSF-sponsored workshops. In all these instances, the makeup of the class is multidisciplinary—engineers, scientists, mathematicians and computer scientists. Convincing researchers to concentrate on correctness is usually easy, but most non-computational scientists are unsure what computational correctness is or how to achieve it. Non-computer scientists are also unlikely to have a realistic vision of program development. Computer scientists, on the other hand, are often math-phobic or science-phobic or both. Much more detail is available in [8].

All of this leads to difficulties in the development of scientific and engineering models. We have tried to describe those parts of the problem that can be positively impacted by computer scientists and software engineers. It is our hope that these two groups realize there are important and challenging problems in computational science and engineering and that their participation is important.

References

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

[2] Federal Coordinating Council for Science, Engineering and Technology. *Grand Challenges 1994: High Performance Computing and Communications.* NSF, National Science Foundation, Computer and Information Science and Engineering Directorate, 1800 G Street N.W., Washington, DC 20550, 1994. May be obtained via e-mail through *pubsnote.nsf.gov.*

[3] G. E. Forsythe, M. A. Malcum, and C. B. Moler. *Computer Methods for Mathematical Computation.* Prentice-Hall, 1977.

[4] W. Kahan and J. Palmer. On a proposed floating-point standard. *SIGNUM Newsletter, Special Issue*, pages 13–21, October 1979.

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

[6] John R. Rice. Academic programs in computational science and engineering. *IEEE Computational Science & Engineering*, 1(1), 1994.

[7] Ahmed Sameh and John Riganati. Computational science & engineering. *IEEE Computer*, 26(10):8–12, 1993.

[8] D. E. Stevenson. Science, computational science, and computer science: At a crossroads. *Comm. ACM*, 37(12), 1994.