

LA-UR-04-0388

Approved for public release;
distribution is unlimited.

Title:	The Coming Crisis in Computational Science
Author(s):	Douglass Post
Submitted to:	Keynote Address Proceedings of the IEEE International Conference on High Performance Computer Architecture: Workshop on Productivity and Performance in High-End Computing, Madrid, Spain, February 14, 2004



Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

The Coming Crisis in Computational Science

Douglass E. Post

Los Alamos National Laboratory, Los Alamos, NM 87544

LA-UR-04-0388

post@lanl.gov

Abstract

Computational science faces three major challenges: “The Performance Challenge,” “The Programming Challenge” and the “The Prediction Challenge”. The exponential growth in processor speed and the advent of massive parallelization have increased computing power by a factor of 10^{13} since 1945. This has enabled scientists and engineers to tackle important problems of unparalleled size and complexity. However, the complicated architectures of these new platforms have made programming more difficult. Furthermore, much of the improved predictive power has been achieved by increasing the complexity of the application models and algorithms. This has raised the level of the challenges associated with developing and using the resulting large, complex computer codes. As a community we are meeting the first challenge—“The Performance Challenge,” but are not doing as well with the other two challenges—“The Programming Challenge” and “The Prediction Challenge.” Computer capability appears likely continue to grow exponentially in the near term. On the other hand, crises loom for programming and prediction. For the “Programming Challenge,” even short programs are often difficult to write for massively parallel platforms. The time scale for developing large-scale applications is often longer than the life cycle of a single platform architecture. Porting applications to new platforms is difficult and challenging. Existing programming tools are inadequate for rapid code development and optimization. Many, if not most, application codes achieve only a small fraction of the potential peak performance. The High Performance Computing Community must make programming easier, or at least no harder, as it builds ever more powerful—and complex—computers. With regard to the “Prediction Challenge,” computational science does not have the predictive reliability of traditional methodologies such as theory, experiment and engineering design. The results of many major computer applications are often wrong or are misinterpreted, sometimes with disastrous consequences. Computational science must mature as a field if it is to become a reliable methodology for addressing important problems. History indicates that it takes time—and quite a few major and possibly dramatic mistakes—for new methodologies to mature. Such major mistakes are occurring now in computational science. Just as other disciplines have learned from their mistakes, we, as a community, must analyze our mistakes and successes and adopt the “lessons learned”. The Computational Science community must improve the predictive capability of application codes if computational science is to become a useful tool for solving society’s problems. A key figure of merit is the “time to solution”, the time between the identification of a problem and the delivery of validated and analyzed computational solution. For the reasons quoted earlier, the “time to solution” is growing in many cases, not decreasing. Reducing it requires that we address all three challenges.

1. Introduction

Computational science—the use of large-scale computers to address and solve important technical problems—is becoming an everyday tool for design and analysis of complex technical issues. Applications include scientific research, engineering design, policy analysis, training and emergency response and environmental analyses. Computational science has the potential to address complex issues with a degree of realism that has heretofore only been imagined. This exciting and very important —indeed revolutionary— potential is due to the enormous growth in computer power (speed and data storage) over the last 50 years. This growth shows no sign of slowing in the near term. Yet computational science is a very new and immature discipline. It has not achieved the level of maturity of traditional methodologies such as experiment, theory, engineering design and conventional policy analysis for solving problems.

At least three distinct challenges face the computational science community. First exponential growth in computer power gives us greater ability to tackle difficult and more important problems. But as computers have become more powerful, they have also become more complex. Simple computer architectures have evolved into massively parallel structures with very complex designs and connections. This expanding computer power—larger memory and data storage, and faster processing speed—is enabling very large application programs to treat many very complex and strongly interacting effects. Climate models now include models of dozens of effects where before they included only a few.

These developments lead to three distinct challenges:

1. “The Performance Challenge”: Designing and building high performance computers.
—> Complex Computers
2. “The Programming Challenge”: Programming for complex computers.
—> Complex coding
3. “The Prediction Challenge”: Developing codes with complex physical models that are truly predictive.
—> Complex applications and codes

The exponential growth in microchip processing power described by “Moore’s Law” together with the concomitant increase in memory and disk speed and size and the advent of massively parallel platform architectures have resulted in a factor of 10^{13} improvement in computer processing power since 1945. This expansion of raw computing power is enabling computational science to address many important problems with a degree of realism and fidelity that were only dreamt of ten or twenty years ago. However, the increased complexity of computers has made programming for them more difficult and time consuming. Optimizing code performance is becoming more difficult but the performance analysis and debugging tools for massively parallel platforms are still in their infancy. Programming models (MPI, OpenMP, HPC, etc.) have evolved slowly and vary among platform vendors and architectures. Developing ways to reduce the difficulty of programming high performance computers for optimal performance is a key requirement for computational science to advance.

The main topic of this paper is “The Prediction Challenge,” the challenge of successfully developing codes with complex scientific and engineering models that can make accurate predictions or analyze data correctly. Based on surveys of many technical code projects and case studies to develop “lessons learned”, I can identify three major “lessons learned” (and many more slightly less important ones) that need much more emphasis by the computational science community if scientific and technical computational results are to have sufficient credibility for

the problem at issue that it is worth the expense to build the platforms, program the application codes and apply them:

- Verification
- Validation
- Code Project Management and Quality

Every code consists of models of real effects in nature and mathematics. First, the code must solve the models correctly. The solution algorithms must be applied correctly. The code needs to be free of bugs that significantly affect the results. If the code is not mathematically correct, then any conclusions derived from the code are likely wrong. Second, the models in the code must represent the real world with sufficient accuracy that the code predictions and analysis provide a valid basis for decisions. The models must be checked with experimental data for the regimes of interest. Third, the development of complex, large codes is a complex undertaking itself. Now the development process often involves teams that can be large as 20 or 30 staff, or even larger. The teams need many different skills: science, programming, computer science and computational mathematics. Twenty years ago, most scientific code development teams were much smaller and the range of required skills much less broad. The project has to be organized so that the team members developing the code know what they are trying to accomplish, how to work together productively, what programming and physical models are appropriate, how they will go about developing the code, how long it should take, what resources they will need, what constitutes success and who is customer. Unfortunately, many computational science projects are seriously deficient in one or more of the three areas highlighted above, and the results of those codes are therefore often of little value. Even in the cases when the code is credible, it may have been applied inappropriately. The traditional values of independent assessment, confirmation and repetition of results is missing. Since these are necessary conditions for credibility, it is difficult for an outsider to judge the validity of scientific computational predictions and analyses with the result that computational science does not have the credibility of the more mature problem solving methodologies of theory, experiments and engineering design.

To summarize, there are three major challenges for Computational Science. The High Performance Computing community is successfully meeting the first, “The Performance Challenge”—developing high performance computers. The second—“The Programming Challenge,” involves both the High Performance Computer development community and the computer science community. Both communities must find ways to reduce the difficulty of developing application codes for today’s and tomorrow’s High Performance Computers. The third—“The Prediction Challenge,” primarily involves the computational science community. We must improve the predictive capability of these increasingly complicated programs. A key measure of the effectiveness of computational science is the “time to solution.” This is the time required to conceive and develop a validated computational solution to a problem. The time includes the calendar time required to develop and deploy a computer platform, develop the application, obtain validated solutions and analyze the solutions. From the perspective of the sponsors who need accurate answers to important problems, this is the critical issue. A faster computer platform may run problems more quickly, but it takes a lot longer to develop a code for the faster platform, the time to solution may be larger, not less. Reducing the time to solution requires addressing all three Challenges: Performance, Programming and Prediction. This is the goal of the Defense Advanced Research Projects Agency (DARPA) High Productivity Computing Systems (HPCS) project. The HPCS project has focused initiatives in all three challenge areas involving major high performance computing vendors (IBM, Cray and SUN), the

computer science community and the technical code development community and has the stated goal of reducing the time to solution.

2. The Performance Challenge

Computer power measured in Floating Point Operations per second (FLOPs) has grown exponentially from about 10 FLOPs in 1945 to about 35×10^{12} FLOPs in 2004. This expansion in capability has been achieved by a combination of increased processor speed (characterized by “Moore’s Law”) and improvements in computer architecture, networks and data storage. Greater processor speed has been achieved partially by technological innovation with electronic switches, starting with relatively slow mechanical relays to vacuum tubes to discrete component transistors to integrated circuits. Present processors have millions of transistors and other components on a single chip with multiple arithmetic units. The clock speeds already are in the Giga-Hertz range. The feature size is a fraction of a micron. At some point within the next 40 to 50 years, Moore’s law will saturate due to finite size of atoms and molecules, the irreducible thermodynamic minimum heat associated with a bit of information, quantum interference between adjacent components, etc. However, all indications are that these limits on processor speed won’t be reached for the next couple of decades (Frank, 2002).

Data storage capability has largely kept pace with processor speed. Mechanical relays and vacuum tubes were replaced by magnetic cores followed by transistors and capacitors. Now Gigabyte memories with access rates of several nanoseconds or less are common. Data storage capacity with 10,000 rpm rotating disks is approaching a terabyte per disk.

This exponential growth appears likely to continue for the near term. Processing capability has also been accelerated by the use of many processors operating in parallel. Computers with as many as 8,000 processors are now in operation, and ones with 100,000 processors are under development. It is reasonable to expect a 100 TeraFlops computer by 2005 and a peta-flops computer in the 2010 time-frame. While the capability of computer platforms has been increasing exponentially, the cost per FLOPS has been dropping rapidly. Indeed the cost of the largest supercomputer has remained in the \$100M range (in 2000 \$) for over 40 years.

These achievements have been remarkable and possibly unique in the history of technology. One can buy a computer with processing capability, memory and disk storage at the local office supply company for around \$1000 that is as powerful as the biggest and most capable computer available for any amount of money in 1990. In 2005, the PlayStation 3 will contain a TeraFlops computer and cost about \$200.

Although each significant advance in computer power involves technological innovation, it appears to everyone that computer capability will continue to grow exponentially, at least for the next 10 to 20 years. Predictions beyond that are hard to make. Many new technologies offer promise, including advanced materials, optical logic units, superconducting elements, and the ultimate, “Holy Grail”, quantum computers. Although no technology grows exponentially forever, there appear to be no near term limits for computer capability.

This technological innovation has strong economic drivers. The market for faster processors and networks and larger memory is immense. High Performance Computing is a very small part of that market, but will continue to benefit from the progress driven by the whole computer market.

The “Performance Challenge” is being met. The capability it provides for addressing the important technical problems humanity faces is tremendous. However, realizing this capability

leads to the second two challenges: “The Programming Challenge” and “The Prediction Challenge.”

3. The Programming Challenge

The programming challenge is daunting and continues to grow. As noted in section 2, modern computers contain hundreds to thousands of processors linked together in complicated networks, with local and distributed hierarchical data storage. The next generation of high performance computers will have 10,000 to 100,000 processors. Success requires that we be able to rapidly develop codes that will run efficiently on these complex platforms. A modern application code will often have millions to billions of computational cells. It can contain several hundred thousand to several million lines of code. It consists of dozens of complex strongly interacting components and modules. The code can produce terabytes of output data, and require hours to days to weeks of run time to complete a problem. The programming challenges include: problem setup and mesh or cell generation, domain decomposition, load balancing, run problem management, checkpoint restart, debugging on thousands of processors, configuration management, data output and storage for many terabytes of data, data analysis and visualization, and job scheduling. Achieving good performance requires performance analysis tools to identify blocks in data transmission between processors, race conditions, data transmission inconsistencies and errors, cache use efficiency, etc. All of this is staggeringly complex compared to code development requirements only 20 years ago. High quality compilers are needed as well. MPI and P-threads have become the standard parallel programming models, but OpenMP and other languages such as UPC, Co-Array Fortran, HPC, etc. are becoming common. MPI and P-threads are fairly low level programming methods, not too far removed from assembly language. While the programming challenge for large complicated codes is immense, it is often not much less for short, simple codes that need high speed processing of large data sets. In the latter case, a problem that could be addressed and solved in a few days in the 1990’s may take weeks to months to address with massively parallel architectures due largely to the complications of writing the program.

It is as if there is a second “More’s Law” for programming to complement the conventional hardware “Moore’s Law”. As the speed of computers increases each year, it takes “More” time to develop codes that can run efficiently on the “Moore” complex platforms. Indeed, programming complexity accounts for much of the popularity of high level languages such as MATLAB™. While MATLAB™ is a proprietary code, and a massively parallel version isn’t commonly available, MATLAB™ does allow many code developers to develop powerful and complex codes much more quickly than they could using FORTRAN, C or C++. The code performance with MATLAB™ and other higher level languages may not be as good as with FORTRAN or C, but developing accurate codes rapidly is often much easier and faster.

For prior generations of computers, the development of operating systems, performance tools, debuggers, compilers, visualization tools, etc. was the responsibility of the platform vendor. Now, this software is often developed through Open Source venues, small development companies and university and national laboratory groups. This is leading to problems with the availability and reliability of the software. Who is going to develop the next generation of tools to take the place of CVS, MAKE, Vampir, Enight, Totalview,... for the next generation of supercomputers?

Most of the massively parallel platforms are designed for speed, sometimes specifically designed to place high on the top 500 list by running LINPACK efficiently. Unfortunately, few applications exercise massively parallel computers the same way as LINPACK. Benchmarks and “synthetic workloads” are needed so that vendors can test their platforms with software that places the same demands on the platform as “real” applications.

If high performance computers are to be useful tools for solving real problems, programmer productivity must improve at least as rapidly as the difficulty of programming for new computer platforms increases.

4. The Prediction Challenge

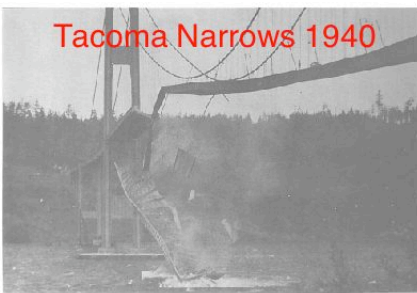
While “The Programming Challenge” is daunting, “The Predictive Challenge” is even greater. The difficulty of programming massively parallel computers is largely a question of efficiency. It may take more time and be more difficult for most codes, but more resources and time will generally result in a working code. However, if we don’t have reasonable assurance that the predictions of a code are accurate and can be trusted or, at a minimum, a good idea of how reliable the predictions are, then the predictions are largely worthless. There is then no reason to spend the resources to run the code, to develop the code or to develop the computer to run it on.

A key part of the problem is that it is often very difficult to judge whether a code result is right or wrong. For experiments or theory, the peer review process for published papers is the filter that separates the wheat from the chaff. However, the existing peer review process for computational science doesn’t work well. When a scientist receives a computational science paper from a journal to referee, he has no definitive way to determine if the paper is correct. He cannot reproduce the results in the paper, and generally he can’t check the important results with experimental data. The most important results typically make predictions for situations for which there is no data. That’s often the purpose of the calculation, after all. Even if he had a listing of the code—and he almost never does—the listing is not enough to determine the validity of a very complex and large calculation. All that the referee can do is to subject the paper to a series of “plausibility” checks. Is the paper consistent with known physical laws? Is the author a reputable scientist, known for careful work? Are the results consistent with other work in the field? Is the simulation validated with data as close as possible the regimes of application? Do the computational methods seem sound and applicable to the problem? Are the original models and fundamental equations correct?

Tragically, these criteria discriminate against new and exciting results, since such results usually cannot be thoroughly checked, and may be wrong. Major new contributions are thus less likely to survive the refereeing process in favor of more modest extensions of previously accepted work.

These criteria are not nearly as reliable and solid as the criteria used for theoretical or experimental papers. A knowledgeable reviewer can re-derive the important formulae in a theoretical paper. Experimental science is a well-established methodology, and important experiments are duplicated fairly quickly. In fact, important experimental results are usually not accepted by the general scientific community until they are confirmed by independent experiments. A similar practice is probably necessary for computational science. “Discoveries” like cold fusion have their moment of fame then fade into infamy as “irreproducible” results.

Reproducibility and the professional integrity of the scientist and engineer is a cornerstone of sound science.



Lessons Learned Case Studies

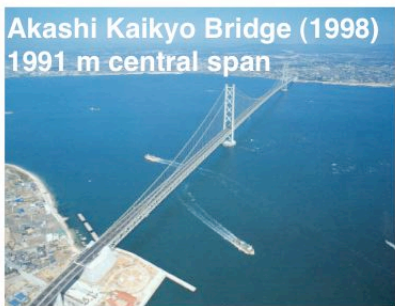


Figure 1. Four stages of suspension bridge development.

engineering technology to reach maturity (Petroski, 1994). These stages can be illustrated using

time



Many things could be wrong with the computational science paper that the referee could not detect. The code could have errors in the way it was written such as bugs, the wrong use of computer or mathematical algorithms, inadequate resolution in time or space, unconverged solutions, etc. Even if the code had few errors, the models and equations in the code could be inadequate or wrong. As Robert Laughlin points out, “One generally can’t get the right answer with the wrong equations.” (Laughlin, 2002). The physical data used in the code may not have adequate resolution or may be inaccurate. The scientist or engineer running the code may not know how to set up or run the problem correctly. He may not know how to interpret the results of the code accurately. Yet referees are expected to judge the correctness of the paper. It’s a challenge the community must address if computational science is to become a mature field.

Important scientific, engineering design and public policy questions are beginning to be decided using predictions by computational scientists. As a community, it is our responsibility to ensure that computational science achieves the same level of reliability as theoretical and experimental science and engineering design. It’s a question of professional integrity. If we don’t meet our professional responsibilities, computational science will not become a credible methodology, and its potential for contributing to the betterment of the human condition will not be realized. If a significant number computer predictions and analysis are wrong, and there is no way to determine which ones are right and which are wrong, people will not rely on them and will not support the development of our field.

What steps have other fields gone through as they matured? In “Design Paradigms”, Henry Petroski traces the history of a number of technology fields as they mature. From his history, I identified four stages needed for an

his example of suspension bridges (Figure 1). The first stage involved the design and construction of early suspension bridges. The designers and construction crews did not know the design limits and were deeply afraid of failures. The designs therefore were very conservative and extensively over-engineered. Although there were some initial failures, the early suspension bridges generally worked. An example is the Széchenyi chain bridge over the Danube joining Buda and Pest constructed in 1840. It stood for 105 years until the Germans destroyed it in World War II. It was rebuilt in the 1990's and stands today.

The second stage involved cautious design improvement and optimization based on the first generation of bridges. The Brooklyn Bridge was constructed by John and Washington Roebling in 1880. It is still standing and carrying a modern traffic load after 120 years.

The third stage involved the development of continually more ambitious designs that pushed the limits of the existing technologies until large-scale failures occurred. The cautious approaches and the deep fear of failure of the prior generations of designers were often forgotten in the enthusiasm to go beyond the achievements of the past. The Tacoma Narrows bridge, constructed in 1940, failed catastrophically due to the excitation of wind-driven harmonic oscillations. Such bridge failures are spectacular. Almost everyone who reads this paper has seen the short movie of the galloping Tacoma Bridge as it bucked and moved in the wind until it collapsed into the water. The civil engineering community studied and analyzed the causes for the failures, then developed solutions that became part of the design methodology for all future suspension bridges.

Advancement to the fourth stage—that of a mature field—is based on the development and adoption of the “lessons learned” from the failures and successes. The field of suspension bridge design and construction is today a mature field. Very large suspension bridges are being built, such as the 2 kilometer span Akashi Kaikyo Bridge in 1998. A measure of the maturity of a field is the level of professional integrity of people in the field. Suppose that a government agency puts a prospective new bridge out for bid with the hope that they can spend \$50M and have a bridge in 2 years. When the bids come in, the lowest bid is \$100M with a construction time of 4 years. If the agency tries to convince the lowest bidder to do job for \$50M in 2 years, the bidder will walk away from the job, rather than build a bridge that will almost certainly fail. The industry knows how to build safe bridges. It can predict how long it will take and what it will cost to build a bridge. When was the last time that you know of when a technical software project manager walked away from a scientific code project he thought would take 4 years for 15 people to complete after the sponsor told him that the project had to be completed in 2 years with only 9 people? When that happens our field will begin to be mature.

I assert that computational science is in the midst of the third step on the path to maturity. The first generation of computational scientists used the supercomputers of the 1950's, 1960's and 1970's. They developed and used codes to analyze data, design nuclear weapons, model supernovae, conduct engineering analyses, etc. Computational science was a new field and everyone was very aware that it had limitations. Due to restrictions in memory and processing speed, the problems generally did not have adequate spatial or temporal resolution and the solutions were often not converged. Often only very approximate models were employed for the problems being addressed. Nonetheless, computational tools were a step forward over existing analysis tools, and—used with caution and careful verification and validation—produced better answers than other methodologies.

As computers became more powerful, the DOE and the NSF established “supercomputer” centers in the US in the 1975 to 1985 timeframe to provide supercomputer

capability to the academic and general national laboratory community. The DoD used supercomputers to address important national security issues. Industries such as Boeing and General Motors used supercomputers for engineering analyses of aircraft, engine and structural automobile components. There was still generally a strong component of skepticism about computational results and as a consequence, computational predictions were usually thoroughly checked and validated.

By the 1990's, computing power had reached the point where many of the prior limitations on resolution and ability to solve complex mathematical systems had been overcome. Computational techniques began to have the potential to seriously address difficult and important problems such as climate change and weather prediction, nuclear weapons design, astrophysics, non-linear turbulence, chemistry, biology and human event simulation. This coincided with the advent of a new generation of scientists and engineers specifically trained as computational scientists. They began to use computational techniques to tackle many very difficult and complex problems. While these scientists and engineers were highly skilled at using computers, many have not had the inherent skepticism about computational results that was characteristic of prior generations. Although they know that computational models are only incomplete models of nature, they have sometimes placed an unwarranted faith in the validity of the computational results.

There are many documented failures of computational science. The Columbia Space Shuttle Accident was caused by a piece of foam that broke off from main fuel tank of the shuttle and struck the wing. The foam damaged the wing enough that hot gases entered the wing body during re-entry and destroyed the wing (Gehman, Barry et al., 2003). Just as soon as the foam was observed to have hit the wing during launch, the potential wing damage was assessed computationally. The computational analysis results were ultimately interpreted as indicating that significant wing damage was unlikely. There were, however, many problems with the analysis. The analysis was carried out by an inexperienced engineer. The foam-wing collision conditions were outside the range of validation for the computational model, CRATER. The engineer's management didn't pass along all of the engineer's analysis to the upper level NASA engineers making the crucial decisions on what to do about the rest of the flight of the Columbia.

CRATER was intended to model the impact of small objects, such as meteorites, on the shuttle tiles. The piece of foam was more than 400 times the size of the impacting objects used in the CRATER validation tests. In addition, CRATER did not treat the strength of multiple layers of the shuttle tiles correctly. A more capable tool, such as LS-DYNA™—used by the aeronautics, defense and automobile communities to study the effects of the impact of large size objects (e.g. cars, projectiles, etc.)(Hallquist, 2003)—generally was not used by NASA because of the detailed setup required. Although CRATER was a much simpler and less appropriate tool than LS-DYNA for this problem, results could be obtained much more quickly. While most of the code results indicated that the damage would be minimal, some of the CRATER analyses did indicate that there might be a problem. The senior engineering managers discounted the negative results because the CRATER model had given conservative results for the smaller scale validation experiments. For calculating the impact of a large piece of foam on the shuttle wing, the code was, in fact, not conservative. Upper level NASA management was misled into believing that it was unlikely that wing was fatally damaged. As the NASA accident report stated, it may not have been possible to avoid the loss of the shuttle even if it had been apparent that the wing was seriously damaged. However no effort was made to look for damage or to fix

it, partially because the CRATER(Gehman, Barry et al., 2003) analysis suggested that the damage was likely insignificant.

A second example recently occurred in the field of sonoluminescence. In early 2002, Taleyarkhan and coworkers at the Oak Ridge National Laboratory formed sound bubbles in deuterated acetone that collapsed and produced light(Shapira and Saltmarsh, 2002). They reported tritium decay and the emission of 14 MeV neutrons. This could only be true if the temperature achieved in the collapse was in the range of 10^6 to 10^7 degrees Kelvin, far higher than the 10^3 degrees Kelvin range normally produced in similar experiments. If such a high temperature was real, this would possibly be the most important scientific result of the 21st Century. Nuclear fusion energy production might be achievable with tabletop conditions. These results were “confirmed” by computational modeling. “Hydrodynamic shock code simulations supported the observed data and indicated highly compressed, hot (10^6 to 10^7 degrees Kelvin) bubble implosion conditions, as required for nuclear fusion reactions.” Unfortunately the authors employed an arbitrary factor of ten enhancement in the driving pressure to achieve agreement with the reported tritium and neutron results. The general physics experimental community quickly rushed to confirm such important results. No one else, including another group at Oak Ridge(Shapira and Saltmarsh, 2002), found significant levels of either tritium or 14 MeV neutrons as reported by Taleyarkhan, et al. The final conclusion has been that the reported experimental results were erroneous, and that the assumption that the driving force should be enhanced by a factor of ten was unwarranted. The fact that the original code results could be interpreted as confirming the erroneous experimental results gave the experimentalists additional encouragement to proceed with publication. In reality the code was misapplied, and an erroneous result was reported to the scientific community.

A third case involved theoretical predictions of the performance of a proposed new experimental facility. Based on extensive analysis of the results of smaller facilities by the international community in a particular field, it was proposed to build a “next step” large experiment. Just at the time that the design of this large experiment was being completed by the international design team, and approval was being sought for construction, three theorists completed a computer simulation of the expected performance of the proposed facility using a new code. The new results indicated that the proposed experiment would not meet its performance objectives. The new results and their implications for the proposed international project were widely reported in the popular media. The publicity strongly contributed to the US withdrawing from the project. Extensive analysis by the international community during the following year led to the realization that the three theorists had left out important effects that would increase the predicted performance. The theorists had overstated the validity of their preliminary results. The more complete results of other groups indicated that the expected performance would be roughly what the original design team had predicted. In this case, a computational prediction that was later proven to be wrong had an important impact on a scientific policy issue.

Many other examples are also available. They illustrate that computational science is beginning to play an important role in society, but not always a positive role. If this role is to be a positive one, we, as a community, must work to achieve the level of maturity for which our results are accurate and reliable. As in the case of suspension bridges, we must start analyzing our failures and successes, and learn from them. Our professional integrity demands no less. To illustrate some of the kinds of “lessons learned” analyses we will need to conduct, I describe in the next two sections the analysis that Richard Kendall and I carried out for six computer

simulation projects in the nuclear weapons program(Post, 2003). This analysis emphasizes both the importance of the code development process and the validity of the results of the computations. Both points are important because reliable answers require a mature methodology for development of the analysis tools.

5. “Lessons Learned from ASCI”

In 2002, Richard Kendall and I analyzed a group of application code projects and developed a set of “lessons learned” from the US Department of Energy’s Accelerated Strategic Computing Initiative (ASCI) program(Post, 2003). Since 1996, the ASCI program has spent almost \$6B to develop the predictive nuclear weapons simulation capability required for certification of the US nuclear stockpile in the absence nuclear testing. If the simulations could be used to make accurate predictions, then the US would not have to resume testing. The US would then be in a better position to discourage testing by other nations, thus slowing nuclear proliferation and the spread of the “ultimate” weapon of mass destruction while the US maintains a much reduced but sufficient nuclear deterrent.

The ASCI program includes the development of large scale, massively parallel computer platforms, the associated operating systems and code development tools, application codes and supporting algorithms and models. Some of the ASCI applications development projects have been successful in meeting their objectives and some have not. We analyzed the application projects at the Lawrence Livermore National Laboratory (LLNL) and the Los Alamos National Laboratory (LANL) utilizing metrics and case studies that focused on the history, organization and institutional support of the code projects. By identifying the common elements that led to success or failure to achieve objectives and comparing them to the experience of the information technology (IT) community (e.g.(DeMarco, 1997; Beck, 2000; Remer, 2000; Vliet, 2000; Thomsett, 2002)), we developed a set of recommended practices for large-scale technical code projects (Table 1).

Table 1: Code Development “Lessons Learned” from the ASCI Program at LANL and LLNL

- Build on the successful code development history and prototypes for your organization.
- Good people in a good team with a balanced skill mix are essential for successful code development projects.
- Software Project Management: Run the code project like a project.
- Risk identification, management and mitigation are essential for successful code development.
- Determine the schedule and resources from the requirements (goals and objectives, quality, team building and survival, and added value), not independently.
- A strong customer focus is essential for success.
- Better physics—and mathematics—in a physics code is much more important than better computer science.
- Use modern but proven Computer Science techniques; do not let your project become a Computer Science research project unless it is one.
- Train the teams in project management, code development techniques and the physics and numerical techniques used in the code
- Software Quality Engineering: Use Best Practices to improve quality rather than processes.

- Validation and Verification of codes are essential.

While the “lessons learned” list may seem obvious and certainly contains no surprises, implementing them in practice appears challenging. Every code project we studied violated at least a few. Almost all were violated for the least successful projects. These lessons are generally not new. Indeed, many of these lessons can be found in Fred Brooks’ 1975 classic: “The Mythical Man-Month” (Brooks, 1995) as well as a host of IT industry books and courses. Also, many of these principles apply to almost every organized human activity (e.g. (Ruskin and Estes, 1995; Verzuh, 1999)).

1. Identify the things your organization or institution does well and build on them. Introduce change with clearly defined goals in an evolutionary fashion. Even though you may think that the ideal structure for effective code development might be radically different from the existing organization and culture, radical, changes imposed too rapidly will disrupt whatever is working, and likely will not lead to success. Successful change takes time and requires that the people in the institution feel “safe” and trust the management to treat them fairly.
2. Teams, not organizations or processes, develop software. Form the best team you can, support it, and help it “jell.” A good team is the strongest asset an institution can have. Developing good teams is the key to developing good software. The teams need to have a balanced skill mix of scientists, programmers, mathematicians and computer scientists. A good team is also a crucial deliverable for a successful project because all further progress must build on the team.
3. Run the code project like the project that it is, with requirements, deliverables, a sound plan, realistic schedules and adequate resources. Align authority with responsibility. The project manager must be able to control the resources and the team, and have the active support of senior management. Otherwise, he is a project “cheerleader,” not a project leader, and the project will fail.
4. The development of large, complex technical codes is inherently very risky. Many, if not most, code development efforts fail to meet their initial objectives, and many fail completely. Identifying the major risks, minimizing them and providing contingency and mitigation is essential. The major risk factors for software projects are (DeMarco and Lister, 2003):
 - Uncertain goals, objectives and requirements;
 - Inadequate resources and support, including an overly ambitious schedule;
 - Institutional turmoil, including too much employee turnover;
 - Requirements and goals that change too rapidly or increase too fast; and
 - Poor team performance.

Two additional risk factors specific to technical software are:

- Non-delivery of essential components from outside contract organizations
- Poor judgment about the technical feasibility of candidate approaches and methods for meeting the requirements (c.f.(Glass, 1998)).

Poor team performance was the smallest risk factor for the ASCI code projects (Post and Kendall, 2003) as well as for the Information Technology (IT) software industry in general (Demarco and Lister, 2002). The other risk factors strongly dominate. Most code project failures (for general IT community and also for ASCI (Demarco and Lister, 2002;

- Thomsett, 2002)) are due to the failure of senior management to fulfill its responsibility to provide proper sponsorship, guidance, oversight and support for the code projects.
5. If adequate resources and schedule are not provided, the project will fail to meet its objectives on time. The failure to meet the initial objectives on time is regrettable but not fatal. However, this failure may cause management to take actions that punish rather than help the code team, and thus contribute to the failure of the entire project. If the resource levels continue to be inadequate, and the schedule continues to be too ambitious, the project will fail. As opposed to conventional projects, where one can fix two of the three factors—objectives, resources and schedules—software projects can only fix the goals. The objectives and goals determine the resources and schedule (DeMarco, 1997; Capers-Jones, 1998; Thomsett, 2002). The rate limiting process for code development is the rate at which people can analyze problems and develop solutions. The ability to increase the schedule is severely limited. As Tim Lister states: “People don’t think faster under pressure.” Similarly, the maximum size of a code team is limited by the ability of people to communicate complicated information with each other. This is reflected in the quantitative analysis that follows in the next section. The standard estimation techniques indicate that the optimal schedule and team size are a function only of the size and complexity of the code (Capers-Jones, 1998). Frederick Brooks put it another way: “Adding more staff to a late project will only make it later. (Brooks, 1995)” Ed Yourdon wrote a book entitled “Death March” about the disastrous consequences of overly ambitious code project schedules (Yourdon, 1997).
 6. Codes that customers do not want to use are like experiments that do not take data or equipment that people do not use. Such codes are a waste of resources and the efforts of creative people. The code team and management must focus on providing what the customer both needs and wants. If the code can’t provide the customer with the capability he needs, he can’t meet the need that justified the code development project. If the customer does not want or like the product, the code will fail even if it is what he really needs.
 7. The value of the code to the ASCII customer is the physics capability of the code. The degree of innovative computer science in the code is of little interest to him. The most successful ASCII codes have concentrated on improved physics and have been very conservative in their use of cutting edge computer science. By computer science, I mean methodologies for code development and programming techniques. I do not mean the development of more powerful mathematical techniques and algorithms. Indeed, many of the most significant improvements in the physics have been due to the development and use of new and more powerful computational mathematics techniques and algorithms that, together with increased computer power, enable the scientist to solve problems that couldn’t be addressed with prior generations of computers.
 8. Computer science research within the context of an application project greatly adds to the risks and often results in code project failure. Use modern, but proven techniques. Improving the physics is risky enough. Leave computer science experiments to those who can afford to fail a few times. Developing improved code development methods is very important and deserves support and emphasis. Such development should be carried out as an independent activity. The new methodologies should be tested in a way that doesn’t add to the risks of important projects, and be adopted only when the methods have proved their worth.

9. Invest in your people through training and professional development. They will become more capable as they acquire new skills and will be more productive. It is a good way to encourage change and to get the team members to see how other groups and industries tackle their problems. In addition, their morale will increase in proportion to the support of their management. Training also provides an opportunity for code team members to share experiences with the rest of the team and with other teams.
10. Software quality is important. High quality software has fewer defects, is more reliable and is easier to develop, maintain and use. However, research-oriented staff will not take a series of processes defined in a book and follow them blindly because someone in authority tells them to. They will apply the same standard to software development methods that they apply to their science. They have to be able to convince themselves that any proposed new process adds value to their work. For improving software quality, it is more successful to convince the teams that each individual practice add value (configuration management, etc.) than to try to convince them to blindly embrace on faith a large system of processes just because the management orders it. Software quality, however, can't be ignored. If you don't give it sufficient emphasis, your sponsor may impose software quality procedures that will very likely be much more onerous and less effective than the ones you would identify yourself.
11. Physics codes are an incomplete representation of reality. The models have shortcomings and often have mistakes in their implementation. Without a verification and validation program for the codes and their applications, there is no reason to believe that the code results have any validity at all.

6. Quantitative Estimation

These “lessons learned” were based on a qualitative and a quantitative analysis of the history of the different ASCI code projects and comparison with the Information Technology industry and conventional project management and scientific research. The quantitative analysis was a key element in establishing that the ASCI code projects had not been given a consistent set of requirements, resources and schedules. While our analysis (Post and Kendall, 2003) was relatively simple compared to the methods often employed in the Information Technology (IT) community (Capers-Jones, 1998), the conclusions are very clear. We found that the key predictor of success was the age of the code project and the amount of time allocated to complete the project and meet milestones. Our analysis of the historical data indicated that it takes about 8 years to develop an ASCI weapons code. The projects that had 8 years of development often succeeded, and all those that did not have 8 years of development time failed to meet their initial milestones. This result emphasized the crucial need to get the requirements right then to allocate sufficient resources and time (i.e. schedule) to meet those requirements.

The case studies included metrics (code size, team size, age, etc.). To see if the ASCI experience was consistent with the Information Technology (IT) community experience, we analyzed the case studies using a generic “function point” model (Capers-Jones, 1998) widely used by the IT industry. We calibrated this model for scientific code projects using the ASCI case study data. Function points are a weighted total of inputs, outputs, inquiries, logical files and interfaces (Symons, 1988; Capers-Jones, 1998). Functions points specifically developed for technical software (computational science software) do not yet exist. IT function point measures do exist and were something we could use to make the present argument.

$$FP = \left(\frac{C++SLOC}{53} + \frac{C SLOC}{128} + \frac{F77 SLOC}{107} \right) \quad (eq.1)$$

$$Schedule(months) = FP^x \quad 0.4 < x < 0.5; \quad use \quad x = 0.47 \quad (eq.2)$$

$$Team Size = \frac{FP}{150} \quad (eq.3)$$

$$Schedule = Contingency \times Function Point schedule + Delays \quad (eq.4)$$

$$Team Size = 3 + 0.6 \frac{FP}{150} \quad (eq.5)$$

We first converted the single lines of code to Function Points (FP)(e.g. eq. 1). T. Capers Jones lists the equivalent single lines of code (SLOC) per function point (FP) for the common computer languages (Capers-Jones, 1998) since computer languages have different information densities.

In this model, the required schedule and average team size are determined by the Function Point (FP) count (eqs. 2,3). We calibrated and modified these general scalings to account for the added complexity and viscosity associated with developing scientific codes specifically for the nuclear weapons complex. We increased the schedule by 1.5 years to account for the additional time it takes to recruit, hire, train and get security clearances for code development staff. Using a methodology developed by the Lawrence Livermore National Laboratory Engineering Department(Remer, 2000), we calculated a contingency factor of 1.6 to account for the additional risks, uncertainties, complexities, etc. for the LANL and LLNL computing environments (eq. 4). We modified the standard FP scaling for the size of the code team (eq. 5) (Capers-Jones, 1998)to match the ASCI data. This included a correction for small code teams.

We analyzed seven code projects, three at LLNL and four at LANL (Table 2). For national security classification reasons, we have identified the LLNL codes with the letters A, B and C. Table 2 lists the size of the code in function points, the time estimated by equation 4 to develop the initial capability of the code project, the actual age of the code at the point it was expected to accomplish its first milestone, whether or not the project succeeded, the optimal code team size estimated from equation 3 and the actual size of the team. The sizes of the codes (e.g. lines of code, loc) were approximate estimates by the code teams. Establishing the size of the code teams was challenging. In general, good records were not available. Thus the code team sizes were generally estimated by the code team leaders. Because good records were not kept, it was also difficult to account for staff who worked on the code project but were part of other organizations. More than one-half of the Blanca code project team, for instance, was part of other organizations. Where this was an issue, we used conservative estimates. For example, the Blanca code project staff probably had a staffing level of about 50 people for the first 4 or 5 years of its life instead of the 35 we assumed. We used a smaller number based on the actual number of people we could definitely identify as having worked on the project.

The case histories and the estimation procedures indicate that it generally takes a minimum of 8 years for a code team to develop an initial capability for a weapons code project. The requirements for a weapons code are determined by the physics necessary to simulate a nuclear weapon. LANL and LLNL have over 50 years of experience in this area, and know these requirements in detail. Weapons code projects require between 3000 and 6000 function points (Fig.2).

Some of the ASCI codes were started before ASCI began in 1996 (ASCI B, Legacy A for

LLNL, and the LANL Crestone code project). ASCI B was started roughly in 1992 and had a working prototype in 1994. The Crestone code project was started before 1992. ASCI A and the Shavano and Antero code projects were started around early 1997. Legacy A was started over 30 years ago and was included for comparison and normalization. Since we are able to match the history of weapons codes with scalings derived from the experience of the commercial software industry, we conclude that the constraints, computer science practices and management issues that generally apply to the IT industry generally apply to the development of weapons codes as well (i.e. there is no “Silver Bullet” that can radically reduce the development time(Brooks, 1987)).

Table 2 Software Resource Estimates for the LLNL and LANL Code Projects*

	LLNL			LANL			
	ASCI A	ASCI B	Legacy A	Antero Code Project	Shavano Code Project	Blanca Code Project	Crestone Code Project
Single Lines of Code	184000	640000	410550	300000	500000	200000	314000
Function Points (Eq.1)	4800	6100	5400	2900	4800	3800	2900
estimated schedule(Eq.4)	8.7	9	6.9	6.6	8.1	7.4	6.7
Project age (at initial milestone date)	3	9	N/A	4	3.5	8	8
Successful in achieving initial ASCI milestone	No	Yes	N/A	No	No	No	Yes
Estimated staff requirements (Eq.3)	22	27	24	14	22	18	14
real team size	20	22	8	17	8	35	12

*Yellow shading denotes historical data; white background denotes computed numbers(equations 1-5))

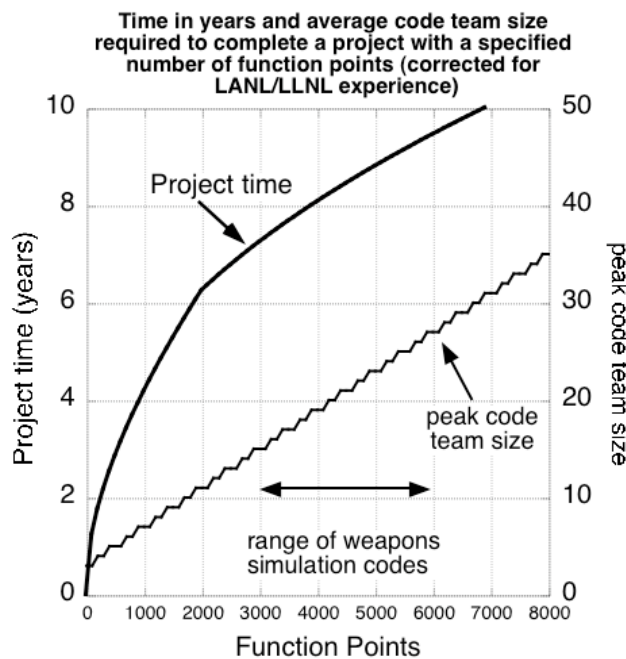


Figure 2 Time required to complete a project and average code team size as a function of code capability measured in function points.

We found that the dominant factor for success is the age of the code project. The code projects that did not have sufficient time (8 years) to complete their projects failed to meet their milestones. All but one of the code projects that had 8 years succeeded in meeting their milestones. This is clear evidence that schedules and requirements must be consistent. The schedule cannot be fixed independently of the requirements, a fact long appreciated by the IT industry (DeMarco, 1997; Capers-Jones, 1998) but not adequately taken into account in the early planning for ASCI. The ASCI program set the milestone for demonstrating the capability of each code project to be three and a half years (December 1999) after the beginning of ASCI (~mid 1996) and three years after the date that many of the code projects were launched (~January 1997).

Adequate development time is necessary—but not sufficient—for success. Several code projects failed in spite of having adequate time. Poor practices and inadequate support—implicitly included in the contingency factor—hurt many of the projects as well. The Blanca code project failed to meet its milestones even with adequate time and ample resources.

Another point is that it is clear from the function point scaling relations (eqs. 1-5) that the code requirements determine both the schedule and resources needed for success. This estimating analysis indicates the importance of a realistic set of requirements, schedule and resources. Without them, projects will fail and the needed applications will not be developed.

These case studies helped persuade the ASCI senior management that the “younger” code teams (those started less than 8 years before the milestone) were not necessarily incompetent, but were just unable to do 8 years of work in less than 4 years. The management was then able to recognize that several (but not all) of these “younger” projects were actually making very good progress compared to “normal” code development rates and had very high potential for producing successful codes that would give the ASCI program substantially improved tools. Partly motivated by the case studies, the ASCI management then developed a more realistic schedule for code development, placed more emphasis on the needs of the users and provided better support for the code teams.

Three issues identified as “lessons learned” are expanded on in the following two sections: verification and validation and software quality. Both areas are crucial for success for technical software projects, and have special—and often not well understood—requirements.

7. Verification and Validation

An application code typically solves a model problem that is only an abstraction of reality. Many things can limit the validity of a code calculation. The models and solution algorithms may be implemented incorrectly. The models may not accurately reflect the phenomena of interest (Roache, 1998; Oberkampf and Trucano, 2002). Verification is the determination that the code solves the model correctly. Validation is the determination that the models in the code capture, with adequate fidelity, the phenomena of interest. Both are essential elements of a program to develop and apply application codes to problems of interest (Roache, 1998). Without adequate verification and validation, there is no reason to believe any part of a code result. Unfortunately, for much of computational science, verification and validation efforts fall far short of what is needed.

Both verification and validation become more difficult as codes become more complicated and their applications more important. A typical application might have many

different components. A sophisticated climate modeling code might include models for ocean evaporation, ocean currents, ocean salinity, atmospheric flow, clouds, precipitation, CO₂ sequestration, radiation transport, atmospheric chemistry, ground water flow, vegetation growth, ice formation, etc. The code might predict many observables, such as average surface temperature, precipitation levels, etc. The accuracy of these observables depends on the accuracy of each component model, the completeness of the set of all the models (i.e. does the code treat all of the important phenomena), the accuracy of the solution method for the model including its interaction with the other models, the physical data used in the models, the adequacy of the problem generation and the ability of the user to correctly set up the problem, run it and interpret the results. Verifying and validating all of these is a major challenge.

The accuracy of the multi-model code depends first on the accuracy of each component, as well as the accuracy of their interactions. In practice, first one has to verify each component, then validate each component for the relevant regimes, then verify and validate progressively larger collections of interacting components, until the entire integrated code has been “verified” and “validated” for the problem regimes of interest.

There are at least four common verification techniques, all with serious shortcomings:

1. Comparison of the code results with the analytic results for a problem with an exact answer,
2. Establishing that the convergence rate of the truncation error is consistent with the expected convergence rate, and
3. Comparison of the observed results with the expected results for a problem specially manufactured to test the model (or models)(Boehm, 2002),
4. Computation and monitoring “conserved” quantities and parameters that should be constant or are predictable.

The first method is worthwhile, but extremely limited in practice. There are usually few (if any) relevant problems with exact answers, especially with realistic boundary conditions, realistic geometries, realistic data, non-linear conditions, or multiple-component systems. The computational fluid dynamics community widely uses the convergence rate of the truncation error to verify programs(Roache, 1998; Hallquist, 2003). This technique, too, is limited in applicability. It works best when the expected truncation rate can be determined from the basic difference equations and boundary conditions. That is often not possible. Convergence rates often aren’t useful to check two or more interacting modules. The third technique, the Method of Manufactured Solutions, is, in principle, very powerful(Roache, 1998; Pautz, 2001; Roache, 2002). It works for almost arbitrarily complicated and strongly coupled models, and almost arbitrarily complicated boundary conditions. However, problems with real data, moving or adaptive meshes, non-analytic (and non-differential) terms and real physical data are difficult to treat. These challenges, as well as the complexity of implementing the manufactured solutions, seem to prevent its wide-spread use. A fourth technique is monitoring behavior the developer know has to be correct, such as “conserved” quantities (e.g. total energy, momentum, mass, etc.), quantities whose evolution can be estimated (e.g. entropy) to check the accuracy of individual components and of the whole code, or procedures that can be predicted (e.g. procedural behaviors designed into the code). Yet, in spite of all these limitations, verification must be done as thoroughly as possible. If a code isn’t solving the models correctly, then the answers are worthless. Any correspondence of the answers with reality is completely fortuitous. Verification needs to be performed every time the code or operating system (compilers, etc.) changes. A code has to be verified before it can be validated. Validating an unverified code is generally a waste of

time. Given the deficiencies of existing practices, better verification techniques are desperately needed.

Comparing the results of a problem for two different codes (Benchmarking) can increase the likelihood of catching errors, but only to a limited degree. Both codes could be wrong. Two codes usually have different ways of solving a problem and sorting those effects can be time-consuming and potentially impossible. Benchmarking is worthwhile because it can catch errors, but it isn't a substitute for a mathematically rigorous verification procedure.

As a practical matter, diligent code developers do as much verification as they judge feasible, and then keep their eyes open for suspicious behavior by the code. However, this is far from a guarantee that the code is free of errors. Also, not all code developers (and users) are sufficiently diligent or knowledgeable.

Once a code has been verified as much as possible, the code must be validated for the problem regimes of interest. A code is never a valid tool for all conceivable problems. It can only be validated for specific regimes, and the validity in adjacent regimes estimated (Figure 3). The entire calculational system including the user, computer system, problem set-up, problem running and results analysis for each user and computer system must be validated because all elements are important. An inexperienced or non-expert user can easily get wrong answers using a good code in a validated regime.

Validation has a number of challenges. Each individual component and all important combinations of the components must be validated. Validation data and experiments have a variety of forms (Table 3).

Table 3 Four types of experiments used to validate codes:

1. Passive observations of physical events (e.g. supernovae explosions or the weather),
2. Experiments designed to certify a physical component or physical system (tests of an engineering component such a scaled airplane wing, car crash, etc.),
3. Experiments designed to elucidate a general physics or engineering principle or law (e.g. wind tunnel studies of turbulent eddies around airfoils), and
4. Experiments specifically designed to validate a code application (e.g. wind tunnel tests designed to provide data to validate a code calculation).

Each type of experiment can be done before or after the code prediction has been completed and can address single-effect issues or integrated phenomena.

The best validation consists of the comparison of predictions made before an experiment with data from experiments designed specifically for validation. Successful prediction of experimental results is a better test than successful reproduction of existing experiments. Since few codes have no uncertainties, "tuning" a code for an application is usually necessary to get reasonable answers. The experienced user has learned how to set up an appropriately zoned mesh, how to vary the physical data within the known uncertainties to get reasonable answers, which effects are essential for the application and which are inappropriate, how to interpret the results, when the code has is outside the region of validity, etc. With this freedom, it is thus often feasible to tune a code to match many of the salient points of an existing experiment. It is a much more rigorous test of the code application to predict experimental results before the experiment has been conducted. This is also prototypical of the purpose of the code and computer system, i.e. to make accurate predictions of unknown events using known data before the events occur. An additional benefit of the validation process is that it trains the users how to use the code to get

reasonable results. The entire calculational system needs to be validated (code, user, computer system). As we have seen, an inexperienced user can get the wrong result.

For many applications, controlled experiments are not feasible or are impractical. For them validation is especially challenging. Models of astrophysical and large-scale geophysical phenomena (weather, climate, volcanoes, asteroid impact, watersheds, etc.) and large scale economic and political systems, must rely on historical data and current observations. We will not be able to conduct controlled supernovae explosions in the near future or schedule earthquakes, volcanoes or asteroid impacts. For these phenomena, the best that can be done is to collect as extensive sets of data as possible, especially data that is fundamental to the correctness of the code. For these systems it is often not possible to get data for all conditions, a complete time history, adequately resolved data, and data for many of the quantities of interest.

However, many, if not most, applications can be validated with data from controlled experiments. Key issues include adequate coverage in space and time of the appropriate experimental initial conditions and the behavior of the important variables. An accurate description of the initial and boundary conditions is essential.

The types of experiments used for validation listed in Table 3 are also listed in order of their utility for validation. Aeronautical Computational Fluid Dynamics (CFD) codes were first validated using wind tunnel tests of scaled aircraft parts (Experiment type 2, Table 3). The object of the experiment was to test the aircraft part. The use of the data for validation was largely incidental and occurred after the experiment. Most of experiments of type 2 were integral, in that

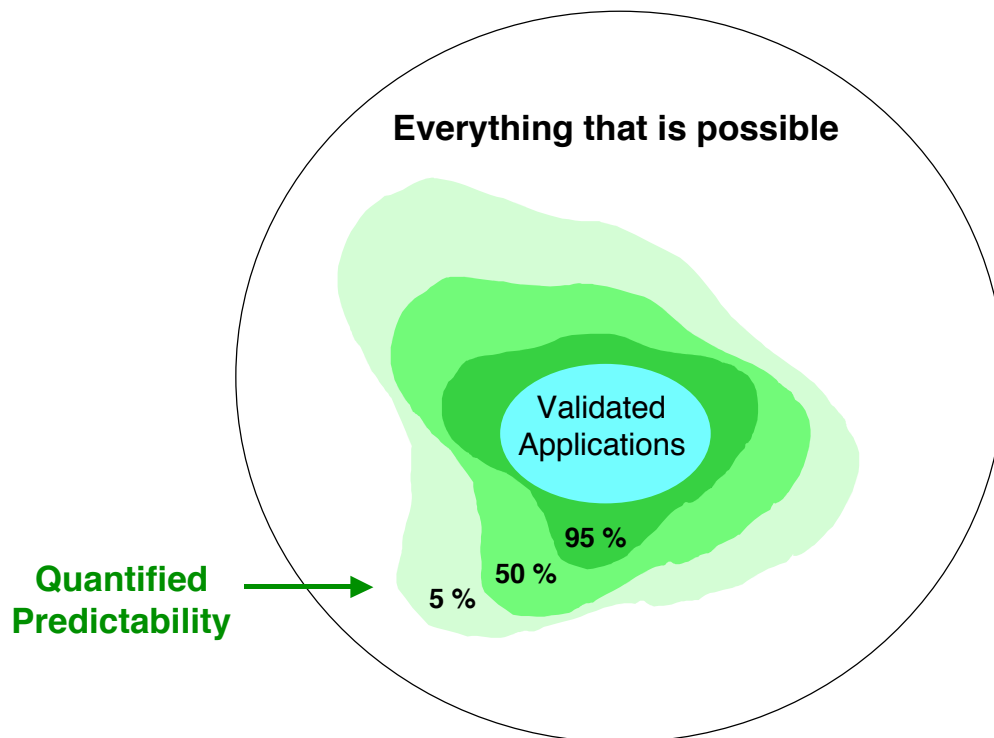


Figure 3 Schematic illustration of code and problem validation and predictive validity range (Courtesy, D. Tubbs)

they gave data that reflected the behavior of the trade-offs of a number of competing effects. Code developers recognized that data for specific effects was needed to validate each component

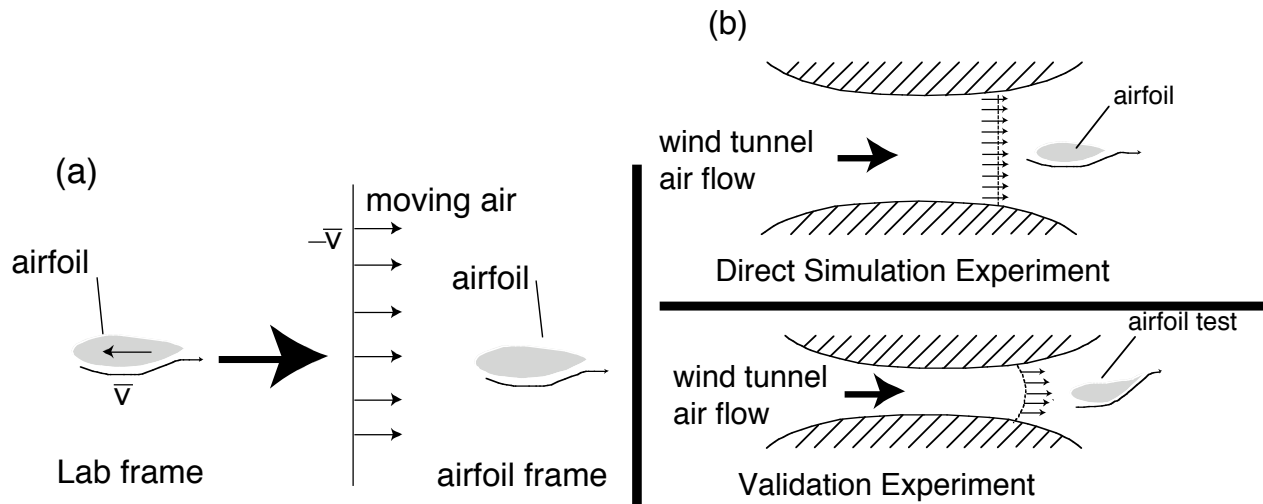


Figure 4 (a) Wind tunnel matching planar air flow conditions of an airfoil moving through free space air to test aircraft components. (b) Comparison of large wind tunnel for testing aircraft components and small wind tunnel experiments designed to validate CFD codes.

in their codes. They therefore used data from single effect experiments designed to study a single, isolated phenomenon. Such data might be yield strength data for metal components, thermal conductivity measurements, etc. Again, validation of a code was usually not the primary purpose of the experiment, although such experiments were often cheap enough that they could have been used for explicit validation experiments. The fourth type of experiments are those designed explicitly for code validation. The purpose of those experiments is to test the models in the code. The code is often used to design the experiment. Some of these points are illustrated in Figure 4. An airfoil moving through the air sees a plane front of air rushing toward it. Fifty years ago, wind tunnels were used to faithfully reproduce the plane air front conditions to test aircraft components. Achievement of a plane front required a large wind tunnel to minimize the effects of drag by the wall. Now, much smaller wind tunnels are used to validate the codes that are used to design airfoils. Once the requirement for a planar air front was removed, a much smaller and cheaper wind tunnel could be used. The validation wind tunnel facility can also have shorter set-up and experimental turnaround times and be more easily and thoroughly diagnosed. The idea is to test the code, not the component. A final test of the component may be advisable, but a CFD code validated for the appropriate conditions can be used for many of the design studies, especially if the final results of a computational optimization study are checked experimentally.

In fact, data from experiments not designed for validation can sometimes be misleading or inaccurate for validation. The experiment may have been designed to measure a particular effect. The data for other effects may not have been checked sufficiently and may be inaccurate, misleading or wrong. As noted in the sonoluminescence example earlier, codes can be, and have been, forced to match incorrect experimental data.

A paradigm shift with regard to the value and importance of validation experiments is needed in the experimental community. Experimentalists and funding agencies understand the value of experiments designed to explore new scientific phenomena, test theories or certify and test the performance of a design component. Few appreciate the value of experiments explicitly conducted solely for the purpose of code validation. There generally exist no mechanisms to get validation experiments funded even if experimentalists are interested.

Finally, since the value of verification and validation is to ensure that the code can give accurate predictions for the phenomena of interest, a written record of the verification and validation of the code is extremely important. That record is necessary to establish the credibility of the code predictions with the code project sponsors and customers. In fact, validation needs to be organized like a project, with goals and requirements, a plan, resources, a schedule, and deliverables including a documented record of the validation project.

Few existing computational science projects practice systematic verification or validation. Almost none have dedicated experimental validation programs with dedicated validation experiments. Yet, without such programs, computational science will never achieve credibility.

8. Software Quality and Software Project Management

Software quality and software project management are very important issues. Improvements in quality offer the promise of greater longevity and easier maintenance. Attention to quality will likely improve the code. Inattention to quality will almost certainly contribute to poor quality (high defect rates, and code that is hard to maintain and upgrade). It can also leave the code project vulnerable to the Software Quality Assurance (SQA) mafia. If poor quality becomes an issue, the sponsors and customers will take action. The DoD and other sponsors have developed fairly rigid processes for code development and software quality assurance in response to disasters caused by buggy aircraft and satellite control software. Bugs in aircraft control software can cause airplane crashes. To reduce the defect rate, the Air Force established a very rigorous procedure for vendors to follow to develop such software (Paulk, 1994).

Similarly, sound software project management can do a lot to speed code development, increase the likelihood of a successful product and minimize the defect rate.

Quality was an issue for the US automobile industry in the 1970's and 1980's (Halberstam, 1986). The American automobile industry produced poor quality cars that people didn't buy. The Japanese built high quality cars that people did buy. A basic difference was that the US automobile industry did not emphasize quality on the assembly line and in the externally supplied components. They mostly tested the cars after they came off the assembly line and tried to fix the worst ones. The Japanese, on the other hand, emphasized quality at every step of the assembly process and for components. They tested the cars at many points along the assembly line and tested components before installation. The result was that Japanese cars had much higher quality, and the American automobile industry lost many customers.

Similarly, software quality engineering is most effective when it is applied at each step of the software development process. This is much better than the all too common practice of waiting until the code is nearly complete to begin testing the code. However, just as on the assembly line, different development processes require different methods. No one size fits all. Also, just as the Japanese auto makers emphasized input from the assembly line workers, the code developers themselves are often the best judges of how to implement quality. A process rigidly imposed by senior management will likely get the same type of token compliance observed in the US auto industry.

Quality assurance for technical software has an important sociological dimension. Technical software is developed by teams of scientists and engineers. Scientists and engineers are trained to question everything, and accept nothing purely on the basis of authority. After all, even though he might want to, your boss can't change the laws of nature—and that's what you

are trying to model. In fact, that's why we hire scientists to develop scientific software. The models in the codes have to be right. If the models don't reflect reality, the code results are worthless. We will then make decisions that will be wrong, often with tragic consequences. Giving scientists a "bible" that describes an elaborate, rigid process for developing software, but which provides little in the way of justification is counter-productive. It seems to be more successful to work with each team to identify the "practices" that add value to the scientific code development process, and encourage the teams to implement the practices they helped to identify (Phillips, 1997). It's also necessary to provide support, especially to carry out some of the more routine practices. For large projects, it's better to hire a "code librarian" to implement and maintain the configuration management system and a dedicated "tester" to design, implement and run regression test suites than just telling the team to do it. Without additional resources, the team will have to drop other tasks to complete newly assigned software quality jobs. The practices that technical software development groups have found useful include configuration management, requirements definition, sound software project management, regression testing, adequate documentation, design and code reviews, etc.

A lot of technical software is developed for various government agencies. The contracting officers for these agencies often aren't very knowledgeable about the challenges of developing large, technical software projects. They are, however, accountable for delivery of the programs and projects they sponsor. Large technical software projects have substantial risks. The record indicates that they are often behind schedule, over-budget, don't deliver exactly what was promised, and even fail entirely. To succeed, sponsors have to hold the code development organizations accountable. It is therefore very tempting for the government agencies to require that the organizations they sponsor follow a "process" model like the Capability Maturity Model (CMM) developed by the Software Engineering Institute at Carnegie-Mellon University for the Air Force (Paulk, 1994). After all, there is a lot of data that indicates that code development organizations that follow the CMM processes produce "better" code, meet milestones, etc., and, in the end, who can be against quality? This kind of quality for scientific software, however, comes with a severe price. A detailed analysis of the CMM processes indicates that it works well for software that must have no bugs (e.g. the airplane control software mentioned above). Implementing the CMM process, however, takes a lot of time. History shows that several years are required for each step to move from one CMM level to the next. There are five distinct CMM Levels. In addition it requires a lot of additional resources. The major problem with applying the full CMM to the development of scientific software is that the strong emphasis on avoiding and reducing bugs and defects adds a lot of viscosity to the development process.

Computational science has different goals and requirements from aircraft control. It is much more important that the physics or chemistry be right and that the solution algorithms be right than that every last bug be eliminated. Developing the right physics or chemistry package usually takes a lot of experimentation and creativity. It is impossible to plan every detailed facet of a large complex code with scientific and mathematical challenges. The code development team must be very creative. It must develop and test many new algorithms and models to find ones that work. A rigid code development process impedes the flexibility and creativity needed to develop new codes. This is not only the case for scientific codes, but also for most really innovative software development. There is a running debate on this issue in the software literature between the "rigid process" community and the "agile software" community. The "agile software" community stresses the importance of innovation and the difficulty of being innovative if one is constrained by rigid processes (Highsmith and Cockburn, 2001; Boehm,

2002; DeMarco and Boehm, 2002). The “rigid process” community stresses the importance of reduced defects and efficient code development (Herbsleb, Zubrow et al., 1997). Both positions have valid points, but the reality is that there is no “one size fits all” answer. Just as there is no “one way” to do laboratory experiments in physics, chemistry or biology, theoretical work in chemistry, physics or biology, or engineering design and analysis, there is no “one way” to develop technical software. There is no “fool proof” way to develop codes, or as Frederick Brooks states: “There is no silver bullet for software development” (Brooks, 1995). Just as in other scientific methodologies, one has to do the intellectually hard work of examining and testing candidate practices and then use the ones that work for the problem at hand. But this does not mean that “any old method” is acceptable and will work. It only means that not every development problem has the same answer. We can’t be lazy. While we can’t blindly accept what people hand us, we do have to find something that works well.

A constant theme that seems to always emerge from case studies is that good software project management is essential. It is usually more important than any set of externally imposed processes. It is noteworthy that the Software Engineering Institute has recently recognized the importance of software project management. It has developed the “Team Software Process” (Humphrey, 2001) that appears to be very similar to the software project management methods long advocated in the general IT industry, especially in the non-government IT industry (e.g. (Brooks, 1995; DeMarco, 1997; Remer, 2000; Thomsett, 2002)). The SEI data shows that introducing sound software project management achieves a greater proportional reduction in the defect rate than moving many levels up in CMM process level.

The burden of identifying code development methods that work well falls on every code team. As noted before, if the team doesn’t find methods that work, the sponsor will attempt to force the team to use processes and methods that he selects on the basis of what he has been told by others. The processes he picks likely won’t be the ones that the team would pick. Developing a good set of practices and implementing them is the beginning of a good defense against being forced to follow externally imposed practices. The team also has to be able to articulate their practices and be able to demonstrate to management and, in some cases, to auditors from DoD, DOE, NASA, etc. that the team’s practices work. There is no one solution to this problem either. The team has to work to establish credibility with its management so management will trust the team to do things right.

While the technical software community has many unique issues, it nonetheless can learn much from the general IT industry. The IT community has had to address the problem of how to plan and coordinate the activities of large numbers of programmers writing fairly complex software. I have found that few of even the simplest well-known and proven methods for organizing and managing code development teams and projects are being employed by the technical software community. The most common approach seems to be to independently rediscover the IT industry “lessons”. This unfortunately leads to wasted effort and all too often results in failure.

Several of these “lessons” are worth highlighting. We need to learn how to develop specifications and requirements for technical projects. Most technical projects start out with a vision of what the code team leaders want to accomplish. Unfortunately, the leaders don’t develop requirements and specifications at the level of detail that the other members of a large team can follow to produce an integrated code at the end. There is relatively little planning and almost no estimation of resources and schedule. This often leads to overly ambitious goals and unrealistic schedules, missed milestones, and sometimes to project failure. While good

estimation is hard, one commonly recommended technique is to develop a prototype that requires 5 to 10% of the full project resources, and use it for estimation (McConnell, 1997). Another technique is to look at similar projects and scale from them. In fact, most technical code projects don't appear to follow very many of the "lessons learned" from the ASCI code projects listed in Table 1.

A final issue is the need to balance the requirement to improve the computer science techniques and methodologies used for code development while using conservative and reliable practices for the development of essential applications. A good example is the effort to develop the Common Components Architecture as a way to standardize component development. In principle, this is a great idea. If one develops a module, it would be wonderful if it could be used in many applications. The core of the idea is to develop interface specifications for modules. Where this is possible, it should greatly help code development. Unfortunately, every module has a different purpose, and usually requires different interfaces for each technical problem. The hard part is to define the specific interfaces and it's not clear that this can be done in a general way. It's difficult to see how a computer scientist will be able to anticipate what interfaces are needed so that a module that calculates the thrust from rotor on a bacterium can be successfully integrated into a unified calculation of a swimming bacterium. Clearly new code development methodologies must be developed and tested on real problems. Identifying ways to develop these new methodologies and test them without unduly impeding application code development and greatly increasing application code development risk will continue to be a major challenge for the computational science community.

Another challenge will be to develop appropriate metrics for the development of technical software. Clearly conventional function points are inadequate. Technical software has additional complexity and challenges beyond those faced by the IT industry. Developing those metrics should be a key goal of any "lessons learned" activity. Gathering data on code projects will be essential. Without real data on the code development process and the codes themselves, it will be difficult to identify what works and what lessons can be learned and applied to other projects.

There is also a tendency toward the formation of "virtual teams", teams of non-located software developers at geographically separated sites. Such teams have the advantage of bringing varied skill sets to the project without the need to relocate and the potential for tapping the expertise of a number of institutions and generating the political as well as technical support of many institutions. Collocated development teams face communication and coordination challenges. Those problems are more severe for virtual teams, and success will require addressing these challenges.

9. Conclusions and Path Forward

Computational science has an important role to play in society. The High Performance Computing community is meeting "The Performance Challenge" to provide us unprecedented power to tackle important problems. However, two additional challenges must be met before that potential can be realized. First, the community must be able to efficiently develop programs for the ever more powerful and ever more complicated computer platforms—"The Programming Challenge." Secondly, the application models must become sufficiently accurate that they can be used for prediction with confidence "The Prediction Challenge." To meet "The Programming Challenge," the High Performance Computer and operations and development software

community (industry, government and academia) must develop the tools and methods to facilitate the development and running of codes so that application codes can be developed quickly and reliably and can be run efficiently on the High Performance Platforms. To meet “The Prediction Challenge,” the computational science community (industry, government and academia) will need to become as mature as the theoretical and experimental scientific and engineering design communities. The computational science community must develop methods to ensure that the equations and models in the codes accurately reflect the real world, that the equations and models are solved correctly, that the applications are set up and run correctly by knowledgeable and careful people, and that the results are interpreted correctly. Accurate equations and correctly implemented models, as well as efficient and economic development, require attention to the code development process. The process must be consistent with the general “lessons learned” discussed in the paper. One of the most important “lessons learned” is that an intensive verification and validation program is an essential element of ensuring that computational results are accurate. Unfortunately, not only is the level of verification and validation usually insufficient, there is inadequate effort devoted to developing new methodologies and concepts for verification and validation. Much is needed and little is being done. Finally, those developing the code and those using the code must have a deep appreciation of the limits of the code and a deep-rooted appreciation that the results may not be correct.

As in other methodologies, retrospective case studies of past practices are an essential part of the path toward maturity. It is imperative that we as a discipline continuously examine and assess our mistakes and our successes. Without such a continuous re-assessment, we will continue to make the same mistakes. Our field will never be able to fulfill the tremendous promise that powerful computers give us.

Another way to look at it is as an issue of professional integrity. Unless our field has the same level of professional integrity as other methodologies (experiment, theory and engineering design), we will never be as credible as the other methodologies. We will continue to hear the refrain: “Who can believe that? It’s just a code result and we know they can get anything they want if they play with the code enough.” Scientists who conduct experiments irresponsibly find that their professional reputations are discredited quickly and thoroughly. Who knows where Ponds or Fleischman (the “discoverers of cold fusion) are today (Huizenga, Harris et al., 1989)? It is rare that anyone in computational science gets even the slightest rebuke for a misleading or incorrect result.

It is not enough for 95% of the work in computational science to be reliable, and 5% to be wrong. Unless the outside world can tell which 5% is bogus, none of our work will have the impact it deserves.

The DARPA High Productivity Computing Systems (HPCS) project is focusing on reducing the time to solution for important problems by meeting the Performance, the Programming and the Prediction challenges. It is working with three vendors, IBM, Cray and Sun to design and build peta-flop platforms. Part of the HPCS project is the development of benchmarks for the platforms that are prototypical of real applications. Attention is being paid to the development of programming models and development tools for optimizing parallel performance. The HPCS project is sponsoring case studies of representative computational science projects in the DoD, DOE, NASA, NOAA, industry and academia to identify the lessons learned and document and publish them for the benefit of the computational science community. I have outlined a number of “lessons learned” that have already been developed from the ASCI code projects. As we assess a wider range of projects, we will refine those lessons and identify

new ones. Adoption of these “lessons learned” by the computational science community will help the field to mature just as the development of “lessons learned” and their adoption has helped other fields to mature.

10. Acknowledgements:

The author is grateful for discussions with Don Burton, Bill Carlson, John Cerutti, Linnea Cook, Larry Cox, Tom DeMarco, Dale Henderson, Leo Kadanoff, Richard Kendall, Jeremy Kepner, Joseph Kindel, William Krauser, Ken Koch, Steve Libby, Bob Lucas, Tom McAbee, Doug Miller, Pat Miller, Jim Rathkopf, Don Remer, Rob Thomsett, Tim Trucano, David Tubbs and Larry Votta and to the Los Alamos National Laboratory and Department of Energy for support.

11. References

- Beck, K. 2000. *Extreme Programming Explained*. Boston: Addison Wesley.
- Boehm, B. 2002. "Get ready for agile methods, with care." *Computer* **35**(1): 64-69.
- Brooks, F. 1995. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition. Menlo Park: Addison-Wesley Publishing Co.
- Brooks, F. P. 1987. "No Silver Bullet: Essence and Accidents of Software Engineering." *Computer* **20**(4): 10-19.
- Capers-Jones, T. 1998. *Estimating Software Costs*. New York: McGraw-Hill.
- DeMarco, T. 1997. *The Deadline*. New York, New York: Dorset House Publishing.
- DeMarco, T. and B. Boehm 2002. "The Agile Methods Fray." *Computer* **35**(6): 90-92.
- DeMarco, T. and T. Lister (2002). *Risk Management for Software*. Arlington, MA, The Cutter Consortium.
- DeMarco, T. and T. Lister 2003. *Waltzing with Bears, Managing Risk on Software Projects*. New York, New York: Dorset House Publishing.
- Frank, M. P. 2002. "The Physical Limits of Computing." *Computing in Science and Engineering* **4**(3): 16-26.
- Gehman, H. W., J. L. Barry, et al. 2003. *Report of the Columbia Accident Investigation Board*. Washington, DC, National Aeronautics and Space Administration.
- Glass, R. L. 1998. *Software Runaways: Monumental Software Disasters*. New York: Prentice Hall PTR.
- Halberstam, D. 1986. *The Reckoning*. New York: William Morrow and Co.
- Hallquist, J. O. 2003. *Current and Future Developments of LS-DYNA-1*. 4th European LS-DYNA Conference, ULM, Germany, Livermore Software Technology Corporation.
- Herbsleb, J., D. Zubrow, et al. 1997. "Software Quality and the Capability Maturity Model." *Communications of the ACM* **40**(June, 1997): 30-40.
- Highsmith, J. and A. Cockburn 2001. "Agile software development: the business of innovation." *Computer* **34**(9): 120-127.
- Huizenga, J., T. H. Harris, et al. 1989. *Cold Fusion Research, A Report of the Energy Research Advisory Board to the United States Department of Energy*. DOE/S-0073 DE90 005611. Washington, DC 20585, United States Department of Energy.
- Humphrey, W. S. 2001. *Winning with Software: An Executive Strategy*. Pittsburg: Software Engineering Institute.

- Laughlin, R. 2002. "The Physical Basis of Computability." *Computing in Science and Engineering* **4**(3): 27-30.
- McConnell, S. C. 1997. *Software Project Survival*: Microsoft Press.
- Oberkampf, W. and T. Trucano 2002. "Verification and Validation in computational fluid mechanics." *Progress in Aerospace Studies* **38**: 209-272.
- Paulk, M. 1994. *The Capability Maturity Model*. New York: Addison-Wesley.
- Pautz, S. D. 2001. *Verification of Transport Codes by the Method of Manufactured Solutions: the ATTILA Experience*. ANS International Meeting on Mathematical Methods for Nuclear Applications, Salt Lake City, Utah, American Nuclear Society.
- Petroski, H. 1994. *Design Paradigms: Case Histories of Error and Judgement in Engineering*. New York: Cambridge University Press.
- Phillips, D. 1997. *The Software Project Manager's Handbook*. Los Alamitos: IEEE Computer Society.
- Post, D. E. 2003. *Lessons Learned from ASCI applied to the Fusion Simulation Project*. LA-UR-03-7540. Los Alamos, Los Alamos National Laboratory.
- Post, D. E. and R. Kendall 2003. *Software Project Management and Quality Engineering Practices for Complex, Coupled Multi-Physics, Massively Parallel Computational Simulations—Lessons Learned from ASCI*. LA-UR-03-1274.
- Remer, D. 2000. *Managing Software Projects*. UCLA Technical Management Institute, Los Angeles, CA, UCLA Extension Courses.
- Roache, P. J. 1998. *Verification and Validation in Computational Science and Engineering*. Albuquerque: Hermosa Publishers.
- Roache, P. J. 2002. "Code Verification by the Method of Manufactured Solutions." *Transactions of the ASME* **124**: 4-10.
- Ruskin, A. M. and W. E. Estes 1995. *What Every Engineer Should Know About Project Management*. New York: Marcel Dekker, Inc.
- Shapira, D. and M. Saltmarsh 2002. "Nuclear Fusion in Collapsing Bubbles—Is It There? An Attempt to Repeat the Observation of Nuclear Emissions from Sonoluminescence." *Physical Review Letters* **89**(10): 104302-104305.
- Symons, C. R. 1988. "Function Point Analysis: Difficulties and Improvements." *IEEE Transactions on Software Engineering* **14**(1): 2-11.
- Thomsett, R. 2002. *Radical Project Management*. Upper Saddle River, NJ: Prentice Hall.
- Verzuh, E. 1999. *The Fast forward MBA in Project Management*, John Wiley.
- Vliet, H. v. 2000. *Software Engineering, Principles and Practice*. Chichester: John Wiley and Sons, Ltd.
- Yourdon, E. 1997. *Death March*. Upper Saddle River, NJ: Prentice Hall PTR.