Chapter 2

Assessment of Accuracy and Reliability

Ronald F. Boisvert,[11] **Ronald Cools,**[12] and **Bo Einarsson**[13]

One of the principal goals of scientific computing is to provide predictions of the behavior of systems to aid in decision making. Good decisions require good predictions. But how are we to be assured that our predictions are good? Accuracy and reliability are two qualities of good scientific software. Assessing these qualities is one way to provide confidence in the results of simulations. In this chapter we provide some background on the meaning of these terms.

2.1 Models of Scientific Computing

Scientific software is particularly difficult to analyze. One of the main reasons for this is that it is inevitably infused with uncertainty from a wide variety of sources. Much of this uncertainty is the result of approximations. These approximations are made in the context of each of the physical world, the mathematical world, and the computer world.

To make these ideas more concrete, let's consider a scientific software system designed to allow *virtual experiments* to be conducted on some physical system. Here, the scientist hopes to develop a computer program which can be used as a proxy for some real world system to facilitate understanding. The reason for conducting virtual experiments is that developing and running a computer program can be much more cost effective than developing and running a fully instrumented physical experiment (consider a series of crash tests for a car, for example). In other cases performing the physical experiment can be

[11] Mathematical and Computational Sciences Division, National Institute of Standards and Technology (NIST), Mail Stop 8910, Gaithersburg, MD 20899, USA, email: boisvert@nist.gov. Portions of this chapter were contributed by NIST, and are not subject to copyright in the USA.

[12] Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium, email: Ronald.Cools@cs.kuleuven.ac.be

[13] National Supercomputer Centre and the Mathematics Department, Link¨opings universitet, SE-581 83 Link¨oping, Sweden, email: boein@nsc.liu.se

practically impossible, for example a physical experiment to understand the formation of galaxies! The process of abstracting the physical system to the level of a computer program is illustrated in Figure 2.1. This process occurs in a sequence of steps.

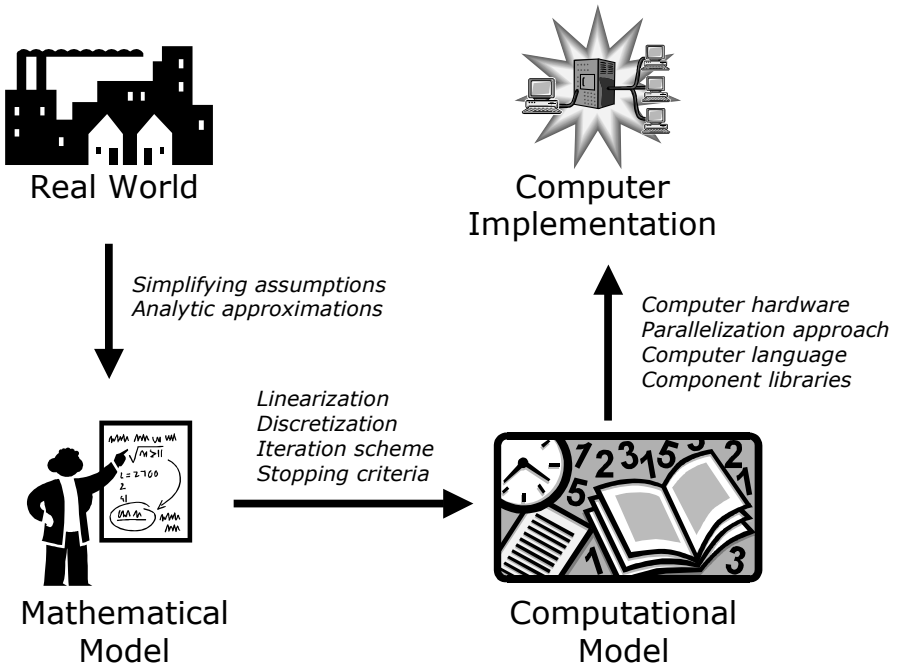

Figure 2.1. *A model of computational modeling.*

- **From Real World to Mathematical Model**
  A length scale is selected which will allow the determination of the desired results using a reasonable amount of resources, for example, atomic scale (nanometers) or macro scale (kilometers). Next, the physical quantities relevant to the study, such as temperature and pressure, are selected (and all other effects implicitly discarded). Then, the physical principles underlying the real world system, such as conservation of energy and mass, lead to mathematical relations, typically partial differential equations (PDEs), that express the mathematical model. Additional mathematical approximations may be introduced to further simplify the model. *Approximations can include:* discarded effects, inadequately modeled effects (e.g., discarded terms in equations, linearization).

- **From Mathematical Model to Computational Model**
  The equations expressing the mathematical model are typically set in some infinite dimensional space. In order to admit a numerical solution, the problem is transformed to a finite dimensional space by some discretization process. Finite differences and finite elements are examples of discretization methods for PDE models. In

such computational models one must select an order of approximation for derivatives. One also introduces a computational grid of some type. It is desirable that the discrete model converges to the continuous model as either the mesh width approaches zero or the order of approximation approaches infinity. In this way, accuracy can be controlled using these parameters of the numerical method. A specification for how to solve the discrete equations must also be provided. If an iterative method is used (which is certainly the case for nonlinear problems), then the solution is obtained only in the limit, and hence a criterion for stopping the iteration must be specified. *Approximations can include:* discretization of domain, truncation of series, linearization, stopping before convergence.

- **From Computational Model to Computer Implementation**
  The computational model and its solution procedure are implemented on a particular computer system. The algorithm may be modified to make use of parallel processing capabilities or to take advantage of the particular memory hierarchy of the device. Many arithmetic operations are performed using floating-point arithmetic. *Approximations can include:* floating-point arithmetic, approximation of standard mathematical or special functions (typically via calls to library routines).

2.2 Verification and Validation

If one is to use the results of a computer simulation, then one must have confidence that the answers produced are correct. However, absolute correctness may be an elusive quantity in computer simulation. As we have seen, there will always be uncertainty, uncertainty in the mathematical model, the computational model, in the computer implementation, and in the input data. A more realistic goal is to carefully characterize this uncertainty. This is the main goal of verification and validation.

- **Code verification**
  This is the process of determining the extent to which the computer implementation corresponds to the computational model. If the latter is expressed as a *specification*, then code verification is the process of determining whether an implementation corresponds to its lowest level algorithmic specification. In particular, we ask whether the specified algorithm has been correctly implemented, not whether it is an effective algorithm.

- **Solution verification**
  This is the process of determining the extent to which the computer implementation corresponds to the mathematical model. Assuming that the code has been verified (i.e., that the algorithm has been correctly implemented), solution verification asks whether the underlying numerical methods correctly produce solutions to the abstract mathematical problem.

- **Validation**
  This is the process of determining the extent to which the computer implementation corresponds to the real world. If solution verification has already been demonstrated,

then the validation asks whether the mathematical model is effective in simulating those aspects of the real world system under study.

Of course, neither the mathematical nor the computational model can be expected to be valid in all regions of their own parameter spaces. The validation process must confirm these regions of validity.

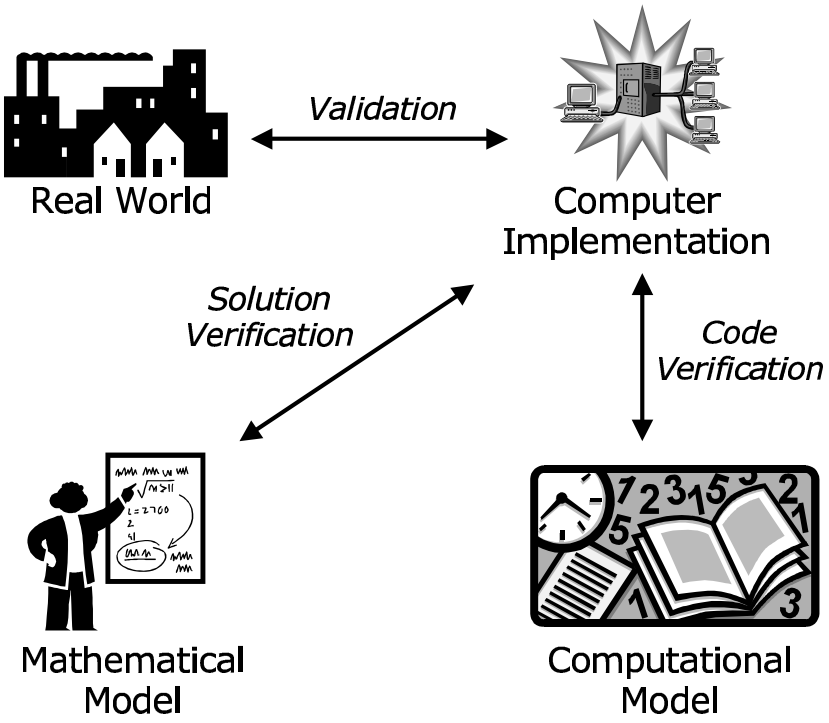

Figure 2.2. *A model of computational validation.*

Figure 2.2 illustrates how verification and validation are used to quantify the relationship between the various models in the computational science and engineering process. In a rough sense, validation is the aim of the application scientist (e.g., the physicist or chemist) who will be using the software to perform virtual experiments. Solution verification is the aim of the numerical analyst. Finally, code verification is the aim of the programmer.

There is now a large literature on the subject of verification and validation. Nevertheless, the words themselves remain somewhat ambiguous, with different authors often assigning slightly different meanings. For software in general, the IEEE adopted the following definitions in 1984 (they were subsequently adopted by various other organizations

and communities, such as the ISO[14]).

- Verification: The process of evaluating the products of a software development phase to provide assurance that they meet the requirements defined for them by the previous phase.

- Validation: The process of testing a computer program and evaluating the results to ensure compliance with specific requirements.

These definitions are general, in that "requirements" can be given different meaning for different application domains. For computational simulation, the US Defense Modeling and Simulation Office (DMSO) proposed the following definitions (1994), which were subsequently adopted in the context of computational fluid dynamics by the American Institute of Aeronautics and Astronautics [AIAA, 1998].

- Verification: The process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model.

- Validation: The process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended users of the model.

The DMSO definitions can be regarded as special cases of the IEEE ones, given appropriate interpretations of the word "requirements" in the general definitions. In the DMSO proposal, the verification is with respect to the requirements that the implementation should correctly realize the mathematical model. The DMSO validation is with respect to the requirement that the results generated by the model should be sufficiently in agreement with the real world phenomena of interest, so that they can be used for the intended purpose.

The definitions used in the present book, given in the beginning of this chapter, differ from those of DMSO in that they make a distinction between the computational model and the mathematical model. In our opinion, this distinction is so central in scientific computing that it deserves to be made explicit in the verification and validation process. The model actually implemented in the computer program is the computational one. Consequently, according to the IEEE definition, validation is about testing that the *computational* model fulfills certain requirements, ultimately those of the DMSO definition of validation. The full validation can then be divided into two levels. The first level of validation (solution verification) will be to demonstrate that the computational model is a sufficiently accurate representation of the mathematical one. The second level of validation is to determine that the mathematical model is sufficiently effective in reproducing properties of the real world.

The book by Roache [1998] and the more recent paper by Oberkampf and Trucano [2000] present extensive reviews of the literature in computational validation, arguing for the separation of the concepts of error and uncertainty in computational simulations. A special issue of *Computing in Science and Engineering*, see Trucano and Post [2004], has been devoted to this topic.

[14] International Standards Organization

2.3 Errors in Software

When we ask whether a program is "correct" we want to know whether it faithfully follows its lowest-level specifications. Code verification is the process by which we establish correctness in this sense. In order to understand the verification process, it is useful to be mindful of the most typical types of errors encountered in scientific software. Scientific software is prone to many of the same problems as software in other areas of application. In this section we consider these. Problems unique to scientific software are considered in Section 2.5.

Chapter 5 of the book by Telles and Hsieh [2001] introduces a broad classification of bugs organized by the original source of the error, i.e., how the error gets into the program, that they call "Classes of Bugs". We summarize them in the following list.

- *Requirement bugs.*
  The specification itself could be inadequate. For example, it could be too vague, missing a critical requirement, or have two requirements in conflict.
- *Implementation bugs.*
  These are the bugs in the logic of the code itself. They include problems like not following the specification, not correctly handling all input cases, missing functionality, problems with the graphic user interface, improper memory management, and coding errors.
- *Process bugs.*
  These are bugs in runtime environment of the executable program, such as improper versions of dynamic linked libraries or broken databases.
- *Build bugs.*
  These are bugs in the procedure used to build the executable program. For example, a product may be built for a slightly wrong environment.
- *Deployment bugs.*
  These are problems with the automatic updating of installed software.
- *Future planning bugs.*
  These are bugs like the year 2000 problem, where the lifetime of the product was underestimated, or the technical development went faster than expected (e.g., the 640 KiB[15] limit of MS-DOS).
- *Documentation bugs.*
  The software documentation, which should be considered as an important part of the software itself, might be vague, incomplete, or inaccurate. For example, when providing information on a library routine's procedure call it is important to provide not only the meaning of each variable, but also its exact type, as well as any possible side effects from the call.

Telles and Hsieh [2001] also provide a list of common bugs in the implementation of software. We summarize them in the following list.

[15] "Ki" is the IEC standard for the factor $2^{10} = 1024$, corresponding to "k" for the factor 1000, and "B" stands for "byte". See for example `http://physics.nist.gov/cuu/Units/binary.html`.

- *Memory or resource leaks.*
  A memory leak occurs when memory is allocated but not deallocated when it is no longer required. It can cause the memory associated with long-running programs to grow in size to the point that they overwhelm existing memory. Memory leaks can occur in any programming language, and are sometimes caused by programming errors.
- *Logic errors.*
  A logic error occurs when a program is syntactically correct but does not perform according to the specification.
- *Coding errors.*
  Coding errors occur when an incorrect list of arguments to a procedure is used, or a part of the intended code is simply missing. Others are more subtle. A classical example of the latter is the Fortran statement `DO 25 I = 1.  10` which in Fortran 77 (and earlier, and also in the obsolescent Fortran 90/95 fixed form) assigns the variable `DO25I` the value 1.10 instead of creating the intended `DO` loop, which requires the period to be replaced with a comma.
- *Memory overruns.*
  Computing an index incorrectly can lead to the access of memory locations outside the bounds of an array, with unpredictable results. Such errors are less likely in modern high level programming languages, but may still occur.
- *Loop errors.*
  Common cases are unintended infinite loops, off-by-one loops (i.e., loops executed once too often or not enough), and loops with improper exits.
- *Conditional errors.*
  It is quite easy to make a mistake in the logic of an *if-then-else-endif* construct. A related problem arises in some languages, such as Pascal, that do not require an explicit *endif*.
- *Pointer errors.*
  Pointers may be uninitialized, deleted (but still used), or invalid (pointing to something that has been removed).
- *Allocation errors.*
  Allocation and deallocation of objects must be done according to proper conventions. For example, if you wish to change the size of an allocated array in Fortran, you must first check if it is allocated, then deallocate it (and lose its content), and finally allocate to its correct size. Attempting to reallocate an existing array will lead to an error condition detected by the runtime system.
- *Multithreaded errors.*
  Programs made up of multiple independent and simultaneously executing threads are subject to many subtle and difficult to reproduce errors sometimes known as race conditions. These occur when two threads try to access or modify the same memory address simultaneously, but correct operation requires a particular order of access.
- *Timing errors.*
  A timing error occurs when two events are designed and implemented to occur at a certain rate or within a certain margin. They are most common in connection with

interrupt service routines and are restricted to environments where the clock is important. One symptom is input or output hanging and not resuming.

- *Distributed application errors.*
  Such an error is defined as an error in the interface between two applications in a distributed system.
- *Storage errors.*
  These errors occur when a storage device gets a soft or hard error and is unable to proceed.
- *Integration errors.*
  These occur when two fully tested and validated individual subsystems are combined, but do not cooperate as intended when combined.
- *Conversion errors.*
  Data in use by the application might be given in the wrong format (integer, floating point, ...) or in the wrong units (m, cm, feet, inches, ...). An unanticipated result from a type conversion can also be classified as a conversion error. A problem of this nature occurs in most traditional compiled programming languages by the assignment of `1/2` to a variable of floating-point type. The rules of integer arithmetic usually state that when two integers are divided there is an integer result. Thus, if the variable `A` is of a floating-point type, then the statement `A = 1/2` will result in an integer zero, converted to a floating-point zero assigned to the variable `A`, since `1` and `2` are integer constants, and integer division is rounded to zero.
- *Hard-coded lengths or sizes.*
  If sizes of objects like arrays are defined to be of a fixed size, then care must be taken that no problem instance will be permitted a larger size. It is best to avoid such hard-coded sizes, either by using allocatable arrays that yield a correctly sized array at runtime, or by parameterizing object sizes so that they are easily and consistently modified.
- *Version bugs.*
  In this case the functionality of a program unit or a data storage format is changed between two versions, without backward compatibility. In some cases individual version changes may be compatible, but not over several generations of changes.
- *Inappropriate reuse bugs.*
  Program reuse is normally encouraged, both to reduce effort and to capitalize upon the expertize of subdomain specialists. However, old routines that have been carefully tested and validated under certain constraints may cause serious problems if those constraints are not satisfied in a new environment. It is important that high standards of documentation and parameter checking be set for reuse libraries to avoid incompatibilities of this type.
- *Boolean bugs.*
  Telles and Hsieh [2001] note on page 159 that "Boolean algebra has virtually nothing to do with the equivalent English words. When we say 'and', we really mean the Boolean 'or' and vice versa." This leads to misunderstandings with both users and programmers. Similarly, the meaning of 'true' and 'false' in the code may be unclear.

Looking carefully at the examples of the Patriot missile, Section 1.4.1.1, and the Ariane 5 rocket, Section 1.4.2, we observe that in addition to the obvious *conversion errors* and *storage errors*, *inappropriate reuse bugs* were also involved. In each case the failing software was taken from earlier and less advanced hardware equipment, where the software had worked well for many years. They were both well tested, but not in the new environment.

2.4 Precision, Accuracy and Reliability

Many of the bugs listed in the previous section lead to anomalous behavior that is easy to recognize, i.e., the results are clearly wrong. For example, it is easy to see if the output of a program to sort data is really sorted. In scientific computing things are rarely so clear. Consider a program to compute the roots of a polynomial. Checking the output here seems easy; one can evaluate the function at the computed points. However, the result will seldom be exactly zero. How close to zero does this residual have to be to consider the answer correct? Indeed, for ill-conditioned polynomials the relative sizes of residuals provide a very unreliable measure of accuracy. In other cases, such as the evaluation of a definite integral, there is no such "easy" method.

Verification and validation in scientific computing, then, is not a simple process that gives yes or no as an answer. There are many gradations. The concepts of accuracy and reliability are used to characterize such gradations in the verification and validation of scientific software. In everyday language the words accuracy, reliability and the related concept of precision are somewhat ambiguous. When used as quality measures they should not be ambiguous. We use the following definitions.

- **Precision** refers to the number of digits used for arithmetic, input, and output.
- **Accuracy** refers to the absolute or relative error of an approximate quantity.
- **Reliability** measures how "often" (as a percentage) the software fails, in the sense that the true error is larger than what is requested.

Accuracy is a measure of the quality of the result. Since achieving a prescribed accuracy is rarely easy in numerical computations, the importance of this component of quality is often underweighted.

We note that determining accuracy requires the comparison to something external (the "exact" answer). Thus, stating the accuracy requires that one specifies what one is comparing against. The "exact" answer may be different for each of the computational model, the mathematical model, and the real world. In this book most of our attention will be on solution verification; hence we will be mostly concerned with comparison to the mathematical model.

To determine accuracy one needs a means of measuring (or estimating) error. Absolute error and relative error are two important such measures. *Absolute error* is the magnitude of the difference between a computed quantity x and its true value x^*, i.e.,

$$|x - x^*|. \tag{2.1}$$

Relative error is the ratio of absolute error to the magnitude of the true value, i.e.,

$$|x - x^*|/|x^*|. \tag{2.2}$$

Relative error provides a method of characterizing the percentage error; when the relative error is less than one, the negative of the $\log_{10}$ of the relative error gives the number of significant decimal digits in the computer solution. Relative error is not so useful a measure as x^* approaches 0; one often switches to absolute error in this case. When the computed solution is a multi-component quantity, such as a vector, then one replaces the absolute values by an appropriate norm.

The terms precision and accuracy are frequently mixed. Furthermore, the misconception that high precision implies high accuracy is almost universal.

Dahlquist and Björck [1980] use the term precision for the accuracy with which the basic arithmetic operations $+$, $-$, $\times$, and $/$ are performed by the underlying hardware. For floating-point operations this is given by the unit roundoff $\mathbf{u}$[16]. But even on that, there is no general agreement. One should be specific about the rounding mode used.

The difference between (traditional) rounding and truncation played an amusing role in the 100-digit challenge posed by Trefethen [2002b,a]. Trefethen did not specify whether digits should be rounded or truncated when he asked for "correct digits". First he announced 18 winners, but he corrected this a few days later to 20. In the interview included in Bornemann et al. [2004] he explains that two teams persuaded him that he had misjudged one of their answers by 1 digit. This was a matter of rounding instead of truncating.

Reliability in scientific software is considered in detail in Chapter 13. **Robustness** is a concept related to reliability that indicates how "gracefully" the software fails *(this is non-quantitative)*, and also its sensitivity to small changes in the problem *(that's related to condition numbers)*. A robust program knows when it might have failed, and reports that fact.

2.5 Numerical Pitfalls Leading to Anomalous Program Behavior

In this section we illustrate some common pitfalls unique to scientific computing that can lead to the erosion of accuracy and reliability in scientific software. These "numerical bugs" arise from the complex structure of the mathematics underlying the problems being solved, and the sometimes fragile nature of the numerical algorithms and floating-point arithmetic necessary to solve them. Such bugs are often subtle and difficult to diagnose. Some of these are described more completely in Chapter 4.

- *Improper treatment of constants with infinite decimal expansions.*
  The coding of constants with infinite decimal expansions, like π, $\sqrt{2}$, or even $1/9$, can have profound effects on the accuracy of a scientific computation. One will never achieve 10 decimal digits of accuracy in a computation in which π is encoded as 3.14159 or $1/9$ is encoded as 0.1111. To obtain high accuracy and portability such constants should, whenever possible, be declared as constants (and thus computed at compile time) or be computed at runtime. In some languages, e.g. MATLAB, π is stored to double precision accuracy and is available by a function call.

[16] The unit roundoff $\mathbf{u}$ can roughly be considered as the largest positive floating-point number $\mathbf{u}$, such that $1+\mathbf{u}=1$ in computer arithmetic. Because repeated rounding may occur this is not very useful as a strict definition. The formal definition of $\mathbf{u}$ is given by equation (4.1) on page 45.

For the constants above we can in Fortran 95 use the working precision `wp`, with at least 10 significant decimal digits:

```
integer, parameter :: wp = selected_real_kind(10)
real(kind=wp), parameter :: one = 1.0_wp, two = 2.0_wp,  &
   & four = 4.0_wp,  ninth = 1.0_wp/9.0_wp
real(kind=wp) :: pi, sqrt2
pi = four*atan(one)
sqrt2 = sqrt(two)
```

- *Testing on floating-point equality.*
  In scientific computations approximations and roundoff lead to quantities that are rarely exact. So, testing whether a particular variable that is the result of a computation is 0.0 or 1.0 is rarely correct. Instead, one must determine what interval around 0.0 or 1.0 is sufficient to satisfy the criteria at hand, and then test for inclusion in that interval. See, e.g., Example 1.1.
- *Inconsistent precision.*
  The IEEE standard for floating-point arithmetic defined in [ISO, 1989a] requires the availability of at least two different arithmetic precisions. If you wish to obtain a correct result in the highest precision available, then it is usually imperative that all floating-point variables and constants are in (at least) that precision. If variables and constants of different precisions (i.e., different floating-point types) are mixed in an arithmetic expression, this can result in a loss of accuracy in the result, dropping it to that of the lowest precision involved.
- *Faulty stopping criteria.*
  See, e.g., Example 1.1.
- *Not obviously wrong code.*
  A code can be wrong but it can still work more or less correctly. That is, the code may produce results that are acceptable, though they are somewhat less accurate than expected, or are generated more slowly than expected. This can happen when small errors in the coding of arithmetic expressions are made. For example, if one makes a small mistake in coding a Jacobian in Newton's method for solving nonlinear systems, the program may still converge to the correct solution, but, if it does, it will do so more slowly than the quadratic convergence that one expects of Newton's method.
- *Not recognizing ill-conditioned problems.*
  Problems are ill-conditioned when their solutions are highly sensitive to perturbations in the input data. In other words, small changes in the problem to be solved (such as truncating an input quantity to machine precision) leads to a computational problem whose exact solution is far away from the solution to the original problem. Ill-conditioning is an intrinsic property of the problem which is independent of the method used to solve it. Robust software will recognize ill-conditioned problems and will alert the user. See Sections 4.3 and 4.4 for illuminating examples.

- *Unstable algorithms and regions of instability.*
  A method is unstable when rounding errors are magnified without bound in the solution process. Some methods are stable only for certain ranges of its input data. Robust software will either use stable methods, or will notify the user when the input is outside the region of guaranteed stability. See Section 4.4.2 for examples.

2.6 Methods of Verification and Validation

In this section we summarize some of the techniques that are used in the verification and validation of scientific software. Many of these are well-known in the field of software engineering; see Adrion et al. [1982] and Telles and Hsieh [2001], for example. Others are specialized to the unique needs of scientific software; see Roache [1998] for a more complete presentation. We emphasize that none of these techniques is foolproof. It is rare that correctness of scientific software can be rigorously demonstrated. Instead, the verification and validation process provides a series of techniques, each of which serves to increase our confidence that the software is behaving in the desired manner.

2.6.1 Code verification

In code verification we seek to determine how faithfully the software is producing the solution to the computational model, i.e., to its lowest level of specification. In effect, we are asking whether the code correctly implements the specified numerical procedure. Of course, the numerical method may be ineffective in solving the target mathematical problem; that is not the concern at this stage.

Sophisticated software engineering techniques have been developed in recent years to improve and automate the verification process. Such techniques, known as *formal methods*, rely on mathematically rigorous specifications for the expected behavior of software. Given such a specification, one can (a) prove theorems about the projected program's behavior, (b) automatically generate much of the code itself, and/or (c) automatically generate tests for the resulting software system. See [Clarke and Wing, 1996], for example. Unfortunately, such specifications are quite difficult to write, especially for large systems. In addition, such specifications do not cope well with the uncertainties of floating-point arithmetic. Thus, they have rarely been employed in this context. A notable exception is the formal verification of low-level floating-point arithmetic functions. See, for example, Harrison [2000, 2003]. Gunnels et al. [2001] have employed formal specifications to automatically generate linear algebra kernels. In our discussion we will concentrate on more traditional code verification techniques. The two general approaches that we will consider are code analysis and testing.

2.6.1.1 Code Analysis

Analysis of computer code is an important method of exposing bugs. Software engineers have devised a wealth of techniques and tools for analyzing code.

One effective means of detecting errors is to have the code read and understood by someone else. Many software development organizations use formal *code reviews* to uncover misunderstandings in specifications or errors in logic. These are most effectively

done at the component level, i.e., for portions of code that are easier to assimilate by persons other than the developer.

Static code analysis is another important tool. This refers to automated techniques for determining properties of a program by inspection of the code itself. Static analyzers study the flow of control in the code to look for anomalies, such as "dead code" (code that cannot be executed) or infinite loops. They can also study the flow of data, locating variables that are never used, variables used before they are set, variables defined multiple times before their first use, or type mismatches. Each of these are symptoms of more serious logic errors. Tools for static code analysis are now widely available; indeed, many compilers have options that provide such analysis.

Dynamic code analysis refers to techniques that subject a code to fine scrutiny as it is executed. Tools that perform such analysis must first transform the code itself, inserting additional code to track the behavior of control flow or variables. The resulting "instrumented" code is then executed, causing data to be collected as it executes. Analysis of the data can be done either interactively or as a postprocessing phase. Dynamic code analysis can be used to track the number of times that program units are invoked and the time spent in each. Changes in individual variables can be traced. Assertions about the state of the software at any particular point can be inserted and checked at runtime. Interactive code debuggers, as well as code profiling tools, are now widely available to perform these tasks.

2.6.1.2 Software Testing

Exercising a code actually performing the task for which it was designed, i.e., testing, is an indispensable component of software verification. Verification testing requires a detailed specification of the expected behavior of the software to all of its potential inputs. Tests are then designed to determine whether this expected behavior is achieved.

Designing test sets can be quite difficult. The tests must span all of the functionality of the code. To the extent possible, they should also exercise all paths through the code. Special attention should be paid to provide inputs that are boundary cases or that trigger error conditions.

Exhaustive testing is rarely practical, however. Statistical design of experiments [Box et al., 1978, Montgomery, 2001] provides a collection of guiding principles and techniques that comprise a framework for maximizing the amount of useful information resident in a resulting data set, while attending to the practical constraints of minimizing the number of experimental runs. Such techniques have begun to be applied to software testing. Of particular relevance to code verification are orthogonal fractional factorian designs, as well as the covering designs; see [Dalal and Mallows, 1998].

Because of the challenges in developing good test sets, a variety of techniques have been developed to evaluate the test sets themselves. For example, dynamic analysis tools can be used to assess the extent of code coverage provided by tests. *Mutation testing* is another valuable technique. Here, faults are randomly inserted into the code under test. Then the test suite is run, and the ability of the suite to detect the errors is thereby assessed.

Well-designed software is typically composed of a large number of components (e.g., procedures) whose inputs and outputs are limited and easier to characterize than the complete software system. Similarly, one can simplify the software testing process by first testing the behavior of each of the components in turn. This is called *component testing*

or *unit testing*. Of course, the interfaces between components are some of the most error-prone parts of software systems. Hence, component testing must be followed by extensive *integration testing*, which verifies that the combined functionality of the components is correct.

Once a software system attains a certain level of stability, changes to the code are inevitably made in order to add new functionality or to fix bugs. Such changes run the risk of introducing new bugs into code that was previously working correctly. Thus, it is important to maintain a battery of tests that extensively exercise all aspects of the system. When changes are applied to the code, then these tests are rerun to provide confidence that all other aspects of the code have not been adversely affected. This is termed *regression testing*. In large active software projects it is common to run regression tests on the current version of the code each night.

Elements of the computing environment itself, such as the operating system, compiler, number of processors, and floating-point hardware, also have an effect on the behavior of software. In some cases faulty system software can lead to erroneous behavior. In other cases errors in the software itself may not be exposed until the environment changes. Thus, it is important to perform exhaustive tests on software in each environment in which it will execute. Regression tests are useful for such testing.

Another useful technique is to enlist a team other than the developers to provide a separate evaluation of the software. The testing team obtains the requirements for the code, inspects it, analyzes it, develops tests, and runs them. Such a team often brings a new perspective which can uncover problems overlooked by the developers. One form of this approach is to enlist prospective users of the software to perform early testing; the alpha and beta releases now common in open source software development are examples of this.

The mathematical software community has established a similar type of evaluation process for reusable components. Since 1975 the *ACM Transactions on Mathematical Software* has used a formal refereeing process to evaluate software which is then published as part of the Collected Algorithms of the ACM [CALGO]. The journal also publishes independent certifications of previously published algorithms.

2.6.2 Sources of Test Problems for Numerical Software

Both the code verification and the solution verification steps require the use of test data, or test problems, with which to exercise the software or its components. Such tests are particular instances of the abstract mathematical problem to be solved. Methods for obtaining such data sets are somewhat problem-dependent. Nevertheless there are a few general techniques.

2.6.2.1 Analytical Solutions

A test problem is most useful if one knows its solution, since it is only then that we can assess the error in the computation with absolute certainty. In many cases there are problems whose analytical solutions are well-known that can be used for testing. If the software is to compute something straightforward like definite integrals, then one can test the software by integrating elementary functions. One can also find challenging functions in a table of integrals. In more complex mathematical models, such as those describing fluid

flow, analytical solutions may be more difficult to find, although there are often special cases (possibly simple or degenerate) whose solutions are known that can be used. It is sometimes difficult to find such problems that will exercise all aspects of a code, however.

In many cases it is possible to artificially construct any number of problems with known solutions. Consider the case of software for solving Poisson's equation,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \tag{2.3}$$

on the unit square $(0,1) \times (0,1)$ subject to Dirichlet boundary conditions (that is, $u(x,y)$ is specified on the boundary). To construct a test problem, first pick any (smooth) function u to use as the solution, for example,

$$u(x,y) = x + \sin(\pi x)\cos(\pi y). \tag{2.4}$$

Then, put the solution into the Equation (2.3), differentiating as necessary to produce a right-hand side function f, i.e.,

$$f(x,y) = -2\pi^2 \sin(\pi x)\cos(\pi y).$$

Similarly, one can obtain boundary conditions directly from Equation (2.4).

So, we now have the complete specification for a Poisson problem, i.e.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2\pi^2 \sin(\pi x)\cos(\pi y) \qquad \text{on } (0,1) \times (0,1)$$

subject to

$$\begin{array}{rcll} u &=& 0, & x = 0, \quad 0 < y < 1 \\ u &=& 1, & x = 1, \quad 0 < y < 1 \\ u &=& x + \sin \pi x, & 0 < x < 1, \quad y = 0 \\ u &=& x - \sin \pi x, & 0 < x < 1, \quad y = 1 \end{array}$$

The solution to this problem is given by Equation (2.4).

Such a procedure can be used to generate test problems for most operator equations, e.g., differential equations, integral equations, and systems of algebraic equations. The method is sometimes known as the *method of manufactured solutions*.

One can argue that the problems generated by this technique are not representative of real problems found in practice. However, if one is simply searching for bugs in the code, i.e., doing verification testing, then this procedure can be used, with care, to generate a wealth of problems that exercise all aspects of a code. For example, in our example of Poisson's equation, the artificiality comes from an unnatural forcing function f. However, the main part of the computation that one is testing is the assembly of a sparse matrix and the solution of the resulting linear system. The structure of this matrix depends only on the form of the differential operator and the boundary conditions, not on the forcing function.

Particularly helpful test problems for verification testing are those for which the underlying numerical method is exact. For example, Simpson's rule for numerical integration is exact when integrating cubic polynomials. Since the only numerical errors for such a problem are round-off errors, then (provided the underlying method is stable) one should

obtain a result whose error is of the same size as the machine precision. Using such problems it is easy to determine whether the underlying coding of the method is incorrect. Many numerical methods for operator equations are exact on spaces of polynomials; in these cases the method of manufactured solutions can be used to generate problems for which the solution should be exact to machine precision.

2.6.2.2 Test Set Repositories

A number of collections of test problems suitable for code verification and solution verification testing have been developed. These are often put together by communities of researchers interested in numerical methods for particular problem classes. Although the main interest here is the development of a set of standard problems or benchmarks to support the comparison of competing methods, such problems are also useful in verification and validation. Such test problem sets often contain problems whose exact solution is known. However, problems whose analytical solution is not known are sometimes also used. In the latter case test comparisons is typically made with best effort computations from the literature.

Some examples of test data set collections used in the numerical analysis community are provided in Table 2.1. More are available on the accompanying website `http://www.nsc.liu.se/wg25/book/`.

2.6.3 Solution Verification

In solution verification we try to determine the extent to which the program can be used to solve the abstract mathematical problem that has been posed. Since it is unrealistic to expect an exact solution to be obtained in scientific software, we must be more precise about what we mean here. Most numerical methods have some discretization parameter, such as a characteristic mesh size, h, or the number of terms in a critical approximating series or the number of iterations, n, such that the computed solution approaches the exact solution as $h \to 0$ or $n \to \infty$. A numerical method with this property is termed *convergent*. If a numerical method is convergent, then given a target accuracy, it will be possible to select method parameters (i.e., h or n) such that the code will deliver a solution to the problem with the desired accuracy. (Note that different values of h and n may be needed for different problem instances.) Of course, in practice computing resources such as CPU time, memory, or arithmetic precision may not be available to achieve arbitrarily high accuracy. The solution verification process, then, establishes what levels of numerical convergence can be expected in practice.

2.6.3.1 Convergence Testing

Testing numerical convergence is an important part of solution verification. To make our discussion concrete, consider the case of a numerical method with a discretization mesh size h. (Examples include methods for computing integrals or solving differential equations.) *Convergence testing* solves a test problem multiple times with different values of h to verify that some measure e of the norm of the error in the computed solution is approaching zero as h decreases.

Statistics http://www.itl.nist.gov/div898/strd/
Sparse Linear Algebra, [Boisvert et al., 1997] http://math.nist.gov/MatrixMarket/
Sparse Linear Algebra http://www.cise.ufl.edu/research/sparse/matrices/
Initial Value ODEs http://pitagora.dm.uniba.it/∼testset/
Initial Value ODEs http://www.ma.ic.ac.uk/∼jcash/IVP_software/readme.php
Stiff ODEs, [Hairer and Wanner, 1991] http://www.unige.ch/math/folks/hairer/testset/testset.html
Boundary Value ODEs http://www.ma.ic.ac.uk/∼jcash/BVP_software/readme.php
Elliptic PDEs, [Rice and Boisvert, 1985]
Multivariate Definite Integrals http://www.math.wsu.edu/faculty/genz/software/software.html
Optimization http://plato.la.asu.edu/bench.html
Optimization, [Floudas, 1999] http://titan.princeton.edu/TestProblems/
Optimization http://www-fp.mcs.anl.gov/otc/Guide/TestProblems/

Table 2.1. *Sources of Test Problems for Mathematical Software*

Not only can convergence testing be used to demonstrate the validity of the code, it can also be used to uncover errors in the code. Suppose that theory tells us that the norm of the error e behaves as follows,

$$e = ch^p \tag{2.5}$$

for some c independent of h. Such a method is said to have *order of convergence* p. Typically such error behavior is based upon an asymptotic analysis, i.e., the behavior is valid only for sufficiently small h. During the validation process, one can experimentally determine an observed order of convergence $\widehat{p}$ and determine whether the observed value approaches the theoretical value as h gets small, i.e., whether $\widehat{p} \to p$ as $h \to 0$. In particular, given distinct mesh sizes $h_i < h_{i-1}$ leading to computed errors $e_i < e_{i-1}$, respectively, the observed convergence rate can be computed from

$$\widehat{p}_i = \frac{\ln(e_i/e_{i-1})}{\ln(h_i/h_{i-1})} \tag{2.6}$$

If one computes this value for a sequence of decreasing h_i one should see $\widehat{p}_i \to p$; if not, then there may be a problem with the code. One instance is with the classical Runge-Kutta method for solving an ordinary differential equation (ODE), which wrongly coded may

give $\widehat{p} = 1$ instead of the theoretical $p = 4$. However, this result may also occur because the user's coding of the function defining the ODE has been done in such a way, as to make the definition less smooth than is required for the software to function at the higher order. Order of convergence $\widehat{p} = 1$ is often what results. But this error must also be addressed.

It is also possible to estimate the convergence rate in cases where the solution is unknown. If you do not know the exact solution you may perform three computations using the step lengths h, qh, q^2h and get the results $y(h), y(qh), y(q^2h)$. Now compare with the unknown solution y_0. Using the formula $error = y(h) - y_0 = ch^p$ several times you form

$$\frac{y(q^2h) - y(qh)}{y(qh) - y(h)} = \frac{c(q^2h)^p - c(qh)^p}{c(qh)^p - c(h)^p} = q^p \tag{2.7}$$

The value on the far left is a known computed quantity, and the parameter p (or rather $\widehat{p}$) on the far right is now the only unknown, and is easy to determine. You can therefore test if your program behaves as theory predicts.

A related testing technique is available when one is solving time-dependent systems of differential equations, such as fluid flow problems. Here one can often find problems with known unique steady-state solutions. One can use such problems to check for convergence to steady state as t gets large. Of course, this does not tell us that correct transient behavior was computed.

It may also be the case that the numerical method in use by the code does not always converge. The method may be stable only for certain combinations of the problem or method parameters. Similarly, approximations used in the method may only be valid for certain inputs. Solution verification testing must also determine regions of convergence for the software. To help determine good tests for determining these, the analyst should be familiar with the theoretical properties of the numerical method in use. Note that good software should be sensitive to these parameters and report failure on its own rather than simply deliver an incorrect result.

Chapter 6 of this volume describes a general-purpose tool, PRECISE, which can be helpful in performing numerical experiments to determine the quality of convergence of numerical software components. There are many other types of diagnostics that can be computed to help in the code verification and solution verification process. Often these rely on particular properties of the given problem domain. Examples of such indicators for the case of initial-value problems for ODEs are provided in Chapter 7.

2.6.3.2 Dynamic Analysis Methods

In many cases it is possible to develop procedures for estimating errors in computations. For example, the problem may be solved using two different computational meshes, the second a refinement of the first. The difference between the two solutions may then be taken as an estimate of the error for the solution on the coarser mesh. Other procedures use the known form of the theoretical error to develop estimation procedures. Providing an estimate of the error is a very useful feature in mathematical software components, although the error estimate itself must be the subject of verification testing.

Such *a posteriori* error estimates can be used to improve the functionality of the software itself, of course. For example, error estimates can be used to make a code *self-adaptive*. Given a target error tolerance from the user, the software can solve the problem

on a sequence of grids until the error estimate is small enough. In other cases one can use local error estimates to automatically determine where in the computational domain the grid should be refined, or where it can be made coarser. If estimates are good, then the software can make optimal use of resources to solve a given problem. Such adaptive methods have become quite common in computing integrals and in solving differential equations. Adaptive software of this type poses particular difficulties for verification and validation since the adaptivity may hide problems with the implementation of the numerical methods that underlie it.

Another type of dynamic error monitoring is the use of *interval arithmetic*. Here one replaces floating-point numbers with intervals. The width of the interval indicates the uncertainty in the value. All arithmetic operations can be performed using intervals. By using intervals one can automatically track the uncertainty in input quantities through a computation, providing an uncertainty in the result. A variety of tools for computing with intervals are now available. Details on the use of intervals is provided in Chapter 9. When used carefully, intervals can provide the basis for computer-assisted proofs and self-validating numerical methods, see Chapter 10. Such methods can take reliability of numerical software to new levels.

2.6.4 Validation

Validation is the process of determining the extent to which the computer implementation corresponds to the real world system being modeled. This is the most important concern for the scientist who wishes to predict the behavior of a physical system. (For software designed only to solve a class of abstract mathematical problems, validation does not apply.)

In some cases, computational models are useful only for qualitative understanding. In other cases, specific levels of quantitative understanding is desired. For the developer of a multi-purpose modeling system, the goal of validation testing is to determine the regions within parameter space that the model produces acceptable accuracy for its intended purpose. For someone using a modeling system for a particular application, the goal of validation is to characterize the uncertainty of the results. Methods for validation are typically highly problem-dependent.

For simulation software, the most meaningful validation procedures involve the comparison with data from real physical systems. However, collecting such real world data is often challenging, and sometimes impossible. If the model is computing something simple, like the density of a fluid system, then there may be a wealth of existing benchmark data upon which to draw. But, if the simulation output is more complex, such as the flow field about an aircraft wing, then it can be difficult to obtain suitable benchmark data for comparison. Experimental conditions in previously published data may not have been specified in enough detail to be able to specify the mathematical model parameters accurately. In other cases, the exact experimental conditions may be difficult to model. Conversely, experimentalists may not be able to match the somewhat idealized conditions that may be assumed in the mathematical model. Some problem domains are just impossible to realize experimentally; consider long-term climate change!

Even when detailed comparisons of results with real systems is impractical, there are often well-known quantities that can be used as indicators of success, such as the time until a particular event occurs. In most problems a variety of consistency checks are also

possible. For example, one can check whether energy or mass is properly conserved, but these measures can also be deceptive if the conservation law is built into the underlying numerical model, i.e. the solution may be nowhere near as accurate as the quantity being conserved. Indeed, it is possible for a solution to be totally incorrect yet a physical quantity be conserved to high accuracy.

Another technique that can be used in these cases is comparison with other computational models and simulations. Such comparisons are most meaningful when the underlying mathematical model and the numerical solution methods are completely different. A recent development in the area of finite element software is encouraging in this regard. Preprocessing tools are now available that take as input a description of a problem and computational grid and are capable of producing the input for several different finite element solution packages for the given problem. This greatly eases the ability of users to compare results of multiple packages.

Finally, a sound statistical basis can be critical to the success of the validation process. For example, statistical design of experiments [Box et al., 1978, Montgomery, 2001] is very useful in maximizing the amount of useful information while minimizing the number of experiments. Historically, experiment design has played an important role in the physical sciences, and will no doubt play an increasingly important role in validation for the computational sciences. Such techniques are especially important when (a) models have large numbers of input parameters that must be studied, (b) computational experiments require long runs on high-end computers, and/or (c) physical experiments are difficult or expensive to perform.

Statistical techniques originally developed for physical metrology may also have a role here. In such studies, the analysis of measurement errors from all sources (both random and systematic), and their combination into a formal "error budget" is performed, in order to develop rigorous estimates of the uncertainties in a measurement [Fuller, 1987, Dietrich, 1991].

Bayesian inference techniques have also been successfully exploited to develop procedures for quantifying uncertainties in complex computer simulations. Here one develops probability models for distributions of errors from various sources, such as errors in models, input parameters, and numerical solutions to governing equations. By sampling from the relevant probability distributions a series of computer experiments are specified and performed. The Bayesian framework provides systematic methods for updating knowledge of the system based on data from such experiments. See DeVolder et al. [2002] for a description of how this has been applied to problems as wide ranging as flow in porus media and shock wave interactions.