Part II

Code verification

As we begin to address issues of validation in Part IV: Model validation and prediction (Chapters 10–13), the focus will be on whether the proper mathematical model has been chosen, where mathematical model refers to the governing partial differential or integral equations along with any auxiliary algebraic relations. Since the exact solutions to complex mathematical models are extremely rare, we generally use numerical solutions to the discretized equations as a surrogate for the exact solutions. Verification provides a framework for quantifying the numerical approximation errors in the discrete solution relative to the exact solution to the mathematical model. Since verification deals purely with issues of mathematics, no references to the actual behavior of real-world systems or experimental data will be found in Chapters 4 through 9.

Code verification ensures that the computer program (alternatively referred to as the computer code) is a faithful representation of the original mathematical model. It is accomplished by employing appropriate software engineering practices (Chapter 4), and by using order verification (Chapter 5) to ensure that there are no mistakes in the computer code or inconsistencies in the discrete algorithm. This part of the book dealing with code verification is completed by a discussion of exact solutions to mathematical models in Chapter 6. A key part of Chapter 6 is the Method of Manufactured Solutions (MMS), which is a powerful method for performing order verification studies on complex, nonlinear, coupled sets of partial differential or integral equations.

4

Software engineering

Software engineering encompasses the tools and methods for defining requirements for, designing, programming, testing, and managing software. It consists of monitoring and controlling both the software processes and the software products to ensure reliability. Software engineering was developed primarily from within the computer science community, and its use is essential for large software development projects and for high-assurance software systems such as those for aircraft control systems, nuclear power plants, and medical devices (e.g., pacemakers).

The reader may wonder at this point why a book on verification and validation in scientific computing includes a chapter on software engineering. The reason is that software engineering is critical for the efficient and reliable development of scientific computing software. Failure to perform good software engineering throughout the life cycle of a scientific computing code can result in much more additional code verification testing and debugging. Furthermore, it is extremely difficult to estimate the effect of *unknown* software defects on a scientific computing prediction (e.g., see Knupp *et al.*, 2007). Since this effect is so difficult to quantify, it is prudent to minimize the introduction of software defects through good software engineering practices.

Software engineers will no doubt argue that we have it backwards: *code verification* is really just a part of the software engineering process known as *software verification and validation*. While this is technically true, the argument for instead including software engineering as a part of code verification can be made as follows. Recall that we have defined scientific computing as the approximate solution to mathematical models consisting of partial differential or integral equations. The "correct" answer that should result from running a scientific computing code on any given problem is therefore not known: it will depend on the chosen discretization scheme, the chosen mesh (both its resolution and quality), the iterative convergence tolerance, the machine round-off error, etc. Thus special procedures must be used to test scientific computing software for coding mistakes and other problems that do not need to be considered for more general software. The central role of code verification in establishing the correctness of scientific computing software justifies our inclusion of software engineering as part of the code verification process. Regardless of the relation between software engineering and code verification, they are both important factors in developing and maintaining reliable scientific computing codes.

Computational scientists and engineers generally receive no formal training in modern software engineering practices. Our own search of the software engineering literature found a large number of contributions in various textbooks, on the web, and in scientific articles – mostly dominated by software engineering practices and processes that do not consider some of the unique aspects of scientific software. For example, most software engineering practices are driven by the fact that data organization and access is the primary factor in the performance efficiency of the software, whereas in scientific computing the speed of performing floating point operations is often the overriding factor. The goal of this chapter is to provide a brief overview of recommended software engineering practices for scientific computing. The bulk of this chapter can be applied to all scientific computing software projects, large or small, whereas the final section addresses additional software engineering practices that are recommended for large software projects.

Software engineering is an enormously broad subject which has been addressed by numerous books (e.g., Sommerville, 2004; McConnell, 2004; Pressman, 2005) as well as a broad array of content on the World Wide Web (e.g., SWEBOK, 2004; Eddins, 2006; Wilson, 2009). In 1993, a comprehensive effort was initiated by the Institute of Electrical and Electronics Engineers (IEEE) Computer Society to "establish the appropriate set(s) of criteria and norms for professional practice of software engineering upon which industrial decisions, professional certification, and educational curricula can be based" (SWEBOK, 2004). The resulting book was published in 2004 and divides software engineering into ten knowledge areas which comprise the Software Engineering Body of Knowledge (SWEBOK). In addition, there have been several recent workshops which address software engineering issues specifically for scientific computing (SE-CSE 2008, 2009) and high-performance computing (e.g., SE-HPC, 2004). In this chapter we will cover in detail the following software engineering topics: software development, version control, software testing, software quality and reliability, software requirements, and software management. An abbreviated discussion of many of the topics presented in this chapter can be found in Roy (2009).

4.1 Software development

Software development encompasses the design, construction, and maintenance of software. While software testing should also be an integral part of the software development process, we will defer a detailed discussion of software testing until a later section.

4.1.1 Software process models

A software process is an activity that leads to the creation or modification of software products. There are three main software process models: the waterfall model, the iterative and incremental development model, and component-based software engineering (Sommerville, 2004). In the traditional *waterfall model*, the various aspects of the software development process (requirements specification, architectural design, programming,

testing, etc.) are decomposed into separate phases, with each phase beginning only after the previous phase is completed. In response to criticisms of the waterfall software development model, a competing approach called *iterative and incremental development* (also called the spiral model) was proposed. This iterative, or evolutionary, development model is based on the idea of interweaving each of the steps in the software development process, thus allowing customer feedback early in the development process through software prototypes which may initially have only limited capabilities. These software prototypes are then refined based on the customer input, resulting in software with increasing capability. A third model, *component-based software engineering*, can be used when a large number of reusable components are available, but often has only limited applicability to scientific computing (e.g., for linear solver libraries or parallel message passing libraries). Most modern software development models, such as the rational unified process (Sommerville, 2004) and agile software development (discussed later in this section), are based on the iterative and incremental development model.

4.1.2 Architectural design

Software architectural design is the process of identifying software sub-systems and their interfaces before any programming is done (Sommerville, 2004). The primary products of architectural design are usually documents (flowcharts, pseudocode, etc.) which describe the software subsystems and their structure. A software subsystem is defined as a subset of the full software system that does not interact with other subsystems. Each software subsystem is made up of components, which are subsets of the full system which interact with other components. Components may be based on a procedural design (subroutines, functions, etc.) or an object-oriented design, and both approaches are discussed in more detail in the next section.

4.1.3 Programming languages

There are a variety of factors to consider when choosing a programming language. The two main programming paradigms used in scientific computing are procedural programming and object-oriented programming. *Procedural programming* relies on calls to different procedures (routines, subroutines, methods, or functions) to execute a series of sequential steps in a given programming task. A significant advantage of procedural programming is that it is modular, i.e., it allows for reuse of procedures when tasks must be performed multiple times. In *object-oriented programming*, the program is decomposed into objects which interact with each other through the sending and receiving of messages. Objects typically make use of private data which can only be accessed through that object, thus providing a level of independence to the objects. This independence makes it easier to modify a given object without impacting other parts of the code. Object-oriented programming also allows for the reusability of components across the software system.

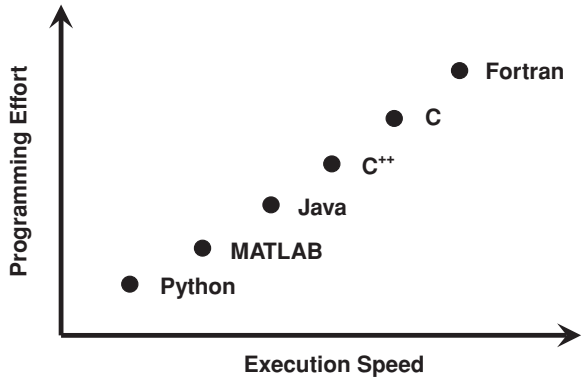

Figure 4.1 *Qualitative* example of programming effort versus execution speed (adapted from Wilson, 2009).

Most modern, higher-level programming languages used for scientific computing support both procedural and object-oriented programming. Programming languages that are primarily procedural in nature include BASIC, C, Fortran, MATLAB, Pascal, and Perl, while those that are primarily object-oriented include C++, Java, and Python. Procedural programming is often used when mathematical computations drive the design, whereas object-oriented programming is preferred when the problem is driven by complex data relationships.

Low level computing languages such as machine language and assembly language (often found in simple electronic devices) execute extremely fast, but require additional time and effort during the programming and debugging phases. One factor to consider is that high-level languages, which often make use of more natural language syntax and varying levels of programming abstraction, have the advantage of making programming complex software projects easier, but generally will not execute as fast as a lower-level programming language. A qualitative comparison of selected programming languages is shown in Figure 4.1 which compares programming effort to execution speed. In scientific computing, the higher-level programming languages such as Python, MATLAB, and Java are ideal for small projects and prototyping, while production-level codes are usually programmed in C, C++, or Fortran due to the faster execution speeds.

Another factor to consider when choosing a programming language is the impact on the software defect rate and the subsequent maintenance costs (Fenton and Pfleeger, 1997). Here we define a *software defect* as an error in the software that could potentially lead to software failure (e.g., incorrect result produced, premature program termination) and the *software defect rate* as the number of defects per 1000 lines of executable code. Evidence suggests that the software defect rate is at best weakly dependent on the choice of programming language (Hatton, 1997a). However, Hatton (1996) found that defects in object-oriented languages can be more expensive to find and fix, possibly by as much as a factor of three. The choice of compiler and diagnostic/debugging tool can also have

a significant impact on the software defect rate as well as the overall code development productivity.

Standards for programming languages are generally developed by a costly and complex process. However, most programming language standards still contain coding constructs that are prone to producing software failures. These failure-prone constructs can arise in a number of different ways (Hatton, 1997a) including simple oversight by the standards process, lack of agreement on the content of the standards, because the decision was explicitly made to retain the functionality provided by the construct, or because of errors in the programming language standards documentation. In some cases, less fault-prone subsets of a programming language exist which reduce or eliminate the presence of the dangerous coding constructs. One example of a safe subset for the C programming language is Safer C (Hatton, 1995).

4.1.4 Agile programming

Most software development processes call for the requirements specification, design, implementation, and testing of the software to be performed sequentially. In this approach, changes to requirements can be costly and lead to extensive delays since they will impact the entire software development process. One notable exception is *agile programming* (also referred to as rapid software development, see agilemanifesto.org/), where requirements specification, design, implementation, and testing occur simultaneously (Sommerville, 2004).

Agile programming is iterative in nature, and the main goal is to develop useful software quickly. Some features of agile programming methods include:

- concurrency of development activities,
- minimal or automatic design documentation,
- only high-priority user requirements specified up front,
- significant user involvement in the development process, and
- incremental software development.

One of the intended advantages of agile programming is to provide software delivery (although initially with reduced capability) that allows for user involvement and feedback during the software design process and not just after the final software product has been delivered. Agile programming methods are good for small and medium-size software development efforts, but their efficiency for larger software development efforts which generally require more coordination and planning is questionable (Sommerville, 2004). Agile programming appears to be particularly suited for small to moderate size scientific computing software projects (Allen, 2009).

A popular form of the agile programming approach is extreme programming, or XP (Beck, 2000), and is so-called because it takes the standard software engineering practices to their extreme. In XP, requirements are expressed as potential scenarios that lead to

software development tasks, which are then programmed by a pair of developers working as a team. Evidence suggests that pair programming productivity is similar to that of solo programmers (Williams and Kessler, 2003), but results in fewer errors since any code produced has necessarily undergone an informal software inspection process (Pressman, 2005). Unit tests (Section 4.3.3.1) must be developed for each task before the code is written, and all such tests must be successfully executed before integration into the software system, a process known as continuous integration testing (Duvall *et al.*, 2007). This type of test-first procedure also provides an implicit definition of the interface as well as proper usage examples of the component being developed. Software development projects employing XP usually have frequent releases and undergo frequent refactoring (Section 4.1.6) to improve quality and maintain simplicity. For an example of XP applied to scientific computing, see Wood and Kleb (2003).

4.1.5 Software reuse

Software reuse has become an important part of large software development projects (Sommerville, 2004). While its use in scientific computing is not as extensive, there are a number of areas in scientific computing where reuse is commonly found. Some examples of software reuse in scientific computing are: mathematical function and subroutine libraries (e.g., Press *et al.*, 2007), parallel message passing libraries (e.g., MPI), pre-packaged linear solvers such as the Linear Algebra Package (LAPACK) or the Portable Extensible Toolkit for Scientific Computation (PETSc), and graphics libraries.

4.1.6 Refactoring

Often, at the end of a software development effort, the developer realizes that choices made early in the software design phase have led to computationally inefficient or cumbersome programming. Refactoring is the act of modifying software such that the internal software structure is changed, but the outward behavior is not. Refactoring can reduce the complexity, computational time, and/or memory requirements for scientific software. However, refactoring should not be undertaken until a comprehensive test suite (Section 4.3.4) is in place to ensure that the external behavior is not modified and that programming errors are not introduced.

4.2 Version control

Version control tracks changes to source code or other software products. A good version control system can tell you what was changed, who made the change, and when the change was made. It allows a software developer to undo any changes to the code, going back to any prior version. This can be particularly helpful when you would like to reproduce

results from an earlier paper, report, or project, and merely requires documentation of the version number or the date the results were generated. Version control also provides a mechanism for incorporating changes from multiple developers, an essential feature for large software projects or projects with geographically remote developers. All source code should be maintained in a version control system, regardless of how large or small the software project (Eddins, 2006).

Some key concepts pertaining to version control are discussed below (Collins-Sussman *et al.*, 2009). Note that the generic descriptor "file" is used, which could represent not only source code and other software products, but also any other type of file stored on a computer.

Repository	single location where the current and all prior versions of the files are stored. The repository can only be accessed through check-in and check-out procedures (see below).
Working copy	the local copy of a file from the repository which can be modified and then checked in to the repository.
Check-out	the process of creating a working copy from the repository, either from the current version or an earlier version.
Check-in	a check-in (or commit) occurs when changes made to a working copy are merged into the repository, resulting in a new version.
Diff	a summary of the differences between a working copy and a file in the repository, often taking the form of the two files shown side-by-side with differences highlighted.
Conflict	occurs when two or more developers attempt to make changes to the same file and the system is unable to reconcile the changes. Conflicts generally must be resolved by either choosing one version over the other or by integrating the changes from both into the repository by hand.
Update	merges recent changes to the repository from other developers into a working copy.
Version	a unique identifier assigned to each version of the file held in the repository which is generated by the check-in process.

The basic steps that one would use to get started with a version control tool are as follows. First, a *repository* is created, ideally on a network server which is backed up frequently. Then a software project (directory structure and/or files) is *imported* to the repository. This initial version can then be *checked-out* as a *working copy*. The project can then be modified in the working copy, with the differences between the edited working copy and the original repository version examined using a *diff* procedure. Before checking the working copy into the repository, two steps should be performed. First, an *update* should be performed to integrate changes that others have made in the code and to identify *conflicts*. Next, a set of predefined tests should be run to ensure that the modifications do not unexpectedly change the code's behavior. Finally, the *working copy* of the project can be *checked-in* to the repository, generating a new *version* of the software project.

There is a wide array of version control systems available to the software developer. These systems range from free, open-source systems such as Concurrent Versions Systems

(CVS) and Subversion (SVN) (Collins-Sussman *et al.*, 2009) to commercially available systems. A short tutorial showing the basic steps for implementing version control with a Windows-based tool can be found at www.aoe.vt.edu/~cjroy/MISC/TortoiseSVN-Tutorial.pdf.

4.3 Software verification and validation

4.3.1 Definitions

The definitions accepted by AIAA (1998) and ASME (2006) for verification and validation as applied to scientific computing address the mathematical accuracy of a numerical solution (verification) and the physical accuracy of a given model (validation); however, the definitions used by the software engineering community (e.g., ISO, 1991; IEEE, 1991) are different. In software engineering, verification is defined as ensuring that software conforms to it specifications (i.e., requirements) and validation is defined as ensuring that software actually meets the customer's needs. Some argue that these definitions are really the same; however, upon closer examination, they are in fact different.

The key differences in these definitions for verification and validation are due to the fact that, in scientific computing, we begin with a governing partial differential or integral equation, which we will refer to as our mathematical model. For problems that we are interested in solving, there is generally no known exact solution to this model. It is for this reason that we must develop numerical approximations to the model (i.e., the numerical algorithm) and then implement that numerical algorithm within scientific computing software. Thus the two striking differences between how the scientific computing community and the software engineering community define verification and validation are as follows. First, in scientific computing, validation requires a comparison to experimental data. The software engineering community defines validation of the software as meeting the customer's needs, which is, in our opinion, too vague to tie it back to experimental observations. Second, in scientific computing, there is generally *no true system-level software test* (i.e., a test for correct code output given some code inputs) for real problems of interest. The "correct" output from the scientific software depends on the number of significant figures used in the computation, the computational mesh resolution and quality, the time step (for unsteady problems), and the level of iterative convergence. Chapters 5 and 6 of this book address the issue of system-level tests for scientific software.

In this section, we will distinguish between the two definitions of verification and validation by inserting the word "software" when referring to the definitions from software engineering. Three additional definitions that will be used throughout this section are those for software defects, faults, and failures (Hatton, 1997b). A *software defect* is a coding mistake (bug) or the misuse of a coding construct that could potentially lead to a software failure. A *software fault* is a defect which can be detected without running the code, i.e., through static analysis. Examples of defects that can lead to software faults include

dependence on uninitialized variables, mismatches in parameter arguments, and unassigned pointers. A *software failure* occurs when the software returns an incorrect result or when it terminates prematurely due to a run-time error (overflow, underflow, division by zero, etc.). Some examples of catastrophic software failures are given by Hatton (1997a).

4.3.2 Static analysis

Static analysis is any type of assessment of software correctness that does not require program execution. Examples of static analysis methods include software inspection, peer review, compiling of the code, and the use of automatic static analyzers. Hatton (1997a) estimates that approximately 40% of software failures are due to static faults. Some examples of static faults are:

- dependency on uninitialized or undeclared variables,
- interface faults: too few, too many, or wrong type of arguments passed to a function/subroutine,
- casting a pointer to a narrower integer type (C), and
- use of non-local variables in functions/subroutines (Fortran).

All of these static faults, as well as others that have their origins in ambiguities in the programming language standards, can be prevented by using static analysis.

4.3.2.1 Software inspection

Software inspection (or review) refers to the act of reading through the source code and other software products to find defects. Although software inspections are time intensive, they are surprisingly effective at finding software defects (Sommerville, 2004). Other advantages of software inspections are that they are not subject to interactions between different software defects (i.e., one defect will not hide the presence of another one), incomplete and nonfunctional source code can be inspected, and they can find other issues besides defects such as coding inefficiencies or lack of compliance with coding standards. The rigor of the software inspection depends on the technical qualifications of the reviewer as well as their level of independence from the software developers.

4.3.2.2 Compiling the code

Any time the code is compiled it goes through some level of static analysis. The level of rigor of the static analysis often depends on the options used during compilation, but there is a trade-off between the level of static analysis performed by the compiler and the execution speed. Many modern compilers provide different modes of compilation such as a release mode, a debug mode, and a check mode that perform increasing levels of static analysis. Due to differences in compilers and operating systems, many software developers make it standard practice to compile the source code with different compilers and on different platforms.

4.3.2.3 Automatic static analyzers

Automatic static analyzers are external tools that are meant to complement the checking of the code by the compiler. They are designed to find inconsistent or undefined use of a programming language that the compiler will likely overlook, as well as coding constructs that are generally considered as unsafe. Some static analyzers available for C/C^{++} include the Safer C Toolset, CodeWizard, CMT^{++}, Cleanscape LintPlus, PC-lint/FlexeLint, and QA C. Static analyzers for Fortran include floppy/fflow and ftnchek. There is also a recently-developed static analyzer for MATLAB called M-Lint (MATLAB, 2008). For a more complete list, or for references to each of these static analyzers, see www.testingfaqs.org/t-static.html.

4.3.3 Dynamic testing

Dynamic software testing can be defined as the "dynamic verification of the behavior of a program on a finite set of test cases . . . against the expected behavior" (SWEBOK, 2004). Dynamic testing includes any type of testing activity which involves running the code, thus run-time compiler checks (e.g., array bounds checking, pointer checking) fall under the heading of dynamic testing. The types of dynamic testing discussed in this section include defect testing (at the unit, component, and complete system level), regression testing, and software validation testing.

4.3.3.1 Defect testing

Defect testing is a type of dynamic testing performed to uncover the presence of a software defect; however, defect testing cannot be used to prove that no errors are present. Once a defect is discovered, the process of finding and fixing the defect is usually referred to as debugging. In scientific computing, it is convenient to decompose defect testing into three levels: unit testing which occurs at the smallest level in the code, component testing which occurs at the submodel or algorithm level, and system testing where the desired output from the software is evaluated. While unit testing is generally performed by the code developer, component and system-level testing is more reliable when performed by someone outside of the software development team.

Unit testing

Unit testing is used to verify the execution of a single routine (e.g., function, subroutine, object class) of the code (Eddins, 2006). Unit tests are designed to check for the correctness of routine output based on a given input. They should also be easy to write and run, and should execute quickly. Properly designed unit tests also provide examples of proper routine use such as how the routine should be called, what type of inputs should be provided, what type of outputs can be expected.

While it does take additional time to develop unit tests, this extra time in code development generally pays off later in reduced time debugging. The authors' experience with even

Table 4.1 *Example of a component-level test fixture for Sutherland's viscosity law (adapted from Kleb and Wood, 2006).*

Input: T (K)	Output: μ (kg/s-m)
$200 \leq T \leq 3000$	$B^* \frac{T^{1.5}}{T+110.4}$
199	error
200	1.3285589×10^{-5}
2000	6.1792781×10^{-5}
3000	7.7023485×10^{-5}
3001	error

* where $B = 1.458 \times 10^{-6}$

small scientific computing code development in university settings suggests that the typical ratio of debugging time to programming time for students who do not employ unit tests is at least five to one. The wider the unit testing coverage (i.e., percentage of routines that have unit tests), the more reliable the code is likely to be. In fact, some software development strategies such as Extreme Programming (XP) require tests to be written before the actual routine to be tested is created. Such strategies require the programmer to clearly define the interfaces (inputs and outputs) of the routine up front.

Component testing

Kleb and Wood (2006) make an appeal to the scientific computing community to implement the scientific method in the development of scientific software. Recall that, in the scientific method, a theory must be supported with a corresponding experiment that tests the theory, and must be described in enough detail that the experiment can be reproduced by independent sources. For application to scientific computing, they recommend testing at the component level, where a component is considered to be a submodel or algorithm. Furthermore, they strongly suggest that model and algorithm developers publish *test fixtures* with any newly proposed model or algorithm. These test fixtures are designed to clearly define the proper usage of the component, give examples of proper usage, and give sample inputs along with correct outputs that can be used for testing the implementation in a scientific computing code. An example of such a test fixture for Sutherland's viscosity law is presented in Table 4.1.

Component-level testing can be performed when the submodel or algorithm are algebraic since the expected (i.e., correct) solution can be computed directly. However, for cases where the submodel involves numerical approximations (e.g., many models for fluid turbulence involve differential equations), then the expected solution will necessarily be a function of the chosen discretization parameters, and the more sophisticated code verification methods

discussed in Chapters 5 and 6 should be used. For models that are difficult to test at the system level (e.g., the `min` and `max` functions significantly complicate the code verification process discussed in Chapter 5), component-level testing of the models (or different parts of the model) can be used. Finally, even when all components have been successfully tested individually, one should not get a false sense of security about how the software will behave at the system level. Complex interactions between components can only be tested at the system level.

System testing

System-level testing addresses code as a whole. For a given set of inputs to the code, what is the correct code output? In software engineering, system level testing is the primary means by which one determines if the software requirements have been met (i.e., software verification). For nonscientific software, it is often possible to *a priori* determine what the correct output of the code should be. However, for scientific computing software where partial differential or integral equations are solved, the "correct" output is generally not known ahead of time. Furthermore, the code output will depend on the grid and time step chosen, the iterative convergence level, the machine precision, etc. For scientific computing software, system-level testing is generally addressed through *order of accuracy verification*, which is the main subject of Chapter 5.

4.3.3.2 Regression testing

Regression testing involves the comparison of code or software routine output to the output from earlier versions of the code. Regression tests are designed to prevent the introduction of coding mistakes by detecting unintended consequences of changes in the code. Regression tests can be implemented at the unit, component, and system level. In fact, all of the defect tests described above can also be implemented as regression tests. The main difference between regression testing and defect testing is that regression tests do not compare code output to the correct expected value, but instead to the output from previous versions of the code. Careful regression testing combined with defect testing can minimize the chances of introducing new software defects during code development and maintenance.

4.3.3.3 Software validation testing

As discussed earlier, software validation is performed to ensure that the software actually meets the customer's needs in terms of software function, behavior, and performance (Pressman, 2005). Software validation (or acceptance) testing occurs at the system level and usually involves data supplied by the customer. Software validation testing for scientific computing software inherits all of the issues discussed earlier for system-level testing, and thus special considerations must be made when determining what the expected, correct output of the code should be.

4.3.4 Test harness and test suites

Many different types of dynamic software tests have been discussed in this section. For larger software development projects, it would be extremely tedious if the developer had to run each of the tests separately and then examine the results. Especially in the case of larger development efforts, automation of software testing is a must.

A *test harness* is the combination of software and test data used to test the correctness of a program or component by automatically running it under various conditions (Eddins, 2006). A test harness is usually composed of a test manager, test input data, test output data, a file comparator, and an automatic report generator. While it is certainly possible to create your own test harness, there are a variety of test harnesses that have been developed for a wide range of programming languages. For a detailed list, see en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

Once a suite of tests has been set up to run within a test harness, it is recommended that these tests be run automatically at specified intervals. Shorter tests can be run in a nightly test suite, while larger tests which require more computer time and memory may be set up in weekly or monthly test suites. In addition, an approach called *continuous integration testing* (Duvall *et al.*, 2007) requires that specified test suites be run before any new code modifications are checked in.

4.3.5 Code coverage

Regardless of how software testing is done, one important aspect is the coverage of the tests. *Code coverage* can be defined as the percentage of code components (and possibly their interactions) for which tests exist. While testing at the unit and component levels is relatively straightforward, system-level testing must also address interactions between different components. Large, complex scientific computing codes generally have a very large number of options for models, submodels, numerical algorithms, boundary conditions, etc. Assume for the moment that there are 100 different options in the code to be tested, a conservative estimate for most production-level scientific computing codes. Testing each option independently (although generally not possible) would require 100 different system-level tests. Testing pair-wise combinations for interactions between these different options would require 4950 system level tests. Testing the interactions between groups of three would require 161 700 tests. While this is clearly an upper bound since many options may be mutually exclusive, it does provide a sense of the magnitude of the task of achieving complete code coverage of model/algorithm interactions. Table 4.2 provides a comparison of the number of system-level tests required to ensure code coverage with different degrees of option interactions for codes with 10, 100, and 1000 different code options. Clearly, testing the three-way interactions for our example of 100 coding options is impossible, as would be testing all pair-wise interactions when 1000 coding options are available, a number not uncommon for commercial scientific computing codes. One possible approach for addressing this combinatorial explosion of tests for component interactions is

Table 4.2 *Number of system-level tests required for complete code coverage for codes with different numbers of options and option combinations to be tested.*

Number of options	Option combinations to be tested	System-level tests required
10	1	10
10	2	45
10	3	720
100	1	100
100	2	4950
100	3	161 700
1000	1	1000
1000	2	499 500
1000	3	$\sim 1.7 \times 10^8$

application-centric testing (Knupp and Ober, 2008), where only those components and component interactions which impact a specific code application are tested.

4.3.6 Formal methods

Formal methods use mathematically-based techniques for requirements specification, development, and/or verification testing of software systems. Formal methods arise from discrete mathematics and involve set theory, logic, and algebra (Sommerville, 2004). Such a rigorous mathematical framework is expensive to implement, thus it is mainly used for high-assurance (i.e., critical) software systems such as those found in aircraft controls systems, nuclear power plants, and medical devices such as pacemakers (Heitmeyer, 2004). Some of the drawbacks to using formal methods are that they do not handle user interfaces well and they do not scale well for larger software projects. Due to the effort and expense required, as well as their poor scalability, we do not recommend formal methods for scientific computing software.

4.4 Software quality and reliability

There are many different definitions of quality applied to software. The definition that we will use is: *conformance to customer requirements and needs*. This definition implies not only adherence to the formally documented requirements for the software, but also those requirements that are not explicitly stated by the customer that need to be met. However, this definition of quality can often only be applied after the complete software product is delivered to the customer. Another aspect of software quality that we will find useful is

software reliability. One definition of *software reliability* is the probability of failure-free operation of software in a given environment for a specified time (Musa, 1999). In this section we present some explicit and implicit methods for measuring software reliability. A discussion of recommended programming practices as well as error-prone coding constructs that should be avoided when possible, both of which can affect software reliability, can be found in the Appendix.

4.4.1 Reliability metrics

Two quantitative approaches for measuring code quality are *defect density analysis*, which provides an explicit measure of reliability, and *complexity analysis*, which provides an implicit measure of reliability. Additional information on software reliability can be found in Beizer (1990), Fenton and Pfleeger (1997), and Kaner *et al.* (1999).

4.4.1.1 Defect density analysis

The most direct method for assessing the reliability of software is in terms of the number of defects in the software. Defects can lead to static errors (faults) and dynamic errors (failures). The *defect density* is usually reported as the number of defects per executable source lines of code (SLOC). Hatton (1997a) argues that it is only by measuring the defect density of software, through both static analysis and dynamic testing, that an objective assessment of software reliability can be made. Hatton's T Experiments (Hatton, 1997b) are discussed in detail in the next section and represent the largest known defect density study of scientific software.

A significant limitation of defect density analysis is that the defect rate is a function of both the number of defects in the software and the specific testing procedure used to find the defects (Fenton and Pfleeger, 1997). For example, a poor testing procedure might uncover only a few defects, whereas a more comprehensive testing procedure applied to the same software might uncover significantly more defects. This sensitivity to the specific approach used for defect testing represents a major limitation of defect density analysis.

4.4.1.2 Complexity analysis

Complexity analysis is an indirect way of measuring reliability because it requires a model to convert internal code quality attributes into code reliability (Sommerville, 2004). The most frequently used model is to assume that a high degree of complexity in a component (function, subroutine, object class, etc.) is bad while a low degree of complexity is good. In this case, components which are identified as being too complex can be decomposed into smaller components. However, Hatton (1997a) used defect density analysis to show that the defect density in components follows a U-shaped curve, with the minimum occurring at 150–250 lines of source code per component, independent of both programming language and application area. He surmised that the increase in defect density for smaller components may be related to the inadvertent adverse effects of component reuse (see Hatton (1996)

for more details). Some different internal code attributes that can be used to indirectly assess code reliability are discussed in this subsection, and in some cases, tools exist for automatically evaluating these complexity metrics.

Lines of source code

The simplest measure of complexity can be found by counting the number of executable source lines of code (SLOC) for each component. Hatton (1997a) recommends keeping components between 150 and 250 SLOC.

NPATH metric

The NPATH metric simply counts the number of possible execution paths through a component (Nejmeh, 1988). Nejmeh (1988) recommends keeping this value below 200 for each component.

Cyclomatic complexity

The cyclomatic, or McCabe, complexity (McCabe, 1976) is defined as one plus the number of decision points in a component, where a decision point is defined as any loop or logical statement (`if`, `elseif`, `while`, `repeat`, `do`, `for`, `or`, etc.). The maximum recommended value for cyclomatic complexity of a component is ten (Eddins, 2006).

Depth of conditional nesting

This complexity metric provides a measure of the depth of nesting of if-statements, where larger degrees of nesting are assumed to be more difficult to understand and track, and therefore are more error prone (Sommerville, 2004).

Depth of inheritance tree

Applicable to object-oriented programming languages, this complexity metric measures the number of levels in the inheritance tree where sub-classes inherit attributes from super-classes (Sommerville, 2004). The more levels that exist in the inheritance tree, the more classes one needs to understand to be able to develop or modify a given object class.

4.5 Case study in reliability: the T experiments

In the early 1990s, Les Hatton undertook a broad study of scientific software reliability known collectively as the "T Experiments" (Hatton, 1997b). This study was broken into two parts: the first (T1) examined codes from a wide range of scientific disciplines using static analysis, while the second (T2) examined codes in a single discipline using dynamic testing.

The T1 study used static deep-flow analyzers to examine more than 100 different codes in 40 different application areas. All codes were written in C, FORTRAN 66, or FORTRAN 77, and the static analyzers used were QA C (for the C codes) and QA Fortran (for the

FORTRAN codes). The main conclusion of the T1 study was that the C codes contained approximately eight serious static faults per 1000 lines of executable code, while the FORTRAN codes contained approximately 12 faults per 1000 lines. A serious static fault is defined as a statically-detectable defect that is likely to cause the software to fail. For more details on the T1 study, see Hatton (1995).

The T2 study examined a subset of the codes from the T1 study in the area of seismic data processing which is used in the field of oil and gas exploration. This study examined nine independent, mature, commercial codes which employed the same algorithms, the same programming language (FORTRAN), the same user-defined parameters, and the same input data. Hatton refers to such a study as N-version programming since each code was developed independently by a different company. Each of the codes consisted of approximately 30 sequential steps, 14 of which used unambiguously defined algorithms, referred to in the study as primary calibration points. Agreement between the codes after the first primary calibration point was within 0.001% (i.e., approximately machine precision for single-precision computations); however, agreement after primary calibration point 14 was only within a factor of two. It is interesting to note that distribution of results from the various codes was found to be non-Gaussian with distinct groups and outliers, suggesting that the output from an N-version programming test should not be analyzed with Bayesian statistics. Hatton concluded that the disagreements between the different codes are due primarily to software errors. Such dismal results from the T2 study prompted Hatton to conclude that *"the results of scientific calculations carried out by many software packages should be treated with the same measure of disbelief researchers have traditionally attached to the results of unconfirmed physical experiments."* For more details on the T2 study, see Hatton and Roberts (1994).

These alarming results from Hatton's "T Experiments" highlight the need for employing good software engineering practices in scientific computing. At a minimum, the simple techniques presented in this chapter such as version control, static analysis, dynamic testing, and reliability metrics should be employed for all scientific computing software projects to improve quality and reliability.

4.6 Software engineering for large software projects

Up to this point, the software engineering practices discussed have been applicable to all scientific computing project whether large or small. In this section, we specifically address software engineering practices for large scientific computing projects that may be less effective for smaller projects. The two broad topics addressed here include software requirements and software management.

4.6.1 Software requirements

A software requirement is a "property that must be exhibited in order to solve some real-world problem" (SWEBOK, 2004). Uncertainty in requirements is a leading cause of

failure in software projects (Post and Kendall, 2004). While it is certainly ideal to have all requirements rigorously and unambiguously specified at the beginning of a software project, this can be difficult to achieve for scientific software. Especially in the case of large scientific software development projects, complete requirements can be difficult to specify due to rapid changes in models, algorithms, and even in the specialized computer architectures used to run the software. While lack of requirements definition can adversely affect the development of scientific software, these negative effects can be mitigated somewhat if close communication is maintained between the developer of the software and the user (Post and Kendall, 2004) or if the developer is also an expert in the scientific computing discipline.

4.6.1.1 Types of software requirements

There are two main types of software requirements. User requirements are formulated at a high level of abstraction, usually in general terms which are easily understood by the user. An example of a user requirement might be: *this software should produce approximate numerical solutions to the Navier–Stokes equations*. Software system requirements, on the other hand, are a precise and formal definition of a software system's functions and constraints. The software system requirements are further decomposed as follows:

1 functional requirements – rigorous specifications of required outputs for a given set of inputs,
2 nonfunctional requirements – additional nonfunctional constraints such as programming standards, reliability, and computational speed, and
3 domain requirements – those requirements that come from the application domain such as a discussion of the partial differential or integral equations to be solved numerically for a given scientific computing application.

The domain requirements are crucial in scientific computing since these will be used to define the specific governing equations, models, and numerical algorithms to be implemented. Finally, if the software is to be integrated with existing software, then additional specifications may be needed for the procedure interfaces (application programming interfaces, or APIs), data structures, or data representation (e.g., bit ordering) (Sommerville, 2004).

4.6.1.2 Requirements engineering process

The process for determining software requirements contains four phases: elicitation, analysis, specification, and validation. *Elicitation* involves the identification of the sources for requirements, which includes the code customers, users, and developers. For larger software projects these sources could also include managers, regulatory authorities, third-party software providers, and other stakeholders. Once the sources for the requirements have been identified, the requirements are then collected either individually from those sources or by bringing the sources together for discussion.

In the *analysis* phase, the requirements are analyzed for clarity, conflicts, and the need for possible requirements negotiation between the software users and developers. In scientific

computing, while the users typically want a code with a very broad range of capabilities, the developers must weigh trade-offs between capability and the required computational infrastructure, all while operating under manpower and budgetary constraints. Thus negotiation and compromise between the users and the developers is critical for developing computational tools that balance capability with feasibility and available resources.

Specification deals with the documentation of the established user and system requirements in a formal software requirements document. This requirements document should be considered a living document since requirements often change during the software's life cycle. Requirements *validation* is the final confirmation that the software meets the customer's needs, and typically comes in the form of full software system tests using data supplied by the customer. One challenge specific to scientific computing software is the difficulty in determining the correct code output due to the presence of numerical approximation errors.

4.6.1.3 Requirements management

Requirements management is the process of understanding, controlling, and tracking changes to the system requirements. It is important because software requirements are usually incomplete and tend to undergo frequent changes. Things that can cause the requirements to change include installing the software on a new hardware system, identification of new desired functionality based on user experience with the software, and, for scientific computing, improvements in existing models or numerical algorithms.

4.6.2 Software management

Software management is a broad topic which includes the management of the software project, cost, configuration, and quality. In addition, effective software management strategies must include approaches for improvement of the software development process itself.

4.6.2.1 Project management

Software project management addresses the planning, scheduling, oversight, and risk management of a software project. For larger projects, planning activities encompass a wide range of different areas, and separate planning documents should be developed for quality, software verification and validation, configuration management, maintenance, staff development, milestones, and deliverables. Another important aspect of software project management is determining the level of formality required in applying the software engineering practices. Ultimately, this decision should be made by performing a risk-based assessment of the intended use, mission, complexity, budget, and schedule (Demarco and Lister, 2003).

Managing software projects is generally more difficult than managing standard engineering projects because the product is intangible, there are usually no standard software management practices, and large software projects are usually one-of-a-kind endeavors

(Sommerville, 2004). According to Post and Kendall (2004), ensuring consistency between the software schedule, resources, and requirements is the key to successfully managing a large scientific computing software project.

4.6.2.2 Cost estimation

While estimating the required resources for a software project can be challenging, semi-empirical models are available. These models are called algorithmic cost models, and in their simplest form (Sommerville, 2004) can be expressed as:

$$Effort = A \times (Size)^b \times M. \tag{4.1}$$

In this simple algorithmic cost model, A is a constant which depends on the type of organization developing the software, their software development practices, and the specific type of software being developed. *Size* is some measure of the size of the software project (estimated lines of code, software functionality, etc.). The exponent b typically varies between 1 and 1.5, with larger values indicative of the fact that software complexity increases nonlinearly with the size of the project. M is a multiplier that accounts for various factors including risks associated with software failures, experience of the code development team, and the dependability of the requirements. *Effort* is generally in man-months, and the cost is usually assumed to be proportional to the effort. Most of these parameters are subjective and difficult to evaluate, thus they should be determined empirically using historical data for the organization developing the software whenever possible. When such data are not available, historical data from similar organizations may be used.

For larger software projects where more accurate cost estimates are required, Sommerville (2004) recommends the more detailed Constructive Cost Model (COCOMO). When software is developed using imperative programming languages such as Fortran or C using a waterfall model, the original COCOMO model, now referred to as COCOMO 81, can be used (Boehm, 1981). This algorithmic cost model was developed by Boehm while he was at the aerospace firm TRW Inc., and drew upon the historical software development data from 63 different software projects ranging from 2000 to 10000 lines of code. An updated model, COCOMO II, has been developed which accounts for object-oriented programming languages, software reuse, off-the-shelf software components, and a spiral software development model (Boehm *et al.*, 2000).

4.6.2.3 Configuration management

Configuration management deals with the control and management of the software products during all phases of the software product's lifecycle including planning, development, production, maintenance, and retirement. Here software products include not only the source code, but also user and theory manuals, software tests, test results, design documents, web pages, and any other items produced during the software development process. Configuration management tracks the way software is configured over time and is used for controlling changes to, and for maintaining integrity and traceability of, the software products

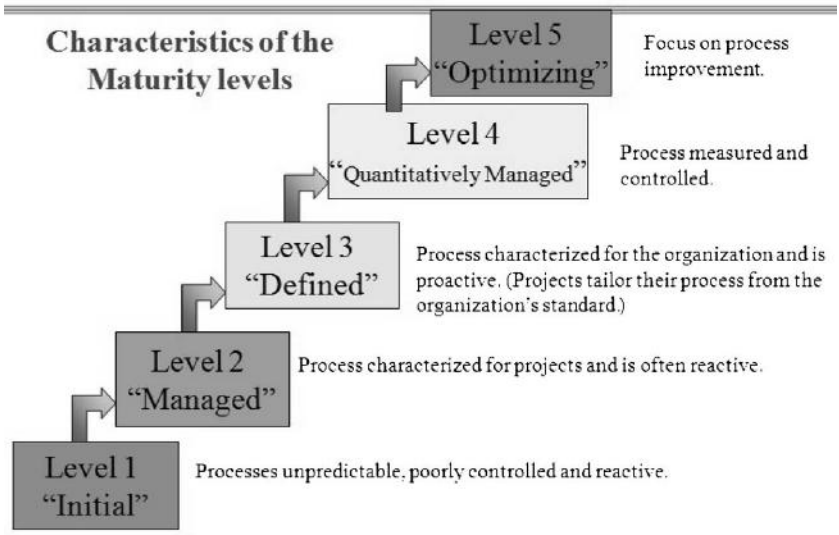

Figure 4.2 Characteristics of the maturity levels in CMMI (from Godfrey, 2009).

(Sommerville, 2004). The key aspects of configuration management include using *version control* (discussed in Section 4.2) for source code and other important software products, identification of the software products to be managed, recording, approving, and tracking issues with the software, managing software releases, and ensuring frequent backups are made.

4.6.2.4 *Quality management*

Software quality management is usually separated into three parts: quality assurance, quality planning, and quality control (Sommerville, 2004). *Quality assurance* is the definition of a set of procedures and standards for developing high-quality software. *Quality planning* is the process of selecting from the above procedures and standards for a given software project. *Quality control* is a set of processes that ensure the quality plan was actually implemented. It is important to maintain independence between the quality management team and the code development team (Sommerville, 2004).

4.6.2.5 *Process improvement*

Another way to improve the quality of software is to improve the processes which are used to develop it. Perhaps the most well-known software process improvement model is the Capability Maturity Model, or CMM (Humphrey, 1989). The successor to CMM, the Capability Maturity Model Integration (CMMI) integrates various process improvement models and is more broadly applicable to the related areas of systems engineering and integrated product development (SEI, 2009). The five maturity levels in CMMI are shown in Figure 4.2, and empirical evidence suggests that both software quality and developer

productivity will improve as higher levels of process maturity are reached (Gibson *et al.*, 2006).

Post and Kendall (2004) found that not all software engineering practices are helpful for developing scientific software. They cautioned against blindly applying rigorous software standards such as CMM/CMMI without first performing a cost-benefit analysis. Neely (2004) suggests a risk-based approach to applying quality assurance practices to scientific computing projects. High-risk projects are defined as those that could potentially involve "great loss of money, reputation, or human life," while a low risk project would involve at most inconvenience to the user. High-risk projects would be expected to conform to more formal software quality standards, whereas low-risk projects would allow more informal, ad-hoc implementation of the standards.

4.7 References

AIAA (1998). *Guide for the Verification and Validation of Computational Fluid Dynamics Simulations*. AIAA-G-077–1998, Reston, VA, American Institute of Aeronautics and Astronautics.

Allen, E. B. (2009). Private communication, February 11, 2009.

ASME (2006). *Guide for Verification and Validation in Computational Solid Mechanics*. ASME V&V 10–2006, New York, NY, American Society of Mechanical Engineers.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*, Reading, PA, Addison-Wesley.

Beizer, B. (1990). *Software Testing Techniques*, 2nd edn., New York, Van Nostrand Reinhold.

Boehm, B. W. (1981). *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall.

Boehm, B. W., C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece (2000). *Software Cost Estimation with Cocomo II*, Englewood Cliffs, NJ, Prentice-Hall.

Collins-Sussman, B., B. W. Fitzpatrick, and C. M. Pilato (2009). *Version Control with Subversion: For Subversion 1.5: (Compiled from r3305)* (see svnbook.red-bean.com/en/1.5/svn-book.pdf).

Demarco, T. and T. Lister (2003). *Waltzing with Bears: Managing Risk on Software Projects*, New York, Dorset House.

Duvall, P. F., S. M. Matyas, and A. Glover (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*, Upper Saddle River, NJ, Harlow: Addison-Wesley.

Eddins, S. (2006). Taking control of your code: essential software development tools for engineers, *International Conference on Image Processing*, Atlanta, GA, Oct. 9 (see blogs.mathworks.com/images/steve/92/handout_final_icip2006.pdf).

Fenton, N. E. and S. L. Pfleeger (1997). *Software Metrics: a Rigorous and Practical Approach*, 2nd edn., London, PWS Publishing.

Gibson, D. L., D. R. Goldenson, and K. Kost (2006). *Performance Results of CMMI®-Based Process Improvement*, Technical Report CMU/SEI-2006-TR-004, ESC-TR-2006–004, August 2006 (see www.sei.cmu.edu/publications/documents/06.reports/06tr004.html).

Godfrey, S. (2009). *What is CMMI?* NASA Presentation (see software.gsfc.nasa.gov/docs/What%20is%20CMMI.ppt).

Hatton, L. (1995). *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*, New York, McGraw-Hill International Ltd.

Hatton, L. (1996). Software faults: the avoidable and the unavoidable: lessons from real systems, *Proceedings of the Product Assurance Workshop, ESA SP-377*, Noordwijk, The Netherlands.

Hatton, L. (1997a). Software failures: follies and fallacies, *IEEE Review*, March, 49–52.

Hatton, L. (1997b). The T Experiments: errors in scientific software, *IEEE Computational Science and Engineering*, **4**(2), 27–38.

Hatton, L., and A. Roberts (1994). How accurate is scientific software? *IEEE Transactions on Software Engineering*, **20**(10), 785–797.

Heitmeyer, C. (2004). Managing complexity in software development with formally based tools, *Electronic Notes in Theoretical Computer Science*, **108**, 11–19.

Humphrey, W. (1989). *Managing the Software Process*. Reading, MA, Addison-Wesley Professional.

IEEE (1991). *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12–1990, New York, IEEE.

ISO (1991). *ISO 9000–3: Quality Management and Quality Assurance Standards – Part 3: Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software*. Geneva, Switzerland, International Organization for Standardization.

Kaner, C., J. Falk, and H. Q. Nguyen (1999). *Testing Computer Software*, 2nd edn., New York, Wiley.

Kleb, B., and B. Wood (2006). Computational simulations and the scientific method, *Journal of Aerospace Computing, Information, and Communication*, **3**(6), 244–250.

Knupp, P. M. and C. C. Ober (2008). *A Code-Verification Evidence-Generation Process Model and Checklist*, Sandia National Laboratories Report SAND2008–4832.

Knupp, P. M., C. C., Ober, and R. B. Bond (2007). *Impact of Coding Mistakes on Numerical Error and Uncertainty in Solutions to PDEs*, Sandia National Laboratories Report SAND2007–5341.

MATLAB (2008). *MATLAB® Desktop Tools and Development Environment*, Natick, MA, The Mathworks, Inc. (see www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matlab_env.pdf).

McCabe, T. J. (1976). A complexity measure, *IEEE Transactions on Software Engineering*, **2**(4), 308–320.

McConnell, S. (2004). *Code Complete: a Practical Handbook of Software Construction*, 2nd edn., Redmond, WA, Microsoft Press.

Musa, J. D. (1999). *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, New York, McGraw-Hill.

Neely, R. (2004). Practical software quality engineering on a large multi-disciplinary HPC development team, *Proceedings of the First International Workshop on Software Engineering for High Performance Computing System Applications*, Edinburgh, Scotland, May 24, 2004.

Nejmeh, B. A. (1988). Npath: a measure of execution path complexity and its applications, *Communications of the Association for Computing Machinery*, **31**(2), 188–200.

Post, D. E., and R. P. Kendall (2004). Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel

computational simulations: lessons learned from ASCI, *International Journal of High Performance Computing Applications*, **18**(4), 399–416.
Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (2007). *Numerical Recipes: the Art of Scientific Computing*, 3rd edn., Cambridge, Cambridge University Press.
Pressman, R. S. (2005). *Software Engineering: a Practitioner's Approach*, 6th edn., Boston, MA, McGraw-Hill.
Roy, C. J. (2009). Practical software engineering strategies for scientific computing, AIAA Paper 2009–3997, *19th AIAA Computational Fluid Dynamics, San Antonio, TX*, June 22–25, 2009.
SE-CSE (2008). *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering*, Leipzig, Germany, May 13, 2008 (see cs.ua.edu/∼SECSE08/).
SE-CSE (2009). *Proceedings of the Second International Workshop on Software Engineering for Computational Science and Engineering*, Vancouver, Canada, May 23, 2009 (see cs.ua.edu/∼SECSE09/).
SE-HPC (2004). *Proceedings of the First International Workshop On Software Engineering for High Performance Computing System Applications*, Edinburgh, Scotland, May 24, 2004.
SEI (2009). CMMI Main Page, Software Engineering Institute, Carnegie Mellon University (see www.sei.cmu.edu/cmmi/index.html).
Sommerville, I. (2004). *Software Engineering*, 7th edn., Harlow, Essex, England, Pearson Education Ltd.
SWEBOK (2004), *Guide to the Software Engineering Body of Knowledge: 2004 Edition*, P. Borque and R. Dupuis (eds.), Los Alamitos, CA, IEEE Computer Society (www.swebok.org).
Williams, L. and R. Kessler (2003). *Pair Programming Illuminated*, Boston, MA, Addison-Wesley.
Wilson, G. (2009). *Software Carpentry*, www.swc.scipy.org/.
Wood, W. A. and W. L. Kleb (2003). Exploring XP for scientific research, *IEEE Software*, **20**(3), 30–36.

5

Code verification

In scientific computing, the goal of code verification is to ensure that the code is a faithful representation of the underlying mathematical model. This mathematical model generally takes the form of partial differential or integral equations along with associated initial condition, boundary conditions, and auxiliary relationships. Code verification thus addresses both the correctness of the chosen numerical algorithm and the correctness of the instantiation of that algorithm into written source code, i.e., ensuring there are no coding mistakes or "bugs."

A computer program, referred to here simply as a code, is a collection of instructions for a computer written in a programming language. As discussed in Chapter 4, in the software engineering community code verification is called software verification and is comprised of software tests which ensure that the software meets the stated requirements. When conducting system-level testing of non-scientific software, in many cases it is possible to exactly determine the correct code output for a set of given code inputs. However, in scientific computing, the code output depends on the numerical algorithm, the spatial mesh, the time step, the iterative tolerance, and the number of digits of precision used in the computations. Due to these factors, it is not possible to know the correct code output (i.e., numerical solution) *a priori*. The developer of scientific computing software is thus faced with the difficult challenge of determining appropriate system-level software tests.

This chapter discusses various procedures for verifying scientific computing codes. Although a formal proof of the "correctness" of a complex scientific computing code is probably not possible (Roache, 1998), code testing using the order verification procedures discussed in this chapter can provide a high degree of confidence that the code will produce the correct solution. An integral part of these procedures is the use of systematic mesh and time step refinement. For rigorous code verification, an exact solution to the underlying governing equations (i.e., the mathematical model) is required. We defer the difficult issue of how to obtain these exact solutions until Chapter 6, and for now simply assume that an exact solution to the mathematical model is available. Finally, unless otherwise noted, the code verification procedures discussed in this chapter do not depend on the discretization approach, thus we will defer our discussion of the different discretization methods (finite difference, finite volume, finite element, etc.) until Chapter 8.

5.1 Code verification criteria

Before choosing a criterion for code verification, one must first select the code outputs to be tested. The first code outputs to be considered are the dependent variables in the mathematical model. For all but the simplest code verification criteria, we will compare the solution to a reference solution, ideally an exact solution to the mathematical model. In this case, we can convert the difference between the code output and the reference solution over the entire domain into a single, scalar error measure using a norm.

If a continuous representation of the numerical solution is available (e.g., from the finite element method), then a continuous norm can be used. For example, the L_1 norm of the solution error over the domain is given by

$$\|u - u_{\text{ref}}\|_1 = \frac{1}{\Omega} \int_{\Omega} |u - u_{\text{ref}}| \, \mathrm{d}\omega, \tag{5.1}$$

where u is the numerical solution, u_{ref} the reference solution, and Ω is the domain of interest. The L_1 norm is the most appropriate norm to use when discontinuities or singularities exist in the solution (Rider, 2009). If instead a discrete representation of the numerical solution is available (e.g., from a finite difference or finite volume method), then a discrete norm of the error can be used. The discrete L_1 norm provides a measure of the average absolute error over the domain and can be defined as

$$\|u - u_{\text{ref}}\|_1 = \frac{1}{\Omega} \sum_{n=1}^{N} \omega_n \left| u_n - u_{\text{ref},n} \right|, \tag{5.2}$$

where the subscript n refers to a summation over all N cells of size ω_n in both space and time. Note that for uniform meshes (i.e., those with constant cell spacing in all directions), the cell sizes cancel resulting in simply:

$$\|u - u_{\text{ref}}\|_1 = \frac{1}{N} \sum_{n=1}^{N} \left| u_n - u_{\text{ref},n} \right|. \tag{5.3}$$

Another commonly used norm for evaluating the discretization error is the L_2 (or Euclidean) norm, which effectively provides the root mean square of the error. For a uniform mesh, the discrete L_2 norm is given by

$$\|u - u_{\text{ref}}\|_2 = \left(\frac{1}{N} \sum_{n=1}^{N} \left| u_n - u_{\text{ref},n} \right|^2 \right)^{1/2}. \tag{5.4}$$

The max (or infinity) norm returns the maximum absolute error over the entire domain, and is generally the most sensitive measure of error:

$$\|u - u_{\text{ref}}\|_\infty = \max \left| u_n - u_{\text{ref},n} \right|, \quad n = 1 \text{ to } N. \tag{5.5}$$

In addition to the dependent variables, one should also examine any system response quantities that may be of interest to the code user. These quantities can take the form of derivatives (e.g., local heat flux, local material stress), integrals (e.g., drag on an object,

net heat flux through a surface), or other functionals of the solution variables (e.g., natural frequency, maximum deflection, maximum temperature). All system response quantities that may potentially be of interest to the user should be included as part of the code verification process to ensure that both the dependent variables and the procedures for obtaining the system response quantities are verified. For example, a numerical solution for the dependent variables may be verified, but if the subsequent numerical integration used for a system response quantity contains a mistake, then incorrect values of that quantity will be produced.

There are a number of different criteria that can be used for verifying a scientific computing code. In order of increasing rigor, these criteria are:

1 simple tests,
2 code-to-code comparisons,
3 discretization error quantification,
4 convergence tests, and
5 order-of-accuracy tests.

The first two, simple tests and code-to-code comparisons, are the least rigorous but have the advantage that they can be performed for cases where an exact solution to the mathematical model is not available. The remaining criteria require that an exact solution to the mathematical model be available, or at the very least a demonstrably accurate surrogate solution. These five criteria are discussed in more detail below.

5.1.1 Simple tests

The following simple tests, while not a replacement for rigorous code verification studies, can be used as part of the code verification process. They have the advantage that they can be applied even when an exact solution to the mathematical model is not available since they are applied directly to the numerical solution.

5.1.1.1 Symmetry tests

In most cases, when a code is provided with a symmetric geometry, initial conditions, and boundary conditions, it will produce a symmetric solution. In some cases, physical instabilities can lead to solutions that are asymmetric at any given point in time, but may still be symmetric in a statistical sense. One example is the laminar, viscous flow past a circular cylinder, which will be symmetric for Reynolds numbers below 40, but will generate a von Karman vortex street at higher Reynolds numbers (Panton, 2005). Note that this test should not be used near a bifurcation point (i.e., near a set of conditions where the solution can rapidly change its basic character).

5.1.1.2 Conservation tests

In many scientific computing applications, the mathematical model will be based on the conservation of certain properties such as mass, momentum, and energy. In a discrete

sense, different numerical approaches will handle conservation differently. In the finite-difference method, conservation is only assured in the limit as the mesh and/or time step are refined. In the finite element method, conservation is strictly enforced over the global domain boundaries, but locally only in the limiting sense. For finite volume discretizations, conservation is explicitly enforced at each cell face, and thus this approach should satisfy the conservation requirement even on very coarse meshes. An example conservation test for steady-state heat conduction is to ensure that the net energy flux into the domain minus the net energy flux out of the domain equals zero, either within round-off error (finite element and finite volume methods) or in the limit as the mesh is refined (finite difference method). See Chapter 8 for a more detailed discussion of the differences between these discretization approaches.

5.1.1.3 Galilean invariance tests

Most scientific computing disciplines have their foundations in Newtonian (or classical) mechanics. As such, solutions to both the mathematical model and the discrete equations should obey the principle of Galilean invariance, which states that the laws of physics are valid for all inertial reference frames. Inertial reference frames are allowed to undergo linear translation, but not acceleration or rotation. Two common Galilean invariance tests are to allow the coordinate system to move at a fixed linear velocity or to simply exchange the direction of the coordinate axes (e.g., instead of having a 2-D cantilevered beam extend in the x-direction and deflect in the y-direction, have it extend in the y-direction and deflect in the x-direction). In addition, for structured grid codes that employ a global transformation from physical space (x, y, z) to computational space (ξ, η, ζ), certain mistakes in the global mesh transformations can be found by simply re-running a problem with the computational coordinates reoriented in different directions; again this procedure should have no effect on the final numerical solution.

5.1.2 Code-to-code comparisons

Code-to-code comparisons are among the most common approaches used to assess code correctness. A *code-to-code comparison* occurs when the output (numerical solution or system response quantity) from one code is compared to the output from another code. Following Trucano *et al*. (2003), code-to-code comparisons are only useful when (1) the two codes employ the same mathematical models and (2) the "reference" code has undergone rigorous code verification assessment or some other acceptable type of code verification. Even when these two conditions are met, code-to-code comparisons should be used with caution.

If the same models are not used in the two codes, then differences in the code output could be due to model differences and not coding mistakes. Likewise, agreement between the two codes could occur due to the serendipitous cancellation of errors due to coding mistakes and differences due to the model. A common mistake made while performing code-to-code

comparisons with codes that employ different numerical schemes (i.e., discrete equations) is to assume that the codes should produce the same (or very similar) output for the same problem with the same spatial mesh and/or time step. On the contrary, the code outputs will only be the same if exactly the same algorithm is employed, and even subtle algorithm differences can produce different outputs for the same mesh and time step.

For the case where the reference code has not itself been verified, agreement between the two codes *does not* imply correctness for either code. The fortuitous agreement (i.e., a false positive for the test) can occur due to the same algorithm deficiency being present in both codes. Even when the above two requirements have been met, "code comparisons do not provide substantive evidence that software is functioning correctly" (Trucano *et al*., 2003). Thus code-to-code comparisons should not be used as a substitute for rigorous code verification assessments. See Trucano *et al*. (2003) for a more detailed discussion of the proper usage of code-to-code comparisons.

5.1.3 Discretization error evaluation

Discretization error evaluation is the traditional method for code verification that can be used when an exact solution to the mathematical model is available. This test involves the quantitative assessment of the error between the numerical solution (i.e., the code output) and an exact solution to the mathematical model using a single mesh and/or time step. The main drawback to this test is that once the discretization error has been evaluated, it requires a subjective judgment of whether or not the error is sufficiently small. See Chapter 6 for an extensive discussion of methods for obtaining exact solutions to the mathematical model.

5.1.4 Convergence tests

A *convergence test* is performed to assess whether the error in the discrete solution relative to the exact solution to the mathematical model (i.e., the discretization error) reduces as the mesh and time step are refined. (The formal definition of convergence will be given in Section 5.2.3.) As was the case for discretization error evaluation, the convergence test also requires an exact solution to the mathematical model. However, in this case, it is not just the magnitude of the discretization error that is assessed, but whether or not that error reduces with increasing mesh and time step refinement. The convergence test is the minimum criterion that should be used for rigorous code verification.

5.1.5 Order-of-accuracy tests

The most rigorous code verification criterion is the *order-of-accuracy test*, which examines not only the convergence of the numerical solution, but also whether or not the discretization error is reduced at the theoretical rate as the mesh and/or time step are refined. This theoretical rate is called the *formal order of accuracy* and it is usually found by performing

a truncation error analysis of the numerical scheme (see Section 5.3.1). The actual rate at which the discretization error is reduced is called the *observed order of accuracy*, and its calculation requires two systematically refined meshes and/or time steps when the exact solution to the mathematical model is available. The procedures for computing the observed order of accuracy in such cases are presented in Section 5.3.2.

The order-of-accuracy test is the most difficult test to satisfy; therefore it is the most rigorous of the code verification criteria. It is extremely sensitive to even small mistakes in the code and deficiencies in the numerical algorithm. The order-of-accuracy test is the most reliable code verification criterion for finding coding mistakes and algorithm deficiencies which affect the order of accuracy of the computed solutions. Such order-of-accuracy problems can arise from many common coding mistakes including implementation of boundary conditions, transformations, operator splitting, etc. For these reasons, the order-of-accuracy test is the recommended criterion for code verification.

5.2 Definitions

The definitions presented in this section follow the standard definitions from the numerical analysis of partial differential and integral equations (e.g., Richtmyer and Morton, 1967). A firm grasp of these definitions is needed before moving on to the concepts behind the formal and observed order of accuracy used in the order-verification procedures.

5.2.1 Truncation error

The *truncation error* is the difference between the discretized equations and the original partial differential (or integral) equations. It is *not* the difference between a real number and its finite representation for storage in computer memory; this "digit truncation" is called *round-off error* and is discussed in Chapter 7. Truncation error necessarily occurs whenever a mathematical model is approximated by a discretization method. The form of the truncation error can usually be found by performing Taylor series expansions of the dependent variables and then inserting these expansions into the discrete equations. Recall the general Taylor series representation for the smooth function $u(x)$ expanded about the point x_0:

$$u(x) = u(x_0) + \left.\frac{\partial u}{\partial x}\right|_{x_0} (x - x_0) + \left.\frac{\partial^2 u}{\partial x^2}\right|_{x_0} \frac{(x - x_0)^2}{2} + \left.\frac{\partial^3 u}{\partial x^3}\right|_{x_0} \frac{(x - x_0)^3}{6} + O\left[(x - x_0)^4\right],$$

where the $O[(x - x_0)^4]$ term denotes that the leading term that is omitted is on the order of $(x - x_0)$ to the fourth power. This expansion can be represented more compactly as:

$$u(x) = \sum_{k=0}^{\infty} \left.\frac{\partial^k u}{\partial x^k}\right|_{x_0} \frac{(x - x_0)^k}{k!}.$$

5.2.1.1 Example: truncation error analysis

For a simple example of truncation error analysis, consider the following mathematical model for the 1-D unsteady heat equation:

$$\frac{\partial T}{\partial t} - \alpha \frac{\partial^2 T}{\partial x^2} = 0, \tag{5.6}$$

where the first term is the unsteady contribution and the second term represents thermal diffusion with a constant diffusivity α. Let $L(T)$ represent this partial differential operator and let $\tilde{T}$ be the exact solution to this mathematical model assuming appropriate initial and boundary conditions. Thus we have

$$L(\tilde{T}) = 0. \tag{5.7}$$

For completeness, this mathematical model operator $L(\cdot)$ should be formulated as a vector containing the partial differential equation along with appropriate initial and boundary conditions. For simplicity, we will omit the initial and boundary conditions from the following discussion.

Equation (5.6) can be discretized with a finite difference method using a forward difference in time and a centered second difference in space, resulting in the simple explicit numerical scheme

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} - \alpha \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta x)^2} = 0, \tag{5.8}$$

where the i subscripts denote spatial location, the n superscripts denote the temporal step, Δx is the constant spatial distance between nodes, and Δt is the time step. We can represent this discrete equation compactly using the discrete operator $L_h(T)$ which is solved exactly by the numerical solution T_h, i.e., we have

$$L_h(T_h) = 0. \tag{5.9}$$

The variable h is a single parameter that is used to denote systematic mesh refinement, i.e., refinement over the entire spatial domain, in all spatial coordinate directions, and in time (for unsteady problems). For the current example, this parameter is given by

$$h = \frac{\Delta x}{\Delta x_{\text{ref}}} = \frac{\Delta t}{\Delta t_{\text{ref}}}, \tag{5.10}$$

where Δx_{ref} and Δt_{ref} refer to some arbitrary reference spatial node spacing and time step, respectively. Later, the h parameter will be extended to consider different refinement ratios in time and even in the different coordinate directions. For the current purposes, the important point is that when h goes to zero, it implies that Δx and Δt also go to zero at the same rate. Note that, for this finite difference discretization, T_h represents a vector of temperature values defined at each node and time step.

In order to find the truncation error for the numerical scheme given in Eq. (5.8), we can expand the above temperature values in a Taylor series about the temperature at spatial

location i and time step n (assuming sufficient differentiability of T):

$$T_i^{n+1} = T_i^n + \left.\frac{\partial T}{\partial t}\right|_i^n \frac{\Delta t}{1!} + \left.\frac{\partial^2 T}{\partial t^2}\right|_i^n \frac{(\Delta t)^2}{2!} + \left.\frac{\partial^3 T}{\partial t^3}\right|_i^n \frac{(\Delta t)^3}{3!} + O\left(\Delta t^4\right),$$

$$T_{i+1}^n = T_i^n + \left.\frac{\partial T}{\partial x}\right|_i^n \frac{\Delta x}{1!} + \left.\frac{\partial^2 T}{\partial x^2}\right|_i^n \frac{(\Delta x)^2}{2!} + \left.\frac{\partial^3 T}{\partial x^3}\right|_i^n \frac{(\Delta x)^3}{3!} + O\left(\Delta x^4\right),$$

$$T_{i-1}^n = T_i^n + \left.\frac{\partial T}{\partial x}\right|_i^n \frac{(-\Delta x)}{1!} + \left.\frac{\partial^2 T}{\partial x^2}\right|_i^n \frac{(-\Delta x)^2}{2!} + \left.\frac{\partial^3 T}{\partial x^3}\right|_i^n \frac{(-\Delta x)^3}{3!} + O\left(\Delta x^4\right).$$

Substituting these expressions into the discrete equation and rearranging yields

$$\underbrace{\frac{T_i^{n+1} - T_i^n}{\Delta t} - \alpha \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta x)^2}}_{L_h(T)} = \underbrace{\frac{\partial T}{\partial t} - \alpha \frac{\partial^2 T}{\partial x^2}}_{L(T)} + \underbrace{\left[\frac{1}{2}\frac{\partial^2 T}{\partial t^2}\right]\Delta t + \left[-\frac{\alpha}{12}\frac{\partial^4 T}{\partial x^4}\right](\Delta x)^2 + O\left(\Delta t^2, \Delta x^4\right)}_{\text{truncation error: } TE_h(T)}. \quad (5.11)$$

Thus we have the general relationship that the discrete equation equals the mathematical model plus the truncation error. In order for this equality to make sense, it is implied that either (1) the continuous derivatives in $L(T)$ and $TE_h(T)$ are restricted to the nodal points or (2) the discrete operator $L_h(T)$ is mapped onto a continuous space.

5.2.1.2 Generalized truncation error expression (GTEE)

Using the operator notation discussed earlier and substituting in the generic (sufficiently smooth) dependent variable u into Eq. (5.11) yields

$$L_h(u) = L(u) + TE_h(u), \quad (5.12)$$

where we again assume an appropriate mapping of the operators onto either a continuous or discrete space. We refer to Eq. (5.12) as the *generalized truncation error expression (GTEE)* and it is used extensively in this chapter as well as in Chapters 8 and 9. It relates the discrete equations to the mathematical model in a very general manner and is one of the most important equations in the evaluation, estimation, and reduction of discretization errors in scientific computing. When set to zero, the right hand side of the GTEE can be thought of as the actual mathematical model that is solved by the discretization scheme $L_h(u_h) = 0$. The GTEE is the starting point for determining the *consistency* and the *formal order of accuracy* of the numerical method. While the GTEE can be derived even for nonlinear mathematical models, for linear (or linearized) mathematical models, it also explicitly shows the relationship between the truncation error and the discretization error. As will be shown in Chapter 8, the GTEE can also be used to provide estimates of the truncation error. Finally, this equation provides a general relationship between the discrete equation and the (possibly nonlinear) mathematical model since we have not

specified what the function u is, only that it satisfies certain differentiability constraints. It is relatively straightforward (although somewhat tedious) to show that the general polynomial function

$$u(x,t) = \sum_{i=0}^{N_x} a_i x^i + \sum_{j=0}^{N_t} b_j t^j$$

will satisfy Eq. (5.12) exactly for the example problem of 1-D unsteady heat conduction given above (hint: choose N_x and N_t small enough such that the higher-order terms are zero).

Most authors (e.g., Richtmyer and Morton, 1967; Ferziger and Peric, 2002) formally define the truncation error only when the exact solution to the mathematical model is inserted into Eq. (5.12), thus resulting in

$$L_h(\tilde{u}) = TE_h(\tilde{u}) \tag{5.13}$$

since $L(\tilde{u}) = 0$. For our specific finite difference example for the 1D unsteady heat equation given above, we have

$$\frac{\tilde{T}_i^{n+1} - \tilde{T}_i^n}{\Delta t} - \alpha \frac{\tilde{T}_{i+1}^n - 2\tilde{T}_i^n + \tilde{T}_{i-1}^n}{(\Delta x)^2} = \left[\frac{1}{2}\frac{\partial^2 \tilde{T}}{\partial t^2}\right]\Delta t + \left[-\frac{\alpha}{12}\frac{\partial^4 \tilde{T}}{\partial x^4}\right](\Delta x)^2 + O\left(\Delta t^2, \Delta x^4\right) = TE_h\left(\tilde{T}\right), \tag{5.14}$$

where the notation $\tilde{T}_i^n$ implies that the exact solution is restricted to spatial location i and temporal location n. For the purposes of this book, we will employ the GTEE found from Eq. (5.12) since it will provide more flexibility in how the truncation error is used.

5.2.2 Discretization error

Discretization error is formally defined as the difference between the exact solution to the discrete equations and the exact solution to the mathematical model. Using our earlier notation, we can thus write the discretization error for the general dependent variable u as

$$\varepsilon_h = u_h - \tilde{u}, \tag{5.15}$$

where again the h subscript denotes the exact solution to the discrete equations and the overtilde denotes the exact solution to the mathematical model.

5.2.3 Consistency

For a numerical scheme to be *consistent*, the discretized equations $L_h(\cdot)$ must approach the mathematical model equations $L(\cdot)$ in the limit as the discretization parameters (Δx, Δy, Δz, Δt, denoted collectively by the parameter h) approach zero. In terms of the truncation error discussion above, a consistent numerical scheme can be defined as one in which the

truncation error vanishes in the limit as $h \to 0$. Not all numerical schemes are consistent, and one notable example is the DuFort–Frankel finite difference method applied to the unsteady heat conduction equation, which has a leading truncation error term proportional to $(\Delta t/\Delta x)^2$ (Tannehill *et al.*, 1997). This scheme is only consistent under the restriction that Δt approach zero at a faster rate than Δx.

5.2.4 Stability

For initial value (i.e., hyperbolic and parabolic) problems, a discretization scheme is said to be *stable* if numerical errors do not grow unbounded in the marching direction. The numerical errors are typically considered to come from computer round-off (Ferziger and Peric, 2002), but in fact can come from any source. The idea of numerical stability originally derives from initial value problems for hyperbolic and parabolic partial differential equations (Crank and Nicolson, 1947), but the concepts can also be applied to relaxation methods for elliptic problems (e.g., see Hirsch, 2007). It is important to note that the concept of numerical stability applies only to the discrete equations (Hirsch, 2007), and should not be confused with natural instabilities that can arise in the mathematical model itself.

Most approaches for analyzing numerical stability apply only to linear partial differential equations with constant coefficients. The most popular approach for determining stability is von Neumann stability analysis (Hirsch, 2007). Also referred to as Fourier stability analysis, von Neumann's method employs a Fourier decomposition of the numerical error and neglects the boundary conditions by assuming these error components are periodic. The fact that von Neumann's method neglects the boundary conditions is not overly restrictive in practice and results in a fairly straightforward stability analysis (e.g., see Richtmyer and Morton, 1967; Hirsch, 2007). However, the restriction to linear differential equations with constant coefficients is significant. Time and time again, we will find that many of our tools for analyzing the behavior of numerical schemes are only applicable to linear equations. We are thus left in the uncomfortable situation of hoping that we can simply extend the results of these methods to the complicated, nonlinear mathematical models of interest, and this fact should not be forgotten. As a practical matter when dealing with nonlinear problems, a stability analysis should be performed for the linearized problem to provide initial guidance on the stability limits; then numerical tests should be performed to confirm the stability restrictions for the nonlinear problem.

5.2.5 Convergence

Convergence addresses whether or not the exact solution to the discrete equations approaches the exact solution to the mathematical model in the limit of decreasing mesh spacing and time step size. Whereas convergence and consistency both address the limiting behavior of the discrete method relative to the mathematical model, convergence deals with the solution while consistency deals with the equations. This definition of convergence

should not be confused with convergence of an iterative method (see Chapter 7), which we will refer to as iterative convergence.

For marching problems, convergence is determined by Lax's equivalence theorem, which is again valid only for linear equations. Lax's theorem states that, given a well-posed initial value problem and a consistent numerical scheme, stability is the necessary and sufficient condition for convergence (Richtmyer and Morton, 1967). When used for code verification purposes, convergence is demonstrated by examining the actual behavior of the discretization error ε_h as $h \to 0$.

Recent work with finite volume methods (Despres, 2004) suggests that some modifications to (or perhaps clarifications of) Lax's equivalence theorem may be needed. In his work, Despres claims that the finite volume method is formally inconsistent for 2-D triangular meshes, but found it to be convergent assuming certain solution regularity constraints. It is interesting to note that while Despres does provide theoretical developments, numerical examples are not included. Given the work of Despres (2004) and the references cited therein, it is possible that Lax's theorem should be augmented with the additional assumption of systematic mesh refinement (discussed in Section 5.4) along with mesh topology restrictions. Additional work is required to understand these mesh quality and topology issues as they relate to the consistency and convergence of discretization schemes.

5.3 Order of accuracy

The term *order of accuracy* refers to the rate at which the discrete solution approaches the exact solution to the mathematical model in the limit as the discretization parameters go to zero. The order of accuracy can be addressed in either a theoretical sense (i.e., the order of accuracy of a given numerical scheme assuming it has been implemented correctly) or in a more empirical manner (i.e., the actual order of accuracy of discrete solutions). The former is called the *formal order of accuracy*, while the latter is the *observed order of accuracy*. These two terms are discussed in detail below.

5.3.1 Formal order of accuracy

The *formal order of accuracy* is the theoretical rate of convergence of the discrete solution u_h to the exact solution to the mathematical model $\tilde{u}$. This theoretical rate is defined only in an asymptotic sense as the discretization parameters (Δx, Δy, Δt, etc., currently represented by the single parameter h) go to zero in a systematic manner. It will be shown next that the formal order of accuracy can be related back to the truncation error; however, we are limited to linear (or linearized) equations to show this relationship.

The key relationship between the discrete equation and the mathematical model is the GTEE given by Eq. (5.12), which is repeated here for convenience:

$$L_h(u) = L(u) + TE_h(u). \tag{5.12}$$

Inserting the exact solution to the discrete equation u_h into Eq. (5.12), then subtracting the original mathematical model equation $L(\tilde{u}) = 0$, yields

$$L(u_h) - L(\tilde{u}) + TE_h(u_h) = 0.$$

If the mathematical operator $L(\cdot)$ is linear (or linearized), then $L(u_h) - L(\tilde{u}) = L(u_h - \tilde{u})$. The difference between the discrete solution u_h and the exact solution to the mathematical model $\tilde{u}$ is simply the discretization error ε_h defined by Eq. (5.15), thus we find that the discretization error and the truncation error are related by

$$L(\varepsilon_h) = -TE_h(u_h). \tag{5.16}$$

Equation (5.16) governs the transport of the discretization error and is called the *continuous discretization error transport equation* since it employs the continuous mathematical operator (Roy, 2009). According to this equation, the discretization error is propagated in the same manner as the original solution u. For example, if the original mathematical model contains terms governing the convection and diffusion of u, then the discretization error ε_h will also be convected and diffused. More importantly in the context of our present discussion, Eq. (5.16) also shows that the truncation error serves as the local source for the discretization error (Ferziger and Peric, 2002); thus the rate of reduction of the local truncation error with mesh refinement will produce corresponding reductions in the discretization error. Applying this continuous error transport equation to the 1-D unsteady heat conduction example from Section 5.2.1 results in:

$$\frac{\partial \varepsilon_h}{\partial t} - \alpha \frac{\partial^2 \varepsilon_h}{\partial x^2} = -\left[\frac{1}{2}\frac{\partial^2 T_h}{\partial t^2}\right]\Delta t - \left[-\frac{\alpha}{12}\frac{\partial^4 T_h}{\partial x^4}\right](\Delta x)^2 + O\left(\Delta t^2, \Delta x^4\right).$$

Having tied the rate of reduction of the discretization error to the truncation error, we are now in the position to define the formal order of accuracy of a numerical scheme as the smallest exponent which acts upon a discretization parameter in the truncation error, since this will dominate the limiting behavior as $h \to 0$. For problems in space and time, it is sometimes helpful to refer to the formal order of accuracy in time separately from the formal order of accuracy in space. For the 1-D unsteady heat conduction example above, the simple explicit finite difference discretization is formally first-order accurate in time and second-order accurate in space. While it is not uncommon to use different order-of-accuracy discretizations for different terms in the equations (e.g., third-order convection and second-order diffusion), the formal order of accuracy of such a mixed-order scheme is simply equal to the lowest order accurate discretization employed.

The truncation error can usually be derived for even complicated, nonlinear discretization methods (e.g., see Grinstein *et al.*, 2007). For cases where the formal order of accuracy has not been determined from a truncation error analysis, there are three approaches that can be used to estimate the formal order of accuracy (note that Knupp (2009) refers to this as the "expected" order of accuracy). The first approach is to approximate the truncation error by inserting the exact solution to the mathematical model into the discrete equations.

Since the exact solution to the mathematical model will not satisfy the discrete equation, the remainder (i.e., the discrete residual) will approximate the truncation error as shown in Eq. (5.13). By evaluating the discrete residual on successively finer meshes (i.e., as $h \to 0$), the rate of reduction of the truncation error can be estimated, thus producing the formal order of accuracy of the discretization scheme (assuming no coding mistakes are present). This first approach is called the residual method and is discussed in detail in Section 5.5.6.1. The second approach is to compute the observed order of accuracy for a series of meshes as $h \to 0$, and this approach is addressed in the next section. In this case, if the observed order of accuracy is found to be two, then one is safe to assume that the formal order of accuracy is *at least* second order. The final approach is to simply assume the expected order of accuracy from the quadrature employed in the discretization. For example, a linear basis function used with the finite element method generally results in a second-order accurate scheme for the dependent variables, as does linear interpolation/extrapolation when used to determine the interfacial fluxes in the finite volume method (a process sometimes referred to as flux quadrature).

In the development of the truncation error, a certain degree of solution smoothness was assumed. As such, the formal order of accuracy can be reduced in the presence of discontinuities and singularities in the solution. For example, the observed order of accuracy for inviscid gas dynamics problems containing shock waves has been shown to reduce to first order for a wide range of numerical discretization approaches (e.g., Engquist and Sjogreen, 1998; Carpenter and Casper, 1999; Roy, 2003; Banks *et al.*, 2008), regardless of the formal order of accuracy of the scheme on smooth problems. Furthermore, for linear discontinuities (e.g., contact discontinuities and slip lines in inviscid gas dynamics), the formal order generally reduces to $p/(p+1)$ (i.e., below one) for methods which have a formal order p for smooth problems (Banks *et al.*, 2008). In some situations, the formal order of accuracy of a numerical method may be difficult to determine since it depends on the nature and strength of the discontinuities/singularities present in the solution.

5.3.2 Observed order of accuracy

As discussed above, the *observed order of accuracy* is the actual order of accuracy obtained on a series of systematically-refined meshes. For now, we only consider the case where the exact solution to the mathematical model is known. For this case, the discretization error can be evaluated exactly (or at least within round-off and iterative error) and only two mesh levels are required to compute the observed order of accuracy. The more difficult case of computing the observed order of accuracy when the exact solution to the mathematical model is not known is deferred to Chapter 8.

Consider a series expansion of the solution to the discrete equations u_h in terms of the mesh spacing h in the limit as $h \to 0$:

$$u_h = u_{h=0} + \left.\frac{\partial u}{\partial h}\right|_{h=0} h + \left.\frac{\partial^2 u}{\partial h^2}\right|_{h=0} \frac{h^2}{2} + \left.\frac{\partial^3 u}{\partial h^3}\right|_{h=0} \frac{h^3}{6} + O(h^4). \tag{5.17}$$

If a convergent numerical scheme is employed (i.e., if it is consistent and stable), then we have $u_{h=0} = \tilde{u}$. Furthermore, for a formally second-order accurate scheme, by definition we will have $\frac{\partial u}{\partial h}\big|_{h=0} = 0$ since terms of order h do not appear in the truncation error (recall Eq. (5.16)). Employing the definition of the discretization error from Eq. (5.15) we find that for a general second-order accurate numerical scheme

$$\varepsilon_h = g_2 h^2 + O(h^3), \tag{5.18}$$

where the coefficient $g_2 = g_2(x, y, z, t)$ only and is thus independent of h (Ferziger and Peric, 2002). Note that for discretizations that exclusively employ second-order accurate central-type differencing, the truncation error only contains even powers of h, and thus the higher order terms in Eq. (5.18) would be $O(h^4)$. For a more general pth-order accurate scheme, we have

$$\varepsilon_h = g_p h^p + O(h^{p+1}) \tag{5.19}$$

unless again central-type differencing is used, whereupon the higher order terms will be $O(h^{p+2})$.

Equation (5.19) provides an appropriate theoretical starting point for computing the observed order of accuracy. In the limit as $h \to 0$, the higher-order terms in Eq. (5.19) will become small relative to the leading term and can be neglected. Consider now two discrete solutions, one computed on a fine mesh with spacing h and another computed on a coarse mesh with spacing $2h$ found by eliminating every other cell or node from the fine mesh. Neglecting the higher-order terms (whether they are small or not), Eq. (5.19) can be written for the two solutions as

$$\varepsilon_{2h} = g_p (2h)^{\hat{p}},$$
$$\varepsilon_h = g_p h^{\hat{p}}.$$

Dividing the first equation by the second one, then taking the natural log, we can solve for the observed order of accuracy to give

$$\hat{p} = \frac{\ln\left(\frac{\varepsilon_{2h}}{\varepsilon_h}\right)}{\ln(2)}. \tag{5.20}$$

Here the "^" is used to differentiate this *observed order of accuracy* from the formal order of accuracy of the method. The observed order can be computed regardless of whether or not the higher-order terms in Eq. (5.19) are small; however, this observed order of accuracy $\hat{p}$ will only match the formal order of accuracy when the higher-order terms are in fact small, i.e., in the limiting sense as $h \to 0$.

A more general expression for the observed order of accuracy can be found that applies to meshes that are systematically refined by an arbitrary factor. Introducing the grid refinement factor r which is defined as the ratio of coarse to fine grid mesh spacing,

$$r \equiv \frac{h_{\text{coarse}}}{h_{\text{fine}}}, \tag{5.21}$$

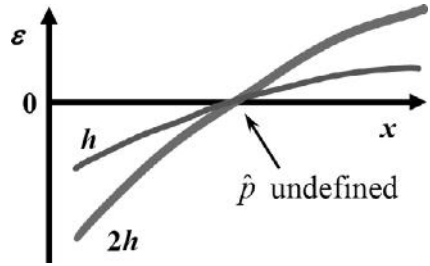

Figure 5.1 Qualitative plot of local discretization error on a coarse mesh ($2h$) and a fine mesh (h) showing that the observed order of accuracy from Eq. (5.20) can be undefined when examined locally.

where we require $r > 1$, the discretization error expansion for the two mesh levels becomes

$$\varepsilon_{rh} = g_p (rh)^{\hat{p}},$$
$$\varepsilon_h = g_p h^{\hat{p}}.$$

Again dividing the first equation by the second and taking the natural log, we find the more general expression for the observed order of accuracy

$$\hat{p} = \frac{\ln\left(\frac{\varepsilon_{rh}}{\varepsilon_h}\right)}{\ln(r)}. \tag{5.22}$$

At this point, it is important to mention that the exact solution to the discretized equations u_h is generally unknown due to the presence of round-off and iterative convergence errors in the numerical solutions. While round-off and iterative errors are discussed in detail in Chapter 7, their impact on the observed order of accuracy will be addressed in Section 5.3.2.2.

The observed order of accuracy can be evaluated using the discretization error in the dependent variables, norms of those errors, or the discretization error in any quantities that can be derived from the solution. When applied to norms of the discretization error, the relationship for the observed order of accuracy becomes

$$\hat{p} = \frac{\ln\left(\frac{\|\varepsilon_{rh}\|}{\|\varepsilon_h\|}\right)}{\ln(r)}, \tag{5.23}$$

where any of the norms discussed in Section 5.1 can be used. Care must be taken when computing the observed order of accuracy locally since unrealistic orders can be produced. For example, when the discrete solutions approach the exact solution to the mathematical model from below in some region and from above in another, the observed order of accuracy will be undefined at the crossover point. Figure 5.1 gives an example of just such a case and shows the discretization error versus a spatial coordinate x. Applying Eq. (5.22) would likely produce $\hat{p} \approx 1$ almost everywhere except at the crossover point, where if $\varepsilon_h = \varepsilon_{2h} = 0$, the observed order of accuracy from Eq. (5.22) is undefined (Potter *et al.*,

2005). For this reason, global quantities are recommended rather than local quantities for code verification purposes.

The observed order of accuracy can fail to match the nominal formal order due to mistakes in the computer code, discrete solutions which are not in the asymptotic range (i.e., when the higher-order terms in the truncation error are not small), and the presence of round-off and iterative error. The latter two issues are discussed in the following sections.

5.3.2.1 Asymptotic range

The *asymptotic range* is defined as the range of discretization sizes (Δx, Δy, Δt, etc., denoted here collectively by the parameter h) where the lowest-order terms in the truncation error and discretization error expansions dominate. It is only in the asymptotic range that the limiting behavior of these errors can be observed. For code verification purposes, the order of accuracy test will only be successful when the solutions are in this asymptotic range. In our experience, the *asymptotic range is surprisingly difficult to identify and achieve* for all but the simplest scientific computing applications. Even an experienced code user with a good intuition on the mesh resolution required to obtain a "good" solution will generally underestimate the resolution required to obtain the asymptotic range. As we shall see later in Chapter 8, all approaches for estimating discretization error also rely on the solution(s) being asymptotic.

5.3.2.2 Effects of iterative and round-off error

Recall that the underlying theory used to develop the general observed order-of-accuracy expression given in Eq. (5.22) made use of the exact solution to the discrete equation u_h. In practice, u_h is only known within some tolerance determined by the number of digits used in the computations (i.e., round-off error) and the criterion used for iterative convergence. The discrete equations can generally be iteratively solved to within machine round-off error; however, in practice, the iterative procedure is usually terminated earlier to reduce computational effort. Round-off and iterative error are discussed in detail in Chapter 7. To ensure that the computed solutions are accurate approximations of the exact solution to the discrete equations u_h, both round-off and iterative error should be at least 100 times smaller than the discretization error on the finest mesh employed (i.e., $\leq 0.01 \times \varepsilon_h$) (Roy, 2005).

5.4 Systematic mesh refinement

Up to this point, the asymptotic behavior of the truncation and discretization error (and thus the formal and observed orders of accuracy) has been addressed by the somewhat vague notion of taking the limit as the discretization parameters (Δx, Δy, Δt, etc.) go to zero. For time-dependent problems, refinement in time is straightforward since the time step is fixed over the spatial domain and can be coarsened or refined by an arbitrary factor, subject of course to stability constraints. For problems involving the discretization of a spatial domain, the refinement process is more challenging since the spatial mesh resolution and quality can

vary significantly over the domain depending on the geometric complexity. In this section, we introduce the concept of systematic mesh refinement which requires uniformity of the refinement over the spatial domain and consistency of the refinement as $h \to 0$. These two requirements are discussed in detail, as well as additional issues related to the use of local and/or global mesh transformations and mesh topology.

5.4.1 Uniform mesh refinement

Local refinement of the mesh in selected regions of interest, while often useful for reducing the discretization error, is not appropriate for assessing the asymptotic behavior of discrete solutions. The reason is that the series expansion for the discretization error given in Eq. (5.17) is in terms of a single parameter h which is assumed to apply over the entire domain. When the mesh is refined locally in one region but not in another, then the refinement can no longer be described by a single parameter. This same concept holds for mesh refinement in only one coordinate direction, which requires special procedures when used for evaluating the observed order of accuracy (see Section 5.5.3) or for estimating the discretization error. The requirement that the mesh refinement be uniform is not the same as requiring that the mesh itself be uniform, only that it be refined in a uniform manner. Note that this requirement of uniform refinement is not restricted to integer refinement factors between meshes. Assuming that a coarse and fine mesh are related through *uniform refinement*, the grid refinement factor can be computed as

$$r_{12} = \left(\frac{N_1}{N_2} \right)^{1/d}, \tag{5.24}$$

where N_1 and N_2 are the number of nodes/cells/elements on the fine and coarse meshes, respectively, and d is the number of spatial dimensions.

An example of uniform and nonuniform mesh refinement is presented in Figure 5.2. The initial coarse mesh (Figure 5.2a) has 4×4 cells, and when this mesh is refined by a factor of two in each direction, the resulting uniformly refined mesh (Figure 5.2b) has 8×8 cells. The mesh shown in Figure 5.2c also has 8×8 cells, but has selectively refined to the x-axis and in the middle of the two bounding arcs. While the average cell length scale is refined by a factor of two, the local cell length scale varies over the domain, thus this mesh has not been uniformly refined.

5.4.2 Consistent mesh refinement

In general, one should not expect to obtain convergent solutions with mesh refinement when poor quality meshes are used. This is certainly true for the extreme example of a mesh with degenerate cells, such as those involving mesh crossover (see Figure 5.3), that persist with uniform refinement. We now introduce the concept of *consistent mesh refinement* which requires that mesh quality must either stay constant or improve in the limit as $h \to 0$.

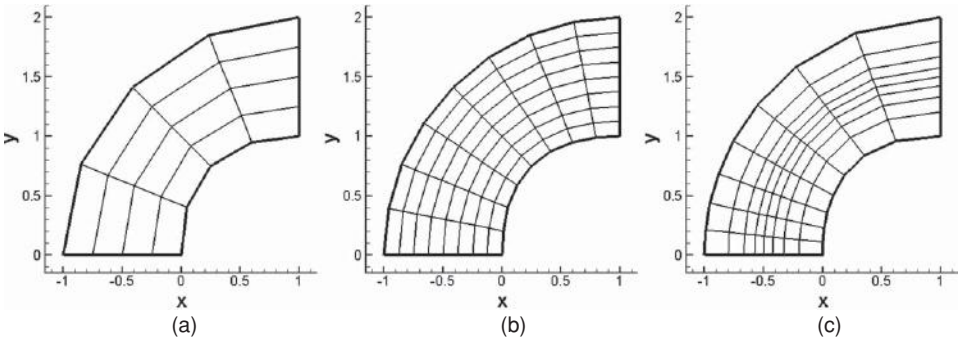

Figure 5.2 Example of uniform and nonuniform mesh refinement: (a) coarse mesh with 4×4 cells, (b) uniformly refined mesh with 8×8 cells, and (c) nonuniformly refined mesh with 8×8 cells.

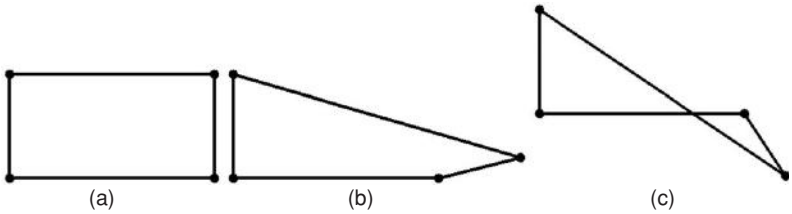

Figure 5.3 Example of a degenerate cell due to mesh crossover: (a) initial quadrilateral cell, (b) intermediate skewing of the cell, and (c) final skewing resulting in mesh crossover.

Examples of mesh quality metrics include cell aspect ratio, skewness, and stretching rate (i.e., the rate at which the mesh transitions from coarse to fine spacing).

To further illustrate the concept of consistent mesh refinement, consider the simple 2-D triangular mesh over the square domain given in Figure 5.4a. While this initial coarse mesh certainly has poor quality, it is the approach used for refining this mesh that will determine the consistency. Consider now three cases where this initial coarse mesh is uniformly refined. In the first case, the midpoints of each edge are connected so that each coarse mesh cell is decomposed into four finer cells with similar shape, as shown in Figure 5.4b. Clearly, if this refinement procedure is performed repeatedly, then in the limit as $h \to 0$ even very fine meshes will retain the same mesh qualities related to cell skewness, cell volume variation, and cell stretching (i.e., the change in cell size from one region to another). Another refinement approach might allow for different connectivity of the cells and also provide more flexibility in choosing new node locations (i.e., not necessarily at edge midpoints) as shown in Figure 5.4c. As this refinement approach is applied in the limit as $h \to 0$, the mesh quality will generally improve. A third refinement strategy might employ arbitrary edge node placement while not allowing changes in the cell-to-cell connectivity (Figure 5.4d). Considering Figure 5.4, refinement strategy (b) employs fixed quality meshes and strategy (c) employs meshes with improving quality as $h \to 0$, thus both are considered

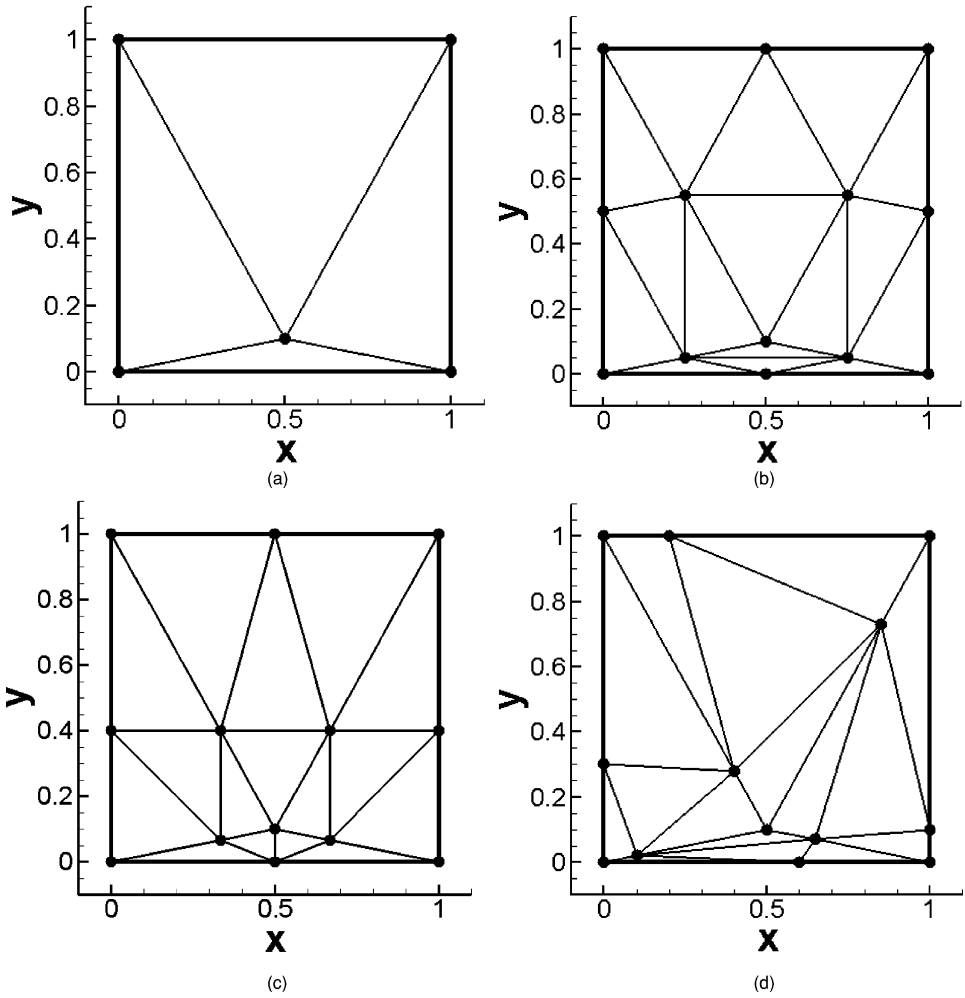

Figure 5.4 Example showing consistent and inconsistent mesh refinement: (a) poor quality coarse mesh with four unstructured triangular cells, (b) uniformly refined mesh that retains a fixed mesh quality, (c) uniformly refined mesh with improved mesh quality, and (d) uniformly refined mesh with inconsistent refinement.

consistent. Strategy (d) is an inconsistent mesh refinement approach since the quality of the mesh degrades with refinement.

Borrowing concepts from Knupp (2003), we assume the existence of a global mesh quality metric σ which varies between 0 and 1, with $\sigma = 1$ denoting an isotropic mesh, i.e., one with "ideal" mesh quality (square quadrilaterals, cubic hexahedrals, equilateral triangles, etc.). Smaller values of σ would denote anisotropic meshes with lower quality (skewness, stretching, curvature, etc). Consistent mesh refinement can thus be defined by requiring that $\sigma_{\text{fine}} \geq \sigma_{\text{coarse}}$ during refinement. Consistent refinement with $\sigma \to 1$ as $h \to 0$

can place significant burdens on the mesh generation and refinement procedure (especially for unstructured meshes), but provides the easiest criteria for discretization schemes to satisfy since meshes become more isotropic with refinement (e.g., they become Cartesian for quadrilateral and hexahedral meshes). A more difficult mesh quality requirement to satisfy from the code verification point of view is to require convergence of the numerical solutions for meshes with a fixed quality measure σ as $h \rightarrow 0$. Such nuances relating the numerical scheme behavior to the mesh quality fall under the heading of solution verification and are addressed in more detail in Chapters 8 and 9. For code verification purposes, it is important to document the asymptotic behavior of the quality of the meshes used for the code verification study.

5.4.3 Mesh transformations

Inherent in some discretization methods is an assumption that the mesh employs uniform spacing (i.e., Δx, Δy, and Δz are constant). When applied blindly to the cases with nonuniform meshes, schemes that are formally second-order accurate on uniform meshes will often reduce to first-order accuracy on nonuniform meshes. While Ferziger and Peric (1996) argue that these first-order errors will either be limited to small fractions of the domain or even vanish as the mesh is refined, this may result in extremely fine meshes to obtain the asymptotic range. The key point is that additional discretization errors will be introduced by nonuniform meshes.

In order to mitigate these additional errors, mesh transformations are sometimes used to handle complex geometries and to allow local mesh refinement. For discretization methods employing body-fitted structured (i.e., curvilinear) meshes, these transformations often take the form of global transformations of the governing equations. For finite-volume and finite-element methods on structured or unstructured meshes, these transformations usually take the form of local mesh transformations centered about each cell or element. An example of a global transformation for a body-fitted structured grid in 2-D is presented in Figure 5.5. The transformation must ensure a one-to-one mapping between the grid line intersections in physical space (a) and those in computational coordinates (b).

Consider the 2-D steady transformation from physical space (x, y) to a uniform computational space (ξ, η) given by:

$$\xi = \xi(x, y),$$
$$\eta = \eta(x, y). \tag{5.25}$$

Using the chain rule, it can be shown that derivatives in physical space can be converted to derivatives in the uniform computational space (Thompson *et al.*, 1985), e.g.,

$$\frac{\partial u}{\partial x} = \frac{y_\eta}{J}\frac{\partial u}{\partial \xi} - \frac{y_\xi}{J}\frac{\partial u}{\partial \eta}, \tag{5.26}$$

where y_η and y_ξ are metrics of the transformation and J is the Jacobian of the transformation defined as $J = x_\xi y_\eta - x_\eta y_\xi$. The accuracy of the discrete approximation of the solution

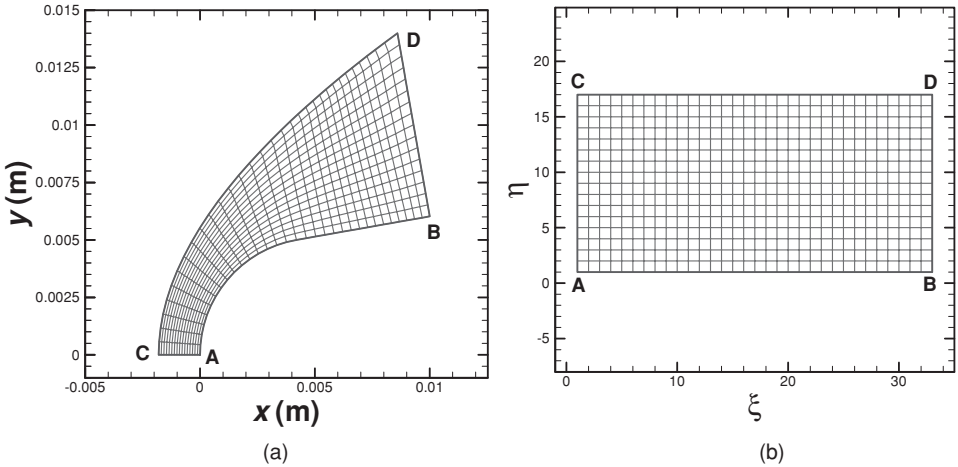

Figure 5.5 Example of a global transformation of a 2-D body-fitted structured mesh: (a) mesh in physical (x, y) coordinates and (b) mesh in the transformed computational (ξ, η) coordinates.

derivatives will depend on the chosen discretization scheme, the mesh resolution, the mesh quality, and the solution behavior (Roy, 2009).

The transformation itself can either be analytic or discrete in nature. Thompson *et al.* (1985) point out that using the same discrete approximation for the metrics that is used for solution derivatives can often result in smaller numerical errors compared to the case where purely analytic metrics are used. This surprising result occurs due to error cancellation and can be easily shown by examining the truncation error of the first derivative in one dimension (Mastin, 1999). As an example of a discrete approximation of the metrics, consider the metric term x_ξ which can be approximated using central differences to second-order accuracy as

$$x_\xi = \frac{x_{i+1} - x_{i-1}}{2\Delta\xi} + O(\Delta\xi^2).$$

When discrete transformations are used, they should be of the same order as, or possibly higher-order than, the underlying discretization scheme to ensure that the formal order of accuracy is not reduced. While mistakes in the discrete transformations can adversely impact the numerical solutions, these mistakes can be detected during the code verification process assuming sufficiently general mesh topologies are employed.

5.4.4 Mesh topology issues

There are many different mesh topologies that can be used in scientific computing. When conducting code verification studies, it is recommended that the most general mesh topology that will be employed for solving the problems of interest be used for the code verification studies. For example, if simulations will only be performed on Cartesian meshes, then

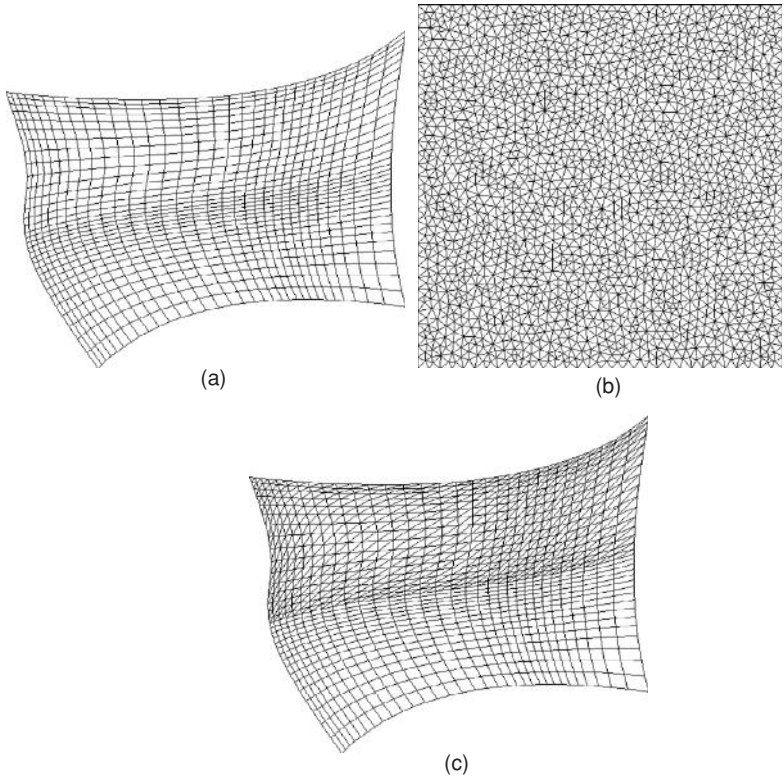

Figure 5.6 Example mesh topologies in 2-D: (a) structured curvilinear, (b) unstructured triangles, and (c) hybrid structured/unstructured curvilinear (adapted from Veluri *et al.*, 2008).

it is sufficient to conduct the code verification studies on Cartesian meshes. However, if simulations will be performed on non-isotropic (i.e., nonideal) meshes consisting of a combination of hexahedral, prismatic, and tetrahedral cells, then those mesh topologies should be used during code verification.

Meshes in 1-D consist of an order set of nodes or cells that may either be uniformly or nonuniformly distributed. For 2-D meshes, the nodes/cells may either be structured quadrilaterals, unstructured triangles, unstructured polygons with an arbitrary number of sides, or some hybrid combination of these. Examples of a hierarchy of 2-D mesh topologies appropriate for code verification are given in Figure 5.6 (Veluri *et al.*, 2008).

In 3-D, structured meshes can either be Cartesian, stretched Cartesian, or curvilinear (i.e., body fitted). 3-D unstructured meshes can contain cells that are tetrahedral (four-side pyramids), pyramidal (five-sided pyramids), prismatic (any 2-D cell type extruded in the third direction), hexahedral, polyhedral, or hybrid combinations of these. An example of a general 3-D hybrid mesh topology that has been employed for performing code verification on a scientific computing code with general unstructured mesh capabilities is given in Figure 5.7. This mesh consists of hexahedral and prismatic triangular cells extruded from

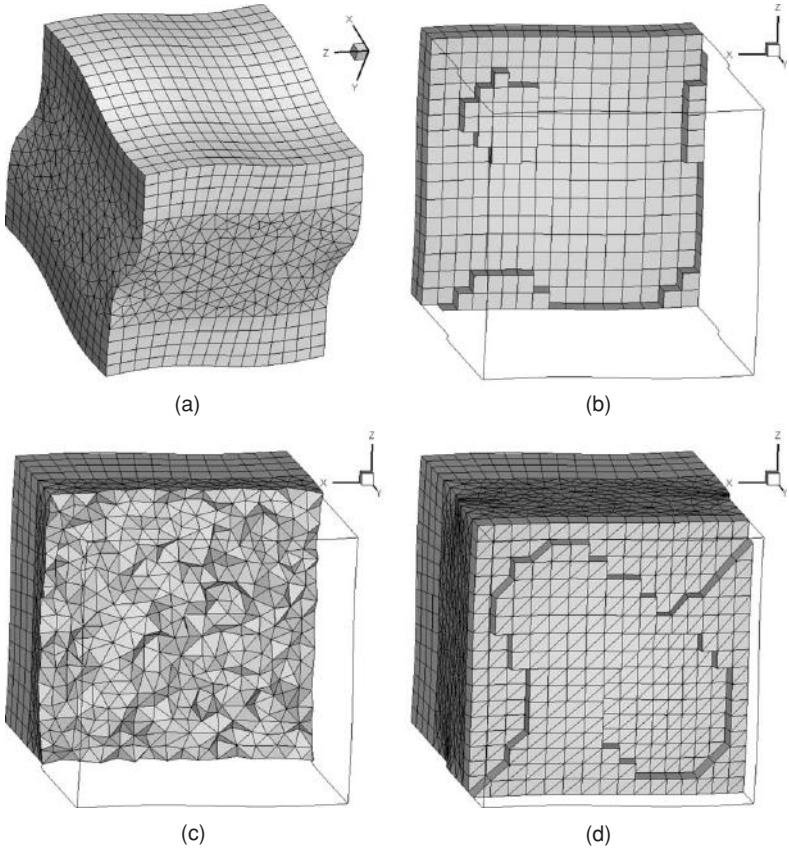

Figure 5.7 A general hybrid mesh topology in 3-D: (a) full 3-D mesh, (b) internal view showing hexahedral cells, (c) internal view showing tetrahedral cells, and (d) internal view showing prismatic cells.

the curved $y_{\min}$ and $y_{\max}$ boundaries joined together with a region of tetrahedral cells in the middle.

5.5 Order verification procedures

There are two books which deal with the subject of order-of-accuracy verification. Roache (1998) provides an overview of the subject, with emphasis on order verification using the method of manufactured solutions (discussed in Chapter 6). The book by Knupp and Salari (2003) is entirely dedicated to code order verification and is one of the most comprehensive references on the subject. Although Knupp and Salari prefer the terminology "code order verification," here we will simply use "order verification" to refer to order-of-accuracy verification of a scientific computing code. More recent reviews of order verification procedures are provided by Roy (2005) and Knupp *et al*. (2007).

Order verification entails a comparison between the limiting behavior of the observed order of accuracy and the formal order. Once an order verification test has been passed, then the code is considered verified for the code options (submodels, numerical algorithms, boundary conditions, etc.) exercised in the verification test. Any further order verification performed for those code options is simply considered confirmation of code correctness (Roache, 1998).

This section addresses the order verification procedures applicable for discretizations in space and/or time. These procedures can be invaluable for identifying the presence of coding mistakes (i.e., bugs) and problems with the numerical algorithms. Techniques are also discussed to aid in the debugging process once a coding mistake is found to exist. Limitations of the order verification procedure are then described, as well as different variants of the standard order verification procedure. This section concludes with a discussion of who bears the responsibility for code verification.

5.5.1 Spatial discretization

This section describes the order verification procedure for steady-state problems, i.e., those that do not have time as an independent variable. The order verification procedure discussed here is adapted from the procedure recommended by Knupp and Salari (2003). In brief, this procedure is used to determine whether or not the code output (numerical solution and other system response quantities) converges to the exact solution to the mathematical model at the formal rate with systematic mesh refinement. If the formal order of accuracy is observed in an asymptotic sense, then the code is considered verified for the coding options exercised. Failure to achieve the formal order of accuracy indicates the presence of a coding mistake or a problem with the numerical algorithm. The steps in the order verification procedure for steady-state problems are presented in Figure 5.8, and these steps are discussed in detail below.

1 Define mathematical model

The governing equations (i.e., the mathematical model) generally occur in partial differential or integral form and must be specified unambiguously along with any initial conditions, boundary conditions, and auxiliary equations. Small errors in defining the mathematical model can easily cause the order verification test to fail. For example, an error in the fourth significant digit of the thermal conductivity caused an order verification test to fail for a computational fluid dynamics code used to solve the Navier–Stokes equations (Roy *et al.*, 2007).

2 Choose numerical algorithm

A discretization scheme, or numerical algorithm, must be chosen. This includes both the general discretization approach (finite difference, finite volume, finite element, etc.) and the specific approaches to spatial quadrature. Discretization of any boundary or initial conditions involving spatial derivatives (e.g., Neumann-type boundary conditions) must

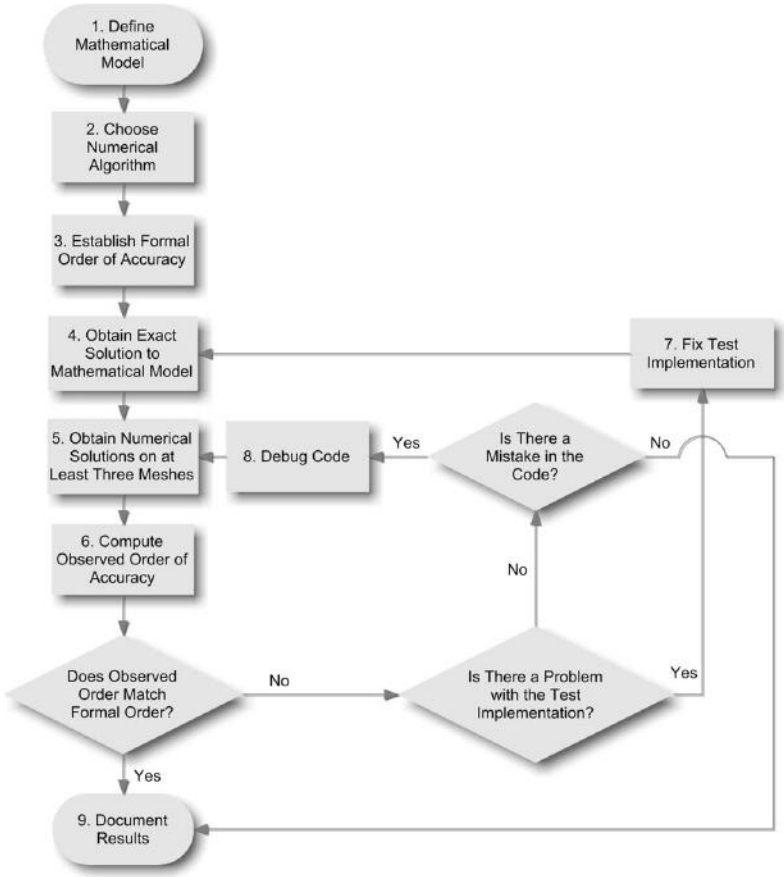

Figure 5.8 Flowchart showing the order verification procedure (adapted from Knupp and Salari, 2003).

also be considered. Note that different iterative solvers can be tested by simply starting the iterations with the final solution values from an iterative solver that has been used in the verification test, thus testing alternative iterative solvers does not require additional code verification tests (Roache, 1998).

3 Establish formal order of accuracy

The formal order of accuracy of the numerical scheme should be established, ideally by an analysis of the truncation error. As discussed in Section 5.3.1, for cases where a truncation error analysis is not available, either the residual method (Section 5.5.6.1), the order verification procedure itself (i.e., the observed order of accuracy), or the expected order of accuracy can be substituted.

4 Obtain exact solution to mathematical model

The exact solution to the mathematical model must be obtained, including both the solution (i.e., the dependent variables) and the system response quantities. New exact solutions are needed any time the governing equations are changed (e.g., when a new model is examined). The same exact solution can be reused if the only changes in the code relate to the numerical scheme (e.g., a different flux function is employed). A key point is that actual numerical values for this exact solution must be computed, which may reduce the utility of series solution as they are no longer exact once the series is truncated. See Chapter 6 for a description of various methods for obtaining exact solutions to mathematical models for scientific computing applications.

5 Obtain numerical solutions on at least four meshes

While only two mesh levels are required to compute the observed order of accuracy when the exact solution to the mathematical model is known, it is strongly recommended that at least four mesh levels be used to demonstrate that the observed order of accuracy is asymptotically approaching the formal order as the mesh discretization parameters (e.g., Δx, Δy, Δz) approach zero. The mesh refinement must be performed in a systematic manner as discussed in Section 5.4. If only one grid topology is to be tested, then the most general type of grid that will be run with the code should be used.

6 Compute observed order of accuracy

With numerical values from both the numerical solution and the exact solution to the mathematical model now available, the discretization error can be evaluated. Global norms of the solution discretization error should be computed as opposed to examining local values. In addition to the error norms for the solution, discretization errors in all system response quantities of interest should also be examined. Recall that the iterative and round-off errors must be small in order to use the numerical solution as a surrogate for the exact solution to the discrete equations when computing the discretization error. For highly-refined spatial meshes and/or small time steps, the discretization error can be small and thus round-off error can adversely impact the order-of-accuracy test. The observed order of accuracy can be computed from Eq. (5.22) for the system response quantities and from Eq. (5.23) for the norms of the discretization error in the solution.

Note that only for the simplest scientific computing cases (e.g., linear elliptic problems) will the observed order of accuracy match the formal order to more than approximately two significant figures during a successful order verification test. For complex scientific computing codes, it is more common that the observed order of accuracy will approach the formal order with increasing mesh refinement. Thus, it is the asymptotic behavior of the observed order of accuracy that is of interest. In addition, observed orders of accuracy that converge to a value higher than the formal order can either indicate the presence of unforeseen error cancellation (which should not be cause for concern) or mistakes in establishing the formal order of accuracy.

If the observed order of accuracy does not match the formal order in an asymptotic sense, then one should first troubleshoot the test implementation (see Step 7) and then debug the code (Step 8) if necessary. If the observed order does match the formal order, then the verification test is considered successful, and one should proceed to Step 9 to document the test results.

7 Fix test implementation

When the observed order of accuracy does not match the formal order of accuracy, the first step is to make sure that the test was implemented correctly. Common examples of incorrect test implementations include mistakes in constructing or evaluating the exact solution to the mathematical model and mistakes made during the comparison between the numerical and exact solutions. If the test implementation is found to be flawed, then the test should be fixed and then repeated. If the test was correctly implemented, then a coding mistake or algorithm inconsistency is likely present and one should proceed to Step 8 to debug the code.

8 Debug the code

When an order-of-accuracy test fails, it indicates either a mistake in the programming of the discrete algorithm, or worse, an inconsistency in the discrete algorithm itself. See Section 5.5.4 for a discussion of approaches to aid in debugging the scientific computing code.

9 Document results

All code verification results should be documented so that a subsequent user understands the code's verification status and does not duplicate the effort. In addition to documenting the observed order of accuracy, the magnitude of the discretization error should also be reported for both system response quantities and the numerical solution (i.e., the norms of the discretization error). It is further recommended that the meshes and solutions used in the code verification test be added to one of the less-frequently run dynamic software test suites (e.g., a monthly test suite), thus allowing the order-of-accuracy verification test to be repeated on a regular basis. Coarse grid cases, which can typically be executed rapidly, can be added as system-level regression tests to a test suite that is run more frequently, as discussed in Chapter 4.

5.5.2 Temporal discretization

The order verification procedure for temporal problems with no spatial dependency is essentially the same as for spatial problems described above. The only difference is that the time step Δt is refined rather than a spatial mesh, thus mesh quality is not a concern. For unsteady problems, the temporal discretization error can sometimes be quite small.

Therefore the round-off error should be examined carefully to ensure that it does not adversely impact the observed order-of-accuracy computation.

5.5.3 Spatial and temporal discretization

It is more difficult to apply the order verification procedure to problems that involve both spatial and temporal discretization, especially for the case where the spatial order of accuracy is different from the temporal order. In addition, temporal discretization errors can in some cases be much smaller than spatial errors (especially when explicit time marching schemes are used), thus making it more difficult to verify the temporal order of accuracy.

For numerical schemes involving spatial and temporal discretization, it is helpful to rewrite the discretization error expansion given in Eq. (5.19) by separating out the spatial and temporal terms as

$$\varepsilon_{h_x}^{h_t} = g_x h_x^p + g_t h_t^q + O(h_x^{p+1}) + O(h_t^{q+1}), \tag{5.27}$$

where h_x denotes the spatial discretization (i.e., $h_x = \Delta x/\Delta x_{\text{ref}} = \Delta y/\Delta y_{\text{ref}}$, etc.), h_t the temporal discretization (i.e., $h_t = \Delta t/\Delta t_{\text{ref}}$), p is the spatial order of accuracy, and q is the temporal order of accuracy. If adaptive time-stepping algorithms are employed, the adaptive algorithm should be disabled to provide explicit control over the size of the time step. Procedures similar to those described in this section can be used for independent refinement in the spatial coordinates, e.g., by introducing $h_x = \frac{\Delta x}{\Delta x_{\text{ref}}}$ and $h_y = \frac{\Delta y}{\Delta y_{\text{ref}}}$, etc. This section will discuss different spatial and temporal order verification procedures that can be either conducted separately or in a combined manner.

5.5.3.1 Separate order analysis

The simplest approach for performing order verification on a code with both spatial and temporal discretization is to first verify the spatial discretization on a steady-state problem. Once the spatial order of accuracy has been verified, then the temporal order can be investigated separately. The temporal order verification can employ a problem with no spatial discretization such as an unsteady zero-dimensional case or a case with linear spatial variations which can generally be resolved by second-order methods to within round-off error. Alternatively, a problem can be chosen which includes spatial discretization errors, but on a highly-refined spatial mesh (Knupp and Salari, 2003). In the latter approach, the use of a highly-refined spatial mesh is often required to reduce the spatial errors to negligible levels, thus allowing the spatial discretization error term $g_x h_x^p$ term to be neglected in Eq. (5.27) relative to the temporal error term $g_t h_t^q$. In practice, this can be difficult to achieve due to stability constraints (especially for explicit methods) or the expense of computing solutions on the highly-refined spatial meshes. An alternative is to reduce the spatial dimensionality of the problem in order to allow a highly-refined spatial mesh to be used. The drawback to using separate order analysis is that it will not uncover issues related to the interaction between the spatial and temporal discretization.

Table 5.1 *Mesh levels used in the spatial and temporal code verification study of Kamm* et al. *(2003).*

Spatial step, h_x	Temporal step, h_t
Δx	Δt
$\Delta x/r_x$	Δt
$\Delta x/r_x^2$	Δt
Δx	$\Delta t/r_t$
Δx	$\Delta t/r_t^2$
$\Delta x/r_x$	$\Delta t/r_t$
$\Delta x/r_x$	$\Delta t/r_t{}^2$

5.5.3.2 Combined order analysis

A combined spatial and temporal order verification method has been developed by Kamm *et al*. (2003). Their procedure begins with a general formulation of the discretization error, which for a global system response quantity can be written in the form

$$\varepsilon_{h_x}^{h_t} = g_x h_x^{\hat{p}} + g_t h_t^{\hat{q}} + g_{xt} h_x^{\hat{r}} h_t^{\hat{s}}, \tag{5.28}$$

where the observed orders of accuracy $\hat{p}$, $\hat{q}$, $\hat{r}$, and $\hat{s}$ are to be solved for along with the three coefficients g_x, g_t, and g_{xt}. In addition, for the norms of the discretization error, a similar expansion is employed,

$$\left\| \varepsilon_{h_x}^{h_t} \right\| = g_x h_x^{\hat{p}} + g_t h_t^{\hat{q}} + g_{xt} h_x^{\hat{r}} h_t^{\hat{s}}, \tag{5.29}$$

where of course the observed orders and coefficients may be different. In order to solve for these seven unknowns, seven independent levels of spatial and/or temporal refinement are required. Beginning with an initial mesh with Δx and Δt Kamm *et al*. (2003) alternately refined in space (by r_x) and time (by r_t) to obtain the seven mesh levels shown in Table 5.1. By computing these seven different numerical solutions, the discretization error expressions for all seven mesh levels result in a coupled, nonlinear set of algebraic equations. The authors solved this nonlinear system of equations using a Newton-type iterative procedure. An advantage of this approach is that it does not require that the three terms in the discretization error expression be the same order of magnitude. The main drawback is the computational expense since a single calculation of the observed orders of accuracy requires seven different numerical solutions. Additional solutions should also be computed to ensure the asymptotic behavior of the observed order of accuracy has been achieved.

Kamm *et al*. (2003) included the mixed spatial/temporal term because their application employed the explicit Lax–Wendroff temporal integration scheme combined with a Godunov-type spatial discretization, which has formal orders of accuracy of $p = 2$, $q = 2$,

Table 5.2 *Mesh levels recommended for the simpler error expansion of Eqs. (5.30) and (5.31) which omit the mixed spatial/temporal term.*

Spatial step, h_x	Temporal step, h_t
Δx	Δt
$\Delta x/r_x$	Δt
Δx	$\Delta t/r_t$
$\Delta x/r_x$	$\Delta t/r_t$

and $r = s = 1$ (i.e., it is formally second-order accurate). For many spatial/temporal discretization approaches, the mixed spatial/temporal term can be omitted because it does not appear in the truncation error, thereby reducing the unknowns and the required number of independent mesh levels down to four. The resulting error expansion for global system response quantities becomes

$$\varepsilon_{h_x}^{h_t} = g_x h_x^{\hat{p}} + g_t h_t^{\hat{q}}, \tag{5.30}$$

while the expansion for discretization error norms becomes

$$\left\| \varepsilon_{h_x}^{h_t} \right\| = g_x h_x^{\hat{p}} + g_t h_t^{\hat{q}}. \tag{5.31}$$

Although not unique, recommended mesh levels to employ for the simpler error expansions given by Eqs. (5.30) and (5.31) are given in Table 5.2. The resulting four nonlinear algebraic equations have no closed form solution and thus must be solved numerically (e.g., using Newton's method) for the orders of accuracy $\hat{p}$ and $\hat{q}$ and the coefficients g_x and g_t.

An alternative based on the discretization error expansions of Eqs. (5.30) and (5.31) that does not require the solution to a system of nonlinear algebraic equations can be summarized briefly as follows. First, a spatial mesh refinement study using three meshes is performed with a fixed time step to obtain $\hat{p}$ and g_x. Then a temporal refinement study is performed using three different time steps to obtain $\hat{q}$ and g_t. Once these four unknowns have been estimated, the spatial step size h_x and the temporal step size h_t can be chosen such that the spatial discretization error term ($g_x h_x^p$) has the same order of magnitude as the temporal error term ($g_t h_t^q$). Once these two terms are approximately the same order of magnitude, a combined spatial and temporal order verification is conducted by choosing the temporal refinement factor such that the temporal error term drops by the same factor as the spatial term with refinement. This procedure is explained in detail below.

In order to estimate $\hat{p}$ and g_x, a spatial mesh refinement study is performed with a fixed time step. Note that this will introduce a fixed temporal discretization error (i.e., $g_t h_t^q$) for all computations, thus the standard observed order-of-accuracy relationship from Eq. (5.22)

cannot be used. Considering only the discretization error norms for now (the same analysis will also apply to the discretization error of the system response quantities), Eq. (5.31) can be rewritten as

$$\left\| \varepsilon_{h_x}^{h_t} \right\| = \phi + g_x h_x^{\hat{p}}, \tag{5.32}$$

where $\phi = g_t h_t^{\hat{q}}$ is the fixed temporal error term. For this case, the observed order-of-accuracy expression from Chapter 8 given by Eq. (8.70) can be used, which requires three mesh solutions, e.g., coarse ($r_x^2 h_x$), medium ($r_x h_x$), and fine (h_x):

$$\hat{p} = \frac{\ln\left(\frac{\left\| \varepsilon_{r_x^2 h_x}^{h_t} \right\| - \left\| \varepsilon_{r_x h_x}^{h_t} \right\|}{\left\| \varepsilon_{r_x h_x}^{h_t} \right\| - \left\| \varepsilon_{h_x}^{h_t} \right\|} \right)}{\ln(r_x)} \tag{5.33}$$

By calculating $\hat{p}$ in this manner, the constant temporal error term ϕ will cancel out. The spatial discretization error coefficient g_x can then be found from

$$g_x = \frac{\left\| \varepsilon_{r_x h_x}^{h_t} \right\| - \left\| \varepsilon_{h_x}^{h_t} \right\|}{h_x^{\hat{p}} \left(r_x^{\hat{p}} - 1 \right)}. \tag{5.34}$$

A similar analysis is then performed by refining the time step with a fixed spatial mesh to obtain $\hat{q}$ and g_t. Once these orders of accuracy and coefficients have been estimated, then the leading spatial and temporal discretization error terms can be adjusted to approximately the same order of magnitude by coarsening or refining in time and/or space (subject to numerical stability restrictions). If the discretization error terms are of drastically different magnitude, then extremely refined meshes and/or time steps may be needed to detect coding mistakes.

At this point, if the formal spatial and temporal orders of accuracy are the same (i.e., if $p = q$), then the standard order verification procedure with only two mesh levels can be applied using Eqs. (5.22) or (5.23) since $r_x = r_t$. For the more complicated case when $p \neq q$, temporal refinement can be conducted by choosing the temporal refinement factor r_t according to Eq. (5.35) following Richards (1997):

$$r_t = (r_x)^{p/q}. \tag{5.35}$$

This choice for r_t will ensure that the spatial and temporal discretization error terms are reduced by the same factor with refinement when the solutions are in the asymptotic range. Some recommended values of r_t for $r_x = 2$ are given in Table 5.3 for various formal spatial and temporal orders of accuracy. The observed orders of accuracy in space and time are then computed using two mesh levels according to

$$\hat{p} = \frac{\ln\left(\frac{\left\| \varepsilon_{r_x h_t}^{r_t h_t} \right\|}{\left\| \varepsilon_{h_x}^{h_t} \right\|} \right)}{\ln(r_x)} \quad \text{and} \quad \hat{q} = \frac{\ln\left(\frac{\left\| \varepsilon_{r_x h_t}^{r_t h_t} \right\|}{\left\| \varepsilon_{h_x}^{h_t} \right\|} \right)}{\ln(r_t)}. \tag{5.36}$$

Table 5.3 *Temporal refinement factors required to conduct combined spatial and temporal order verification using only two numerical solutions.*

Spatial order, p	Temporal order, q	Spatial refinement factor, r_x	Temporal refinement factor, r_t	Expected error reduction ratio (coarse/fine)
1	1	2	2	2
1	2	2	$\sqrt{2}$	2
1	3	2	$\sqrt[3]{2}$	2
1	4	2	$\sqrt[4]{2}$	2
2	1	2	4	4
2	2	2	2	4
2	3	2	$\sqrt[3]{4}$	4
2	4	2	$\sqrt[4]{4}$	4
3	1	2	8	8
3	2	2	$\sqrt{8}$	8
3	3	2	2	8
3	4	2	$\sqrt[4]{8}$	8

The analysis using system response quantities from Eq. (5.30) is exactly the same as given above in Eq. (5.36), only with the norm notation omitted.

5.5.4 Recommendations for debugging

Order verification provides a highly-sensitive test as to whether there are mistakes in the computer code and/or inconsistencies in the discrete algorithm. In addition, aspects of the order verification procedure can be extremely useful for tracking down the mistakes (i.e., debugging the code) once they have been found to exist. Once an order verification test fails, and assuming the test was properly implemented, the local variation of discretization error in the domain should be examined. Accumulation of error near a boundary or a corner cell generally indicates that the errors are in the boundary conditions. Errors in regions of mild grid clustering or skewness can indicate mistakes in the mesh transformations or spatial quadratures. Mesh quality problems could be due to the use of a poor quality mesh, or worse, due to a discrete algorithm that is overly-sensitive to mesh irregularities. The effects of mesh quality on the discretization error are examined in detail in Chapter 9.

5.5.5 Limitations of order verification

A significant limitation of the order verification process is that the formal order of accuracy can change due to the level of smoothness of the solution, as discussed in Section 5.3.1. In

addition, since order verification can only detect problems in the solution itself, it generally cannot be used to detect coding mistakes affecting the efficiency of the code. For example, a mistake that causes an iterative scheme to converge in 500 iterations when it should converge in ten iterations will not be detected by order verification. Note that this type of mistake would be found with an appropriate component-level test fixture for the iterative scheme (as discussed in Chapter 4). Similarly, mistakes which affect the robustness of the numerical algorithm will not be detected, since these mistakes also do not affect the final numerical solution.

Finally, the standard order verification procedure does not verify individual terms in the discretization. Thus a numerical scheme which is formally first-order accurate for convection and second-order accurate for diffusion results in a numerical scheme with first-order accuracy. Mistakes reducing the order of accuracy of the diffusion term to first order would therefore not be detected. This limitation can be addressed by selectively turning terms on and off, which is most easily accomplished when the method of manufactured solutions is employed (see Chapter 6).

5.5.6 Alternative approaches for order verification

In recent years, a number of variants of the order verification procedure have been proposed. Most of these alternative approaches were developed in order to avoid the high cost of generating and computing numerical solutions on highly-refined 3-D meshes. All of the approaches discussed here do indeed reduce the cost of conducting the order verification test relative to the standard approach of systematic mesh refinement; however, each approach is also accompanied by drawbacks which are also discussed. These alternative approaches are presented in order of increasing reliability, with a summary of the strengths and weaknesses of each approach presented at the end of this section.

5.5.6.1 Residual method

In general, when the discrete operator $L_h(\cdot)$ operates on anything but the exact solution to the discrete equations u_h, the nonzero result is referred to as the *discrete residual*, or simply the *residual*. This discrete residual is not to be confused with the iterative residual which is found by inserting an approximate iterative solution into the discrete equations (see Chapter 7). Recall that the truncation error can be evaluated by inserting the exact solution to the mathematical model $\tilde{u}$ into the discrete operator as given previously in Eq. (5.13). Assuming an exact solution to the mathematical model is available, the truncation error (i.e., the discrete residual) can be evaluated directly on a given mesh without actually solving for the numerical solution on this grid. Since no iterations are needed, this truncation error evaluation is usually very inexpensive. The truncation error found by inserting the exact solution to the mathematical model into the discrete operator can be evaluated on a series of systematically-refined meshes. The observed order of accuracy is then computed by Eq. (5.23), but using norms of the truncation error rather than the discretization error.

There are a number of drawbacks to the residual form of order verification that are related to the fact that the residual does not incorporate all aspects of the code (Ober, 2004). Specifically, the residual method does not test:

- boundary conditions that do not contribute to the residual (e.g., Dirichlet boundary conditions),
- system response quantities such as lift, drag, combustion efficiency, maximum heat flux, maximum stress, oscillation frequency, etc.,
- numerical algorithms where the governing equations are solved in a segregated or decoupled manner (e.g., the SIMPLE algorithm (Patankar, 1980) for incompressible fluids problems where the momentum equations are solved, followed by a pressure projection step to satisfy the mass conservation equation), and
- explicit multi-stage schemes such as Runge-Kutta.

In addition, it has been observed that the truncation error can converge at a lower rate than the discretization error for finite-volume schemes on certain unstructured mesh topologies (Despres, 2004; Thomas *et al*., 2008). In these cases, it is possible that the residual method might exhibit a lower order of accuracy than a traditional order-of-accuracy test applied to the discretization error. For examples of the residual method applied for code verification see Burg and Murali (2004) and Thomas *et al*. (2008).

5.5.6.2 Statistical method

The *statistical form of order verification* was proposed by Hebert and Luke (2005) and uses only a single mesh, which is successively scaled down and used to sample over the chosen domain. The sampling is performed randomly, and norms of the volume-weighted discretization error are examined. The main advantage of the statistical method is that it is relatively inexpensive because it does not require refinement of the mesh. There are, however, a number of disadvantages. First, since the domain is sampled statistically, convergence of the statistical method must be ensured. Second, it tends to weight the boundary points more heavily as the grids are shrunk relative to traditional order verification since the ratio of boundary points to interior points is fixed rather than reducing with mesh refinement. Finally, this approach assumes that the discretization errors are independent random variables, thus neglecting the transported component of error into the refined domains. (See Chapter 8 for a discussion of the difference between transported and locally-generated components of the discretization error.) Due to these issues, it is possible to pass a statistical order verification test for a case that would fail a traditional order verification test based on systematic mesh refinement.

5.5.6.3 Downscaling method

The *downscaling approach* to order verification (Diskin and Thomas, 2007; Thomas *et al*., 2008) shares a number of attributes with the statistical method described above. The major difference is that instead of statistically sampling the smaller meshes in the domain, the mesh is scaled down about a single point in the domain, which eliminates the statistical convergence issues associated with statistical order verification. The focal point to which

Table 5.4 *Comparison of different order verification approaches showing the cost and the type of order-of-accuracy estimate produced (adapted from Thomas* et al.*, 2008).*

Verification method	Cost	Type of order estimate
Standard order verification with systematic mesh refinement	High	Precise order of accuracy
Downscaling method	Moderate to low	Admits false positives
Statistical method	Moderate (due to sampling)	Admits false positives
Residual method	Very low	Admits false positives and false negatives

the grids are scaled can be chosen to emphasize the internal discretization, the boundary discretization, or singularities (Thomas *et al.*, 2008). A major benefit of the downscaling method is that it allows for boundary condition verification in the presence of complex geometries. The mesh scaling can be performed in a very simple manner when examining the interior discretization or straight boundaries, but must be modified to ensure the proper scaling of a mesh around a curved boundary. The downscaling method also neglects the possibility of discretization error transport into the scaled-down domain, and thus can provide overly optimistic estimates of the actual convergence rate.

5.5.6.4 Summary of order verification approaches

In order to summarize the characteristics of the different order verification approaches, it is first helpful to categorize the results of an order verification test. Here we will define a positive result as one in which the observed order of accuracy is found to match the formal order in an asymptotic sense, while a negative result is one where the observed order is less than the formal order. We now define a false positive as a case where a less-rigorous order verification test achieves a positive result, but the more rigorous order verification procedure with systematic mesh refinement produces a negative result. Similarly, a false negative occurs when the test result is negative, but standard order verification with systematic mesh refinement is positive. The characteristics of the four approaches for order-of-accuracy verification are given in Table 5.4. As one might expect, the cost of conducting and order verification study varies proportionately with the reliability of the observed order of accuracy estimate.

5.6 Responsibility for code verification

The ultimate responsibility for ensuring that rigorous code verification has been performed lies with the user of the scientific computing code. This holds true whether the code was developed by the user, by someone else in the user's organization, or by an independent

organization (government laboratory, commercial software company, etc.). It is not sufficient for the code user to simply assume that code verification studies have been successfully performed.

In the ideal case, code verification should be performed during, and as an integrated part of, the software development process. While code verification studies are most often conducted by the code developers, a higher level of independence can be achieved when they are conducted by a separate group, by the code customer, or even by an independent regulatory agency (recall the discussion of independence of the verification and validation process in Chapter 2). While there are often fewer coding mistakes to find in a scientific computing code that has a prior usage history, performing code verification studies for a mature code can be expensive and challenging if the code was not designed with code verification testing in mind.

Commercial companies that produce scientific computing software rarely perform rigorous code verification studies, or if they do, they do not make the results public. Most code verification efforts that are documented for commercial codes seem to be limited to simple benchmark examples that demonstrate "engineering accuracy" rather than verifying the order of accuracy of the code (Oberkampf and Trucano, 2008). Recently, Abanto *et al*. (2005) performed order verification studies on three different commercial computational fluid dynamics codes which were formally at least second-order accurate. Most tests resulted in either first-order accuracy or nonconvergent behavior with mesh refinement. It is our opinion that code users should be aware that commercial software companies are unlikely to perform rigorous code verification studies unless users request it.

In the absence of rigorous, documented code verification evidence, there are code verification activities that can be performed by the user. In addition to the simple code verification activities discussed in Section 5.1, order verification tests can also be conducted when an exact solution (or a verifiably accurate surrogate solution) to the mathematical model is available. While the traditional approach for finding exact solutions can be used, the more general method of manufactured solutions procedure requires the ability to employ user-defined boundary conditions, initial conditions, and source terms, and thus can be difficult to implement for a user who does not have access to source code. The next chapter focuses on different approaches for obtaining exact solutions to the mathematical model including the method of manufactured solutions.

5.7 References

Abanto, J., D. Pelletier, A. Garon, J-Y. Trepanier, and M. Reggio (2005). *Verication of some Commercial CFD Codes on Atypical CFD Problems*, AIAA Paper 2005–682.

Banks, J. W., T. Aslam, and W. J. Rider (2008). On sub-linear convergence for linearly degenerate waves in capturing schemes, *Journal of Computational Physics*. **227**, 6985–7002.

Burg, C. and V. Murali (2004). *Efficient Code Verification Using the Residual Formulation of the Method of Manufactured Solutions*, AIAA Paper 2004–2628.

Carpenter, M. H. and J. H. Casper (1999). Accuracy of shock capturing in two spatial dimensions, *AIAA Journal*. **37**(9), 1072–1079.
Crank, J. and P. A. Nicolson (1947). Practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type, *Proceedings of the Cambridge Philosophical Society*. **43**, 50–67.
Despres, B. (2004). Lax theorem and finite volume schemes, *Mathematics of Computation*. **73**(247), 1203–1234.
Diskin, B. and J. L. Thomas (2007). *Accuracy Analysis for Mixed-Element Finite Volume Discretization Schemes*, Technical Report TR 2007–8, Hampton, VA, National Institute of Aerospace.
Engquist, B. and B. Sjogreen (1998). The convergence rate of finite difference schemes in the presence of shocks, *SIAM Journal of Numerical Analysis*. **35**(6), 2464–2485.
Ferziger, J. H. and M. Peric (1996). Further discussion of numerical errors in CFD, *International Journal for Numerical Methods in Fluids*. **23**(12), 1263–1274.
Ferziger, J. H. and M. Peric (2002). *Computational Methods for Fluid Dynamics*, 3rd edn., Berlin, Springer-Verlag.
Grinstein, F. F., L. G. Margolin, and W. J. Rider (2007). *Implicit Large Eddy Simulation: Computing Turbulent Fluid Dynamics*, Cambridge, UK, Cambridge University Press.
Hebert, S. and E. A. Luke (2005). *Honey, I Shrunk the Grids! A New Approach to CFD Verification Studies*, AIAA Paper 2005–685.
Hirsch, C. (2007). *Numerical Computation of Internal and External Flows (Vol. 1)*, 2nd edn., Burlington, MA, Elsevier.
Kamm, J. R., W. J. Rider, and J. S. Brock (2003). *Combined Space and Time Convergence Analyses of a Compressible Flow Algorithm*, AIAA Paper 2003–4241.
Knupp, P. M. (2003). Algebraic mesh quality metrics for unstructured initial meshes, *Finite Elements in Analysis and Design*. **39**(3), 217–241.
Knupp, P. M. (2009). Private communication, March 9, 2009.
Knupp, P. M. and K. Salari (2003). *Verification of Computer Codes in Computational Science and Engineering*, K. H. Rosen (ed.), Boca Raton, FL, Chapman and Hall/CRC.
Knupp, P., C. Ober, and R. Bond (2007). Measuring progress in order-verification within software development projects, *Engineering with Computers*. **23**, 271–282.
Mastin, C. W. (1999). Truncation Error on Structured Grids, in *Handbook of Grid Generation*, J. F. Thompson, B. K. Soni, and N. P. Weatherill, (eds.), Boca Raton, CRC Press.
Ober, C. C. (2004). Private communication, August 19, 2004.
Oberkampf, W. L. and T. G. Trucano (2008). Verification and validation benchmarks, *Nuclear Engineering and Design*. **238**(3), 716–743.
Panton, R. L. (2005). *Incompressible Flow*, 3rd edn., Hoboken, NJ, John Wiley and Sons.
Patankar, S. V. (1980). *Numerical Heat Transfer and Fluid Flow*, New York, Hemisphere Publishing Corp.
Potter, D. L., F. G. Blottner, A. R. Black, C. J. Roy, and B. L. Bainbridge (2005). *Visualization of Instrumental Verification Information Details (VIVID): Code Development, Description, and Usage*, SAND2005–1485, Albuquerque, NM, Sandia National Laboratories.
Richards, S. A. (1997). Completed Richardson extrapolation in space and time, *Communications in Numerical Methods in Engineering*. **13**, 573–582.
Richtmyer, R. D. and K. W. Morton (1967). *Difference Methods for Initial-value Problems*, 2nd edn., New York, John Wiley and Sons.

Rider, W. J. (2009). Private communication, March 27, 2009.
Roache, P. J. (1998). *Verification and Validation in Computational Science and Engineering*, Albuquerque, NM, Hermosa Publishers.
Roy, C. J. (2003). Grid convergence error analysis for mixed-order numerical schemes, *AIAA Journal*. **41**(4), 595–604.
Roy, C. J. (2005). Review of code and solution verification procedures for computational simulation, *Journal of Computational Physics*. **205**(1), 131–156.
Roy, C. J. (2009). *Strategies for Driving Mesh Adaptation in CFD*, AIAA Paper 2009–1302.
Roy, C. J., E. Tendean, S. P. Veluri, R. Rifki, E. A. Luke, and S. Hebert (2007). *Verification of RANS Turbulence Models in Loci-CHEM using the Method of Manufactured Solutions*, AIAA Paper 2007–4203.
Tannehill, J. C., D. A. Anderson, and R. H. Pletcher (1997). *Computational Fluid Mechanics and Heat Transfer*, 2nd edn., Philadelphia, PA, Taylor and Francis.
Thomas, J. L., B. Diskin, and C. L. Rumsey (2008). Toward verification of unstructured-grid solvers, *AIAA Journal*. **46**(12), 3070–3079.
Thompson, J. F., Z. U. A. Warsi, and C. W. Mastin (1985). *Numerical Grid Generation: Foundations and Applications*, New York, Elsevier. (www.erc.msstate.edu/publications/gridbook).
Trucano, T. G., M. M. Pilch, and W. L. Oberkampf (2003). *On the Role of Code Comparisons in Verification and Validation*, SAND 2003–2752, Albuquerque, NM, Sandia National Laboratories.
Veluri, S., C. J. Roy, S. Hebert, and E. A. Luke (2008). *Verification of the Loci-CHEM CFD Code Using the Method of Manufactured Solutions*, AIAA Paper 2008–661.

6

Exact solutions

The primary focus of this chapter is on the use of exact solutions to mathematical models for code verification. Recall that, in some cases, software testing can be performed by simply running the code and comparing the results to the correct code output. However, in scientific computing, "correct" code output depends on the chosen spatial mesh, time step, iterative convergence tolerance, machine precision, etc. We are thus forced to rely on other less definitive methods for assessing code correctness. In Chapter 5, the order of accuracy test was argued to be the most rigorous approach for code verification. When the order of accuracy test fails, or when the formal order of accuracy has not been determined, then the less rigorous convergence test may be used. In either case, an exact solution to the underlying mathematical model is required. When used for code verification, the ability of this exact solution to exercise all terms in the mathematical model is more important than any physical realism of the solution. In fact, realistic exact solutions are often avoided for code verification due to the presence of singularities and/or discontinuities. Numerous examples will be given in this chapter of exact solutions and their use with the order verification test. The final example given in this chapter employs the less rigorous convergence test with benchmark numerical solutions. In addition to code verification applications, exact solutions to mathematical models are extremely valuable for evaluating the accuracy of numerical schemes, determining solution sensitivity to mesh quality and topology, evaluating the reliability of discretization error estimators, and evaluating solution adaptation schemes. For these secondary applications, physically realistic exact solutions are preferred (see Section 6.4).

This chapter discusses methods for obtaining exact solutions to mathematical models used in scientific computing. These mathematical models typically take the form of either integral or differential equations. Because we will have to compute actual numerical values for these exact solutions, we will use a different definition for an exact solution than found in standard mathematics textbooks. The definition used here for an exact solution is a solution to the mathematical model that is in closed form, i.e., in terms of elementary functions (trigonometric functions, exponentials, logarithms, powers, etc.) or readily-computed special functions (e.g., gamma, beta, error, Bessel functions) of the independent variables. Solutions involving infinite series or a reduction of a partial differential equation to a system of ordinary differential equations which do not have exact solutions will be considered as approximate solutions and are addressed at the end of the chapter.

In scientific computing, the mathematical models can take different forms. When a set of physical laws such as conservation of mass, momentum, or energy (i.e., the conceptual model) are formulated for an infinitesimally small region of space, then the resulting mathematical model generally takes the form of differential equations. When applied to a region of space with finite size, the mathematical model takes the form of integral (or integro-differential) equations. The differential form is called the *strong form* of the equations, whereas the integral form, after application of the divergence theorem (which converts the volume integral of the gradient of a quantity to fluxes through the boundary), is called the *weak form*. The finite difference method employs the strong form of the equations, while the finite element and finite volume methods employ the weak form.

The strong form explicitly requires solutions that are differentiable, whereas the weak form admits solutions that can contain discontinuities while still satisfying the underlying physical laws, and these discontinuous solutions are called *weak solutions*. Weak solutions satisfy the differential equation only in a restricted sense. While exact solutions to the weak form of the equations exist that do contain discontinuities (e.g., the Riemann or shock tube problem in gas dynamics), the more general approaches for generating exact solutions such as the method of manufactured solutions discussed in Section 6.3 have not to our knowledge encompassed nondifferentiable weak solutions (although this extension is needed). This chapter will assume the strong (i.e., differential) form of the mathematical models unless otherwise noted. Since strong solutions also satisfy the weak form of the equations, the finite element and finite volume methods will not be excluded by this assumption, and we are simply restricting ourselves to smooth solutions.

Because scientific computing often involves complex systems of coupled partial differential equations (PDEs) which have relatively few exact solutions, the organization of this chapter is very different from that of standard mathematics texts. After a short introduction to differential equations in Section 6.1, a discussion of "traditional" exact solutions and solution methods is presented in Section 6.2. The method of manufactured solutions (MMS) is a more general approach for obtaining exact solutions to complicated mathematical models and is discussed in detail in Section 6.3. When physically realistic manufactured solutions are desired, the approaches discussed in Section 6.4 can be used. As discussed above, solutions involving infinite series, reduction of PDEs to ordinary differential equations, or numerical solutions of the underlying mathematical model with established numerical accuracy are relegated to Section 6.5.

6.1 Introduction to differential equations

Differential equations are ubiquitous in the study of physical processes in science and engineering (O'Neil, 2003). A differential equation is a relation between a variable and its derivatives. When only one independent variable is present, the equation is called an *ordinary differential equation*. A differential equation involving derivatives with respect to two or more independent variables (e.g., x and t, or x, y, and z) is called a *partial differential equation* (PDE). A differential equation can be a single equation with a single

dependent variable or a system of equations with multiple dependent variables. The *order* of a differential equation is the largest number of derivatives applied to any dependent variable. The *degree* of a differential equation is the highest power of the highest derivative found in the equation.

A differential equation is considered to be *linear* when the dependent variables and all of their derivatives occur with powers less than or equal to one and there are no products involving derivatives and/or functions of the dependent variables. Solutions to linear differential equations can be combined to form new solutions using the linear superposition principle. A *quasi-linear* differential equation is one that is linear in the highest derivative, i.e., the highest derivative appears to the power one.

General solutions to PDEs are solutions that that satisfy the PDE but involve arbitrary constants and/or functions and thus are not unique. To find *particular solutions* to PDEs, additional conditions must be supplied on the boundary of the domain of interest, i.e., *boundary conditions*, at an initial data location i.e., *initial conditions*, or some combination of the two. Boundary conditions generally come in the form of *Dirichlet* boundary conditions which specify values of the dependent variables or *Neumann* boundary conditions which specify the values of derivatives of the dependent variables normal to the boundary. When both Dirichlet and Neumann conditions are applied at a boundary it is called a *Cauchy* boundary condition, whereas a linear combination of a dependent variable and its normal derivative is called a *Robin* boundary condition. The latter is often confused with *mixed* boundary conditions, which occur when different boundary condition types (Dirichlet, Neumann, or Robin) are applied at different boundaries in a given problem.

Another source of confusion is related to the order of the boundary conditions. The maximum order of the boundary conditions is at least one less than the order of the differential equation. Here *order* refers to the highest number of derivatives applied to any dependent variable as discussed above. This requirement on the order of the boundary condition is sometimes erroneously stated as a requirement on the order of accuracy of the discretization of a derivative boundary condition when the PDE is solved numerically. On the contrary, a reduction in the formal order of accuracy of a discretized boundary condition often leads to a reduction in the observed order of accuracy of the entire solution.

Partial differential equations can be classified as *elliptic*, *parabolic*, *hyperbolic*, or a combination of these types. For scalar equations, this classification is fairly straightforward in a manner analogous to determining the character of algebraic equations. For systems of PDEs written in quasi-linear form, the eigenvalues of the coefficient matrices can be used to determine the mathematical character (Hirsch, 2007).

6.2 Traditional exact solutions

The standard approach to obtaining an exact solution to a mathematical model can be summarized as follows. Given the governing partial differential (or integral) equations on some domain with appropriately specified initial and/or boundary conditions, find the exact solution. The main disadvantage of this approach is that there are only a limited number

of exact solutions known for complex equations. Here the complexity of the equations could be due to geometry, nonlinearity, physical models, and/or coupling between multiple physical phenomena such as fluid–structure interaction.

When exact solutions are found for complex equations, they often depend on significant simplifications in dimensionality, geometry, physics, etc. For example, the flow between infinite parallel plates separated by a small gap with one plate moving at a constant speed is called Couette flow and is described by the Navier–Stokes equations, a nonlinear, second-order system of PDEs. In Couette flow, the velocity profile is linear across the gap, and this linearity causes the diffusion term, a second derivative of velocity, to be identically zero. In addition, there are no solution variations in the direction that the plate is moving. Thus the exact solution to Couette flow does not exercise many of the terms in the Navier–Stokes equations.

There are many books available that catalogue a vast number of exact solutions for differential equations found in science and engineering. These texts address ordinary differential equations (e.g., Polyanin and Zaitsev, 2003), linear PDEs (Kevorkian, 2000; Polyanin, 2002; Meleshko, 2005), and nonlinear PDEs (Kevorkian, 2000; Polyanin and Zaitsev, 2004; Meleshko, 2005). In addition, many exact solutions can be found in discipline-specific references such as those for heat conduction (Carslaw and Jaeger, 1959), fluid dynamics (Panton, 2005; White, 2006), linear elasticity (Timoshenko and Goodier, 1969; Slaughter, 2002), elastodynamics (Kausel, 2006), and vibration and buckling (Elishakoff, 2004). The general Riemann problem involves an exact weak (i.e., discontinuous) solution to the 1-D unsteady inviscid equations for gas dynamics (Gottlieb and Groth, 1988).

6.2.1 Procedures

In contrast to the method of manufactured solutions discussed in Section 6.3, the traditional method for finding exact solutions solves the forward problem: given a partial differential equation, a domain, and boundary and/or initial conditions, find the exact solution. In this section, we present some of the simpler classical methods for obtaining exact solutions and make a brief mention of more advanced (nonclassical) techniques. Further details on the classical techniques for PDEs can be found in Ames (1965) and Kevorkian (2000).

6.2.1.1 Separation of variables

Separation of variables is the most common approach for solving linear PDEs, although it can also be used to solve certain nonlinear PDEs. Consider a scalar PDE with dependent variable u and independent variables t and x. There are two forms for separation of variables, multiplicative and additive, and these approaches can be summarized as:

$$\text{multiplicative: } u(t,x) = \phi(t)\psi(x),$$

$$\text{additive: } u(t,x) = \phi(t) + \psi(x).$$

The multiplicative form of separation of variables is the most common.

For an example of separation of variables, consider the 1-D unsteady heat equation with constant thermal diffusivity α,

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}, \tag{6.1}$$

where $T(t, x)$ is the temperature. Let us first simplify this equation by employing the simple transformations $t = \alpha \bar{t}$ and $x = \alpha \bar{x}$. With these transformations, the heat equation can be rewritten in simpler form as:

$$\frac{\partial T}{\partial \bar{t}} = \frac{\partial^2 T}{\partial \bar{x}^2}. \tag{6.2}$$

Using the multiplicative form of separation of variables $T(\bar{x}, \bar{t}) = \phi(\bar{t})\psi(\bar{x})$, the differential equation can be rewritten as

$$\frac{\phi_t(\bar{t})}{\phi(\bar{t})} = \frac{\psi_{xx}(\bar{x})}{\psi(\bar{x})}, \tag{6.3}$$

where the subscript denotes differentiation with respect to the subscripted variable. Since the left hand side of Eq. (6.3) is independent of $\bar{x}$ and the right hand side is independent of $\bar{t}$, both sides must be equal to a constant a, i.e.,

$$\frac{\phi_t(\bar{t})}{\phi(\bar{t})} = \frac{\psi_{xx}(\bar{x})}{\psi(\bar{x})} = a. \tag{6.4}$$

Each side of Eq. (6.3) can thus be written as

$$\begin{aligned} \frac{\mathrm{d}\phi}{\mathrm{d}\bar{t}} - a\phi &= 0, \\ \frac{\mathrm{d}^2\psi}{d\bar{x}^2} - a\psi &= 0. \end{aligned} \tag{6.5}$$

Equations (6.5) can be integrated using standard methods for ordinary differential equations. After substituting back in for x and t, we finally arrive at two general solutions (Meleshko, 2005) depending on the sign of a:

$$\begin{aligned} a = \lambda^2 &: u(t, x) = \exp\left(\frac{\lambda^2 t}{\alpha}\right)\left[c_1 \exp\left(\frac{-\lambda x}{\alpha}\right) + c_2 \exp\left(\frac{\lambda x}{\alpha}\right)\right], \\ a = -\lambda^2 &: u(t, x) = \exp\left(\frac{-\lambda^2 t}{\alpha}\right)\left[c_1 \sin\left(\frac{\lambda x}{\alpha}\right) + c_2 \cos\left(\frac{\lambda x}{\alpha}\right)\right], \end{aligned} \tag{6.6}$$

where c_1, c_2, and λ are constants that can be determined from the initial and boundary conditions.

6.2.1.2 Transformations

Transformations can sometimes be used to convert a differential equation into a simpler form that has a known solution. Transformations that do not involve derivatives are called point transformations (Polyanin and Zaitsev, 2004), while transformations that involve

derivatives are called tangent transformations (Meleshko, 2005). An example of a point transformation is the hodograph transformation, which exchanges the roles between the independent and the dependent variables. Examples of tangent transformations include Legendre, Hopf–Cole, and Laplace transformations.

A well-known example of a tangent transformation is the Hopf–Cole transformation (Polyanin and Zaitsev, 2004). Consider the nonlinear Burgers' equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \tag{6.7}$$

where the viscosity ν is assumed to be constant. Burgers' equation serves as a scalar model equation for the Navier–Stokes equations since it includes an unsteady term $\left(\frac{\partial u}{\partial t}\right)$, a nonlinear convection term $\left(u\frac{\partial u}{\partial x}\right)$, and a diffusion term $\left(\nu\frac{\partial^2 u}{\partial x^2}\right)$. The Hopf–Cole transformation is given by

$$u = \frac{-2\nu}{\phi}\phi_x, \tag{6.8}$$

where again ϕ_x denotes partial differentiation of ϕ with respect to x. Substituting the Hopf–Cole transformation into Burgers' equation, applying the product rule, and simplifying results in

$$\frac{\phi_{tx}}{\phi} - \frac{\phi_t\phi_x}{\phi^2} - \nu\left(\frac{\phi_{xxx}}{\phi} - \frac{\phi_{xx}\phi_x}{\phi^2}\right) = 0, \tag{6.9}$$

which can be rewritten as

$$\frac{\partial}{\partial x}\left[\frac{1}{\phi}\left(\frac{\partial\phi}{\partial t} - \nu\frac{\partial^2\phi}{\partial x^2}\right)\right] = 0. \tag{6.10}$$

The terms in parenthesis in Eq. (6.10) are simply the 1-D unsteady heat equation (6.1) written with $\phi(t, x)$ as the dependent variable. Thus any nonzero solution to the heat equation $\phi(t, x)$ can be transformed into a solution to Burgers' equation (6.7) using the Hopf–Cole transformation given by Eq. (6.8).

6.2.1.3 Method of characteristics

An approach for finding exact solutions to hyperbolic PDEs is the *method of characteristics*. The goal is to identify characteristic curves/surfaces along which certain solution properties will be constant. Along these characteristics, the PDE can be converted into a system of ordinary differential equations which can be integrated by starting at an initial data location. When the resulting ordinary differential equations can be solved analytically in closed form, then we will consider the solution to be an exact solution, whereas solutions requiring numerical integration or series solutions will be considered approximate (see Section 6.5).

Table 6.1 *Exact solutions to the 1-D unsteady heat conduction equation.*

Solutions to $\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$
$T(t,x) = A\left(x^3 + 6\alpha t x\right) + B$
$T(t,x) = A\left(x^4 + 12\alpha t x^2 + 12\alpha^2 t^2\right) + B$
$T(t,x) = x^{2n} + \sum_{k=1}^{n} \frac{(2n)(2n-1)\cdots(2n-2k+1)}{k!}(\alpha t)^k x^{2n-2k}$
$T(t,x) = A\exp\left(\alpha\mu^2 t \pm \mu x\right) + B$
$T(t,x) = A\frac{1}{\sqrt{t}}\exp\left(\frac{-x^2}{4\alpha t}\right) + B$
$T(t,x) = A\exp(-\mu x)\cos\left(\mu x - 2\alpha\mu^2 t + B\right) + C$
$T(t,x) = A\,\mathrm{erf}\left(\frac{x}{2\sqrt{\alpha t}}\right) + B$

6.2.1.4 *Advanced approaches*

Additional approaches for obtaining exact solutions to PDEs were developed in the latter half of the twentieth century. One example is the method of differential constraints developed by Yanenko (1964). Another example is the application of group theory, which has found extensive applications in algebra and geometry, for finding solutions to differential equations. While a discussion of these nonclassical analytic solutions techniques is beyond the scope of this book, additional information on the application of these methods for PDEs can be found in Polyanin (2002), Polyanin and Zaitsev (2004), and Meleshko (2005).

6.2.2 *Example exact solution: 1-D unsteady heat conduction*

Some general solutions to the 1-D unsteady heat conduction equation given by Eq. (6.1) are presented in Table 6.1 above, where A, B, C, and μ are arbitrary constants and n is a positive integer. These solutions, as well as many others, can be found in Polyanin (2002). Employing Eq. (6.8), these solutions can also be transformed into solutions to Burgers' equation.

6.2.3 *Example with order verification: steady Burgers' equation*

This section describes an exact solution for the steady Burgers' equation, which is then employed in an order verification test for a finite difference discretization. Benton and Platzmann (1972) describe 35 exact solutions to Burgers' equation, which is given above in Eq. (6.7). The steady-state form of Burgers' equation is found by restricting the solution u to be a function of x only, thus reducing Eq. (6.7) to the following ordinary differential

equation

$$u\frac{\mathrm{d}u}{\mathrm{d}x} = \nu\frac{\mathrm{d}^2u}{\mathrm{d}x^2}, \tag{6.11}$$

where u is a velocity and ν is the viscosity. The exact solution to Burgers' equation for a steady, viscous shock (Benton and Platzmann, 1972) is given in dimensionless form, denoted by primes, as

$$u'(x') = -2\tanh(x'). \tag{6.12}$$

This dimensionless solution for Burgers' equation can be converted to dimensional quantities via transformations given by $x' = x/L$ and $u' = uL/\nu$ where L is a reference length scale. This solution for Burgers' equation is also invariant to scaling by a factor α as follows: $\bar{x} = x/\alpha$ and $\bar{u} = \alpha u$. Finally, one can define a Reynolds number in terms of L, a reference velocity u_{ref}, and the viscosity ν as

$$\text{Re} = \frac{u_{\text{ref}}L}{\nu}, \tag{6.13}$$

where the domain is generally chosen as $-L \leq x \leq L$ and u_{ref} the maximum value of u on the domain.

For this example, the steady Burgers' equation is discretized using the simple implicit finite difference scheme given by

$$\bar{u}_i\left(\frac{u_{i+1}^{k+1} - u_{i-1}^{k+1}}{2\Delta x}\right) - \nu\left(\frac{u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1}}{\Delta x^2}\right) = 0, \tag{6.14}$$

where a uniform mesh with spacing Δx is used between the spatial nodes which are indexed by i. The formal order of accuracy of this discretization scheme is two and can be found from a truncation error analysis. The above discretization scheme is linearized by setting $\bar{u}_i = u_i^k$ and then iterated until the solution at iteration k satisfies Eq. (6.14) within round-off error using double precision computations. This results in the iterative residuals, found by substituting $\bar{u}_i = u_i^{k+1}$ into Eq. (6.14), being reduced by approximately fourteen orders of magnitude. Thus, iterative convergence and round-off errors can be neglected (see Chapter 7), and the numerical solutions effectively contain only discretization error.

The solution to Burgers' equation for a Reynolds number of eight is given in Figure 6.1, which includes both the exact solution (in scaled dimensional variables) and a numerical solution obtained using 17 evenly-spaced points (i.e., nodes). The low value for the Reynolds number was chosen for this code verification exercise to ensure that both the convection and the diffusion terms are exercised. Note that choosing a large Reynolds number effectively scales down the diffusion term since it is multiplied by 1/Re when written in dimensionless form. For higher Reynolds numbers, extremely fine meshes would be needed to detect coding mistakes in the diffusion term.

Numerical solutions for the steady Burgers' equation were obtained on seven uniform meshes from the finest mesh of 513 nodes ($h = 1$) to the coarsest mesh of nine nodes

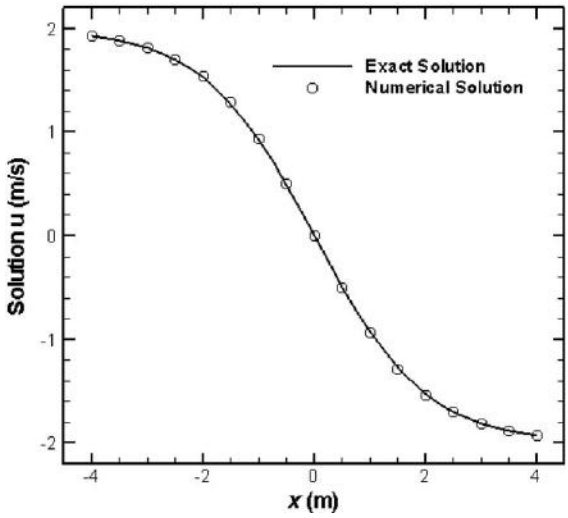

Figure 6.1 Comparison of the numerical solution using 17 nodes with exact solution for the steady Burgers equation with Reynolds number Re = 8.

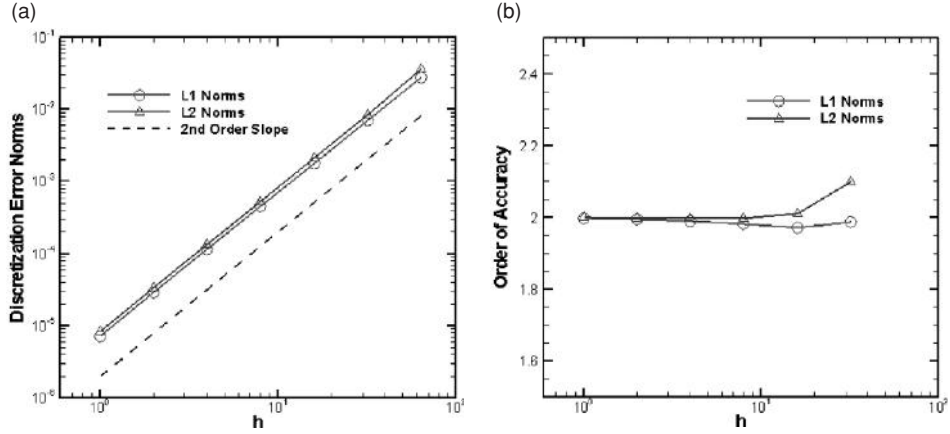

Figure 6.2 Discretization error (a) and observed orders of accuracy (b) for steady Burger's equation with Reynolds number Re = 8.

($h = 64$). Discrete L_1 and L_2 norms of the discretization error are given in Figure 6.2a and both norms appear to reduce with mesh refinement at a second-order rate. The order of accuracy, as computed from Eq. (5.23), is given in Figure 6.2b. Both norms rapidly approach the scheme's formal order of accuracy of two with mesh refinement for this simple problem. The code used to compute the numerical solutions to Burgers' equation is thus considered to be verified for the options exercised, namely steady-state solutions on a uniform mesh.

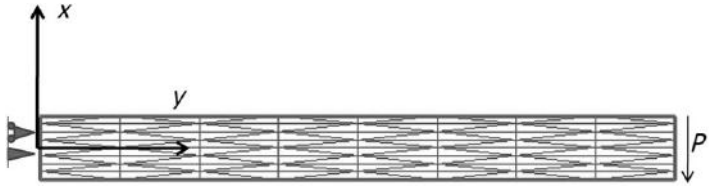

Figure 6.3 2-D linear elasticity problem for plane stress in a cantilevered beam loaded at the tip; an unstructured mesh with 64 triangular elements is also shown.

6.2.4 Example with order verification: linear elasticity

This section describes an exact solution for linear elasticity and includes an example of order verification for a finite element code. The problem of interest is a cantilevered beam that is loaded at the tip, as shown in Figure 6.3. The equations governing the displacements u and v in the x and y directions, respectively, arise from the static equilibrium linear momentum equations for plane stress. An isotropic linear elastic material is assumed along with small strain (i.e., small deformation gradient) assumptions. The equations governing the displacements can be written as

$$\left(1+\frac{1-\alpha}{2\alpha-1}\right)\frac{\partial^2 u}{\partial x^2}+\frac{1}{2}\frac{\partial^2 u}{\partial y^2}+\left(\frac{1}{2}+\frac{1-\alpha}{2\alpha-1}\right)\frac{\partial^2 v}{\partial x\partial y}=0,$$
$$\left(1+\frac{1-\alpha}{2\alpha-1}\right)\frac{\partial^2 v}{\partial y^2}+\frac{1}{2}\frac{\partial^2 v}{\partial x^2}+\left(\frac{1}{2}+\frac{1-\alpha}{2\alpha-1}\right)\frac{\partial^2 u}{\partial x\partial y}=0, \tag{6.15}$$

where for plane stress

$$\alpha=\frac{1}{1+\nu}$$

and ν is Poisson's ratio.

For a beam of length L, height h, and width (in the z direction) of w, an exact solution can be found for the displacements (Slaughter, 2002). This solution has been modified (Seidel, 2009) using the above coordinate system resulting in an Airy stress function of

$$\Phi=-2\frac{xPy^3}{h^3w}+2\frac{LPy^3}{h^3w}+3/2\frac{xyP}{hw}, \tag{6.16}$$

which exactly satisfies the equilibrium and compatibility conditions. The stresses can then be easily obtained by

$$\sigma_{xx}=\frac{\partial^2\Phi}{\partial y^2}=-12\frac{xyP}{h^3w}+12\frac{LPy}{h^3w},$$
$$\sigma_{yy}=\frac{\partial^2\Phi}{\partial x^2}=0,$$
$$\sigma_{xy}=\frac{\partial^2\Phi}{\partial x\partial y}=6\frac{Py^2}{h^3w}-3/2\frac{P}{hw}. \tag{6.17}$$

The stress–strain relationship is given by

$$\sigma_{xx} = \frac{E}{1-\nu^2}\left(\varepsilon_{xx} + \nu\varepsilon_{yy}\right),$$
$$\sigma_{yy} = \frac{E}{1-\nu^2}\left(\nu\varepsilon_{xx} + \varepsilon_{yy}\right),$$
$$\sigma_{xy} = \frac{E}{1+\nu}\varepsilon_{xy},$$

and the strain is related to the displacements by

$$\varepsilon_{xx} = \frac{\partial u}{\partial x},$$
$$\varepsilon_{yy} = \frac{\partial v}{\partial y},$$
$$\varepsilon_{xy} = \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right).$$

This solution results in traction-free conditions on the upper and lower surfaces, i.e.,

$$\sigma_{yy}(x, h/2) = \sigma_{xy}(x, h/2) = \sigma_{yy}(x, -h/2) = \sigma_{xy}(x, -h/2) = 0,$$

and static equivalent tip loads of zero net axial force, zero bending moment, and the applied shear force at the tip of $-P$:

$$\int_{-h/2}^{h/2} \sigma_{xx}(L, y)\mathrm{d}y = 0,$$
$$\int_{-h/2}^{h/2} y\sigma_{xx}(L, y)\mathrm{d}y = 0,$$
$$\int_{-h/2}^{h/2} \sigma_{xy}(L, y)\mathrm{d}y = -P/w.$$

The conditions at the wall are fully constrained at the neutral axis ($y = 0$) and no rotation at the top corner ($y = h/2$):

$$u(0, 0) = 0,$$
$$v(0, 0) = 0,$$
$$u(0, h/2) = 0.$$

The displacement in the x and y directions thus become

$$u = 1/2\left(2\frac{Py^3}{h^3 w} - 3/2\frac{yP}{hw} + \alpha\left(-6\frac{x^2 Py}{h^3 w} + 12\frac{LPyx}{h^3 w} + 2\frac{Py^3}{h^3 w} - 1/2\frac{(-2P + \alpha P)y}{w\alpha h}\right)\right)\mu^{-1},$$

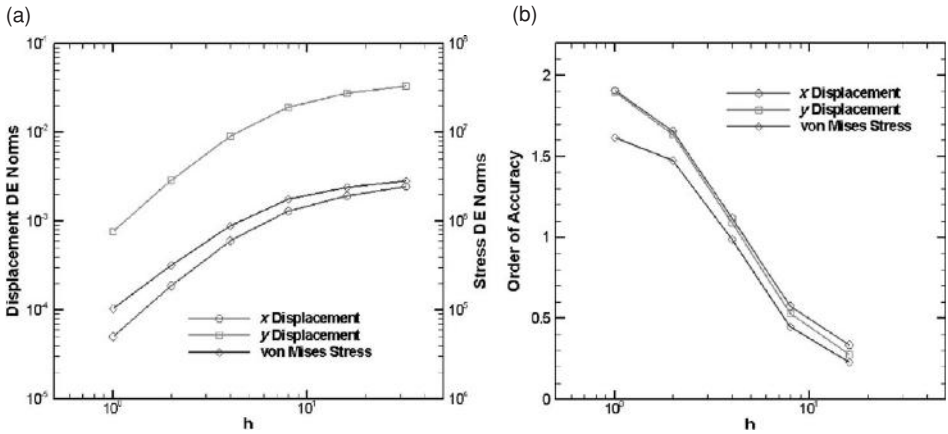

Figure 6.4 Discretization error (a) and observed orders of accuracy (b) for the steady Burger's equation with Reynolds number Re = 8.

$$v = 1/2\left(6\frac{xPy^2}{h^3w} - 6\frac{xPLy^2}{h^3w} - 3/2\frac{xP}{hw} + \alpha\left(-6\frac{xPy^2}{h^3w} + 6\frac{PLy^2}{h^3w} + 2\frac{x^3P}{h^3w} - 6\frac{PLx^2}{h^3w} + 1/2\frac{(-2P+\alpha P)x}{w\alpha h}\right)\right)\mu^{-1}, \tag{6.18}$$

where μ is the shear modulus.

Finite element solutions were obtained for the weak form of Eq. (6.15) using linear basis functions, which gives a formally second-order accurate scheme for the displacements (Seidel, 2009). The maximum von Mises stress J_2 was also computed according to

$$J_2 = \frac{1}{6}\left[\left(\sigma_{xx} - \sigma_{yy}\right)^2 + \sigma_{xx}^2 + \sigma_{yy}^2 + 6\sigma_{xy}^2\right].$$

Simulations were run using six systematically-refined mesh levels from a coarse mesh of eight elements to a fine mesh of 8192 elements. Figure 6.4a shows the L_2 norms of the discretization error in the displacements and the discretization error in maximum von Mises stress, with all three quantities displaying convergent behavior. The orders of accuracy for these three quantities are given in Figure 6.4b and show that the displacements asymptote to second-order accuracy while the maximum von Mises stress appears to converge to somewhat less than second order.

6.3 Method of manufactured solutions (MMS)

This section addresses the difficult question of how to create an exact solution for complex PDEs, where complexity refers to characteristics such as nonlinearity, nonconstant coefficients, irregular domain shape, higher dimensions, multiple submodels, and coupled systems of equations. The traditional methods for obtaining exact solutions discussed

earlier generally cannot handle this level of complexity. The primary need for exact solutions to complex PDEs is for order of accuracy testing during code verification.

The *method of manufactured solutions* (MMS) is a general and very powerful approach for creating exact solutions. Rather than trying to find an exact solution to a PDE with given initial and boundary conditions, the goal is to "manufacture" an exact solution to a slightly modified equation. For code verification purposes, it is not required that the manufactured solution be related to a physically realistic problem; recall that code verification deals only with the mathematics of a given problem. The general concept behind MMS is to choose a solution *a priori*, then operate the governing PDEs onto the chosen solution, thereby generating additional analytic source terms that require no discretization. The chosen (manufactured) solution is then the exact solution to the modified governing equations made up of the original equations plus the additional analytic source terms. Thus, MMS involves the solution to the backward problem: given an original set of equations and a chosen solution, find a modified set of equations that the chosen solution will satisfy.

While the MMS approach for generating exact solutions was not new (e.g., see Zadunaisky, 1976; Stetter, 1978), Roache and Steinberg (1984) and Steinberg and Roache (1985) appear to be the first to employ these exact solutions for the purposes of code verification. Their original work looked at the asymptotic rate of convergence of the discretization errors with systematic mesh refinement. Shih (1985) independently developed a similar procedure for debugging scientific computing codes, but without employing mesh refinement to assess convergence or order of accuracy.

The concepts behind MMS for the purpose of code verification were later refined by Roache *et al*. (1990) and Roache (1998). The term "manufactured solution" was coined by Oberkampf *et al*. (1995) and refers to the fact that the method generates (or manufactures) a related set of governing equations for a chosen analytic solution. An extensive discussion of manufactured solutions for code verification is presented by Knupp and Salari (2003) and includes details of the method as well as application to a variety of different PDEs. Recent reviews of the MMS procedure are presented by Roache (2002) and Roy (2005). While it is not uncommon for MMS to be used for verifying computational fluid dynamics codes (e.g., Roache *et al*., 1990; Pelletier *et al*., 2004; Roy *et al*., 2004; Eca *et al*., 2007), it has also begun to appear in other disciplines, for example, fluid–structure interaction (Tremblay *et al*., 2006).

MMS allows the generation of exact solutions to PDEs of nearly arbitrary complexity, with notable exceptions discussed in Section 6.3.3. When combined with the order verification procedure described in Chapter 5, MMS provides a powerful tool for code verification. When physically realistic manufactured solutions are desired, the modified MMS approaches discussed in Section 6.4 can be used.

6.3.1 Procedure

The procedure for creating an exact solution using MMS is fairly straightforward. For a scalar mathematical model, this procedure can be summarized as follows.

Step 1 Establish the mathematical model in the form $L(u) = 0$, where $L(\bullet)$ is the differential operator and u the dependent variable.

Step 2 Choose the analytic form of the manufactured solution $\hat{u}$.

Step 3 Operate the mathematical model $L(\bullet)$ onto the manufactured solution $\hat{u}$. to obtain the analytic source term $s = L(\hat{u})$.

Step 4 Obtain the modified form of the mathematical model by including the analytic source term $L(u) = s$.

Because of the manner in which the analytic source term is obtained, it is a function of the independent variables only and does not depend on u. The initial and boundary conditions can be obtained directly from the manufactured solution $\hat{u}$. For a system of equations, the manufactured solution $\hat{u}$ and the source term s are simply considered to be vectors, but otherwise the process is unchanged.

An advantage of using MMS to generate exact solutions to general mathematical models is that the procedure is not affected by nonlinearities or coupled sets of equations. However, the approach is conceptually different from the standard training scientists and engineers receive in problem solving. Thus it is helpful to examine a simple example which highlights the subtle nature of MMS.

Consider again the unsteady 1-D heat conduction equation that was examined in Section 6.2.1.1. The governing partial differential equation written in the form $L(T) = 0$ is

$$\frac{\partial T}{\partial t} - \alpha \frac{\partial^2 T}{\partial x^2} = 0. \tag{6.19}$$

Once the governing equation has been specified, the next step is to choose the analytic manufactured solution. A detailed discussion on how to choose the solution will follow in Section 6.3.1.1, but for now, consider a combination of exponential and sinusoidal functions:

$$\hat{T}(x, t) = T_o \exp(t/t_o) \sin(\pi x/L). \tag{6.20}$$

Due to the analytic nature of this chosen solution, the derivatives that appear in the governing equation can be evaluated exactly as

$$\frac{\partial \hat{T}}{\partial t} = T_o \sin(\pi x/L) \frac{1}{t_o} \exp(t/t_o)$$
$$\frac{\partial^2 \hat{T}}{\partial x^2} = -T_o \exp(t/t_o) (\pi/L)^2 \sin(\pi x/L).$$

We now modify the mathematical model by including the above derivatives, along with the thermal diffusivity α, on the right hand side of the equation. The modified governing equation that results is

$$\frac{\partial T}{\partial t} - \alpha \frac{\partial^2 T}{\partial x^2} = \left[\frac{1}{t_0} + \alpha \left(\frac{\pi}{L} \right)^2 \right] T_o \exp(t/t_o) \sin(\pi x/L). \tag{6.21}$$

The left hand side of Eq. (6.21) is the same as in the original mathematical model given by Eq. (6.19), thus no modifications to the underlying numerical discretization in the code

under consideration need to be made. The right hand side could be thought of in physical terms as a distributed source term, but in fact it is simply a convenient mathematical construct that will allow straightforward code verification testing. The key concept behind MMS is that the exact solution to Eq. (6.21) is known and is given by Eq. (6.20), the manufactured solution $\hat{T}(x, t)$ that was chosen in the beginning.

As the governing equations become more complex, symbolic manipulation tools such as Mathematica™, Maple™, or MuPAD should be used. These tools have matured greatly over the last two decades and can produce rapid symbolic differentiation and simplification of expressions. Most symbolic manipulation software packages have built-in capabilities to output the solution and the source terms directly as computer source code in both Fortran and C/C^{++} programming languages.

6.3.1.1 Manufactured solution guidelines for code verification

When used for code verification studies, manufactured solutions should be chosen to be analytic functions with smooth derivatives. It is important to ensure that no derivatives vanish, including cross-derivatives if these show up in the governing equations. Trigonometric and exponential functions are recommended since they are smooth and infinitely differentiable. Recall that the order verification procedures involve systematically refining the spatial mesh and/or time step, thus obtaining numerical solutions can be expensive for complex 3-D applications. In some cases, the high cost of performing order verification studies in multiple dimensions using MMS can be significantly reduced simply by reducing the frequency content of the manufactured solution over the selected domain. In other words, it is not important to have a full period of a sinusoidal manufactured solution in the domain, often only a fraction (one-third, one-fifth, etc.) of a period is sufficient to exercise the terms in the code.

Although the manufactured solutions do not need to be physically realistic when used for code verification, they should be chosen to obey certain physical constraints. For example, if the code requires the temperature to be positive (e.g., in the evaluation of the speed of sound which involves the square root of the temperature), then the manufactured solution should be chosen to give temperature values significantly larger than zero.

Care should be taken that one term in the governing equations does not dominate the other terms. For example, even if the actual application area for a Navier–Stokes code will be for high-Reynolds number flows, when performing code verification studies, the manufactured solution should be chosen to give Reynolds numbers near unity so that convective and diffusive terms are of the same order of magnitude. For terms that have small relative magnitudes (e.g., if the term is scaled by a small parameter such as 1/Re), mistakes can still be found through order verification, but possibly only on extremely fine meshes.

A more rigorous approach to ensuring that all of the terms in the governing equations are roughly the same order of magnitude over some significant region of the domain is to examine ratios of those terms. This process has been used by Roy *et al*. (2007b) as part of the order of accuracy verification of a computational fluid dynamics code including

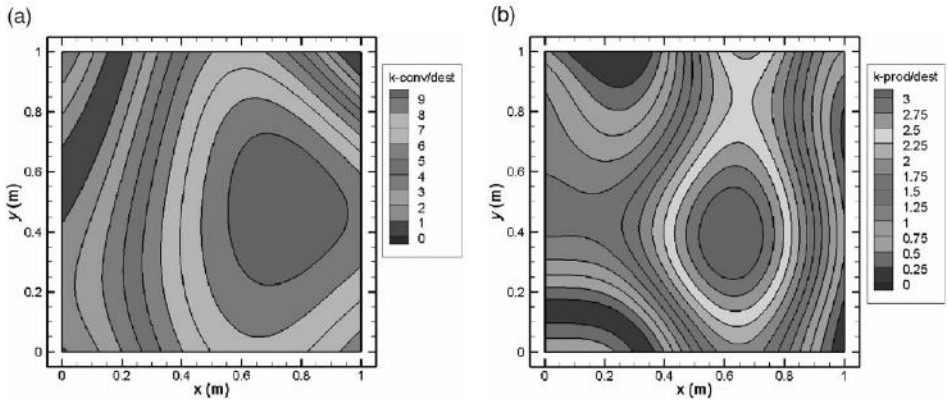

Figure 6.5 Ratios of (a) the convection terms and (b) the production terms to the destruction terms for a turbulent kinetic energy transport equation (from Roy *et al*., 2007b).

a complicated two-equation turbulence model. Example ratios for different terms in the equation governing the transport of turbulent kinetic energy are given in Figure 6.5. These plots show that the convection, production, and destruction terms in this transport equation are of roughly the same order of magnitude over most of the domain.

6.3.1.2 Boundary and initial conditions

The discretization schemes for the PDE and the various submodels comprise a significant fraction of the possible options in most scientific computing codes. When performing code verification studies on these options, there are two approaches for handling boundary and initial conditions. The first approach is to impose the mathematically consistent boundary and initial conditions that are required according to the mathematical character of the differential equations. For example, if Dirichlet (fixed-value) or Neumann (fixed-gradient) boundary conditions are required, these can be determined directly from the analytic manufactured solution (although these will generally not be constant along the boundary). The second option is to simply specify all boundary values with Dirichlet or Neumann values from the manufactured solution. This latter approach, although mathematically ill-posed, often does not adversely affect the order of accuracy test. In any case, over-specification of boundary conditions will not lead to a false positive for order of accuracy testing (i.e., a case where the order of accuracy is verified but there is a mistake in the code or inconsistency in the discrete algorithm).

In order to verify the implementation of the boundary conditions, the manufactured solution should be tailored to exactly satisfy a given boundary condition on a domain boundary. Bond *et al*. (2007) present a general approach for tailoring manufactured solutions to ensure that a given boundary condition is satisfied along a general boundary. The method involves multiplying any standard manufactured solution by a function which has values and/or derivatives which are zero over a specified boundary. To modify the standard form of the manufactured solution for a 2-D steady-state solution for temperature, one may simply

write the manufactured solution as follows:

$$\hat{T}(x, y) = T_0 + \hat{T}_1(x, y), \tag{6.22}$$

where $\hat{T}_1(x, y)$ is any baseline manufactured solution. For example, this manufactured solution could take the form

$$\hat{T}_1(x, y) = T_x f_s\left(\frac{a_x \pi x}{L}\right) + T_y f_s\left(\frac{a_y \pi y}{L}\right), \tag{6.23}$$

where the $f_s(\cdot)$ functions represent a mixture of sines and cosines and T_x, T_y, a_x, and a_y are constants (note the subscripts used here do not denote differentiation).

For 2-D problems, a boundary can be represented by the general curve $F(x, y) = C$, where C is a constant. A new manufactured solution appropriate for verifying boundary conditions can be found by multiplying the spatially-varying portion of $\hat{T}(x, y)$ by the function $[C - F(x, y)]^m$, i.e.,

$$\hat{T}_{\text{BC}}(x, y) = T_0 + \hat{T}_1(x, y)\,[C - F(x, y)]^m. \tag{6.24}$$

This procedure will ensure that the manufactured solution is equal to the constant T_0 along the specified boundary for $m = 1$. Setting $m = 2$, in addition to enforcing the above Dirichlet BC for temperature, will ensure that the gradient of temperature normal to the boundary is equal to zero, i.e., the adiabatic boundary condition for this 2-D steady heat conduction example. In practice, the curve $F(x, y) = C$ is used to define the domain boundary where the boundary conditions will be tested.

To illustrate this procedure, we will choose the following simple manufactured solution for temperature:

$$\hat{T}(x, y) = 300 + 25\cos\left(\frac{7}{4}\frac{\pi x}{L}\right) + 40\sin\left(\frac{4}{3}\frac{\pi y}{L}\right), \tag{6.25}$$

where the temperature is assumed to be in units of Kelvin. For the surface where the Dirichlet or Neumann boundary condition will be applied, choose:

$$F(x, y) = \frac{1}{2}\cos\left(\frac{0.4\pi x}{L}\right) - \frac{y}{L} = 0. \tag{6.26}$$

A mesh bounded on the left by $x/L = 0$, on the right by $x/L = 1$, on the top by $y/L = 1$, and on the bottom by the above defined surface $F(x, y) = 0$ is shown in Figure 6.6a. The standard manufactured solution given by Eq. (6.25) is also shown in Figure 6.6b, where clearly the constant value and gradient boundary conditions are not satisfied.

Combining the baseline manufactured solution with the boundary specification yields

$$\hat{T}_{\text{BC}}(x, y) = 300 + \left[25\cos\left(\frac{7}{4}\frac{\pi x}{L}\right) + 40\sin\left(\frac{4}{3}\frac{\pi y}{L}\right)\right]\left[-\frac{1}{2}\cos\left(\frac{0.4\pi x}{L}\right) + \frac{y}{L}\right]^m. \tag{6.27}$$

When $m = 1$, the curved lower boundary will satisfy the constant temperature condition of 300 K as shown in the manufactured solutions of Figure 6.7a. When $m = 2$ (Figure 6.7b),

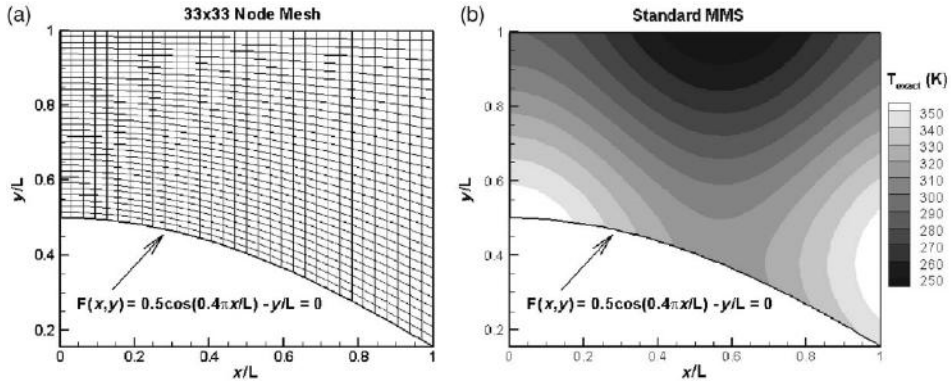

Figure 6.6 Grid (a) and baseline manufactured solution (b) for the MMS boundary condition example.

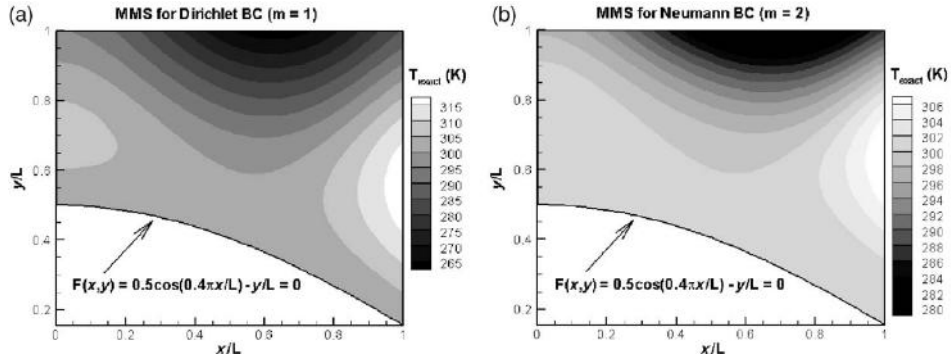

Figure 6.7 Manufactured solutions given by Eq. (6.27): (a) fixed temperature boundary condition ($m = 1$) and (b) fixed temperature and zero gradient (adiabatic) boundary condition ($m = 2$).

the zero gradient (i.e., adiabatic) boundary condition is also satisfied. For more details on this approach as well as extensions to 3-D, see Bond *et al*. (2007).

6.3.2 *Benefits of MMS for code verification*

There are a number of benefits to using the MMS procedure for code verification. Perhaps the most important benefit is that it handles complex mathematical models without additional difficulty since the procedures described above are readily extendible to nonlinear, coupled systems of equations. In addition, MMS can be used to verify most of the coding options available in scientific computing codes, including the consistency/convergence of numerical algorithms. The procedure is not tied to a specific discretization scheme, but works equally well for finite-difference, finite-volume, and finite-element methods.

The use of MMS for code verification has been shown to be remarkably sensitive to mistakes in the discretization (see for example Chapter 3.11 of Roache (1998)). In one

particular case of a compressible Navier–Stokes code for unstructured grids (Roy *et al*., 2007b), global norms of the discretization error were found to be non-convergent. Further investigation found a small discrepancy (in the fourth significant figure!) for the constant thermal conductivity between the governing equations used to generate the source terms and the model implementation in the code. When this discrepancy was corrected, the order verification test was passed. The same study uncovered an algorithm inconsistency in the discrete formulation of the diffusion operator that resulted in non-ordered behavior on skewed meshes. This same formulation had been implemented in at least one commercial computational fluid dynamics code (see Roy *et al*. (2007b) for more details).

In addition to its ability to indicate the presence of coding mistakes (i.e., bugs), the MMS procedure combined with order verification is also a powerful tool for finding and removing those mistakes (i.e., debugging). After a failed order of accuracy test, individual terms in the mathematical model and the numerical discretization can be omitted, allowing the user to quickly isolate the terms with the coding mistake. When combined with the approach for verifying boundary conditions discussed in the previous section and a suite of meshes with different topologies (e.g., hexahedral, prismatic, tetrahedral, and hybrid meshes, as discussed in Chapter 5), the user has a comprehensive set of tools to aid in code debugging.

6.3.3 Limitations of MMS for code verification

There are some limitations to using the MMS procedure for code verification. The principal disadvantage is that it requires the user to incorporate arbitrary source terms, initial conditions, and boundary conditions in a code. Even when the code provides a framework for including these additional interfaces, their specific form changes for each different manufactured solution. The MMS procedure is thus code-intrusive and generally cannot be performed as a black-box testing procedure where the code simply returns some outputs based on a given set of inputs. In addition, each code option which changes the mathematical model requires new source terms to be generated. Thus order verification with MMS can be time consuming when many code options must be verified.

Since the MMS procedure for code verification relies on having smooth solutions, the analysis of discontinuous weak solutions (e.g., solutions with shock-waves) is still an open research issue. Some traditional exact solutions exist for discontinuous problems such as the generalized Riemann problem (Gottlieb and Groth, 1988) and more complicated solutions involving shock waves and detonations have been developed that involve infinite series (Powers and Stewart, 1992) or a change of dependent variable (Powers and Aslam, 2006). However, to our knowledge, discontinuous manufactured solutions have not been created. Such "weak" exact solutions are needed for verifying codes used to solve problems with discontinuities.

Difficulties also arise when applying MMS to mathematical models where the governing equations themselves contain `min`, `max`, or other nonsmooth switching functions. These functions generally do not result in continuous manufactured solution source terms. These

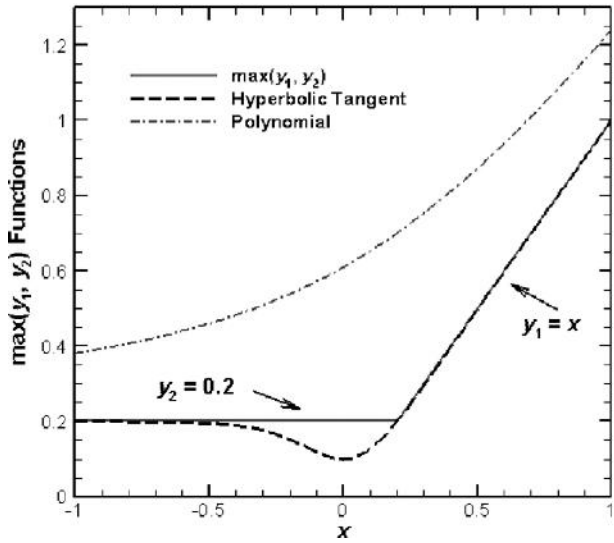

Figure 6.8 Examples of smooth approximations for $\max(y_1, y_2)$ using the hyperbolic tangent from Eq. (6.28) and the polynomial from Eq. (6.29).

switching functions can be dealt with by simply turning off different branches of the switching functions (Eca *et al.*, 2007) or by tailoring the manufactured solution so that only one switching branch will be used for a given verification test (Roy *et al.*, 2007b). The former is simpler but more code intrusive than the latter approach. We recommend that model developers employ smooth blending functions such as the hyperbolic tangent both to simplify the code verification testing and to possibly make the numerical solution process more robust. For example, consider the function $\max(y_1, y_2)$, where y_1 and y_2 are given by

$$y_1(x) = x,$$
$$y_2 = 0.2.$$

One approach for smoothing this max function in the region of $x = 0.2$ is the hyperbolic tangent smoothing function given by

$$\max(y_1, y_2) \approx F y_1 + (1 - F) y_2,$$
$$\text{where } F = \frac{1}{2}\left[\tanh\left(y_1 / y_2\right) + 1\right]. \tag{6.28}$$

Another approach is to use the following polynomial expression:

$$\max(y_1, y_2) \approx \frac{\sqrt{(y_1 - y_2)^2 + 1} + y_1 + y_2}{2}. \tag{6.29}$$

The two approximations of $\max(y_1, y_2)$ are shown graphically in Figure 6.8. The hyperbolic tangent approximation provides less error relative to the original $\max(y_1, y_2)$ function, but

creates an inflection point where the first derivative (slope) of this function will change sign. The polynomial function is monotone, but gives a larger error magnitude. Models that rely on tabulated data (i.e., look-up tables) suffer from similar problems, and smooth approximations of such data should be considered. MMS is also limited when the mathematical model contains complex algebraic submodels which do not have closed-form solutions and thus must be solved numerically (e.g., by a root-finding algorithm). Such complex submodels are best addressed separately through unit and/or component level software testing discussed in Chapter 4.

6.3.4 Examples of MMS with order verification

Two examples are now presented which use MMS to generate exact solutions. These manufactured solutions are then employed for code verification using the order of accuracy test.

6.3.4.1 2-D steady heat conduction

Order verification using MMS has been applied to steady-state heat conduction with a constant thermal diffusivity. The governing equation simply reduces to Poisson's equation for temperature:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = s(x, y), \tag{6.30}$$

where $s(x, y)$ is the manufactured solution source term. Coordinate transformations of the form

$$(x, y) \rightarrow (\xi, \eta)$$

are used to globally transform the governing equation into a Cartesian computational space with $\Delta\xi = \Delta\eta = 1$ (Thompson *et al.*, 1985). The transformed governing equation thus becomes

$$\frac{\partial F_1}{\partial \xi} + \frac{\partial G_1}{\partial \eta} = \frac{s(x, y)}{J}, \tag{6.31}$$

where J is the Jacobian of the mesh transformation. The fluxes F_1 and G_1 are defined as

$$F_1 = \frac{\xi_x}{J} F + \frac{\xi_y}{J} G,$$
$$G_1 = \frac{\eta_x}{J} F + \frac{\eta_y}{J} G,$$

where

$$F = \xi_x \frac{\partial T}{\partial \xi} + \eta_x \frac{\partial T}{\partial \eta},$$
$$G = \xi_y \frac{\partial T}{\partial \xi} + \eta_y \frac{\partial T}{\partial \eta}.$$

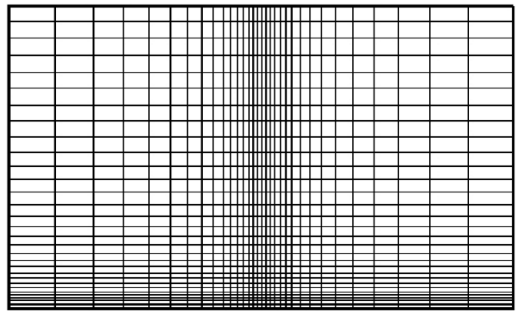

Figure 6.9 Stretched Cartesian mesh with 33×33 nodes for the 2-D steady heat conduction problem.

An explicit point Jacobi method (Tannehill *et al.*, 1997) is used to advance the discrete equations towards the steady-state solution. Standard, three-point, centered finite differences are used in the transformed coordinates and central differences are also employed for the grid transformation metric terms (x_ξ, x_η, y_ξ, etc.), thus resulting in a discretization scheme that is formally second-order accurate in space. The numerical solutions were iteratively converged to machine zero, i.e., the iterative residuals were reduced by approximately 14 orders of magnitude for the double precision computations employed. Thus iterative and round-off error are assumed to be negligible and the numerical solutions are effectively the exact solution to the discrete equation.

The following manufactured solution was chosen

$$\hat{T}(x, y) = T_0 + T_x \cos\left(\frac{a_x \pi x}{L}\right) + T_y \sin\left(\frac{a_y \pi y}{L}\right) + T_{xy} \sin\left(\frac{a_{xy} \pi xy}{L^2}\right), \qquad (6.32)$$

where

$$T_0 = 400 \text{ K}, \quad T_x = 45 \text{ K}, \quad T_y = 35 \text{ K}, \quad T_{xy} = 27.5 \text{ K},$$
$$a_x = 1/3, \quad a_y = 1/4, \quad a_{xy} = 1/2, \quad L = 5 \text{ m},$$

and Dirichlet (fixed-value) boundary conditions were applied on all four boundaries as determined by Eq. (6.32). A family of stretched Cartesian meshes was created by first generating the finest mesh (129×129 nodes), and then successively eliminating every other gridline to create the coarser meshes. Thus systematic refinement (or coarsening in this case) is ensured. The 33×33 node mesh is presented in Figure 6.9, showing significant stretching in the x-direction in the center of the domain and in the y-direction near the bottom boundary. The manufactured solution from Eq. (6.32) is shown graphically in Figure 6.10. The temperature varies smoothly over the domain and the manufactured solution gives variations in both coordinate directions.

Discrete L_2 norms of the discretization error (i.e., the difference between the numerical solution and the manufactured solution) were computed for grid levels from 129×129 nodes ($h = 1$) to 9×9 nodes ($h = 16$). These norms are given in Figure 6.11a and closely follow the expected second order slope. The observed order of accuracy of the L_2 norms of the discretization error was computed from Eq. (5.23) for successive mesh levels, and the

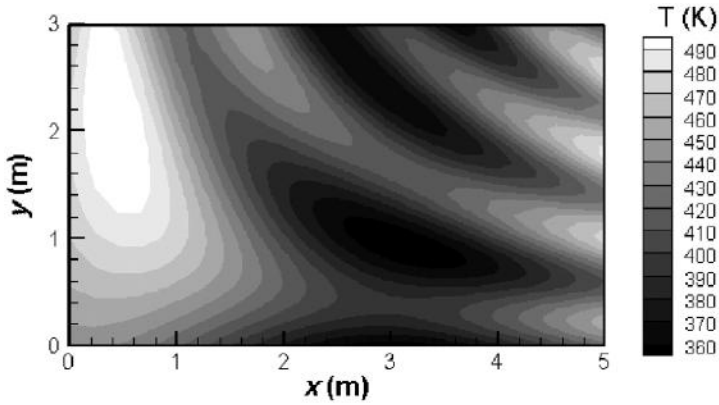

Figure 6.10 Manufactured solution for temperature for the 2-D steady heat conduction problem.

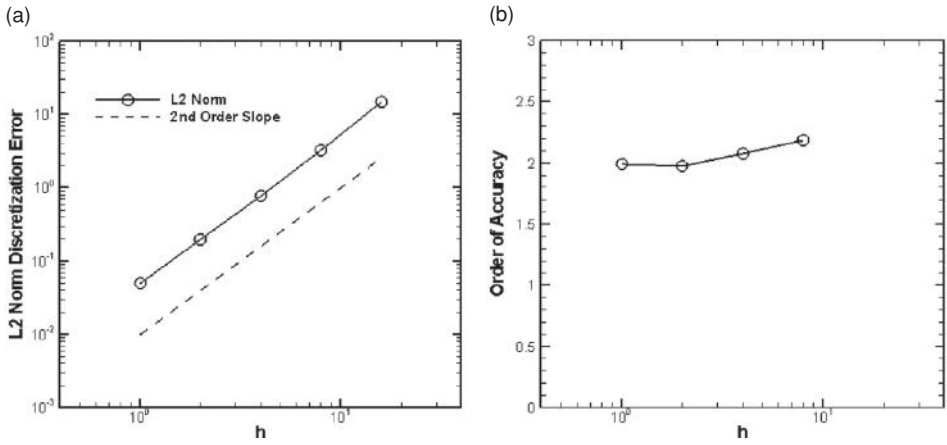

Figure 6.11 Code verification for the 2-D steady heat conduction problem: (a) discrete L2 norms of the discretization error and (b) observed order of accuracy.

results are shown in Figure 6.11b. The observed order of accuracy is shown to converge to the formal order of two as the meshes are refined, thus the code is considered to be verified for the options tested. Note that while the discrete transformations were tested with respect to the clustering of the mesh, this choice of grid topologies would not test the implementation of the grid metric terms dealing with cell skewness or coordinate rotation (e.g., ξ_y, η_x).

6.3.4.2 2D Steady Euler equations

This MMS example deals with the Euler equations, which govern the flow of an inviscid fluid. This example is adapted from Roy *et al.* (2004) and we will demonstrate the steps of both generating the exact solution with MMS and the order verification procedure. The

two-dimensional, steady-state form of the Euler equations is given by

$$\begin{aligned}
\frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} &= s_m(x, y), \\
\frac{\partial(\rho u^2 + p)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} &= s_x(x, y), \\
\frac{\partial(\rho vu)}{\partial x} + \frac{\partial(\rho v^2 + p)}{\partial y} &= s_y(x, y), \\
\frac{\partial(\rho u e_t + pu)}{\partial x} + \frac{\partial(\rho v e_t + pv)}{\partial y} &= s_e(x, y),
\end{aligned} \tag{6.33}$$

where arbitrary source terms $s(x, y)$ are included on the right hand side for use with MMS. In these equations, u and v are the Cartesian components of velocity in the x- and y-directions, ρ the density, p the pressure, and e_t is the specific total energy, which for a calorically perfect gas is given by

$$e_t = \frac{1}{\gamma - 1} RT + \frac{u^2 + v^2}{2}, \tag{6.34}$$

where R is the specific gas constant, T the temperature, and γ the ratio of specific heats. The final relation used to close the set of equations is the perfect gas equation of state:

$$p = \rho RT. \tag{6.35}$$

The manufactured solutions for this case are chosen as simple sinusoidal functions given by

$$\begin{aligned}
\rho(x, y) &= \rho_0 + \rho_x \sin\left(\frac{a_{\rho x}\pi x}{L}\right) + \rho_y \cos\left(\frac{a_{\rho y}\pi y}{L}\right), \\
u(x, y) &= u_0 + u_x \sin\left(\frac{a_{ux}\pi x}{L}\right) + u_y \cos\left(\frac{a_{uy}\pi y}{L}\right), \\
v(x, y) &= v_0 + v_x \cos\left(\frac{a_{vx}\pi x}{L}\right) + v_y \sin\left(\frac{a_{vy}\pi y}{L}\right), \\
p(x, y) &= p_0 + p_x \cos\left(\frac{a_{px}\pi x}{L}\right) + p_y \sin\left(\frac{a_{py}\pi y}{L}\right).
\end{aligned} \tag{6.36}$$

The subscripts here refer to constants (not differentiation) with the same units as the variable, and the dimensionless constants a generally vary between 0.5 and 1.5 to provide low frequency solutions over an $L \times L$ square domain. For this example, the constants were chosen to give supersonic flow in both the positive x and positive y directions. While not necessary, this choice simplifies the inflow boundary conditions to Dirichlet (specified) values at the inflow boundaries, whereas outflow boundary values are simply extrapolated from the interior. The inflow boundary conditions are specified directly from the manufactured solution. The specific constants chosen for this example are shown in Table 6.2, and a plot of the manufactured solution for the density is given in Figure 6.12. The density varies smoothly in both coordinate directions between 0.92 and 1.13 kg/m^3.

Table 6.2 *Constants for the supersonic Euler manufactured solution*

Equation, ϕ	ϕ_0	ϕ_x	ϕ_y	$a_{\phi x}$	$a_{\phi y}$
ρ(kg/m^3)	1	0.15	−0.1	1	0.5
u(m/s)	800	50	−30	1.5	0.6
v(m/s)	800	−75	40	0.5	2./3.
p(N/m^2)	1×10^5	0.2×10^5	0.5×10^5	2	1

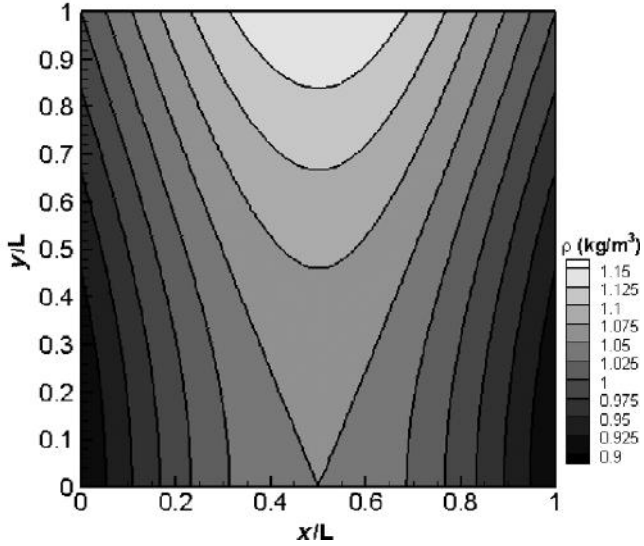

Figure 6.12 Manufactured solution for density for the Euler equations.

Substitution of the chosen manufactured solutions into the governing equations allows the analytic determination of the source terms. For example, the source term for the mass conservation equation is given by:

$$
\begin{aligned}
s_{\mathrm{m}}(x,\,y) = {} & \frac{a_{ux}\pi u_x}{L}\cos\left(\frac{a_{ux}\pi x}{L}\right)\left[\rho_0 + \rho_x \sin\left(\frac{a_{\rho x}\pi x}{L}\right) + \rho_y \cos\left(\frac{a_{\rho y}\pi y}{L}\right)\right] \\
& + \frac{a_{vy}\pi v_y}{L}\cos\left(\frac{a_{vy}\pi y}{L}\right)\left[\rho_0 + \rho_x \sin\left(\frac{a_{\rho x}\pi x}{L}\right) + \rho_y \cos\left(\frac{a_{\rho y}\pi y}{L}\right)\right] \\
& + \frac{a_{\rho x}\pi \rho_x}{L}\cos\left(\frac{a_{\rho x}\pi x}{L}\right)\left[u_0 + u_x \sin\left(\frac{a_{ux}\pi x}{L}\right) + u_y \cos\left(\frac{a_{uy}\pi y}{L}\right)\right] \\
& + \frac{a_{\rho y}\pi \rho_y}{L}\sin\left(\frac{a_{\rho y}\pi y}{L}\right)\left[v_0 + v_x \cos\left(\frac{a_{vx}\pi x}{L}\right) + v_y \sin\left(\frac{a_{vy}\pi y}{L}\right)\right].
\end{aligned}
$$

The source terms for the momentum and energy equations are significantly more complex, and all source terms were obtained using Mathematica™. A plot of the source term for the energy conservation equation is given in Figure 6.13. Note the smooth variations of the source term in both coordinate directions.

Table 6.3 *Cartesian meshes employed in the Euler manufactured solution*

Mesh name	Mesh nodes	Mesh spacing, h
Mesh 1	129×129	1
Mesh 2	65×65	2
Mesh 3	33×33	4
Mesh 4	17×17	8
Mesh 5	9×9	16

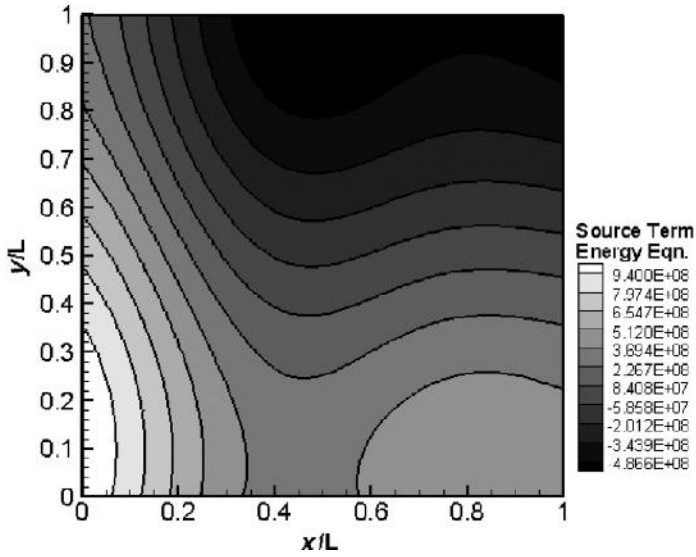

Figure 6.13 Analytic source term for the energy conservation equation.

The governing equations are discretized and solved on multiple meshes. In this case, two different finite-volume computational fluid dynamics codes were employed: Premo, an unstructured grid code, and Wind, a structured grid code (see Roy *et al.* (2004) for more details). Both codes utilized the second-order Roe upwind scheme for the convective terms (Roe, 1981). The formal order of accuracy of both codes is thus second order for smooth problems.

The five Cartesian meshes employed are summarized in Table 6.3. The coarser meshes were found by successively eliminating every other grid line from the fine mesh (i.e., a grid refinement factor of $r = 2$). It is important to note that while the current example was performed on Cartesian meshes for simplicity, a more general code verification analysis would employ the most general meshes which will be used by the code (e.g., unstructured

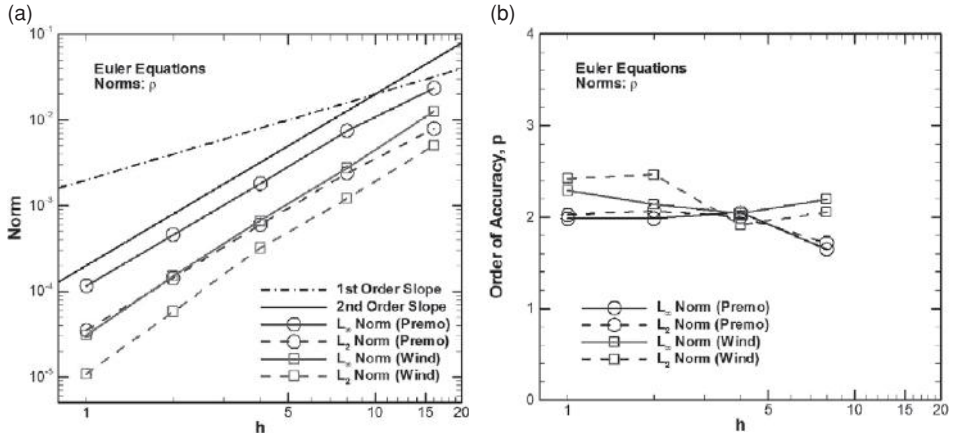

Figure 6.14 Code verification results for the 2-D steady Euler equations: (a) norms of the discretization error and (b) observed order of accuracy.

meshes with significant stretching, skewness, boundary orientations). See Section 5.4 for additional discussion of mesh topology issues.

The global discretization error is measured using discrete L_∞ and L_2 norms of the discretization error, where the exact solution comes directly from the chosen manufactured solution given by Eqs. (6.36). The behavior of these two discretization error norms for the density ρ as a function of the cell size h is given in Figure 6.14a. On the logarithmic scale, a first-order scheme will display a slope of unity, while a second-order scheme will give a slope of two. The discretization error norms for the density appear to converge with second-order accuracy.

A more quantitative method for assessing the observed order of accuracy is to calculate it using the norms of the discretization error. Since the exact solution is known, the relation for the observed order of accuracy of the discretization error norms comes from Eq. (5.23). A plot of the observed order of accuracy as a function of the element size h is presented in Figure 6.14b. The Premo code clearly asymptotes to second-order accuracy, while the Wind code appears to asymptote to an order of accuracy that is slightly higher than two. In general, an observed order of accuracy higher than the formal order can occur due to error cancellation and should not be considered as a failure of the order verification test (although it may indicate mistakes is determining the formal order of accuracy of the method). Further grid refinement would possibly provide more definitive results for the Wind code. In this case, the observed order of accuracy for both codes is near two, thus the formal order of accuracy is recovered, and the two codes are considered verified for the options examined.

6.4 Physically realistic manufactured solutions

The MMS procedure discussed in the previous section is the most general method for obtaining exact solutions to mathematical models for use in code verification studies. Since

physical realism of the solutions is not required during code verification, the solutions are somewhat arbitrary and can be tailored to exercise all terms in the mathematical model. However, there are many cases in which physically realistic exact solutions are desired, such as assessing sensitivity of a numerical scheme to mesh quality, evaluating the reliability of discretization error estimators, and judging the overall effectiveness of solution adaptation schemes. There are two main approaches for obtaining physically realistic manufactured solutions to complex equations, and these two approached are discussed below.

6.4.1 Theory-based solutions

One approach to obtaining physically realistic manufactured solutions is to use a simplified theoretical model of the physical phenomenon as a basis for the manufactured solution. For example, if a physical process is known to exhibit an exponential decay in the solution with time, then a manufactured solution of the form

$$\alpha \exp(-\beta t)$$

could be employed, where α and β could be chosen to provide physically meaningful solutions.

There are two examples of this approach applied to the modeling of fluid turbulence. Pelletier *et al*. (2004) have verified a 2-D incompressible finite element code that employs a k-ε two-equation turbulence model. They constructed manufactured solutions which mimic turbulent shear flows, with the turbulent kinetic energy and the turbulent eddy viscosity as the two quantities specified in the manufactured solution. More recently, Eca and Hoekstra (2006) and Eca *et al*. (2007) developed physically realistic manufactured solutions mimicking steady, wall-bounded turbulence for 2-D incompressible Navier–Stokes codes. They examined both one- and two-equation turbulence models and noted challenges in generating physically realistic solutions in the near-wall region.

6.4.2 Method of nearby problems (MNP)

The second approach for generating physically realistic manufactured solutions is called the *method of nearby problems* (MNP) and was proposed by Roy and Hopkins (2003). This approach involves first computing a highly-refined numerical solution for the problem of interest, then generating an accurate curve fit of that numerical solution. If both the underlying numerical solution and the curve fit are "sufficiently" accurate, then it will result in a manufactured solution which has small source terms. The sufficiency conditions for the "nearness" of the problem have been explored for first-order quasi-linear ordinary differential equations (Hopkins and Roy, 2004) but rigorous bounds on this nearness requirement for PDEs have not yet been developed.

MNP has been successfully demonstrated for one-dimensional problems by Roy *et al*. (2007a) who used the procedure to create a nearby solution to the steady-state Burgers'

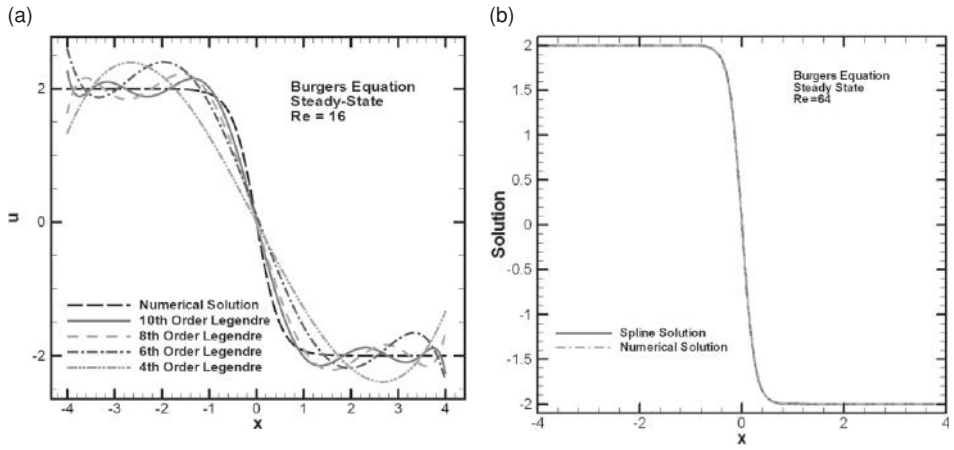

Figure 6.15 Examples of curve fitting for the viscous shock wave solution to Burgers' equation: (a) global Legendre polynomial fits for Re = 16 and (b) fifth-order Hermite splines for Re = 64 (from Roy *et al.*, 2007a).

equation for a viscous shock wave. They used fifth-order Hermite splines to generate the exact solutions for Reynolds numbers of 8, 64, and 512. To explain why spline fits must be used, rather than global curve fits, consider Figure 6.15. Global Legendre polynomial fits for the steady-state Burgers' equation for a viscous shock at a Reynolds number of 16 are given in Figure 6.15a. Not only is the viscous shock wave not adequately resolved, but the global fits also exhibit significant oscillations at the boundaries. Hermite spline fits for an even higher Reynolds number of 64 are given in Figure 6.15b, with the spline fit in very good qualitative agreement with the underlying numerical solution. MNP has been extended to 2-D problems by Roy and Sinclair (2009) and the further extension to higher dimensions is straightforward. A 2-D example of MNP used to generate an exact solution to the incompressible Navier–Stokes equations will be given in Section 6.4.2.2.

6.4.2.1 Procedure

The steps for generating physically realistic manufactured solutions using the MNP approach are:

1 compute the original numerical solution on a highly refined grid,
2 generate an accurate spline or curve fit to this numerical solution, thereby providing an analytic representation of the numerical solution,
3 operate the governing partial differential equations on the curve fit to generate analytic source terms (which ideally will be small), and
4 create the nearby problem by appending the analytic source terms to the original mathematical model.

If the source terms are indeed small, then the new problem will be "near" the original one, hence the name "method of nearby problems." The key point to this approach is that, by

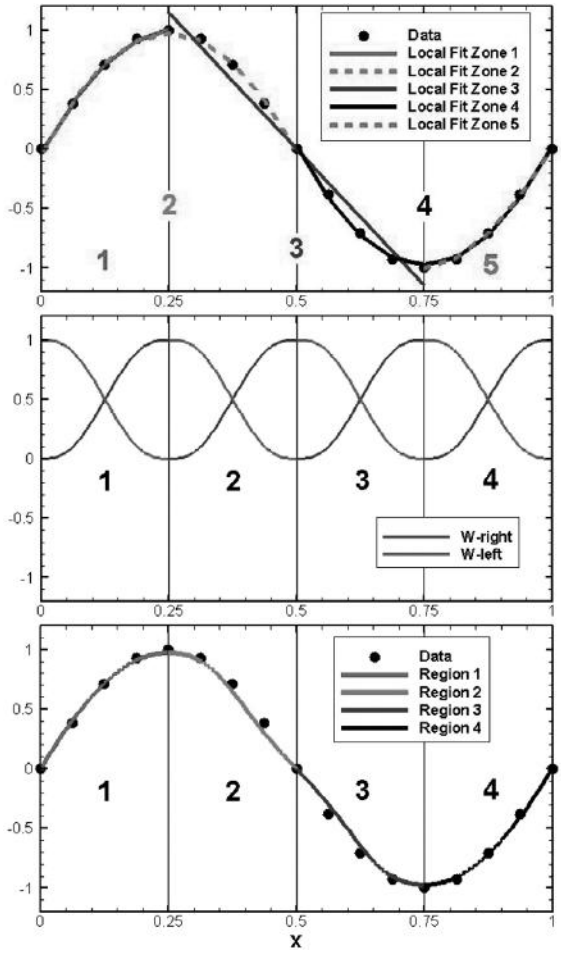

Figure 6.16 Simple one-dimensional example of the weighting function approach for combining local quadratic least squares fits to generate a C^2 continuous spline fit: local fits (top), weighting functions (middle), and resulting C^2 continuous spline fit (bottom) (from Roy and Sinclair, 2009). (See color plate section.)

definition, the curve fit generated in step 2 is the exact solution to the nearby problem. While the approach is very similar to MMS, in MNP the addition of the curve fitting step is designed to provide a physically realistic exact solution.

To demonstrate the MNP procedure, a simple 1-D example is presented in Figure 6.16, where the original data used to generate the curve fit are 17 points sampled at equal intervals from the function $\sin(2\pi x)$. The goal of this example is to create a spline fit made up of four spline regions that exhibits C^2 continuity at the spline zone interfaces (i.e., continuity up to the second derivative). The spline fitting is performed in a manner that allows arbitrary levels of continuity at spline boundaries and is readily extendible to multiple dimensions following Junkins *et al*. (1973).

The first step is to generate five overlapping local fits Z_1 through Z_5, with each of the interior fits spanning two spline regions (see top of Figure 6.16). A least squares method is used to find a best-fit quadratic function in each of the five regions:

$$Z_n(\bar{x}) = a_n + b_n\bar{x} + c_n\bar{x}^2. \tag{6.37}$$

The overbars in Eq. (6.37) specify that the spatial coordinate x is locally transformed to satisfy $0 \le \bar{x} \le 1$ in each of the interior spline zones. Since each spline zone now has two different local fits, one from the left and the other from the right, these two local fits are combined together with the left and right weighting functions shown in Figure 6.16 (middle). The form of the 1-D weighting function used here for C^2 continuity is

$$W_{\text{right}}(\bar{x}) = \bar{x}^3\left(10 - 15\bar{x} + 6\bar{x}^2\right)$$

and the corresponding left weighting function is defined simply as

$$W_{\text{left}}(\bar{x}) = W_{\text{right}}(1 - \bar{x}).$$

Thus the final fit in each region can be written as

$$F(x, y) = W_{\text{left}}Z_{\text{left}} + W_{\text{right}}Z_{\text{right}}.$$

For example, for region 2, one would have $Z_{\text{left}} = Z_2$ and $Z_{\text{right}} = Z_3$. Note that, in addition to providing the desired level of continuity at spline boundaries, the weighting functions are also useful in reducing the dependence near the boundaries of the local fits where they often exhibit the poorest agreement with the original data. When these final fits are plotted (bottom of Figure 6.16), we see that they are indeed C^2 continuous, maintaining continuity of the function value, slope, and curvature at all three interior spline boundaries.

6.4.2.2 *Example exact solution: 2-D steady Navier–Stokes equations*

An example of the use of MNP to generate physically realistic manufactured solutions is now given for the case of viscous, incompressible flow in a lid-driven cavity at a Reynolds number of 100 (Roy and Sinclair, 2009). This flow is governed by the incompressible Navier–Stokes equations, which for constant transport properties are given by

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = s_{\text{m}}(x, y),$$

$$\rho u\frac{\partial u}{\partial x} + \rho v\frac{\partial u}{\partial y} + \frac{\partial p}{\partial x} - \mu\frac{\partial^2 u}{\partial x^2} - \mu\frac{\partial^2 u}{\partial y^2} = s_x(x, y),$$

$$\rho u\frac{\partial v}{\partial x} + \rho v\frac{\partial v}{\partial y} + \frac{\partial p}{\partial y} - \mu\frac{\partial^2 v}{\partial x^2} - \mu\frac{\partial^2 v}{\partial y^2} = s_y(x, y),$$

where $s(x, y)$ are the manufactured solution source terms. These equations are solved in finite-difference form on a standard Cartesian mesh by integrating in pseudo-time using Chorin's artificial compressibility method (Chorin, 1967). In addition, second- and fourth-derivative damping (Jameson *et al.*, 1981) was employed to prevent odd–even decoupling (i.e., oscillations) in the solution. Dirichlet boundary conditions are used for velocity, with

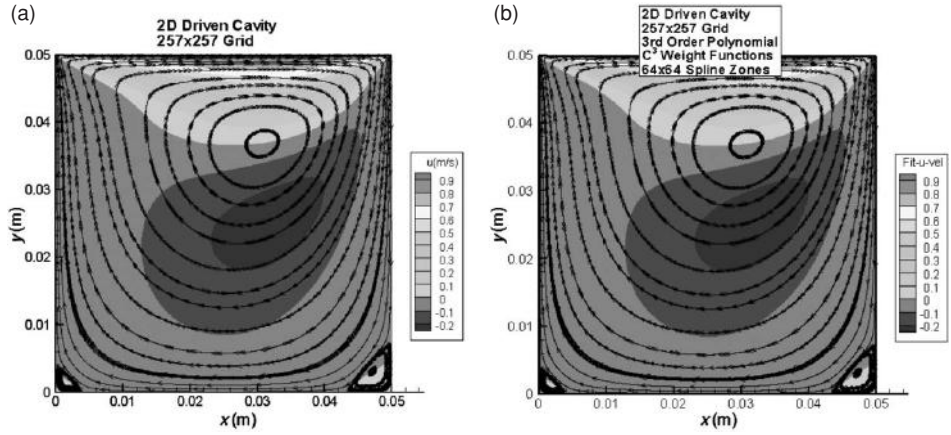

Figure 6.17 Contours of u-velocity and streamlines for a lid-driven cavity at Reynolds number 100: (a) 257×257 node numerical solution and (b) C^3 continuous spline fit using 64×64 spline zones (from Roy and Sinclair, 2009). (See color plate section.)

all boundary velocities equal to zero except for the u-velocity on the top wall which is set to 1 m/s.

A contour plot of the u-velocity (i.e., the velocity in the x-direction) from a numerical solution on a 257×257 grid is given in Figure 6.17a. Also shown in the figure are streamlines which denote the overall clockwise circulation induced by the upper wall velocity (the upper wall moves from left to right) as well as the two counter-clockwise rotating vortices in the bottom corners. A spline fit was generated using third order (i.e., bi-cubic) polynomials in x and y with C^3 continuous weighting functions and 64×64 spline zones. Note that while no additional boundary constraints are placed on the velocity components for the spline fit, the maximum deviations from the original boundary conditions are on the order of 1×10^{-7} m/s and are thus quite small. The u-velocity contours and streamlines for the spline fit are presented in Figure 6.17b. The fit solution is qualitatively the same as the underlying numerical solution. The streamlines were injected at exactly the same locations in both figures and are indistinguishable from each other. Furthermore, in both cases the streamlines near the center of the cavity follow the same path for multiple revolutions.

A more quantitative comparison between the underlying numerical solution and the spline fits is presented in Figure 6.18, which shows discrete norms of the spline fitting error in u-velocity relative to the underlying numerical solution as a function of the number of spline zones in each direction. The average error magnitude (L_1 norm) decreases from 1×10^{-3} m/s to 3×10^{-6} m/s with increasing number of spline zones from 8×8 to 64×64, while the maximum error (infinity norm) decreases from 0.7 m/s to 0.01 m/s.

6.5 Approximate solution methods

This section describes three methods for approximating exact solutions to mathematical models. The first two, series and similarity solutions, are often considered to be exact, but are

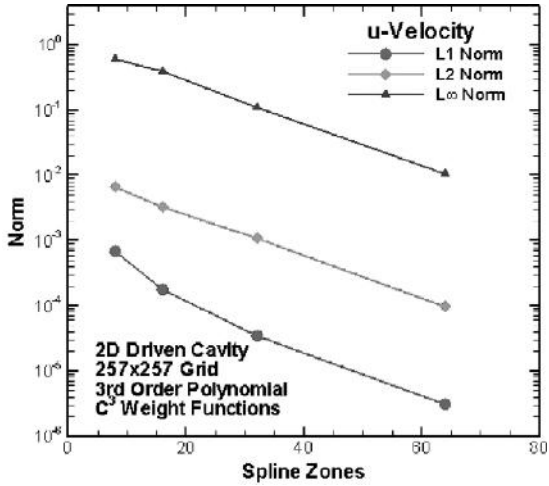

Figure 6.18 Variation of the error in u-velocity between the spline fits and the underlying 257×257 numerical solution as a function of the number of spline zones in each direction for the lid-driven cavity at Reynolds number 100 (from Roy and Sinclair, 2009).

treated as approximate here since we assume that numerical values for the solution must be computed. Furthermore, infinite series and similarity solutions are usually only available for simple PDEs. The third method involves computing a highly-accurate numerical solution to a given problem and is called a numerical benchmark solution.

6.5.1 Infinite series solutions

Solutions involving infinite series are sometimes used to solve differential equations with general boundary and initial conditions. The primary application of series solutions has been for linear differential equations, but they are also a useful tool for obtaining solutions to certain nonlinear differential equations. While these solutions are "analytic," they are not closed form solutions since they involve infinite series. When using infinite series as an approximation of an exact solution to the mathematical model, care must be taken to ensure that the series is in fact convergent and the numerical approximation error created by truncating the series is sufficiently small for the intended application. As Roache (1998) points out, there are many cases where subtle issues arise with the numerical evaluation of infinite series solutions, so they should be used with caution.

6.5.2 Reduction to ordinary differential equations

In some cases, a suitable transformation can be found which reduces a system of PDEs to a system of ordinary differential equations. Methods are available to compute highly-accurate numerical or series solutions for ordinary differential equations. One example is the well-known Blasius solution to the laminar boundary layer equations in fluid dynamics (Schetz, 1993). This solution employs similarity transformations to reduce the incompressible

boundary layer equations for conservation of mass and momentum to a single, nonlinear ordinary differential equation, which can then be accurately solved using series solution (the original approach of Blasius) or by numerical approximation. Consider the situation where a code based on the full Navier–Stokes equation is used to solve for the laminar boundary layer flow over a flat plate. In this case, the solution from the Navier–Stokes code would not converge to the Blasius solution since these are two different mathematical models; the Navier–Stokes equations contain terms omitted from the boundary layer equations which are expected to become important near the leading edge singularity.

6.5.3 Benchmark numerical solutions

Another approximate solution method is to compute a *benchmark numerical solution* with a high-degree of numerical accuracy. In order for a numerical solution to a complex PDE to qualify as a benchmark solution, the problem statement, numerical scheme, and numerical solution accuracy should be documented (Oberkampf and Trucano, 2008). Quantifying the numerical accuracy of benchmark numerical solutions is often difficult, and at a minimum should include evidence that (1) the asymptotic convergence range has been achieved for the benchmark problem and (2) the code used to generate the benchmark solution has passed the order of accuracy code verification test for the options exercised in the benchmark problem. Extensive benchmark numerical solutions for solid mechanics applications are discussed in Oberkampf and Trucano (2008).

6.5.4 Example series solution: 2-D steady heat conduction

The problem of interest in this example is steady-state heat conduction in an infinitely long bar with a rectangular cross section of width L and height H (Dowding, 2008). A schematic of the problem is given in Figure 6.19. If constant thermal conductivity is assumed, then the conservation of energy equation reduces to a Poisson equation for temperature,

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{-\dot{g}'''}{k}, \tag{6.38}$$

where T is the temperature, k is the thermal conductivity, and $\dot{g}'''$ is an energy source term. The bottom and left boundaries employ zero heat flux (Neumann) boundary conditions, the right boundary a fixed temperature (Dirichlet) boundary condition, and the top boundary is a convective heat transfer (Robin) boundary condition. These boundary conditions are also given in Figure 6.19 and can be summarized as

$$\begin{aligned} &\frac{\partial T}{\partial x}(0, y) = 0, \\ &\frac{\partial T}{\partial y}(x, 0) = 0, \\ &T(L, y) = T_\infty, \\ &-k\frac{\partial T}{\partial y}(x, H) = h\,[T(x, H) - T_\infty], \end{aligned} \tag{6.39}$$

where h is the film coefficient from convective cooling.

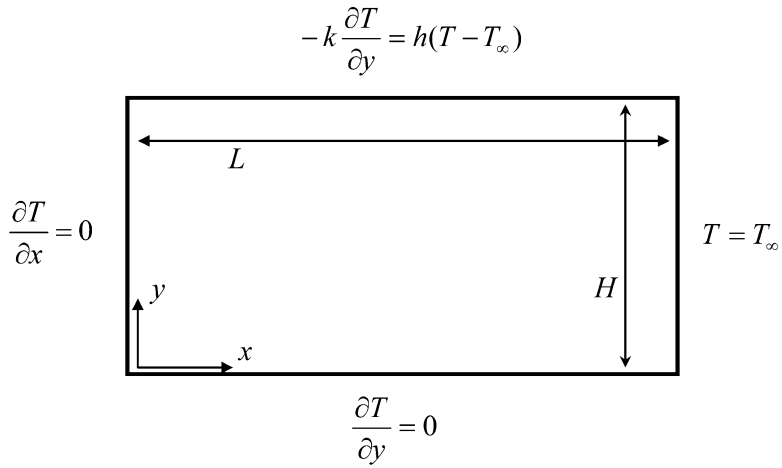

Figure 6.19 Schematic of heat conduction problem in an infinitely long bar of rectangular cross section (Dowding, 2008).

In order to make the Dirichlet boundary condition homogeneous (i.e., equal to zero), the following simple transformation is used:

$$\omega(x, y) = T(x, y) - T_\infty. \tag{6.40}$$

Note that the use of this transformation does not change the form of the governing equation, which can be rewritten in terms of ω as

$$\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2} = \frac{-\dot{g}'''}{k}. \tag{6.41}$$

The problem statement in terms of $\omega(x, y)$ is shown in Figure 6.20.

The solution to the transformed problem in terms of ω can be found using separation of variables and is

$$\frac{\omega(x, y)}{\frac{\dot{g}''' L^2}{k}} = \frac{1}{2}\left(1 - \frac{x^2}{L^2}\right) + 2\sum_{n=1}^{\infty} \frac{(-1)^n}{\mu_n^3} \frac{\cosh\left(\frac{\mu_n y}{aH}\right)\cos\left(\mu_n \frac{x}{L}\right)}{\frac{1}{Bi}\frac{\mu_n}{a}\sinh\left(\frac{\mu_n}{a}\right) + \cosh\left(\frac{\mu_n}{a}\right)}, \tag{6.42}$$

where the eigenvalues μ_n are given by

$$\mu_n = (2n - 1)\frac{\pi}{2}, \quad n = 1, 2, 3, \ldots \quad \text{and} \cos(\mu_n) = 0,$$

and the constant a and the Biot number Bi are defined as:

$$a = \frac{L}{H}, \quad Bi = \frac{hH}{k}.$$

The infinite series is found to converge rapidly everywhere except near the top wall, where over 100 terms are needed to obtain accuracies of approximately seven significant figures (Dowding, 2008). The following parameters have been used to generate the exact solution given in Figure 6.21, which is presented in terms of the temperature by using the simple

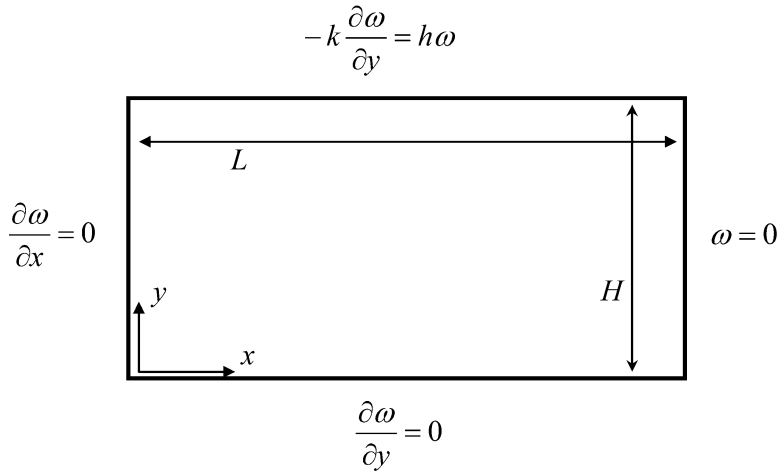

Figure 6.20 Schematic of heat conduction problem in terms of ω (Dowding, 2008).

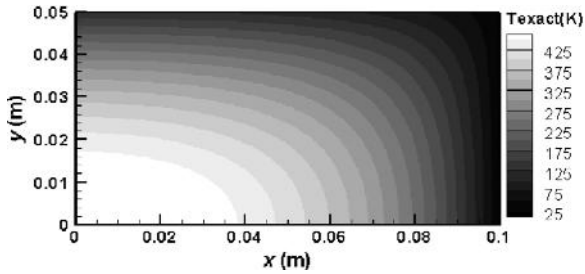

Figure 6.21 Infinite series solution for 2-D steady-state heat conduction.

transformation from Eq. (6.40):

$$\begin{aligned} \dot{g}''' &= 135\,300 \text{ W/m}^3, \\ k &= 0.4 \text{ W/(m} \cdot \text{K)}, \\ L &= 0.1 \text{ m}, \\ H &= 0.05 \text{ m}, \\ T_\infty &= 25 \text{ K}. \end{aligned}$$

These parameters correspond to an aspect ratio of 2, a Biot number of 7.5, and a dimensionless heat source $\dot{g}''' L^2/k$ of 3 382.5.

6.5.5 *Example benchmark convergence test: 2-D hypersonic flow*

An example of benchmark numerical solutions used with the convergence test for code verification is given by Roy *et al*. (2003). They considered the Mach 8 inviscid flow

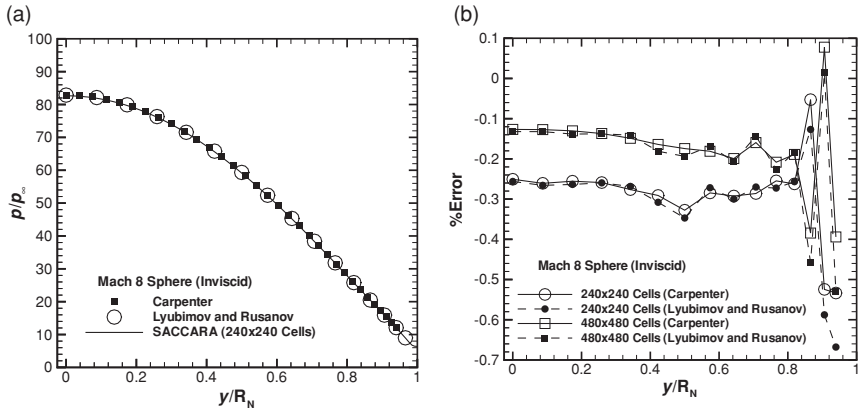

Figure 6.22 Compressible computational fluid dynamics predictions for the Mach 8 flow over a sphere: (a) surface pressure distributions and (b) discretization error in the surface pressure from the SACCARA code relative to the two benchmark solutions (from Roy *et al*., 2003).

of a calorically perfect gas over a sphere-cone geometry. This flow is governed by the Euler equations in axisymmetric coordinates. Two benchmark numerical solutions were employed: a higher-order spectral solution (Carpenter *et al*., 1994) and an accurate finite-difference solution (Lyubimov and Rusanov, 1973). Numerical solutions for surface pressure were computed using the compressible computational fluid dynamics code SACCARA (see Roy *et al*., 2003 for details) and are compared to these two benchmark solutions on the spherical nose region in Figure 6.22. While the pressure distributions appear identical in Figure 6.22a, examination of the discretization error in the SACCARA solution relative to the benchmark solutions in Figure 6.22b shows that the discretization error is small (less than 0.7%) and that it decreases by approximately a factor of two with mesh refinement (i.e., the numerical solutions are convergent). Variations are seen in the discretization error near the sphere-cone tangency point due to the presence of the geometric singularity in the computational fluid dynamics solution (a discontinuity in surface curvature). While these results do demonstrate that the SACCARA code has passed the convergence test, it is generally difficult to assess the order of accuracy using benchmark numerical solutions. A similar benchmark solution for the Navier–Stokes equations involving viscous laminar flow can also be found in Roy *et al*. (2003).

6.6 References

Ames, W. F. (1965). *Nonlinear Partial Differential Equations in Engineering*, New York, Academic Press Inc.

Benton, E. R. and G. W. Platzman (1972). A table of solutions of the one-dimensional Burgers' equation, *Quarterly of Applied Mathematics*. **30**(2), 195–212.

Bond, R. B., C. C. Ober, P. M. Knupp, and S. W. Bova (2007). Manufactured solution for computational fluid dynamics boundary condition verification, *AIAA Journal*. **45**(9), 2224–2236.

Carpenter, M. H., H. L. Atkins, and D. J. Singh (1994). Characteristic and finite-wave shock-fitting boundary conditions for Chebyshev methods, In *Transition, Turbulence, and Combustion*, eds. M. Y. Hussaini, T. B. Gatski, and T. L. Jackson, Vol. 2, Norwell, MA, Kluwer Academic, pp. 301–312.

Carslaw, H. S. and J. C. Jaeger (1959). *Conduction of Heat in Solids*, 2nd edn., Oxford, Clarendon Press.

Chorin, A. J. (1967). A numerical method for solving incompressible viscous flow problems, *Journal of Computational Physics*. **2**(1), 12–26.

Dowding, K. (2008). Private communication, January 8, 2008.

Eca, L. and M. Hoekstra (2006). Verification of turbulence models with a manufactured solution, *European Conference on Computational Fluid Dynamics, ECCOMAS CFD 2006*, Wesseling, P., Onate, E., and Periaux, J. (eds.), Egmond ann Zee, The Netherlands, ECCOMAS.

Eca, L., M. Hoekstra, A. Hay, and D. Pelletier (2007). On the construction of manufactured solutions for one and two-equation eddy-viscosity models, *International Journal for Numerical Methods in Fluids*. **54**(2), 119–154.

Elishakoff, I. (2004). *Eigenvalues of Inhomogeneous Structures: Unusual Closed-Form Solutions*, Boca Raton, FL, CRC Press.

Gottlieb, J. J. and C. P. T. Groth (1988). Assessment of Riemann solvers for unsteady one-dimensional inviscid flows of perfect gases, *Journal of Computational Physics*. **78**(2), 437–458.

Hirsch, C. (2007). *Numerical Computation of Internal and External Flows: Fundamentals of Computational Fluid Dynamics*, 2nd edn., Oxford, Butterworth-Heinemann.

Hopkins, M. M. and C. J. Roy (2004) Introducing the method of nearby problems, *European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS 2004*, P. Neittaanmaki, T. Rossi, S. Korotov, E. Onate, J. Periaux, and D. Knorzer (eds.), University of Jyväskylä (Jyvaskyla), Jyväskylä, Finland, July 2004.

Jameson, A., W. Schmidt, and E. Turkel (1981). *Numerical Solutions of the Euler Equations by Finite Volume Methods Using Runge-Kutta Time-Stepping Schemes*, AIAA Paper 81–1259.

Junkins, J. L., G. W. Miller, and J. R. Jancaitis (1973). A weighting function approach to modeling of irregular surfaces, *Journal of Geophysical Research*. **78**(11), 1794–1803.

Kausel, E. (2006). *Fundamental Solutions in Elastodynamics: a Compendium*, New York, Cambridge University Press.

Kevorkian, J. (2000). *Partial Differential Equations: Analytical Solution Techniques*, 2nd edn., Texts in Applied Mathematics, 35, New York, Springer.

Knupp, P. and K. Salari (2003). *Verification of Computer Codes in Computational Science and Engineering*, K. H. Rosen (ed.), Boca Raton, Chapman and Hall/CRC.

Lyubimov, A. N. and V. V. Rusanov (1973). *Gas Flows Past Blunt Bodies, Part II: Tables of the Gasdynamic Functions*, NASA TT F-715.

Meleshko, S. V. (2005). *Methods of Constructing Exact Solutions of Partial Differential Equations: Mathematical and Analytic Techniques with Applications to Engineering*, New York, Springer.

Oberkampf, W. L. and T. G. Trucano (2008). Verification and validation benchmarks, *Nuclear Engineering and Design*. **238**(3), 716–743.

Oberkampf, W. L., F. G. Blottner, and D. P. Aeschliman (1995). *Methodology for Computational Fluid Dynamics Code Verification/Validation*, AIAA Paper 95–2226

(see also Oberkampf, W. L. and Blottner, F. G. (1998). Issues in computational fluid dynamics code verification and validation, *AIAA Journal*. **36**(5), 687–695).
O'Neil, P. V. (2003). *Advanced Engineering Mathematics*, 5th edn., Pacific Grove, CA, Thomson Brooks/Cole.
Panton, R. L. (2005). *Incompressible Flow*, Hoboken, NJ, Wiley.
Pelletier, D., E. Turgeon, and D. Tremblay (2004). Verification and validation of impinging round jet simulations using an adaptive FEM, *International Journal for Numerical Methods in Fluids*. **44**, 737–763.
Polyanin, A. D. (2002). *Handbook of Linear Partial Differential Equations for Engineers and Scientists*, Boca Raton, FL, Chapman and Hall/CRC.
Polyanin, A. D. and V. F. Zaitsev (2003). *Handbook of Exact Solutions for Ordinary Differential Equations*, 2nd edn., Boca Raton, FL, Chapman and Hall/CRC.
Polyanin, A. D. and V. F. Zaitsev (2004). *Handbook of Nonlinear Partial Differential Equations*, Boca Raton, FL, Chapman and Hall/CRC.
Powers, J. M. and T. D. Aslam (2006). Exact solution for multidimensional compressible reactive flow for verifying numerical algorithms, *AIAA Journal*. **44**(2), 337–344.
Powers, J. M. and D. S. Stewart (1992). Approximate solutions for oblique detonations in the hypersonic limit, *AIAA Journal*. **30**(3), 726–736.
Roache, P. J. (1998). *Verification and Validation in Computational Science and Engineering*, Albuquerque, NM, Hermosa Publishers.
Roache, P. J. (2002). Code verification by the method of manufactured solutions, *Journal of Fluids Engineering*. **124**(1), 4–10.
Roache, P. J. and S. Steinberg (1984). Symbolic manipulation and computational fluid dynamics, *AIAA Journal*. **22**(10), 1390–1394.
Roache, P. J., P. M. Knupp, S. Steinberg, and R. L. Blaine (1990). Experience with benchmark test cases for groundwater flow. In *Benchmark Test Cases for Computational Fluid Dynamics*, I. Celik and C. J. Freitas (eds.), New York, American Society of Mechanical Engineers, Fluids Engineering Division, Vol. **93**, Book No. H00598, pp. 49–56.
Roe, P. L. (1981). Approximate Riemann solvers, parameter vectors, and difference schemes, *Journal of Computational Physics*. **43**, 357–372.
Roy, C. J. (2005). Review of code and solution verification procedures for computational simulation, *Journal of Computational Physics*. **205**(1), 131–156.
Roy, C. J. and M. M. Hopkins (2003). *Discretization Error Estimates using Exact Solutions to Nearby Problems*, AIAA Paper 2003–0629.
Roy, C. J. and A. J. Sinclair (2009). On the generation of exact solutions for evaluating numerical schemes and estimating discretization error, *Journal of Computational Physics*. **228**(5), 1790–1802.
Roy, C. J., M. A. McWherter-Payne, and W. L. Oberkampf (2003). Verification and validation for laminar hypersonic flowfields Part 1: verification, *AIAA Journal*. **41**(10), 1934–1943.
Roy, C. J., C. C. Nelson, T. M. Smith, and C.C. Ober (2004). Verification of Euler/Navier–Stokes codes using the method of manufactured solutions, *International Journal for Numerical Methods in Fluids*. **44**(6), 599–620.
Roy, C. J., A. Raju, and M. M. Hopkins (2007a). Estimation of discretization errors using the method of nearby problems, *AIAA Journal*. **45**(6), 1232–1243.
Roy, C. J., E. Tendean, S. P. Veluri, R. Rifki, E. A. Luke, and S. Hebert (2007b). *Verification of RANS Turbulence Models in Loci-CHEM using the Method of Manufactured Solutions*, AIAA Paper 2007–4203.

Schetz, J. A. (1993). *Boundary Layer Analysis*, Upper Saddle River, NJ, Prentice-Hall.
Seidel, G. D. (2009). Private communication, November 6, 2009.
Shih, T. M. (1985). Procedure to debug computer programs, *International Journal for Numerical Methods in Engineering*. **21**(6), 1027–1037.
Slaughter, W. S. (2002). *The Linearized Theory of Elasticity*, Boston, MA, Birkhauser.
Steinberg, S. and P. J. Roache (1985). Symbolic manipulation and computational fluid dynamics, *Journal of Computational Physics*. **57**(2), 251–284.
Stetter, H. J. (1978). The defect correction principle and discretization methods, *Numerische Mathematik*. **29**(4), 425–443.
Tannehill, J. C., D. A. Anderson, and R. H. Pletcher (1997). *Computational Fluid Mechanics and Heat Transfer*, 2nd edn., Philadelphia, PA, Taylor and Francis.
Thompson, J. F., Z. U. A. Warsi, and C. W. Mastin (1985). *Numerical Grid Generation: Foundations and Applications*, New York, Elsevier. (www.erc.msstate.edu/publications/gridbook)
Timoshenko, S. P. and J. N. Goodier (1969). *Theory of Elasticity*, 3rd edn., New York, McGraw-Hill.
Tremblay, D., S. Etienne, and D. Pelletier (2006). *Code Verification and the Method of Manufactured Solutions for Fluid–Structure Interaction Problems*, AIAA Paper 2006–3218.
White, F. M. (2006) *Viscous Fluid Flow*, New York, McGraw-Hill.
Yanenko, N. N. (1964). Compatibility theory and methods of integrating systems of nonlinear partial differential equations, *Proceedings of the Fourth All-Union Mathematics Congress*, Vol. **2**, Leningrad, Nauka, pp. 613–621.
Zadunaisky, P. E. (1976). On the estimation of errors propagated in the numerical integration of ordinary differential equations, *Numerische Mathematik*. **27**(1), 21–39.