7
Solution verification

Solution verification addresses the question of whether a given simulation (i.e., numerical approximation) of a mathematical model is sufficiently accurate for its intended use. It includes not only the accuracy of the simulation for the case of interest, but also the accuracy of inputs to the code and any post-processing of the code results. Quantifying the numerical accuracy of scientific computing simulations is important for two primary reasons: as part of the quantification of the total uncertainty in a simulation prediction (Chapter 13) and for establishing the numerical accuracy of a simulation for model validation purposes (Chapter 12).

Most solution verification activities are focused on estimating the numerical errors in the simulation. This chapter addresses in detail round-off error, statistical sampling error, and iterative convergence error. These three numerical error sources should be sufficiently small so as not to impact the estimation of discretization error, which is discussed at length in Chapter 8. Discretization errors are those associated with the mesh resolution and quality as well as the time step chosen for unsteady problems. Round-off and discretization errors are always present in scientific computing simulations, while the presence of iterative and statistical sampling errors will depend on the application and the chosen numerical algorithms. This chapter concludes with a discussion of numerical errors and their relationship to uncertainties.

7.1 Elements of solution verification

Solution verification begins after the mathematical model has been embodied in a verified code, the initial and boundary conditions have been specified, and any other auxiliary relations have been determined. It includes the running of the code on a mesh, or series of meshes, possibly to a specified iterative convergence tolerance. Solution verification ends after all post-processing of the simulation results are completed to provide the final simulation predictions. There are thus three aspects of solution verification:

1 verification of input data,
2 verification of post-processing tools, and
3 numerical error estimation.

Verification of input and output data is particularly important when a large number of simulations are performed, such as for a matrix of simulations that vary different input conditions. Issues associated with the verification of input and output data are discussed in this section. The third aspect of solution verification, numerical error estimation, is discussed in detail in the remainder of this chapter as well as in Chapter 8.

Input data is any required information for running a scientific computing code. Common forms of input data include:

- input files describing models, submodels, and numerical algorithms,
- domain grids,
- boundary and initial conditions,
- data used for submodels (e.g., chemical species properties, reaction rates),
- information on material properties, and
- computer-aided drawing (CAD) surface geometry information.

There are various techniques that can aid in the verification of the input data. Checks for consistency between model choices should be made at the beginning of code execution. For example, the code should not allow a no-slip (viscous) wall boundary condition to be used for an inviscid simulation involving the Euler equations. For more subtle modeling issues, an expert knowledge database could be used to provide the user with warnings when a model is being used outside of its range of applicability (e.g., Stremel *et al.*, 2007). In addition, all input data used for a given simulation should be archived in an output file so that the correctness of the input data can be confirmed by post-simulation inspection if needed. Verification of input data also includes the verification of any pre-processing software that is used to generate the input data and thus the standard software engineering practices discussed in Chapter 4 should be used.

Post-processing tools are defined as any software that operates on the output from a scientific computing code. If this post-processing involves any type of numerical approximation such as discretization, integration, interpolation, etc., then the order of accuracy of these tools should be verified (e.g., by order verification), otherwise the standard software engineering practices should be followed. If possible, the post-processing steps should be automated, and then verified, to prevent common human errors such as picking the wrong solution for post-processing. If the user of the code is to perform the post-processing, then a checklist should be developed to ensure this process is done correctly.

Numerical errors occur in *every* scientific computing simulation, and thus need to be estimated in order to build confidence in the mathematical accuracy of the solution. In other words, numerical error estimation is performed to ensure that the solution produced by running the code is a sufficiently accurate approximation of the exact solution to the mathematical model. When numerical errors are found to be unacceptably large, then they should either be accounted for in the total uncertainty due to the modeling and simulation prediction (see Chapter 13) or reduced to an acceptable level by refining the mesh, reducing the iterative tolerance, etc. The four types of numerical error are:

1 round-off error,
2 statistical sampling error,
3 iterative error, and
4 discretization error.

This chapter discusses the first three numerical error sources in detail, while discretization error is addressed separately in Chapter 8.

7.2 Round-off error

Round-off errors arise due to the use of finite arithmetic on digital computers. For example, in a single-precision digital computation the following result is often obtained:

$$3.0^*(1.0/3.0) = 0.999\,9999,$$

while the true answer using infinite precision is 1.0. Round-off error can be significant for both ill-conditioned systems of equations (see Section 7.4) and time-accurate simulations. Repeated arithmetic operations will degrade the accuracy of a scientific computing simulation, and generally not just in the last significant figure of the solution. Round-off error can be reduced by using more significant digits in the computation. Although round-off error can be thought of as the truncation of a real number to fit it into computer memory, it should not be confused with truncation error which is a measure of the difference between a partial differential equation and its discrete approximation as defined in Chapter 5.

7.2.1 Floating point representation

Scientific computing applications require the processing of real numbers. Even when limiting these real numbers to lie within a certain range, say from -1 million to $+1$ million, there are infinitely many real numbers to be considered. This poses a problem for digital computers, which must store real numbers in a finite amount of computer memory. In order to fit these numbers into memory, both the precision (the number of significant figures) and the range of the exponent must be limited (Goldberg, 1991). An efficient way to do this is through an analogy with scientific notation, where both large and small numbers can be represented compactly. For example, 14 000 000 can be represented as 1.4×10^7 and 0.000 0014 represented by 1.4×10^{-6}. In digital computers, floating point numbers are more generally represented as

$$S \times B^E,$$

where S is the significand (or mantissa), B is the base (usually 2 for binary or 10 for decimal), and E is the exponent. The term floating point number comes from the fact that in this notation, the decimal point is allowed to move to represent the significand S more efficiently.

Table 7.1 *Summary of floating point number formats in IEEE Standard 754 (IEEE, 2008).*

Precision format	Total # of bits used	Bits used for significand	Bits used for exponent	Approximate number of significant digits	Exponent range (base 10)
Single	32	24	8	7	±38
Double	64	53	11	15	±308
Half[a]	16	11	5	3	±5
Extended[a]	80	64	16	18	±9864
Quadruple[a]	128	113	15	34	±4932

[a] not available in some programming languages and/or compilers

For digital computer hardware and software, the most widely-used standard for floating point numbers is the IEEE Standard 754 (IEEE, 2008). This standard addresses number format, rounding algorithms, arithmetic operations, and exception handling (division by zero, numerical overflow, numerical underflow, etc.). While the IEEE standard addresses both binary and decimal formats, nearly all software and hardware used for scientific computing employ binary storage of floating point numbers. The most commonly used formats are single precision and double precision. Additional standard formats that may or may not be available on a given computer hardware or software system are half precision, extended precision, and quadruple precision.

Single precision employs 32 bits, or four bytes, of computer memory. The significand is stored using 24 bits, one of which is used to determine the sign of the number (positive or negative). The exponent is then stored in the remaining eight bits of memory, with one bit generally used to store the sign of the exponent. The significand determines the precision of the floating point number, while the exponent determines the range of numbers that can be represented. Single precision provides approximately seven significant decimal digits and can represent positive or negative numbers with magnitudes as large as $\sim 3.4 \times 10^{38}$ and as small as $\sim 1.1 \times 10^{-38}$. For double precision numbers, 64 bits (8 bytes) of memory are used, with 53 bits assigned to the significand and 11 bits to the exponent, thus providing approximately 15 significant decimal digits. The five standard binary formats are summarized in Table 7.1. The last two columns of Table 7.1 give the maximum and minimum precision and range where, for example, single precision numbers will be represented in base 10 as:

$$1.234567 \times 10^{\pm 38}.$$

The use of single (32 bit) and double (64 bit) precision for representing floating point numbers should not be confused with 32-bit and 64-bit computer architectures. The wide availability of 64-bit processors in desktop computers beginning in 2003 was initially

Table 7.2 *Data types used to specify the different precision formats in C/C++, Fortran, and MATLAB®.*

Precision format	C/C++	Fortran 95/2003	MATLAB®
Single	float	real, real*4 (default)[a,b]	single
Double	double (default)	double precision, real*8[a,b]	double (default)
Half	n/a	[a,b]	n/a
Extended	long double[a]	[a,b]	n/a
Quadruple	long double[a]	[a,b]	n/a
Arbitrary	[c]	[c]	vpa[d]

[a] compiler dependent; [b] accessible via the "`kind`" attribute; [c] accessible via third-party libraries; [d] variable precision arithmetic (see Section 7.2.2.3)

driven by the fact that the amount of random access memory (RAM) addressable by a 32-bit integer is only 4 GB (2^{32} bytes or approximately 4.29×10^9 bytes). The 64-bit processors were developed for large databases and applications requiring more than 4 GB of addressable memory, providing a theoretical upper bound of approximately 17 billion GB (17×10^{18} bytes), although in practice the maximum addressable memory is much smaller. In addition to providing more addressable memory, 64-bit processors also may perform arithmetic faster on double precision (64 bit) floating point numbers, the most common floating point data type used in scientific computing applications. This speed-up occurs because the data path between memory and the processor is more likely to be 64 bits wide rather than 32 bits, so only one memory read instruction is needed to move a double precision floating point number from memory to the processor.

7.2.2 Specifying precision in a code

The approach for specifying the precision for floating point numbers generally depends on the programming language, the compiler, and the hardware. The data types used for specifying the precision of real numbers (variables, constants, and functions) in C/C++, Fortran, and MATLAB® are summarized in Table 7.2. In addition, the C/C++ and Fortran programming languages have the capability to employ arbitrary precision floating point numbers through the use of third-party software libraries such as the GNU Multiple Precision (GMP) arithmetic library for C/C++ (GNU, 2009) and FMLIB for Fortran (Smith, 2009). In general, when more digits of precision are employed, program execution will be slower. A more detailed discussion of procedures for specifying the floating point precision for each programming language follows.

7.2.2.1 C/C++ programming languages

The C and C++ family of programming languages requires that variable types be explicitly declared at the beginning of each routine. The available floating point types are "`float`" (single precision) and "`double`" (double precision). In addition, some compilers support the "`long double`" data type, which can refer to extended precision, quadruple precision, or simply revert back to double precision depending on the compiler. The number of bytes used for storing a floating point number can be determined in C/C++ using the "`sizeof()`" function. In addition, the number of digits of precision for floating point output can be specified using the "`cout.precision(X)`" function where `X` is an integer determining the number of significant digits to output. A short C++ code segment which provides an example of single, double, and extended precision is given below.

```
float a;
double b;
long double c;
a = 1.F/3.F;
b = 1./3.;
c = 1.L/3.L;
cout.precision(25);
cout <<''a = '';cout << a; cout << ''\n'';
cout <<''b = '';cout << b; cout << ''\n'';
cout <<''c = '';cout << c; cout << ''\n'';
cout <<''Size of a = ''; cout << sizeof(a); cout << ''\n'';
cout <<''Size of b = ''; cout << sizeof(b); cout << ''\n'';
cout <<''Size of c = ''; cout << sizeof(c); cout << ''\n'';
```

When this code is compiled with the GNU C++ compiler (available on most Linux platforms) and executed, it produces the following output:

```
a = 0.33333334326744079589843750
b = 0.33333333333333333148296163
c = 0.33333333333333333333423684
Size of a = 4
Size of b = 8
Size of c = 16
```

which indicates the expected seven digits of precision for the single precision float, 16 digits of precision for the default double precision, and 19 digits of precision for long double. Note that omitting the ".`L`" in the definition of `c` will instead produce

```
c = 0.33333333333333333148296163
Size of c = 16
```

since the initial computation of 1./3. will by default produce a double precision number (only 16 digits of precision) which is then stored in the long double variable `c`.

7.2.2.2 Fortran 95/2003 programming languages

The standards documents for modern variants of Fortran such as Fortran 95 and Fortran 2003 require that two different floating point types be supported, but do not specify what those two types must be (Chapman, 2008). While this flexibility in the standard does complicate the type specification of real numbers, it also can have some advantages in scientific computing applications that required additional precision. For most Fortran compilers, the default data type for real numbers is single precision (32 bit). To complicate matters further, some Fortran compilers use the term "single precision" to refer to 64-bit floating point numbers and "double precision" to refer to 128-bit floating point numbers.

To specify the precision for floating point numbers in an unambiguous way, Fortran 95/2003 uses the "`Kind`" attribute, where `Kind` is an integer. For single precision real numbers, `Kind` is usually equal to 1 or 4, while for double precision real number, Kind is usually equal to 2 or 8, depending on the compiler and platform. In order to specify the level of precision in a platform- and compiler-independent manner, the Fortran programming language provides the "`Selected_Real_Kind`" function. This function has arguments that allow the specification of the desired decimal precision p and the decimal exponent range r, and returns an integer equal to the smallest floating point `Kind` type that matches the requirements. If none of the available types will meet the requirements, then the function will return `-1`. For example, to print the `Kind` number for the real data type that has at least 13 decimal digits of precision and maximum exponent of ± 200, one would use:

```
write(*,*) 'Kind = ',Selected_Real_Kind(p=13,r=200)
```

Following Chapman (2008), the selection of the desired floating point precision can be placed in a Fortran module to be used in every procedure of the code. An example of such a module is presented below.

```
Module Select_Precision
Implicit None
Save
Integer, Parameter :: hlf = Selected_Real_Kind(p=2)
Integer, Parameter :: sgl = Selected_Real_Kind(p=6)
Integer, Parameter :: dbl = Selected_Real_Kind(p=14)
Integer, Parameter :: ext = Selected_Real_Kind(p=17)
Integer, Parameter :: quad = Selected_Real_Kind(p=26)
! Default precision is set on the next line
Integer, Parameter :: Prec = dbl
End Module
```

The integer parameter "`Prec`" now contains the desired `Kind` number; in this case, the `kind` number for double precision. Examples of variable declarations using this chosen precision are as follows:

```
Real(kind=Prec) :: x = 0.1_Prec
Real(kind=Prec) :: y
Real(kind=Prec) :: Pi = acos(-1.0_Prec)
y = 1.0_Prec/3.0_Prec
```

Note that double precision variables and constants should be initialized with and operated on using double precision values as shown above. When double precision is desired (i.e., `Prec = dbl`), omitting the `_Prec` suffix from the argument of the arccosine function "`acos`" in the definition of `Pi` above will cause a loss of precision when the default is single precision. The desired double precision result is

```
Pi = 3.141592653589793
```

while omitting the `_Prec` results in

```
Pi = 3.141592741012573
```

because the arccosine function operating on a single precision argument will return a single precision result. Also, the `_Prec` is needed in the assignment for `y` to give

```
y = 0.333333333333333
```

whereas omitting the `_Prec` from the numbers 1.0 and 3.0 results in:

```
y = 0.3333333432674408
```

7.2.2.3 MATLAB® Programming Language

In MATLAB®, the default data type is double precision, but it supports the standard single (32 bit) and double (64 bit) floating point precision. One can convert any data type to single precision using the "`single()`" function and to double precision using the "`double()`" function. Arithmetic operations involving both single and double precision numbers will default to the single precision data type. In addition, the Symbolic Math Toolbox™ allows the specification of arbitrary levels of precision for variables using the Variable Precision Arithmetic (`vpa`) function (MATLAB, 2009), although it may result in significantly slower execution times. Scientific computing codes in MATLAB® can be written in such a way as to take advantage of all three floating point precision capabilities using function handles (MATLAB, 2009), which allow a variable to operate in the same manner as the function it is assigned to. For example, at the beginning of a code, one could simply insert the following:

```
digits(32); % Specify # of digits of precision for vpa

Prec = @vpa; % Set to use variable precision arithmetic

% Prec = @double; % Set to use double precision

% Prec = @single; % Set to use single precision

x = Prec(1)/Prec(3) % Precision of x defined by variable Prec
```

To ensure that variables do indeed employ the desired precision, all variables should be explicitly declared at the beginning of each routine. Although variable declarations are not required in MATLAB®, as discussed in the appendix, it is a recommended programming practice. Operations involving single or double precision numbers and variable precision

numbers will inherit the variable precision type; however, any loss of precision that has occurred during the initial single or double precision computations is lost.

7.2.3 Practical guidelines for estimating round-off error

While the precision of a number is determined by the number of bits used for the significand (mantissa), the precision of a simulation also depends on all of the arithmetic and functional operations used to obtain the solution. Some scientific computing applications are prone to round-off errors. Examples include explicit time marching solutions which require small time steps but that simulate a long period of time. Another example is a "stiff" system where both large and small temporal and/or spatial scales are present. In both cases, the addition or subtraction of both large and small numbers will significantly reduce the precision of the computed solution. An extreme example of when such a loss of precision can occur is for the computation

$$\mathtt{x} = (\mathtt{1.E-9} + \mathtt{1.0}) - \mathtt{1.0}$$

which will return `x = 0.0` for single precision computations and the correct `x = 1.E-9` for double precision computations. In this case, simply rearranging the parenthesis will produce the correct result. Thus the expected magnitude of variables should be considered during the programming of arithmetic operations.

A practical approach for assessing the effects of round-off error on a simulation prediction is to run the simulation with the desired precision, then re-run the simulation with higher precision and compare the solutions. The same mesh and/or time step should be employed in both cases, and when iterative methods are used then both simulations should be iteratively converged to within machine precision, i.e., until the iterative residuals can no longer be reduced due to round-off error (see Section 7.4). This assessment will be complicated in cases where statistical sampling error is present since extremely large samples may be needed to reduce the statistical error to the point where the sensitivity to round-off error is discernable.

7.3 Statistical sampling error

In most cases, system response quantities in scientific computing are averaged quantities such as average lift and drag coefficient from an aerodynamics simulation or the vibrational modes from a structural dynamics simulation. If the simulations are steady and deterministic, then there will be no statistical sampling error. Statistical sampling error can occur in scientific computing predictions for many reasons. Certain scientific computing approaches are inherently stochastic in nature (e.g., stochastic differential equations, direct simulation Monte Carlo, lattice Boltzmann) and require time or ensemble averaging to determine mean system response quantities. If a submodel used in a scientific computing prediction is stochastic (e.g., a random walk model for Brownian motion), then a number of realizations may be needed to determine mean values. In addition, many unsteady simulations require

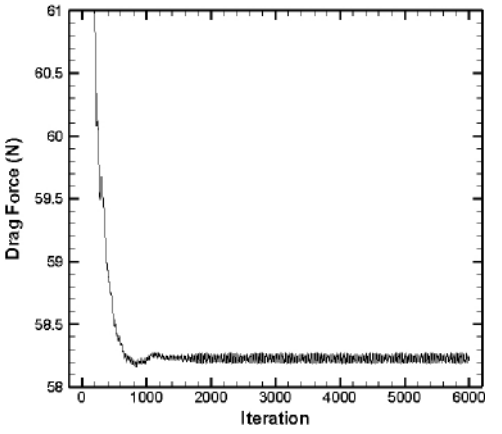

Figure 7.1 Instantaneous drag force on a simplified truck model (adapted from Veluri *et al*., 2009).

averaging over time to determine mean properties. In some cases, steady simulations which employ iterative techniques can display unsteadiness due to strong physical instabilities in the problem (Ferziger and Peric, 2002). Solution unsteadiness can also occur due to numerical sources, even when stable numerical schemes are employed. Finally, the use of non-deterministic simulations (e.g., to propagate input uncertainties to the outputs) must also deal with statistical sampling error.

7.3.1 Estimation of statistical sampling error

Statistical sampling errors can be estimated by assessing the convergence of the desired system response quantities with increased number of realizations, iterations, or time steps. Consider Figure 7.1 which gives the instantaneous drag force on a simple, reduced-scale tractor trailer geometry as a function of iteration number (Veluri *et al*., 2009). In this case, the steady-state Navier–Stokes equations are being solved with a submodel for turbulence. However, due to a small amount of vortex shedding that occurs behind the six support posts, the drag force oscillates about a steady-state value.

A number of approaches could be used to find the mean drag value. Perhaps the simplest approach is to simply inspect the instantaneous drag force plot and estimate the mean value. A slightly more sophisticated approach is to plot the running mean of the instantaneous drag force f_n at iteration n,

$$\bar{f} = \frac{1}{N}\sum_{n=1}^{N} f_n, \tag{7.1}$$

beginning at an iteration number where the large initial transients appear to have died out (approximately iteration 500 in the above example). The running average from this point could then be compared to the long time average (between iterations 2000 and 6000 and referred to as the "true mean") to estimate the statistical error, and the results of applying

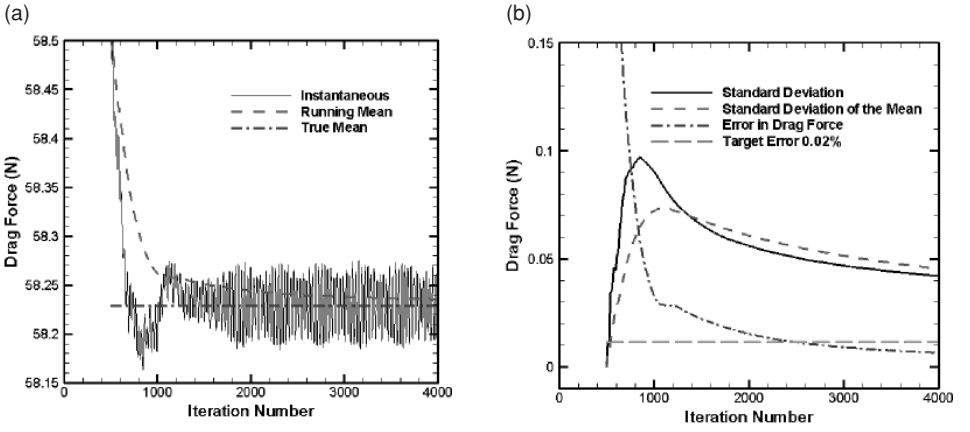

Figure 7.2 Statistical analysis of the drag force on a simplified tractor trailer: (a) instantaneous and mean values and (b) standard deviations and drag force error.

this procedure are shown in Figure 7.2a. The problem with this approach is that it requires the long time average to estimate the statistical error. Thus it is only after running for 6000 iterations that one can determine that the statistical errors in the mean value are sufficiently small (in this case 0.02% of the mean) at iteration 2500. An even better approach would be to determine a stopping criterion for the iterations (or samples) based on convergence of the standard deviation σ of the instantaneous drag force,

$$\sigma = \left[\left(\frac{1}{N} \sum_{n=1}^{N} f_n^2 \right) - \left(\bar{f} \right)^2 \right]^{1/2}, \tag{7.2}$$

or even the standard deviation of the running mean drag values. These approaches are applied in Figure 7.2b. While these approaches are problem dependent and also depend on the beginning point for the statistical analysis (in this case iteration 500), when a large number of parametric runs are to be performed they can provide a heuristic stopping criterion. For example, for a desired 0.02% statistical error in the mean value, Figure 7.2b shows that the standard deviation of the running mean values should be reduced to 0.056 N. Note that as the running mean converges, the standard deviation of the running mean will go to zero.

7.4 Iterative error

Iterative error is the difference between the current approximate solution to an equation or system of equations and the exact solution. It occurs any time an iterative (or relaxation) method is used to solve algebraic equations. In scientific computing, the most common use of iterative methods is for solving the system of algebraic equations resulting from the discretization of a mathematical model. When this system of equations is nonlinear, then a linearization step must first be used. In most cases, each of these coupled algebraic

equations contains only a few of the unknowns, and this type of system is referred to as a sparse system. For cases where the discretized equations are linear, direct solution methods may be used where the algebraic equations are solved exactly in a finite number of steps, which will recover the exact solution to the discrete equations (within round-off error). However, for nonlinear discretizations employing implicit algorithms iterative methods are required. Even in the case of linear systems, for a large number of unknowns (which generally occurs for 3-D scientific computing applications) the resulting system of algebraic equations can be solved to a sufficient level of iterative error much more efficiently using iterative methods than for direct methods.

The first iterative method for systems of equations was developed by Gauss nearly 200 years ago to handle the relatively large number of unknowns arising from his newly-developed least squares method (Hackbush, 1994). The advent of digital computers led to an explosion in the interest in iterative methods for solving the systems that arise from scientific computing applications. There are two main types of iterative method: stationary and Krylov subspace methods. Stationary iterative methods garnered much attention in the mid-1900s, whereas Krylov subspace methods were developed around the same time but were not applied to scientific computing applications until the 1970s. Current research focuses not only on developing new iterative methods, but also on combining stationary iterative methods with Krylov subspace methods, either through acceleration of the former (Hageman and Young, 1981) or preconditioning of the latter (Saad, 2003).

This section provides a broad overview of iterative methods for both single equations and systems of equations. The subject of iterative methods for systems of equations, especially those arising from the discretization of mathematical models, is an extremely broad subject. Here we provide only a limited overview of this topic, with some additional details provided for the simpler stationary iterative methods. Further information on iterative methods can be found in the textbooks by Hageman and Young (1981), Hackbush (1994), and Saad (2003) as well as on the World Wide Web by Barrett *et al*. (1994) and Saad (1996). A review of basic linear algebra and matrix analysis can be found in Golub and Van Loan (1996) and Meyer (2000).

7.4.1 Iterative methods

Iterative methods can be used to solve both single equations and systems of equations. The iteration process itself seeks to find a series of approximate solutions to the equations by starting with an initial guess. The difference between these approximate solutions and the exact solution to the equations is the iterative error. This section begins by first discussing iterative methods applicable to a single equation with a single unknown, which sometimes occurs in scientific computing. Linear systems of algebraic equations are then considered, which can arise from either linear mathematical models or from a linearization of a nonlinear mathematical model. The techniques discussed for solving these linear systems include direct solution methods, stationary methods, Krylov-subspace methods,

and hybrid methods. Examples of simple iterative methods arising from simple scientific computing applications are then given, including both linear and nonlinear cases.

7.4.1.1 Equations with a single unknown

For single equations, iterative methods are often used to find solutions (i.e., roots) when the equation cannot be solved algebraically. This situation commonly occurs for transcendental equations (i.e., those involving transcendental functions such as exponentials, logarithms, or trigonometric functions) and most polynomials of fifth order (quintic) and higher. For such equations, the simplest iterative solution approach is the direct substitution method (e.g., Chandra, 2003). Consider the equation $F(x) = 0$ which we will assume cannot be solved algebraically. This equation is rearranged, possibly by adding x to both sides, to obtain

$$x = g(x), \tag{7.3}$$

where $g(x)$ is some new function of x. Given a suitable initial guess, Eq. (7.3) can be applied in an iterative fashion as

$$x^{k+1} = g(x^k), \quad k = 1, 2, 3, \ldots, \tag{7.4}$$

where k is called the iteration number. The nonzero result obtained by inserting the approximate solution x^k into the function

$$F(x^k) = \Re^k$$

is called the *iterative residual*, not to be confused with the discrete residual discussed in Chapter 5. While extremely simple, the drawback to the direct substitution approach is that it will only converge when $|\mathrm{d}g/\mathrm{d}x| < 1$.

More sophisticated methods for iteratively solving a single algebraic equation are based on either bracketing a solution (i.e., a root of the equation) or employing information on the function gradient $\mathrm{d}F/\mathrm{d}x$. The most basic bracketing approach is the bisection method, whereupon one starts with two initial guesses x_1 and x_2 for which $F(x_1) \cdot F(x_2) < 0$. This inequality ensures a solution will be within the interval; if the product is equal to zero then one of the endpoints itself is a solution. The interval is then repeatedly subdivided using $x_3 = (x_1 + x_2)/2$, with the new interval containing the solution being the one for which the product is again less than or equal to zero. This process is repeated (i.e., iterated) until a solution is found which provides for $F(x)$ sufficiently close to zero, i.e., until convergence of the iterative residual. Convergence of this method is linear, meaning that the iterative error will reduce by a constant factor (in this case 0.5) each iteration. While this convergence rate is quite slow, the bisection method is guaranteed to converge to a solution.

There are more advanced iterative methods that incorporate function gradient information to allow faster convergence, but at the cost of a decrease in the robustness of the iteration (i.e., convergence is no longer guaranteed) (Süli and Mayers, 2006). In order of increasing efficiency, and therefore decreasing robustness, these methods can be summarized as:

1 the *false position method* – similar to bisection but employing gradient information to aid in bracketing the solution,
2 the *secant method* $x^{k+1} = x^k - \dfrac{x^k - x^{k-1}}{F(x^k) - F(x^{k-1})} F(x^k)$, and
3 *Newton's method* $x^{k+1} = x^k - \dfrac{F(x^k)}{\frac{\mathrm{d}F}{\mathrm{d}x}(x^k)}$.

The secant method is preferred for cases where the analytic gradient $\mathrm{d}F/\mathrm{d}x$ is not well-behaved or is undefined. When sufficiently close to a solution, Newton's method (also called Newton–Raphson) converges quadratically (see Section 7.4.2) while the secant method converges somewhat slower. However, far from a solution these two methods may not converge at all. For all of the above methods, convergence of the iterative procedure is usually monitored with the iterative residual

$$\Re^k = F(x^k). \tag{7.5}$$

For a more detailed discussion on the convergence of iterative methods, see Section 7.4.2.

7.4.1.2 Systems of equations

Systems of equations often arise from the discretization of mathematical models in scientific computing applications. Iterative methods are often, but not always, used when such systems of equations are large and/or nonlinear. For moderate size scientific computing applications in one or two spatial dimensions, the number of unknowns is usually between a thousand and a million. For larger 3-D applications, computations are routinely performed on spatial grids with 10–20 million cells/elements. For mathematical models that employ multiple, coupled differential equations, it is not uncommon to have five or more unknowns per grid point, resulting in around 100 million total unknowns. The largest scientific computing applications today often have up to 1 billion total unknowns.

Scientific computing applications do not always employ iterative methods. In some cases direct solution methods can be used; in addition, when explicit methods are used to solve marching problems then no iterations are involved. To understand when iterative methods may be used, and therefore where iterative error may occur, one must first have an understanding of the linearity of the mathematical model (discussed in Chapter 6), the type of numerical algorithm employed (explicit or implicit), and the mathematical character of the discrete algorithm. For explicit algorithms (e.g., Euler explicit, Runge–Kutta, Adams–Bashforth), each unknown can be computed as a function of known values, either from initial conditions or from a previous iteration/time step. Implicit algorithms require the solution of a simultaneous set of algebraic equations. While the mathematical character of the discrete algorithm often matches that of the underlying mathematical model (see Section 6.1), in some cases steady elliptic or mixed elliptic-hyperbolic problems are solved by marching in time to the steady-state solution, thus converting them into a hyperbolic system in time.

Table 7.3 *Use of iterative and direct solution methods in scientific computing based on the characteristics of the discrete model: mathematical character, linearity, and algorithm type.*

Mathematical character	Linearity	Explicit algorithm	Implicit algorithm
Initial-boundary value problem	Linear	No iteration	Iteration or direct
(hyperbolic/parabolic)	Nonlinear	No iteration	Iteration
Boundary-value problem	Linear	Iteration	Iteration or direct
(elliptic and mixed)	Nonlinear	Iteration	Iteration

Mathematical models that are elliptic in nature are called boundary-value problems. Regardless of whether the algorithms involved are explicit or implicit, boundary-value problems are usually solved using iterative methods. The only exception is when linear elliptic problems are solved implicitly, in which case direct solution methods may be used, although these tend to be expensive for large systems. Hyperbolic and parabolic mathematical models are called initial-boundary value problems and are solved via marching techniques. When explicit algorithms are used to solve initial-boundary value problems, then no iteration is needed. Initial-boundary value problems solved with implicit algorithms require iterative methods unless the problem is linear, in which case a direct solution approach can also be used. The situations where iterative methods can be used, and thus where iterative error will be present, are summarized in Table 7.3.

The remainder of this section will provide an overview of the different methods for solving linear systems of algebraic equations. The linear systems can arise from either linear or nonlinear mathematical models, with the latter requiring an additional linearization step (e.g., from a Picard iteration or Newton's method). A basic understanding of linear algebra is assumed (e.g., see Meyer, 2000).

Consider a linear system of algebraic equations of the form

$$A\vec{x} = \vec{b}, \tag{7.6}$$

where A is an $N \times N$ matrix of scalars, $\vec{b}$ a column vector of length N, $\vec{x}$ the desired solution vector of length N, and N is the total number of unknowns. In general, linear systems arising from the discretization of mathematical models will be sparse, meaning that most of the entries in the A matrix are zero. When the total number of unknowns N is sufficiently small, then direct solution techniques can be used (see Section 7.4.1.2). For large linear systems, iterative methods are much more efficient. The basic concept behind iterative methods is to first make an initial guess at the solution to Eq. (7.6) and then make successive approximations to improve the solution until satisfactory iterative convergence is reached. Because the approximate solutions will not exactly satisfy Eq. (7.6), convergence of these iterative methods is generally determined by monitoring the difference between the right and left hand sides of the equation. The iterative residual associated with the approximate solution at iteration k is thus defined as:

$$\vec{\Re}^k = \vec{b} - A\vec{x}^k. \tag{7.7}$$

Stationary iterative methods are discussed in some detail below, whereas only a broad overview of the more advanced Krylov subspace methods is given later. In addition, hybrid approaches that combine the stationary and Krylov subspace methods are also mentioned. This section concludes with examples of some stationary iterative methods applied to both linear and nonlinear mathematical models.

Direct solution methods

Direct solution methods are those which, neglecting round-off error, will produce the exact solution to the linear system given by Eq. (7.6) in a finite number of steps. In general, direct methods require on the order of N^3 operations (additions, subtractions, multiplications, and divisions) to find the solution, where N is the total number of equations to be solved. Unless the sparse structure of the linear system can be used to reduce this operation count, direct solution methods are prohibitively expensive for large scientific computing applications and iterative methods should be employed instead. The use of iterative methods is further supported by the fact that one is generally not required to drive the iterative error to zero, but only down to an acceptably small level.

An example of a direct solution method which makes efficient use of the sparse matrix structure is the *Thomas algorithm*. When the matrix A contains non-zero entries only along the three diagonals L_T, D, and U_T, e.g.,

$$A = \begin{bmatrix} D & U_\mathrm{T} & 0 & \dots & \dots & \dots & 0 \\ L_\mathrm{T} & D & U_\mathrm{T} & \ddots & & & \vdots \\ 0 & L_\mathrm{T} & D & U_\mathrm{T} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & L_\mathrm{T} & D & U_\mathrm{T} & 0 \\ \vdots & & & \ddots & L_\mathrm{T} & D & U_\mathrm{T} \\ 0 & \dots & \dots & \dots & 0 & L_\mathrm{T} & D \end{bmatrix}, \tag{7.8}$$

then a simplification of the standard direct solution technique of Gaussian elimination followed by back substitution can be used which only requires on the order of N operations. This tridiagonal matrix structure commonly arises during the discretization of 1-D mathematical models using second-order accurate methods. A similar approach called the block tridiagonal solver can be used when the L_T, D, and U_T entries themselves are square matrices. The approximate number of arithmetic operations (i.e., the operation count) for some common direct solution techniques is given in Table 7.4. Note that in Table 7.4, only the Thomas algorithm makes use of a sparse matrix structure. To get an idea of the computational time for these direct solution methods, today's modern desktop processors can perform computations at the gigaflops rate, meaning roughly 10^9 floating point operations per second. Thus, with 1 million unknowns, the Thomas algorithm would require approximately 0.008 seconds while Gaussian elimination would require 22.2 years. It is

Table 7.4 *Operation count for some direct solution techniques.*

Direct solution technique	Approximate operation count	Approximate operations for $N = 1000$	Approximate operations for $N = 1$ million
Thomas algorithm[a]	$8N$	8×10^3	8×10^6
Gaussian elimination	$\frac{2}{3}N^3$	7×10^8	7×10^{17}
LU decomposition	$\frac{2}{3}N^3$	7×10^8	7×10^{17}
Gauss–Jordan	N^3	1×10^9	1×10^{18}
Matrix inversion	$2N^3$	2×10^9	2×10^{18}
Cramer's rule	$(N+1)!$	4.0×10^{2570}	8.3×10^{5565714}

[a] for tridiagonal systems only.

clear that as the total number of unknowns N becomes large, the direct solution techniques are no longer feasible unless a sparse matrix structure can be exploited.

Stationary iterative methods

Stationary iterative methods use approximations of the full linear matrix operator A to successively improve the solution. By splitting the matrix as $A = M - N$, these methods can be written in the form

$$M\vec{x}^{k+1} = N\vec{x}^{k} + \vec{b}, \tag{7.9}$$

where k refers to the iteration number and the matrices M and N and the vector $\vec{b}$ are generally constant (i.e., they do not depend on the iteration number). In the limit as the iterations converge, $\vec{x}^{k} = \vec{x}^{k+1}$ and thus the original linear system given in Eq. (7.6) is recovered. Note that for some approximations to nonlinear systems, the matrices M and N and the vector $\vec{b}$ may in fact be updated during the iteration process. Examples of stationary iterative methods include Newton's method, Jacobi iteration, Gauss–Seidel iteration, and algebraic multigrid methods.

Left-multiplying Eq. (7.9) by the inverse of M results in

$$\vec{x}^{k+1} = M^{-1}N\vec{x}^{k} + M^{-1}\vec{b} \tag{7.10}$$

or simply

$$\vec{x}^{k+1} = G\vec{x}^{k} + M^{-1}\vec{b}, \tag{7.11}$$

where $G = M^{-1}N$ is called the iteration matrix. As will be discussed in Section 7.4.2, the convergence rate of stationary iterative methods depends on the properties of this iteration matrix.

Some of the simpler stationary iterative methods can be developed by splitting the matrix A into its diagonal D, a lower-triangular matrix L, and an upper triangular matrix U:

$$A = \begin{bmatrix} D & U & \dots & \dots & U \\ L & D & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & D & U \\ L & \dots & \dots & L & D \end{bmatrix} = \begin{bmatrix} D & 0 & \dots & \dots & 0 \\ 0 & D & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & D & 0 \\ 0 & \dots & \dots & 0 & D \end{bmatrix} + \begin{bmatrix} 0 & 0 & \dots & \dots & 0 \\ L & 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 0 & 0 \\ L & \dots & \dots & L & 0 \end{bmatrix} + \begin{bmatrix} 0 & U & \dots & \dots & U \\ 0 & 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 0 & U \\ 0 & \dots & \dots & 0 & 0 \end{bmatrix}. \tag{7.12}$$

With this splitting, the Jacobi method can be defined with $M = D$ and $N = -L - U$, resulting in the following iterative scheme:

$$D\vec{x}^{k+1} = -(L + U)\vec{x}^{k} + \vec{b}. \tag{7.13}$$

The choice of $M = D$ decouples the equations at each iteration, thus resulting in an explicit iterative scheme where each component of the solution vector x_i^{k+1} depends only on known values from the previous iteration $\vec{x}$.

The Gauss–Seidel method is similar; however it uses the most recently updated information available during the solution. For a forward sweep through the unknowns, the Gauss–Seidel method uses the splitting $M = D + L$ and $N = -U$ to give

$$(D + L)\vec{x}^{k+1} = -U\vec{x}^{k} + \vec{b}. \tag{7.14}$$

A symmetric Gauss–Seidel method would employ a forward sweep with

$$(D + L)\vec{x}^{k+1/2} = -U\vec{x}^{k} + \vec{b} \tag{7.15}$$

followed by a backward sweep with

$$(D + U)\vec{x}^{k+1} = -L\vec{x}^{k+1/2} + \vec{b}, \tag{7.16}$$

where $k = 1/2$ indicates an intermediate iteration step. All three of these Gauss–Seidel iteration methods can be performed explicitly, i.e., without solving for multiple unknowns at the same time.

Another technique that should be mentioned since it has an impact on the convergence of stationary iterative methods is over/under-relaxation. Rather than employing the stationary iterative methods as discussed previously, one instead replaces the solution components

found by applying the standard iterative method x_i^{k+1} with the following:

$$\hat{x}_i^{k+1} = x_i^k + \omega\left(x_i^{k+1} - x_i^k\right), \tag{7.17}$$

where $0 < \omega < 2$. Setting the relaxation parameter to one recovers the baseline iterative methods, whereas values greater than one or less than one result in over-relaxation or under-relaxation, respectively. When implemented with the forward Gauss–Seidel approach and with $\omega > 1$, this approach is called successive over-relaxation (SOR). The splitting for the SOR method is $M = D + \omega L$ and $N = -\omega U + (1 - \omega)D$ which can be written in matrix form as:

$$(D + \omega L)\vec{x}^{k+1} = [\omega U + (1 - \omega D)]\vec{x}^{k} + \omega\vec{b}. \tag{7.18}$$

In certain cases, the optimal relaxation factor can be estimated from properties of the iteration matrix (Saad, 1996).

Krylov subspace methods

In order to understand the basic ideas behind Krylov subspace methods, a quick review of matrix theory (Meyer, 2000) is first required. A vector space is a mathematical construct which includes definitions of vector addition and scalar multiplication. A vector subspace is any non-empty subset of a vector space. A basis for a subspace is a linearly independent set of vectors that can be linearly combined to form any vector in the subspace. The span of a set of vectors is the subspace formed by all linear combinations of those vectors, and when a set of spanning vectors are linearly independent then they also form a basis for that subspace.

Krylov subspace methods, also called nonstationary iterative methods, are popular for large sparse systems (Saad, 2003). They work by minimizing the iterative residual defined by Eq. (7.7) over the Krylov subspace. This subspace is the vector space spanned by the set of vectors formed by premultiplying an initial iterative residual vector by the matrix A to various powers. For any positive integer $m < N$ (where N is the number of unknowns), this space is defined as

$$K_m\left(A, \vec{\Re}^{k=0}\right) = \text{span}\left\{\vec{\Re}^{k=0}, A\vec{\Re}^{k=0}, A^2\vec{\Re}^{k=0}, \ldots, A^{m-1}\vec{\Re}^{k=0}\right\}. \tag{7.19}$$

An advantage of the Krylov subspace methods is that they can be implemented such that they only require matrix–vector multiplication and avoid expensive matrix–matrix multiplication.

The most commonly used Krylov subspace methods are the conjugate gradient method (for symmetric matrices only), the biconjugate gradient method, the stabilized biconjugate gradient method, and the generalized minimum residual method (GMRES) (Saad, 2003). Because the sequence of vectors given in Eq. (7.19) tends to become nearly linearly dependent, Krylov subspace methods generally employ an orthogonalization procedure such as Arnoldi orthogonalization (for the conjugate gradient and GMRES methods) or a Lanczos biorthogonalization (for the biconjugate gradient and stabilized biconjugate

gradient methods) (Saad, 2003). Krylov subspace methods are technically direct solution methods since they form a basis in the vector space which contains the solution vector $\vec{x}$ and will thus converge (within round-off error) in N steps. However, since the number of unknowns N is usually large for scientific computing applications, the process is almost always terminated early when the iterative residuals are sufficiently small, resulting in an iterative method.

Hybrid methods

Stationary and nonstationary iterative methods are often combined to take advantage of the strengths of each method. Preconditioned Krylov subspace methods premultiply the standard linear system given in Eq. (7.6) by an approximate inverse of the A matrix. For example, a Jacobi preconditioner is $P = D$ (the diagonal of matrix A), and the preconditioned system becomes

$$P^{-1}A\vec{x} = P^{-1}\vec{b}. \tag{7.20}$$

The preconditioner should be chosen such that P^{-1} is inexpensive to evaluate and the resulting system is easier to solve (Saad, 2003). Alternatively, the basic concepts behind Krylov subspace methods can be applied to the vector basis formed by linearly combining the sequence of iterative solutions $\vec{x}^{k}$ produced by a stationary iterative method. Hageman and Young (1981) refer to this process as polynomial acceleration.

Examples of iterative methods in scientific computing

This section provides some simple examples of linear systems that arise from the discretization of mathematical models. Various discretization approaches are applied to linear steady and unsteady 2-D heat conduction mathematical models. Explicit and implicit discretizations of unsteady Burgers equation are also examined, with the latter requiring a linearization step. In all cases, the types of linear systems arising from the discretization are discussed.

Linear equations While it is possible to employ direct solution techniques on linear equations, for a desired iterative error tolerance it is often more efficient to employ an iterative method, especially for large linear systems. Let us first consider the case of 2-D unsteady heat conduction with constant thermal diffusivity α:

$$\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 0. \tag{7.21}$$

This linear partial differential equation is parabolic in time and is thus an initial-boundary value problem which must be solved by a temporal marching procedure. A simple explicit finite difference discretization in Cartesian coordinates results in

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} - \alpha\left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2}\right) = 0, \tag{7.22}$$

where the n superscript denotes the temporal level and the i and j subscripts denote nodal points in the x and y directions, respectively. Applying this discrete equation over all of the interior grid nodes would result in the presence of temporal and spatial discretization error, but since no iterative method is employed, there would be no iterative error. In other words, this approach falls under the heading of a linear initial-boundary value problem discretized with an explicit algorithm in Table 7.3.

An implicit discretization could instead be chosen for Eq. (7.21), such as the Euler implicit method, which evaluates the spatial derivative terms at time level $n+1$:

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} - \alpha \left(\frac{T_{i+1,j}^{n+1} - 2T_{i,j}^{n+1} + T_{i-1,j}^{n+1}}{\Delta x^2} + \frac{T_{i,j+1}^{n+1} - 2T_{i,j}^{n+1} + T_{i,j-1}^{n+1}}{\Delta y^2} \right) = 0. \quad (7.23)$$

When this discrete equation is applied over each of the grid nodes, a linear matrix system results with a pentadiagonal structure, i.e.,

$$A = \begin{bmatrix} D & U_{\mathrm{T}} & 0 & \dots & 0 & U_{\mathrm{P}} & 0 & \dots & 0 \\ L_{\mathrm{T}} & D & U_{\mathrm{T}} & \ddots & & \ddots & U_{\mathrm{P}} & \ddots & \vdots \\ 0 & L_{\mathrm{T}} & D & U_{\mathrm{T}} & \ddots & & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \ddots & U_{\mathrm{P}} \\ 0 & & \ddots & \ddots & \ddots & \ddots & \ddots & & 0 \\ L_{\mathrm{P}} & \ddots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & & \ddots & L_{\mathrm{T}} & D & U_{\mathrm{T}} & 0 \\ \vdots & \ddots & L_{\mathrm{P}} & \ddots & & \ddots & L_{\mathrm{T}} & D & U_{\mathrm{T}} \\ 0 & \dots & 0 & L_{\mathrm{P}} & 0 & \dots & 0 & L_{\mathrm{T}} & D \end{bmatrix}. \quad (7.24)$$

While this pentadiagonal matrix could be solved directly and therefore with no iterative error, it is usually more efficient to use an iterative technique, especially for large systems. Introducing the superscript k to denote the iteration number, a line implicit discretization can be defined in the i direction such that

$$\frac{T_{i,j}^{n+1,k+1} - T_{i,j}^{n}}{\Delta t} - \alpha \left(\frac{T_{i+1,j}^{n+1,k+1} - 2T_{i,j}^{n+1,k+1} + T_{i-1,j}^{n+1,k+1}}{\Delta x^2} + \frac{T_{i,j+1}^{n+1,k} - 2T_{i,j}^{n+1,k+1} + T_{i,j-1}^{n+1,k}}{\Delta y^2} \right) = 0, \quad (7.25)$$

where the pentadiagonal contributions to the matrix structure (at $j \pm 1$) are evaluated at the known iteration level k. This discretization results in a tridiagonal system of equations for the unknowns at iteration level $k+1$ with the same structure as the matrix given in Eq. (7.8), which can again be solved efficiently with the Thomas algorithm. This discretization scheme thus requires an iterative solution to a linear system to be performed at each time step.

This line implicit iterative scheme can be written in matrix form as follows. Consider a splitting of the full matrix A from Eq. (7.23) as

$$A = L_{\mathrm{P}} + L_{\mathrm{T}} + D + U_{\mathrm{T}} + U_{\mathrm{P}}, \tag{7.26}$$

where D represents the diagonal terms (i, j), L_{T} the $(i-1, j)$ terms, L_{P} the $(i, j-1)$ terms, U_{T} the $(i+1, j)$ terms, and U_{P} the $(i, j+1)$ terms. Equation (7.25) is thus equivalent to splitting the A matrix by separating out the pentadiagonal terms, i.e.,

$$M = (L_{\mathrm{T}} + D + U_{\mathrm{T}}), \quad N = -(L_{\mathrm{P}} + U_{\mathrm{P}}). \tag{7.27}$$

The iteration scheme can then be written as

$$(L_{\mathrm{T}} + D + U_{\mathrm{T}})\vec{T}^{n+1,k+1} = -(L_{\mathrm{P}} + U_{\mathrm{P}})\vec{T}^{n+1,k} + \vec{b} \tag{7.28}$$

or simply

$$\vec{T}^{n+1,k+1} = (L_{\mathrm{T}} + D + U_{\mathrm{T}})^{-1}\left[-(L_{\mathrm{P}} + U_{\mathrm{P}})\vec{T}^{n+1,k} + \vec{b}\right] = M^{-1}\left[N\vec{T}^{n+1,k} + \vec{b}\right]. \tag{7.29}$$

As discussed in Section 7.4.2, the convergence of this iterative scheme is governed by the eigenvalues of the iteration matrix $G = M^{-1}N$.

If we instead consider the steady-state form of the 2-D heat conduction equation, then the temporal derivative drops out, resulting in the following elliptic boundary value problem:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0. \tag{7.30}$$

Discretizing with second-order accurate spatial finite differences on a Cartesian mesh results in

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} = 0, \tag{7.31}$$

which is again a pentadiagonal system of equations. This system could be solved with a direct method without producing any iterative error; however, this approach is impractical for large systems. If instead an iterative scheme such as the line-implicit approach in the i direction is employed, then the following linear system results:

$$\frac{T^{k+1}_{i+1,j} - 2T^{k+1}_{i,j} + T^{k+1}_{i-1,j}}{\Delta x^2} + \frac{T^{k}_{i,j+1} - 2T^{k+1}_{i,j} + T^{k}_{i,j-1}}{\Delta y^2} = 0. \tag{7.32}$$

This choice for the iterative scheme is called the line relaxation method and again results in a tridiagonal system of equations which can be easily solved.

Even when steady-state solutions are desired, it is common in many scientific computing disciplines to include a temporal discretization term in order to stabilize the iteration process. This occurs because the temporal terms tend to increase the diagonal dominance of the resulting iteration matrix. These temporal terms effectively serve as an under-relaxation

technique, with the under-relaxation factor approaching unity as the time step is increased. When temporal terms are included for steady-state problems, one must keep in mind that it is only the steady-state portion of the iterative residuals that should be monitored for iterative convergence.

Nonlinear equations Nonlinear systems of equations often employ the same iterative methods used for linear systems, but they first require a linearization procedure such as a Picard iteration or a Newton's method. Consider the 1-D unsteady Burgers' equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0, \tag{7.33}$$

which is parabolic in time and thus solved by temporal marching (i.e., an initial value problem). Applying a first-order accurate forward difference in time and second order accurate central differences for the spatial terms results in the following simple explicit scheme:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} - \nu\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} = 0. \tag{7.34}$$

While this explicit solution scheme is generally not used due to the severe stability restrictions and the presence of numerical oscillations in the solution, it requires no iterations and thus there is no accompanying iterative error.

If instead each of the dependent variables in the spatial terms were evaluated at time level $n+1$, the resulting implicit method would be:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^{n+1} \frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\Delta x} - \nu\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} = 0. \tag{7.35}$$

Due to the nonlinearity in the convection term, this approach results in a *nonlinear* system of algebraic equations which is not in the form of Eq. (7.6) and thus cannot be readily solved by a digital computer. For nonlinear equations, we are required to perform a linearization step and then solve the resulting tridiagonal system of equations in an iterative manner. In this example, we will use a simple Picard iteration where the u_i^{n+1} in the convective term is evaluated at the known iterative solution location k:

$$\frac{u_i^{n+1,k+1} - u_i^n}{\Delta t} + u_i^{n+1,k} \frac{u_{i+1}^{n+1,k+1} - u_{i-1}^{n+1,k+1}}{2\Delta x} - \nu\frac{u_{i+1}^{n+1,k+1} - 2u_i^{n+1,k+1} + u_{i-1}^{n+1,k+1}}{\Delta x^2} = 0. \tag{7.36}$$

The solution procedure would be as follows:

1 provide an initial guess for all u values at the first iteration ($k = 1$),
2 solve the tridiagonal system of equations for the solution at iteration $k + 1$,
3 update the nonlinear term based on the new iterative solution, and
4 repeat the process until iterative residual convergence is achieved.

Thus for nonlinear equations, even when direct methods are used to solve the resulting linear system, the solution approach will still be iterative due to the linearization step (recall Table 7.3).

7.4.2 Iterative convergence

This section provides a brief discussion of the convergence of stationary iterative methods. See Saad (2003) for a discussion of the convergence of Krylov subspace methods. As mentioned previously, the iterative error is defined as the difference between the current numerical solution at iteration k and the exact solution to the discrete equations. For some global solution property f, we can thus define the iterative error at iteration k as

$$\varepsilon_h^k = f_h^k - f_h \tag{7.37}$$

where h refers to the discrete equation on a mesh with discretization parameters (Δx, Δy, Δt, etc.) represented by h, f_h^k is the current iterative solution, and f_h is the exact solution to the discrete equations, not to be confused with the exact solution to the mathematical model. In some cases, we might instead be concerned with the iterative error in the entire solution over the domain (i.e., the dependent variables in the mathematical model). In this case, the iterative error for *each* dependent variable u should be measured as a norm over the domain, e.g.,

$$\varepsilon_h^k = \left\| \vec{u}_h^k - \vec{u}_h \right\|, \tag{7.38}$$

where the vector signs denote a column vector composed of a single dependent variable at each discrete location in the domain. When multiple dependent variables are included (e.g., x-velocity, y-velocity, and pressure in a fluid dynamics simulation), then these should be monitored separately. The common norms employed include discrete L_1, L_2, and L_∞ norms (see Chapter 5 for norm definitions). Note that the standard texts on iterative methods do not differentiate between the different dependent variables, and thus lump all dependent variables at all domain locations into the unknown variable vector $\vec{x}$. We will switch between the scientific computing notation $\vec{u}$ and the linear systems notation $\vec{x}$ as necessary.

For stationary iterative methods applied to linear systems, iterative convergence is governed by the eigenvalues of the iteration matrix. Recall that M and N come from the splitting of the full matrix A in Eq. (7.6). The iteration matrix G comes from left-multiplying Eq. (7.9) by M^{-1}, i.e.,

$$\vec{x}^{k+1} = M^{-1}N\vec{x}^k + M^{-1}\vec{b}, \tag{7.39}$$

and can thus be expressed as $G = M^{-1}N$. Specifically, convergence is related to the eigenvalue of largest magnitude λ_G for the iteration matrix G. The absolute value of this eigenvalue is also called the *spectral radius* of the matrix,

$$\rho(G) = |\lambda_G|. \tag{7.40}$$

Convergence of the iterative method requires that the spectral radius be less than one, and the closer to unity the slower the convergence rate of the method (Golub and Van Loan, 1996). A crude approximation of the spectral radius of a matrix (Meyer, 2000) can be found by using the fact that the spectral radius is bounded from above by any valid matrix norm, i.e.,

$$\rho(G) \leq \|G\|. \tag{7.41}$$

For the Jacobi and Gauss–Seidel methods, convergence is guaranteed if the matrix A is strictly diagonally dominant, meaning that the magnitude of the diagonal entry of each row is larger than the sum of the magnitudes of other entries in the row, i.e.,

$$|A_{ii}| > \sum_{i \neq j} |A_{ij}|. \tag{7.42}$$

For linear problems, when the maximum eigenvalue λ_G of the iteration matrix G is real, then the limiting iterative convergence behavior will be monotone. When λ_G is complex, however, the limiting iterative convergence behavior will generally be oscillatory (Ferziger and Peric, 1996). For nonlinear problems, the linearized system is often not solved to convergence, but only solved for a few iterations (sometimes as few as one) before the nonlinear terms are updated. Iterative convergence of the nonlinear system is much more difficult to assess and, like equations with a single unknown, is often related to proximity of the initial guess to the converged solution. See Kelley (1995) for a discussion of iterative methods for nonlinear systems.

7.4.2.1 Types of iterative convergence

This section describes the different types of iterative convergence that may be observed. For simple scientific computing applications, iterative convergence is often found to be monotone or oscillatory. For more complex applications, a more general iterative convergence behavior is often found.

Monotone convergence

Monotone iterative convergence occurs when subsequent iterative solutions approach the exact solution to the discrete equations in a monotone fashion with no local minima or maxima in the iterative error plotted as a function of the iteration number. Monotone convergence is usually either linear or quadratic. Linear convergence occurs when the ratio of iterative error from one iteration to the next is constant, i.e.,

$$\varepsilon_h^{k+1} = C\varepsilon_h^k, \quad \text{where } C < 1, \tag{7.43}$$

thus the iterative error is reduced by the same fraction C for every iteration. Many scientific computing codes converge at best at a linear rate. Quadratic convergence occurs when the iterative errors drop with the square of the iterative errors from the previous iterations

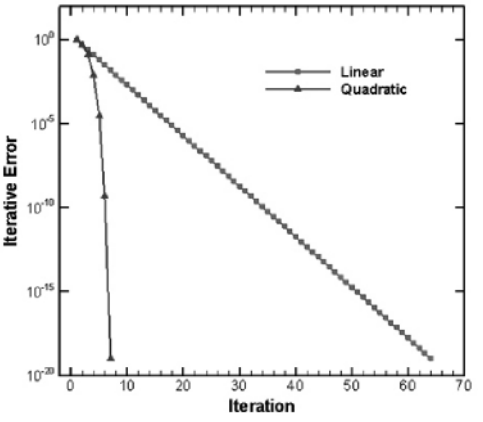

Figure 7.3 Example of monotone iterative convergence showing both linear and quadratic behavior.

(Süli and Mayers, 2006), i.e.,

$$\left|\varepsilon_h^{k+1}\right| = C\left(\varepsilon_h^k\right)^2, \quad \text{where } C < 1, \tag{7.44}$$

For nonlinear governing equations, quadratic convergence can be achieved for advanced iterative techniques (e.g., full Newton's method) as long as the current iterative solution is in the vicinity of the exact solution to the discrete equations (Kelley, 1995). Examples of linear and quadratic convergence are given in Figure 7.3, which shows the iterative error on a logarithmic scale as a function of the number of iterations. Although clearly desirable, quadratic convergence is difficult to obtain for practical scientific computing problems. In some cases, monotone iterative convergence can be accelerated by using over-relaxation ($\omega > 1$).

Oscillatory convergence

Oscillatory iterative convergence occurs when the iterative solution converges in an oscillatory manner to the exact solution to the discrete equations. An example of oscillatory iterative convergence is given in Figure 7.4, which shows the iterative convergence error in a general solution functional reducing in an oscillatory but convergent manner. Oscillatory convergence occurs when the largest eigenvalues of the iteration matrix $G = M^{-1}N$ are complex (Ferziger and Peric, 1996). In some cases, oscillatory iterative convergence can be accelerated by using under-relaxation ($\omega < 1$).

General convergence

General iterative convergence occurs when the iterative solutions converge in a complicated manner and exhibit a mixture of monotone and oscillatory convergence. An example of general convergence is given in Figure 7.5, which shows the iteration history for the yaw force on a missile geometry from an inviscid computational fluid dynamics simulation. This complex type of iterative convergence history is commonplace in practical scientific

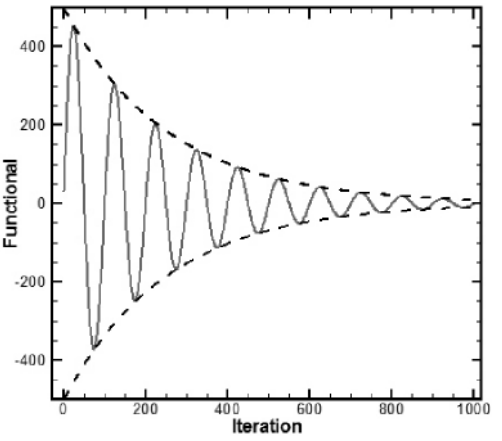

Figure 7.4 Example of oscillatory iterative convergence of a solution functional versus iteration number.

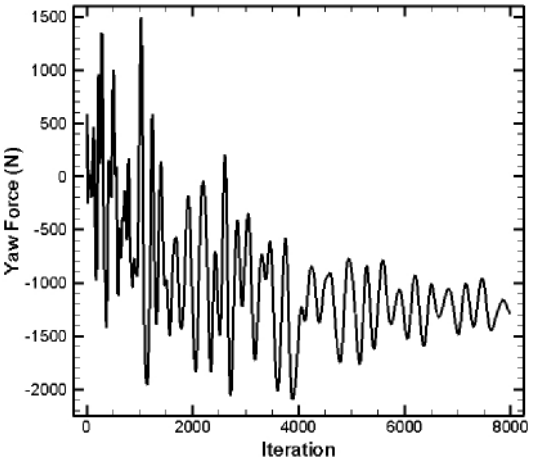

Figure 7.5 Example of general iterative convergence for the yaw force in an inviscid computational fluid dynamics simulation of the flow over a missile (adapted from Roy *et al*., 2007).

computing simulations. The source of the oscillations for general convergence may be physical (e.g., strong physical instabilities or unsteadiness in a problem) or numerical.

7.4.2.2 Iterative convergence criteria

Two common methods for assessing iterative convergence are discussed in this section. The first examines the difference in solutions between successive iterates and can be misleading. The second approach is to examine the iterative residual and provides a direct measure of how well the discrete equations are solved by the current iterative solution.

Difference between iterates

One method commonly used to report the convergence of iterative methods is in terms of the difference (either absolute or relative) between successive iterates, e.g.,

$$f_h^{k+1} - f_h^k \quad \text{or} \quad \frac{f_h^{k+1} - f_h^k}{f_h^k}.$$

However, these approaches can be extremely misleading when convergence is slow or stalled (Ferziger and Peric, 2002). It is strongly recommended that these convergence criteria not be used.

Iterative residuals

A much better approach is to evaluate the iterative residuals of the discrete equations at each step in the iteration process. For linear systems, this iterative residual is given by Eq. (7.7). For scientific computing applications, this iterative residual is found by plugging the current iterative solution into the discretized form of the equations. Recall from Chapter 5 that the discrete equations can be written in the form

$$L_h(u_h) = 0, \tag{7.45}$$

where L_h is the linear or nonlinear discrete operator and u_h is the exact solution to the discrete equations. The iterative residual is found by plugging the current iterative solution u_h^{k+1} into Eq. (7.45), i.e.,

$$\Re_h^{k+1} = L_h(u_h^{k+1}), \tag{7.46}$$

where $\Re_h^{k+1} \to 0$ as $u_h^{k+1} \to u_h$. Note that although it appears to be evaluated in a similar fashion, this iterative residual is completely different from the discrete and continuous residuals discussed in Chapter 5, and this is a source of much confusion in scientific computing.

The form of the iterative residuals for the examples presented in Section 7.4.1.2 will now be given. For the 2-D steady-state heat conduction example using line relaxation given in Eq. (7.32), one would simple evaluate the iterative residual $\Re$ as:

$$\Re_{i,j}^{k+1} = \frac{T_{i+1,j}^{k+1} - 2T_{i,j}^{k+1} + T_{i-1,j}^{k+1}}{\Delta x^2} + \frac{T_{i,j+1}^{k+1} - 2T_{i,j}^{k+1} + T_{i,j-1}^{k+1}}{\Delta y^2}. \tag{7.47}$$

The residuals are generally monitored using discrete norms over the mesh points, as discussed in Chapter 5. The residual norms will approach zero as the iterations converge, thus ensuring that the numerical solution satisfies the discrete equations within round-off

error; however, in practice, the iterations are generally terminated when the iterative error is considered sufficiently small (see Section 7.4.3). For the unsteady 2-D heat conduction example given by Eq. (7.25), the unsteady iterative residual must be used:

$$\Re_{i,j}^{n+1,k+1} = \frac{T_{i,j}^{n+1,k+1} - T_{i,j}^{n}}{\Delta t} - \alpha\left(\frac{T_{i+1,j}^{n+1,k+1} - 2T_{i,j}^{n+1,k+1} + T_{i-1,j}^{n+1,k+1}}{\Delta x^2} + \frac{T_{i,j+1}^{n+1,k+1} - 2T_{i,j}^{n+1,k+1} + T_{i,j-1}^{n+1,k+1}}{\Delta y^2}\right). \tag{7.48}$$

However, in cases where the unsteady form of the governing equations is used to obtain solutions to the steady-state governing equations, the steady-state iterative residual (given by Eq. (7.47) in this case, either with or without the diffusivity α) must be used.

For the implicit discretization of the unsteady Burgers equation given by Eq. (7.36), the form of the iterative residual to be converged at each time step n is

$$\Re_{i}^{n+1,k+1} = \frac{u_i^{n+1,k+1} - u_i^n}{\Delta t} + u_i^{n+1,k+1}\frac{u_{i+1}^{n+1,k+1} - u_{i-1}^{n+1,k+1}}{2\Delta x} - \nu\frac{u_{i+1}^{n+1,k+1} - 2u_i^{n+1,k+1} + u_{i-1}^{n+1,k+1}}{\Delta x^2}, \tag{7.49}$$

where the velocity modifying the convective term must be evaluated at iteration $k+1$. Again, when the unsteady formulation is used to obtain steady-state solutions, then the iterative residual that needs to be monitored is simply the steady-state residual given by

$$\Re_{i}^{k+1} = u_i^{k+1}\frac{u_{i+1}^{k+1} - u_{i-1}^{k+1}}{2\Delta x} - \nu\frac{u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1}}{\Delta x^2}. \tag{7.50}$$

7.4.3 Iterative error estimation

While monitoring the iterative residuals often serves as an adequate indication as to whether iterative convergence has been achieved, it does not by itself provide any guidance as to the size of the iterative error in the solution quantities of interest. In this section, we discuss approaches for estimating the iterative error in general solution quantities. These approaches can be used to obtain a quantitative estimate of the iterative error. If the numerical solution is to be used in a code order verification study (Chapter 5) or a discretization error study (Chapter 8), the recommended iterative error levels are 100 times smaller than the discretization error levels so as not to adversely impact the discretization error evaluation/estimation process. For cases where the solution will not be used for discretization error evaluation/estimation (e.g., parametric studies where a large number of simulations are required), then larger iterative error levels can often be tolerated.

7.4.3.1 Machine zero method

Recall that the iterative error is defined as the difference between the current iterative solution and the exact solution to the discrete equations. We can get a very good estimate of the iterative error by first converging the solution down to machine zero. The iterative error in a solution function f at iteration k is then approximated as

$$\varepsilon_h^k = f_h^k - f_h \cong f_h^k - f_h^{k\to\infty}. \tag{7.51}$$

where $f_h^{k\to\infty}$ is the machine zero solution, i.e., the iterative solution found in the limit as the number of iterations goes to infinity. This location can be identified as the point in the iteration history where the residuals no longer reduce but instead display seemingly random oscillations about some value. This leveling off generally occurs near residual reductions of approximately seven orders of magnitude for single precision computations and 15 orders of magnitude for double precision computations (see Section 7.2). Due to the expense of converging complex scientific computing codes to machine zero, this approach is usually only practical when applied to a small subset of cases in a large parametric study.

7.4.3.2 Local convergence rate

Rather than converging the iterative residuals to machine zero, a more efficient approach is to use the current iterative solution and its neighboring iterates to estimate the iterative error during the solution process. This approach requires some knowledge of the type of iterative convergence displayed by the numerical solution. These iterative error estimates are most useful for larger scientific computing applications where the convergence is slow.

Monotone iterative convergence

Eigenvalue method For scientific computing analyses where iterative convergence is monotone, the convergence rate is often linear. For linear monotone convergence, the maximum eigenvalue λ_G of the iteration matrix $G = M^{-1}N$ is real. It can be shown (Ferziger, 1988; Golub and Van Loan, 1996) that the iterative error in any general solution quantity f can be approximated by

$$\varepsilon_h^k \cong \frac{f_h^{k+1} - f_h^k}{\lambda_G - 1}, \tag{7.52}$$

where λ_G can be approximated by

$$\lambda_G \cong \frac{\left|f_h^{k+1} - f_h^k\right|}{\left|f_h^k - f_h^{k-1}\right|}. \tag{7.53}$$

Solving instead for the *estimate* of the exact solution to the discrete equations $\hat{f}_h$, one obtains:

$$\hat{f}_h = \frac{f^{k+1} - \lambda_G f^k}{1 - \lambda_G}. \tag{7.54}$$

See Ferziger (1988) or Golub and Van Loan (1996) for additional details.

Blottner's method A similar approach was developed by Blottner for cases where linear monotone convergence is observed in the iterative residuals and/or the iterative error (Roy and Blottner, 2001). When linear monotone convergence occurs, iterative solutions at three different iteration levels can be used to estimate the iterative error. Blottner's approach can be summarized as follows. Recall the iterative error in any quantity f from Eq. (7.37),

$$\varepsilon_h^k = f_h^k - f_h, \tag{7.55}$$

where f_h is the exact solution to the discrete equations. If convergence is linear monotonic, then the error will decrease exponentially as a function of iteration number,

$$\varepsilon_h^k = \alpha e^{-\beta k}, \tag{7.56}$$

where α and β are constants which are independent of the iteration number. Equations (7.55) and (7.56) can be combined and rewritten as

$$\beta k = \ln \alpha - \ln\left(f_h^k - f_h\right). \tag{7.57}$$

Equation (7.57) is then evaluated at three different iteration levels, for example, $(k-1)$, k, and $(k+1)$, and these relationships are then used to eliminate α and obtain

$$\begin{aligned} \beta\left(k-(k-1)\right) &= \ln\left[\left(f_h^{k-1} - f_h\right)/\left(f_h^k - f_h\right)\right], \\ \beta\left(k+1-(k)\right) &= \ln\left[\left(f_h^{k} - f_h\right)/\left(f_h^{k+1} - f_h\right)\right]. \end{aligned} \tag{7.58}$$

Assuming the increments between iterations are equal (as in the current example), then the left hand sides of Eq. (7.58) are equivalent. Equating the right hand sides gives

$$\ln\left[\left(f_h^{k-1} - f_h\right)/\left(f_h^k - f_h\right)\right] = \ln\left[\left(f_h^{k} - f_h\right)/\left(f_h^{k+1} - f_h\right)\right]$$

or simply

$$\left(f_h^{k-1} - f_h\right)/\left(f_h^k - f_h\right) = \left(f_h^{k} - f_h\right)/\left(f_h^{k+1} - f_h\right). \tag{7.59}$$

An estimate of the exact solution to the discretized equations $\hat{f}_h$ is then given by

$$\hat{f}_h = \frac{f_h^k - \Lambda^k f_h^{k-1}}{1-\Lambda^k}, \quad \text{where } \Lambda^k = \frac{f_h^{k+1} - f_h^k}{f_h^k - f_h^{k-1}}. \tag{7.60}$$

This final result is closely related to the eigenvalue approach given above in Eq. (7.54), where here Λ^k plays the role of the estimated eigenvalues with the largest magnitude λ_G. When convergence is slow, it is often advantageous to use nonconsecutive iterations (e.g., $k-10$, k, and $k+10$) due to the relative importance of round-off error when the iterative solutions change very little from iteration to iteration.

Blottner's method has been used to estimate the iterative error for a Navier–Stokes fluids simulation by Roy and Blottner (2003). They examined the iterative error in the surface shear stress for the hypersonic flow over flat plate aligned with the flow. Figure 7.6a shows

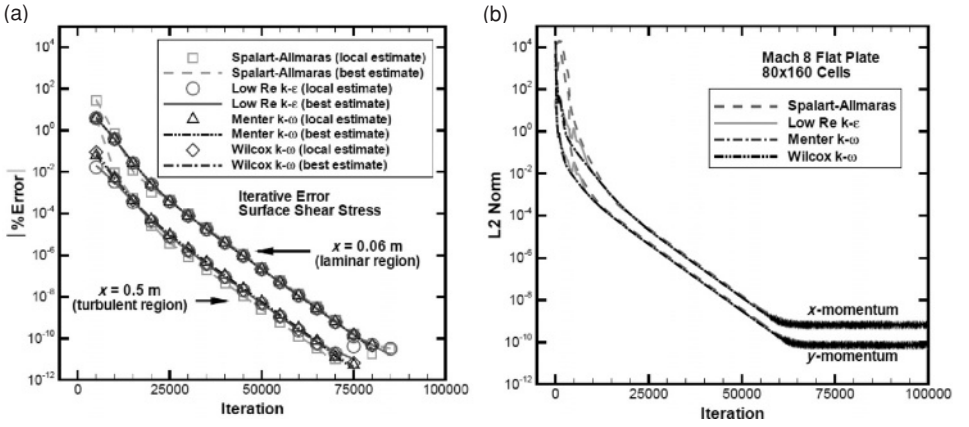

Figure 7.6 Mach 8 flow over a flat plate using the Navier–Stokes equations: (a) iterative error estimates from Blottner's method (local estimate) along with the machine zero method (best estimate) and (b) L_2 norms of the iterative residuals for the x- and y-momentum equations (reproduced from Roy and Blottner, 2003).

the above iterative error estimator applied to two different regions of the plate: one with laminar flow and the other with turbulent flow. The symbols labeled "local estimate" are from Blottner's iterative error estimation procedure, while the lines labeled "best estimate" were found from the machine zero method discussed in Section 7.4.3.1. Machine zero is reached at an iterative residual reduction of approximately 14 orders of magnitude for these double precision computations, as shown in Figure 7.6b. Blottner's method agrees quite well with the machine zero method and has the advantage of allowing early termination of the iterations once the iterative error is sufficiently small. However, Blottner's method should only be applied for cases where the iterative convergence is known (or demonstrated) to be monotone and linear.

Oscillatory iterative convergence

Ferziger and Peric (1996) have developed an iterative error estimator which addresses oscillatory iterative convergence. Recall that in the eigenvalue approach for monotone solutions (discussed in the last section), the eigenvalue of the iteration matrix with the largest magnitude λ_G is used to estimate the iterative error. For oscillatory convergence, the eigenvalues with the largest magnitude are complex (and occur as conjugate pairs) and can also be used to estimate the iterative error. See Ferziger and Peric (1996) and the cited references therein for details.

7.4.4 Relation between iterative residuals and iterative error

A number of studies have shown that the iterative residual reduction tracks extremely well with actual iterative error for a wide range of linear and nonlinear problems in scientific

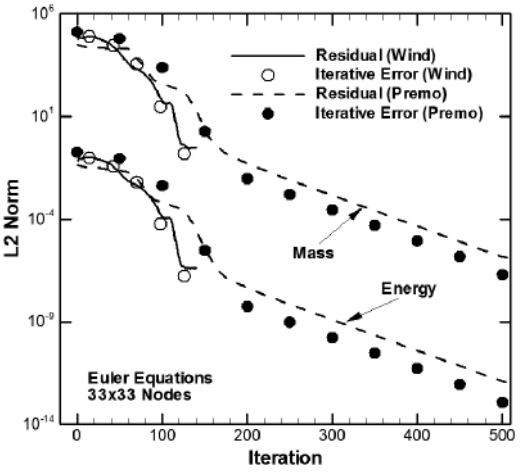

Figure 7.7 Discrete L_2 norms of the iterative convergence error and iterative residuals for manufactured solutions of the Euler equations (reproduced from Roy *et al*., 2004).

computing (e.g., Ferziger and Peric, 2002; Roy and Blottner, 2003). This was demonstrated earlier in Figure 7.6 for a compressible Navier–Stokes example, which shows that the slope of the iterative error (Figure 7.6a) corresponds to the slope of the iterative residuals (Figure 7.6b), and thus they converge at the same rate. This observation supports the common practice of assessing iterative convergence indirectly by examining the iterative residuals. The key is then to find the appropriate scaling factor for a given class of problems to relate the norms of the iterative residuals to the iterative error in the quantity of interest.

Another example showing the relationship between iterative error and iterative residuals for the Euler equations is presented in Figure 7.7. These solutions were obtained for a manufactured solution computation with both a double precision code (Premo) and a single precision code (Wind) (Roy *et al*., 2004). The system response quantity of interest is the full flowfield solution, so L2 norms of the iterative error over the domain for the mass and total energy per unit volume are found from the machine zero method (Section 7.4.3.1). These iterative error norms are then compared with L2 norms of the iterative residuals of the mass and total energy conservation equations. For both codes, the iterative residual norms closely follow the actual iterative error norms from the machine zero method. Note that, for these cases, the iterative residuals were scaled by a constant factor so as to line up more closely with the iterative errors, which will not affect the slopes. This scaling is required because the iterative residuals have the same general behavior, but not the same magnitude, as the iterative error.

7.4.5 Practical approach for estimating iterative error

For practical scientific computing problems, it may be difficult to apply the iterative error estimators discussed in the Section 7.4.3. This is particularly true when a large number

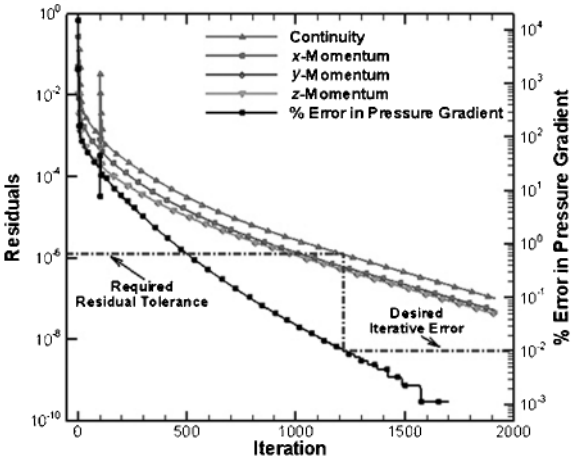

Figure 7.8 Norms of the iterative residuals (left axis) and percentage error in pressure gradient (right axis) for laminar flow through a packed bed (reproduced from Duggirala *et al.*, 2008).

of computational predictions must be made (e.g., for a parametric study or for conducting nondeterministic simulations). Most scientific codes monitor iterative convergence by examining norms of the iterative residuals. Since the iterative residual norms have been shown to follow closely with actual iterative errors for many problems, a small number of computations should be sufficient to determine how the iterative errors in the system response quantity scale with the iterative residuals for the cases of interest.

An example of this procedure is given in Figure 7.8 for laminar viscous flow through a packed bed of spherical particles (Duggirala *et al.*, 2008). The quantity of interest is the average pressure gradient across the bed, and the desired iterative error level in the pressure gradient is 0.01%. The iterative residuals in the conservation of mass (continuity) and conservation of momentum equations are first converged to 10^{-7}, then the value of the pressure gradient at this point is taken as an approximation to the exact solution to the discrete equations $\hat{f}_h$. The iterative error for all previous iterations is then approximated using this estimate of f_h. Figure 7.8 shows that for the desired iterative error level in the pressure gradient of 0.01%, the iterative residual norms should be converged down to approximately 10^{-6}. Simulations for similar problems can be expected to require approximately the same level of iterative residual convergence in order to achieve the desired iterative error tolerance in the pressure gradient.

7.5 Numerical error versus numerical uncertainty

All of the sources of numerical error that have been discussed in this chapter are correctly classified as errors. If the value (including the sign) of these errors is known with a high degree of certainty, then these errors can either be removed from the numerical solutions (a process similar to the removal of well-characterized bias errors from an experimental

measurement) or, if sufficiently small, simply neglected. In cases where numerical errors cannot be estimated with a great deal of reliability (as is often the case in scientific computing), while they are still truly errors, our knowledge of these errors is uncertain. Thus when numerical error estimates are not reliable, they can correctly be treated as epistemic uncertainties. That is, they are uncertainties due to a lack of knowledge of the true value of the error.

When the errors and uncertainties due to all numerical error sources have been estimated, a quantitative assessment of the total numerical uncertainty budget can provide guidance as to the efficient allocation of resources. For example, if the total numerical uncertainty budget is estimated for a given class of problems, and the largest contributor to the total uncertainty is the spatial discretization error, then this suggests that an efficient use of resources is to focus on refining the spatial mesh. Just as important, this analysis will also suggest areas where resources should not be focused (e.g., in reducing round-off errors).

7.6 References

Barrett, R., M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van Der Vorst (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, (www.netlib.org/linalg/html_templates/Templates.html).

Chandra, S. (2003). *Computer Applications in Physics: with Fortran and Basic*, Pangbourne, UK, Alpha Science International, Ltd.

Chapman, S. J. (2008). *Fortran 95/2003 for Scientists and Engineers*, 3rd edn., McGraw-Hill, New York.

Duggirala, R., C. J. Roy, S. M. Saeidi, J. Khodadadi, D. Cahela, and B. Tatarchuck (2008). Pressure drop predictions for microfibrous flows using CFD, *Journal of Fluids Engineering*. **130**(DOI: 10.1115/1.2948363).

Ferziger, J. H. (1988). A note on numerical accuracy, *International Journal for Numerical Methods in Fluids*. **8**, 995–996.

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.

GNU (2009). GNU Multiple Precision Arithmetic Library, gmplib.org/.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic, *Computing Surveys*. **23**(1), 91–124 (docs.sun.com/source/806–3568/ncg_goldberg.html).

Golub, G. H. and C. F. Van Loan (1996). *Matrix Computations*, 3rd edn., Baltimore, The Johns Hopkins University Press.

Hackbush, W. (1994). *Iterative Solution of Large Sparse Systems of Equations*, New York, Springer-Verlag.

Hageman, L. A. and D. M. Young (1981). *Applied Iterative Methods*, London, Academic Press.

IEEE (2008). *IEEE Standard for Floating-Point Arithmetic*, New York, Microprocessor Standards Committee, Institute of Electrical and Electronics Engineers Computer Society.

Kelley, C. T. (1995). *Iterative Methods for Linear and Nonlinear Equations*, SIAM Frontiers in Applied Mathematics Series, Philadelphia, Society for Industrial and Applied Mathematics.

MATLAB (2009). *MATLAB® 7 Programming Fundamentals*, Revised for Version 7.8 (Release 2009a), Natick, The MathWorks, Inc.

Meyer, C. D. (2000). *Matrix Analysis and Applied Linear Algebra*, Philadelphia, Society for Industrial and Applied Mathematics.

Roy, C. J. and F. G. Blottner (2001). Assessment of one- and two-equation turbulence models for hypersonic transitional flows, *Journal of Spacecraft and Rockets*. **38**(5), 699–710 (see also Roy, C. J. and F. G. Blottner (2000). *Assessment of One- and Two-Equation Turbulence Models for Hypersonic Transitional Flows*, AIAA Paper 2000–0132).

Roy, C. J. and F. G. Blottner (2003). Methodology for turbulence model validation: application to hypersonic flows, *Journal of Spacecraft and Rockets*. **40**(3), 313–325.

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., C. J. Heintzelman, and S. J. Roberts (2007). *Estimation of Numerical Error for 3D Inviscid Flows on Cartesian Grids*, AIAA Paper 2007–0102.

Saad, Y. (1996). *Iterative Methods for Sparse Linear Systems*, 1st edn., Boston, PWS Publishing (updated manuscript www-users.cs.umn.edu/∼saad/books.html).

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*, 2nd edn., Philadelphia, PA, Society for Industrial and Applied Mathematics.

Smith, D. M. (2009). FMLIB, Fortran Library, myweb.lmu.edu/dmsmith/FMLIB.html.

Stremel, P. M., M. R. Mendenhall, and M. C. Hegedus (2007). *BPX – A Best Practices Expert System for CFD*, AIAA Paper 2007–974.

Süli, E. and D. F. Mayers (2006). *An Introduction to Numerical Analysis*. Cambridge, Cambridge University Press.

Veluri, S. P., C. J. Roy, A. Ahmed, R. Rifki, J. C. Worley, and B. Recktenwald (2009). Joint computational/experimental aerodynamic study of a simplified tractor/trailer geometry, *Journal of Fluids Engineering*. **131**(8) (DOI: 10.1115/1.3155995).