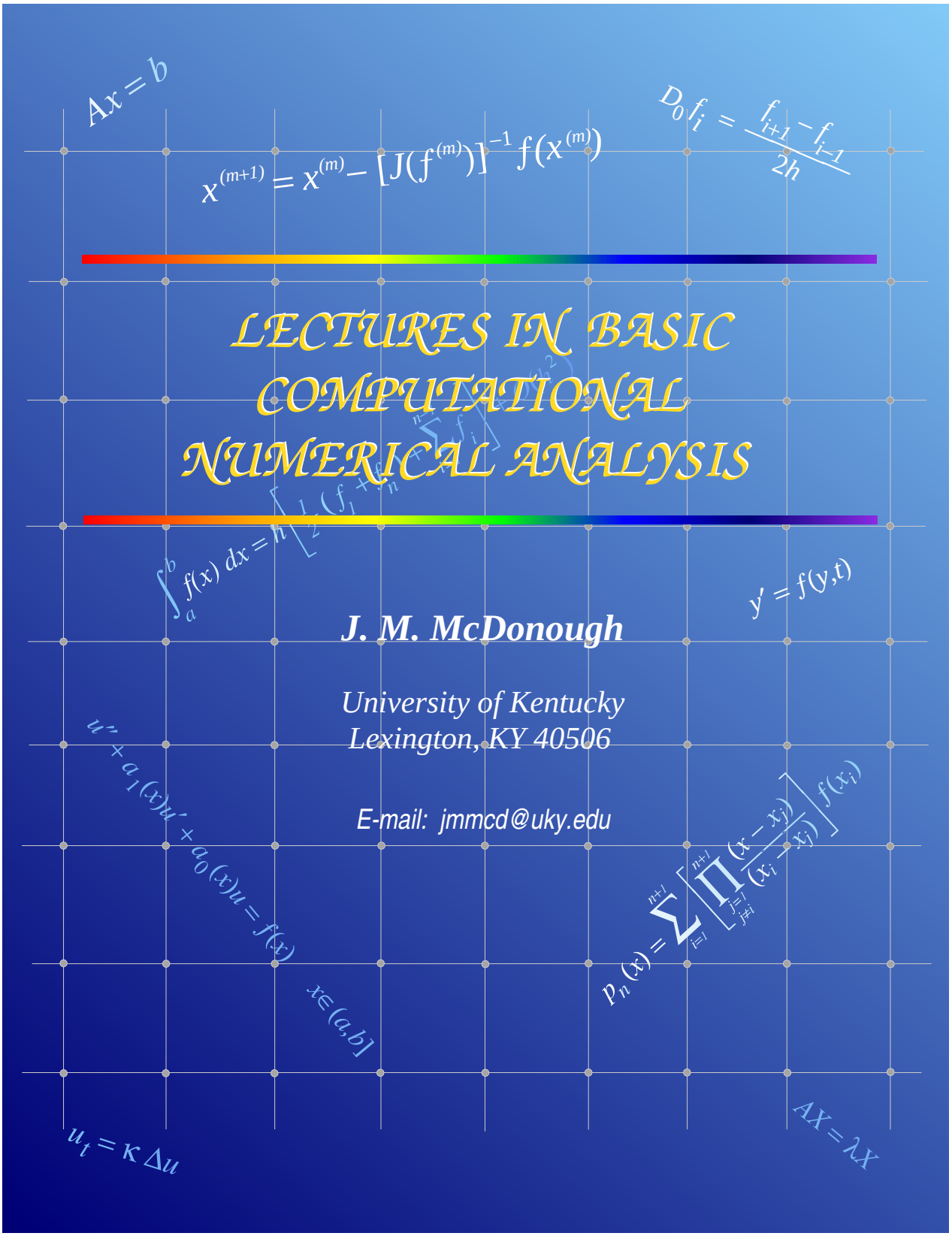

$Ax = b$
$x^{(m+1)} = x^{(m)} - [J(f^{(m)})]^{-1} f(x^{(m)})$
$D_0 f_i = \frac{f_{i+1} - f_{i-1}}{2h}$
LECTURES IN BASIC COMPUTATIONAL NUMERICAL ANALYSIS
$\int_a^b f(x)\,dx = h\left[\frac{1}{2}(f_1 + f_n) + \sum_{i=2}^{n-1} f_i\right] - \frac{(b-a)h^2}{12} f''(\xi)$
$y' = f(y,t)$
J. M. McDonough
University of Kentucky
Lexington, KY 40506
E-mail: jmmcd@uky.edu
$u'' + a_1(x)u' + a_0(x)u = f(x)$
$x \in (a,b]$
$p_n(x) = \sum_{i=1}^{n+1} \left[\prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{(x - x_j)}{(x_i - x_j)}\right] f(x_i)$
$u_t = \kappa \Delta u$
$AX = \lambda X$

LECTURES IN BASIC COMPUTATIONAL NUMERICAL ANALYSIS

J. M. McDonough

Departments of Mechanical Engineering and Mathematics
University of Kentucky

Contents

List of Figures

Chapter 1

Numerical Linear Algebra

From a practical standpoint numerical linear algebra is without a doubt the single most important topic in numerical analysis. Nearly all other problems ultimately can be reduced to problems in numerical linear algebra; *e.g.*, solution of systems of ordinary differential equation initial value problems by implicit methods, solution of boundary value problems for ordinary and partial differential equations by any discrete approximation method, construction of splines, and solution of systems of nonlinear algebraic equations represent just a few of the applications of numerical linear algebra. Because of this prevalence of numerical linear algebra, we begin our treatment of basic numerical methods with this topic, and note that this is somewhat nonstandard.

In this chapter we begin with discussion of some basic notations and definitions which will be of importance throughout these lectires, but especially so in the present chapter. Then we consider the two main problems encountered in numerical linear algebra: *i*) solution of linear systems of equations, and *ii*) the algebraic eigenvalue problem. Much attention will be given to the first of these because of its wide applicability; all of the examples cited above involve this class of problems. The second, although very important, occurs less frequently, and we will provide only a cursory treatment.

1.1 Some Basic Facts from Linear Algebra

Before beginning our treatment of numerical solution of linear systems we will review a few important facts from linear algebra, itself. We typically think of linear algebra as being associated with vectors and matrices in some finite-dimensional space. But, in fact, most of our ideas extend quite naturally to the infinite-dimensional spaces frequently encountered in the study of partial differential equations.

We begin with the basic notion of *linearity* which is crucial to much of mathematical analysis.

Definition 1.1 *Let S be a vector space defined on the real numbers $\mathbb{R}$ (or the complex numbers $\mathbb{C}$), and let L be an operator (or transformation) whose domain is S. Suppose for any $u, v \in S$ and $a, b \in \mathbb{R}$ (or $\mathbb{C}$) we have*

$$L(au + bv) = aLu + bLv. \tag{1.1}$$

Then L is said to be a linear operator.

Examples of linear operators include $M \times N$ matrices, differential operators and integral operators.

It is generally important to be able to distinguish linear and nonlinear operators because problems involving only the former can often be solved without recourse to iterative procedures. This

is seldom true for nonlinear problems, with the consequence that corresponding algorithms must be more elaborate. This will become apparent as we proceed.

One of the most fundamental properties of any object, be it mathematical or physical, is its size. Of course, in numerical analysis we are always concerned with the size of the error in any particular numerical approximation, or computational procedure. There is a general mathematical object, called the *norm*, by which we can assign a number corresponding to the size of various mathematical entities.

Definition 1.2 *Let S be a (finite- or infinite-dimensional) vector space, and let $\|\cdot\|$ denote the mapping $S \to \mathbb{R}_+ \cup \{0\}$ with the following properties:*

i) $\|v\| \geq 0, \ \forall\ v \in S$ *with* $\|v\| = 0$ *iff* $v \equiv 0$,

ii) $\|av\| = |a|\ \|v\|, \quad \forall\ v \in S,\ a \in \mathbb{R}$,

iii) $\|v + w\| \leq \|v\| + \|w\| \quad \forall\ v, w \in S$.

Then $\|\cdot\|$ is called a <u>norm</u> for S.

Note that we can take S to be a space of vectors, functions or even operators, and the above properties apply. It is important to observe that for a given space S there are, in general, many different mappings $\|\cdot\|$ having the properties required by the above definition. We will give a few specific examples which are of particular importance in numerical linear algebra.

If S is a finite-dimensional space of vectors with elements $v = (v_1, v_2, \ldots, v_N)^T$ then a familiar measure of the size of v is its Euclidean length,

$$\|v\|_2 = \left(\sum_{i=1}^{N} v_i^2 \right)^{\frac{1}{2}} . \tag{1.2}$$

The proof that $\|\cdot\|_2$, often called the *Euclidean norm*, or simply the *2-norm*, satisfies the three conditions of the definition is straightforward, and is left to the reader. (We note here that it is common in numerical analysis to employ the subscript E to denote this norm and use the subscript 2 for the "spectral" norm of matrices. But we have chosen to defer to notation more consistent with pure mathematics.) Another useful norm that we often encounter in practice is the *max norm* or *infinity norm* defined as

$$\|v\|_\infty = \max_{1 \leq i \leq N} |v_i| . \tag{1.3}$$

In the case of Euclidean spaces, we can define another useful object related to the Euclidean norm, the *inner product* (often called the "dot product" when applied to finite-dimensional vectors).

Definition 1.3 *Let S be a N-dimensional Euclidean space with $v, w \in S$. Then*

$$\langle v, w \rangle \equiv \sum_{i=1}^{N} v_i w_i \tag{1.4}$$

is called the <u>inner product</u>.

It is clear that $\langle v, v \rangle = \|v\|_2^2$ for this particular kind of space; moreover, there is a further property that relates the inner product and the norm, the *Cauchy–Schwarz inequality*.

Theorem 1.1 *(Cauchy–Schwarz) Let S be an inner-product space with inner product $\langle \cdot\ , \cdot \rangle$ and norm $\| \cdot \|_2$. If $v, w \in S$, then*

$$\langle v, w \rangle \leq \|v\|_2 \, \|w\|_2 \ . \tag{1.5}$$

We have thus far introduced the 2-norm, the infinity norm and the inner product for spaces of finite-dimensional vectors. It is worth mentioning that similar definitions hold as well for infinite-dimensional spaces, *i.e.*, spaces of functions. For example, suppose $f(x)$ is a function continuous on the closed interval $[a, b]$, denoted $f \in C[a, b]$. Then

$$\|f\|_\infty = \max_{x \in [a,b]} |f(x)|. \tag{1.6}$$

Similarly, if f is *square integrable* on $[a, b]$, we have

$$\|f\|_2 = \left(\int_a^b f^2 dx \right)^{\frac{1}{2}} .$$

The space consisting of all functions f such that $\|f\|_2 < \infty$ is the canonical *Hilbert space*, $L^2[a, b]$. The Cauchy–Schwarz inequality holds in any such space, and takes the form

$$\int_a^b fg \ dx \leq \left(\int_a^b f^2 \ dx \right)^{\frac{1}{2}} \left(\int_a^b g^2 \ dx \right)^{\frac{1}{2}} \quad \forall \ f, g \in L^2[a, b].$$

We next need to consider some corresponding ideas regarding specific calculations for norms of operators. The general definition of an *operator norm* is as follows.

Definition 1.4 *Let A be an operator whose domain is $\mathcal{D}$. Then the norm of A is defined as*

$$\|A\| \ \equiv \max_{\substack{\|x\|=1 \\ x \in \mathcal{D}(A)}} \|Ax\| \, . \tag{1.7}$$

It is easy to see that this is equivalent to

$$\|A\| \ = \max_{\substack{\|x\| \neq 0 \\ x \in \mathcal{D}(A)}} \frac{\|Ax\|}{\|x\|} \ ,$$

from which follows an inequality similar to the Cauchy–Schwarz inequality for vectors,

$$\|Ax\| \ \leq \ \|A\| \|x\|. \tag{1.8}$$

We should remark here that (1.8) actually holds only in the case when the matrix and vector norms appearing in the expression are "compatible," and this relationship is often used as the definition of *compatibility*. We will seldom need to employ this concept in the present lectures, and the reader is referred to, *e.g.*, Isaacson and Keller [15] (Chap. 1) for additional information.

We observe that neither (1.7) nor the expression following it is suitable for practical calculations; we now present three norms that are readily computed, at least for $M \times N$ matrices. The first of these is the *2-norm*, given in the matrix case by

$$\|A\|_2 \ = \left(\sum_{i,j=1}^{M,N} a_{ij}^2 \right)^{\frac{1}{2}} \ . \tag{1.9}$$

Two other norms are also frequently employed. These are the *1-norm*

$$\|A\|_1 = \max_{1\le j\le N} \sum_{i=1}^{M} |a_{ij}| \ , \tag{1.10}$$

and the *infinity norm*

$$\|A\|_\infty = \max_{1\le i\le M} \sum_{j=1}^{N} |a_{ij}| \ . \tag{1.11}$$

We note that although the definition of the operator norm given above was not necessarily finite-dimensional, we have here given only finite-dimensional practical computational formulas. We will see later that this is not really a serious restriction because problems involving differential operators, one of the main instances where norms of infinite-dimensional operators are needed, are essentially always solved via discrete approximations leading to finite-dimensional matrix representations.

There is a final, general comment that should be made regarding norms. It arises from the fact, mentioned earlier, that in any given vector space many different norms might be employed. A comparison of the formulas in Eqs. (1.2) and (1.3), for example, will show that the number one obtains to quantify the size of a mathematical object, a vector in this case, will change according to which formula is used. Thus, a reasonable question is, "How do we decide which norm to use?" It turns out, for the finite-dimensional spaces we will deal with herein, that it really does not matter which norm is used, provided only that the same one is used when making comparisons between similar mathematical objects. This is the content of what is known as the *norm equivalence theorem*: all norms are equivalent on finite-dimensional spaces in the sense that if a sequence converges in one norm, it will converge in any other norm (see Ref. [15], Chap. 1). This implies that in practice we should usually employ the norm that requires the least amount of floating-point arithmetic for its evaluation. But we note here that the situation is rather different for infinite-dimensional spaces. In particular, for problems involving differential equations, determination of the function space in which a solution exists (and hence, the appropriate norm) is a significant part of the overall problem.

We will close this subsection on basic linear algebra with a statement of the problem whose numerical solution will concern us throughout most of the remainder of this chapter, and provide the formal, exact solution. We will study solution procedures for the linear system

$$Ax = b, \tag{1.12}$$

where $x, b \in \mathbb{R}^N$, and A: $\mathbb{R}^N \to \mathbb{R}^N$ is a nonsingular matrix. If A is singular, *i.e.*, $\det(A) = 0$, then (1.12) does not, in general, admit a solution; we shall have nothing further to say regarding this case. In the nonsingular case we study here, the formal solution to (1.12) is simply

$$x = A^{-1}\ b. \tag{1.13}$$

It was apparently not clear in the early days of numerical computation that direct application of (1.13), *i.e.*, computing A^{-1} and multiplying b, was very inefficient—and this approach is rather natural. But if A is a $N\times N$ matrix, as much as $\mathcal{O}(N^4)$ floating-point arithmetic operations may be required to produce A^{-1}. On the other hand, if the Gaussian elimination procedure to be described in the next section is used, the system (1.12) can be solved for x, directly, in $\mathcal{O}(N^3)$ arithmetic operations. In fact, a more cleverly constructed matrix inversion routine would use this approach to obtain A^{-1} in $\mathcal{O}(N^3)$ arithmetic operations, although the precise number would be considerably greater than that required to directly solve the system. It should be clear from this that one should never invert a matrix to solve a linear system unless the inverse matrix, itself, is needed for other purposes, which is not usually the case for the types of problems treated in these lectures.

1.2 Solution of Linear Systems

In this section we treat the two main classes of methods for solving linear systems: *i*) direct elimination, and *ii*) iterative techniques. For the first of these, we will consider the general case of a nonsparse $N \times N$ system matrix, and then study a very efficient elimination method designed specifically for the solution of systems whose matrices are sparse, and banded. The study of the second topic, iterative methods, will include only very classical material. It is the author's opinion that students must be familiar with this before going on to study the more modern, and much more efficient, methods. Thus, our attention here will be restricted to the topics Jacobi iteration, Gauss–Seidel iteration and successive overrelaxation.

1.2.1 Numerical solution of linear systems: direct elimination

In this subsection we will provide a step-by-step treatment of Gaussian elimination applied to a small, but general, linear system. From this we will be able to discern the general approach to solving *nonsparse* (*i.e.*, having few zero elements) linear systems. We will give a general theorem that establishes the conditions under which Gaussian elimination is guaranteed to yield a solution in the absence of round-off errors, and we will then consider the effects of such errors in some detail. This will lead us to a slight modification of the basic elimination algorithm. We then will briefly look theoretically at the effects of rounding error. The final topic to be covered will be yet another modification of the basic Gaussian elimination algorithm, in this case designed to very efficiently solve certain sparse, banded linear systems that arise in many practical problems.

Gaussian Elimination for Nonsparse Systems

We will begin by considering a general 3×3 system of linear algebraic equations:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \ , \tag{1.14}$$

where the matrix A with elements a_{ij} is assumed to be nonsingular. Moreover, we assume that no a_{ij} or b_i is zero. (This is simply to maintain complete generality.) If we perform the indicated matrix/vector multiplication on the left-hand side of (1.14) we obtain

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \ , \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \ , \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \ . \end{aligned} \tag{1.15}$$

It is clear from this representation that if $a_{21} = a_{31} = a_{32} = 0$, the solution to the whole system can be calculated immediately, starting with

$$x_3 = \frac{b_3}{a_{33}},$$

and working backward, in order, to x_1. This motivates trying to find combinations of the equations in (1.15) such that the lower triangle of the matrix A is reduced to zero. We will see that the resulting formal procedure, known as *Gaussian elimination*, or simply *direct elimination*, is nothing more than a systematic approach to methods from high school algebra, organized to lend itself to machine computation.

The Gaussian elimination algorithm proceeds in stages. At each stage a *pivot element* is selected so as to appear in the upper left-hand corner of the largest submatrix not processed in an earlier stage. Then all elements in the same column below this pivot element are zeroed by means of *elementary operations* (multiplying by a constant, replacing an equation with a linear combination of equations). This continues until the largest submatrix consists of only the element in the lower right-hand corner. We demonstrate this process with the system (1.15).

At stage 1, the pivot element is a_{11}, and in carrying out calculations for this stage we must zero the elements a_{21} and a_{31}. If we multiply the first equation by the *Gauss multiplier*

$$m_{21} \equiv \frac{a_{21}}{a_{11}}\,,$$

and subtract the result from the second equation, we obtain

$$\left[a_{21} - a_{11}\left(\frac{a_{21}}{a_{11}}\right)\right] x_1 \;+\; \left[a_{22} - a_{12}\left(\frac{a_{21}}{a_{11}}\right)\right] x_2 \;+\; \left[a_{23} - a_{13}\left(\frac{a_{21}}{a_{11}}\right)\right] x_3 \;=\; b_2 \;-\; b_1\left(\frac{a_{21}}{a_{11}}\right) \quad ,$$

or

$$0 \cdot x_1 + (a_{22} - m_{21}a_{12})x_2 + (a_{23} - m_{21}a_{13})x_3 = b_2 - m_{21}b_1.$$

We see that if we replace the second equation of (1.15) with this result we will have achieved the goal of zeroing a_{21}.

Similarly, define $m_{31} \equiv a_{31}/a_{11}$, and carry out analogous operations between the first and third equations of (1.15). This completes the first stage, because we have now zeroed all elements below the pivot element in the first column of the matrix A. The system can now be expressed as

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a^*_{22} & a^*_{23} \\ 0 & a^*_{32} & a^*_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b^*_2 \\ b^*_3 \end{bmatrix} \quad ,$$

where

$$\begin{aligned} a^*_{22} &= a_{22} - m_{21}a_{12}, \quad a^*_{23} = a_{23} - m_{21}a_{13}, \quad b^*_2 = b_2 - m_{21}b_1, \\ a^*_{32} &= a_{32} - m_{31}a_{12}, \quad a^*_{33} = a_{33} - m_{31}a_{13}, \quad b^*_3 = b_3 - m_{31}b_1. \end{aligned}$$

Itis worthwhile to observe that no element of the first row (equation) has been changed by this process; but, in general, all other elements will have been changed.

We are now prepared to continue to stage 2. Now the pivot element is a^*_{22} (not a_{22}!) and we have only to zero the element a^*_{32}. To show more clearly what is happening, we write the part of the system being considered at this stage as

$$\begin{aligned} a^*_{22}x_2 + a^*_{23}x_3 &= b^*_2\,, \\ a^*_{32}x_2 + a^*_{33}x_3 &= b^*_3\,. \end{aligned}$$

Now we define $m^*_{32} \equiv a^*_{32}/a^*_{22}$, multiply this by the first equation (of the current stage), and subtract from the second. This yields

$$\left[a^*_{32} - a^*_{22}\left(\frac{a^*_{32}}{a^*_{22}}\right)\right] x_2 + \left[a^*_{33} - a^*_{23}\left(\frac{a^*_{32}}{a^*_{22}}\right)\right] x_3 = b^*_3 - b^*_2\left(\frac{a^*_{32}}{a^*_{22}}\right)\,.$$

Thus, if we define

$$a^{**}_{33} \equiv a^*_{33} - m^*_{32}a^*_{23}\,, \qquad \text{and} \qquad b^{**}_3 \equiv b^*_3 - m^*_{32}b^*_2\,,$$

then the system takes the desired form, namely,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{*} & a_{23}^{*} \\ 0 & 0 & a_{33}^{**} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2^{*} \\ b_3^{**} \end{bmatrix} \quad . \tag{1.16}$$

In summarizing this elimination process there are several important observations to be made regarding the form of (1.16), and how it was achieved. First, we note that the matrix in (1.16) is of the form called *upper triangular*; and as suggested earlier, such a system can be easily solved. In arriving at this form, the first equation in the original system was never changed, as already noted, but more importantly, neither was the solution vector, or the ordering of its elements, changed. Thus, the solution to (1.16) is identical to that of (1.14). Obviously, if this were not true, this whole development would be essentially worthless. Next we observe that for the 3×3 matrix treated here, only two elimination stages were needed. It is easily seen that this generalizes to arbitrary $N \times N$ matrices for which $N-1$ elimination stages are required to achieve upper triangular form. Finally, we point out the tacit assumption that the pivot element at each stage is nonzero, for otherwise the Gauss multipliers, m_{ij} would not be defined. It is quite possible for this assumption to be violated, even when the matrix A is nonsingular. We will treat this case later.

We can now complete the solution of Eq. (1.14) via the upper triangular system (1.16). We have

$$\begin{aligned} x_3 &= b_3^{**}/a_{33}^{**} \ , \\ x_2 &= (b_2^{*} - a_{23}^{*}x_3)/a_{22}^{*} \ , \\ x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \ . \end{aligned}$$

From the preceding we should see that Gaussian elimination proceeds in two basic steps. The first transforms the original system to an *equivalent* (in the sense that it has the same solution) system with an upper triangular matrix. This is called the (*forward*) *elimination step*. The second step merely solves the upper triangular system obtained in the first step. As just seen, this proceeds backwards from the last component of the solution vector, and is termed *backward substitution*.

We summarize the above in a *pseudo-language* (or *meta-language*) *algorithm* from which a computer code can be easily written. We will use this type of structure in various forms throughout the text. It has the advantage of generality, thus permitting actual code generation in any language, just as would be true with a flow chart; but use of pseudo-language algorithms is often more appropriate than use of flow charts in scientific computing where one encounters difficulties in making long complicated formulas fit inside the boxes of a flow chart.

Algorithm 1.1 *(Gaussian Elimination)*

1. *Forward Elimination*

Do $k = 1, N-1$	*(increment stage counter)*
Do $i = k+1, N$	*(increment row index)*
$m_{ik} = a_{ik}/a_{kk}$	*(calculate Gauss multiplier)*
$b_i = b_i - m_{ik}b_k$	*(modify RHS of* i^{th} *equation)*
Do $j = k+1, N$	*(increment column index)*
$a_{ij} = a_{ij} - m_{ik}a_{kj}$	*(modify matrix components of* i^{th} *equation)*
Repeat j	

Repeat i

Repeat k

2. Backward Substitution

$x_N = b_N/a_{NN}$
Do $i = N-1, 1$
  $x_i = 0$
  Do $j = i+1, N$
    $x_i = x_i + a_{ij}x_j$
  Repeat j
$x_i = (b_i - x_i)/a_{ii}$
Repeat i

At this point we comment that it should be clear from the structure of this algorithm that $\mathcal{O}(N^3)$ arithmetic operations are required to obtain a solution to a linear system using Gaussian elimination. In particular, in the forward elimination step there is a nesting of three DO loops, each of which runs $\mathcal{O}(N)$ times. In addition, the backward substitution step requires $\mathcal{O}(N^2)$ operations; but for large N this is negligible compared with $\mathcal{O}(N^3)$. It is important to realize that even on modern supercomputers, there are many situations in which this amount of arithmetic is prohibitive, and we will investigate ways to improve on this operation count. However, for nonsparse matrices Gaussian elimination is in most cases the preferred form of solution procedure.

We now state a theorem that provides conditions under which Gaussian elimination is guaranteed to succeed.

Theorem 1.2 *(LU Decomposition) Given a square $N\times N$ matrix A, let A_k denote the principal minor constructed from the first k rows and k columns. Suppose that $det(A_k) \neq 0 \ \forall \ k = 1, 2, \ldots, N-1$. Then $\exists$ a unique lower triangular matrix $L = (\ell_{ij})$ with $\ell_{11} = \ell_{22} = \cdots = \ell_{NN} = 1$ and a unique upper triangular matrix $U = (u_{ij}) \ni LU = A$. Furthermore,*

$$\det(A) = \prod_{i=1}^{N} u_{ii}.$$

We remark that it is not entirely obvious that Gaussian elimination, as presented above, actually corresponds to LU decomposition of the theorem; but, in fact, the lower triangular matrix of the theorem can be recovered from the Gauss multipliers, as demonstrated in Forsythe and Moler [9], for example. The importance of LU decomposition, *per se*, will be more apparent later when we treat direct solution of sparse systems having band matrices.

Rounding Errors in Gaussian Elimination

We next consider a well-known example (see Ref. [9]), the purpose of which is to demonstrate the effects of rounding errors in Gaussian elimination. The system

$$\begin{bmatrix} 0.0001 & 1.0 \\ 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix} \tag{1.17}$$

has the solution $x_1 = 1.000$, $x_2 = 1.000$ when solved by Cramer's rule with rounding to three significant digits. We now perform Gaussian elimination with *the same arithmetic precision.* Since

this is a 2×2 system, there will be only a single forward elimination stage. We choose the natural pivot element, $a_{11} = 0.0001$, and compute the Gauss multiplier

$$m_{21} = a_{21}/a_{11} = 10^4 \ .$$

Then, to three significant digits,

$$a^*_{22} = a_{22} - m_{21}a_{12} = 1.0 - (10^4)(1.0) = -10^4,$$

and

$$b^*_2 = b_2 - m_{21}b_1 = 2.0 - (10^4)(1.0) = -10^4.$$

We then have

$$x_2 = b^*_2/a^*_{22} = 1.0,$$

and

$$x_1 = \frac{b_1 - a_{12}x_2}{a_{11}} = \frac{1.0 - (1.0)(1.0)}{10^{-4}} = 0.0.$$

Thus, the second component of the solution is accurate, but the first component is completely wrong.

An obvious remedy is to increase the precision of the arithmetic, since three significant digits is not very accurate to begin with. In particular, the rounding errors arise in this case during subtraction of a large number from a small one, and information from the small number is lost due to low arithmetic precision. Often, use of higher precision might be the simplest approach to handle this. Unfortunately, even for the simple 2×2 system just treated we could choose a_{11} in such a way that Gaussian elimination would fail on any finite machine. The desire to produce machine-independent algorithms led numerical analysts to study the causes of rounding errors in direct elimination methods. It is not too surprising that the error originates in the forward elimination step, and is caused by relatively (with respect to machine precision) small values of the pivot element used at any particular stage of the elimination procedure which, in turn, can result in large Gauss multipliers, as seen in the preceding example. From this piece of information it is clear that at the beginning of each stage it would be desirable to arrange the equations so that the pivot element for that stage is relatively large.

There are three main strategies for incorporating this idea: *i*) *column pivoting*, *ii*) *row pivoting* and *iii*) *complete pivoting*. The first of these is the easiest to implement because it requires only row interchanges, and therefore does not result in a re-ordering of the solution vector. Row pivoting requires column interchanges, and thus does reorder the solution vector; as would be expected, complete pivoting utilizes both row and column interchanges.

Numerical experiments, performed for a large collection of problems, have shown that complete pivoting is more effective than either of the *partial pivoting* strategies, which are generally of comparable effectiveness. Thus, we recommend row interchange (column pivoting) if a partial pivoting strategy, which usually is sufficient, is employed. The Gaussian elimination algorithm given earlier requires the following additional steps, inserted between the "k" and "i" DO loops of the forward elimination step, in order to implement the row interchange strategy.

Algorithm 1.2 *(Row Interchange, i.e., Column Pivoting)*

1. *Locate largest (in absolute value) element in column containing current pivot element*

   $a_{max} = |a_{kk}|$
   $i_{max} = k$

Do $i = k+1, N$

If $|a_{ik}| > a_{max}$, *then*

$a_{max} = |a_{ik}|$

$i_{max} = i$

Repeat i

If $i_{max} = k$, *begin i-loop of forward elimination*

2. *Interchange rows to place largest element of current column in pivot position*

Do $j = k, N$

$a_{temp} = a_{kj}$

$a_{kj} = a_{i_{max},j}$

$a_{i_{max},j} = a_{temp}$

Repeat j

$b_{temp} = b_k$

$b_k = b_{i_{max}}$

$b_{i_{max}} = b_{temp}$

It is important to note that these steps must be executed at every stage of the forward elimination procedure. It is not sufficient to seek the largest pivot element prior to the first stage, and then simply use the natural pivot elements thereafter. It should be clear from the detailed analysis of the 3×3 system carried out earlier that the natural pivot element at any given stage may be very different from the element in that position of the original matrix.

We should also point out that there are numerous other procedures for controlling round-off errors in Gaussian elimination, treatment of which is beyond the intended scope of these lectures. Some of these, such as "balancing" of the system matrix, are presented in the text by Johnson and Riess [16], for example.

Condition Number of a Matrix

We shall conclude this treatment of basic Gaussian elimination with the derivation of a quantity which, when it can be calculated, provides an *a priori* estimate of the effects of rounding and data errors on the accuracy of the solution to a given linear system. This quantity is called the condition number of the system matrix.

We begin with the general system

$$Ax = b$$

with exact solution $x = A^{-1}b$. We suppose that A can be represented exactly, and that somehow A^{-1} is known exactly. We then inquire into the error in the solution vector, x, if the forcing vector b is in error by δb. If the error in x due to δb is denoted δx, we have

$$A(x + \delta x) = b + \delta b,$$

from which it follows that

$$\delta x = A^{-1}\delta b.$$

Thus, (assuming use of compatible matrix and vector norms—recall Eq. (1.8))

$$\|\delta x\| \;\leq\; \|A^{-1}\|\,\|\delta b\|\,. \tag{1.18}$$

Now from the original equation we have

$$\|b\| \;\leq\; \|A\|\,\|x\|\,,$$

and using this with (1.18) yields

$$\|\delta x\|\;\|b\| \;\leq\; \|A^{-1}\|\;\|\delta b\|\;\|A\|\;\|x\|\,,$$

or

$$\frac{\|\delta x\|}{\|x\|} \;\leq\; \|A\|\,\|A^{-1}\|\frac{\|\delta b\|}{\|b\|}\,. \tag{1.19}$$

The inequality in (1.19) shows that the relative error in the data vector b is amplified by an amount $\|A\|\|A^{-1}\|$ to produce a corresponding relative error in the solution vector. This quantity is called the *condition number* of A, denoted

$$\mathrm{cond}(A) \equiv\; \|A\|\,\|A^{-1}\|\,, \tag{1.20}$$

or often simply $\kappa(A)$. With a bit more work, it can also be shown (see [9], for example) that

$$\frac{\|\delta x\|}{\|x+\delta x\|} \;\leq\; \mathrm{cond}(A)\frac{\|\delta A\|}{\|A\|}\,. \tag{1.21}$$

This inequality, together with (1.19), shows the gross effects of rounding errors on the numerical solution of linear systems. In this regard it is of interest to note that the elimination step of Gaussian elimination results in the replacement of the original system matrix A with the LU decomposition $LU-\delta A$, and the original forcing vector b with $b^*-\delta b$, where δA and δb are due to round-off errors.

There are several remarks to make regarding the condition number. The first is that *its value depends on the norm* in which it is calculated. Second, determination of its exact value depends on having an accurate A^{-1}. But if A is *badly conditioned*, it will be difficult (and costly) to compute A^{-1} accurately. Hence, cond(A) will be inaccurate. There has been considerable research into obtaining accurate and efficiently-computed approximations to cond(A). (The reader is referred to Dongarra *et al.* [6] for computational procedures.) A readily calculated rough approximation is the following:

$$\mathrm{cond}(A) \approx \frac{\|A\|_2}{|\det(A)|}\,, \tag{1.22}$$

which is given by Hornbeck [14] in a slightly different context. It should be noted, however, that this is only an approximation, and not actually a condition number, since it does not satisfy the easily proven inequality,

$$\mathrm{cond}(A) \geq 1\,.$$

A system is considered badly conditioned, or *ill conditioned*, when cond$(A) \gg 1$, and *well conditioned* when cond$(A) \sim \mathcal{O}(1)$. However, ill conditioning must be viewed relative to the arithmetic precision of the computer hardware being employed, and relative to the required precision of computed results. For example, in current single-precision arithmetic (32-bit words), condition numbers up to $\mathcal{O}(10^3)$ can typically be tolerated; in double precision (64-bit words) accurate solutions can be obtained even when cond$(A) \gtrsim \mathcal{O}(10^6)$. The interested reader may find further discussions of this topic by Johnson and Riess [16] quite useful.

LU Decomposition of Sparse, Band Matrices

Our last topic in this subsection is treatment of certain sparse, band matrices by means of direct elimination. We note that iterative procedures of the sort to be discussed later are generally preferred for solving linear systems having sparse coefficient matrices. There is, however, one particular situation in which this is not the case, and we will consider this in the present discussion.

Sparse band matrices arise in a number of important applications as will become apparent as we proceed through these lectures; specific instances include construction of spline approximations and numerical solution of boundary value problems for ordinary and partial differential equations. Thus, efficient treatment of such systems is extremely important.

Sparse, band matrices appear, generally, as in Fig. 1.1. Here, the diagonal lines represent bands

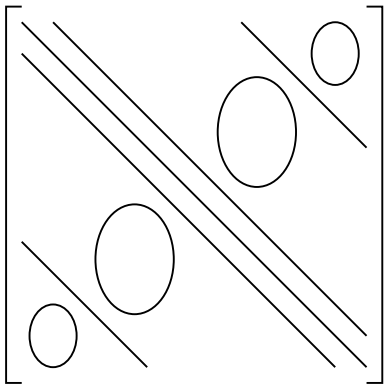


Figure 1.1: Sparse, band matrix

of (mostly) nonzero elements, and all other elements are zero. This particular matrix structure is still too general to permit application of direct band elimination. In general, this requires that matrices be in the form called *compactly banded*, shown in Fig. 1.2. Part (a) of this figure shows

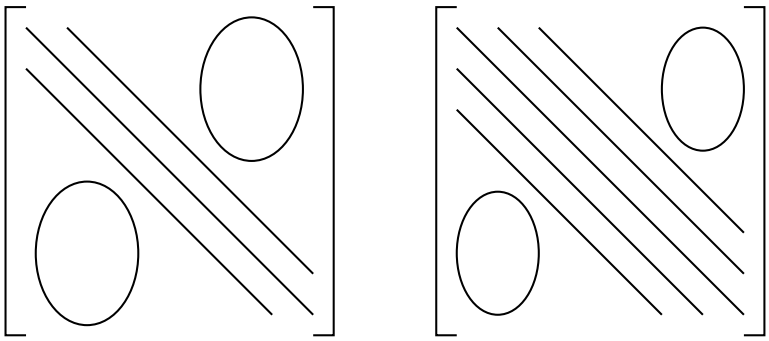
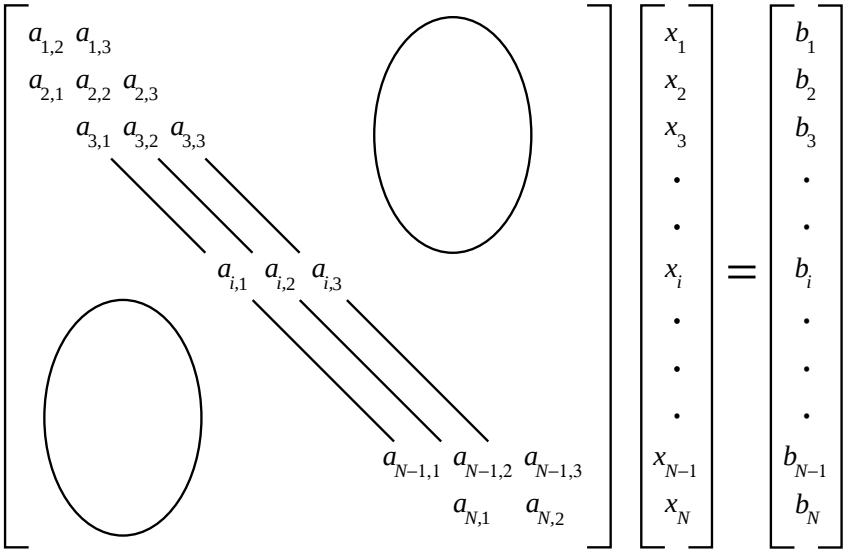


Figure 1.2: Compactly-banded matrices: (a) tridiagonal, (b) pentadiagonal

a three-band, or *tridiagonal* matrix, while part (b) displays a *pentadiagonal* matrix. Both of these frequently occur in practice in the contexts mentioned above; but the former is more prevalent, so it will be the only one specifically treated here. It will be clear, however, that our approach directly

extends to more general cases. In fact, it is possible to construct a single algorithm that can handle any compactly-banded matrix.

The approach we shall use for solving linear systems with coefficient matrices of the above form is formal LU decomposition as described in Theorem 1.2. The algorithm we obtain is completely equivalent (in terms of required arithmetic and numerical stability with respect to rounding errors) to Gaussian elimination, and to the well-known *Thomas algorithm* [34] for tridiagonal systems. We write the system as

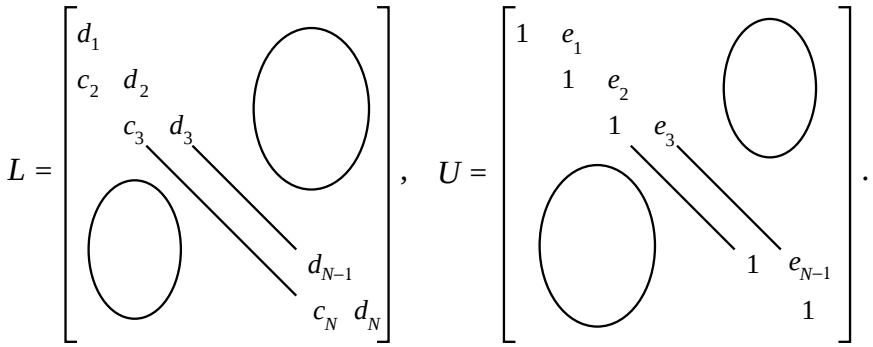


(1.23)

Observe that indexing of the elements of A is done differently than in the nonsparse case treated earlier. In particular, the first index in the present sparse case is the equation (row) index, as was true for nonsparse matrices; but the second index is now the band number, counting from lower left to upper right corners of the matrix, rather than the column number.

In order to derive the LU decomposition formulas, we assume the matrix A can be decomposed into the lower- and upper-triangular matrices, L and U, respectively (as guaranteed by the LU-decomposition Theorem). We then formally multiply these matrices back together, and match elements of the resulting product matrix LU with those of the original matrix A given in Eq. (1.23). This will lead to expressions for the elements of L and U in terms of those of A.

A key assumption in this development is that L and U are also band matrices with the structure of the lower and upper, respectively, triangles of the original matrix A. We thus have



(1.24)

Multiplication of L times U then yields

$$LU = \begin{bmatrix} d_1 & d_1 e_1 & & & & & \\ c_2 & d_2 + c_2 e_1 & d_2 e_2 & & & & \\ & c_3 & d_3 + c_3 e_2 & d_3 e_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & c_i & d_i + c_i e_{i-1} & d_i e_i & \\ & & & & \ddots & \ddots & d_{N-1} e_{N-1} \\ & & & & & c_N & d_N + c_N e_{N-1} \end{bmatrix} .$$

Thus, as we would hope, the matrices of our proposed LU decomposition when multiplied together lead to the same tridiagonal matrix structure as that of the original matrix A in Eq. (1.23), and from the LU-decomposition theorem we know that the matrix elements must equal those of the original matix A—element by element. Thus, we immediately see that $c_i = a_{i,1}$ must hold $\forall\ i = 2, \ldots, N$. Also, $d_1 = a_{12}$, which permits calculation of $e_1 = a_{13}/d_1 = a_{13}/a_{12}$. In general, we see that we can immediately calculate d_i and e_i at each row of LU, since c_i and e_{i-1} are already known. In particular, we have

$$\begin{aligned} d_i &= a_{i,2} - c_i e_{i-1} \\ &= a_{i,2} - a_{i,1} e_{i-1} \,, \end{aligned}$$

and

$$e_{i-1} = a_{i-1,3}/d_{i-1} \,.$$

Hence, determination of the components of L and U is a straightforward, well-defined process.

Once we have obtained an LU decomposition of any matrix, whether or not it is sparse or banded, solution of the corresponding system is especially simple. Consider again the general system

$$Ax = b,$$

and suppose we have obtained upper and lower triangular matrices as in the LU-decomposition Theorem, *i.e.*, such that $A = LU$. Then

$$LUx = b,$$

and if we set $y = Ux$, we have

$$Ly = b.$$

But b is given data, and L is a lower triangular matrix; hence, we can directly solve for y by *forward substitution*, starting with the first component of y. Once y has been determined, we can find the desired solution, x, from $Ux = y$ via *backward substitution*, since U is upper triangular. Again, we emphasize that this procedure works for any LU decomposition; its use is not restricted to the tridiagonal case that we have treated here.

We now summarize the preceding derivation for solution of tridiagonal systems with the following pseudo-language algorithm.

Algorithm 1.3 *(Solution of Tridiagonal Systems by LU Decomposition)*

1. *Construction of L and U from elements of A*

   Do $i = 1, N$

   If $i > 1$, *then* $a_{i,2} = a_{i,2} - a_{i,1}a_{i-1,3}$

   If $i < N$, *then* $a_{i,3} = a_{i,3}/a_{i,2}$

   Repeat i

2. *Forward substitution (Solve $Ly = b$)*

   Do $i = 1, N$

   If $i = 1$, *then* $b_1 = b_1/a_{1,2}$

   else $b_i = (b_i - a_{i,1}b_{i-1})/a_{i,2}$

   Repeat i

3. *Backward substitution (Solve $Ux = y$)*

   $x_N = b_N$

   Do $i = N - 1, 1$

   $x_i = b_i - a_{i,3}x_{i+1}$

   Repeat i

It is worthwhile to compare some of the details of this algorithm with Gaussian elimination discussed earlier for nonsparse systems. Recall in that case we found, for a $N \times N$ system matrix, that the total required arithmetic was $\mathcal{O}(N^3)$, and the necessary storage was $\mathcal{O}(N^2)$. For tridiagonal systems the situation is much more favorable. Again, for a $N \times N$ system (*i.e.*, N equations in N unknowns) only $\mathcal{O}(N)$ arithmetic operations are needed to complete a solution. The reason for this significant reduction of arithmetic can easily be seen by noting that in the algorithm for the tridiagonal case there are no nested DO loops. Furthermore, the longest loop is only of length N. It is clear from (1.23) that at most $5N$ words of storage are needed, but in fact, this can be reduced to $4N - 2$ by storing the solution vector, x, in the same locations that originally held the right-hand side vector, b. Rather typical values of N are in the range $\mathcal{O}(10^2)$ to $\mathcal{O}(10^3)$. Hence, very significant savings in arithmetic (and storage) can be realized by using band elimination in place of the usual Gaussian elimination algorithm. We should note, however, that it is rather difficult (and seldom done) to implement pivoting strategies that preserve sparsity. Consequently, tridiagonal LU decomposition is typically applied only in those situations where pivoting would not be necessary.

A final remark, of particular importance with respect to modern vector and parallel supercomputer architectures, should be made regarding the tridiagonal LU-decomposition algorithm just presented. It is that this algorithm can be neither vectorized nor parallelized in any direct manner; it works best for scalar processing. There are other band-matrix solution techniques, generally known as *cyclic reduction* methods, that can be both vectorized and parallelized to some extent. Discussion of these is beyond the intended scope of the present lectures; the reader is referred to Duff *et al.* [8], for example, for more information on this important topic. On the other hand, parallelization of application of the algorithm (rather than of the algorithm, itself) is very effective, and widely used, in numerically solving partial differential equations by some methods to be treated in Chap. 5 of these notes.

1.2.2 Numerical solution of linear systems: iterative methods

From the preceding discussions of direct elimination procedures it is easily seen that if for any reason a system must be treated as nonsparse, the arithmetic required for a solution can become prohibitive already for relatively small N, say $N \sim \mathcal{O}(10^5)$. In fact, there are many types of sparse systems whose structure is such that the sparsity cannot be exploited in a direct elimination algorithm. Figure 1.1 depicts an example of such a matrix. Matrices of this form arise in some finite-difference and finite-element discretizations of fairly general two-dimensional elliptic operators, as we shall see in Chap. 5; for accurate solutions one may wish to use at least 10^4 points in the finite-difference mesh, and it is now not unusual to employ as many as 10^6 points, or even more. This results in often unacceptably long execution times, even on supercomputers, when employing direct elimination methods. Thus, we are forced to attempt the solution of this type of problem via iterative techniques. This will be the topic of the present section.

We will begin with a general discussion of *fixed-point iteration*, the basis of many commonly used iteration schemes. We will then apply fixed-point theory to linear systems, first in the form of Jacobi iteration, then with an improved version of this known as Gauss–Seidel iteration, and finally with the widely-used successive overrelaxation.

Fundamentals of Fixed-Point Iteration

Rather than start immediately with iterative procedures designed specifically for linear problems, as is usually done in the present context, we will begin with the general underlying principles because these will be of use later when we study nonlinear systems, and because they provide a more direct approach to iteration schemes for linear systems as well. Therefore, we will briefly digress to study some elementary parts of the *theory of fixed points*.

We must first introduce some mathematical notions. We generally view iteration schemes as methods which somehow generate a sequence of approximations to the desired solution for a given problem. This is often referred to as "successive approximation," or sometimes as "trial and error." In the hands of a novice, the latter description may, unfortunately, be accurate. Brute-force, trial-and-error methods often used by engineers are almost never very efficient, and very often do not work at all. They are usually constructed on the basis of intuitive perceptions regarding the physical system or phenomenon under study, with little or no concern for the mathematical structure of the equations being solved. This invites disaster.

In *successive approximation* methods we start with a function, called the *iteration function*, which maps one approximation into another, hopefully better, one. In this way a sequence of possible solutions to the problem is generated. The obvious practical question that arises is, "When have we produced a sufficient number of approximations to have obtained an acceptably accurate answer?" This is simply a restatement of the basic convergence question from mathematical analysis, so we present a few definitions and notations regarding this.

Definition 1.5 *Let $\{y_m\}_{m=1}^{\infty}$ be a sequence in $\mathbb{R}^N$. The sequence is said to converge to the limit $y \in \mathbb{R}^N$ if $\forall\, \epsilon > 0\ \exists\, M$ (depending on ϵ) $\ni \forall\, m \geq M$, $\|y - y_m\| < \epsilon$. We denote this by $\lim_{m\to\infty} y_m = y$.*

We note here that the norm has not been specified, and we recall the earlier remark concerning equivalence of norms in finite-dimensional spaces. (More information on this can be found, for example, in Apostol [1].) We also observe that when $N = 1$, we merely replace norm $\|\cdot\|$ with absolute value $|\cdot|$.

It is fairly obvious that the above definition is not generally of practical value, for if we knew the limit y, which is required to check convergence, we probably would not need to generate the

sequence in the first place. To circumvent such difficulties mathematicians invented the notion of a *Cauchy sequence* given in the following definition.

Definition 1.6 *Let* $\{y_m\}_{m=1}^{\infty} \in \mathbb{R}^N$, *and suppose that* $\forall\, \epsilon > 0\ \exists\ M$ *(depending on* ϵ*)* $\ni \forall\, m, n \geq M$, $\|y_m - y_n\| < \epsilon$. *Then* $\{y_m\}$ *is a Cauchy sequence.*

By itself, this definition would not be of much importance; but it is a fairly easily proven fact from elementary analysis (see, *e.g.*, [1]) that every Cauchy sequence in a complete metric space converges to an element of that space. It is also easy to show that $\mathbb{R}^N$ is a complete metric space $\forall\ N < \infty$. Thus, we need only demonstrate that our successive approximations form a Cauchy sequence, and we can then conclude that the sequence converges in the sense of the earlier definition.

Although this represents a considerable improvement over trying to use the basic definition in convergence tests, it still leaves much to be desired. In particular, the definition of a Cauchy sequence requires that $\|y_m - y_n\| < \epsilon$ hold $\forall\ \epsilon > 0$, and $\forall\ m, n \geq M$, where M, itself, is not specified, *a priori*. For computational purposes it is completely unreasonable to choose ϵ smaller than the absolute normalized precision (the *machine* ϵ) of the floating-point arithmetic employed. It is usually sufficient to use values of $\epsilon \sim \mathcal{O}(e/10)$, where e is the acceptable error for the computed results. The more difficult part of the definition to satisfy is " $\forall\ m, n \geq M$." However, for well-behaved sequences, it is typically sufficient to choose $n = m + k$ where k is a specified integer between one and, say 100. (Often $k = 1$ is used.) The great majority of computer-implemented iteration schemes test convergence in this way; that is, the computed sequence $\{y_m\}$ is considered to be converged when $\|y_{m+1} - y_m\| < \epsilon$ for some prescribed (and often completely fixed) ϵ. In many practical calculations $\epsilon \approx 10^{-3}$ represents quite sufficient accuracy, but of course this is problem dependent.

Now that we have a means by which the convergence of sequences can be tested, we will study a systematic method for generating these sequences in the context of solving equations. This method is based on a very powerful and basic notion from mathematical analysis, the *fixed point* of a function, or mapping.

Definition 1.7 *Let* $f\colon \mathcal{D} \to \mathcal{D}$, $\mathcal{D} \subset \mathbb{R}^N$. *Suppose* $x \in \mathcal{D}$, *and* $x = f(x)$. *Then* x *is said to be a fixed point of* f *in* $\mathcal{D}$.

We see from this definition that a fixed point of a mapping is simply any point that is mapped back to itself by the mapping. Now at first glance this might not seem too useful—some point being repeatedly mapped back to itself, over and over again. But the expression $x = f(x)$ can be rewritten as

$$x - f(x) = 0, \tag{1.25}$$

and in this form we recognize that a fixed point of f is a *zero* (or *root*) of the function $g(x) \equiv x - f(x)$. Hence, if we can find a way to compute fixed points, we automatically obtain a method for solving equations.

From our earlier description of successive approximation, intuition suggests that we might try

to find a fixed point of f via the following iteration scheme:

$$\begin{aligned} x_1 &= f(x_0) \\ x_2 &= f(x_1) \\ &\vdots \\ x_m &= f(x_{m-1}) \\ &\vdots \end{aligned}$$

where x_0 is an initial guess. This procedure generates the sequence $\{x_m\}$ of approximations to the fixed point x^*, and we continue this until $\|x_{m+1} - x_m\| < \epsilon$.

It is of interest to analyze this scheme graphically for f: $\mathcal{D} \to \mathcal{D}$, $\mathcal{D} \subset \mathbb{R}^1$. This is presented in the sketch shown as Fig. 1.3. We view the left- and right-hand sides of $x = f(x)$ as two separate functions, $y = x$ and $z = f(x)$. Clearly, there exists $x = x^*$ such that $y = z$ is the fixed point of f; that is, $x^* = f(x^*)$. Starting at the initial guess x_0, we find $f(x_0)$ on the curve $z = f(x)$. But according to the iteration scheme, $x_1 = f(x_0)$, so we move horizontally to the curve $y = x$. This locates the next iterate, x_1 on the x-axis, as shown in the figure. We now repeat the process by again moving vertically to $z = f(x)$ to evaluate $f(x_1)$, and continuing as before. It is easy to see that the iterations are very rapidly converging to x^* for this case.

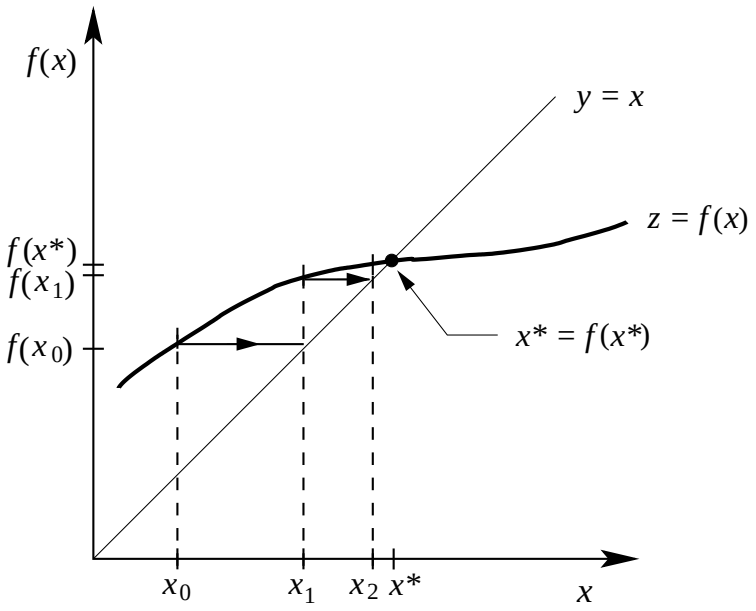


Figure 1.3: Graphical analysis of fixed-point iteration: the convergent case

But we should not be lulled into a false sense of confidence by this easy success; in general, the success or failure of fixed-point iteration depends strongly on the behavior of the iteration function f *in a neighborhood of the fixed point.* Consider now the graphical construction presented

in Fig. 1.4. We first observe that the function f indeed has a fixed point, x^*. But the iterations starting from $x_0 < x^*$ do not converge to this point. Moreover, the reader may check that the iterations diverge also for any initial guess $x_0 > x^*$. Comparison of Figs. 1.3 and 1.4 shows that in Fig. 1.3, f has a slope less than unity (in magnitude) throughout a neighborhood containing the fixed point, while this is not true for the function in Fig. 1.4. An iteration function having a

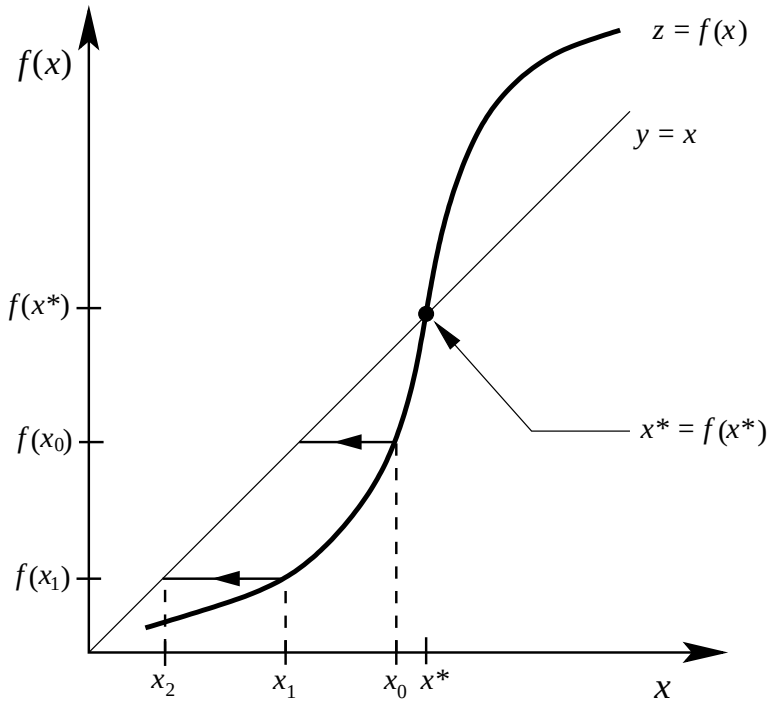


Figure 1.4: Graphical analysis of fixed-point iteration: the divergent case

slope whose absolute value is less than unity in a neighborhood of the fixed point is a fundamental requirement for the success of a fixed-point iteration scheme. This is equivalent to requiring that an interval on the x-axis, containing the fixed point, be mapped into a shorter interval by the iteration function f. In such cases, f is said to be a *contraction*. The following theorem utilizes this basic idea to provide sufficient conditions for convergence of fixed-point iterations in finite-dimensional spaces of dimension N.

Theorem 1.3 *(Contraction Mapping Principle) Let f be continuous on a compact subset $\mathcal{D} \subset \mathbb{R}^N$ with $f\colon \mathcal{D} \to \mathcal{D}$, and suppose $\exists$ a positive constant $L < 1$ $\ni$*

$$\|f(y) - f(x)\| \leq L\|y - x\| \quad \forall\ x, y \in \mathcal{D}\,. \tag{1.26}$$

Then $\exists$ a unique $x^* \in \mathcal{D} \ni x^* = f(x^*)$, and the sequence $\{x_m\}_{m=0}^{\infty}$ generated by $x_{m+1} = f(x_m)$ converges to x^* from any (and thus, every) initial guess, $x_0 \in \mathcal{D}$.

The inequality (1.26) is of sufficient importance to merit special attention.

Definition 1.8 *The inequality,*

$$\|f(y) - f(x)\| \leq L\|y - x\|\,, \quad \forall\ x, y \in \mathcal{D}\,,$$

is called a Lipschitz condition, and L is the Lipschitz constant. Any function f satisfying such a condition is said to be a Lipschitz function.

There are several things to note regarding the above theorem. The first is that satisfaction of the Lipschitz condition with $L < 1$ is sufficient, but not always necessary, for convergence of the corresponding iterations. Second, for any set $\mathcal{D}$, and mapping f with (1.26) holding throughout, x^* is the unique fixed point of f in $\mathcal{D}$. Furthermore, the iterations will converge to x^* using any starting guess, whatever, so long as it is an element of the set $\mathcal{D}$. Finally, the hypothesis that f is continuous in $\mathcal{D}$ is essential. It is easy to construct examples of iteration functions satisfying all of the stated conditions except continuity, and for which the iterations fail to converge.

Clearly, it would be useful to be able to calculate the Lipschitz constant L for any given function f. This is not always possible, in general; however, there are some practical situations in which L can be calculated. In particular, the theorem requires only that f be continuous on $\mathcal{D}$, but if we assume further that f possesses a bounded derivative in this domain, then the *mean value theorem* gives the following for $\mathcal{D} \subset \mathbb{R}^1$:

$$f(b) - f(a) = f'(\xi)(b-a) \ , \quad \text{for some } \xi \in [a,b].$$

Then

$$|f(b) - f(a)| = |f'(\xi)| \ |b-a| \leq \left(\max_{x \in [a,b]} |f'(x)| \right) |b-a|,$$

and we take

$$L = \max_{x \in [a,b]} |f'(x)|.$$

(We comment that, for technical reasons associated with the definition of the derivative, we should take $\mathcal{D} = [a-\epsilon, b+\epsilon]$, $\epsilon > 0$, in this case.)

For $\mathcal{D} \subset \mathbb{R}^N$, the basic idea is still the same, but more involved. The ordinary derivative must be replaced by a Fréchet derivative, which in the finite-dimensional case is just the *Jacobian matrix*, and the "max" operation must be applied to the largest (in absolute value) eigenvalue of this matrix, viewed as a function of $x \in \mathcal{D} \subset \mathbb{R}^N$. We note that there are other possibilities for computing L in multi-dimensional spaces, but we will not pursue these here.

Jacobi Iteration

We are now prepared to consider the iterative solution of systems of linear equations. Thus, we again study the equation $Ax = b$. For purposes of demonstration we will consider the same general 3×3 system treated earlier in the context of Gaussian elimination:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3. \end{aligned} \tag{1.27}$$

We want to rearrange this to obtain a form suitable for application of fixed-point iteration. There are several ways in which this might be done, but the natural approach that is almost always taken is to formally "solve" each equation for its "diagonal" component; *i.e.*, solve the first equation for x_1, the second for x_2, *etc.* This leads to

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3) \equiv f_1(x_1, x_2, x_3) \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3) \equiv f_2(x_1, x_2, x_3) \\ x_3 &= \frac{1}{a_{33}}(b_3 - a_{31}x_1 - a_{32}x_2) \equiv f_3(x_1, x_2, x_3). \end{aligned}$$

It is clear that with $x = (x_1, x_2, x_3)^T$ and $f = (f_1, f_2, f_3)^T$, we have obtained the form $x = f(x)$, as desired. Moreover, the iteration scheme corresponding to successive approximation can be expressed as

$$\begin{aligned} x_1^{(m+1)} &= \frac{1}{a_{11}} \left(b_1 - a_{12} x_2^{(m)} - a_{13} x_3^{(m)} \right) \\ x_2^{(m+1)} &= \frac{1}{a_{22}} \left(b_2 - a_{21} x_1^{(m)} - a_{23} x_3^{(m)} \right) \\ x_3^{(m+1)} &= \frac{1}{a_{33}} \left(b_3 - a_{31} x_1^{(m)} - a_{32} x_2^{(m)} \right), \end{aligned} \tag{1.28}$$

where we have introduced parenthesized superscripts to denote *iteration counters* on individual subscripted solution components.

The above equations are an example of what is known as the *Jacobi iteration method* for solving linear systems. For a system of N equations, the general i^{th} equation of the iteration procedure is

$$x_i^{(m+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{ij} x_j^{(m)} \right] \equiv f_i(x_1, x_2, \ldots, x_N). \tag{1.29}$$

The algorithm for performing Jacobi iteration on a general $N \times N$ system is as follows.

Algorithm 1.4 *(Jacobi Iteration)*

1. *Set $m = 0$, load initial guess into $x^{(m)}$*
2. *Do $i = 1, N$*

$$x_i^{(m+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{ij} x_j^{(m)} \right]$$

   Repeat i

3. *Test convergence:*

$$\begin{aligned} &\textit{If } \max_i \left| x_i^{(m+1)} - x_i^{(m)} \right| < \epsilon \ , \ \textit{then stop} \\ &\qquad\qquad \textit{else set } x_i^{(m)} = x_i^{(m+1)}, \ i = 1, \ldots, N \\ &\qquad\qquad\qquad m = m + 1 \\ &\qquad\qquad\qquad \textit{go to } 2 \end{aligned}$$

It is always desirable to predict, *a priori*, whether an iteration scheme applied to a particular problem will converge. Recall that for fixed-point iteration we must show that the Lipschitz constant is less than unity in absolute value. For the case of Jacobi iteration applied to linear systems this leads to a specific requirement on the system matrix, as we will now show.

Theorem 1.4 *Let A be a $N \times N$ nonsingular matrix, and suppose Jacobi iteration is used to solve the system $Ax = b$. A sufficient condition for convergence is*

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{N} |a_{ij}| \quad \forall \ i = 1, 2, \ldots, N. \tag{1.30}$$

The condition given by inequality (1.30) is known as (strict) *diagonal dominance* of the matrix A. It has not been our previous practice to prove theorems that have been quoted. However, the proof of the above theorem provides a clear illustration of application of the contraction mapping principle, so we will present it here.

Proof. We first observe that any square matrix A can be decomposed as $A = D - L - U$, where D is a diagonal matrix with elements equal to those of the main diagonal of A, and L and U are respectively lower and upper triangular matrices with zero diagonals, and whose (nonzero) elements are the negatives of the corresponding elements of A. (Note that these matrices are not the L and U of the LU-decomposition theorem stated earlier.) Then $(D - L - U)x = b$, and

$$x^{(m+1)} = D^{-1}(L + U)x^{(m)} + D^{-1}b \equiv f\left(x^{(m)}\right). \tag{1.31}$$

It is easily checked that this fixed-point representation is exactly the form of (1.28) and (1.29).

To guarantee convergence of the iteration scheme (1.31) we must find conditions under which the corresponding Lipschitz constant has magnitude less than unity. From (1.31) it follows that the Lipschitz condition is

$$\begin{aligned}\|f(x) - f(y)\| &= \left\|D^{-1}(L + U)x + D^{-1}b - D^{-1}(L + U)y - D^{-1}b\right\| \\ &= \left\|D^{-1}(L + U)(x - y)\right\| \\ &\leq \left\|D^{-1}(L + U)\right\| \|x - y\|.\end{aligned}$$

Hence, the Lipschitz constant is

$$K = \left\|D^{-1}(L + U)\right\|,$$

so convergence of Jacobi iteration is guaranteed whenever $\|D^{-1}(L + U)\| < 1$. If we now take $\|\cdot\|$ to be the infinity-norm, then

$$K = \max_i \left[\frac{1}{|a_{ii}|}\sum_{\substack{j=1\\ j\neq i}}^{N} |a_{ij}|\right].$$

Thus, we find that in order for $K < 1$ to hold, we must have

$$|a_{ii}| > \sum_{\substack{j=1\\ j\neq i}}^{N} |a_{ij}| \quad \forall\ i = 1, 2, \ldots, N.$$

That is, diagonal dominance is required. This concludes the proof.

An important consequence of this is that, typically, only sparse systems can be solved by iteration. As is easily seen, in this case many of the terms in the summation in Eq. (1.29) are zero, and hence need not be evaluated. This, of course, reduces both storage and arithmetic in addition to providing a better opportunity for achieving diagonal dominance.

Gauss–Seidel Iteration

Recall from the algorithm for Jacobi iteration that the right-hand side of the i^{th} equation (1.29) is evaluated using only results from the previous iteration, even though more recent (and presumably more accurate) estimates of all solution components preceding the i^{th} one have already been calculated. Our intuition suggests that we might obtain a more rapidly convergent iteration scheme if we use new results as soon as they are known, rather than waiting to complete the current sweep

through all equations of the system. It turns out that this is, in fact, usually true, and it provides the basis for the method known as *Gauss–Seidel iteration*. In this scheme the general i^{th} equation is given by

$$x_i^{(m+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(m+1)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(m)} \right], \qquad i = 1, 2, \ldots, N\,. \tag{1.32}$$

We note that this is not strictly a fixed-point algorithm because both $x^{(m)}$ and $x^{(m+1)}$ appear in the iteration function. It is fairly easy, however, to effect a rearrangement that formally achieves the fixed-point form. This is important for theoretical studies but is of little value for practical computation, so we will not pursue it here.

It is worthwhile to make some comparisons with Jacobi iteration. Because all equations are updated simultaneously in Jacobi iteration, the rate of convergence (or divergence) is not influenced by the order in which individual solution components are evaluated. This is not the case for Gauss–Seidel iterations. In particular, it is possible for one ordering of the equations to be convergent, while a different ordering is divergent. A second point concerns the relationship between rates of convergence for Jacobi and Gauss–Seidel iterations; it can be shown theoretically that when both methods converge, Gauss–Seidel does so twice as fast as does Jacobi for typical problems having matrix structure such as depicted in Fig. 1.1. This is observed to a high degree of consistency in actual calculations, the implication being that one should usually employ Gauss–Seidel instead of Jacobi iterations.

Successive Overrelaxation

One of the more widely-used methods for iterative solution of sparse systems of linear equations is *successive overrelaxation* (*SOR*). This method is merely an accelerated version of the Gauss–Seidel procedure discussed above. Suppose we let x_i^* represent the i^{th} component of the solution obtained using Gauss–Seidel iteration, Eq. (1.32). Let ω be the so-called *relaxation parameter*. Then we can often improve the convergence rate of the iterations by defining the weighted average

$$\begin{aligned} x_i^{(m+1)} &= (1-\omega) x_i^{(m)} + \omega x_i^* \\ &= x_i^{(m)} + \omega \left(x_i^* - x_i^{(m)} \right) \\ &= x_i^{(m)} + \omega \Delta x_i\ . \end{aligned}$$

Then, replacing x_i^* with (1.32) leads to the general i^{th} equation for SOR:

$$x_i^{(m+1)} = x_i^{(m)} + \omega \left[\frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(m+1)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(m)} \right) - x_i^{(m)} \right], \tag{1.33}$$
$$i = 1, 2, \ldots, N\,.$$

For $1 < \omega < 2$, this corresponds to SOR; for $0 < \omega < 1$, we obtain a *damped* form of Gauss–Seidel often termed *underrelaxation*. When writing a Gauss–Seidel code one should always include the relaxation parameter as an input. Clearly Eq. (1.33), SOR, reduces to (1.32), Gauss–Seidel, when $\omega = 1$. Thus, a computer program written using (1.33) exhibits a great deal of flexibility in terms of controlling the rate of convergence of the iterations.

We now present a pseudo-language algorithm for implementing SOR.

Algorithm 1.5 *(Successive Overrelaxation)*

1. *Input* ω, ϵ *and maxitr; load initial guess into* $x^{(0)}$.

2. *Begin iterations*

   Do $m = 0,\ maxitr$

   set $maxdif = 0$

3. *Evaluate SOR formula*

   Do $i = 1,\ N$

   $$\Delta x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(m+1)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(m)} \right) - x_i^{(m)}$$

   if $|\Delta x_i| > maxdif$, *then* $maxdif = |\Delta x_i|$

   $$x_i^{(m+1)} = x_i^{(m)} + \omega \Delta x_i$$

   Repeat i

4. *Test convergence*

   if $maxdif < \epsilon$, *then print results, and stop*

   Repeat m

5. *Print message that iterations failed to converge in maxitr iterations*

A few comments should be made regarding the above pseudo-language algorithm. First, it should be noticed that we have provided a much more detailed treatment of the SOR algorithm in comparison with what was done for Jacobi iterations; this is because of the overall greater relative importance of SOR as a practical solution procedure. Second, it is written for general (nonsparse) systems of linear equations. We have emphasized that it is almost always sparse systems that are solved via iterative methods. For these, the algorithm simplifies somewhat. We will encounter cases of this later in Chap. 5. Finally, we want to draw attention to the fact that there are really two separate criteria by which the algorithm can be stopped: *i*) satisfaction of the iteration convergence tolerance, ϵ and *ii*) exceeding the maximum permitted number of iterations, *maxitr*. The second of these is crucial in a practical implementation because we do not know ahead of time whether convergence to the required tolerance can be achieved. If it happens that it cannot be (*e.g.*, due to round-off errors), then iterations would continue forever unless they are stopped due to exceeding the maximum specified allowable number.

1.2.3 Summary of methods for solving linear systems

In this subsection we will briefly summarize the foregoing discussions in the form of a table. presents a listing of the main classes of problems one encounters in solving linear systems of equations. For each type we give the preferred solution method, required storage for a typical implementation, and total floating-point arithmetic needed to arrive at the solution, all in terms of the number N of equations in the system. We note that the total arithmetic quoted for iterative solution of sparse systems is a "typical" result occurring for SOR applied with optimal relaxation parameter to solution of the discrete two-dimensional Poisson equation (see Chap. 5 for more details). In

general, for the case of sparse systems of this type, the arithmetic operation count can range from slightly greater than $\mathcal{O}(N)$ to as much as $\mathcal{O}(N^2)$, depending on the specific method employed.

A final remark on the comparison between direct and iterative methods, in general, is also in order. Although the operation count is high for a direct method applied to a nonsparse system, this count is precise: we can exactly count the total arithmetic, *a priori*, for such methods. Moreover, this amount of arithmetic leads to the exact solution to the system of equations, to within the precision of the machine arithmetic.

Table 1.1: Summary of methods for linear systems

System Matrix	Preferred Method	Storage	Arithmetic
Nonsparse	Direct elimination	$O(N^2)$	$O(N^3)$
Sparse	Iteration, *e.g.*, SOR	$O(N)$	$O(N^{1.5})$
Sparse, compactly banded	Sparse band LU decomposition	$O(N)$	$O(N)$

By way of contrast, we can never exactly predict the total arithmetic for an iterative procedure because we do not know ahead of time how many iterations will be required to achieve the specified accuracy (although in simple cases this can be estimated quite well). Furthermore, the solution obtained is, in any case, accurate only to the prescribed iteration tolerance, at best—depending on details of testing convergence. On the other hand, in many practical situations it makes little sense to compute to machine precision because the problem data may be accurate to only a few significant digits, and/or the equations, themselves, may be only approximations. All of these considerations should ultimately be taken into account when selecting a method with which to solve a given linear system.

1.3 The Algebraic Eigenvalue Problem

The second main class of problems encountered in numerical linear algebra is the *algebraic eigenvalue problem*. As noted earlier, eigenvalue problems occur somewhat less frequently than does the need to solve linear systems, so our treatment will be rather cursory. Nevertheless, in certain areas eigenvalue problems are extremely important, *e.g.*, in analysis of stability (in almost any context) and in modal analysis of structures; hence, it is important to have some familiarity with the treatment of eigenvalue problems. In this section we will begin with a brief description of the eigenvalue problem, itself. We will then give a fairly complete treatment of the simplest approach to finding eigenvalues and eigenvectors, the power method. Following this we briefly discuss what is known as inverse iteration, an approach used primarily to find eigenvectors when eigenvalues are already known. We then conclude the section with a short, mainly qualitative description of the QR method, one of the most general and powerful of eigenvalue techniques.

Eigenvalue problems are of the form

$$AX = \lambda X\,, \tag{1.34}$$

where λ is an *eigenvalue* of the $N \times N$ matrix A, and X is the *corresponding eigenvector*. It is usual to rewrite (1.34) as

$$(A - \lambda I)X = 0\,,$$

which clearly displays the homogeneity of the eigenvalue problem. As a result of this, nontrivial solutions, X, exist only for those values of λ such that $A - \lambda I$ is a singular matrix. Thus, we can find nontrivial eigenvectors for every λ such that

$$\det(A - \lambda I) = 0\,, \tag{1.35}$$

and only for such λ. It is not hard to check that (1.35) is a polynomial of degree N in λ if A is a $N \times N$ matrix. Thus, one method for finding eigenvalues of a matrix is to find the roots of this *characteristic polynomial*. If $N \leq 4$ this can be done exactly, although considerable algebraic manipulation is involved for $N > 2$. Moreover, the eigenvectors must still be determined in order to obtain a complete solution to the eigenvalue problem. We shall not consider such approaches any further, and will instead direct our attention toward numerical procedures for calculating approximate results for arbitrary finite N.

1.3.1 The power method

The *power method* is probably the simplest of all numerical methods for approximating eigenvalues. As we will see, it consists of a fixed-point iteration for the eigenvector corresponding to the largest (in magnitude) eigenvalue of the given matrix. Construction of the required fixed-point iteration can be motivated in the following way.

Let $\{X_i\}_{i=1}^N$ denote the set of eigenvectors of the $N \times N$ matrix A. We will assume that these eigenvectors are linearly independent, and thus form a basis for $\mathbb{R}^N$. Then any vector in $\mathbb{R}^N$ can be expressed as a linear combination of these eigenvectors; *i.e.*, for any $Y \in \mathbb{R}^N$ we have

$$Y = c_1 X_1 + c_2 X_2 + \cdots + c_N X_N\,,$$

where the c_is are constants, not all of which can be zero. Multiplication of Y by the original matrix A results in

$$AY = c_1 A X_1 + c_2 A X_2 + \cdots + c_N A X_N\,, \tag{1.36}$$

But since the X_is are eigenvectors, we have $AX_i = \lambda_i X_i \;\forall\; i = 1, 2, \ldots, N$. Thus (1.36) becomes

$$AY = c_1\lambda_1 X_1 + c_2\lambda_2 X_2 + \cdots + c_N\lambda_N X_N\,.$$

Now if Y is close to an eigenvector of A, say X_i, then c_i will be considerably larger in magnitude than the remaining c_js. Furthermore, if we multiply by A a second time, we obtain

$$A^2 Y = c_1\lambda_1^2 X_1 + c_2\lambda_2^2 X_2 + \cdots + c_N\lambda_N^2 X_N\,.$$

Clearly, if $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_N|$ then as we continue to multiply by A, the term corresponding to the largest eigenvalue will dominate the right-hand side; moreover, this process will be accelerated if Y is a good estimate of the eigenvector corresponding to the dominant eigenvalue.

We now observe that the heuristic discussion just given does not supply a useful computational procedure without significant modification because none of the X_is or c_is are known *a priori*. In fact, it is precisely the X_i corresponding to the largest λ that is to be determined. There are a couple standard ways to proceed; here we will employ a construct that will be useful later. From the original eigenvalue problem, $AX = \lambda X$, we obtain

$$\left\langle X^T, AX \right\rangle = \lambda \left\langle X^T, X \right\rangle,$$

or

$$\lambda = \frac{\langle X^T, AX \rangle}{\langle X^T, X \rangle}. \tag{1.37}$$

The right-hand side of (1.37) is called the *Rayleigh quotient*. If X is an eigenvector of A, we can use this formula to directly calculate the corresponding eigenvalue. We remark that (1.37) holds for any operator A acting on elements of an inner-product space; *i.e.*, if an operator and any one of its eigenvectors (eigenfunctions in the infinite-dimensional case) are known, then the associated eigenvalue can be immediately calculated.

We now employ (1.36) to construct a fixed-point iteration procedure to determine the eigenvector X corresponding to the largest eigenvalue, λ. We write

$$\begin{aligned} X^{(m)} = AX^{(m-1)} \equiv A^m Y &= c_1 \lambda_1^m X_1 + \cdots + c_N \lambda_1^m X_N \\ &= \lambda_1^m \left[\sum_{j=1}^{N} \left(\frac{\lambda_j}{\lambda_1} \right)^m c_j X_j \right]. \end{aligned} \tag{1.38}$$

Now notice that in the Rayleigh quotient for finding λ, the true eigenvectors must be replaced by the iterates defined by (1.38). Thus, the value obtained for λ is itself an iterate, defined by

$$\lambda^{(m+1)} = \frac{\left\langle {X^{(m)}}^T, AX^{(m)} \right\rangle}{\left\langle {X^{(m)}}^T, X^{(m)} \right\rangle}.$$

But from (1.38) we have

$$AX^{(m)} = X^{(m+1)}.$$

Thus, we can write the iterate of λ simply as

$$\lambda^{(m+1)} = \frac{\left\langle {X^{(m)}}^T, X^{(m+1)} \right\rangle}{\left\langle {X^{(m)}}^T, X^{(m)} \right\rangle}. \tag{1.39}$$

We now want to show that this iteration procedure based on Eqs. (1.38) and (1.39) does, in fact, converge to the largest eigenvalue of the matrix A. From (1.38) and (1.39) we have

$$\lambda^{(m+1)} = \lambda_1 \left[\frac{\displaystyle\sum_{i,j=1}^{N} \left(\frac{\lambda_i}{\lambda_1} \right)^m \left(\frac{\lambda_j}{\lambda_1} \right)^{m+1} c_i c_j \left\langle X_i^T, X_j \right\rangle}{\displaystyle\sum_{i,j=1}^{N} \left(\frac{\lambda_i}{\lambda_1} \right)^m \left(\frac{\lambda_j}{\lambda_1} \right)^m c_i c_j \left\langle X_i^T, X_j \right\rangle} \right].$$

We have assumed from the start that $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_N|$, so as $m \to \infty$ the first term in the series on the right will dominate. Thus it follows that

$$\lim_{m \to \infty} \lambda^{(m+1)} = \lambda_1 + \mathcal{O}\left((\lambda_2/\lambda_1)^m \right). \tag{1.40}$$

Hence, the sequence $\{\lambda^{(m)}\}$ converges to λ_1 as we expected. This also indirectly implies convergence of the fixed-point iteration given by Eq. (1.38). In addition, it should be noted that in the case of symmetric matrices, the order of the error term in (1.40) increases to $2m$. (For details, see Isaacson

and Keller [15].) It is this feature that makes the present approach (use of the Rayleigh quotient) preferable to the alternative that is usually given (see, *e.g.*, Hornbeck [14]).

The preceding is a rather standard approach to arriving at the convergence rate of the power method, but it does not motivate its construction as will appear in the algorithm given below. To accomplish this we return to Eq. (1.34), the eigenvalue problem, itself, expressed as

$$\lambda X = AX\,.$$

We observe that this immediately takes on the fixed-point form if we simply divide by the eigenvalue λ to obtain

$$X^{(m+1)} = \frac{1}{\lambda} AX^{(m)}\,.$$

We next observe that since eigenvectors are unique only to within multiplicative constants, it is necessary to rescale the computed estimate after each iteration to avoid divergence of the iterations; *viz.*, lack of uniqueness ultimately results in continued growth in the magnitude of X if normalization is not employed. Thus, we replace the above with

$$\frac{X^{(m+1)}}{\|X^{(m+1)}\|} = \frac{1}{\lambda} A \frac{X^{(m)}}{\|X^{(m)}\|}\,,$$

or

$$X^{(m+1)} = \frac{1}{\lambda} \frac{\|X^{(m+1)}\|}{\|X^{(m)}\|} AX^{(m)}\,. \tag{1.41}$$

This result appears rather different from the first equality in (1.38), but in fact, they are equivalent. This follows from the fact that (up to an algebraic sign) it can be shown that the eigenvalue λ is given by

$$\lambda = \lim_{m\to\infty} \frac{\|X^{(m+1)}\|}{\|X^{(m)}\|}\,. \tag{1.42}$$

Indeed, this is often used in computational algorithms with $\|\cdot\|$ taken to be the max norm because this is far more efficient on a per iteration basis than is use of the Rayleigh quotient. On the other hand, as alluded to above, use of the Rayleigh quotient for calculation of λ, as we have emphasized herein, leads to higher convergence rates for the power method. In any case, use of (1.42) in (1.41) leads to the usual fixed-point iteration form

$$X^{(m+1)} = AX^{(m)}\,,$$

which when expanded as

$$X^{(m+1)} = AX^{(m)} = A^2X^{(m-1)} = \cdots = A^{m+1}X^{(0)}$$

clearly demonstrates the rationale for the term *power method*.

We now present a pseudo-language algorithm for the power method. There is one further item that needs emphasis: it is the specific definition of $Y^{(m)}$ used in the computational algorithm. Notice in step 2, below, that this is defined as the normalization of the X vector. As we have just noted, without this normalization the magnitude of the computed quantities grows continually with each iteration, and ultimately results in uncontrollable round-off error and overflow. Computing with the normalized vector Y instead of the original vector X alleviates this problem, and moreover is completely consistent with the derivation just presented.

Algorithm 1.6 *(Power Method)*

1. *Set iteration tolerance ϵ and maximum number of iterations, maxitr;*
   Assign components of initial guess vector $X^{(0)}$ (usually taken to be all 1s).
   Set iteration counter $m = 0$.

2. *Calculate two-norm of $X^{(m)}$, and set*

$$Y^{(m)} = \frac{X^{(m)}}{\left\|X^{(m)}\right\|_2}.$$

3. *Calculate $X^{(m+1)} = AY^{(m)}$.*

4. *Form the Rayleigh quotient*

$$\lambda^{(m+1)} = \frac{\left\langle X^{(m)^T}, X^{(m+1)} \right\rangle}{\left\|X^{(m)}\right\|_2^2}.$$

5. *If $m = 0$, go to 7*

6. *Test convergence of $\lambda^{(m+1)}$:*

   If $|\lambda^{(m+1)} - \lambda^{(m)}| < \epsilon$, then print results and stop.

7. *If $m < maxitr$, then $m = m + 1$*
   go to 2
   else print error message and stop.

The power method provides the advantage of simultaneously calculating both an eigenvalue (the largest in magnitude), and a corresponding eigenvector. On the other hand, a fair amount of effort is required to modify the algorithm so that eigenvalues other than the largest can be found. In addition, because the power method is a basic fixed-point iteration, its convergence rate is only linear, as we shall show in Chapter 2. There are many other schemes for calculating eigenvalues and eigenvectors, and we will briefly describe a couple of these in the following subsections.

1.3.2 Inverse iteration with Rayleigh quotient shifts

The *inverse iteration* algorithm can be derived by viewing the algebraic eigenvalue problem, Eq. (1.34), as a system of N equations in $N + 1$ unknowns. An additional equation can be obtained by the typical normalization constraint needed to uniquely prescribe the eigenvectors, as discussed above. This $(N + 1) \times (N + 1)$ system can be efficiently solved using Newton's method (to be presented in Chap. 2). Here we will merely present the pseudo-language algorithm for inverse iteration, and refer the reader to Ruhe [25] for more details.

Algorithm 1.7 *(Inverse Iteration with Rayleigh Quotient Shifts)*

1. *Set iteration tolerance ϵ and maximum number of iterations, maxitr;*

   Load initial guess for X and λ into $X^{(0)}$, $\lambda^{(0)}$.

   Set iteration counter, $m = 0$.

2. *Normalize eigenvector estimate:*

$$Y^{(m)} = \frac{X^{(m)}}{\left\|X^{(m)}\right\|_2} .$$

3. *Use Gaussian elimination to solve the linear system*

$$\left(A - \lambda^{(m)} I\right) X^{(m+1)} = Y^{(m)}$$

for $X^{(m+1)}$.

4. *Calculate the Rayleigh quotient to update* λ*:*

$$\lambda^{(m+1)} = \frac{\left\langle {X^{(m)}}^T, X^{(m+1)} \right\rangle}{\left\|X^{(m)}\right\|_2^2} .$$

5. *Test convergence:*

If $\left(\left\|Y^{(m+1)} - Y^{(m)}\right\| + |\lambda^{(m+1)} - \lambda^{(m)}|\right) < \epsilon$,
then print results and stop.
else if $m < maxit$, *then* $m = m + 1$
go to 2
else print error message and stop.

As can be deduced from the steps of this algorithm, it has been constructed from two more basic numerical tools, both of which have already been introduced: *i*) Gaussian elimination, and *ii*) the Rayleigh quotient. Inverse iteration has the advantage that it can be used to find any eigenvalue of a matrix (not just the largest), and its corresponding eigenvector. Furthermore, it turns out that the convergence rate is at least quadratic (in a sense to be described in Chap. 2), and is cubic for symmetric matrices. The main use of this algorithm, however, is in finding eigenvectors only. It is widely used in conjunction with the QR algorithm (to be discussed in the next section) for finding all eigenvalues and eigenvectors of an arbitrary matrix. Finally, we remark that the matrix in step 3 of the algorithm is nearly singular (and hence, ill conditioned) and typically nonsparse, so it is necessary to employ high-precision arithmetic and robust pivoting strategies to guarantee accurate results when solving for $X^{(m+1)}$.

1.3.3 The QR algorithm

The QR method is one of the most efficient, and widely used, numerical algorithms for finding all eigenvalues of a general $N \times N$ matrix A. It is constructed in a distinctly different manner from our two previous methods. Namely, like a number of other procedures for computing eigenvalues, the QR method employs *similarity transformations* to isolate the eigenvalues on (or near) the diagonal of a transformed matrix. The basis for such an approach lies in the following concepts.

Definition 1.9 *Let* A *and* P *be* $N \times N$ *matrices with* P *being nonsingular. Then the matrix* $\tilde{A} \equiv PAP^{-1}$ *is said to be similar to the matrix* A*, and* $P(\,\cdot\,)P^{-1}$ *is called a similarity transformation.*

A fundamental property of similarity transformations is contained in the following:

Theorem 1.5 *The spectrum of a matrix is invariant under similarity transformations; i.e.,* $\sigma(\tilde{A}) = \sigma(A)$*, where* σ *denotes the set of eigenvalues* $\{\lambda_i\}_{i=1}^N$.

These ideas provide us with a very powerful tool for computing eigenvalues. Namely, we attempt to construct a sequence of similarity transformations that lead to diagonalization of the original matrix A. Since the new diagonal matrix has the same eigenvalues as the original matrix A (by the preceding theorem), these can be read directly from the new (transformed) diagonal matrix. The main difficulty with such an approach is that of constructing the similarity transformations. There is nothing in the above definition or theorem to suggest how this might be done, and in fact such transformations are not unique. Thus, the various algorithms utilizing this approach for computing eigenvalues can be distinguished by the manner in which the similarity transformations are obtained.

In the *QR method* this is done in the following way. It is assumed that at the m^{th} iteration the matrix $A^{(m)}$ can be decomposed as the product of a unitary matrix $Q^{(m)}$ and an upper triangular matrix $R^{(m)}$. (A unitary matrix is one whose inverse equals its transpose; *i.e.*, $Q^{-1} = Q^T$.) Hence,

$$A^{(m)} = Q^{(m)} R^{(m)}.$$

Then we calculate $A^{(m+1)}$ as $A^{(m+1)} = R^{(m)} Q^{(m)}$. It is easily checked that $A^{(m+1)}$ is similar to $A^{(m)}$ and thus has the same eigenvalues. But it is not so easy to see that the eigenvalues can be more easily extracted from $A^{(m+1)}$ than from the original matrix A. Because A need not be symmetric, we are not guaranteed that it can be diagonalized. Hence, a fairly complicated procedure is necessary in implementations of the QR method. The interested reader is referred to Wilkinson and Reinsch [38] for details.

We have already noted that the QR method is very efficient, and that it is capable of finding all eigenvalues of any given matrix. Its major disadvantage is that if eigenvectors are also needed, a completely separate algorithm must be implemented for this purpose. (Of course, this can be viewed as an advantage if eigenvectors are not needed!) Inverse iteration, as described in the preceding section, is generally used to find eigenvectors when eigenvalues are computed with a QR routine.

We will not include a pseudo-language algorithm for the QR method. As may be inferred from the above discussions, such an algorithm would be quite complicated, and it is not recommended that the typical user attempt to program this method. Well-validated, highly-efficient versions of the QR procedure are included in the numerical linear algebra software of essentially all major computing systems.

1.3.4 Summary of methods for the algebraic eigenvalue problem

In the preceding subsections we have provided a brief account of methods for solving algebraic eigenvalue problems. This treatment began with the power method, an approach that is straightforward to implement, and which can be applied to determine the largest (in magnitude) eigenvalue of a given matrix and its corresponding eigenvector. The second method considered was inverse iteration with Rayleigh quotient shifts. This technique is capable of finding an arbitrary eigenvalue and its associated eigenvector provided a sufficiently accurate initial guess for the eigenvalue is supplied. As we have already noted, this procedure is more often used to determined eigenvectors once eigenvalues have already been found using some other method. One such method for finding all eigenvalues (but no eigenvectors) of a given matrix is the QR method, the last topic of our discussions. This is a highly efficient technique for finding eigenvalues, and in conjunction with inverse iteration provides a very effective tool for complete solution of the algebraic eigenvalue problem.

In closing this section we wish to emphasize that the methods described above, although widely used, are not the only possible ones, and in some cases might not even be the best choice for a given problem. Our treatment has been deliberately brief, and we strongly encourage the reader to consult the references listed herein, as well as additional ones (*e.g.*, the classic by Wilkinson [37], and the EISPACK User's Manual [28]), for further information.

1.4 Summary

This chapter has been devoted to introducing the basics of numerical linear algebra with a distinct emphasis on solution of systems of linear equations, and a much briefer treatment of the algebraic eigenvalue problem. In the case of solving linear systems we have presented only the most basic, fundamental techniques: *i*) the direct methods—Gaussian elimination for nonsparse systems, tridiagonal LU decomposition for compactly-banded tridiagonal systems, and *ii*) the iterative schemes leading to successive overrelaxation, namely, Jacobi and Gauss–Seidel iteration, and then SOR, itself. Discussions of the algebraic eigenvalue problem have been restricted to the power method, inverse iteration with Rayleigh quotient shifts, and a very qualitative introduction to the QR method.

Our principal goal in this chapter has been to provide the reader with sufficient information on each algorithm (with the exception of the QR procedure) to permit writing a (fairly simple) computer code, in any chosen language, that would be able to solve a large percentage of the problems encountered from the applicable class of problems. At the same time, we hope these lectures will provide some insight into the workings of available commercial software intended to solve similar problems.

Finally, we must emphasize that these lectures, by design, have only scratched the surface of the knowledge available for each chosen topic—and barely even that for the algebraic eigenvalue problem. In many cases, especially with regard to iterative methods for linear systems, there are far more efficient and robust (but also much more elaborate and less easily understood) methods than have been described herein. But it is hoped that the present lectures will provide the reader with adequate foundation to permit her/his delving into the theory and application of more modern methods such as *Krylov subspace techniques, multigrid methods* and *domain decomposition*. An introduction to such material can be found in, *e.g.*, Saad [26], Hackbusch [12], Smith *et al.* [27], and elsewhere.

Chapter 2

Solution of Nonlinear Equations

In this chapter we will consider several of the most widely-used methods for solving nonlinear equations and nonlinear systems. Because, in the end, the only equations we really know how to solve are linear ones, our methods for nonlinear equations will involve (local) linearization of the equation(s) we wish to solve. We will discuss two main methods for doing this; it will be seen that they can be distinguished by the number of terms retained in a Taylor series expansion of the nonlinear function whose zeros (the solution) are being sought. Once a method of local linearization has been chosen, it still remains to construct a sequence of solutions to the resulting linear equations that will converge to a solution of the original nonlinear equation(s). Generation of such sequences will be the main topic of this chapter.

We will first present the fixed-point algorithm of Chap. 1 from a slightly different viewpoint in order to easily demonstrate the notion of convergence rate. Then we will consider Newton's method, and several of its many modifications: *i*) damped Newton, *ii*) the secant method and *iii*) regula falsi. We will then conclude the chapter with a treatment of systems of nonlinear equations via Newton's method.

2.1 Fixed-Point Methods for Single Nonlinear Equations

In this section we will derive two fixed-point algorithms for single nonlinear equations: "basic" fixed-point iteration and Newton iteration. The former is linearly convergent, in a sense to be demonstrated below, while the latter is quadratically convergent.

2.1.1 Basic fixed-point iteration

Here we consider a single equation in a single unknown, that is, a mapping F: $\mathbb{R} \to \mathbb{R}$, and require to find $x^* \in \mathbb{R}$ such that $F(x^*) = 0$. We can, as in Chap. 1, express this in fixed-point form

$$x^* = f(x^*) \tag{2.1}$$

for some function f that is easily derivable from F. If we approximate f by a Taylor series expansion about a current estimate of x^*, say $x^{(m)}$, and retain only the first term, we have

$$x^* = f(x^{(m)}) + \mathcal{O}(x^* - x^{(m)})\,.$$

Since the second term on the right cannot be evaluated without knowing x^*, we are forced to neglect it and replace x^* on the left-hand side with a new estimate $x^{(m+1)}$, resulting in the usual fixed-point form

$$x^{(m+1)} = f(x^{(m)})\,, \tag{2.2}$$

often called *Picard iteration*.

If we subtract this from (2.1) we obtain

$$x^* - x^{(m+1)} = f(x^*) - f(x^{(m)})\,. \tag{2.3}$$

Now define the error at the m^{th} iteration, e_m, by

$$e_m \equiv x^* - x^{(m)}\,. \tag{2.4}$$

Then if f is Lipschitz with Lipschitz constant L, we find that

$$|e_{m+1}| \le L|e_m|\,.$$

We see from this that at each iteration the error will be reduced by a factor L. This also shows that the error at the $(m+1)^{th}$ iteration depends linearly on the error at the m^{th} iteration. Moreover, it is easily seen that the iterations converge more rapidly when $L \ll 1$ (and diverge for $L > 1$). However, given a particular nonlinear function F whose zero is to be found, we have very little control over the value of L; it is mainly a property of the function F. So if more rapid convergence of iterations is desired, a different approach must be used. This leads us to a study of Newton methods.

2.1.2 Newton iteration

Here, we begin by deriving the Newton iteration formula via a Taylor expansion, and we show that it is in the form of a fixed-point iteration. Then we consider the convergence rate of this procedure and demonstrate that it achieves quadratic convergence. Following this we view Newton's method from a graphical standpoint, and as was the case in Chap. 1 with basic fixed-point methods, we will see how Newton's method can fail—and we introduce a simple modification that can sometimes prevent this. Finally, we present a pseudo-language algorithm embodying this important technique.

Derivation of the Newton iteration formula

We again consider the equation

$$F(x^*) = 0\,, \tag{2.5}$$

F: $\mathbb{R} \to \mathbb{R}$. But now, instead of writing this in fixed-point form, we will employ a Taylor expansion applied directly to the function F. For the m^{th} approximation to x^* we have

$$F(x^*) = F(x^{(m)}) + F'(x^{(m)})(x^* - x^{(m)}) + \mathcal{O}((x^* - x^{(m)})^2) + \cdots\,.$$

If we now note that the left-hand side must be zero by Eq. (2.5), we obtain

$$x^* \approx x^{(m)} - \frac{F\big(x^{(m)}\big)}{F'\big(x^{(m)}\big)} + \mathcal{O}((x^* - x^{(m)})^2) + \cdots\,.$$

Applying arguments analogous to those used in the preceding section, we replace x^* with $x^{(m+1)}$ on the left-hand side and drop the last term on the right-hand side to arrive at

$$x^{(m+1)} = x^{(m)} - \frac{F\big(x^{(m)}\big)}{F'\big(x^{(m)}\big)}\,. \tag{2.6}$$

This is the well-known *Newton iteration formula*.

Clearly, if we define

$$G(x^{(m)}) \equiv x^{(m)} - \frac{F\big(x^{(m)}\big)}{F'\big(x^{(m)}\big)} \tag{2.7}$$

and write

$$x^{(m+1)} = G(x^{(m)})\,,$$

we see that Newton's method is also in fixed-point form; but the iteration function is more complicated than that for Picard iteration.

It can be shown that if $F \in C^2$ and $F' \neq 0$ in an interval containing x^*, then for some $\epsilon > 0$ and $x \in [x^* - \epsilon, x^* + \epsilon]$, the Lipschitz constant for G is less than unity, and the sequence generated by (2.6) is guaranteed to converge to x^*. In particular, we have

$$G'(x^{(m)}) = 1 - \frac{F'(x^{(m)})}{F'\big(x^{(m)}\big)} + \frac{F''\big(x^{(m)}\big)\,F\big(x^{(m)}\big)}{\big[F'\big(x^{(m)}\big)\big]^2}\,.$$

Since $F \in C^2$ for $x \in [x^* - \epsilon, x^* + \epsilon]$, F'' is bounded, say by $M < \infty$. Moreover, $F' \neq 0$ on this interval, so $|G'(x^{(m)})| \leq M^* |F(x^{(m)})|$. Clearly, if $x^{(m)}$ is close to x^*, $F(x^{(m)})$ will be close to zero (by continuity of F). It follows that the Lipschitz constant is less than unity, and as a consequence of the contraction mapping principle, the iterations converge. This provides a sketch of a proof of "local" convergence of Newton's method; that is, the method converges if the initial guess is "close enough" to the solution. Global convergence (*i.e.*, starting from any initial guess) can also be proven, but only under more stringent conditions on the nature of F. The interested reader should consult Henrici [13] or Stoer and Bulirsch [30] for more details.

Convergence rate of the Newton's method

It is of interest to examine the rate of convergence of the iterations corresponding to Eq. (2.6). This is done in a manner quite similar to that employed earlier for Picard iteration. The first step is to subtract (2.6) from the Newton iteration formula evaluated at the solution to obtain

$$x^* - x^{(m+1)} = x^* - x^{(m)} + \frac{F\big(x^{(m)}\big)}{F'\big(x^{(m)}\big)}\,,$$

since $F(x^*) = 0$. We also have, as used earlier,

$$0 = F(x^*) = F(x^{(m)}) + F'(x^{(m)})(x^* - x^{(m)}) + \frac{1}{2}F''(x^{(m)})(x^* - x^{(m)})^2 + \cdots\,.$$

Thus,

$$F(x^{(m)}) = -F'(x^{(m)})(x^* - x^{(m)}) - \frac{1}{2}F''(x^{(m)})(x^* - x^{(m)})^2 - \cdots\,.$$

We substitute this into the above, and use the definition of e_m, Eq. (2.4), to obtain

$$e_{m+1} = e_m - e_m - \frac{1}{2}\frac{F''\big(x^{(m)}\big)}{F'\big(x^{(m)}\big)}e_m^2 - \cdots\,.$$

Hence, with $F \in C^2$ and $F'(x^{(m)}) \neq 0$, this reduces to

$$|e_{m+1}| = K|e_m|^2\,, \tag{2.8}$$

with $K < \infty$. So the error at the $(m+1)^{th}$ iteration of Newton's method is proportional to the square of the error at the m^{th} iteration. This gives rise to the often used terminology *quadratic convergence* of Newton's method. It is easy to see from (2.8) that once the error is $\leq \mathcal{O}(1)$ it decays very rapidly. In fact, once the error is reduced to single precision machine ϵ, only one additional iteration is required to achieve double precision accuracy. This very rapid convergence is one of the main reasons for the wide use of Newton's method.

Graphical interpretation of Newton's method

It is worthwhile to examine Newton's method from a geometric standpoint, just as we did for fixed-point iteration in Chap. 1. Consider the following figure depicting a sequence of Newton iterations. Geometrically, the iterations proceed as follows. Starting from an initial guess $x^{(0)}$, "evaluate" F at $x^{(0)}$. Then from the point $\big(x^{(0)}, F(x^{(0)})\big)$ draw a tangent $\big($corresponding to $F'\big(x^{(0)}\big)\big)$ to intersect the x-axis. This point of intersection is the next iterate, $x^{(1)}$. It is of interest to note that if F is linear, Newton's method converges in a single iteration, independent of the initial guess, since the slope through the point $\big(x^{(0)}, F(x^{(0)})\big)$ is the graph of F, itself. Furthermore, it is easily seen from the graphical construction that convergence is very rapid, even when F is not linear, in agreement with what we would expect from (2.8).

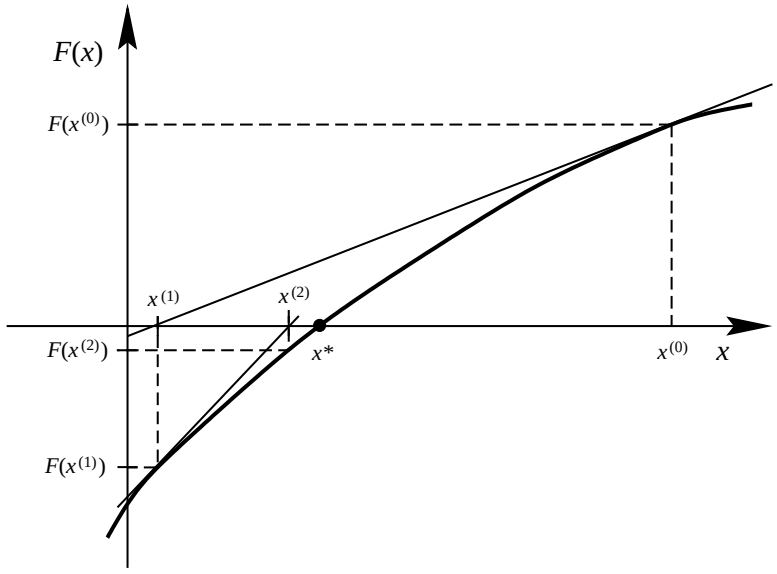


Figure 2.1: Geometry of Newton's method

Newton's method is one of the most widely used of all root-finding algorithms. Nevertheless, it can sometimes fail unexpectedly. Here, we provide an example of such a failure, and demonstrate a useful remedy. Suppose we wish to solve

$$F(x) = \tanh x = 0\,.$$

It is easily checked that $F \in C^\infty(\mathbb{R})$, and that $F' \neq 0$, except at $\pm\infty$. Furthermore, we of course know that there is only a single, simple zero, $x = 0$, as indicated in Fig. 2.2. If we start at $x = x^{(0)}$, the tangent constructed at $\big(x^{(0)}, F(x^{(0)})\big)$ intersects the x-axis at $x^{(1)}$ where $F'(x^{(1)}))$ is rather small. The next iteration would result in an even smaller value $F'(x^{(2)})$, and the iterations diverge

rapidly. This unfortunate behavior occurs in this particular problem simply because the *Newton step*,

$$\Delta x = -\frac{F(x)}{F'(x)}\,,$$

is too large, since F' is very small and $F \approx 1$. The remedy is to replace Δx with $\delta\Delta x$, with $0 < \delta \leq 1$.

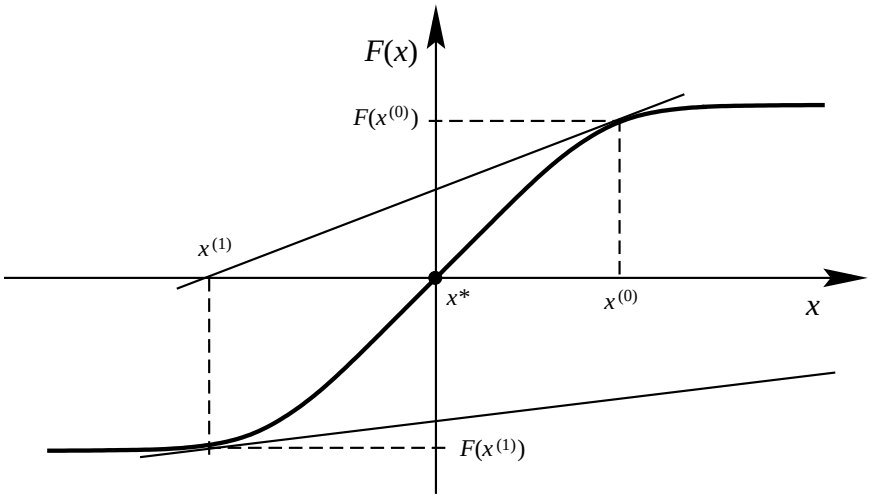


Figure 2.2: Newton's method applied to $F(x) = \tanh x$

In the particular construction of Fig. 2.2, if $\delta \lesssim 2/3$ is used, quite rapid convergence ensues. Thus, we replace Eq. (2.6) with

$$x^{(m+1)} = x^{(m)} - \delta\frac{F\big(x^{(m)}\big)}{F'\big(x^{(m)}\big)}\,, \qquad 0 < \delta \leq 1\,. \tag{2.9}$$

It should be observed, however, that with this so-called *damping*, Newton's method no longer exhibits the quadratic convergence proven earlier unless $\delta = 1$. In general, it converges only linearly when damping is employed. Proof of this is left as an exercise for the reader.

Pseudo-language algorithm for Newton's method

We close this section by presenting a pseudo-language algorithm for implementing Newton's method.

Algorithm 2.1 *(Newton's Method)*

1. *Input* δ, ϵ, *maxitr and initial guess,* $x^{(0)}$*; set* $m = 0$.
2. *Evaluate* $F(x^{(m)})$ *and* $F'(x^{(m)})$.
3. *Form the Newton step,*
$$\Delta x = -\frac{F\big(x^{(m)}\big)}{F'\big(x^{(m)}\big)}\,,$$
4. *Update solution estimate:*
$$x^{(m+1)} = x^{(m)} + \delta\Delta x\,.$$

5. *Test convergence:*

$$\begin{aligned} &\textit{if } |\Delta x| < \epsilon, \textit{ then print results and stop} \\ &\qquad \textit{else if } m < maxitr, \textit{ then } m = m+1 \\ &\qquad\qquad \textit{go to } 2 \\ &\qquad \textit{else print error message and stop} \end{aligned}$$

We note that convergence tests other than the one employed in the above algorithm may be used. Philosophically, at least, we would expect to have $|F(x^{(m)})| < \epsilon$ since we are seeking $x^* \ni F(x^*) = 0$. Nevertheless, the test given above is probably the most widely used because one can argue that once Δx becomes sufficiently small, additional corrections will have no effect, and will thus be a waste of arithmetic. But the following should be noted. Clearly, if $|F'|$ is very small but nonzero close to the desired root, the test used in the above algorithm may be unreasonably conservative. On the other hand, if $|F'| \gg |F|$, convergence may be indicated for an inaccurate approximation to the root. It is thus sometimes useful to employ a test of the form $\max(|\Delta x|, |F|) < \epsilon$.

2.2 Modifications to Newton's Method

One of the main objections to the use of Newton's method has always been the requirement to calculate the derivative of the function F whose zero is sought. In some situations this is merely tedious, but in others either the derivative does not exist at all, or it cannot be expressed in terms of elementary functions. In the former case, it may be advisable to reformulate the problem so that the new function is differentiable, or possibly an iteration scheme not requiring derivatives (or their approximations) might be used. On the other hand, if difficulty in evaluating the derivative of F is the only problem, the methods to be presented in this section should be of value.

2.2.1 The secant method

These first of these methods to be considered is the *secant method*. To construct the iteration function for this method we begin with the Newton iteration formula, Eq. (2.6):

$$x^{(m+1)} = x^{(m)} - \frac{F\left(x^{(m)}\right)}{F'\left(x^{(m)}\right)} \ ,$$

and replace F' with an approximation that depends only on values of F, itself. In the next chapter we will consider several such approximations in fair detail. The secant method is typically constructed using a simple, intuitive one. In particular, we use

$$F'(x^{(m)}) = \frac{F\left(x^{(m)}\right) - F\left(x^{(m-1)}\right)}{x^{(m)} - x^{(m-1)}} + \mathcal{O}(x^m - x^{(m-1)}) \ ,$$

and the iteration formula becomes

$$\begin{aligned} x^{(m+1)} &= x^{(m)} - \frac{\left(x^{(m)} - x^{(m-1)}\right) F\left(x^{(m)}\right)}{F\left(x^{(m)}\right) - F\left(x^{(m-1)}\right)} \\ &= x^{(m)} + \Delta x \ . \end{aligned} \tag{2.10}$$

There are several things to notice with regard to this iteration scheme. The first is that it is not a fixed-point method because the iteration function depends on more than just the most recent

approximation to x^*. (It is sometimes called a fixed-point method with "memory.") Nevertheless, the behavior of the secant method is quite similar to that of Newton iterations. The convergence rate is nearly quadratic with error reduction from one iteration to the next proportional to the error raised to approximately the 1.6 power. (Proof of this can be found in standard numerical analysis texts, *e.g.*, Stoer and Bulirsch [30].) Second, if we always keep two successive values of F in storage, then only one function evaluation is required per iteration, after the first iteration. This is in contrast to two evaluations (the function and its derivative) needed for Newton's method. Finally, it is necessary to supply two initial guesses for the root being approximated. It is often sufficient to merely select these as estimated upper and lower bounds for the solution, but other approaches are also used; for example one might start the secant iterations with an initial guess plus one fixed point iteration employing this initial guess to obtain the second required guess.

Figure 2.3 presents a schematic of the application of secant method iterations. We note that

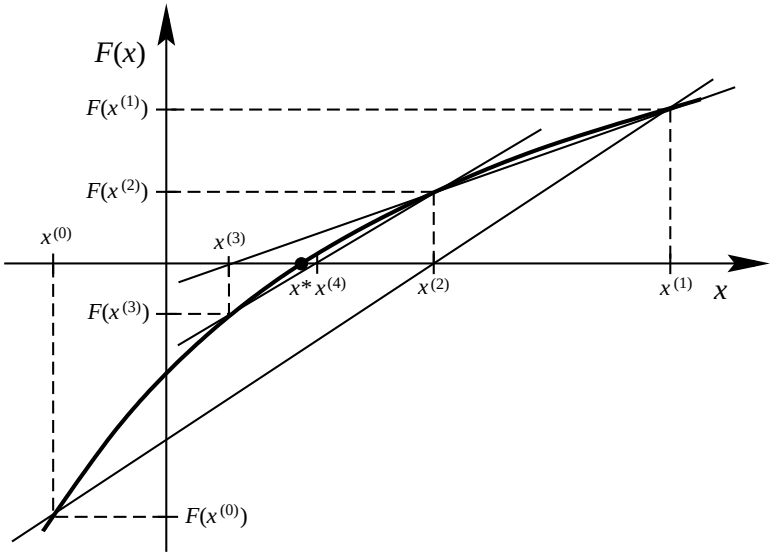


Figure 2.3: Geometry of the secant method

many of the difficulties encountered with Newton's method can also occur with the secant method, and that one of the remedies is use of a damping factor. In particular, we can replace Δx with $\delta\Delta x$ in Eq. (2.10), as was done in Eq. (2.6) for Newton's method. An additional problem with the secant method, not present in Newton's method, occurs as the zero of F is approached. Namely, $F(x^{(m)}) - F(x^{(m-1)})$ can approach zero, leading to floating-point overflow during evaluation of Eq. (2.10). The main remedy for this problem is use of higher-precision machine arithmetic.

The algorithm for the secant method is very similar to the Newton algorithm, with the exception of the definition of Δx. Hence, we shall not present a formal pseudo-language treatment for this case, and instead leave this construction as an exercise for the reader.

2.2.2 The method of false position

For one reason or another, it is sometimes necessary to bound the interval on which a root of an equation is sought. Usually, but not always, this is because the equation has more than one zero, and a particular one may be desired. For example, if we wish to find all zeros of a polynomial of degree n and we have already found some of them, we would want to restrict the intervals on which

further roots are sought. One procedure for doing this is the *method of false position*, or *regula falsi*. This method is rather similar to the secant method in that it does not require evaluation of derivatives. But at the same time, in contrast to the secant method, the derivative appearing

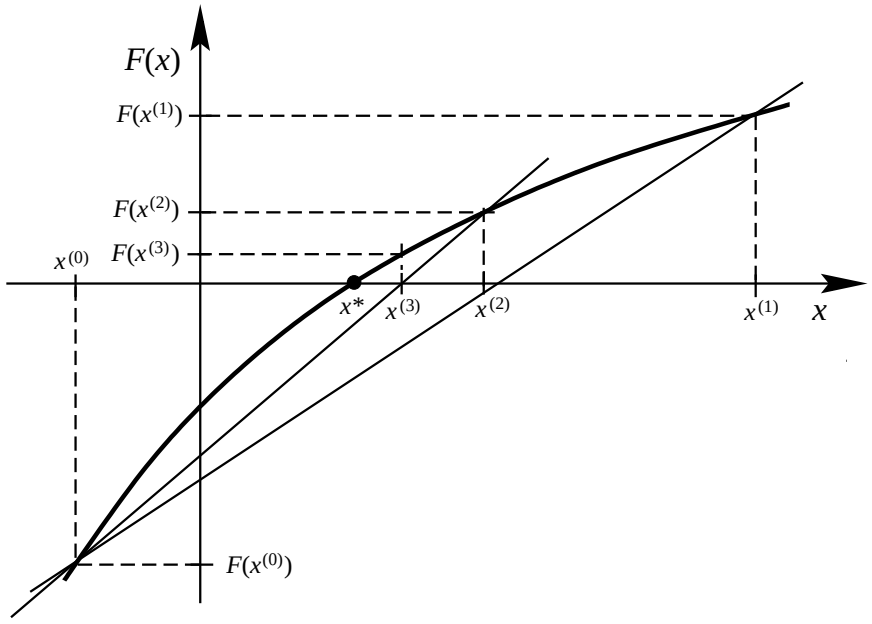


Figure 2.4: Geometry of regula falsi

in Newton's method is not formally approximated. Instead of using a tangent (as in Newton's method) or a local secant (as in the secant method) to determine the next approximation to the root, regula falsi employs a *chord*, or in some sense a nearly global secant.

Probably the best way to understand the regula falsi algorithm is through its geometry. We present this in Fig. 2.4. As can be seen from this figure, the geometry of regula falsi is similar to that of the secant method. But in regula falsi one of the end points of a prescribed interval is always used in constructing the secant. Recall that in the secant method this occurs only for the first two iterations. It is this property of regula falsi that permits bounding of the solution, but at the same time it can greatly degrade the convergence rate.

We will now present the regula falsi algorithm for finding a solution to the equation $F(x) = 0$ on the interval $[a, b]$.

Algorithm 2.2 *(Regula Falsi)*

1. *Input* ϵ, *maxitr*, a *and* b*; set* $m = 2$.

2. *Evaluate* $F(a)$ *and* $F(b)$.

3. *Calculate the first estimate of* x^**:*

$$x^{(2)} = b - \frac{(b-a)F(b)}{F(b) - F(a)}.$$

4. *Evaluate* $F(x^{(m)})$*; if* $|F(x^{(m)})| < \epsilon$*, then print result and stop*

5. *Calculate* $x^{(m+1)}$*:*

If $F(x^{(m)}) \cdot F(a) < 0$*, then*

$$x^{(m+1)} = x^{(m)} - \frac{\left(x^{(m)} - a\right) F\left(x^{(m)}\right)}{F\left(x^{(m)}\right) - F(a)} \ .$$

If $F\left(x^{(m)}\right) \cdot F(b) < 0$*, then*

$$x^{(m+1)} = b - \frac{\left(b - x^{(m)}\right) F(b)}{F(b) - F\left(x^{(m)}\right)} \ .$$

6. *If* $m < maxitr$*, then* $m = m + 1$
go to 4
else print error message and stop

An interesting point to make regarding this algorithm is that, unlike the secant method, this is a fixed-point iteration with iteration function

$$f\left(x^{(m)}\right) = \begin{cases} x^{(m)} - \dfrac{\left(x^{(m)} - a\right) F\left(x^{(m)}\right)}{F\left(x^{(m)}\right) - F(a)} & \text{if } F\left(x^{(m)}\right) \cdot F(a) < 0 \,, \\ \\ b - \dfrac{\left(b - x^{(m)}\right) F(b)}{F(b) - F\left(x^{(m)}\right)} & \text{if } F\left(x^{(m)}\right) \cdot F(b) < 0 \ . \end{cases} \tag{2.11}$$

As might be expected from earlier discussions, it can be shown that this iteration function results in a linear convergence rate. Hence, it is not very efficient. On the other hand, if a zero exists on a specified interval, and F is continuous, the regula falsi method is guaranteed to find it.

2.3 Newton's Method for Systems of Equations

In Chap. 1 we derived a basic fixed-point iteration scheme to treat systems of equations from the start because we wished to apply it to linear systems. In the present chapter we began by considering only a single nonlinear equation because convergence rate estimates are more easily obtained for this case. But in practice we very often must face the task of solving systems of nonlinear equations. Rather typically, Picard iteration is not suitable because either convergence is too slow, or it does not occur at all. Because of this, Newton's method is the most widely-used approach for treating nonlinear systems. It can be shown (via the Newton–Kantorovich theorem, see *e.g.*, Stoer and Bulirsch [30]) that Newton iterations still converge quadratically for nonlinear systems, just as for a single equation. Here, we will present a detailed construction of the method and a pseudo-language algorithm for its implementation.

2.3.1 Derivation of Newton's Method for Systems

While this method can be derived via a formal Taylor series expansion as was used to obtain the 1-D iteration earlier, in multi-dimensional cases this is somewhat tedious, and we will here simply argue from analogy with the 1-D iteration formula, Eq. (2.6).

Let $F\colon\ \mathbb{R}^N \to \mathbb{R}^N$ be in $C^2(\mathcal{D})$, $\mathcal{D} \subset \mathbb{R}^N$, and suppose we seek $x^* \in \mathcal{D} \ni F(x^*) = 0$. Recall that for $N = 1$ we can find x^* via the iterations given by Eq. (2.6); that is

$$x^{(m+1)} = x^{(m)} - \frac{F\left(x^{(m)}\right)}{F'\left(x^{(m)}\right)} \ ,$$

provided $F' \neq 0$ and F'' is bounded for x near x^*. To generalize this to the case $N > 1$, we need a reasonable interpretation of the derivative of mappings $F\colon\ \mathbb{R}^N \to \mathbb{R}^N$. It turns out that the *Jacobian matrix* of F is the correct interpretation of the derivative for this case. Hence, if we let $x = (x_1,\ x_2,\ \ldots,\ x_N)^T$ and $F = (F_1,\ F_2,\ \ldots,\ F_N)^T$, then the Newton iteration scheme becomes

$$\begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_N \end{bmatrix}^{(m+1)} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_N \end{bmatrix}^{(m)} - \begin{bmatrix} \dfrac{\partial F_1}{\partial x_1} & \dfrac{\partial F_1}{\partial x_2} & \cdot\ \cdot\ \cdot & \dfrac{\partial F_1}{\partial x_N} \\ \dfrac{\partial F_2}{\partial x_1} & \dfrac{\partial F_2}{\partial x_2} & \cdot\ \cdot\ \cdot & \dfrac{\partial F_2}{\partial x_N} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \dfrac{\partial F_N}{\partial x_1} & \dfrac{\partial F_N}{\partial x_2} & \cdot\ \cdot\ \cdot & \dfrac{\partial F_N}{\partial x_N} \end{bmatrix}^{-1} \begin{bmatrix} F_1 \\ F_2 \\ \cdot \\ \cdot \\ \cdot \\ F_N \end{bmatrix}^{(m)} \ . \tag{2.12}$$

In more concise notation this is

$$\begin{aligned} x^{(m+1)} &= x^{(m)} - \left[J\left(F^{(m)}\right)\right]^{-1} F^{(m)} \\ &= x^{(m)} + \Delta x \ , \end{aligned} \tag{2.13}$$

where $J(F)$ denotes the Jacobian matrix of F, shown in detail on the right-hand side of (2.12). From this it follows (by definition) that

$$\Delta x \equiv - \left[J\left(F^{(m)}\right)\right]^{-1} F^{(m)} \ .$$

It should be noted that, contrary to what might at first appear to be the case, it is <u>not</u> necessary to compute the inverse Jacobian matrix when using Newton's method. In particular, upon multiplying the above expression by $J\left(F^{(m)}\right)$ we obtain

$$J\left(F^{(m)}\right) \Delta x = -F^{(m)} \ . \tag{2.14}$$

Once $x^{(m)}$ is known—for example, $x^{(0)}$ is specified at the start—$F^{(m)}$ can be evaluated (this is just the system to be solved), and $J\left(F^{(m)}\right)$ can be directly calculated since this simply consists of the partial derivatives of F evaluated at $x^{(m)}$. Thus, Eq. (2.14) is a <u>linear</u> system for Δx, with <u>known</u> system matrix and right-hand side. It can be solved using Gaussian elimination described in Chap. 1. After Δx has been determined, we can immediately obtain the updated approximation $x^{(m+1)}$ to x^* from the second line in (2.13). We see from this that solving systems of nonlinear equations mainly requires that we only be able to solve systems of linear equations.

We note that, in general, damping may be required just as for a single equation. But it can now be introduced in several ways. First, one can use the same amount of damping for every equation by multiplying each component of Δx by the scalar constant $0 < \delta \leq 1$. One might also employ a damping vector (formally, a diagonal matrix) which, in general, would allow damping of each component of the solution by a different amount. Finally, even a damping matrix, say D, might be utilized. Although the latter two alternatives are seldom used, it is important to be aware of these possibilities because they can be valuable in special situations.

2.3.2 Pseudo-language algorithm for Newton's method for systems

We close this section with the pseudo-language algorithm for applying Newton's method to systems of nonlinear equations.

Algorithm 2.3 *(Newton's Method for Systems)*

1. *Input* δ, ϵ *and maxitr; initialize solution vector* $x^{(0)}$*; set* $m = 0$.

2. *Evaluate the vector* $F(x^{(m)})$.

3. *Test convergence:*

   If $\|F\| < \epsilon$, *then print results and stop.*

   else evaluate elements of Jacobian matrix, $J\left(F(x^{(m)})\right)$.

4. *Solve the linear system*

$$J\left(F^{(m)}\right)\Delta x = -F^{(m)}$$

   for Δx *using Gaussian elimination.*

5. *Update approximation to solution vector:*

$$x^{(m+1)} = x^{(m)} + \delta\Delta x$$

6. *If* $m <$ *maxitr*, *then* $m = m + 1$

   go to 2

   else print error message and stop

We remark that the structure of Algorithm 2.3 is somewhat different from that of the corresponding single-equation case, Algorithm 2.1. In particular, we have moved the convergence test ahead of the derivative evaluation in the present algorithm to avoid the expense of an additional evaluation of the Jacobian matrix. It should also be observed that Gaussian elimination used to solve for Δx in step 4 might be replaced by other linear equation solvers, as appropriate. For example, if $J(F)$ is tridiagonal, then tridiagonal LU decomposition would be the correct choice of solver.

2.4 Summary

In this chapter we have considered several of the main methods in wide use for solving nonlinear algebraic equations and systems. The procedures treated here included basic fixed-point iteration already encountered in Chap. 1 for linear equations, Newton's method, the secant method and regula falsi for single equations, and Newton's method for systems. It is clear from the above discussions that the basic fixed-point iterations represent the simplest approach, but they are not usually preferred because of their rather poor convergence rate. Newton's method is probably the most popular technique for solving nonlinear equations because of its quadratic convergence rate. But it sometimes must be damped if poor initial guesses are used. It should also be mentioned that Newton's method is sometimes started with a Picard iteration to improve the initial guess. The secant method is formally the most effective of the approaches we have considered here in the sense that its convergence rate is nearly that of Newton's method, but it requires only a single

function evaluation per iteration. Regula falsi provides the advantage of bounding the domain in which a solution is sought, but as noted above, it is a basic fixed-point iteration; so it converges only linearly.

We have treated only Newton's method for systems of nonlinear algebraic equations. It was seen that this is a natural extension of the single equation case, and this along with its quadratic convergence rate contribute to its popularity. Its main disadvantage is its requirement of calculating Jacobian matrix elements. There are a number of so-called *quasi-Newton* methods (which, in a sense, extend the secant method to systems) that are often used, especially in the context of nonlinear optimization, to alleviate this difficulty. However, it is seldom that full quadratic convergence rates are achieved by these techniques. We refer the interested reader to [30] for further discussions on this topic.

Finally, we wish to emphasize that there are numerous other procedures for solving nonlinear algebraic equations, and the choice of methods presented here at least in part reflects a bias of the author. Conspicuous by its absence is the *bisection method*. This is usually the first method treated in elementary texts because of its intuitive appeal. But its convergence rate is very low, and it is quite difficult to extend to use for systems of equations; hence, we have chosen to ignore it in these lectures. We have also neglected to discuss methods designed specifically for finding zeros of polynomials. The methods we have presented are capable of doing this, but other approaches can be far more efficient in specific situations. The reader should consult books devoted specifically to solution of nonlinear equations by, for example Ostrowski [22] or Traub [36] for information on various of these important and interesting subjects.

Chapter 3

Approximation Theory

In numerical analysis there are three main mathematical objects which we must often approximate: *i*) functions, *ii*) integrals of functions, and *iii*) derivatives of functions. We have already seen a case of the last of these in contructing the secant method for solving nonlinear equations. In this chapter we will treat all three in some detail; but in all cases we will study only rather basic and widely used methods. In the approximation of functions we consider least-squares methods briefly, and then devote some time to exact Lagrange interpolation and construction of cubic splines. Approximation of integrals will be done via the trapezoidal rule, Simpson's rule and Gauss–Legendre quadrature for integrals on subsets of the real line. We then show how these can be applied recursively to evaluate multiple integrals. Following this we will present methods for approximating derivatives, mainly via the finite-difference approach. In a final section we consider one of the most important, yet too often neglected, topics in numerical analysis, namely grid function convergence. We will see that, at least in a limited way, we are able to assess the accuracy of our approximations based only on the computed results, themselves.

3.1 Approximation of Functions

We begin this section with the least-squares method, for the most part because this provides a good application of Newton's method for systems of equations, which was just presented at the end of the preceding chapter. Then Lagrange interpolation is discussed. This method is widely used, especially when abscissas are not equally spaced. We then discuss cubic spline interpolation. This will provide the first concrete application of tridiagonal LU decomposition presented in Chap. 1.

3.1.1 The method of least squares

Least-squares methods are among the most widely-used techniques in the analysis of experimental data. In later sections we will consider the so-called exact approximation methods which force the interpolation function to exactly equal the data at certain prescribed points. This is generally inappropriate in analyses of data from almost any experiment since such data are usually of only limited accuracy. Least-squares methods will not generally lead to exact reproduction of the data value at any particular point. On the other hand, they provide the best possible fit of all the data in a certain global sense.

The reader may have previously encountered the use of *least-squares* procedures under the name *linear regression*, or multiple regression. The linear regression formulas are derived from the same basic principles we will use here. But when data are to be fit to linear functions, specific use can be

made of linearity to simplify the computational formulas. In the present treatment we will assume the data are to be fit to nonlinear functions (*i.e.*, nonlinear in the parameters to be determined), and no simplifications will be made. Our method will still be valid in the linear case, but it may be computationally less efficient than would be use of linear regression formulas.

Some General Practical Considerations

As a starting point we consider the graph of some experimental data, displayed in Fig. 3.1. It may be desirable for any of a number of reasons to find a single function with which to represent these data. The first step in the general nonlinear case is to select a functional form that exhibits many of the qualitative features of the data. It is always a good idea to first plot the data to gain some insight into the choice of function for the attempt at curve fitting. In the present case, the data are somewhat suggestive of a damped sine wave, but with phase shift, and possibly x-varying frequency. We could no doubt fit these data reasonably well by utilizing a polynomial of degree at least three (why degree three?). Often there may be theoretical reasons for choosing a particular function with which to fit the data; but in the absence of any theory, a polynomial of sufficient degree to accommodate all of the zeros and local maxima and/or minima is a reasonable choice. For the data in Fig. 3.1, we might on intuitive grounds choose

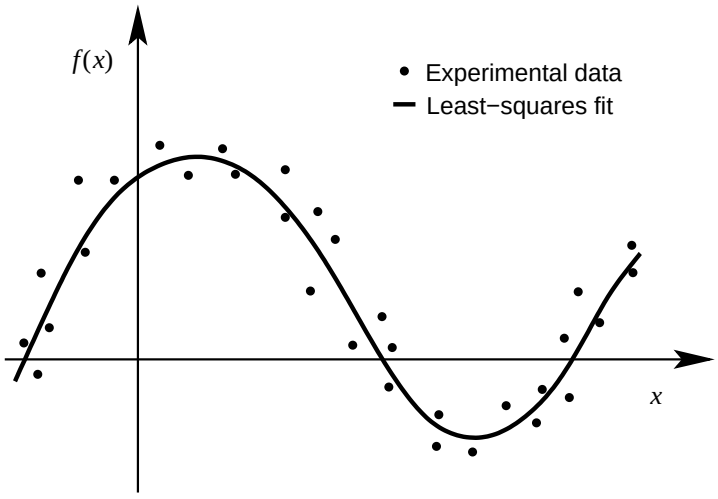


Figure 3.1: Least-squares curve fitting of experimental data

$$g(x) = Ae^{\beta x}\sin(\alpha x + \gamma) + B \tag{3.1}$$

to perform the least-squares correlation. In this expression α, β, γ, A and B are all unknown parameters to be determined from the data via the least-squares method. It is not unusual to fit data to this number of parameters, and it is worthwhile in the present case to examine just what influence each of these has on the value of the function g. The parameter A is the amplitude of the sine function with respect to the mean value, B. For the data shown in Fig. 3.1 we would probably estimate $B \simeq 0$ and A less than or equal to the absolute maximum data point value. The parameter β is the exponential decay rate, which can be easily estimated from the data; γ is the phase shift, which should be approximately $\pi/2$ in the present case. Finally, α is the frequency, which again can be estimated from specific data. It is important to be able to make reasonably

good estimates of the parameters for a nonlinear curve fit of data because these are needed as initial guesses in the iteration scheme (usually Newton's method) that must be employed to solve the nonlinear equations whose solutions are the parameter values.

Least-Squares Method Construction

We now consider more details of how to determine the parameters of the least-squares fit. We first need to make clear what it is that we wish to accomplish. Suppose there are N data pairs $(x_i, f(x_i)),\ i = 1, \ldots, N$ that we wish to represent by the function g. This function, of course, depends on x, but it also depends on the five unknown parameters α, β, γ, A and B. Let the vector $\eta = (\eta_1, \eta_2, \ldots, \eta_M)^T$ formally represent these parameters, and express g as $g(x, \eta)$. The goal of a least-squares correlation (fit) of data is to select the components η_i of the *parameter vector* such that the sum of the squared differences $[g(x_i, \eta) - f(x_i)]^2$ over all data points is as small as possible. In other words, we want to choose η to minimize

$$S(x, \eta) = \sum_{i=1}^{N} [g(x_i, \eta) - f(x_i)]^2 . \tag{3.2}$$

$S(x, \eta)$ is often referred to as the *least-squares functional*, and we should observe that it is actually a function of only the parameter vector η—that is, dependence on x has been "summed out."

We know from vector calculus that the minimum of a function occurs at the point where its gradient is zero, and at the same time its Hessian matrix (the matrix of second partial derivatives of $S(\eta)$) is positive definite. We shall assume, for convenience that the latter holds, but we note that this is not automatic. We now view S as a function only of η since, as already noted, performing the summation removes the x dependence. Then the minima of S occur for those η such that

$$\nabla_\eta S = 0 \ ;$$

that is for η such that

$$\frac{\partial S}{\partial \eta_j} = 0, \quad \forall\ j = 1, 2, \ldots, M . \tag{3.3}$$

Equations (3.3) comprise a system of M nonlinear equations called the *normal equations*, for the $M(< N)$ unknown parameters, η_i (in the case of Eq. (3.1), $M = 5$). Thus, we expect to be able to solve these via Newton's method. Define

$$F_j(\eta_1, \eta_2, \ldots, \eta_M) \equiv \frac{\partial S}{\partial \eta_j}, \quad \forall\ j = 1, 2, \ldots, M \ ;$$

then for $F = (F_1, \ldots, F_M)^T$ we have

$$\eta^{(m+1)} = \eta^{(m)} - \left[J\left(F^{(m)}\right) \right]^{-1} F^{(m)}. \tag{3.4}$$

From Eq. (3.2) it follows that

$$\frac{\partial S}{\partial \eta_j} = 2 \sum_{i=1}^{N} \frac{\partial g}{\partial \eta_j} [g(x_i, \eta) - f(x_i)] = 0, \quad \forall\ j = 1, 2, \ldots, M . \tag{3.5}$$

A second differentiation of this result leads to the Jacobian matrix required for application of Newton's method, which is the Hessian matrix of the least-squares functional:

$$\frac{\partial F_j}{\partial \eta_k} = \frac{\partial^2 S}{\partial \eta_j \partial \eta_k} = 2 \sum_{i=1}^{N} \left[\frac{\partial^2 g}{\partial \eta_j \partial \eta_k} (g(x_i, \eta) - f_i) + \frac{\partial g}{\partial \eta_j} \frac{\partial g}{\partial \eta_k} \right], \qquad j, k = 1, 2, \ldots, M . \tag{3.6}$$

Detailed Example

Our main task in constructing the least-squares formulation becomes one of finding the analytical expressions for the components of F and $J(F)$. We will carry out part of this calculation for the function given in Eq. (3.1) to provide a concrete demonstration of the technique.

Now in our example function given in Eq. (3.3), $\eta_1 = \alpha$, $\eta_2 = \beta$, $\eta_3 = \gamma$, $\eta_4 = A$, and $\eta_5 = B$. Thus,

$$\begin{aligned}
\frac{\partial g}{\partial \eta_1} &= \frac{\partial g}{\partial \alpha} = x_i A e^{\beta x_i} \cos(\alpha x_i + \gamma) \\
\frac{\partial g}{\partial \eta_2} &= \frac{\partial g}{\partial \beta} = x_i A e^{\beta x_i} \sin(\alpha x_i + \gamma) \\
\frac{\partial g}{\partial \eta_3} &= \frac{\partial g}{\partial \gamma} = A e^{\beta x_i} \cos(\alpha x_i + \gamma) \\
\frac{\partial g}{\partial \eta_4} &= \frac{\partial g}{\partial A} = e^{\beta x_i} \sin(\alpha x_i + \gamma) \\
\frac{\partial g}{\partial \eta_5} &= \frac{\partial g}{\partial B} = 1 \,.
\end{aligned}$$

We now use the common shorthand notation $f(x_i) = f_i$ that is used extensively in numerical analysis and write the system of five equations to be solved for $\eta = (\alpha, \beta, \gamma, A, B)^T$ by Newton's method:

$$\begin{aligned}
F_1(\eta) &= \sum_{i=1}^{N} \left\{ x_i A e^{\beta x_i} \cos(\alpha x_i + \gamma) \left[A e^{\beta x_i} \sin(\alpha x_i + \gamma) + B - f_i \right] \right\} = 0 \\
F_2(\eta) &= \sum_{i=1}^{N} \left\{ x_i A e^{\beta x_i} \sin(\alpha x_i + \gamma) \left[A e^{\beta x_i} \sin(\alpha x_i + \gamma) + B - f_i \right] \right\} = 0 \\
F_3(\eta) &= \sum_{i=1}^{N} \left\{ A e^{\beta x_i} \cos(\alpha x_i + \gamma) \left[A e^{\beta x_i} \sin(\alpha x_i + \gamma) + B - f_i \right] \right\} = 0 \\
F_4(\eta) &= \sum_{i=1}^{N} \left\{ e^{\beta x_i} \sin(\alpha x_i + \gamma) \left[A e^{\beta x_i} \sin(\alpha x_i + \gamma) + B - f_i \right] \right\} = 0 \\
F_5(\eta) &= \sum_{i=1}^{N} \left\{ A e^{\beta x_i} \sin(\alpha x_i + \gamma) + B - f_i \right\} = 0 \,.
\end{aligned}$$

The next step is to derive the Jacobian matrix $J(F)$ of the system, corresponding to Eq. (3.6). We already have the expressions required to construct the second term in brackets in these equations. All that remains is to derive the second partial derivatives of g. We will carry this out

only for F_1 ($= \partial g/\partial \eta_1 = \partial g/\partial \alpha$).

$$\begin{aligned}
\frac{\partial^2 g}{\partial \alpha^2} &= -x_i^2 A e^{\beta x_i} \sin(\alpha x_i + \gamma) \\
\frac{\partial^2 g}{\partial \alpha \partial \beta} &= x_i^2 A e^{\beta x_i} \cos(\alpha x_i + \gamma) \\
\frac{\partial^2 g}{\partial \alpha \partial \gamma} &= -x_i A e^{\beta x_i} \sin(\alpha x_i + \gamma) \\
\frac{\partial^2 g}{\partial \alpha \partial A} &= x_i e^{\beta x_i} \cos(\alpha x_i + \gamma) \\
\frac{\partial^2 g}{\partial \alpha \partial B} &= 0\,.
\end{aligned}$$

We leave the remaining calculations to the reader, and note that this might typically be done using symbolic manipulation languages such as Maple or Macsyma, for example.

The algorithm for solving least-squares problems merely consists of the Newton's method algorithm of Chap. 2 in which the nonlinear equations are evaluated using Eq. (3.5) (with the factor of 2 deleted), and Eqs. (3.6) are employed to construct the required Jacobian matrix. Thus, we shall not present additional details; but it is worth mentioning that the same procedure can be employed for minimization of general nonlinear functions. Its application is not restricted to least-squares problems.

3.1.2 Lagrange interpolation polynomials

The next method to be considered for function approximation is Lagrange interpolation. We should all be familiar with linear interpolation; we will see that this is the simplest case of Lagrange interpolation, one of a class of methods termed *exact*. This terminology comes from the fact that Lagrange polynomials are constructed so that they exactly reproduce the function values at the abscissas of the given data points. For example, recall that the linear interpolation formula is constructed so that it exactly reproduces both endpoint function values on the interval of interpolation because it is simply a straight line through these two points. Hence, it is a first degree exact interpolation polynomial, and we will see below that it is a Lagrange polynomial.

We begin this section with an heuristic approach to exact interpolation that can sometimes be useful in situations more general than the polynomial case to be treated, and we follow this with a theorem (restricted to the polynomial case)on uniqueness of such results. We then present construction of the Lagrange interpolation polynomials, analyze their accuracy and extend the constuction procedure to two independent variables. Finally, we briefly remark on some weaknesses of this form of interpolation.

A Basic Exact Interpolation Construction Method

It is probably of some value to build an exact interpolation polynomial in an intuitive (although computationally inefficient) manner first in order to demonstrate some of the basic ideas. Suppose we are given three points (x_1, f_1), (x_2, f_2) and (x_3, f_3), and we are required to construct an exact interpolation polynomial $p_n(x)$ of degree n which takes on the values f_1, f_2, f_3 at x_1, x_2, x_3, respectively. Our first task is to determine the appropriate degree, n, of the polynomial with which we intend to represent these data. In general, a polynomial of degree n is of the form

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n. \tag{3.7}$$

Thus, it contains $n+1$ coefficients to be determined, and this implies that $n+1$ points (x_i, f_i) will be required. Conversely, if $n+1$ points are given, and it is required that they be interpolated with an exact polynomial, the polynomial must in general be of degree at least n. In our particular case we are given three points, so we must employ a polynomial of degree at least two, *i.e.*, a quadratic.

The next step is to formally evaluate Eq. (3.7) at each of the data points. We obtain

$$\begin{aligned} p_2(x_1) = f_1 &= a_0 + a_1 x_1 + a_2 x_1^2 \\ p_2(x_2) = f_2 &= a_0 + a_1 x_2 + a_2 x_2^2 \\ p_2(x_3) = f_3 &= a_0 + a_1 x_3 + a_2 x_3^2 \end{aligned} \tag{3.8}$$

with the left-hand equalities arising due to the requirement of exactness. Now since the f_is and x_is are given, we see that (3.8) is merely a linear system of three equations for the three a_is. It can be demonstrated that the system matrix (sometimes called the Vandemonde matrix) is nonsingular if and only if $x_i \neq x_j$ when $i \neq j$; *i.e.*, the data are not multivalued. Thus, the solution exists and is unique, and this implies the following:

Theorem 3.1 *Let $\{x_i\}_{i=1}^{n+1}$ be a set of distinct points on $\mathbb{R}^1$, and let $f_i = f(x_i)$. Then $\exists$ a unique polynomial p_n: $\mathbb{R}^1 \to \mathbb{R}^1$ of degree n,*

$$p_n(x) = \sum_{i=0}^{n} a_i x^i \, ,$$

such that

$$p_n(x_i) = f_i \quad \forall \; i = 1, 2, \ldots, n+1 \; . \tag{3.9}$$

Several remarks are of value at this time. The first is that polynomials of higher degree can be constructed through the same set of points merely by deleting one or more, as required, lower-degree terms and replacing these with higher-degree ones. Second, although the system matrix correponding to (3.8) can be shown to be nonsingular, it is often ill-conditioned. So coefficients obtained as solutions to (3.8) may be subject to large rounding errors. Finally, although we will restrict our attention to exact polynomial fits in these lectures, the procedure just described can be used with other functional forms, *e.g.*, rational functions, exponentials, *etc.*

Construction of Lagrange interpolation polynomials

We will now present a construction method that is numerically stable, and more efficient than Gaussian elimination for the present application. To begin we assume that $p_n(x)$ can be expressed as

$$p_n(x) = \sum_{i=1}^{n+1} \ell_i(x) f_i \; , \tag{3.10}$$

where the f_is are the given ordinates. The problem now reduces to determining the ℓ_is. Clearly, at least one of these must be a polynomial of degree n (and none can have degree $> n$) if this is to be true of p_n, and in Lagrange interpolation this is true for all the ℓ_is. Moreover, if $p_n(x_i) = f_i$ is to hold, we should expect that

$$\ell_i(x_j) = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j; \end{cases} \tag{3.11}$$

or

$$\ell_i(x_j) = \delta_{ij} \; ,$$

where δ_{ij} is the *Kronecker* δ (which, itself, is defined exactly as (3.11)).

The above requirements lead us to the following prescription for the ℓ_is:

$$\ell_i(x) = \prod_{\substack{j=1 \\ j\neq i}}^{n+1} \frac{(x - x_j)}{(x_i - x_j)} \ , \quad i = 1, 2, \ldots, n+1. \tag{3.12}$$

It is easily checked that when $x = x_i$, each factor in the product is unity, so $\ell_i(x_i) = 1$. Similarly for $x = x_j$ for some $j \neq i$, one of the factors appearing in Eq. (3.12) will be zero, and we have $\ell_i(x_j) = 0$. Finally, if $x \neq x_i \ \forall \ i = 1, \ldots, n+1$, all of the factors are nonzero. There are n such factors, because the one corresponding to $j = i$ is deleted. Hence, each ℓ_i is of degree n.

The pseudo-language algorithm for Lagrange interpolation is easily constructed (simply evaluate Eq. (3.12) for any desired value of x, and substitute the results into Eq. (3.10)), so we leave this to the reader. However, the following remarks are of value. Each ℓ_i requires $2n-1$ multiplies (actually, n divisions and $n-1$ multiplies) for its construction. Then an additional $n+1$ multiplies and n adds are needed to evaluate Eq. (3.10). In any case the total arithmetic required to evaluate p_n at any desired point is only $\mathcal{O}(n^2)$ by this method. If we were to follow the intuitive idea of solving Eq. (3.8) for the polynomial coefficients a_i by Gaussian elimination, $\mathcal{O}(n^3)$ arithmetic operations would be required just to determine the a_is. Additional work would still be needed to evaluate $p_n(x)$. However, it should be mentioned that in many applications n is quite small; in these cases there may be situations in which there is no distinct advantage in employing (3.10) and (3.12) beyond the fact that they are general formulas that can be used for all $n < \infty$.

A specific example—linear interpolation

We now present a specific example of applying the preceding Lagrange interpolation formulas. To provide a familiar context, and at the same time avoid undue algebraic complications, we will construct the formula for linear interpolation between the two points (x_1, f_1) and (x_2, f_2). Since there are two points, the Lagrange polynomial must be of degree one; *i.e.*, it is a (generally affine) linear function.

From (3.10), we have for $n = 1$

$$p_1(x) = \sum_{i=1}^{2} \ell_i(x) f_i = \ell_1(x) f_1 + \ell_2(x) f_2 \ .$$

We next employ (3.12) to construct ℓ_1 and ℓ_2:

$$\ell_i(x) = \prod_{\substack{j=1 \\ j\neq i}}^{n+1} \frac{(x - x_j)}{(x_i - x_j)} \ \Rightarrow \ \ell_1(x) = \prod_{\substack{j=1 \\ j\neq 1}}^{2} \frac{(x - x_j)}{(x_1 - x_j)} = \frac{x - x_2}{x_1 - x_2} \ ,$$

and

$$\ell_2(x) = \prod_{\substack{j=1 \\ j\neq 2}}^{2} \frac{(x - x_j)}{(x_2 - x_j)} = \frac{x - x_1}{x_2 - x_1} \ .$$

Hence,

$$p_1(x) = \frac{x - x_2}{x_1 - x_2} f_1 + \frac{x - x_1}{x_2 - x_1} f_2 \ , \tag{3.13}$$

or, after slight rearrangement,

$$p_1(x) = f_1 + \frac{f_2 - f_1}{x_2 - x_1}(x - x_1) \ . \tag{3.14}$$

The form of (3.14) is equivalent to the standard one which would be obtained by writing the equation of the straight line between (x_1, f_1) and (x_2, f_2)—and also what would be found via Taylor expansion of f about $x = x_1$ followed by replacing the derivative with a first "divided difference," hence, a *Taylor polynomial*—thus suggesting uniqueness of at least the first-degree Lagrange interpolation polynomial.

Accuracy of Lagrange polynomial interpolation

Another thing that can be deduced from Eq. (3.14) is the accuracy achieved by linear interpolation. Suppose we were given only the point (x_1, f_1), but in addition were also given the first two derivatives of f evaluated at x_1, *i.e.*, $f'(x_1)$ and $f''(x_1)$. Then for any x close to x_1 we can expand f in a Taylor series about x_1:

$$f(x) = f(x_1) + f'(x_1)(x - x_1) + \frac{1}{2}f''(x_1)(x - x_1)^2 + \cdots \ . \tag{3.15}$$

We notice that this is the same as (3.14) through the first two terms, except that $f'(x_1)$ in (3.15) is replaced by the slope $(f_2 - f_1)/(x_2 - x_1)$ in (3.14). Now if we approximate $f'(x_1)$ by a forward difference (similar to what was used to develop the secant method in Chap. 2), we obtain

$$f'(x_1) \equiv f_1' = \frac{f_2 - f_1}{x_2 - x_1} - \frac{1}{2}f_1''(x_2 - x_1) + \cdots \ ,$$

as we will show in more detail later in this chapter. Substitution of this into Eq. (3.14) yields

$$f(x) = f_1 + \frac{f_2 - f_1}{x_2 - x_1}(x - x_1) + \frac{1}{2}f_1''\left[(x - x_1)^2 - (x_2 - x_1)(x - x_1)\right] + \cdots \ . \tag{3.16}$$

We see that $f(x)$ in (3.16), which is a precise (infinite Taylor expansion) representation, agrees with $p_1(x)$ up to the third term. This term is called the *dominant truncation error*, and since $x \le x_2$, the term is bounded by $C(x_2 - x_1)^2$, where C is a constant. That is,

$$f(x) - p_1(x) = C(x_2 - x_1)^2 \, .$$

From this we see that linear interpolation is *second-order accurate* in the step size $\Delta x = x_2 - x_1$. Thus, as Δx is reduced $p_1(x)$ more accurately approximates $f(x)\ \forall\ x \in [x_1, x_2]$, in agreement with our intuition. But beyond this we see that the error in this approximation decreases with the square of Δx.

Similar analyses show that, in general, a Lagrange interpolation polynomial of degree n provides an approximation of order $n + 1$. In particular, it is shown (for example, in Isaacson and Keller [15]) that if $f(x)$ is approximated by a polynomial of degree n, then

$$f(x) - p_n(x) = \frac{1}{(n+1)!}\left(\prod_{i=1}^{n+1}(x - x_i)\right) f^{(n+1)}(\xi) \ ,$$

where the x_i are the abscissas of the data points used to construct p_n, $f^{(n+1)}$ denotes the $(n+1)^{th}$ derivative of f, and

$$\min(x_1, \ldots, x_{n+1}, x) < \xi < \max(x_1, \ldots, x_{n+1}, x) \ .$$

It is clear that if we define

$$\Delta x \equiv \max_{1 \le i \le n} (x_{i+1} - x_i) \ ,$$

we have

$$f(x) - p_n(x) \le C \Delta x^{n+1} \sim \mathcal{O}(\Delta x^{n+1}) \, .$$

Thus, in general, a n^{th}-degree Lagrange polynomial provides a $(n+1)^{th}$-order approximation.

Linear interpolation in two dimensions

We conclude this section on Lagrange interpolation by demonstrating how to apply linear interpolation in two independent variables. Rather than present a theoretical development, we will employ an intuitive approach. In Fig. 3.2 we depict a grid of function values corresponding to a function $f(x, y)$: $\mathbb{R}^2 \to \mathbb{R}^1$. Suppose we are given the values of f at the four points (x_1, y_1), (x_2, y_1), (x_1, y_2) and (x_2, y_2), and we are required to find a linear approximation to f at the point (x^*, y^*). The

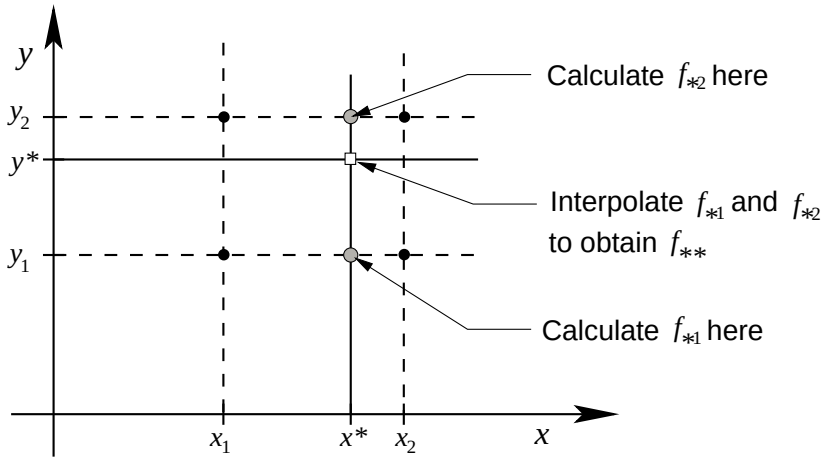


Figure 3.2: Linear interpolation of $f(x, y)$: $\mathbb{R}^2 \to \mathbb{R}^1$

intuitive approach, which turns out to work quite well in this case, is to first hold y fixed and interpolate in x, at each of the two fixed values of y namely, at y_1 and y_2. Once this has been done, we interpolate in the y direction with x fixed. It is not hard to check that the order of these operations can be reversed without affecting the result.

One can easily derive a single closed formula for multi-dimensional linear interpolation, but it is just as easy to recursively apply the formula for a single dimension, that is, Eq. (3.14). Based on this, we have the following algorithm.

Algorithm 3.1 *(Two-Dimensional Linear Interpolation)*
Assume the four triples of numbers (x_1, y_1, f_{11}), (x_2, y_1, f_{21}), (x_1, y_2, f_{12}) *and* (x_2, y_2, f_{22}) *are given.*

1. *For fixed* $y = y_1$ *interpolate between* x_1 *and* x_2*:*

$$f_{*1} = f_{11} + \frac{f_{21} - f_{11}}{x_2 - x_1}(x^* - x_1)$$

2. For fixed $y = y_2$ interpolate between x_1 and x_2:

$$f_{*2} = f_{12} + \frac{f_{22} - f_{12}}{x_2 - x_1}(x^* - x_1)$$

3. For fixed $x = x^*$ interpolate between y_1 and y_2:

$$f_{**} = f_{*1} + \frac{f_{*2} - f_{*1}}{y_2 - y_1}(y^* - y_1)$$

As already indicated, these expressions can be combined, analytically, to obtain the above mentioned *bilinear interpolation* formula:

$$f_{**} = f_{11} + \frac{f_{21} - f_{11}}{x_2 - x_1}(x^* - x_1) + \frac{f_{12} - f_{11}}{y_2 - y_1}(y^* - y_1) + \frac{f_{22} - f_{21} - f_{12} + f_{11}}{(x_2 - x_1)(y_2 - y_1)}(x^* - x_1)(y^* - y_1) \ .$$

We leave as an exercise to the reader showing that this expression provides a second-order approximation to $f(x, y)$ in the same sense already discussed in the one-dimensional case. (Hint: note the resemblance of the preceding expression to a multi-dimensional Taylor expansion of f.) Finally, we mention that even higher-dimensional linear approximations can be constructed in a manner analogous to that summarized in Algorithm 3.1, and furthermore, higher-degree interpolation polynomials might be used in place of the linear ones treated here.

Difficulties with high-degree interpolation polynomials

We conclude this section by observing that one of the major shortcomings of using Lagrange polynomials is that the high-degree polynomials required for formal accuracy often exhibit very poor behavior away from the abscissas, as is illustrated in Fig. 3.3. Over long intervals it is impossible

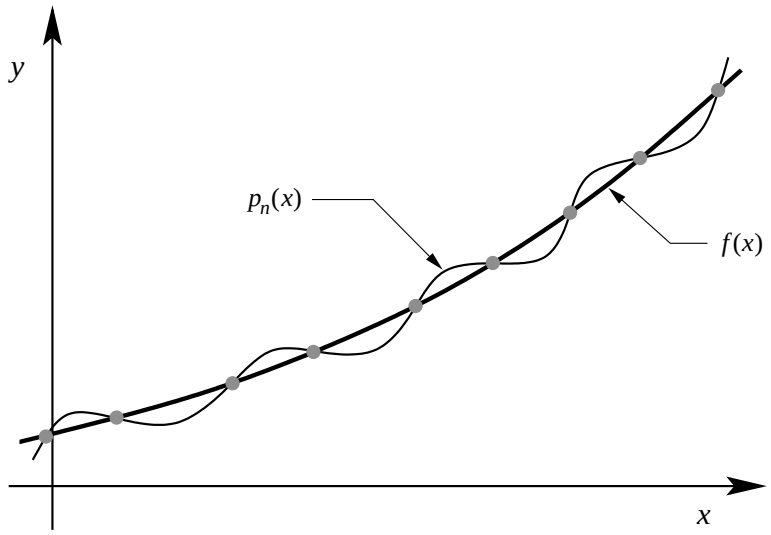


Figure 3.3: Ill-behavior of high-order Lagrange polynomials

to obtain reasonable accuracy with a single low-degree polynomial; but the "wiggles" that occur when high-degree global approximations are used may also be unacceptable. For example, it is clear from Fig. 3.3 that $p_n'(x)$ is not generally a good approximation to $f'(x)$—at some points even its sign is incorrect, and this often can be a severe disadvantage. In the next section we will present a partial remedy.

3.1.3 Cubic spline interpolation

We probably know from experience that linear interpolation usually works reasonably well. The reason for this it is that is usually applied over short subintervals of the domain of definition of the function being interpolated. This is often sufficient when only function values are needed. But if derivatives of the interpolated function are required, for example to be employed in Newton's method, local linear interpolation should not be used. This can be seen from Fig. 3.4. In particular, the derivative of the interpolation polynomial $p_1(x)$ for $x \in [x_1, x_2]$ is exactly the constant $(f_2 - f_1)/(x_2 - x_1)$. On the other hand for $x \in [x_2, x_3]$ the derivative is $(f_3 - f_2)/(x_3 - x_2)$. Hence, there is a discontinuity in p_1' at $x = x_2$.

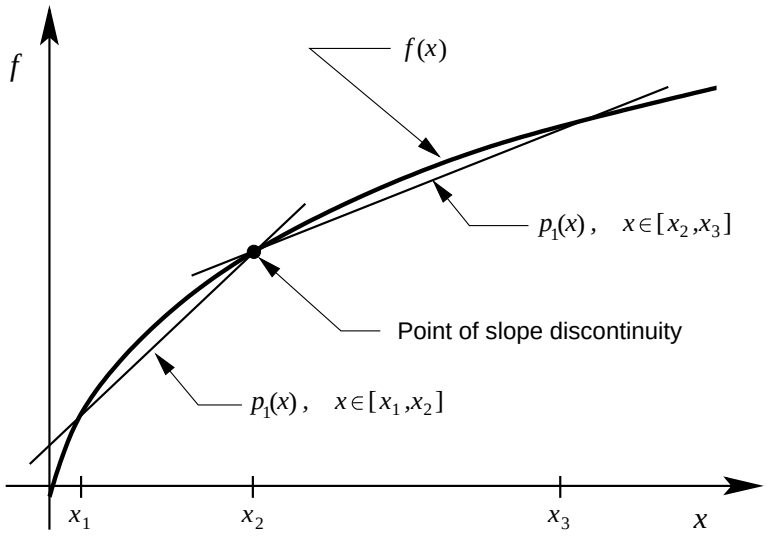


Figure 3.4: Discontinuity of 1^{st} derivative in local linear interpolation

An approximation procedure that provides at least a partial remedy both to the wiggles of high-degree polynomials and to the derivative discontinuities of local low-degree approximations is the cubic spline. The cubic spline is applied locally, although the procedure differs considerably from linear interpolation in the details of its implementation. In addition, it is constructed in such a way that the interpolation formula is globally twice continuously differentiable. This is sufficient smoothness for most applications.

Theoretical Construction

We begin with the formal definition of the *cubic spline*. Let P be a partition of $[a, b]$ such that $a = x_1 < x_2 < \ldots < x_n = b$.

Definition 3.1 *A cubic spline $S(x)$ on P is a function S: $[a, b] \to \mathbb{R}$ with the properties:*

i) $S \in C^2[a, b]$, and

ii) S coincides with an exact interpolation polynomial of degree three on every subinterval $[x_i, x_{i+1}]$, $i = 1, \ldots, n - 1$.

The x_is of the partition P are called the *knots* of the spline, and the values $S''(x_i)$, *i.e.*, the second derivatives of the spline evaluated at knots, are called *moments*.

The first step in deriving the equations for determining the coefficients for spline interpolation is to recognize that since the spline is a cubic on each subinterval, its second derivatives must be linear functions on each subinterval. Thus, if S_i'' and S_{i+1}'' are the moments at successive knots, then from Eq. (3.13) we see that

$$S''(x) = \frac{x - x_i}{h_{i+1}}\, S''_{i+1} - \frac{x - x_{i+1}}{h_{i+1}}\, S''_i\,,$$

where $h_{i+1} \equiv x_{i+1} - x_i$, and $x \in [x_i, x_{i+1}]$. Observe that by definition S'' is required to be continuous $\forall\ x \in [a, b]$. Note also that we do not assume the h_is to be equal. Next, we integrate the above equation twice: the first integration yields

$$S'(x) = \frac{(x - x_i)^2}{2h_{i+1}}\, S''_{i+1} - \frac{(x - x_{i+1})^2}{2h_{i+1}}\, S''_i + A_i\,, \tag{3.17}$$

and a second integration leads to

$$S(x) = \frac{(x - x_i)^3}{6h_{i+1}}\, S''_{i+1} - \frac{(x - x_{i+1})^3}{6h_{i+1}}\, S''_i + A_i(x - x_i) + B_i\,. \tag{3.18}$$

In order to evaluate the integration constants A_i and B_i, $i = 1, \ldots, n$, we make use of the exactness of S on each of the subintervals. In particular, when $x = x_i$ we have from (3.18),

$$\begin{aligned} S(x_i) &= -\frac{(x_i - x_{i+1})^3}{6h_{i+1}}\, S''_i + B_i = f_i \\ &= \frac{1}{6}h_{i+1}^2 S''_i + B_i = f_i\ . \end{aligned}$$

Similarly, when $x = x_{i+1}$,

$$S(x_{i+1}) = \frac{1}{6}h_{i+1}^2 S''_{i+1} + h_{i+1}A_i + B_i = f_{i+1}\ .$$

Hence, the integration constants are

$$\begin{aligned} A_i &= \frac{f_{i+1} - f_i}{h_{i+1}} - \frac{h_{i+1}}{6}(S''_{i+1} - S''_i)\ , \\ B_i &= f_i - \frac{h_{i+1}^2}{6}(S''_i)\ . \end{aligned} \tag{3.19}$$

If we substitute these into (3.18) and use the fact that $(x - x_{i+1})^3 = (x - x_i - h_{i+1})^3$, after some algebra we arrive at a canonical expression for S that is analogous to $p_1(x)$ in (3.14):

$$\begin{aligned} S(x) = f_i + \left[\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{1}{6}\left(2S''_i + S''_{i+1}\right)h_{i+1}\right](x - x_i) \\ + \frac{1}{2}S''_i(x - x_i)^2 + \frac{1}{6}\frac{(S''_{i+1} - S''_i)}{h_{i+1}}(x - x_i)^3. \end{aligned} \tag{3.20}$$

It should be observed immediately that S is expressed in terms only of the original ordinates, the f_is, and the moments, the S_i'', and that S is continuous on $[a, b]$. Indeed, by the preceding construction, $S \in C^2[a, b]$, which can be directly verified using (3.20). Moreover, it is clear that

(3.20) is in the form of a truncated Taylor series for $f(x)$, expanded about x_i. From earlier results we would expect

$$f(x) - S(x) = \mathcal{O}(h^4)$$

to hold, since S is locally a cubic. It is clear from (3.20) that in order for this to be true, the following must be satisfied:

$$f_i' - \left[\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{1}{6}\left(2S_i'' + S_{i+1}''\right) h_{i+1}\right] = \mathcal{O}(h^3) \ , \tag{3.21}$$

$$f_i'' - S_i'' = \mathcal{O}(h^2) \ , \tag{3.22}$$

$$f_i''' - \left(\frac{S_{i+1}'' - S_i''}{h_{i+1}}\right) = \mathcal{O}(h) \ . \tag{3.23}$$

It is easily shown that (3.22) implies (3.21) and (3.23); we leave this as an excercise for the reader. However, (3.22) is not automatic. The reader is referred to deBoor [5] for details of this, and in general, for extensive treatment of cubic splines.

Computational Algorithm

We now turn to development of a computational scheme for determining the S_i''. The main piece of information which has not yet been used is continuity of $S'(x)$. From (3.17) and (3.19) we have at the i^{th} knot,

$$S'(x) = \frac{(x - x_i)^2}{2h_{i+1}} \ S_{i+1}'' - \frac{(x - x_{i+1})^2}{2h_{i+1}} \ S_i'' - \frac{h_{i+1}}{6}(S_{i+1}'' - S_i'') + \frac{f_{i+1} - f_i}{h_{i+1}} \ . \tag{3.24}$$

We can also write this for $i - 1$:

$$S'(x) = \frac{(x - x_{i-1})^2}{2h_i} \ S_i'' - \frac{(x - x_i)^2}{2h_i} \ S_{i-1}'' - \frac{h_i}{6}(S_i'' - S_{i-1}'') + \frac{f_i - f_{i-1}}{h_i} \ . \tag{3.25}$$

At $x = x_i$, these two expressions must have the same value, by continuity. Thus, after some manipulation, we obtain

$$\frac{h_i}{6}S_{i-1}'' + \frac{1}{3}(h_{i+1} + h_i)S_i'' + \frac{h_{i+1}}{6} \ S_{i+1}'' = \frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i} \ , \tag{3.26}$$

which holds for $i = 2, \ldots, n-1$. (It does not hold at the endpoints since only one formula for S is available at these points.) When $i = 1$ and $x = x_1$, we have from (3.24)

$$\begin{aligned} S'(x_1) &= -\frac{(x_1 - x_2)^2}{2h_2} \ S_1'' - \frac{h_2}{6} \ (S_2'' - S_1'') + \frac{f_2 - f_1}{h_2} \\ &= -\frac{h_2}{3} \ S_1'' - \frac{h_2}{6} \ S_2'' + \frac{f_2 - f_1}{h_2} \ . \end{aligned} \tag{3.27}$$

Similarly, for $i = n$ and $x = x_n$, using (3.25) yields

$$\begin{aligned} S'(x_n) &= -\frac{(x_n - x_{n-1})^2}{2h_n} \ S_n'' - \frac{h_n}{6} \ (S_n'' - S_{n-1}'') + \frac{f_n - f_{n-1}}{h_n} \\ &= \frac{h_n}{6} \ S_{n-1}'' + \frac{h_n}{3} \ S_n'' + \frac{f_n - f_{n-1}}{h_n} \ . \end{aligned} \tag{3.28}$$

If (3.27) and (3.28) are used to complete the system (3.26), then endpoint values of S' must be prescribed. In some cases, they may actually be given as part of the data, but it is more usual to approximate them from the f_is near the endpoints. Recall from our error analysis that we want to obtain S'' as a second-order approximation to f'' in order to maintain fourth-order accuracy for f. We also found that S' must be at least third-order accurate. In a later section we will treat such derivative approximations. For the present, we will merely denote these required derivatives by f'_1 and f'_n, respectively, and assume they are of the required accuracy. Then the system of equations to be solved for the moments of S takes the form

$$2S''_1 + S''_2 = \frac{6}{h_2}\left(\frac{f_2 - f_1}{h_2} - f'_1\right) , \quad \text{for } i = 1 ;$$

$$\frac{h_i}{h_i + h_{i+1}}\, S''_{i-1} + 2S_i + \frac{h_{i+1}}{h_i + h_{i+1}}\, S''_{i+1} = \frac{6}{h_i + h_{i+1}}\, \left(\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}\right)$$

for $i = 2, \ldots, n-1$; and

$$S''_{n-1} + 2S''_n = \frac{6}{h_n}\left(f'_n - \frac{f_n - f_{n-1}}{h_n}\right) , \quad \text{for } i = n.$$

If we define

$$\mu_i \equiv \frac{h_i}{h_i + h_{i+1}} , \quad \lambda_i \equiv \frac{h_{i+1}}{h_i + h_{i+1}} ,$$

and

$$b_i = \begin{cases} \dfrac{6}{h_2}\left(\dfrac{f_2 - f_1}{h_2} - f'_1\right) & \text{for } i = 1 , \\[2ex] \dfrac{6}{h_i + h_{i+1}}\, \left(\dfrac{f_{i+1} - f_i}{h_{i+1}} - \dfrac{f_i - f_{i-1}}{h_i}\right) & \text{for } 1 < i < n , \\[2ex] \dfrac{6}{h_n}\left(f'_n - \dfrac{f_n - f_{n-1}}{h_n}\right) & \text{for } i = n , \end{cases}$$

then we see that the system takes the form

$$\begin{bmatrix} 2 & \lambda_1 & & & & \\ \mu_2 & 2 & \mu_2 & & \mathbf{0} & \\ & \mu_3 & 2 & \mu_3 & & \\ & & \ddots & \ddots & \ddots & \\ & \mathbf{0} & & \mu_{n-1} & 2 & \lambda_{n-1} \\ & & & & \mu_n & 2 \end{bmatrix} \begin{bmatrix} S''_1 \\ S''_2 \\ \cdot \\ \cdot \\ \cdot \\ S''_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ \cdot \\ \cdot \\ b_n \end{bmatrix} . \tag{3.29}$$

This is a tridiagonal system that can be readily solved using the tridiagonal LU decomposition discussed in Chapter 1.

We point out that there are two other commonly used sets of conditions for determining the moments of S. The first is simply $S''_1 = S''_n = 0$. This condition yields what is often called the *natural cubic spline*. Observe that this implies that there must be no curvature in $f(x)$ near the endpoints of the interval $[a, b]$. It is a widely used alternative, but such wide application is due mainly to its convenience. It cannot be generally recommended. When it is employed the system (3.29) contains two fewer equations; namely equations (3.27) and (3.28) are deleted. The tridiagonal character of the system is not altered because for $i = 2$, the first term in (3.26) is zero

and this is also true for the last term when $i = n - 1$. The final condition sometimes employed to augment (3.26) is a periodicity condition. It is not widely used, and we will not discuss it further here. Details can be found in DeBoor [5], a basic practical reference for splines, and in Stoer and Bulirsch [30].

We shall close this section with a complete algorithm for construction and evaluation of cubic splines.

Algorithm 3.2 *(Cubic Spline with 1^{st} Derivative Specified at Endpoints)*

A. Spline Construction

1. *Do* $i = 1,\ n-1$
   $h_{i+1} = x_{i+1} - x_i$
2. *Do* $i = 1,\ n$
   $a_{2,i} = 2$
   If $i = 1$, *then* $a_{1,i} = 0$;
   $$a_{3,i} = 1 \ ;$$
   $$b_i = \frac{6}{h_2}\left(\frac{f_2 - f_1}{h_2} - f_1'\right)$$
   If $1 < i < n$, *then* $a_{1,i} = \mu_i = \dfrac{h_i}{h_i + h_{i+1}}$
   $$a_{3,i} = \lambda_i = \frac{h_{i+1}}{h_i + h_{i+1}}$$
   $$b_i = \frac{6}{h_i + h_{i+1}}\left(\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}\right)$$
   If $i = n$ *then* $a_{1,i} = 1$
   $$a_{3,i} = 0$$
   $$b_i = \frac{6}{h_n}\left(f_n' - \frac{f_n - f_{n-1}}{h_n}\right)$$
3. *Call tridiagonal LU decomposition to calculate* $S_i'',\ i = 1, \ldots, n$.
4. *Calculate local cubic polynomial coefficients*
   Do $i = 1,\ n-1$
   $$C_{1,i} = f_i$$
   $$C_{2,i} = \frac{f_{i+1} - f_i}{h_{i+1}} - \frac{1}{6}(2S_i'' + S_{i+1}'')h_{i+1}$$
   $$C_{3,i} = \frac{1}{2}S_i''$$
   $$C_{4,i} = \frac{1}{6}\frac{S_{i+1}'' - S_i''}{h_{i+1}}$$

B. Evaluate Spline at Desired points.

1. *Input* x *value*
2. *Do* $i = 1,\ n-1$
   If $x_i > x$ *or* $x_{i+1} < x$, *then repeat* i
   else $f = 0$
   Do $j = 1,\ 4$

$$f = f + C_{j,i}(x - x_i)^{j-1}$$

3. Return

3.1.4 Extraplotation

It is sometimes necessary to employ interpolation polynomials (and other function approximations) outside the range of data used for their construction. This is termed *extrapolation*. We emphasize that application of this is, in general, a dangerous practice; but if done with care, it can at times be very useful. Precisely the same formulas we have already derived for interpolation are used, but they now are evaluated at points $x < x_1$ and/or $x > x_{n+1}$. Formally, the same form of error estimates applies, but it must be recognized that actual accuracy is difficult to assess for an approximation of this kind.

3.2 Numerical Quadrature

The term *numerical quadrature* refers to the numerical evaluation of definite integrals, for example, of the form

$$\int_a^b f(x)\ dx\ .$$

In general, either or both of a and b may be transfinite. However in our treatment here we will be concerned only with intervals $[a, b]$ that are of finite length.

The underlying idea in all numerical quadrature is to replace the integrand, $f(x)$, for which we presumably cannot find a *primitive* (antiderivative), with an approximation that can be easily integrated. One of the main approaches to this goal is to approximate f with Lagrange polynomials, p_n, developed in the preceding section, and then integrate p_n. This leads to the *Newton–Cotes quadrature* formulas, of which trapezoidal and Simpson's rules are familiar examples. Another possibility is to approximate f with Legendre polynomials (which we have not discussed, *see, e.g.* [30]), and obtain the *Gauss–Legendre quadrature* methods. In any case *quadrature formulas* can always be written in the form

$$\int_a^b f(x)\ dx \cong h \sum_{i=1}^{n_w} w_i f_i\ ,$$

where the f_i are given values of $f(x)$ as appear, for example, in Lagrange interpolation; h is the *step size* (usually—but not always—the distance between successive abscissas where f is evaluated), and the w_i are the n_w *quadrature weights*. It is important to note that the w_i vary from one method to another, but that they do not depend on the function f being integrated. This makes numerical evaluation of definite integrals extremely simple and efficient, and even very accurate methods can be implemented on hand-held calculators.

We will here consider three different methods: *i*) trapezoidal rule, *ii*) Simpson's rule, and *iii*) Gauss–Legendre quadrature. In addition, we will discuss some methods for improving the accuracy of trapezoidal quadrature, and for applying any given method to evaluation of multiple integrals.

3.2.1 Basic Newton–Cotes quadrature formulas

In this subsection we will consider the two most fundamental Newton–Cotes formulas, the trapezoidal rule, and Simpson's rule. We will also discuss two modifications of the former that significantly improve its accuracy.

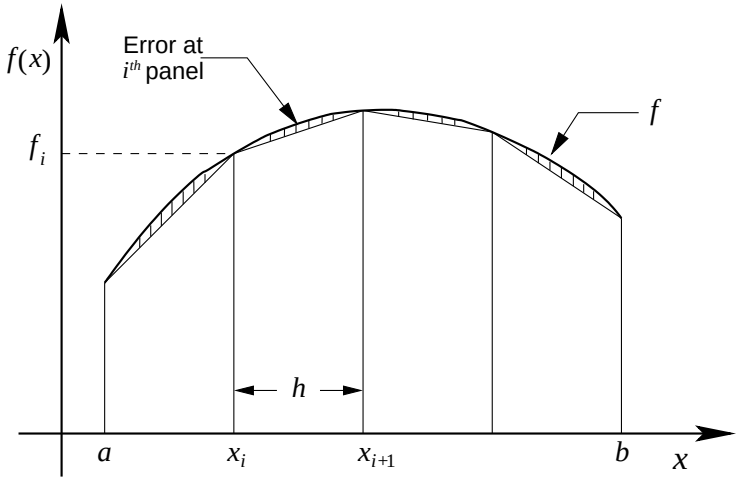


Figure 3.5: Geometry of Trapezoidal Rule

The Trapezoidal Formula

The simplest, nontrivial numerical quadrature scheme is the *trapezoidal rule*. It is worthwhile to begin development of this method via the geometric interpretation of the integral. We recall that the numerical value of a definite integral over an interval $[a, b]$ is merely the area under the graph of the integrand over this interval. In applying the trapezoidal rule, we replace the integrand with a piecewise linear function, as shown in Fig. 3.5. Clearly, the area of the i^{th} trapezoidal panel is

$$\frac{1}{2}\,(f_i + f_{i+1})\,h\ ,$$

where h is the (uniform) step size used to partition $[a, b]$; *i.e.*, $h = (b - a)/(n - 1)$ for a *partition* with n points, $a = x_1 < x_2 < \cdots < x_n = b$. We then sum these contributions over all the panels, which will always be one less than the number of points in the partition. Thus,

$$\int_a^b f(x)\ dx \simeq \frac{h}{2}\sum_{i=1}^{n-1}(f_i + f_{i+1})\ .$$

Now observe that, except for f_1 and f_n, each value of f appears twice. Hence, the above can be written as

$$\int_a^b f(x)\ dx \simeq h\left[\frac{1}{2}(f_1 + f_n) + \sum_{i=2}^{n-1} f_i\right]\ .$$

We see for this *composite* (covering more than one panel) trapezoidal formula that the weights are given by

$$w_i = \begin{cases} \frac{1}{2} & \text{if } i = 1 \text{ or } i = n\,, \\ 1 & \text{otherwise}\,. \end{cases}$$

It is of interest to note that these weights have the favorable properties, with respect to rounding errors, of being all of the same sign and being of approximately the same magnitude. Many quadrature formulas do not possess these attributes.

We now present an heuristic argument to show that the trapezoidal rule is second-order accurate; that is, the error of this approximation is $\mathcal{O}(h^2)$. Recall that we observed at the beginning that quadrature schemes are generally constructed by integrating polynomial approximations to the integrand. The above development depends on this, but has been carried out in a very intuitive manner. Here we will make this slightly more formal, but not completely rigorous. For a complete, rigorous treatment the reader is referred to the standard text on numerical quadrature, Davis and Rabinowitz [4].

From Fig. 3.5 it is clear that we have replaced $f(x)$ with a piecewise linear function; so for $x \in [x_i, x_{i+1}] \subset [a, b]$, we have

$$p_1^i(x) = f(x) + \mathcal{O}\left((x - x_i)^2\right) ,$$

from results in Section 3.1.2. Now integrate f over the i^{th} panel to obtain

$$\int_{x_i}^{x_{i+1}} f(x)\ dx = \int_{x_i}^{x_{i+1}} p_1^i(x) dx + \mathcal{O}\left((x_{i+1} - x_i)^3\right) ,$$

and note that $\mathcal{O}\left((x_{i+1} - x_i)^3\right) \sim \mathcal{O}(h^3)$. By a basic theorem from integral calculus we have for the integral over the entire interval $[a, b]$,

$$\int_a^b f(x)\ dx = \sum_{i=1}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\ dx = \sum_{i=1}^{n-1} \int_{x_i}^{x_{i+1}} p_1^i(x)\ dx + \mathcal{O}\left(\sum_{i=1}^{n-1} h^3\right) .$$

The first term on the far right is simply the trapezoidal approximation, while for the second term we have

$$\sum_{i=1}^{n-1} h^3 = (n-1)h^3 .$$

But by definition, $h = (b - a)/(n - 1)$; hence

$$\sum_{i=1}^{n-1} h^3 = (b-a)h^2 .$$

It follows that

$$\int_a^b f(x)\ dx = h\left[\frac{1}{2}(f_1 + f_n) + \sum_{i=2}^{n-1} f_i\right] + \mathcal{O}(h^2) . \tag{3.30}$$

For the interested reader, we note that it can be shown (*cf.* [4], or Henrici [13]) that the dominant truncation error for the trapezoidal method, here denoted simply by $\mathcal{O}(h^2)$, is actually of the form $\frac{-h^2}{12}(b - a)f''(\xi)$, $\xi \in [a, b]$.

Modifications to Trapezoidal Quadrature

We will now consider two simple modifications to the trapezoidal rule that can be used to significantly improve its accuracy. These are: *i*) use of end corrections, and *ii*) extrapolation. The first of these requires very precise knowledge of the dominant truncation error, while the second requires only that the order of the truncation error be known. Thus, extrapolation is preferred in general, and can be applied to a wide range of numerical procedures, as we will discuss in more detail in a later section.

For trapezoidal quadrature it turns out that the exact truncation error on any given subinterval $[x_i, x_{i+1}] \subset [a, b]$ can be found from a completely alternative derivation. If instead of employing Lagrange polynomials to approximate f we use Hermite polynomials (which, again, we have not discussed), we obtain a fourth-order approximation of f, and thus a locally fifth-order approximation to $\int f$. (For details see [30].) In particular, we have

$$\int_{x_i}^{x_{i+1}} f(x)\ dx = \frac{h}{2}(f_i + f_{i+1}) - \frac{h^3}{12}(f'_{i+1} - f'_i) - \frac{h^5}{720} f^{(4)}(\xi) + \cdots , \tag{3.31}$$

provided $f \in C^4(x_i, x_{i+1})$. Now observe that the first term on the right is exactly the original local trapezoidal formula, while the second term is an approximation to $\frac{-h^4}{12} f''(\xi)$, $\xi \in [x_i, x_{i+1}]$. The important thing to observe regarding (3.31) is that when we sum the contributions from successive subintervals, all but the first and last values of f' in the second term cancel, and we are left with

$$\int_a^b f(x)\ dx = h\left[\frac{1}{2}(f_1 + f_n) + \sum_{i=2}^{n-1} f_i - \frac{h^2}{12}(f'_n - f'_1)\right] + \mathcal{O}(h^4)\,. \tag{3.32}$$

This is called the *trapezoidal rule with end correction* because the additional terms contain information only from the ends of the interval of integration. If f'_1 and f'_n are known exactly, or can be approximated to at least third-order in h, then (3.32) is a fourth-order accurate method.

The second modification to the trapezoidal rule involves use of extrapolation to cancel leading terms in the truncation error expansion. A procedure by means of which this can be accomplished is known as *Richardson extrapolation*, and this can be used in any situation in which *i*) the domain on which the approximations are being done is discretized, and *ii*) an asymptotic expansion of the error in powers of the discretization step size is known. In contrast to endpoint correction, the error terms need not be known exactly for application of Richardson extrapolation. Only the power of the discretization step size in each term is required.

For the trapezoidal rule, it can be shown (see [30]) that

$$T(h) = \mathcal{I} + \tau_1 h^2 + \tau_2 h^4 + \cdots + \tau_m h^{2m} + \cdots , \tag{3.33}$$

where $\mathcal{I}$ is the exact value of the integral,

$$\mathcal{I} = \int_a^b f(x)\ dx\,,$$

and

$$T(h) = h\left[\frac{1}{2}(f_1 + f_n) + \sum_{i=2}^{n-1} f_i\right] ,$$

the trapezoidal quadrature formula.

The basic idea in applying Richardson extrapolation is to approximate the same quantity, $\mathcal{I}$ in this case, using two different step sizes, and form a linear combination of these results to eliminate the dominant term in the truncation error expansion. This procedure can be repeated to successively eliminate higher and higher order errors. It is standard to use successive halvings of the step size, *i.e.*, $h,\ h/2,\ h/4,\ h/8, \ldots$, *etc.*, mainly because this results in the most straightforward implementations. However, it is possible to use different step size ratios at each new evaluation. We will demonstrate the procedure here only for step halving, and treat the general case later.

We begin by evaluating (3.33) with h replaced by $h/2$:

$$T\left(\frac{h}{2}\right) = \mathcal{I} + \tau_1 \frac{h^2}{4} + \tau_2 \frac{h^4}{16} + \cdots . \tag{3.34}$$

Now observe that the dominant error term in (3.34) is exactly 1/4 that in (3.33) since both expansions contain the same coefficients, τ_i. Thus, without having to know the τ_i we can eliminate this dominant error by multiplying (3.34) by four, and substracting (3.33) to obtain

$$4T\left(\frac{h}{2}\right) - T(h) = 3\mathcal{I} - \frac{3}{4}\tau_2 h^4 + \mathcal{O}(h^6)\,.$$

Then division by three leads to the new estimate of $\mathcal{I}$ which is accurate to fourth-order:

$$T^*(h) \equiv \frac{4T(\frac{h}{2}) - T(h)}{3} = \mathcal{I} - \frac{1}{4}\tau_2 h^4 + \mathcal{O}(h^6)\,. \tag{3.35}$$

An important point to note here is that not only has the original dominant truncation error been removed completely, but in addition the new dominant term has a coefficient only 1/4 the size of the corresponding term in the original expansion.

When this procedure is applied recursively to the trapezoidal rule, two orders of accuracy are gained with each application. This occurs because only even powers of h occur in the error expansion, as can be seen from (3.33). This technique can be implemented as an automatic highly-efficient procedure for approximating definite integrals known as *Romberg integration*. Details are given in [30], and elsewhere.

Simpson's Rule Quadrature

We now briefly treat Simpson's rule. There are several ways to derive this fourth-order quadrature method. The basic theoretical approach is to replace the integrand of the required integral with a Lagrange cubic polynomial, and integrate. In Hornbeck [14] Simpson's rule is obtained by a Taylor expansion of an associated indefinite integral. Here, we will use Richardson extrapolation applied to the trapezoidal formula. To do this we must employ the global form of the trapezoidal rule, Eq. (3.30), because we wish to exploit a useful property of the truncation error expansion. As we have already seen, the global trapezoidal rule has an even-power error expansion while the local formula contains all (integer) powers of h greater than the second. Thus, recalling Eq. (3.30), we have

$$\int_a^b f(x)\; dx = h\left[\frac{1}{2}(f_1 + f_n) + \sum_{i=2}^{n-1} f_i\right] + \mathcal{O}(h^2)\,. \tag{3.36}$$

Now we double the value of h and observe (see Fig. 3.6) that on the resulting new partition of $[a, b]$ only the odd-indexed points of the original partition still occur. In particular, the summation (3.36) must now run over only odd integers from $i = 3$ to $n - 2$. This implies that $n - 2$ must be odd, and hence n is odd. We then have

$$\int_a^b f\; dx = 2h\left[\frac{1}{2}(f_1 + f_n) + \sum_{\substack{i=3 \\ i,\,odd}}^{n-2} f_i\right] + \mathcal{O}(h^2)\,. \tag{3.37}$$

We now apply Richardson extrapolation by multiplying Eq. (3.36) by four, substracting Eq. (3.37) and dividing by three:

$$\int_a^b f\; dx = \frac{h}{3}\left\{4\left[\frac{1}{2}(f_1 + f_n) + \sum_{i=2}^{n-1} f_i\right] - 2\left[\frac{1}{2}(f_1 + f_n) + \sum_{\substack{i=3 \\ i,\,odd}}^{n-2} f_i\right]\right\} + \mathcal{O}(h^4)\,.$$

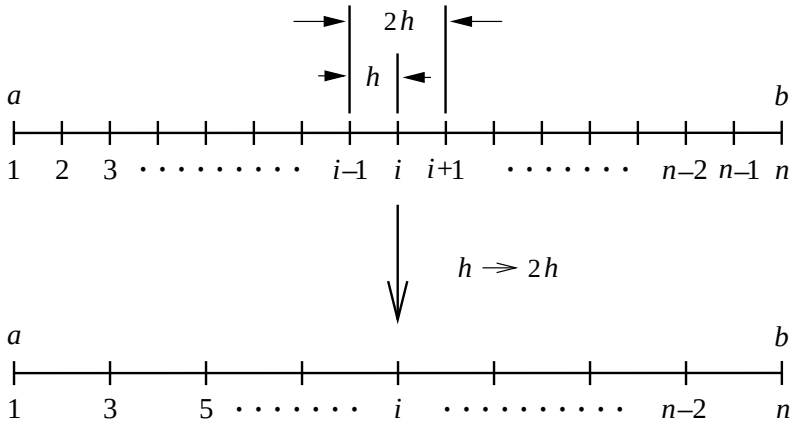


Figure 3.6: Grid-point Indexing on h and $2h$ Grids

We can rearrange this expression to obtain a more convenient form by observing that the first sum contains both even- and odd-indexed terms. Thus,

$$\int_a^b f\ dx = \frac{h}{3}\left[f_1 + f_n + 2\sum_{\substack{i=3\\ i,\,odd}}^{n-2} f_i + 4\sum_{\substack{i=2\\ i,\,even}}^{n-1} f_i\right] + \mathcal{O}(h^4)\,.$$

Finally, it is common for purposes of computer implementation to re-index and write this as

$$\int_a^b f(x)\ dx = \frac{h}{3}\left[f_1 + f_n + 2\sum_{i=2}^{\frac{n-1}{2}} f_{2i-1} + 4\sum_{i=1}^{\frac{n-1}{2}} f_{2i}\right] + \mathcal{O}(h^4)\,. \tag{3.38}$$

This is the form of Simpson's rule from which very efficient algorithms can be constructed. We see from (3.38) that the weights for composite Simpson's rule are as follows:

$$w_i = \begin{cases} \frac{1}{3} & \text{for } i = 1 \text{ or } i = n \\ \frac{2}{3} & \text{for } 3 \le i \le n-2,\ i \text{ odd} \\ \frac{4}{3} & \text{for } 2 \le i \le n-1,\ i \text{ even}\,. \end{cases}$$

Also observe that (3.38), as well as the formula that precedes it, reduces to the familiar local Simpson's rule when $n = 3$.

3.2.2 Gauss–Legendre quadrature

The two Newton–Cotes methods considered in the preceding section require that function values be known at equally spaced points, including the endpoints of integration. It turns out that higher-order methods can be constructed using the same number of function evaluations if the abscissas are not equally spaced. The Gauss–Legendre quadrature scheme to be considered now is a case of this. In particular, a Gauss–Legendre formula employing only n abscissas has essentially the same accuracy as a Newton–Cotes formula using $2n-1$ points. Thus, only about half as many integrand evaluations are required by Gauss–Legendre to achieve accuracy equivalent to a Newton–Cotes quadrature method. However, this sort of comparison is not very precise and should be viewed as

providing only a rough guideline. The more precise statement is the following: A Gauss–Legendre method using n absissas will exactly integrate a polynomial of degree $\leq 2n-1$. By contrast a local Newton–Cotes formula requiring n points will exactly integrate a polynomial of degree $\leq n-1$.

The Gauss–Legendre formulas are always local in the sense that the interval of integration is always $[-1,1]$. This is because the Legendre polynomials from which the methods are derived are defined only on $[-1,1]$. This, however, is not really a serious limitation because any interval $[a,b] \subseteq \mathbb{R}^1$ can be mapped to $[-1,1]$. We will here consider only the case of finite $[a,b]$. If we map a to -1 and b to 1 by a linear mapping we have

$$\frac{y-(-1)}{x-a} = \frac{1-(-1)}{b-a} = \frac{2}{b-a}\,,$$

where $x \in [a,b]$ and $y \in [-1,1]$. Thus, given $y \in [-1,1]$, we can find $x \in [a,b]$ from

$$x = a + \frac{1}{2}(b-a)(y+1)\,. \tag{3.39}$$

Now, suppose we wish to evaluate

$$\int_a^b f(x)\ dx$$

using Gauss–Legendre quadrature. From (3.39) it follows that

$$dx = \frac{1}{2}(b-a)\ dy\ ,$$

and the integral transforms as

$$\int_a^b f(x)\ dx = \frac{1}{2}(b-a)\int_{-1}^{1} f\left(a + \frac{1}{2}(b-a)(y+1)\right) dy\,, \tag{3.40}$$

which is now in a form to which Gauss–Legendre quadrature can be applied.

As we noted earlier, all quadrature formulas take the form

$$\int_a^b f(x)\ dx = h\sum_{i=1}^{n} w_i f_i\,.$$

For the Newton–Cotes formulas h was always the uniform partition step size; but for Gauss–Legendre there is no corresponding quantity. However, if we recall the form of the transformed interval given above, we see that

$$h = \frac{b-a}{2}\,.$$

As can be inferred from our discussion to this point the f_i do not correspond to evaluations of f at points of a uniform partition of $[-1,1]$. Instead the f_i are obtained as $f(y_i)$ where the y_i are the zeros of the Legendre polynomial of degree $n+1$. Tables of the y_i and w_i are to be found, for example in Davis and Rabinowitz [4]. We provide an abbreviated table below for $n = 1$, 2 and 3.

3.2.3 Evaluation of multiple integrals

We will conclude our treatment of basic quadrature methods with a brief discussion of numerical evaluation of multiple integrals. A standard reference is Stroud [33]. Any of the methods discussed above can be easily applied in this case; it should be emphasized, however, that the large number of

Table 3.1: Gauss–Legendre evaluation points y_i and corresponding weights w_i

n	y_i	w_i
1	±0.5773502692	1.0000000000
2	0.0000000000	0.8888888889
	±0.7745966692	0.5555555556
3	±0.3399810436	0.6521451547
	±0.8611363116	0.3478548451

function evaluations generally required of the Newton–Cotes formulas often makes them unsuitable when high accuracy is required for triple (or highrer-order) integrals, although such difficulties can now be mitigated via parallel processing. Here, we will treat only the case of double integrals, but the procedure employed is easily extended to integrals over domains of dimension higher than two.

Consider evaluation of the integral of $f(x, y)$ over the Cartesian product $[a, b] \times [c, d]$. It is not necessary to restrict our methods to the rectangular case, but this is simplest for purposes of demonstration. Moreover, nonrectangular domains can always be transformed to rectangular ones by a suitable change of coordinates. Thus, we evaluate

$$\int_a^b \int_c^d f(x, y)\ dydx\,.$$

If we define

$$g(x) \equiv \int_c^d f(x, y)\ dy\,, \tag{3.41}$$

we see that evaluation of the double integral reduces to evaluation of a sequence of single integrals. In particular, we have

$$\int_a^b \int_c^d f(x, y)\ dydx = \int_a^b g(x)\ dx\,;$$

so if we set

$$\int_a^b g(x)\ dx = h_x \sum_{i=1}^{m} w_i g_i\,, \tag{3.42}$$

then from (3.41) the g_is are given as

$$g_i \equiv g(x_i) = \int_c^d f(x_i, y)\ dy = h_y \sum_{j=1}^{n} w_j f_{ij}\,.$$

Hence, the formula for evaluation of double integrals is

$$\int_a^b \int_c^d f(x, y)\ dydx = h_x h_y \sum_{i,j=1}^{m,n} w_i w_j f_{ij}\,. \tag{3.43}$$

All that is necessary is to choose partitions of $[a, b]$ and $[c, d]$ to obtain h_x and h_y (unless Gauss–Legendre quadrature is used for one, or both, intervals), and then select a method—which determines the w_i and w_j. We note that it is not necessary to use the same method in each direction, although this is typically done. We also note that in the context of implementations on modern parallel processors, it is far more efficient to evaluate the m equations of the form (3.41) in parallel, and then evaluate (3.42) instead of using (3.43) directly.

3.3 Finite-Difference Approximations

Approximation of derivatives is one of the most important and widely-used techniques in numerical analysis, mainly because numerical methods represent the only general approach to the solution of differential equations—the topic to be treated in the final two chapters of these lectures. In this section we will present a formal discussion of difference approximations to differential operators. We begin with a basic approximation obtained from the definition of the derivative. We then demonstrate use of Taylor series to derive derivative approximations, and analyze their accuracy. Following this we will consider approximation of partial derivatives and derivatives of higher order. We then conclude the section with a few remarks and approximation methods that are somewhat different from, but still related to, the finite-difference approximations described here.

3.3.1 Basic concepts

We have already used some straightforward difference approximations in constructing the secant method and cubic spline interpolation. These basic approximations follow from the definition of the derivative, as given in Freshman calculus:

$$\lim_{h\to 0} \frac{f(x+h)-f(x)}{h} = f'(x)\,, \tag{3.44}$$

provided the limit exists. To obtain a *finite-difference approximation* to $f'(x)$ we simply delete the limit operation. The result is the first forward difference,

$$f'(x) \simeq \frac{f(x+h)-f(x)}{h}\,.$$

If we note that on a grid of points $x_{i+1} = x_i + h$, then in our usual notation we see that

$$\frac{f_{i+1}-f_i}{h} = \frac{f_{i+1}-f_i}{x_{i+1}-x_i} \tag{3.45}$$

is the *forward-difference* approximation to $f'(x_i)$.

It is crucial to investigate the accuracy of such an approximation. If we assume that $f \in C^2$ in a neighborhood of $x = x_i$, then for h sufficiently small we have the Taylor expansion

$$f_{i+1} = f_i + f_i' h + \frac{1}{2} f_i'' h^2 + \cdots\,.$$

Substitution into (3.45) yields

$$\begin{aligned}\frac{f_{i+1}-f_i}{h} &= \frac{1}{h}\left[\left(f_i + f_i' h + \frac{1}{2} f_i'' h^2 + \cdots\right) - f_i\right] \\ &= f_i' + \frac{1}{2} f_i'' h + \cdots\,.\end{aligned}$$

Hence, the leading error in (3.45) is $\frac{1}{2} f_i'' h$; so the approximation is first order in the step size h.

3.3.2 Use of Taylor series

There are many different ways to obtain derivative approximations, but probably the most common is by means of the Taylor series. We will demonstrate this now for a *backward-difference* approximation to the first derivative. We again assume $f \in C^2$, and write

$$f_{i-1} = f_i - f_i' h + \frac{1}{2} f_i'' h^2 - \cdots\,.$$

Then it follows immediately that

$$f_i' = \frac{f_i - f_{i-1}}{h} + \mathcal{O}(h)\,. \tag{3.46}$$

In order to obtain derivative approximations of higher-order accuracy, we can carry out Taylor expansions to higher order and form linear combinations so as to eliminate error terms at the desired order(s). For example, we have

$$f_{i+1} = f_i + f_i'h + \frac{1}{2}f_i''h^2 + \frac{1}{6}f_i'''h^3 + \frac{1}{24}f_i''''h^4 + \frac{1}{120}f_i'''''h^5 + \cdots\,,$$

and

$$f_{i-1} = f_i - f_i'h + \frac{1}{2}f_i''h^2 - \frac{1}{6}f_i'''h^3 + \frac{1}{24}f_i''''h^4 - \frac{1}{120}f_i'''''h^5 \pm \cdots\,.$$

If we substract the second from the first, we obtain

$$f_{i+1} - f_{i-1} = 2f_i'h + \frac{1}{3}f_i'''h^3 + \cdots\,,$$

and division by $2h$ leads to

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2h} - \frac{1}{6}f_i'''h^2 - \frac{1}{120}f_i'''''h^4 - \cdots\,. \tag{3.47}$$

This is the *centered-difference* approximation to $f'(x_i)$. As can be seen, it is second-order accurate. But it is also important to notice that its error expansion includes only even powers of h. Hence, its accuracy can be greatly improved with only a single Richardson extrapolation, just as was seen earlier for trapezoidal quadrature.

Use of Richardson extrapolation is a common way to obtain higher-order derivative approximations. To apply this to (3.47) we replace h by $2h$, f_{i+1} with f_{i+2}, and f_{i-1} with f_{i-2}. Then recalling the procedure used to derive Simpson's rule quadrature, we multiply (3.47) by four, substract the result corresponding to $2h$, and divide by three. Thus,

$$\begin{aligned} f_i' &= \frac{1}{3}\left[4\left(\frac{f_{i+1} - f_{i-1}}{2h}\right) - \left(\frac{f_{i+2} - f_{i-2}}{4h}\right)\right] + \mathcal{O}(h^4) \\ &= \frac{1}{12h}\left(f_{i-2} - 8f_{i-1} - 8f_{i+1} - f_{i+2}\right) + \mathcal{O}(h^4)\,. \end{aligned} \tag{3.48}$$

This is the fourth-order accurate centered approximation to the first derivative.

There is yet another way to employ a Taylor expansion of a function to obtain a higher-order difference approximation. This involves expressing derivatives in the leading truncation error terms as low-order difference approximations to eliminate them in favor of (known) grid function values. We demonstrate this by constructing a second-order forward approximation to the first derivative. Since we are deriving a forward approximation we expect to use the value of f at x_{i+1}. Thus, we begin with

$$f_{i+1} = f_i + f_i'h + \frac{1}{2}f_i''h^2 + \frac{1}{6}f_i'''h^3 + \cdots\,,$$

and rearrange this as

$$f_i' = \frac{f_{i+1} - f_i - \frac{1}{2}f_i''h^2}{h} - \frac{1}{6}f_i'''h^2 + \cdots\,. \tag{3.49}$$

We now observe that we can obtain an approximation of the desired order if we have merely a first-order approximation to f_i''. We have not yet discussed approximation of higher derivatives,

but as we will see later the only required idea is simply to mimic what we do analytically for exact derivatives; namely we repeatedly apply the difference approximation. Now recall that

$$f_i' = \frac{f_{i+1} - f_i}{h} + \mathcal{O}(h)\,;$$

so we expect (correctly) that

$$f_i'' = \frac{f_{i+1}' - f_i'}{h} + \mathcal{O}(h)\,.$$

It then follows that

$$\begin{aligned} f_i'' &= \frac{1}{h^2}\left(f_{i+2} - f_{i+1} - f_{i+1} + f_i\right) + \mathcal{O}(h) \\ &= \frac{1}{h^2}\left(f_{i+2} - 2f_{i+1} + f_i\right) + \mathcal{O}(h)\,. \end{aligned}$$

We now substitute this into (3.49):

$$f_i' = \frac{1}{h}\left[f_{i+1} - f_i - \frac{1}{2}\left(f_{i+2} - 2f_{i+1} + f_i\right)\right] + \mathcal{O}(h^2)\,,$$

or, after rearrangement,

$$f_i' = \frac{1}{2h}\left(-3f_i + 4f_{i+1} - f_{i+2}\right) + \mathcal{O}(h^2)\,. \tag{3.50}$$

This is the desired second-order forward approximation. A completely analogous treatment leads to the second-order backward approximation; this is left as an exercise for the reader.

3.3.3 Partial derivatives and derivatives of higher order

The next topics to be treated in this section are approximation of partial derivatives and approximation of higher-order derivatives. We will use this opportunity to introduce some formal notation that is particularly helpful in developing approximations to high-order derivatives and, as will be seen in the next two chapters, for providing concise formulas for discrete approximations to differential equations. The notation for difference approximations varies widely, and that used here is simply the preference of this author.

In general we will take $D(h)$ to be a difference operator based on step size h. (When no confusion is possible we will suppress the notation for h.) We then denote the forward-difference operator by $D_+(h)$, the backward operator by $D_-(h)$ and centered operators by $D_0(h)$. Thus, we have the following:

$$D_+(h)f_i = \frac{f_{i+1} - f_i}{h} = f'(x_i) + \mathcal{O}(h)\ , \qquad \text{(forward)} \tag{3.51a}$$

$$D_-(h)f_i = \frac{f_i - f_{i-1}}{h} = f'(x_i) + \mathcal{O}(h)\ , \qquad \text{(backward)} \tag{3.51b}$$

$$D_0(h)f_i = \frac{f_{i+1} - f_{i-1}}{2h} = f'(x_i) + \mathcal{O}(h^2)\ . \qquad \text{(centered)} \tag{3.51c}$$

When we require partial derivative approximations, say of a function $f(x, y)$, we alter the above notation appropriately with, for example, either $D_x(h)$ or $D_y(h)$. Hence, for the centered difference we have

$$D_{0,x}(h)f_{i,j} = \frac{f_{i+1,j} - f_{i-1,j}}{2h} = \frac{\partial f}{\partial x}(x_i, y_i) + \mathcal{O}(h^2)\ , \tag{3.52}$$

and

$$D_{0,y}(h)f_{i,j} = \frac{f_{i,j+1} - f_{i,j-1}}{2h} = \frac{\partial f}{\partial y}(x_i, y_i) + \mathcal{O}(h^2) \ . \tag{3.53}$$

We noted earlier that approximation of higher derivatives is carried out in a manner completely analogous to what is done in deriving analytical derivative formulas. Namely, we utilize the fact that the $(n+1)^{th}$ derivative is just the (first) derivative of the n^{th} derivative:

$$\frac{d^{n+1}}{dx^{n+1}} f = \frac{d}{dx}\left(\frac{d^n f}{dx^n}\right) .$$

In particular, in difference-operator notation, we have

$$D^{n+1}(h)f_i = D(h)\left(D^n(h)f_i\right) .$$

We previously used this to obtain a first-order approximation of f'', but without the formal notation. We will now derive the centered second-order approximation:

$$\begin{aligned} D_0^2 f_i = D_0(D_0 f_i) &= D_0\left[\frac{1}{2h}(f_{i+1} - f_{i-1})\right] \\ &= \frac{1}{2h}\left[\left(\frac{f_{i+2} - f_i}{2h}\right) - \left(\frac{f_i - f_{i-2}}{2h}\right)\right] \\ &= \frac{1}{(2h)^2}(f_{i-2} - 2f_i + f_{i+2}) . \end{aligned}$$

We observe that this approximation exactly corresponds to a step size of $2h$, rather than to h, since all indices are incremented by two, and only $2h$ appears explicitly. Hence, it is clear that the approximation over a step size h is

$$D_0^2 f_i = \frac{f_{i-1} - 2f_i + f_{i+1}}{h^2} = f''(x_i) + \mathcal{O}(h^2) . \tag{3.54}$$

In recursive construction of centered schemes, approximations containing more than the required range of grid point indices always occur because the basic centered operator spans a distance $2h$. It is left to the reader to verify that (3.54) can be obtained directly, by using the appropriate definition of $D_0(h)$ in terms of indices $i - \frac{1}{2}$ and $i + \frac{1}{2}$. We also note that it is more common to derive this using a combination of forward and backward first-order differences, $D_+ D_- f_i$.

3.3.4 Differentiation of interpolation polynomials

There are two remaining approximation methods which are related to differencing, and which are widely used. The first is divided differences. We will not treat this method here, but instead discuss the second approach, which gives identical results. It is simply differentiation of the Lagrange (or other) interpolation polynomial. Suppose we are required to produce a second-order accurate derivative approximation at the point x_i. Now we expect, on the basis of earlier discussions, that differentiation of a polynomial approximation will reduce the order of accuracy by one power of the step size. Thus, if we need a first derivative approximation that is second-order accurate, we must start with a polynomial which approximates functions to third order.

Hence, we require a quadratic, which we formally express as

$$p_2(x) = \sum_{i=1}^{3} \ell_i(x) f_i = f(x) + \mathcal{O}(h^3) ,$$

where $h = \max |x_i - x_j|$. Then we have

$$\begin{aligned} f'(x) &= p_2'(x) + \mathcal{O}(h^2) \\ &= \sum_{i=1}^{3} \ell_i'(x) f_i + \mathcal{O}(h^2)\,. \end{aligned}$$

We can now obtain values of f' for any $x \in [x_1, x_3]$. In general, we typically choose the x so that $x = x_i$ for some i. The main advantage of this Lagrange polynomial approach is that it does not require uniform spacing of the x_is, such as is required by all of the procedures presented earlier. (It should be noted, however, that Taylor series methods can also be employed to develop difference approximations over nonequally spaced points; but we shall not pursue this here).

3.4 Richardson Extrapolation Revisited

We have previously used Richardson extrapolation to construct Simpson's rule from trapezoidal quadrature, and also to obtain higher-order difference approximations; it is also the basis for the Romberg integration method. In recent years Richardson extrapolation has come into wide use in several important areas of computational numerical analysis, and because of this we feel a more general treatment is needed than can be deduced merely from the specific applications discussed above. In all of these examples we were extrapolating a procedure from second- to fourth-order accuracy, which depends upon the fact that only even powers of the step size appear in the truncation error expansion. Furthermore, extrapolation was always done between step sizes differing by a factor of two. There are many cases in which all powers of h (and for that matter, not just integer powers) may occur in the truncation error expansion. Moreover, for one reason or another, it may not be convenient (or even possible) to employ step sizes differing by a factor of two. Hence, it is important to be able to construct the extrapolation procedure in a general way so as to remove these two restrictions.

Let $\{x_i\}_{i=1}^n$ be a partition of the interval $[a, b]$ corresponding to a uniform step size $h = x_{i+1} - x_i$. Let $f(x_i)$ be the exact values of a function f defined on this interval, and let $\{f_i^h\}_{i=1}^n$ denote the corresponding numerical approximation. Hence,

$$f_i^h = f(x_i) + \tau_1 h^{q_1} + \tau_2 h^{q_2} + \mathcal{O}\left(h^{q_3}\right) \tag{3.55}$$

for some <u>known</u> $q_m \in \mathbb{R},\ m = 1, 2, \ldots$, and (possibly) unknown $\tau_m \in \mathbb{C}$ which also depend on the grid point x_i. We have earlier seen in a special case that it is not necessary to know the τ_m.

The functions f_i^h, usually called *grid functions*, may arise in essentially any way. They may result from interpolation or differencing of f; they may be a definite integral of some other function, say g, as in our quadrature formulas discussed earlier (in which case the i index is superfluous); or they might be the approximate solution to some differential or integral equation. We will not here need to be concerned with the origin of the f_i^hs.

Let us now suppose that a second approximation to $f(x)$ has been obtained on a partition of $[a, b]$ with spacing $rh,\ r > 0$ and $r \neq 1$. We represent this as

$$f_i^{rh} = f(x_i) + \tau_1 (rh)^{q_1} + \tau_2 (rh)^{q_2} + \mathcal{O}\left(h^{q_3}\right). \tag{3.56}$$

We note here that we must suppose there are points x_i common to the two partitions of $[a, b]$. Clearly, this is not a serious restriction when r is an integer or the reciprocal of an integer. In general, if the f_is are being produced by any operation other than interpolation, we can always, in

principle, employ high-order interpolation formulas to guarantee that the grid function values are known at common values of $x \in [a, b]$ for both values of step size.

We now rewrite (3.56) as

$$f_i^{rh} = f(x_i) + r^{q_1}\tau_1 h^{q_1} + r^{q_2}\tau_2 h^{q_2} + \mathcal{O}\left(h^{q_3}\right).$$

From this it is clear that the q_1^{th}-order error term can be removed by multiplying (3.55) by r^{q_1}, and substracting this from (3.56). This leads to

$$\begin{aligned} f_i^{rh} - r^{q_1} f_i^h &= f(x_i) - r^{q_1} f(x_i) + r^{q_2}\tau_2 h^{q_2} - r^{q_1}\tau_2 h^{q_2} + \mathcal{O}\left(h^{q_3}\right) \\ &= \left(1 - r^{q_i}\right) f(x_i) + \mathcal{O}\left(h^{q_2}\right). \end{aligned}$$

We now divide through by $(1 - r^{q_1})$ to obtain

$$\frac{f_i^{rh} - r^{q_1} f_i^h}{1 - r^{q_1}} = f(x_i) + \mathcal{O}\left(h^{q_2}\right).$$

From this it is clear that we should define the general extrapolated quantity f_i^* as

$$f_i^* \equiv \frac{r^{q_1} f_i^h - f_i^{rh}}{r^{q_1} - 1} = f(x_i) + \mathcal{O}\left(h^{q_2}\right). \tag{3.57}$$

We now demonstrate for the special cases treated earlier, for which $q_1 = 2,\ q_2 = 4$, and $r = 2$, that the same result is obtained using Eq. (3.57) as found previously; namely

$$f_i^* = \frac{4f_i^h - f^{2h}}{4 - 1} = \frac{1}{3}\left(4f_i^h - f_i^{2h}\right) = f(x_i) + \mathcal{O}(h^4)\ .$$

Similarly, if the leading term in the truncation error expansion is first order, as was the case with several of the difference approximations presented in the preceding section, we have $q_1 = 1$, $q_2 = 2$, and for $r = 2$ (3.57) yields

$$f_i^* = 2f_i^h - f_i^{2h} = f(x_i) + \mathcal{O}(h^2)\ .$$

We again emphasize that the Richardson extrapolation procedure can be applied to either computed numerical results (numbers), or to the discrete formulas used to produce the results. The optimal choice between these alternatives is typically somewhat problem dependent—there are no general prescriptions.

3.5 Computational Test for Grid Function Convergence

Whenever solutions to a problem are obtained via numerical approximation it is necessary to investigate their accuracy. Clearly, if we know the solution to the problem we are solving ahead of time we can always exactly determine the error of the numerical solution. But, of course, if we already know the answer, we would not need a numerical solution in the first place, in general. (An important exception is the study of "model" problems when validating a new algorithm and/or computer code.)

It turns out that a rather simple test for accuracy can—and should—always be performed on solutions represented by a grid function. Namely, we employ a Cauchy convergence test on the grid function in a manner quite similar to that discussed in Chap. 1 for testing convergence of iteration procedures, now as discretization step sizes are reduced. For grid functions, however, we generally have available additional qualitative information, derived from the numerical method itself, about

the theoretical convergence rate of the grid functions generated by the method. In particular, we almost always have the truncation error expansion at our disposal. We have derived numerous of these throughout this chapter.

For example, from (3.55) we see that

$$f_i^h = f(x_i) + \tau_1 h^{q_1} + \cdots = f(x_i) + \mathcal{O}\left(h^{q_1}\right),$$

and by changing the step size to rh we have

$$f_i^{rh} = f(x_i) + \tau_1 r^{q_1} h^{q_1} + \cdots .$$

The dominant error in the first case is

$$e_i^h \equiv f(x_i) - f_i^h = -\tau_1 h^{q_1} \ , \tag{3.58}$$

and in the second case it is

$$e_i^{rh} = f(x_i) - f_i^{rh} = -\tau_1 r^{q_1} h^{q_1} \ , \tag{3.59}$$

provided h is sufficiently small to permit neglect of higher-order terms in the expansions. Thus, the theoretical ratio of the errors for two different step sizes is known to be simply

$$\frac{e_i^{rh}}{e_i^h} = r^{q_1} \ . \tag{3.60}$$

Hence, for a second-order method ($q_1 = 2$) a reduction in the step size by a factor of two ($r = \frac{1}{2}$) leads to a reduction in error given by

$$r^{q_1} = \left(\frac{1}{2}\right)^2 = \frac{1}{4};$$

i.e., the error is reduced by a factor of four.

In practical problems we usually do not know the exact solution, $f(x)$; hence we cannot calculate the true error. However, if we obtain three approximations to $f(x)$, say $\left\{f_i^h\right\}$, $\left\{f_i^{h/2}\right\}$ and $\left\{f_i^{h/4}\right\}$, we can make good estimates of τ_1, q_1 and $f(x_i)$ at all points x_i for which elements of all three grid functions are available. This merely involves solving the following system of three equations for τ_1, q_1 and $f(x_i)$:

$$\begin{aligned} f_i^h &= f(x_i) + \tau_1 h^{q_1} \ , \\ f_i^{h/2} &= f(x_i) + 2^{-q_1}\tau_1 h^{q_1} \ , \\ f_i^{h/4} &= f(x_i) + 4^{-q_1}\tau_1 h^{q_1} \ . \end{aligned}$$

Now recall that f_i^h, $f_i^{h/2}$, $f_i^{h/4}$ and h are all known values. Thus, we can substract the second equation from the first, and the third from the second, to obtain

$$f_i^h - f_i^{h/2} = \left(1 - 2^{-q_1}\right)\tau_1 h^{q_1} , \tag{3.61}$$

and

$$f_i^{h/2} - f_i^{h/4} = 2^{-q_1}\left(1 - 2^{-q_1}\right)\tau_1 h^{q_1} . \tag{3.62}$$

Then the ratio of these is

$$\frac{f_i^h - f_i^{h/2}}{f_i^{h/2} - f_i^{h/4}} = 2^{q_1} \ , \tag{3.63}$$

which is equivalent to the result (3.60) obtained above using true error. Again note that q_1 should be known, theoretically; but in practice, due either to algorithm/coding errors or simply to use of step sizes that are too large, the theoretical value of q_1 may not be attained at all (or possibly at any!) grid points x_i.

This motivates us to solve Eq. (3.63) for the actual value of q_1:

$$q_1 = \frac{\log\left[\dfrac{f_i^h - f_i^{h/2}}{f_i^{h/2} - f_i^{h/4}}\right]}{\log 2} \ . \tag{3.64}$$

Then from Eq. (3.61) we obtain

$$\tau_1 = \frac{f_i^h - f_i^{h/2}}{\left(1 - 2^{-q_i}\right) h^{q_1}} \ . \tag{3.65}$$

Finally, we can now produce an even more accurate estimate of the exact solution (equivalent to Richardson extrapolation) from any of the original equations; *e.g.*,

$$f(x_i) = f_i^h - \tau_1 h^{q_1} \ . \tag{3.66}$$

In most practical situations we are more interested in simply determining whether the grid functions converge and, if so, whether convergence is at the expected theoretical rate. To do this it is usually sufficient to replace $f(x_i)$ in the original expansions with a value f_i computed on a grid much finer than any of the test grids, or a Richardson extrapolated value obtained from the test grids, say f_i^*. The latter is clearly more practical, and for sufficiently small h it leads to

$$\tilde{e}_i^h = f_i^* - f_i^h \cong -\tau_1 h^{q_1} \ .$$

Similarly,

$$\tilde{e}_i^{h/2} = f_i^* - f_i^{h/2} \cong -2^{-q_1}\tau_1 h^{q_1} \ ,$$

and the ratio of these errors is

$$\frac{\tilde{e}_i^h}{\tilde{e}_i^{h/2}} \cong \frac{f_i^* - f_i^h}{f_i^* - f_i^{h/2}} = 2^{q_1} . \tag{3.67}$$

Yet another alternative (and in general, probably the best one when only grid function convergence is the concern) is simply to use Eq. (3.63), *i.e.*, employ a Cauchy convergence test. As noted above we generally know the theoretical value of q_1. Thus, the left side (obtained from numerical computation) can be compared with the right side (theoretical). Even when q_1 is not known we can gain qualitative information from the left-hand side alone. In particular, it is clear that the right-hand side is always greater than unity. Hence, this should be true of the left-hand side. If the equality in the appropriate one of (3.63) or (3.67) is not at least approximately satisfied, the first thing to do is reduce h, and repeat the analysis. If this does not lead to closer agreement between left- and right-hand sides in these formulas, it is fairly certain that there are errors in the algorithm and/or its implementation.

We note that the above procedures can be carried out for arbitrary sequences of grid spacings, and for multi-dimensional grid functions. But in both cases the required formulas are more involved, and we leave investigation of these ideas as exercises for the reader. Finally, we must recognize that e_i^h (or $\tilde{e}_i^h$) is error at a single grid point. In most practical problems it is more appropriate to employ an error norm computed with the entire solution vector. Then (3.63), for example, would be replaced with

$$\frac{\|e^h\|}{\|e^{h/2}\|} \cong \frac{\|f^h - f^{h/2}\|}{\|f^{h/2} - f^{h/4}\|} = 2^{q_1} \ , \tag{3.68}$$

for some norm $\|\cdot\|$, say, the vector 2-norm.

3.6 Summary

This chapter has been devoted to presenting a series of topics which, taken together, might be called "classical numerical analysis." They often comprise a first course in numerical analysis consisting of interpolation, quadrature and divided differences. As will be evident in the sequel, these topics provide the main tools for development of numerical methods for differential equations.

As we have attempted to do throughout these lectures, we have limited the material of this chapter to only the most basic methods from each class. But we wish to emphasize that, indeed, these are also the most widely used for practical problem solving. All of the algorithms presented herein (and many similar ones) are available in various commercial software suites, and above all else we hope the discussions presented here will have provided the reader with some intuition into the workings of such software.

Chapter 4

Numerical Solution of Ordinary Differential Equations

Development of the ability to solve essentially any ordinary differential equation (ODE) on the digital computer is one of the major achievements of modern numerical analysis. While it is not the case that absolutely any and every ODE problem can be solved easily, it is true that the great majority of problems that occur in practice can be solved quite efficiently with a well-chosen method. This is true generally, independent of whether the problems are linear or nonlinear.

There are two types of problems associated with ODEs: *i*) initial-value problems (IVPs), and *ii*) boundary-value problems (BVPs). Numerical methods for IVPs are highly developed, and except for extremely large systems or problems involving very high-frequency oscillations, there exist many standard software packages that can be readily used. The main purpose of discussions here will be to treat very basic methods that will provide insight into construction of such software.

The situation is not quite so favorable for BVPs, especially for nonlinear problems and for systems. One of the main approaches is to convert these problems to IVPs, and then use the initial-value packages. This, however, has no guarantee of success; our approach to BVPs will be to use techniques developed specifically for such problems. These have been demonstrated to be very effective, and with the aid of Richardson extrapolation can be highly accurate. Moreover, they have direct application to construction of methods for numerical solution of partial differential equation BVPs.

The chapter is subdivided into two main sections, each providing treatment of one of the main types of ODE problems: Sec. 1 will deal with initial-value problems, and Sec. 2 with boundary-value problems.

4.1 Initial-Value Problems

In this section we discuss some of the main methods for solving initial-value problems for ordinary differential equations. We begin with a brief mathematical background for such problems, and then proceed to treat single-step numerical methods, multi-step methods and finally techniques for solving stiff ODEs.

4.1.1 Mathematical Background

Initial-value problems for n^{th}-order ODEs can always be cast in the following general form:

$$F\left(u^{(n)}, u^{(n-1)}, \ldots, u', u, t\right) = 0 \tag{4.1}$$

with initial data

$$\begin{aligned} u(t_0) &= c_1 \\ u'(t_0) &= c_2 \\ &\vdots \\ u^{(n-1)}(t_0) &= c_n \ , \end{aligned} \tag{4.2}$$

where F, $u \in \mathbb{R}^m$ and $t \in \mathbb{R}^1$. Parenthesized superscripts here denote order of differentiation. We should start by noting that an n^{th}-order differential equation will require n initial conditions, as given in (4.2), to permit evaluation of the n integration constants arising during the n formal integrations needed to obtain a solution.

We will not usually encounter systems of ODEs in a form as general as (4.1), (4.2). It is often, but not always, possible to solve each equation in the system for its highest derivative term. When this is the case (4.1) can be written as

$$\frac{d^n u}{dt^n} = f\left(u^{(n-1)}, u^{(n-2)}, \ldots, u', u, t\right), \tag{4.3}$$

with $f \in \mathbb{R}^m$ related to F in an obvious way. The initial data (4.2) still apply in this case.

It is an important fact that any equation of the form (4.3) can be written as a system of n first-order equations. Moreover, initial conditions of the form (4.2) provide the correct initial data for this system, as we will show below. The consequence of all this is that study of the numerical initial-value problem can be restricted to first-order systems without loss of generality. This approach, although not the only possibility, is essentially always employed, and we will follow it here.

To express (4.3) as a first-order system, let

$$y_1 = \frac{du}{dt} \ , \quad y_2 = \frac{dy_1}{dt} \left(= \frac{d^2u}{dt^2}\right) \ , \quad \cdots \ , \quad y_{n-1} = \frac{dy_{n-2}}{dt} \left(= \frac{d^{n-1}u}{dt^{n-1}}\right) .$$

Now, observe that

$$\frac{dy_{n-1}}{dt} = \frac{d}{dt}\left(\frac{dy_{n-2}}{dt}\right) = \frac{d^2y_{n-2}}{dt^2} = \frac{d^3y_{n-3}}{dt^3} = \cdots = \frac{d^{n-1}y_1}{dt^{n-1}} = \frac{d^n u}{dt^n} = f \ ,$$

which shows, in particular, that $dy_{n-1}/dt = f$.

Thus, the system of first-order equations corresponding to (4.3) is

$$\begin{aligned} \frac{du}{dt} &= y_1 \\ \frac{dy_1}{dt} &= y_2 \\ &\cdot \\ &\cdot \\ &\cdot \\ \frac{dy_{n-2}}{dt} &= y_{n-1} \\ \frac{dy_{n-1}}{dt} &= f\left(y_{n-1}, y_{n-2}, \ldots, y_1, u, t\right) . \end{aligned} \tag{4.4}$$

This is a system of n equations for u, the desired solution, and its first $n-1$ derivatives. The required initial data are given in (4.2). In particular, we have

$$\begin{aligned} u(t_0) &= c_1 \\ u'(t_0) &= y_1(t_0) = c_2 \\ &\cdot \\ &\cdot \\ &\cdot \\ u^{(n-2)}(t_0) &= y_{n-2}(t_0) = c_{n-1} \\ u^{(n-1)}(t_0) &= y_{n-1}(t_0) = c_n \,. \end{aligned} \tag{4.5}$$

Equations (4.4) and (4.5) are the form of (4.1) and (4.2) that we would input to standard ODE software packages as found, for example, in the IMSL or NAG libraries, and it is the only form to be treated here.

We next demonstrate the procedure of going from (4.1), (4.2) to (4.4), (4.5) for a specific case. Consider the following problem:

$$\left(\frac{d^2u_1}{dt^2}\right)^2 + \sin\left(\frac{du_1}{dt}\right) - \left(\frac{du_2}{dt}\right)^{\frac{1}{2}} - u_2 - \cos\omega t = 0\,, \tag{4.6a}$$

$$\frac{d^3u_2}{dt^3} - \exp\left(\frac{d^2u_2}{dt^2}\right) + u_2 - \frac{du_1}{dt} + u_1 = 0\,. \tag{4.6b}$$

Since (4.6a) is second order, two initial conditions will be needed, while (4.6b) is third order, and will require three conditions. In particular, we have

$$\begin{aligned} u_1(t_0) &= a \qquad & u_2(t_0) &= c \\ u_1'(t_0) &= b \qquad & u_2'(t_0) &= d \\ & & u_2''(t_0) &= e\ . \end{aligned} \tag{4.7}$$

The first step is to "solve" each of (4.6a) and (4.6b) for their respective highest derivative. For (4.6a) we obtain

$$\frac{d^2u_1}{dt^2} = \left\{\cos\omega t + u_2 + \left(\frac{du_2}{dt}\right)^{\frac{1}{2}} - \sin\left(\frac{du_1}{dt}\right)\right\}^{\frac{1}{2}} \equiv f_1\ , \tag{4.8a}$$

and from (4.6b)

$$\frac{d^3u_2}{dt^3} = \exp\left(\frac{d^2u_2}{dt^2}\right) - u_2 + \frac{du_1}{dt} - u_1 \equiv f_2\ . \tag{4.8b}$$

We are now prepared to convert this to a first-order system. Let

$$\begin{aligned} y_1 = \frac{du_1}{dt}, \quad \frac{dy_1}{dt} = \frac{d^2u_1}{dt^2} = f_1, \\ z_1 = \frac{du_2}{dt}, \quad z_2 = \frac{dz_1}{dt}, \quad \frac{dz_2}{dt} = \frac{d^3u_2}{dt^3} = f_2\ . \end{aligned} \tag{4.9}$$

Then, the system takes the form

$$\begin{aligned}
\frac{du_1}{dt} &= y_1 \\
\frac{dy_1}{dt} &= \left\{ \cos\omega t + u_2 + \left(\frac{du_2}{dt}\right)^{\frac{1}{2}} - \sin\left(\frac{du_1}{dt}\right) \right\}^{\frac{1}{2}} \\
\frac{du_2}{dt} &= z_1 \\
\frac{dz_1}{dt} &= z_2 \\
\frac{dz_2}{dt} &= \exp\left(\frac{d^2u_2}{dt^2}\right) - u_2 + \frac{du_1}{dt} - u_1 \ .
\end{aligned}$$

By making use of the definitions (4.9), we can write these equations entirely in terms of the dependent variables appearing on the left-hand sides. The final form of the system, with appropriate initial conditions is

$$\begin{aligned}
\frac{du_1}{dt} &= y_1 && u_1(t_0) = a \\
\frac{dy_1}{dt} &= \{\cos\omega t + u_2 + \sqrt{z_1} - \sin y_1\}^{\frac{1}{2}} && y_1(t_0) = b \\
\frac{du_2}{dt} &= z_1 && u_2(t_0) = c \\
\frac{dz_1}{dt} &= z_2 && z_1(t_0) = d \\
\frac{dz_2}{dt} &= \exp(z_2) - u_2 + y_1 - u_1 && z_2(t_0) = e \, .
\end{aligned} \tag{4.10}$$

The system (4.10) is coupled and strongly nonlinear. Existence and/or uniqueness of solutions for such systems is not guaranteed, *a priori*. We shall not address such questions in this presentation; but the reader must be warned that since numerical methods for initial-value problems can be applied irrespective of nonlinearities, computed results to problems such as (4.10) must be viewed with some caution.

A second remark is also important at this time. It is that although we have demonstrated that any higher-order system of the form (4.3) can be written as an equivalent first-order system, it is not true that every first-order system arises as a reduction of some higher-order equation. Thus, any general algorithm for solving ODE IVPs must be written for general systems, and not merely for the special form shown in (4.4).

4.1.2 Basic Single-Step Methods

In this section we will begin with a detailed treatment of the simplest single-step method, the forward Euler method; we will then consider the backward Euler procedure. Following this we present analyses of two higher-order methods, the implicit trapezoidal scheme and its explicit counterpart Heun's method, which is an explicit second-order Runge–Kutta method.

Explicit (Forward) Euler Method

The first method we consider, Euler's method, is one that should almost never be used in practice in the context of ODEs (but we will later see that it is often used in solution techniques for partial

differential equations). However, it is extremely simple and thus provides a useful pedagogical tool. It will be evident that the method is easily coded even on a hand-held calculator, and we might at times consider using it in this context. We will treat only a single first-order equation since the extension to systems is mainly a matter of notation.

Consider the equation

$$u' = f(u,t) \ , \tag{4.11}$$

with initial condition $u(t_0) = u_0$. If we replace u' with a first forward difference and evaluate f at time t_n we obtain

$$\frac{u_{n+1} - u_n}{h} \cong f(u_n, t_n) \ ,$$

or

$$u_{n+1} \cong u_n + hf(u_n, t_n) \ . \tag{4.12}$$

Clearly if u_n is known, we can evaluate the right-hand side and thus explicitly calculate u_{n+1}. This is a characteristic feature of *explicit* methods: the grid function values at a new time step can be directly evaluated from values at previous time steps—without numerical linear algebra and/or iteration. We also observe that the right-hand side of (4.12) involves information from only a single previous time step. Hence, Euler's method is an explicit *single-step* method.

We now investigate the truncation error for Euler's method. Intuitively, we would expect this method to be only first-order accurate because we have used a first-order approximation to u'. Indeed this turns out to be true in a certain sense; but it is important to understand some details. Our treatment is not intended to be rigorous, but it is correct with respect to the basic notions. (For a rigorous development, we recommend Gear [10]). In going from the ODE to the difference approximation, the exact result would be (for sufficiently small h)

$$\frac{u_{n+1} - u_n}{h} + \mathcal{O}(h) = f(u_n, t_n) \ .$$

It is clear from this that as $h \to 0$, the original differential equation is recovered. Thus, Euler's method is said to be *consistent* with the differential equation. Analogous to (4.12) we have

$$u_{n+1} = u_n + hf(u_n, t_n) + \mathcal{O}(h^2) \ , \tag{4.13}$$

and in this form Euler's method would appear to be second-order accurate. However, the formula (4.13) advances the solution only a single time step, and during this step an error of order h^2 is incurred. This is called the *local truncation error*, and it is second order for Euler's method.

Now suppose we wish to solve (4.11) on the interval $(0, \tau]$, using N time steps. Then the (uniform) time step is $h = \tau/(N-1)$. From (4.13) we see that after $N-1$ steps (the number needed to obtain u_N) we will have accumulated a truncation error equal to $(N-1) \cdot \mathcal{O}(h^2)$, analogous to what we found earlier in analysis of trapezoidal quadrature in Chap. 3. Thus, the *global truncation error* for Euler's method is $\mathcal{O}(h)$. The *order of a method* is always taken to be the order of the global truncation error; hence, Euler's method is first-order accurate just as we originally expected.

We next consider the *stability* of Euler's method. There are many different definitions employed in studying stability, but basically, we consider a difference approximation to a given problem to be stable if the solution to the difference equation does not blow up any faster than the solution to the differential equation. For the analyses presented herein, we employ the following somewhat more precise statement of this idea.

Definition 4.1 *A method is absolutely stable for a given step size h, and for a given differential equation, if the change due to a perturbation of size δ in any mesh value u_m is no larger than $\delta \ \forall \ u_n$, $n > m$.*

We note that the perturbations referred to in the definition typically arise from round-off error in machine computation.

To examine absolute stability of Euler's method we consider a very simple IVP,

$$u' = \lambda u \ , \qquad u(0) = u_0 \ , \qquad \lambda \equiv \text{const.} < 0 \, . \tag{4.14}$$

The exact solution to this problem is

$$u(t) = u_0 e^{\lambda t} \, ,$$

so if $\lambda < 0$, the solution decays to zero exponentially. The Euler's method approximation to (4.14) is

$$u_{n+1} = u_n + \lambda h u_n = (1 + \lambda h) u_n \, .$$

Thus,

$$\begin{aligned} u_1 &= (1 + \lambda h) u_0 \\ u_2 &= (1 + \lambda h) u_1 = (1 + \lambda h)^2 u_0 \\ &\ \ \vdots \\ u_n &= (1 + \lambda h) u_{n-1} = \cdots = (1 + \lambda h)^n u_0 \, . \end{aligned}$$

Observe that this is an exact solution to the difference equation corresponding to Euler's method.

Now suppose u_0 is replaced by $v_0 = u_0 + \delta$, where δ is the error in machine representation of u_0 and thus corresponds to a perturbation in the sense of the definition of absolute stability. For example, if $u_0 = \pi$, $|\delta| > 0$ will hold on any machine, no matter what the word size happens to be because it will always be finite, and π does not have a finite exact representation. After n steps we have

$$\begin{aligned} v_n &= (1 + \lambda h)^n (u_0 + \delta) \\ &= (1 + \lambda h)^n u_0 + (1 + \lambda h)^n \delta \\ &= u_n + (1 + \lambda h)^n \delta \, . \end{aligned}$$

Now define the error at time step n to be

$$z_n = v_n - u_n = (1 + \lambda h)^n \delta \, ,$$

which represents the growth in time of the perturbation δ.

Then, taking $z_0 = \delta$, we see that the error satisfies the same difference equation as does the solution, u_n, and after n steps the original error δ will have been amplified to

$$(1 + \lambda h)^n \delta \, .$$

It follows that in order to guarantee absolute stability for Euler's method we must have

$$|1 + \lambda h| \leq 1 \ . \tag{4.15}$$

The quantity on the left is called the *amplification factor*.

In general, we permit λ to be complex, and the region of absolute stability of Euler's method is usually drawn in the complex λh-plane, as shown in Fig. 4.1: the stable region for Euler's method applied to (4.14) is simply the disc centered at $(-1, 0)$ and having unit radius. If $\lambda \in \mathbb{R}^1$, then

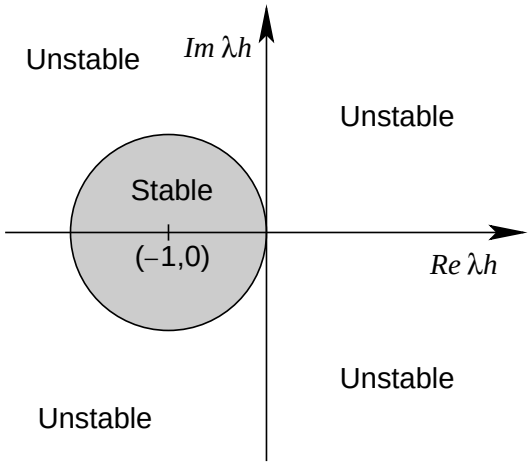


Figure 4.1: Region of absolute stability for Euler's method applied to $u' = \lambda u$

Euler's method is stable for $\lambda h \in [-2, 0]$. This shows that the method is never absolutely stable if $\lambda > 0$, and for $\lambda < 0$, but $|\lambda| \gg 1$, the step sizes required for stability are extremely small. For these reasons, Euler's method is seldom used in practical calculations.

It is interesting to observe the effects of instability on the computed solution. For simplicity, we take $u_0 = 1$ in (4.14), and set $\lambda = -100$. At $t = 1$, the exact solution is $u(1) \cong 3.72 \times 10^{-44} \simeq 0$. From (4.15) we see that to guarantee stability we must choose the step size h so that

$$|1 - 100h| \leq 1 \ ,$$

or

$$1 - 100h \leq 1 \quad \text{and} \quad 1 - 100h \geq -1 \ .$$

The first of these implies $h \geq 0$, which we would expect in any case. From the second we find

$$h \leq \frac{1}{50} \ .$$

With h exactly equal to $\frac{1}{50}$, the amplification factor is unity, and Euler's method becomes

$$u_n = (-1)^n u_0 \ .$$

Hence, as the time steps proceed $u(t) = \pm u_0 = \pm 1$ which remains bounded, but is completely wrong.

Next consider $h = 0.01$. In this case, the amplification factor is zero, and $u_n \equiv 0 \ \ \forall \ n > 0$. This is very inaccurate close to $t = 0$, but asymptotically correct as $t \to \infty$. If we set $h = 0.001$, the amplification factor is 0.9. To reach $t = 1$ we need 1000 time steps, so the Euler's method solution is

$$u_{1000} = (0.9)^{1000} u_0 \cong 1.748 \times 10^{-46} \ ,$$

which is at least beginning to resemble the exact solution. Moreover, the computed solution at $t = 0.1$ is $u_{100} \cong 2.66 \times 10^{-5}$ compared with the exact solution $u(.1) = 4.5 \times 10^{-5}$. Hence, the method produces results that are at least of the correct order of magnitude for short times, and decreasing h to $h = 0.0001$ yields reasonably accurate results.

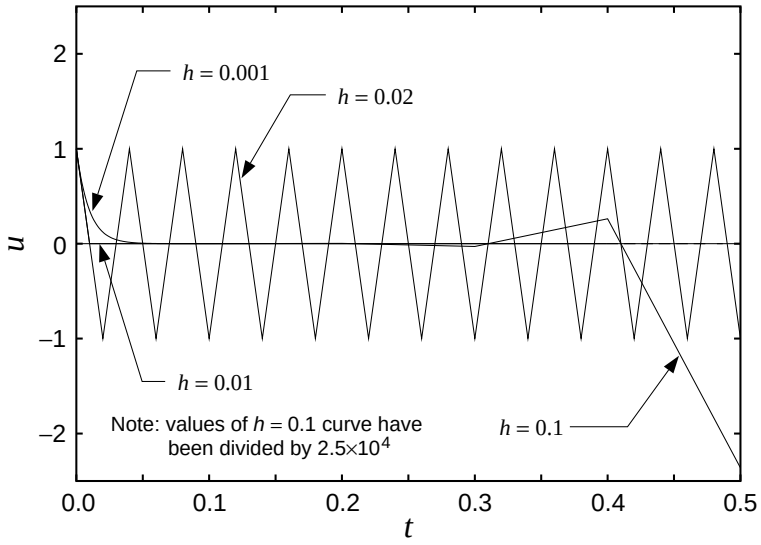


Figure 4.2: Forward-Euler solutions to $u' = \lambda u$, $\lambda < 0$

Finally, we consider $h = 0.1$; then the amplification factor is 9, and Euler's method becomes

$$u_n = (-9)^n u_0 \ .$$

To integrate to $t = 1$, we need 10 steps, and this leads to

$$u_{10} \cong 3.487 \times 10^9 \ ,$$

which is completely ridiculous.

Figure 4.2 summarizes these results. Of particular note is the phenomenon of growing oscillations in the unstable case. Such oscillations provide a tell-tale symptom of instability rather generally in numerical integration procedures, and their appearance is a signal that the time step size should be reduced.

We conclude from all this that Euler's method is generally unreliable because of its poor stability characteristics, and it should be used only with extreme caution.

Implicit (Backward) Euler Method

There is a relatively simple remedy to the stability problems encountered with Euler's method. Recall that in developing this method we replaced u' with a forward-difference approximation. We now instead use a backward approximation to obtain

$$\frac{u_n - u_{n-1}}{h} = f(u_n, t_n) \ ,$$

or

$$u_n = u_{n-1} + hf(u_n, t_n) \ .$$

If we translate this forward by one time step the result is analogous to the earlier form of Euler's method (4.12):

$$u_{n+1} = u_n + hf(u_{n+1}, t_{n+1}) \ . \tag{4.16}$$

This approximation is known as the *backward Euler* method. It is still a single-step method, but it is now *implicit*. In particular, we cannot in general calculate a new time step by merely plugging in results from a preceding time step. Usually, f will be a nonlinear function of u_{n+1}; so (4.16) is a nonlinear equation for u_{n+1} that can be solved by a fixed-point algorithm applied at each time step. Newton's method is typically used. Thus, we write (4.16) as

$$F(u_{n+1}) = u_{n+1} - u_n - hf(u_{n+1}, t_{n+1}) = 0 \ , \tag{4.17}$$

and application of Newton's method leads to

$$u_{n+1}^{(m+1)} = u_{n+1}^{(m)} - \frac{F\left(u_{n+1}^{(m)}\right)}{F'\left(u_{n+1}^{(m)}\right)} \ , \tag{4.18}$$

where " $'$ " denotes differentiation with respect to u_{n+1}:

$$F'(u_{n+1}) = 1 - h\frac{\partial f}{\partial u_{n+1}} \ . \tag{4.19}$$

The truncation error analysis for backward Euler can be carried out analogously to what was done for forward Euler, and the result is the same; the backward Euler method is first-order accurate. We leave demonstration of this as an exercise for the reader.

We now consider the stability properties of (4.16) with respect to the IVP (4.14). In this case f is linear in u, so we have

$$u_{n+1} = u_n + \lambda h u_{n+1} \ ,$$

which can be solved for u_{n+1} without iteration (because of linearity):

$$u_{n+1} - \lambda h u_{n+1} = u_n \ ,$$

and

$$u_{n+1} = \frac{u_n}{1 - \lambda h} \ . \tag{4.20}$$

Analogous to what we did in analyzing absolute stability of (forward) Euler's method, we find for backward Euler that

$$u_n = \frac{1}{(1 - \lambda h)^n} u_0 \ . \tag{4.21}$$

Thus, the amplification factor is $|(1 - \lambda h)^{-1}|$, and the condition for absolute stability is

$$\frac{1}{|1 - \lambda h|} \leq 1 \ ,$$

or

$$|1 - \lambda h| \geq 1 \ . \tag{4.22}$$

This is all of the complex λh-plane except for the open disc of unit radius centered at $(1, 0)$.

Our first observation is that (4.22) is satisfied $\forall\ \lambda \leq 0$ (or $\lambda \ni \operatorname{Re}\lambda \leq 0$ if $\lambda \in \mathbb{C}$), independent of the step size h. Thus, for the problem considered previously, with $\lambda = -100$, there are no restrictions on h for maintaining stability. For example, for $h = \frac{1}{50}$, the amplification factor is $\frac{1}{3}$, so

$$u_n = \left(\frac{1}{3}\right)^n u_0 \ .$$

Hence $u_{50} \sim u(1) = 1.393 \times 10^{-24} \sim 0$. This is actually much larger than the exact result, but both are so near zero that in many practical situations the backward Euler result would be quite acceptable. If we choose $h = \frac{1}{100}$, we have $u_{100} \sim u(1) = 7.889 \times 10^{-31}$ which is significantly closer to the exact result. Finally, for $h = 0.1$, which is very unstable for forward Euler, the backward Euler result at $t = 1$ is $u_{10} \sim u(1) = \left(\frac{1}{11}\right)^{10} = 3.855 \times 10^{-11}$, which is still sufficiently close to zero for many practical purposes.

We close this section on Euler single-step methods with the following remarks. We have shown the Euler methods to be stable, at least for sufficiently small step size h, and we have also indicated that they are consistent with the differential equation whose solution is being approximated. But we have not proven that the numerical solutions actually converge to the solution of the differential equation as $h \to 0$. Here, we will simply note that such convergence is guaranteed by the combination of consistency and stability. We will consider this more formally and in greater detail, in the context of partial differential equations in Chap. 5.

Higher-Order Methods, General Remarks

Despite the stability of backward Euler, it is still only first-order accurate, and if high accuracy is required, very small time steps will be necessary. This means that a great amount of arithmetic will be needed to integrate to large values of final time, and the final accuracy may be significantly degraded by round-off error. Rather generally, numerical methods for solving ODE IVPs have a maximum attainable accuracy which occurs when the sum of truncation and round-off errors is a minimum, as depicted in Fig. 4.3. One can view this figure as representing the case of a fixed initial-value problem, solved to a fixed final time by a single method, using different stable step sizes. The conclusion to be drawn is that we should employ higher-order methods so that truncation error will be relatively small, even when large time steps are used. Then long integrations will be possible without an unreasonable accumulation of round-off error.

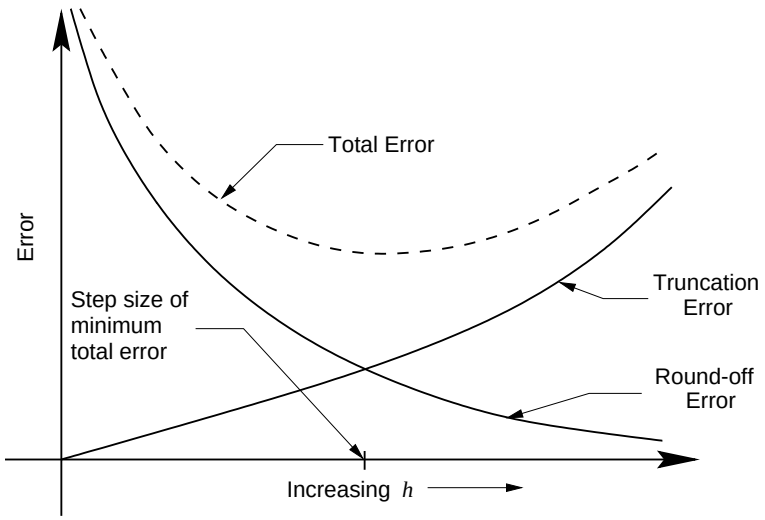


Figure 4.3: Comparison of round-off and truncation error

Trapezoidal Integration

The first higher-order method we shall treat here is the trapezoidal rule. Once again, we consider the first-order differential equation

$$u' = f(u,t) \tag{4.23}$$

with the initial condition

$$u(t_0) = u_0 \ . \tag{4.24}$$

As usual, we take $h = t_{n+1} - t_n$. If we integrate (4.23) between t_n and t_{n+1} we obtain

$$u_{n+1} - u_n = \int_{t_n}^{t_{n+1}} f(u,t)\ dt \ .$$

Now the left-hand side is exact, and we approximate the integral on the right with the (local) trapezoidal rule described in Chap. 3, which results in

$$u_{n+1} = u_n + \frac{h}{2}\left[f(u_{n+1},t_{n+1}) + f(u_n,t_n)\right] + \mathcal{O}(h^3) \ . \tag{4.25}$$

Recall that local truncation error for the trapezoidal rule is $\mathcal{O}(h^3)$, and by an argument analogous to that used for Euler's method, the global truncation error is $\mathcal{O}(h^2)$. Thus, trapezoidal integration is a second-order method, and as a consequence we expect truncation error to be reduced by a factor of four each time the step size h is halved.

An important observation to make regarding (4.25) is that it is implicit, and we must employ an iteration scheme very similar to that used above for the backward Euler method. As a consequence f must generally be evaluated for each Newton iteration, making this approach somewhat inefficient. On the other hand, rather generally, implicit methods have very favorable stability properties, and this is true for the trapezoidal rule. It can be shown that its region of absolute stability is essentially all of the left-half complex λh-plane, as we will indicate later. However, before we present details of a trapezoidal integration algorithm, it is of interest to consider an explicit method which can be easily obtained from the trapezoidal rule.

Heun's Method

Observe that in (4.25) we cannot explicitly evaluate $f(u_{n+1},t_{n+1})$ because we do not yet know u_{n+1}, but if we could estimate u_{n+1} with sufficient accuracy using only previous information we could evaluate (4.25) explicitly. In particular, denote this estimate as u^*_{n+1}, and rewrite (4.25) as

$$u_{n+1} = u_n + \frac{h}{2}\left[f(u^*_{n+1},t_{n+1}) + f(u_n,t_n)\right] . \tag{4.26}$$

The only candidate presently available for explicitly calculating u^*_{n+1} is Euler's method, so we set

$$u^*_{n+1} = u_n + hf(u_n,t_n) \ .$$

Substitution of this into (4.26) leads to

$$u_{n+1} = u_n + \frac{h}{2}\left[f\left(u_n + hf(u_n,t_n),t_{n+1}\right) + f(u_n,t_n)\right] \ . \tag{4.27}$$

This explicit method is known as *Heun's method*. We will later see that it is one of the simplest nontrivial Runge–Kutta schemes. It can also be viewed as a simple predictor-corrector technique. These will be discussed in more detail later in the study of multi-step methods.

We note here that Heun's method is also globally second order, despite the fact that we have used a first-order approximation to obtain u^*_{n+1}. To prove this we need to show that

$$f(u^*_{n+1}, t_{n+1}) = f(u_{n+1}, t_{n+1}) + \mathcal{O}(h^2) \ .$$

Then we see from (4.26) that the local error is $\mathcal{O}(h^3)$. To prove the above equality we will assume f is Lipschitz in u, but we note that this is no more restrictive than what is needed to guarantee uniqueness of solutions to (4.23) in the first place. If f is Lipschitz we have

$$\begin{aligned} \left|f(u^*_{n+1}, t_{n+1}) - f(u_{n+1}, t_{n+1})\right| &\le L \left|u^*_{n+1} - u_{n+1}\right| \\ &= L \left|u_n + hf(u_n, t_n) - u_{n+1}\right| \\ &\sim \mathcal{O}(h^2) \ . \end{aligned}$$

This holds because the expression in absolute values on the right-hand side is just Euler's method, which we have already shown in Eq. (4.13) to be locally second order. This provides the desired result.

Heun's method (4.27) is an easily programmed explicit scheme. It is significantly more accurate than Euler's method, and it also has somewhat better stability properties when applied to problems having oscillatory solutions. Nevertheless, it is important to note that there are many other higher-order methods which are more accurate, more stable and fairly efficient—but more difficult to construct.

Heun/Trapezoidal Algorithm

We will now develop a detailed algorithm that permits solution of general systems of first-order IVPs by either Heun's method or by the trapezoidal rule. As noted earlier, trapezoidal integration is implicit, and at each time step the (generally) nonlinear system

$$F_i(u_{n+1}) \equiv u_{i,n+1} - u_{i,n} - \frac{h}{2}\left[f(u_{i,n+1}, t_{n+1}) + f(u_{i,n}, t_n)\right] = 0 \qquad i = 1, \ldots, p \,, \tag{4.28}$$

with $u,\ f,\ F \in \mathbb{R}^p$ must be solved. This is done via Newton's method, so we must supply the Jacobian matrix of F as part of the problem input. We show that $J_u(F)$ can be expressed entirely in terms of the partial derivatives of f and the step size h. This will permit us to prepare input based only on the analytical problem being solved—thus not requiring problem-dependent changes to the discrete equations of the integration procedure.

Let

$$u_{n+1} = (u_{1,n+1}, u_{2,n+1}, \ldots, u_{p,n+1})^T$$

for a system of p first-order equations. Then for the i^{th} component of F, (4.28) leads to

$$\frac{\partial F_i}{\partial u_{j,n+1}} = \delta_{ij} - \frac{h}{2}\frac{\partial f_i}{\partial u_{j,n+1}} \qquad \forall\ i, j = 1, 2, \ldots, p \ . \tag{4.29}$$

Thus, we see that

$$J_u(F) = I - \frac{h}{2} J_u(f) \ ,$$

so in order to input a problem to be solved by the trapezoidal method we need only code the f_is and the elements of $J_u(f)$.

In addition, at each time step we need to supply an initial guess for the Newton iterations. Since we will produce an algorithm that also can employ Heun's method, it is reasonable to use this as

the initial guess for the trapezoidal rule at each time step. This leads to very rapid convergence of the Newton iterations. But we note that it is not really necessary to use such an elaborately constructed initial guess. It is usually quite acceptable to employ the result from the previous time step, or possibly an Euler's method prediction. We now summarize the preceding ideas in a pseudo-language algorithm.

Algorithm 4.1 *(Heun/Trapezoidal Method)*

1. *Read control flags:* $neqns,\ maxitr,\ nsteps$
   Read numerical parameters: $h,\ \delta,\ \epsilon$
   Read initial data: $(u_{i,0}, i = 1, neqns),\ t_0$

2. *Begin time stepping*
   Do $n = 0,\ nsteps - 1$
   $t_{n+1} = t_n + h$

3. *Begin Newton iterations for current time step*
   Do $m = 1,\ maxitr$
   If $m > 1$, *go to* 6

4. *Evaluate* $u^*_{i,n+1}$ *for use in Heun's method*
   Do $i = 1,\ neqns$
   $g_i = f(i, u_n, t_n)$
   $u^*_{i,n+1} = u_{i,n} + hg_i$
   Repeat i

5. *Calculate initial guess for trapezoidal rule from Heun's method*
   Do $i = 1,\ neqns$
   $u^{(0)}_{i,n+1} = u_{i,n} + \frac{h}{2}(g_i + f(i, u^*_{n+1}, t_{n+1}))$
   Repeat i
   If $maxitr = 1$, *go to* 11 *[Heun's Method]*

6. *Load* $J(f)$ *for Newton iteration*
   Call $Jacobn\left(neqns, u^{(m-1)}_{n+1}, t_{n+1}, J(f)\right)$

7. *Evaluate* $F\left(u^{(m-1)}_{n+1}\right),\ J(F)$
   Do $i = 1,\ neqns$
   $F_i = u^{(m-1)}_{i,n+1} - \frac{h}{2}\ f\left(i, u^{(m-1)}_{n+1}, t_{n+1}\right) - \left(u_{i,n} + \frac{h}{2}g_i\right)$
   Do $j = 1,\ neqns$
   $J(F)_{ij} = \delta_{ij} - \frac{h}{2}J(f)_{ij}$
   Repeat j
   Repeat i

8. *Solve for* Δu_{n+1} *using Gaussian elimination*
   Call $Gauss(\Delta u_{n+1},\ -F,\ J(F), neqns)$

9. *Calculate* $\|\Delta u_{n+1}\|$ *and increment* u_{n+1}
   $\Delta u_{max} = 0$.
   Do $i = 1,\ neqns$
   If $|\Delta u_{i,n+1}| > \Delta u_{max}, \Delta u_{max} = |\Delta u_{i,n+1}|$

$u_{i,n+1}^{(m)} = u_{i,n+1}^{(m-1)} + \delta\Delta u_{i,n+1}$
Repeat i

10. *Test convergence of Newton iterations*
If $\Delta u_{max} < \epsilon$, *go to* 11
Repeat m
Write "Warning: Newton iterations failed to converge at time step $n+1$*"*

11. *Print results:* $n+1,\ t_{n+1},\ (u_{i,n+1},\ i=1,\ neqns)$
Repeat n
End

Function $f(i,u,t)$

Go to $(10,\ 20,\ \ldots),\ i$

10 $f = f_1(u,t)$
Return

20 $f = f_2(u,t)$
Return
.
.
.
End

Subroutine $Jacobn(neqns,\ u,\ t,\ J(f))$

Do $i = 1,\ neqns$
Do $j = 1,\ neqns$
$J(f)_{ij} = \partial f_i/\partial u_j$
Repeat j
Repeat i

Return
End

Note: a Gaussian elimination algorithm such as Algorithm 1.2 is also required.

4.1.3 Runge–Kutta Methods

We observed earlier that Heun's method is actually a second-order Runge–Kutta (R–K) method. Here, we will re-derive Heun's method, via the technique used to derive all explicit R–K methods. The basic idea is to obtain a procedure that agrees with a given Taylor method through a prescribed order by introducing intermediate function evaluations to replace derivative evaluations. To make this clear we need to first briefly consider the Taylor series approach.

As always, we begin with the IVP

$$u' = f(u,t)\ , \quad u(t_0) = u_0\,. \tag{4.30}$$

Suppose $u \in C^{n+1}(t_0,\tau)$, and expand u in a Taylor series:

$$u(t+h) = u(t) + u'h + u''\frac{h^2}{2} + \cdots + u^{(n)}\frac{h^n}{n!} + u^{(n+1)}(\xi)\frac{h^{n+1}}{(n+1)!}\,, \tag{4.31}$$

where $\xi \in [t, t+h]$. Now, at first glance, this does not appear to be very useful since u is the unknown function in (4.30). So in particular, it does not appear that derivatives of u should be known. On the other hand, we have

$$u'(t) = f\,(u(t), t)$$

directly from (4.30). Clearly, this can be evaluated at $t = t_0$; so the first two terms on the right-hand side of (4.31) are known. We should immediately recognize that if we truncate the series after the second term we will have obtained Euler's method, a (locally) second-order R–K scheme.

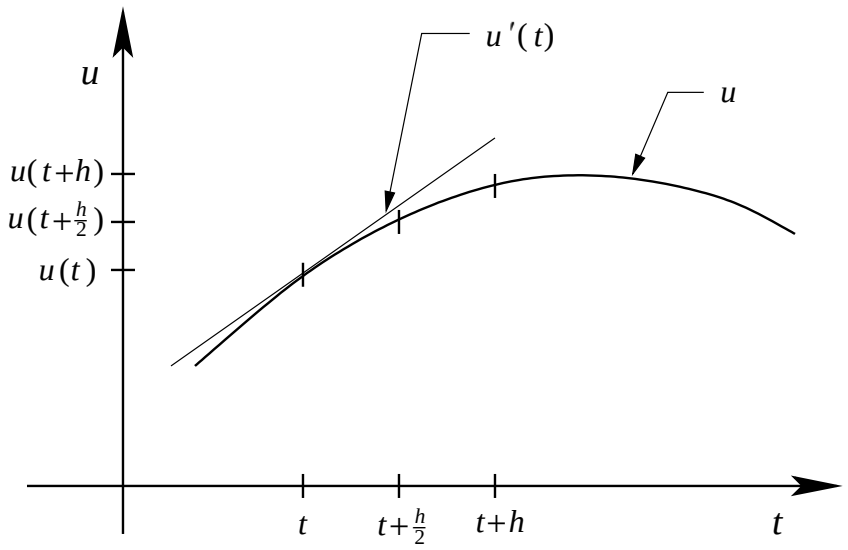


Figure 4.4: Geometry of Runge–Kutta methods

If we differentiate (4.30) with respect to time we obtain

$$u'' = \frac{\partial f}{\partial u}u' + \frac{\partial f}{\partial t} = \frac{\partial f}{\partial u}f + \frac{\partial f}{\partial t}\ . \tag{4.32}$$

Thus, we have produced a means for evaluating yet a third term in (4.31). We could continue in this way for as many terms as desired so long as we are willing to calculate the ever more complicated formulas for higher derivatives. An algorithm based on a Taylor series method is extremely simple once all the derivatives have been derived. But obtaining these in the first place is an arduous task, especially for systems of equations, although we note that software systems such as *Macsyma* and *Maple* can accomplish this quite readily, and even generate source code.

In 1895 Runge conjectured that high-order single-step methods could be obtained without the need to evaluate derivatives, and this idea was later implemented by Kutta. The main notion really involves recognizing that the Taylor series approach is actually an extrapolation procedure, as indicated in Fig. 4.4. In particular, we could calculate $u(t+h)$ either by using $u(t)$, $u'(t)$, $u''(t)$, ..., or by using intermediate (*i.e.*, between t and $t+h$) points to estimate these required derivatives. The latter is the underlying notion employed in the construction of *Runge–Kutta methods*.

We now carry out the formal derivation of Runge–Kutta methods of order two. We first write the Taylor series method of order two in our usual notation:

$$u_{n+1} = u_n + u'_n h + u''_n \frac{h^2}{2}\ . \tag{4.33}$$

From (4.30) $u'_n = f(u_n, t_n)$, and from (4.32)

$$u''_n = \frac{\partial f}{\partial u}(u_n, t_n)\ f(u_n, t_n) + \frac{\partial f}{\partial t}(u_n, t_n)\ .$$

Hence, we can express (4.33) as

$$\begin{aligned} u_{n+1} &= u_n + hf(u_n, t_n) + \frac{h^2}{2}\left[\left(\frac{\partial f}{\partial u}f + \frac{\partial f}{\partial t}\right)(u_n, t_n)\right] \\ &\equiv u_n + h\phi(u_n, t_n, h)\ . \end{aligned} \tag{4.34}$$

(We note in passing that every single-step method can be expressed in the form (4.34).) Our goal now is to construct an approximation to ϕ that differs from ϕ in terms of $\mathcal{O}(h^2)$ and higher, and which does not involve u''.

We begin by expanding f in a formal Taylor series as a function of two independent variables. Thus, we have

$$\begin{aligned} f(u + \Delta u, t + \Delta t) = f(u, t) &+ \frac{\partial f}{\partial u}\Delta u + \frac{\partial f}{\partial t}\Delta t \\ &+ \frac{1}{2}\left[\frac{\partial^2 f}{\partial u^2}(\Delta u)^2 + 2\frac{\partial^2 f}{\partial u\ \partial t}\Delta u\Delta t + \frac{\partial^2 f}{\partial t^2}(\Delta t)^2\right]\ . \end{aligned} \tag{4.35}$$

Now Δt is h in our earlier notation, but for purposes of generality, we set

$$\Delta t = ph$$

since we may want to consider some fraction of a full time step. Furthermore, it is clear from (4.31) that $\Delta u = u'h + \mathcal{O}(h^2)$. But again, for generality, we set

$$\Delta u = qhu' = qhf(u, t)\ .$$

Substitution into (4.35) and introduction of discrete time indexing then results in

$$\begin{aligned} &f(u_n + qhf(u_n, t_n), t_n + ph) = \\ &\qquad f(u_n, t_n) + qhf(u_n, t_n)\frac{\partial f}{\partial u}(u_n, t_n) + ph\frac{\partial f}{\partial t}(u_n, t_n) + \mathcal{O}(h^2)\ . \end{aligned} \tag{4.36}$$

Now consider a function $\phi^*(u_n, t_n, h)$ corresponding to a general single-step method.

$$\phi^*(u_n, t_n, h) = a_1 f(u_n, t_n) + a_2 f(u_n + qhf(u_n, t_n), t_n + ph)\ , \tag{4.37}$$

where $a_1,\ a_2,\ p$ and q are constants to be determined. If we multiply (4.36) by a_2, and substitute into the above, we obtain

$$\begin{aligned} &\phi^*(u_n, t_n, h) = (a_1 + a_2)f(u_n, t_n) \\ &\qquad + h\left[a_2 q f(u_n, t_n)\frac{\partial f}{\partial u}(u_n, t_n) + a_2 p\frac{\partial f}{\partial t}(u_n, t_n)\right] + \mathcal{O}(h^2)\ . \end{aligned} \tag{4.38}$$

We see that this differs from (4.37) by terms of order h^2, and (4.37) depends only on the function f and some as yet undetermined constants. We find these constants by comparing (4.38) and (4.34). Hence,

$$a_1 + a_2 = 1\ , \quad a_2 p = \frac{1}{2}\ , \quad \text{and} \quad a_2 q = \frac{1}{2}\ .$$

From the last two of these we see that $p = q = \frac{1}{2a_2}$, provided $a_2 \neq 0$, and the first relation yields $a_1 = 1 - a_2$. Thus, given $a_2 \neq 0$, we can calculate $p,\ q$ and a_1, and these constants determine a function ϕ^* which differs from ϕ by terms of order h^2, and higher. We see from this that Runge–Kutta methods are not uniquely determined simply by specifying the order. This can be used to great advantage by selecting coefficients that, *e.g.*, minimize truncation error.

There are two choices of a_2 in wide use. The first is $a_2 = \frac{1}{2}$. Then $a_1 = \frac{1}{2}$, and $p = q = 1$; so ϕ^* in (4.37) becomes

$$\phi^*(u_n, t_n, h) = \frac{1}{2} f(u_n, t_n) + \frac{1}{2} f(u_n + hf(u_n + hf(u_n, t_n), t_n + h) \ .$$

Thus (4.34) is

$$u_{n+1} = u_n + \frac{h}{2} \left[f(u_n, t_n) + f(u_n + hf(u_n, t_n), t_n + h) \right] \ , \tag{4.39}$$

which we recognize as Heun's method.

The second choice is $a_2 = 1$. Then $a_1 = 0$ and $p = q = \frac{1}{2}$. This leads to

$$u_{n+1} = u_n + hf \left(u_n + \frac{h}{2} f(u_n, t_n), t_n + \frac{h}{2} \right) , \tag{4.40}$$

which is known variously as modified Euler, modified Euler-Cauchy, or simply the midpoint method.

It is of interest to compare the truncation error for these two methods. From (4.35) we see that the $\mathcal{O}(h^2)$ term for a general R–K method has the form

$$a_2 \frac{h^2}{2} \left[q^2 f(\xi, \eta) \ \frac{\partial^2 f}{\partial u^2}(\xi, \eta) + 2pqf(\xi, \eta) \frac{\partial^2 f}{\partial u \partial t}(\xi, \eta) + p^2 \frac{\partial^2 f}{\partial t^2}(\xi, \eta) \right] \ ,$$

with $\xi,\ \eta\ \in [u, u + \Delta u] \times [t, t + \Delta t]$. For Heun's method $p = q = 1$, and $a_2 = \frac{1}{2}$; so the truncation error term is $\frac{1}{2}$ of the form given above. For the midpoint formulas $p = q = \frac{1}{2}$, and $a_2 = 1$, which leads to a factor of $\frac{1}{4}$. Thus, the leading truncation error of the midpoint rule is only half of that of Heun's method.

We close this section on Runge–Kutta methods by noting that the higher-order R–K schemes are derived by the same procedure as presented here. But this becomes extremely tedious for methods of order higher than two. In general, all Runge–Kutta methods can be written in the form

$$u_{n+1} = u_n + h\phi(u_n, t_n, f, h),$$

where

$$\phi = \sum_{i=1}^{M} w_i k_i \ ,$$

with

$$k_i = f \left(u_n + \sum_{j=1}^{i-1} a_{ij} k_j, t_n + a_i h \right) .$$

For classical fourth-order R–K, $w_1 = w_4 = \frac{1}{6},\ w_2 = w_3 = \frac{1}{3},\ c_1 = 0,\ c_2 = c_3 = \frac{1}{2}$, and $c_4 = 1$. Also, $a_{21} = a_{32} = \frac{1}{2}, a_{43} = 1$ and $a_{31} = a_{41} = a_{42} = 0$. Coefficients for general R–K explicit methods are often summarized in tables of the form

c	A_L
	w^T

where c is a column vector with first component equal to zero, w^T is a row vector, and A_L is a lower triangular matrix. The general representation for a fourth-order R–K method is then

0				
c_2	a_{21}			
c_3	a_{31}	a_{32}		
c_4	a_{41}	a_{42}	a_{43}	
	w_1	w_2	w_3	w_4

Hence, classical fourth-order R–K has the representation

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

Tables of this form can be found for a great variety of R–K methods in Lapidus and Seinfield [19]. They provide a useful way to store R–K coefficients in a general R–K algorithm implementation.

4.1.4 Multi-Step and Predictor-Corrector, Methods

One of the major disadvantages of the R–K methods is that as the order of the method is increased the required number of function evaluations also increases. Hence, if $f(u,t)$ is a complicated vector function, required arithmetic increases rapidly with order of the method. At least a partial remedy to this difficulty can be gained by using the multi-step methods to be discussed in this section. There are many such methods, and many different ways by which they may be derived. Here, we will restrict attention to the class of explicit procedures known as the Adams–Bashforth methods and the related class of implicit schemes known as the Adams–Moulton methods. These are often used in combination to construct predictor-corrector techniques. Our approach to deriving these methods will be similar to that used for the second-order Runge–Kutta method of the previous section, namely via the method of undetermined coefficients. However, we will begin with an outline of the general technique which would be used in a more rigorous construction of these methods.

As the name implies, a multi-step method is advanced in time using information from more than just a single previous step. In general, we can consider k-step methods to solve the by now familiar IVP

$$u' = f(u,t), \quad u(t_0) = u_0 \ .$$

If we assume a constant step size h, then formal integration of this equation between time t_n and t_{n+1} yields

$$\int_{t_n}^{t_{n+1}} u' \ dt = \int_{t_n}^{t_{n+1}} f(u,t) \ dt \ ,$$

or

$$u_{n+1} = u_n + \int_{t_n}^{t_{n+1}} f(u,t)\ dt\ . \tag{4.41}$$

Up to this point everything is exactly the same as in our earlier derivation of the trapezoidal rule. (In fact, the trapezoidal rule is an implicit multi-step method—but a somewhat trivial one.) We can evaluate the integral on the right-hand side by replacing f with an interpolation polynomial spanning the time steps from $n-k+1$ to $n+1$ to obtain a (generally implicit) k-step method. In particular, we obtain from this a quadrature scheme for approximating the integral in (4.41); so from our earlier discussion of quadrature methods we expect (4.41) to take the form

$$u_{n+1} = u_n + h\sum_{i=0}^{k} \beta_i f_{n+1-i}\ . \tag{4.42}$$

It is standard procedure to further generalize this as

$$u_{n+1} = \sum_{i=1}^{k} \alpha_i u_{n+1-i} + h\sum_{i=0}^{k} \beta_i f_{n+1-i}\ , \tag{4.43}$$

but for our purposes here we will always have $\alpha_1 = 1$, and $\alpha_i = 0$, $\forall\ i = 2,\ldots,k$; so (4.42) will be the only form of multi-step methods to be constructed herein. Furthermore, we note that whenever $\beta_0 = 0$, (4.42) is explicit, but otherwise it is implicit and must be solved for u_{n+1} by iteration. We will now derive β_is for both an explicit and an implicit method.

Adams–Bashforth Method

For the explicit method we can write (4.42) as

$$u_{n+1} = u_n + h\sum_{i=1}^{k} \beta_i f_{n+1-i}\ .$$

For simplicity we take $k = 2$, and obtain

$$u_{n+1} = u_n + h(\beta_1 f_n + \beta_2 f_{n-1})\ . \tag{4.44}$$

Since there are actually three coefficients to be determined (including α_1) we would expect to produce a method that is exact for polynomials of degree two, and is thus accurate to third order. Thus, we expand u_n in a Taylor series to third order about u_{n+1}. (Note that we could also expand u_{n+1} about u_n and obtain the same results). This yields

$$u_n = u_{n+1} - u'_{n+1}h + u''_{n+1}\frac{h^2}{2} + \mathcal{O}(h^3)\ .$$

Furthermore, we have

$$u'_n = u'_{n+1} - u''_{n+1}h + u'''_{n+1}\frac{h^2}{2} + \mathcal{O}(h^3) = f_n\ ,$$

and

$$u'_{n-1} = u'_{n+1} - 2u''_{n+1}h + 2u'''_{n+1}h^2 + \mathcal{O}(h^3) = f_{n-1}\ .$$

Substitution of these into (4.44) leads to

$$u_{n+1} = u_{n+1} - u'_{n+1}h + u''_{n+1}\frac{h^2}{2} + h\left[\beta_1(u'_{n+1} - u''_{n+1}h) + \beta_2(u'_{n+1} - 2u''_{n+1}h)\right] + \mathcal{O}(h^3) \ .$$

We now collect terms on the right-hand side according to powers of h. Thus,

$$u_{n+1} = u_{n+1} + (\beta_1 + \beta_2 - 1)hu'_{n+1} + \left(\frac{1}{2} - \beta_1 - 2\beta_2\right)h^2u''_{n+1} \ .$$

Finally, we equate powers of h on the right- and left-hand sides to obtain the two equations for determination of β_1 and β_2:

$$\begin{aligned}\beta_1 + \beta_2 &= 1\\ \beta_1 + 2\beta_2 &= \frac{1}{2} \ .\end{aligned}$$

The solution to this system is $\beta_1 = \frac{3}{2},\ \beta_2 = -\frac{1}{2}$. These are the coefficients of the second-order (global order is order of the method) Adams–Bashforth formula. The higher-order methods can be derived in exactly the same way.

Adams–Moulton Method

We next carry out the same procedure for an implicit k-step Adams–Moulton method, again for $k = 2$. In this case (4.42) is

$$u_{n+1} = u_n + h\left(\beta_0^* f_{n+1} + \beta_1^* f_n + \beta_2^* f_{n-1}\right) \ . \tag{4.45}$$

We see now that for an implicit k-step method there are four (including α_1^*) coefficients to be determined. As a consequence we obtain an increase of one power of h in the formal local accuracy. Thus, the Taylor expansions employed must be carried one term further. We have, for example,

$$u_n = u_{n+1} - u'_{n+1}h + u''_{n+1}\frac{h^2}{2} - u'''_{n+1}\frac{h^3}{6} + \mathcal{O}(h^4) \ .$$

The remaining two expansions given earlier are sufficient. Substitution into (4.45) then yields

$$\begin{aligned}u_{n+1} &= u_{n+1} - u'_{n+1}h + u''_{n+1}\frac{h^2}{2} - u'''_{n+1}\frac{h^3}{6}\\ &\quad + h\left[\beta_0^*u'_{n+1} + \beta_1^*\left(u'_{n+1} - u''_{n+1}h + u'''_{n+1}\frac{h^2}{2}\right) + \beta_2^*\left(u'_{n+1} - 2u''_{n+1}h + 2u'''_{n+1}h^2\right)\right]\\ &= u_{n+1} + \left(\beta_0^* + \beta_1^* - \beta_2^* - 1\right)u'_{n+1}h + \left(\frac{1}{2} - \beta_1^* - 2\beta_2^*\right)u''_{n+1}h^2\\ &\quad + \left(\frac{1}{2}\beta_1^* + 2\beta_2^* - \frac{1}{6}\right)u'''_{n+1}h^3 + \mathcal{O}(h^4) \ .\end{aligned}$$

The system of equations for the β^*s is then

$$\begin{aligned}\beta_0^* + \beta_1^* + \beta_2^* &= 1\\ \beta_1^* + 2\beta_2^* &= \frac{1}{2}\\ \beta_1^* + 4\beta_2^* &= \frac{1}{3} \ ,\end{aligned}$$

and we find

$$\beta_0^* = \frac{5}{12} \ , \quad \beta_1^* = \frac{2}{3} \ , \quad \beta_2^* = -\frac{1}{12} \ .$$

Both Adams–Bashforth (4.44) and Adams–Moulton (4.45) methods can be used separately. But it is typical to use an Adams–Bashforth method as an explicit predictor to begin iterations of an Adams–Moulton method. One can use various combinations of order and k for such a predictor-corrector pair, but it is quite often that a k-step method, with k the same for both predictor and corrector is used. This is because the order of the complete method is always the order of the corrector, provided it is iterated to convergence. Moreover, in any case, each iteration of the corrector results in a gain of one order in h until the order of the corrector is achieved. Since at least one iteration of the corrector will always be used, it is reasonable to use a predictor that is one order less accurate and hence, also a k-step method with k the same as the corrector.

Predictor-Corrector Algorithm

We now provide an algorithm for implementing an arbitrary k-step predictor-corrector pair of Adams–Bashforth/Adams–Moulton type. There are three specific implementational details to be noted. The first occurs in step 2 where we require generation of $k-1$ starting values using a single-step method. The need to do this is one of the major shortcomings of multi-step methods, in general. In practice, if fourth-order Adams multi-step methods are being used, then fourth-order Runge–Kutta is typically employed to compute the starting values. But other approaches are possible.

The second thing to note is that there is really only one term of the implicit method that must be re-evaluated at each iteration. Thus, a significant amount of arithmetic can be saved if at the start of each new time step the remaining terms are calculated and placed in a temporary array to be re-used at each iteration. The elements of this array are denoted $g_i, \ i = 1, \ldots, neqns$ in the algorithm.

Finally, we observe that only a simple fixed-point iteration has been employed for solution of the implicit difference equations, rather than Newton's method, as was used for the trapezoidal rule. This is usual for the Adams–Moulton method because, in general, for methods of order higher than two, the stability properties are not especially good. Thus, the class of problems to which these methods should be applied is somewhat restricted, and for such problems class simple Picard iteration converges very rapidly, usually within a few iterations.

Algorithm 4.2 *(Adams–Bashforth/Adams–Moulton Integrator)*

1. *Read control flags:* $neqns, \ maxitr, \ nsteps, \ k_{AB}, k_{AM}$
   Read numerical parameters: $h, \ \epsilon$
   Read initial data: $(u_{i,0}, i = 1, \ldots, neqns), \ t_0$
   $k = \max(k_{AB}, k_{AM})$

2. *Generate* $k-1$ *starting values for each of the* $i = 1, \ldots,$ $neqns$ *equations using a single-step method; e.g., a* k^{th}*-order Runge–Kutta method.*

3. *Begin time stepping*
   $t_{k-1} = t_0 + (k-1)h$
   Do $n = k-1, \ nsteps$

4. *Evaluate the explicit Adams–Bashforth Predictor Formula*
   Do $i = 1, \ neqns$

$u_{i,n+1} = u_{i,n} + h\sum_{j=1}^{k_{AB}} \beta_i f_{i,n+1-j}$
Repeat i
$t_{n+1} = t_n + h$
If $maxitr < 1$, *go to* 7.

5. *Begin implicit Adams–Moulton Corrector Iterations*
$m = 1$
Do $i = 1,\ neqns$
$g_i = u_n + h\sum_{j=1}^{k_{AM}} \beta_i^* f_{i,n+1-j}$
$u_{i,n+1}^{(m)} = u_{i,n+1}$
Repeat i
Do $m = 1,\ maxitr$
$\Delta u_{max} = 0$
Do $i = 1,\ neqns$
$f_{i,n+1} = f\left(i, u_{i,n+1}^{(m)}, t_{n+1}\right)$
$u_{i,n+1}^{(m+1)} = g_i + h\beta_0^* f_{i,n+1}$
If $\left|u_{i,n+1}^{(m+1)} - u_{i,n+1}^{(m)}\right| > \Delta u_{max}$ *then* $\Delta u_{max} = \left|u_{i,n+1}^{(m+1)} - u_{i,n+1}^{(m)}\right|$
Repeat i

6. *Test convergence of fixed-point iterations*
If $\Delta u_{max} < \epsilon$, *then go to* 7
Repeat m
Print "Warning: iterations failed to converge at $t = t_{n+1}$*; maximum iteration error =* Δu_{max}*"*

7. *Print results for* $t = t_{n+1}$

8. *Shift storage of* f_i*s to begin next time step*
Do $i = 1,\ neqns$
Do $j = k, 0$
$f_{i,n-j} = f_{i,n-j+1}$
Repeat j
Repeat i
Repeat n
End

Function $f(i, u, t)$

Go to $(10, 20, \ldots.),\ i$

10 $f = f_1(u, t)$
Return

20 $f = f_2(u, t)$
Return
.
.
.
End

In closing this section we note that stability analysis for multi-step methods is somewhat more complicated than for single-step methods. In particular, single-step methods can be adequately characterized by their region of absolute stability in the λh-plane. However, for multi-step methods it is not sufficient to consider only absolute stability because the difference equations for multi-step schemes possess multiple solutions. It is always the dominant one that corresponds to the solution of the differential equation; but in long time integrations, unless the difference approximation is such that the "parasitic" solutions remain small compared with the dominant one, the error of the numerical solution will eventually become unacceptable. It is growth of the parasitic solution that makes use of multi-step methods difficult for the class of so-called stiff equations which we treat in the next section.

4.1.5 Solution of Stiff Equations

One of the more difficult classes of problems in numerical IVPs is solution of stiff equations and stiff systems. These problems arise from a variety of physical situations, but probably were first identified in chemical kinetics. They are characterized, intuitively, as having at least two widely disparate time scales. But there is a precise mathematical definition that specifically excludes problems that are actually unstable. Often this distinction is not made; but it is important because stability of the underlying differential problem is necessary to the success of any numerical scheme if long-time integrations are attempted.

The main feature resulting from widely different time scales is that one component of the solution may quite rapidly decay to zero, and be completely unimportant, while other components may be slowly varying. One would expect to be able to employ large time steps, and still be able to predict the slow components with good accuracy. But it turns out that stability is determined by the rapidly changing component(s); so even though they may be too small to be of importance, they may nevertheless impose the requirement of very small time steps to maintain stability.

It should also be noted that stiffness is not always so simple as the above description might imply. In particular, it is possible for all components of a solution to contribute to stiffness, even though the qualitative behavior of none of them indicates rapid decay. An example of this is given by the following problem, which can be found in Gear [10], and elsewhere. Consider the linear system

$$\begin{aligned} u' &= 998u + 1998v & u(0) &= 1 \\ v' &= -999u - 1999v & v(0) &= 0 \ , \end{aligned}$$

with exact solution

$$\begin{aligned} u(t) &= 2e^{-t} - e^{-1000t} \ , \\ v(t) &= -e^{-t} + e^{-1000t} \ . \end{aligned}$$

Notice that both components have terms that very rapidly decay to zero, but the solution itself, changes only slowly, as shown in Fig. 4.5. On the other hand $|u'|$ and $|v'|$ are large and $v' \ll 0$ at early times. This is what causes the stiffness of this system. This example also indicates that, contrary to an often-held belief, stiffness does not necessarily arise as a consequence of nonlinearity; this system is linear.

Formally, for an IVP to be considered *stiff*, it must have the properties given in the following definition.

Definition 4.2 *The ordinary differential equation initial-value problem for $u' = f(u,t)$, $u, f \in \mathbb{R}^p$, is said to be (locally) stiff whenever any eigenvalue λ of $J_u(f)$ has* $\mathrm{Re}\ \lambda \ll 0$.

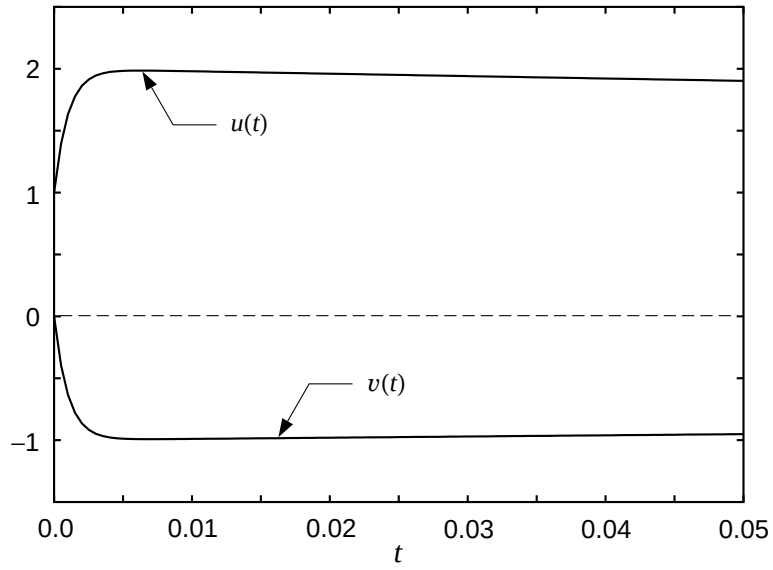


Figure 4.5: Solution of a stiff system

Observe that this implies $u' = \lambda u,\ \lambda < 0$, and $|\lambda| \gg 1$ is, in a sense, stiff; that is, stiffness can occur even for single equations with a single time scale. Recall that this was precisely the problem with which we demonstrated the difficulties encountered with Euler's method, and the remedy of the implicit backward Euler method. It can be shown that the trapezoidal integration exhibits stability properties similar to those of backward Euler, and as we have already seen, it is considerably more accurate. In particular, both of these methods have the property of *A-stability*, defined below, which is theoretically very desirable (but often not very practical) for methods employed for solving stiff equations.

Definition 4.3 *A method is said to be A-stable if all numerical approximations tend to zero as the number of time steps $n \to \infty$ when it is applied to the differential equation $u' = \lambda u$ with a fixed step size $h > 0$, and a (complex) constant λ with* Re $\lambda < 0$.

Because the only restrictions on h and λ are those stated, the implication of the definition is that a method has an absolute stability boundary at $-\infty$ in the left-half complex λh-plane if it is A-stable. The stability diagram for backward Euler is shown in Fig. 4.6, and indicates that this is true. This can be easily derived from the form of the amplification factor of backward Euler given earlier in Eq. (4.22).

Similarly, it can be readily checked that the amplification factor for the trapezoidal rule in this case is

$$\left| \frac{1 + \frac{1}{2}\lambda h}{1 - \frac{1}{2}\lambda h} \right| .$$

For stability we require that this be less than or equal to unity; thus,

$$\left| 1 + \frac{1}{2}\lambda h \right| \leq \left| 1 - \frac{1}{2}\lambda h \right|$$

must hold. Clearly, if $h > 0$ and $\lambda \in \mathbb{R}$ with $\lambda \leq 0$, this always holds. If $\lambda \in \mathbb{C}$ we have $\lambda = \mu + i\nu$,

and we require

$$\left|1 + \frac{1}{2}\mu h + i\frac{1}{2}\nu h\right| \leq \left|1 - \frac{1}{2}\mu h - i\frac{1}{2}\nu h\right|$$

for stability. Calculating the squared modulus on each side yields

$$\left(1 + \frac{1}{2}\mu h\right)^2 + \frac{h^2}{4}\nu^2 \leq \left(1 - \frac{1}{2}\mu h\right)^2 + \frac{h^2}{4}\nu^2 \ .$$

We see from this that the imaginary part will not change the sense of the inequality in this case, so absolute stability holds $\forall\ \lambda$ with Re $\lambda \leq 0$. Hence, trapezoidal integration is A-stable.

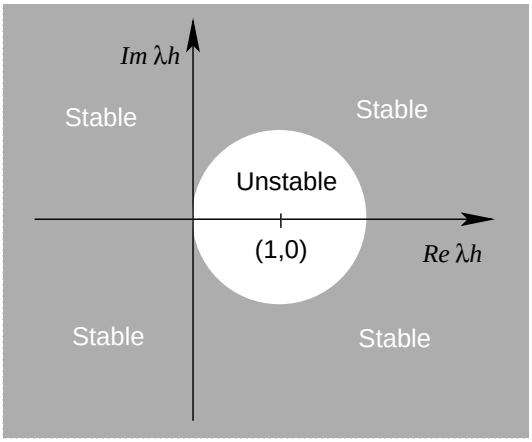


Figure 4.6: Region of absolute stability for backward-Euler method

Of the methods we have discussed in this chapter trapezoidal integration is by far the most effective single method for the solution of stiff IVPs. The individual multi-step methods discussed here are generally not as suitable because of the following theorem due to Dahlquist [3].

Theorem 4.1 *No multi-step method of order greater than two can be A-stable. Moreover, the method of order two having the smallest truncation error is the trapezoidal rule.*

Based on our experience with the explicit Euler method, we would expect that explicit R–K methods also would not be A-stable; however, there are higher-order implicit R–K methods that are A-stable. Unfortunately, these are quite difficult to derive and implement, and they are relatively inefficient.

We close this section by noting that in practice the A-stability requirement is generally more stringent than necessary. Gear [10] presents less restrictive forms of stability, namely *$A(\alpha)$-stability* and *stiff stability*. He has developed a widely-used, very efficient method for solving stiff systems based on the latter of these, using k-step methods with k as large as 6, in which both k and the time step size h are automatically varied from one time step to the next to maintain stability and achieve specified accuracy.

4.2 Boundary-Value Problems for Ordinary Differential Equations

In this section we will consider the solution of boundary-value problems (BVPs) for ODEs. These arise in the analysis of many different problems in engineering and mathematical physics. In the

great majority of cases the problems involve a second-order differential equation with boundary conditions prescribed at each of two ends of a finite interval $[a, b]$. This is often called a *two-point boundary-value problem*. This is the only case we shall treat here; the reader is referred to more advanced treatments, such as Keller [17] for discussion of other types of BVPs.

There are two widely-used general classes of procedures for solving ODE BVPs. These are *i*) shooting, and *ii*) finite-difference methods. We shall discuss the first of these briefly and present the second in considerable detail. We then conclude with a somewhat different approach, the Galerkin procedure, that is more often applied to PDEs but is most easily understood in the ODE context. We note that there are many other somewhat less-often used approaches, treatments of which are omitted here for the sake of brevity.

4.2.1 Mathematical Background

We begin by considering some elementary mathematical background material for ODE BVPs. The most general form of the problem to be considered here is

$$Lu = f(x, u, u'), \qquad x \in [a, b]\,, \tag{4.46}$$

with boundary conditions

$$B_a u(a) = A \tag{4.47a}$$
$$B_b u(b) = B \tag{4.47b}$$

where B_a and B_b are (at most) first-order, but possibly nonlinear, operators. As one might infer from our earlier discussion of coordinate transformations employed with Gauss quadrature, there is no loss in generality in taking $a = 0$ and $b = 1$, and we will sometimes employ this slight simplification. We view f as a generally nonlinear function of u and u', but for the present, we will take $f = f(x)$ only. L is assumed to be a linear second-order operator, so for this case (4.46) is linear and becomes

$$Lu \equiv a_2(x)u'' + a_1(x)u' + a_0(x)u = f(x)\ . \tag{4.48}$$

In our discussions here, the boundary operators in (4.47) will be such as to lead to one of the following three types of conditions, here applied at $x = a$:

$$Bu(a) = u(a) = A\,, \qquad \text{(Dirichlet)} \tag{4.49a}$$
$$Bu(a) = u'(a) = A\,, \qquad \text{(Neumann)} \tag{4.49b}$$
$$Bu(a) = u'(a) + \alpha u(a) = A, \quad \alpha < 0\,. \qquad \text{(Robin)} \tag{4.49c}$$

The same types also apply at $x = b$ where $\alpha > 0$ must hold for the Robin condition. It should be observed that more general boundary conditions sometimes occur, for example, periodicity conditions. The reader is referred to Keller [17] for methods of treatment. Also, we note that the conditions presented in Eqs. (4.49) are often termed "first kind," "second kind" and "third kind," respectively.

A complete boundary-value problem consists of an equation of the form (4.46) (or (4.48)) and a boundary condition of the form of one of (4.49) applied at each end of the interval. We note that a different type of boundary condition may be applied at each end, and that in general the value of A is different at the two ends. For such a problem to have a solution it is generally necessary either that $f(x) \not\equiv 0$ hold, or that $A \neq 0$ at one or both ends of the interval. When $f(x) \equiv 0$, and $A = 0$ at both ends of $[a, b]$ the BVP is said to be homogeneous and will in general have only the trivial solution, $u(x) \equiv 0$.

An exception to this is the *eigenvalue problem* for the differential operator L:

$$Lu = \lambda u\,, \qquad B_0 u = B_1 u = 0\,, \qquad x \in [a, b]\,.$$

For specific values of λ (the eigenvalues) this problem will have nontrivial solutions (the eigenfunctions) analogous to the eigenvalues and eigenvectors of linear algebraic systems studied in Chap. 1. We shall not specifically treat eigenvalue problems for differential operators here except to note that use of finite-difference approximation, as discussed below, reduces this problem to an algebraic eigenvalue problem to which the methods of Chap. 1 may be applied.

4.2.2 Shooting Methods

The earliest numerical procedures applied to solution of two-point boundary-value problems were methods based on initial-value problem techniques, such as discussed earlier. The basic notion is to view (4.48) as an initial-value problem starting at $x = 0$. Since (4.48) is second order, two initial conditions are required. But only one boundary condition can be given at $x = 0$. Suppose it is the Dirichlet condition $u(0) = A$. We must now guess a value for $u'(0)$ in order to start the initial-value problem. Take $u'(0) = \beta^{(1)}$, and integrate (4.48) from $x = 0$ to $x = 1$ using any stable initial-value method. The result might appear as in Fig. 4.7. In particular, the boundary condition, say $u(1) = B$, will not have been satisfied. So we select a second value of $u'(0)$, say $\beta^{(2)}$, and "shoot" again. The original curve suggests that we should take $\beta^{(2)} > \beta^{(1)}$; the result might be as shown.

This approach can be formalized as a Newton iteration by observing that the value of $u(1)$ that is actually obtained is an implicit function of β. In particular, we can employ Newton's method to solve the equation

$$F(\beta) = u(1, \beta) - B = 0\,,$$

for β. For linear equations this turns to out to work fairly well because an auxiliary equation can be derived in such a way that exactly two iterations are always required (see [17]). However, the situation is not nearly so favorable for nonlinear problems.

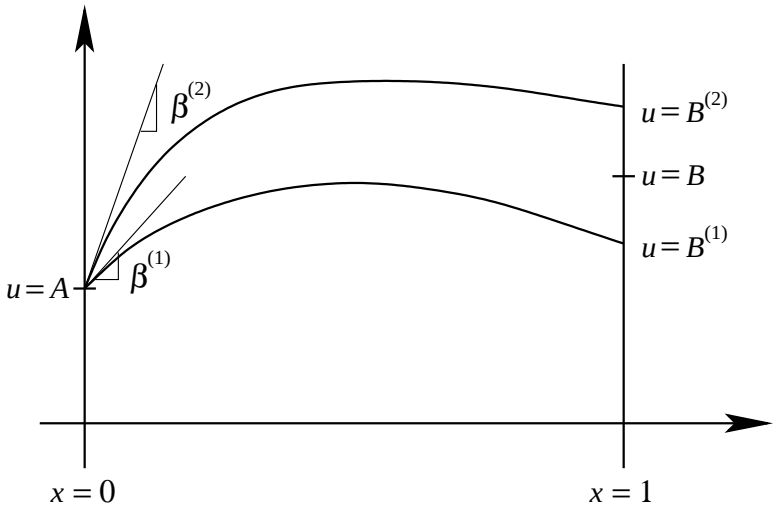


Figure 4.7: Geometric representation of the shooting method

In general, shooting is felt to have the advantage of high-order accuracy available via the well-developed initial-value methods. But it has the disadvantage of being iterative, thus requiring implementation of some algebraic equation solving method in addition to the IVP method. In addition, the method may completely fail for problems whose solutions exhibit steep gradients anywhere in the domain.

4.2.3 Finite-Difference Methods

For the above reasons, finite-difference methods have become more widely used in recent years. It was at one time felt that the standard centered-difference approximations to be employed here were often not sufficiently accurate. However, when these are used in conjunction with Richardson extrapolation, the resulting solutions are accurate to fourth order, which is equivalent to the accuracy attained with usual implementations of the shooting method (typically using a Runge–Kutta initial-value solver). Furthermore, the finite-difference methods still work satisfactorily in regions of large gradients, at least if local mesh refinement is employed.

The basic idea in applying the finite-difference method is extremely simple. We merely replace all of the differential operators in (4.48) and (4.49) with difference operators as treated in Chap. 3. This results in a system of algebraic equations which, as we will see below, is banded and sparse (in fact, tridiagonal). Thus, it can be very efficiently solved by the methods of Chap. 1.

Discretization of the Differential Equation

We begin by introducing a partition of the interval $[0,1]$ consisting of N points: $\{x_i\}_{i=1}^N$, such that $0 = x_1 < x_2 < \cdots < x_{N-1} < x_N = 1$. Furthermore, we shall always employ a uniform spacing for the computational mesh. When local mesh refinement is needed, it is best accomplished via a coordinate-stretching transformation prior to discretization of the domain and equations. However, we shall not treat this matter here. The interested reader is referred to Thompson *et al.* [35]. Figure 4.8 depicts the computational mesh. The solution to (4.48) will be approximated at each of the N grid points, and the ordered collection of these grid point values, $\{u_i\}_{i=1}^N$, constitutes the *grid function*. Finally, we observe that for the general interval $[a,b]$, we would have $h = (b-a)/(N-1)$. We will later see that grid points $i = 0$ and $i = N+1$ are needed for some types of boundary

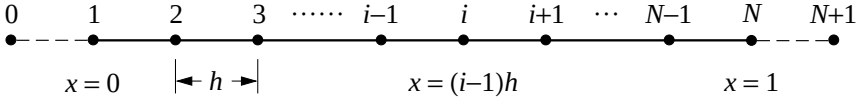


Figure 4.8: Finite-Difference grid for the interval $[0,1]$

condition approximation.

As mentioned at the beginning of this section, we will use only second-order centered approximations in the present treatment. Thus, at the i^{th} grid point (4.48) can be approximated as

$$L_h u_i = a_{2,i} D_0^2(h) u_i + a_{1,i} D_0(h) u_i + a_{0,i} u_i = f_i \ . \tag{4.50}$$

One expects, correctly, that this approximation exhibits a second-order truncation error; moreover, the expansion for the truncation error contains only even powers of h. To demonstrate this, we first carry out the indicated differencing. We have

$$a_{2,i}\left(\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}\right) + a_{1,i}\left(\frac{u_{i+1} - u_{i-1}}{2h}\right) + a_{0,i} u_i = f_i \ .$$

Multiplication by h^2, which is standard, yields

$$a_{2,i}\,(u_{i-1} - 2u_i + u_{i+1}) + \frac{h}{2}a_{1,i}\,(u_{i+1} - u_{i-1}) + a_{0,i}h^2u_i = h^2f_i\ ,$$

and regrouping of terms according to grid function index leads to

$$\left(a_{2,i} - a_{1,i}\frac{h}{2}\right)u_{i-1} + \left(a_{0,i}h^2 - 2a_{2,i}\right)u_i + \left(a_{2,i} + a_{1,i}\frac{h}{2}\right)u_{i+1} = h^2f_i\ . \tag{4.51}$$

In general, this difference equation holds for all interior grid points on $(0,1)$; *i.e.*, for $i = 2,\ldots,N-1$. Some modifications, depending on boundary condition type, will be needed at the two boundary points. We treat this later.

For notational convenience, we now define

$$\begin{aligned} C_{1,i} &\equiv a_{2,i} - a_{1,i}\frac{h}{2}\ , \\ C_{2,i} &\equiv a_{0,i}h^2 - 2a_{2,i}\ , \\ C_{3,i} &\equiv a_{2,i} + a_{1,i}\frac{h}{2}\ . \end{aligned}$$

Then (4.51) can be written as

$$C_{1,i}u_{i-1} + C_{2,i}u_i + C_{3,i}u_{i+1} = h^2f_i\ . \tag{4.52}$$

This form strongly suggests the tridiagonal matrix structure alluded to earlier. But it will be necessary to show that introduction of boundary conditions does not alter this structure.

Truncation Error Analysis

We will next derive the truncation error for the interior grid point approximations (4.52). As the reader should expect by now, the approach makes use of Taylor expansions, and as a consequence, a certain degree of smoothness will be required of $u(x)$ in order for the truncation error estimates to be valid. For u_{i-1} and u_{i+1} expanded about the arbitrary grid point x_i we have

$$u_{i-1} = u_i - u_i'h + u_i''\frac{h^2}{2} - u_i'''\frac{h^3}{6} + u_i''''\frac{h^4}{24} - \cdots\ ,$$

and

$$u_{i+1} = u_i + u_i'h + u_i''\frac{h^2}{2} + u_i'''\frac{h^3}{6} + u_i''''\frac{h^4}{24} + \cdots\ .$$

Substitution of these into (4.52), and rearrangement by grouping terms containing like powers of h, leads to

$$\begin{aligned} (C_{1,i} + C_{2,i} + C_{3,i})u_i + (C_{3,i} - C_{1,i})hu_i' + (C_{1,i} + C_{3,i})\frac{h^2}{2}u_i'' \\ + (C_{3,i} - C_{1,i})\frac{h^3}{6}u_i''' + (C_{1,i} + C_{3,i})\frac{h^4}{24}u_i'''' + \cdots = h^2f_i\ . \end{aligned}$$

Now from the definitions of the $C_{k,i}$s, $k = 1,2,3$, we have

$$\begin{aligned} C_{1,i} + C_{2,i} + C_{3,i} &= a_{0,i}h^2\,, \\ C_{3,i} - C_{1,i} &= a_{1,i}h\,, \\ C_{1,i} + C_{3,i} &= 2a_{2,i}\,. \end{aligned}$$

Thus, the above becomes

$$a_{0,i}h^2u_i + a_{1,i}h^2u_i' + a_{2,i}h^2u_i'' + a_{1,i}\frac{h^4}{6}u_i''' + a_{2,i}\frac{h^4}{12}u_i'''' \cdots = h^2 f_i \,.$$

We now divide by h^2, and rearrange terms to find

$$a_{2,i}u_i'' + a_{1,i}u_i' + a_{0,i}u_i - f_i = -a_{1,i}\frac{h^2}{6}u_i''' - a_{2,i}\frac{h^2}{12}u_i'''' - \cdots \,.$$

The left-hand side is just the original differential equation written at an arbitrary grid point i, while the right-hand side is an $\mathcal{O}(h^2)$ error term, provided $u \in C^4(0,1)$. Hence, as we expected, the differential equation and difference approximation differ by terms that are of $\mathcal{O}(h^2)$. Thus, the method is second-order accurate. We note that for boundary-value problems there is no distinction between local and global approximation because error does not accumulate in moving from one grid point to the next as it does for initial-value methods. On the other hand, incorrect discretization and/or implementation of boundary conditions at any particular grid points can degrade the order of accuracy throughout.

Boundary Condition Implementation

We will now construct second-order accurate boundary approximations. When only Dirichlet conditions are imposed, there is no approximation needed. However, there is an important point to make for this case; namely, the differential operator (*i.e.*, the equation) does not hold at a boundary at which a Dirichlet condition is applied. This is because we have already prescribed the solution at this point by assigning the value given in the boundary condition. Thus, in place of (4.52) we would merely have the equation

$$C_{2,1}u_1 = A \tag{4.53}$$

with $C_{2,1} = 1$ at, for example, the first grid point. In particular, then, we take $C_{1,1} = C_{3,1} = 0$. Clearly, we would not even need to solve the equation at this point, except for the fact that it is coupled with the second equation. If we wish to carry out the algebra ahead of time we can reduce the number of equations in the system by one for each Dirichlet condition, but it is easier to write general codes by using the approach given here.

We next show how to implement Neumann conditions. In this case, since the derivative of the solution, but not the solution itself, is prescribed at the boundary, the differential equation still holds even at the boundary. Thus, we have two relations that must be satisfied at the same point. At first, this might appear to present a dilemma; but, in fact, for the approximations employed here, it is quite fortuitous. We begin by approximating the Neumann condition (4.49b) at $i = 1$ by the centered difference

$$\frac{u_2 - u_0}{2h} = A \,. \tag{4.54}$$

Now observe that the grid function value u_0 is not defined on our original grid (Fig. 4.8), and the point corresponding to $i = 0$ is termed an *image point*. In order to formally solve for u_0 we need an additional relation, and this is provided by the difference approximation to the differential equation written at the boundary point $i = 1$. In particular, (4.52) is

$$C_{1,1}u_0 + C_{2,1}u_1 + C_{3,1}u_2 = h^2 f_1 \,,$$

which also contains the image point value u_0. We now solve (4.54) for u_0 and substitute the result into the above. This yields

$$C_{1,1}(u_2 - 2hA) + C_{2,1}u_1 + C_{3,1}u_2 = h^2 f_1 \,,$$

or

$$C_{2,1}u_1 + (C_{1,1} + C_{3,1})u_2 = h^2 f_1 + 2hAC_{1,1} \ . \tag{4.55}$$

The main observation to be made here is that the tridiagonal matrix structure is now maintained, and at the same time second-order accuracy will be achieved. It is left to the reader to derive a similar result at $i = N$, where an image point corresponding to $i = N + 1$ occurs. Finally we note that the same image points occur for Robin boundary conditions, and that they may be treated in precisely the same manner as shown here for the Neumann condition. We leave demonstration of this as an exercise for the reader.

Complete BVP Discretization

We summarize the treatment given here by presenting the matrix structure and pseudo-language algorithm for solving the following problem.

$$\begin{aligned} a_2(x)u'' + a_1(x)u' + a_0(x)u &= f(x), \qquad x \in [0,1)\,, \\ u'(0) &= A\,, \\ u(1) &= B\,. \end{aligned} \tag{4.56}$$

The first equation of the system will be (4.55). Then the next $N-2$ equations will all be of the form (4.52), and the last equation will be of the form (4.53) but written for the grid point x_N. The matrix form of this is shown in Fig. 4.9. The tridiagonal structure is quite evident from this

$$\begin{bmatrix} C_{2,1} & C_{1,1}+C_{3,1} & & & & & & & \\ C_{1,2} & C_{2,2} & C_{3,2} & & & & \bigcirc & & \\ & C_{1,3} & C_{2,3} & C_{3,3} & & & & & \\ & & \ddots & \ddots & \ddots & & & & \\ & & & C_{1,i} & C_{2,i} & C_{3,i} & & & \\ & & & & \ddots & \ddots & \ddots & & \\ & \bigcirc & & & & C_{1,N-1} & C_{2,N-1} & C_{3,N-1} \\ & & & & & & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_i \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix} = h^2 \begin{bmatrix} f_1 + \frac{2}{h}AC_{1,1} \\ f_2 \\ f_3 \\ \vdots \\ f_i \\ \vdots \\ f_{N-1} \\ B/h^2 \end{bmatrix}$$

Figure 4.9: Matrix structure of discrete equations approximating (4.56)

figure. Clearly, the system can be efficiently solved for the grid function values $\{u_i\}_{i=1}^N$ using the tridiagonal LU-decomposition algorithm of Chap. 1. A pseudo-language algorithm for implementing this approach follows.

Algorithm 4.3 *(Finite-Difference Solution of Linear BVPs)*

1. *Enter problem data: Number of grid points, N; endpoints of solution domain, a, b; boundary condition type flags, ibca, ibcb; boundary values A, B.*

2. *Calculate discretization step size: $h = (b-a)/(N-1)$.*

3. *Load matrix coefficients for all grid points*
   Do $i = 1,\ N$

$C_{1,i} = a_{2,i} - a_{1,i}\frac{h}{2}$
$C_{2,i} = a_{0,i}h^2 - 2a_{2,i}$
$C_{3,i} = a_{2,i} + a_{1,i}\frac{h}{2}$
$b_i = h^2 f_i$

Repeat i

4. *Modify end point coefficients to account for b.c.s*
If ibca $= 1$ *(Dirichlet b.c.), then*

$C_{1,1} = C_{3,1} = 0$
$C_{2,1} = 1$
$b_1 = A$

Else if ibca $\neq 1$ *(Neunmann or Robin b.c.), then*

$C_{3,1} = C_{3,1} + C_{1,1}$
$b_1 = b_1 + 2hAC_{1,1}$
If ibca $= 3$ *(Robin b.c.), then* $C_{2,1} = C_{2,1} + 2h\alpha C_{1,1}$
$C_{1,1} = 0$

End if
If ibcb $= 1$*, then*

$C_{1,N} = C_{3,N} = 0$
$C_{2,N} = 1$
$b_N = B$

Else if ibcb $\neq 1$*, then*

$C_{1,N} = C_{1,N} + C_{3,N}$
$b_N = b_N - 2hBC_{3,N}$
If ibcb $= 3$*, then* $C_{2,N} = C_{2,N} - 2h\alpha C_{3,N}$
$C_{3,N} = 0$

End if

5. *Call Tridiag(C,b,N)*

6. *Do* $i = 1,\ N$

$u_i = b_i$

Repeat i

4.3 Singular and Nonlinear Boundary-Value Problems

In this section we will consider several special topics, any or all of which can be of great utility in applications. We begin with a brief discussion of coordinate singularities, as arise from use of polar coordinate systems. This is followed by a basic treatment of nonlinear equations. Four approaches are presented: *i*) Picard iteration, *ii*) Newton's method, *iii*) quasilinearization, and *iv*) Galerkin procedures.

4.3.1 Coordinate Singularities

Correct treatment of coordinate singularities is important for problems posed in polar geometries; consider the following model problem:

$$\begin{aligned} u'' + \frac{1}{r}u' - u &= f(r) \ , \quad r \in [0, R) \ , \\ u'(0) &= 0 \ , \\ u(R) &= 1 \ . \end{aligned}$$

We begin by noting that this is in the same general form as the boundary-value problems already treated, and thus, away from singularities approximations will be the same as those of the preceding section; but clearly, when $r = 0$, the coefficient of the first-derivative term is singular. We might first consider multiplying through by r, but this simply yields the boundary condition at $r = 0$, and we have no guarantee that the differential equation, itself, will be satisfied. In this particular situation, it is typical that the condition $u'(0) = 0$ results from geometric symmetries, and is not even an actual boundary condition since $r = 0$ is, in fact, not really a boundary point in such a case. Thus, it is crucial to guarantee that the differential equation also be satisfied.

This requirement can be met by considering the limit of the differential equation as $r \to 0$. We have

$$\lim_{r\to 0}\left[u'' + \frac{1}{r}u' - u\right] = \lim_{r\to 0} f(r) \ .$$

If we assume $u \in C^2(0, R)$, and $f(r)$ is bounded on $[0, R]$, only the second term on the left-hand side presents any difficulties. In fact, we have

$$u''(0) + \lim_{r\to 0}\frac{1}{r}u' - u(0) = f(0) \ .$$

Since $u'(0) = 0$, we can apply L'Hospital's rule to the remaining limit:

$$\lim_{r\to 0}\frac{1}{r}u' = \lim_{r\to 0}\frac{u''}{1} = u''(0) \ .$$

Thus, at $r = 0$, the differential equation becomes

$$2u'' - u = f \ ,$$

which no longer contains a singularity.

On a discrete grid this is

$$2u_1'' - u_1 = f_1 \ ,$$

and the difference approximation is

$$2\left(\frac{u_0 - 2u_1 + u_2}{h^2}\right) - u_1 = f_1 \ .$$

We see, once again, that an image point appears; but we still must satisfy the boundary (symmetry) condition $u'(0) = 0$, which in discrete form is

$$\frac{u_2 - u_0}{2h} = 0 \quad \Rightarrow \quad u_0 = u_2 \ .$$

Thus, the difference equation at $r = 0$ $(i = 1)$ is

$$-(h^2 + 4)u_1 + 4u_2 = h^2 f_1 \ .$$

Hence, the tridiagonal form will still be preserved while satisfying both the differential equation and symmetry condition to second order.

We should remark here that although the above development is straightforward and mathematically consistent, other approaches have been widely used; these often yield reasonable results, but can sometimes completely fail unexpectedly—hence, we do not recommend them. One such approach is to implement only the Neumann boundary condition, and ignore the differential equation, at the point of singularity, as mentioned earlier. Now suppose in our preceding example we imposed the condition $u(R) = 0$, and in addition defined $f(r)$ as

$$f(r) = \begin{cases} 1 & r = 0\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

This would correspond to a steady-state heat conduction problem in a cylinder of radius R with heating due to a line source on the axis. If we were to implement only the Neumann condition at $r = 0$, thus avoiding the singularity, we would no longer have any heat input, and the solution would be the trivial one, $u \equiv 0$. The only approach that can consistently circumvent such difficulties is the one utilizing L'Hospital's rule for removal of the coordinate singularity, as we have described above.

4.3.2 Iterative Methods for Nonlinear BVPs

Our next topic is solution of nonlinear two-point boundary-value problems. Recall that our original equation (4.46) was formally nonlinear. We repeat it here:

$$Lu = f(x, u, u')\ , \tag{4.57}$$

where, again, L is a linear operator of order two. We will begin by considering three methods for solving (4.57) in the context of finite-difference approximation: *i*) Picard iteration, *ii*) Newton iteration, and *iii*) quasilinearization. We then conclude our discussions of ODE BVPs in the next section by presenting a Galerkin approximation for a specific case of (4.57).

Picard Iteration

If we replace L with L_h, where from earlier notation

$$L_h \equiv a_{2,i} D_0^2(h) + a_{1,i} D_0(h) + a_{0,i}\ ,$$

then, as we have already seen, the left-hand side of the resulting system of discrete equations is just a matrix times a vector, as shown in detail following Eq. (4.56). Thus, we can write the formal solution representation of (4.57) as

$$u = L_h^{-1} f(x, u, u')\ ,$$

which immediately suggests the fixed-point iteration scheme

$$u^{(m+1)} = L_h^{-1} f\left(x, u^{(m)}, u'^{(m)}\right)\ , \tag{4.58}$$

where we are now viewing u and f to be grid functions.

We know that a sufficient condition for convergence of this iteration procedure is that the Lipschitz constant be less than unity. For many mild nonlinearities in f, this can be proven to hold. In such cases, the solution of (4.57) is straightforward. We simply discretize the equation as

$$L_h u_i = f\left(x_i, u_i, \frac{u_{i+1} - u_{i-1}}{2h}\right) , \qquad \forall\ i = 1, 2, \ldots, N , \tag{4.59}$$

up to boundary condition treatment at $i = 1$ and $i = N$, as discussed previously for linear problems. We then guess a solution to be substituted into the right-hand side, and the resulting difference equation can be solved as already discussed. This is repeated iteratively as indicated by (4.58), until the desired degree of convergence is achieved. Unfortunately, this simple procedure does not always succeed, and even when it does, convergence may be rather slow; so other approaches are often used.

Newton's Method

A remedy to these difficulties is use of Newton's method. We begin by writing (4.59) as

$$F_i(u_{i-1}, u_i, u_{i+1}) = L_h u_i - f\left(x_i, u_i, \frac{u_{i+1} - u_{i-1}}{2h}\right) = 0 . \tag{4.60}$$

As the notation implies, each of the F_is depends on only three grid function values because of our use of second-order centered differences. It then follows that $J(F)$, the Jacobian matrix of F, is tridiagonal; so the Gaussian elimination procedure usually employed for Newton's method should be replaced with the tridiagonal LU-decomposition algorithm of Chap. 1. We will demonstrate these ideas with the following example.

Consider the problem

$$\begin{gathered} u'' = S(x) - (u')^2 - \sin u \equiv f(x, u, u') , \quad x \in (0, 1) , \\ u(0) = u(1) = 0 . \end{gathered} \tag{4.61}$$

Here, $S(x)$ is a known forcing function. The differential equation then has $L = d^2/dx^2$, and the second-order accurate centered-difference approximation is

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = S_i - \left(\frac{u_{i+1} - u_{i-1}}{2h}\right)^2 - \sin u_i .$$

As we have indicated earlier in the linear case, it is preferable to multiply through by h^2, so this becomes

$$u_{i-1} - 2u_i + u_{i+1} = h^2 S_i - \frac{1}{4}(u_{i+1} - u_{i-1})^2 - h^2 \sin u_i .$$

Then for each $i = 2, \ldots, N - 1$, we have

$$F_i = u_{i-1} - 2u_i + u_{i+1} + \frac{1}{4}(u_{i+1} - u_{i-1})^2 + h^2 \sin u_i - h^2 S_i = 0 , \tag{4.62}$$

and for $i = 1$ and $i = N$

$$F_i = u_i = 0 . \tag{4.63}$$

The solution procedure for the nonlinear BVP (4.61) now consists of solving the system of nonlinear algebraic equations, (4.62), (4.63) via Newton's method. We compute the elements of the Jacobian matrix of F in the usual way. For general i, we have

$$\begin{aligned}
&\frac{\partial F_i}{\partial u_1} = 0 \ , \quad \frac{\partial F_i}{\partial u_2} = 0 \ , \quad \dots \ , \quad \frac{\partial F_i}{\partial u_{i-2}} = 0 \ , \\
&\frac{\partial F_i}{\partial u_{i-1}} = 1 - \frac{1}{2}(u_{i+1} - u_{i-1}) \ , \\
&\frac{\partial F_i}{\partial u_i} = -2 + h^2 \cos u_i, \\
&\frac{\partial F_i}{\partial u_{i+1}} = 1 + \frac{1}{2}(u_{i+1} - u_{i-1}) \ , \\
&\frac{\partial F_i}{\partial u_{i+2}} = 0 \ , \quad \dots \ , \quad \frac{\partial F_i}{\partial u_N} = 0 \ , \quad i = 2, \dots, N-1 \ .
\end{aligned}$$

For $i = 1$, $\partial F_1/\partial u_1 = 1$, and all other partial derivatives are zero; similarly, $\partial F_N/\partial u_N = 1$, with all other partial derivatives being zero. It is clear that the Jacobian matrix is tridiagonal as we noted earlier. Thus, Newton's method can be implemented in a very efficient manner.

Quasilinearization

Quasilinearization is generally equivalent to Newton's method, except in the specifics of its implementation. In Newton's method, as just presented, we discretize the nonlinear equation and then locally linearize the nonlinear (algebraic) difference equations at each iteration. By way of contrast, in *quasilinearization* (often called the Newton–Kantorovich procedure) we first linearize the nonlinear operator(s), and then discretize the resulting linear equation(s). Iterations are then performed in a manner similar to that of Picard iteration, but of course, using a different iteration function that results in quadratic convergence just as for Newton's method.

Recall that our prototypical nonlinear equation is (4.57):

$$Lu = f(x, u, u') \ ,$$

where f is generally nonlinear in u and u'. We view u and u' as being distinct independent functions and expand f in a Taylor series, called a Fréchet–Taylor series, in terms of these:

$$f(x, u, u') = f\left(x, u^{(0)}, u'^{(0)}\right) + \left(\frac{\partial f}{\partial u}\right)^{(0)} \left(u - u^{(0)}\right) + \left(\frac{\partial f}{\partial u'}\right)^{(0)} \left(u' - u'^{(0)}\right) + \cdots \ ,$$

where $u^{(0)}$ and $u'^{(0)}$ are initial estimates of u and u', respectively, which are typically updated after each iteration. Equation (4.57) now becomes

$$\left[L - \left(\frac{\partial f}{\partial u'}\right)^{(0)} \frac{d}{dx} - \left(\frac{\partial f}{\partial u}\right)^{(0)}\right] u = f\left(x, u^{(0)}, u'^{(0)}\right) - \left(\frac{\partial f}{\partial u}\right)^{(0)} u^{(0)} - \left(\frac{\partial f}{\partial u'}\right)^{(0)} u'^{(0)} \ . \qquad (4.64)$$

It is clear that this equation is linear in u, and that it can be discretized in the usual manner. In particular, the left-hand side can still be cast in the form of (4.52). To see this we write the left-hand side in detail:

$$a_2 u'' + a_1 u' + a_0 u - \left(\frac{\partial f}{\partial u'}\right)^{(0)} u' - \left(\frac{\partial f}{\partial u}\right)^{(0)} u = a_2 u'' + \left[a_1 - \left(\frac{\partial f}{\partial u'}\right)^{(0)}\right] u' + \left[a_0 - \left(\frac{\partial f}{\partial u}\right)^{(0)}\right] u \ .$$

Thus, for the general m^{th} iteration if we define

$$\tilde{a}_0(x) \equiv a_0(x) - \left(\frac{\partial f}{\partial u}\right)^{(m)}$$
$$\tilde{a}_1(x) \equiv a_1(x) - \left(\frac{\partial f}{\partial u'}\right)^{(m)}$$
$$\tilde{a}_2(x) \equiv a_2(x)$$
$$\tilde{f}(x) \equiv f\left(x, u^{(m)}, u'^{(m)}\right) - \left(\frac{\partial f}{\partial u}\right)^{(m)} u^{(m)} - \left(\frac{\partial f}{\partial u'}\right)^{(m)} u'^{(m)} ,$$

we obtain a linear equation of precisely the same form as treated earlier; namely

$$\tilde{a}_2(x)u'' + \tilde{a}_1(x)u' + \tilde{a}_0(x)u = \tilde{f}(x) . \tag{4.65}$$

As a consequence of the preceding definitions, the difference equation coefficients at the m^{th} iteration take the form

$$C_{1,i}^{(m)} = a_{2,i} - \left[a_{1,i} - \left(\frac{\partial f}{\partial u'}\right)_i^{(m)}\right]\frac{h}{2}$$
$$C_{2,i}^{(m)} = \left[a_{0,i} - \left(\frac{\partial f}{\partial u}\right)_i^{(m)}\right]h^2 - 2a_{2,i}$$
$$C_{3,i}^{(m)} = a_{2,i} + \left[a_{1,i} - \left(\frac{\partial f}{\partial u'}\right)_i^{(m)}\right]\frac{h}{2}$$
$$\tilde{f}_i^{(m)} = f\left(x_i, u_i^{(m)}, D_0 u_i^{(m)}\right) - \left(\frac{\partial f}{\partial u}\right)_i^{(m)} u_i^{(m)} - \left(\frac{\partial f}{\partial u'}\right)_i^{(m)} D_0 u_i^{(m)} .$$

In our usual notation for iteration procedures, the discrete form of (4.65) now can be expressed as

$$C_{1,i}^{(m)} u_{i-1}^{(m+1)} + C_{2,i}^{(m)} u_i^{(m+1)} + C_{3,i}^{(m)} u_{i+1}^{(m+1)} = h^2 \tilde{f}_i^{(m)} ,$$

at the general i^{th} interior grid point. Clearly, boundary condition implementation is done in the same way as demonstrated earlier for linear equations. Furthermore, we note that nonlinear boundary conditions can be linearized via quasilinearization in a manner analogous to the treatment presented here for the differential equation. The algorithm required to implement quasilinearization for solution of (4.57) is the following.

Algorithm 4.4 *(Quasilinearization Solution of Nonlinear ODE Two-Point BVPs)*

1. *Input number of grid points N, maxitr, ϵ, boundary points a, b, boundary condition flags and values, initial guess $\left\{u_i^{(0)}\right\}_{i=1}^{N}$; set iteration counter, $m = 0$; calculate $h = (b-a)/(N-1)$*
2. *Form difference approximations of $u'^{(m)}$, if needed*
3. *Evaluate the nonlinear function f_i, and its Fréchet derivatives at $u_i^{(m)}$, $u_i'^{(m)}$ $\forall\ i = 1, \ldots, N$.*
4. *Calculate $C_{1,i}^{(m)}$, $C_{2,i}^{(m)}$, $C_{3,i}^{(m)}$, $\tilde{f}_i^{(m)}$, $i = 1, 2, \ldots, N$, and store in banded matrix*
5. *Implement boundary conditions as in step 4 of Algorithm 4.3*

6. *Call tridiagonal LU-decomposition to calculate* $\left\{u_i^{(m+1)}\right\}_{i=1}^N$

7. *Test convergence:*
   If $\left\|u^{(m+1)} - u^{(m)}\right\| < \epsilon$, *then stop*
   else if $m < maxitr$, *then* $m = m + 1$
   go to 2.
   else print error message and stop

We note in closing this section that this function space version of Newton's method exhibits the same quadratic convergence as does the usual Newton's method on $\mathbb{R}^N$.

4.3.3 The Galerkin Procedure

The Galerkin procedure possesses features that are fundamentally different from finite differencing. In finite-difference methods, all approximations are local, extending over only a few grid points, and their derivations are based on Taylor series expansion, thus implying considerable smoothness requirements on the quantities being approximated. In the Galerkin procedure we begin by assuming a global functional representation of the solution. For reasons that will become more clear as we proceed, this global representation should be in the form of a (generalized) Fourier series, say

$$u(x) = \sum_{k=1}^{\infty} a_k \phi_k(x) \ , \quad x \in [a, b] \ . \tag{4.66}$$

The functions $\{\phi_k\}_{k=1}^{\infty}$ are called the *basis functions* of the Galerkin approximation. They are usually chosen to automatically satisfy any boundary conditions that are to be satisfied by $u(x)$, although this is not strictly necessary and certainly not always possible.

A more important, and in fact, necessary, requirement is that the set of functions $\{\phi_k(x)\}_{k=1}^{\infty}$ be *complete* in $L^2[a, b]$. In general, completeness is not an easy property to check. However, it is known that the set of eigenfunctions of any regular (or even singular) Sturm–Liouville problem is complete (see, *e.g.*, Gustafson [11]), and this supplies us with a quite rich assortment of possible basis sets. The importance of completeness cannot be overstated, for completeness of a given basis set in $L^2[a, b]$ implies that any function in $L^2[a, b]$ can be represented by a series of the form (4.66). Hence if the solution to the BVP under consideration is in L^2 we are guaranteed that Fourier coefficients $\{a_k\}_{k=1}^{\infty}$ exist, and that the resulting series (4.66) is convergent. Without completeness of the set $\{\phi_k\}$, we have no such guarantee.

We observe that there is one other property of $\{\phi_k\}$ that is desirable. It is *orthonormality*. Clearly, if the set $\{\phi_k\}$ is complete it can be made orthonormal via the *Gram–Schmidt procedure* (see, *e.g.*, Stakgold [29]) followed by normalization. Orthonormality implies the following relationship amongst the ϕ_ks:

$$\langle \phi_i, \phi_j \rangle = \delta_{ij} \ , \quad \forall \ i, j = 1, \ldots, \infty \ , \tag{4.67}$$

where $\langle \phi_i, \phi_j \rangle = \int \phi_i \phi_j$, and δ_{ij} is the Kronecker δ. Although this is not an absolute requirement, it is a useful property in terms of implementing the Galerkin procedure, and also for testing convergence of the series (4.66). It is recommended that orthonormal basis sets essentially always be used.

Finally, we note that the summation indexing, $k = 1, \ldots, \infty$, is not the only possible one. Choice of indexing is dictated by details of the problem being solved and the basis set employed to do this, and could also be either $k = 0, \ldots, \infty$ or $k = -infty, \ldots, 0, \ldots, \infty$ in place of the one used herein.

Example Problem

It is probably best to illustrate implementation of the Galerkin procedure by means of a simple, but fairly general example. We choose the following relatively simple, but nonlinear, problem:

$$u'' - u^2 = f(x) \ , \quad x \in (0,1) \ , \tag{4.68}$$

with boundary conditions

$$u(0) = u(1) = 0 \ . \tag{4.69}$$

Observe that because this is a nonlinear problem some minor complications will arise in implementing the Galerkin procedure, thus providing an opportunity to discuss some subtleties.

Our first task is to choose a set of basis functions. A general way to do this is to consider the eigenvalue problem corresponding to the linear part of the differential operator. Thus, we seek $\{\phi_k\}_{k=1}^{\infty}$ satisfying

$$\phi'' = \lambda\phi \ , \quad x \in (0,1) \tag{4.70}$$

with boundary conditions

$$\phi(0) = \phi(1) = 0 \ . \tag{4.71}$$

Since this is an eigenvalue problem associated with a very basic Sturm–Liouville problem, it is known that the eigenfunctions are complete in $L^2[0,1]$. Furthermore, the *principal part* of the operator (that part consisting of highest-order derivative terms) is the same in both (4.68) and (4.70). Thus, we expect a series of the form (4.66) to provide a solution to the problem (4.68, 4.69) when ϕ_ks are nontrivial solutions of (4.70, 4.71).

It is also worth mentioning that in the case where the boundary conditions (4.69) are nonhomogeneous, we might still use as a basis the same functions ϕ_k obtained from (4.70, 4.71). These would no longer satisfy the boundary conditions, say

$$u(0) = A \ , \quad u(1) = B \ ; \tag{4.72}$$

but we can make a slight modification to (4.66) to remedy this. Namely, we replace (4.66) with

$$u(x) = A(1-x) + Bx + \sum_{k=1}^{\infty} a_k \phi_k(x) \ . \tag{4.73}$$

This implies a new basis set, $\{1, x\} \cup \{\phi_k\}_{k=1}^{\infty}$. We should observe that this basis set is complete, but no longer orthonormal.

Once the basis functions have been chosen and a series representation has been constructed, we are ready to determine the unknown series coefficients. We begin by substituting (4.66) into (4.68). This yields

$$\sum_i a_i \phi_i''(x) - \left(\sum_i a_i \phi_i(x) \right)^2 = f(x) \, , \tag{4.74}$$

provided we assume commutativity of differentiation and series summation in the first term.

We note that the nonlinear term requires some special attention. Namely, it should be rewritten as

$$\begin{aligned} \left(\sum_i a_i \phi_i \right)^2 &= \left(\sum_i a_i \phi_i \right) \left(\sum_i a_i \phi_i \right) \\ &= \left(\sum_i a_i \phi_i \right) \left(\sum_j a_j \phi_j \right) \ , \end{aligned}$$

to prevent inadvertent loss of terms from the final product series. We next rearrange these so that

$$\left(\sum_i a_i\phi_i\right)^2 = \sum_i \sum_j a_i a_j \phi_i \phi_j \ .$$

In general, the validity of such a rearrangement, and hence of the above indicated equality, requires *absolute convergence* of the individual series involved in the rearrangement. However, for the Fourier series considered here it can be shown, via the *Parseval identity* (see, *e.g.*, [29]), that less is required. Indeed, all that is necessary is convergence in ℓ^2. But this is required for (4.66) to be a solution, in the first place.

We next rewrite (4.74) as

$$\sum_i a_i\phi_i''(x) - \sum_{i,j} a_i a_j \phi_i \phi_j = f(x) \ . \tag{4.75}$$

We must now find a way to determine the set of coefficients, $\{a_k\}_{k=1}^{\infty}$. In particular, we need a procedure for obtaining an infinite system of equations from (4.75) since the a_ks comprise an infinite sequence of unknown coefficients. Recall that each ϕ_k is a known function, as is the function f. Thus, if we form the inner product of (4.75) with ϕ_k, we obtain an algebraic equation for the a_ks. Moreover, by (4.66), there are exactly the same number of ϕ_ks as a_ks; so if we form an inner product corresponding to each ϕ_k, we will obtain the required number of equations. Thus, we have

$$\left\langle \sum_i a_i\phi_i'', \phi_k \right\rangle - \left\langle \sum_{i,j} a_i a_j \phi_i \phi_j, \phi_k \right\rangle = \langle f, \phi_k \rangle \tag{4.76}$$

for $k = 1, 2, \ldots$. This expression, or the equivalent,

$$\left\langle u'' - u^2, \phi_k \right\rangle = \langle f, \phi_k \rangle \ ,$$

is called the *weak form* of Eq. (4.68). The reason is that the a_ks obtained from (4.76) do no necessarily lead to a function $u(x)$, via (4.66), that satisfies Eq. (4.68), itself; that is,

$$u'' - u^2 = f(x).$$

The reason for this is quite clear. Equation (4.76) is an integral relation while (4.68) is a differential equation. It is possible that a_ks determined from (4.74) will not lead to a function $u(x)$ that is smooth enough to permit the differentiation required in (4.68). Recall that we only require $\{a_k\} \in \ell^2$, which implies only that $u(x) \in L^2$.

Now we observe that the system of equations, (4.76), is not suitable for numerical computation because it is of infinite (countable in this case) dimension. To implement the Galerkin procedure we must be content to consider approximations

$$u_N(x) = \sum_{k=1}^{N} a_k \phi_k(x) \tag{4.77}$$

such that

$$\begin{aligned} u(x) &= u_N(x) + \sum_{k=N+1}^{\infty} a_k \phi_k(x) \\ &= u_N(x) + R(x) \ . \end{aligned}$$

Clearly, if the series (4.66) is convergent (which is required for existence of a solution) $\exists\ N$ (depending on ϵ) such that

$$\|u(x) - u_N(x)\| = \|R(x)\| < \epsilon \quad \forall\ \epsilon > 0\ ,$$

for some norm, $\|\cdot\|$ (in particular, for the L^2-norm). Thus, for some N we can carry out the preceding development using (4.77) in place of (4.66) and expect to obtain a reasonably good approximation to the solution of (4.68, 4.69).

In place of the infinite system (4.76), we now have the $N \times N$ system

$$\left\langle \sum_{i=1}^{N} a_i \phi_i'', \phi_k \right\rangle - \left\langle \sum_{i,j=1}^{N} a_i a_j \phi_i \phi_j, \phi_k \right\rangle = \langle f, \phi_k \rangle \tag{4.78}$$

for $k = 1, 2, \ldots, N$. We now carry out the indicated integrations (which in some cases may require numerical quadrature) to obtain

$$\sum_{i=1}^{N} a_i \langle \phi_i'', \phi_k \rangle - \sum_{i,j=1}^{N} a_i a_j \langle \phi_i \phi_j, \phi_k \rangle = \langle f, \phi_k \rangle\ ,$$

or

$$\sum_{i=1}^{N} a_i A_{ik} - \sum_{i,j=1}^{N} a_i a_j B_{ijk} = F_k\ , \quad k = 1, 2, \ldots, N\ , \tag{4.79}$$

with $A_{ik} \equiv \langle \phi_i'', \phi_k \rangle$, $B_{ijk} \equiv \langle \phi_i \phi_j, \phi_k \rangle$. Equation (4.79) is a $N \times N$ nonlinear system for the a_ks; it is customary to solve it via Newton's method treated in Chap. 2.

Convergence of Galerkin Procedures

It is important to note that the matrices corresponding to the values of the inner products, A_{ik} and B_{ijk}, are nonsparse. So the values of permissible N are generally much smaller than is the case for finite-difference methods. On the other hand, it can be shown that under certain smoothness assumptions on u, and with appropriate types of boundary conditions, the Galerkin procedure (and spectral methods, in general) are "infinite" order. This means that if we define the error to be

$$e_N \equiv \|u(x) - u_N(x)\|$$

for some norm $\|\cdot\|$, then $e_N \to 0$ faster than any finite power of $\frac{1}{N}$. That is,

$$e_N \sim o\left(\frac{1}{N^p}\right) \quad \forall\ p < \infty\ .$$

There are two separate convergence tests that must be performed on Galerkin solutions to nonlinear problems. The first is convergence of the Newton iterations used in the solution of (4.79). This can be done in the usual manner, since as far as these iterations are concerned, only a finite-dimensional system is involved. The second is convergence of the series representation (4.77). This is analogous to convergence of grid functions in the case of finite-difference solutions. But now we monitor convergence with respect to increasing the number of terms in the series expansion. We will consider three alternative criteria by which to test convergence.

The first is the one that we would always prefer to satisfy because it implies that the solution is as smooth as required by the differential equation. Namely, we check that

$$e_N \equiv \max_{x \in [0,1]} \left| u_N'' - u_N^2 - f \right| < \epsilon\ ,$$

for all N greater than or equal to some value N_0. This is the *strong operator max-norm* error or *max-norm residual*. It is possible, for reasons alluded to earlier, that convergence in this norm may not be achieved.

The second convergence test is *strong convergence* in L^2. This implies that $e_N < \epsilon$ for all N greater than some specific N_0 with e_N defined in terms of the L^2 norm as

$$e_N \equiv \left(\int_0^1 [u_N'' - u_N^2 - f(x)]^2 dx \right)^{\frac{1}{2}} < \epsilon \, .$$

Finally, a weaker, and most widely used convergence test is

$$e_N = \left(\int_0^1 \left[u(x) - u_N(x) \right]^2 \, dx \right)^{\frac{1}{2}} < \epsilon \, .$$

This simply implies convergence of the Fourier representation, but it does not show that the differential equation, itself, has been satisfied. Since, in general, we do not know $u(x)$, we typically replace the above with

$$\left(\int_0^1 \left[u_{N+M}(x) - u_N(x) \right]^2 \, dx \right)^{\frac{1}{2}} < \epsilon \, ,$$

where $M \geq 1$. If the ϕ_ks are orthonormal, it follows from (4.77) and Parseval's identity that we may simply test

$$\sum_{k=N+1}^{N+M} a_k^2 < \epsilon \, .$$

4.3.4 Summary

The initial value problem methods treated in this chapter have included both single-step and multi-step techniques, but with emphasis on low-order versions of the former. This has been done for simplicity of presentation and also due to the wide use of such methods in the context of discretizations of partial differential equations to be studied in the next chapter.

We have also covered a broad range of topics associated with two-point boundary-value problems for ODEs. In particular, shooting, finite-difference and Galerkin methods have all received attention; and in addition special topics including treatment of coordinate singularities and nonlinearities have been discussed. The emphasis, however, has clearly been on finite-difference methods because these will be employed in our numerical treatment of partial differential equations to follow.

Chapter 5

Numerical Solution of Partial Differential Equations

In this chapter we will, in a sense, combine all of the material developed in preceding chapters to construct straightforward, elmentary techniques for solving linear partial differential equations (PDEs). We begin with some introductory mathematical topics involving classification of PDEs and the notion of well posedness. We then consider finite-difference approximation of parabolic, elliptic and hyperbolic equations, and conclude the chapter with a brief introduction to treatment of first-order hyperbolic equations and systems.

5.1 Mathematical Introduction

In the present treatment we will mainly restrict attention to the partial differential equations of classical mathematical physics, namely the heat, wave and Laplace/Poisson equations, and we will treat these only with well-known, "classical" numerical methods. This is done for the sake of brevity; but it is important that these elementary methods be mastered prior to any attempt to study more complicated problems and correspondingly more advanced numerical procedures. Moreover, these relatively simple methods can often be valuable in their own right.

In contrast to the situation for ordinary differential equations, the state of development of numerical methods for partial differential equations is somewhat less advanced. This is so for a variety of reasons, not the least of which is the fact that, theoretically, PDEs are more difficult to treat than are ODEs. In turn, this is true for a number of reasons. Recall that there are only two basic types of problems associated with ODEs: IVPs and BVPs. Moreover, in each of these cases the required auxiliary conditions are readily identified, and they can be implemented numerically in a quite uniform manner across a broad spectrum of individual problems.

In the PDE case there are three separate classes of problems: *i*) IVPs, usually called "Cauchy problems" in the PDE context, *ii*) BVPs, and *iii*) initial boundary value problems (IBVPs). Furthermore, within each of these classes, individual problems may be formulated for one or more types of linear differential operators, namely elliptic, parabolic, or hyperbolic, as well as various less easily classified nonlinear operators. The generic features of solutions, and to some extent solution techniques, differ among these problem types. Additionally, because partial differential equations are often posed on domains of more than one spatial dimension, boundary conditions must usually be imposed along a curve (or even on a surface), rather than simply at two endpoints of an interval as in the ODE case. Likewise, initial data must be prescribed for the entire spatial domain, instead of at a single point as for ODE IVPs. All of these features combine to provide

a richness in variety of PDE problems that is not present for ODE problems. But this richness implies difficulties that also are not present in the ODE case. Nevertheless, within the framework of the problems to be considered in these lectures, the theory—both analytical and numerical—is complete and well understood.

There are two pieces of mathematics regarding partial differential equations that are extremely important for numerical analysis. The first of these, classification, is one of the first topics covered in any elementary PDE course, while the second, well posedness, is generally introduced in more advanced courses. Here, we will briefly discuss each of these before proceeding to numerical treatment of specific classes of problems.

5.1.1 Classification of Linear PDEs

The three above mentioned types of equations are all examples of linear second-order partial differential equations. The most general form of such equations, when restricted to two independent variables and constant coefficients, is

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu = g(x, y)\,, \tag{5.1}$$

where g is a known forcing function; $a, b, c, \ldots$, are given constants, and subscripts denote partial differentiation. In the homogeneous case, *i.e.*, $g \equiv 0$, this form is reminiscent of the general quadratic form from high school analytic geometry:

$$ax^2 + bxy + cy^2 + dx + ey + f = 0\,. \tag{5.2}$$

Equation (5.2) is said to be elliptic, parabolic or hyperbolic according as $b^2 - 4ac$ is less than, equal to, or greater than zero. This same classification is employed for (5.1), independent of the nature of $g(x, y)$. In fact, it is clear that the classification of linear PDEs depends only on the coefficients of the highest-order derivatives. This grouping of terms,

$$au_{xx} + bu_{xy} + cu_{yy}\,,$$

is called the *principal part* of the differential operator in (5.1), and this notion can be extended in a natural way to more complicated operators. Thus, the type of a linear equation is completely determined by its principal part. It should be mentioned that if the coefficients of (5.1) are permitted to vary with x and y, its type may change from point to point within the solution domain. This can pose difficulties whose treatment requires methods far beyond the scope of the material to be presented in these lectures.

We next note that corresponding to each of the three types of equations there is a unique *canonical form* to which (5.1) can always be reduced. We shall not present the details of the transformations needed to achieve these reductions, as they can be found in many standard texts on elementary PDEs (*e.g.*, Berg and MacGregor [2]). On the other hand, it is important to be aware of the possibility of simplifying (5.1), since this may also simplify the numerical analysis required to construct a solution algorithm.

It can be shown when $b^2 - 4ac < 0$, the elliptic case, that (5.1) collapses to the form

$$u_{xx} + u_{yy} + Au = g(x, y)\,, \tag{5.3}$$

with $A = 0, \pm 1$. When $A = 0$ we obtain *Poisson's equation*, or *Laplace's equation* in the case $g \equiv 0$; otherwise, the result is usually termed the *Helmholtz equation*. For the parabolic case, $b^2 - 4ac = 0$, we have

$$u_x - u_{yy} = g(x, y)\,, \tag{5.4}$$

which is the *heat equation*, or the diffusion equation; and for the hyperbolic case, $b^2 - 4ac > 0$, Eq. (5.1) can always be transformed to

$$u_{xx} - u_{yy} + Bu = g(x, y)\,, \tag{5.5}$$

where $B = 0$ or 1. If $B = 0$, we have the *wave equation*, and when $B = 1$ we obtain the linear *Klein–Gordon equation*. The Laplace, heat and wave equations are often termed the "equations of classical mathematical physics," and we will consider methods for solving each of these in the sections that follow. We remark that the other linear equations just mentioned can be solved by the same methods within each classification.

5.1.2 Basic Concept of Well Posedness

Before proceeding to introduce numerical methods for solving each of the three main classes of problems it is worthwhile to give some consideration to the question of under what circumstances these equations do, or do not, have solutions. This is a part of the mathematical concept of *well posedness*. We begin with a formal definition of this property.

Definition 5.1 *A problem consisting of a partial differential equation and boundary and/or initial conditions is said to be well posed in the sense of Hadamard if it satisfies the following conditions:*

i) a solution exists;

ii) the solution is unique;

iii) the solution depends continuously on given data.

The well-posedness property is crucial in solving problems by numerical methods because essentially all numerical algorithms embody the tacit assumption that problems to which they apply are well posed. Consequently, a method is not likely to work correctly on an ill-posed (*i.e.*, a not well-posed) problem. The result may be failure to obtain a solution; but a more serious outcome may be generation of numbers that have no association with reality in any sense. It behooves the user of numerical methods to understand the mathematics of any considered problem sufficiently to be aware of the possible difficulties—and symptoms of these difficulties—associated with problems that are not well posed.

We close this discussion of well posedness by describing a particular problem that is not well posed. This is the so-called "backward" heat equation problem. It arises in geophysical studies in which it is desired to predict the temperature distribution within the Earth at some earlier geological time by integrating backward from the (presumed known) temperature distribution of the present. To demonstrate the difficulties that arise we consider a simple one-dimensional heat equation.

$$u_t = \kappa u_{xx}\,, \qquad x \in (-\infty, \infty)\,, \quad t \in [-T, 0)\,,$$

with

$$u(x, 0) = f(x)\,.$$

Formally, the exact solution is

$$u(x, t) = \frac{1}{\sqrt{4\pi\kappa t}} \int_{-\infty}^{\infty} f(\xi) e^{-\frac{(x-\xi)^2}{4\kappa t}}\, d\xi\,, \tag{5.6}$$

the derivation of which (see, *e.g.*, Berg and MacGregor [2]) imposes no specific restrictions on the sign of t. But we immediately see that if $t < 0$, $u(x, t)$, if it exists at all, is imaginary (since

κ, the thermal diffusivity, is always greater than zero). In fact, unless f decays to zero faster than exponentially at $\pm\infty$, there is no solution because the integral in (5.6) does not exist. It will turn out that this behavior of heat equation solutions will place restrictions on the form of difference approximations that can be used to numerically solve it. Later we will present a very natural difference approximation to the heat equation that completely fails, in part, because of nonexistence of solutions to the backward heat equation.

5.2 Overview of Discretization Methods for PDEs

Although in the sequel we will consider only basic finite-difference methods for approximating solutions of partial differential equations, in the present section we shall provide brief discussions of several of the most widely-used discretization techniques. We note at the outset that temporal discretizations are almost always either finite-difference or quadrature based, but many different methods, and combinations of these, may be employed for spatial approximation. Figure 5.1 depicts the main features of what are the most well-known classes of methods: *i*) finite difference, *ii*), finite element and *iii*) spectral.

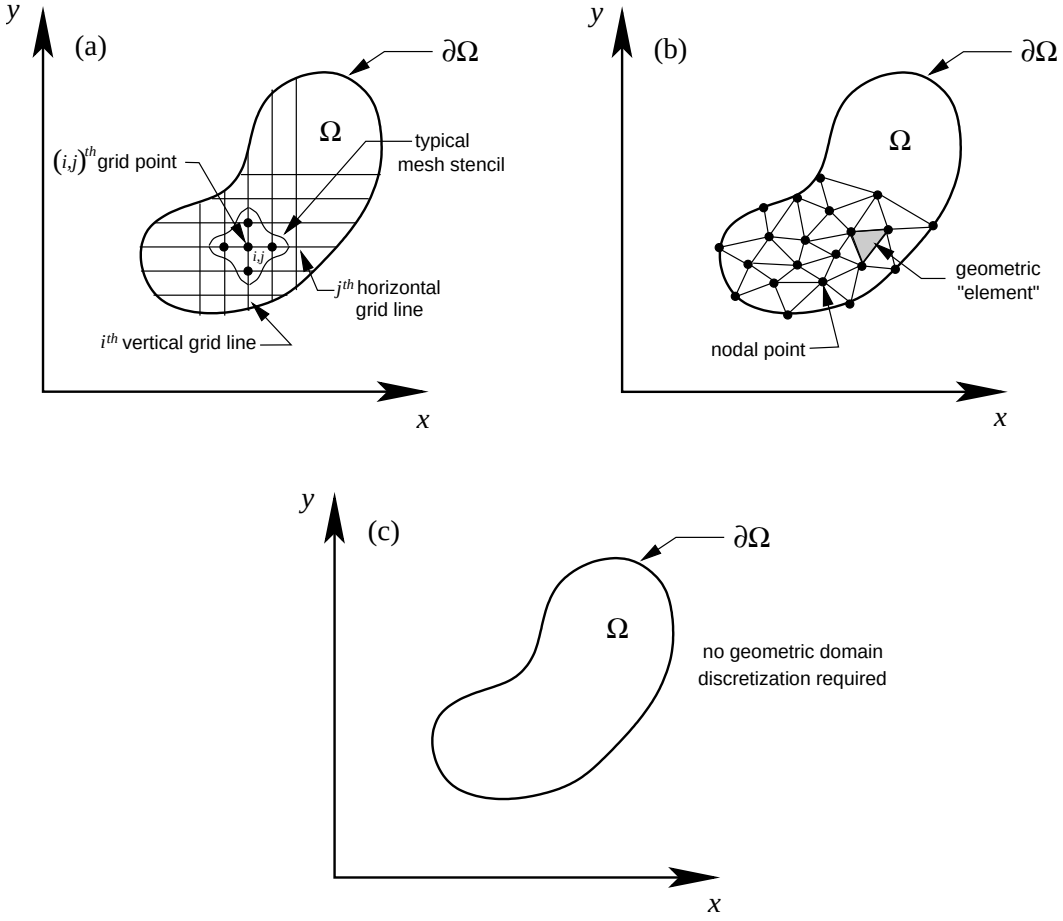


Figure 5.1: Methods for spatial discretization of partial differential equations; (a) finite difference, (b) finite element and (c) spectral.

As we will describe in considerable detail in later sections, finite-difference methods are con-

structed by first "gridding" the solution domain as indicated in Fig. 5.1(a), and then deriving systems of algebraic equations for grid-point values which serve as approximations to the true solution at the discrete set of points defined (typically) by intersections of the grid lines. The figure depicts the 2-D analogue of what was already been done for ODE BVPs in Chap. 4. We remark that the regular "structured" grid shown here is not the only possibility. But we will not treat finite-difference approximations on "unstructured" grids in the present lectures. (The interested reader is referred to Thompson *et al.* [35] for comprehensive treatments of both structured and unstructured gridding techniques, generally referred to as "grid generation.")

Finite-element methods (FEMs) are somewhat similar to finite-difference methods, although they are typically more elaborate, often somewhat more accurate, and essentially always more difficult to implement. As can be seen from Fig. 5.1(b) the problem domain is subdivided into small regions, often of triangular (or, in 3-D, tetrahedral) shape. On each of these subregions a polynomial (not unlike those discussed in Chap. 3, but multidimensional) is used to approximate the solution, and various degrees of smoothness of the approximate solution are achieved by requiring constructions that guarantee continuity of a prescribed number of derivatives across element boundaries. This approximation plus the subregion on which it applies is the "element." It should be remarked that the mathematics of FEMs is highly developed and is based on variational principles and weak solutions (see, *e.g.*, Strang and Fix [31]), in contrast to the Taylor-expansion foundations of finite-difference approaches. In this sense it bears similarities to the Galerkin procedure discussed in Chap. 4.

Figure 5.1(c) displays the situation for spectral methods. One should observe that in this case there is no grid or discrete point set. Instead of employing what are essentially local polynomial approximations as done in finite-difference and finite-element methods, assumed forms of the solution function are constructed as (generalized) Fourier representations that are valid globally over the whole solution domain. In this case, the unknowns to be calculated consist of a finite number of Fourier coefficients leading to what amounts to a projection of the true solution onto a finite-dimensional space. We previously constructed such a method in Chap. 4 in the solution of ODE BVPs via Galerkin procedures. In modern terminology, these are examples of so-called "grid-free" methods.

There are numerous other discretization methods that have attained fairly wide acceptance in certain areas. In computational fluid dynamics (CFD) finite-volume techniques are often used. These can be viewed as lying between finite-difference and finite-element methods in structure, but in fact they are essentially identical to finite-difference methods despite the rather different viewpoint (integral formulation of governing equations) employed for their construction. Other less often-used approaches deserving of at least mention are spectral-element and boundary-element methods. As their names suggest, these are also related to finite-element methods, and particularly in the latter case are applicable mainly for a very restricted class of problems. Beyond these are a number of different "pseudo-spectral" methods that are similar to spectral methods, but for which the basis functions employed are not necessarily eigenfunctions of the principal part of the differential operator(s) of the problem being considered. In addition to all these individual techniques, various calculations have been performed using two or more of these methods in combination. Particular examples of this include finite-difference/spectral and finite-element/boundary-element methods. Discussion of details of such schemes is far beyond the intended scope of the present lectures, and from this point onward our attention will be focused on basic finite-difference methods that have been widely used because of their inherent generality and relative ease of application.

5.3 Parabolic Equations

In this section we will consider several widely-used methods for solving parabolic problems. We begin with forward- and backward-Euler methods applied to the 1-D heat equation, the former for the Cauchy problem and the latter for an initial boundary value problem. Both of these are first-order accurate in time. We then treat two second-order methods. The first is unconditionally unstable and should never be used, but it is a very intuitively natural scheme; the second is the well-known unconditionally stable Crank–Nicolson method. We end the section with analysis of the Peaceman–Rachford alternating direction implicit method, a very efficient technique for applying the Crank–Nicolson scheme to 2-D parabolic equations.

5.3.1 Explicit Euler Method for the Heat Equation

We begin our study of the numerical solution of parabolic equations by introducing the simplest procedure for solving the Cauchy problem for the one-dimensional heat equation. We consider

$$u_t = u_{xx} \ , \quad x \in (-\infty, \infty) \ , \quad t \in (0, T] \ , \tag{5.7}$$

with initial data

$$u(x, 0) = u_0(x) \ . \tag{5.8}$$

This can be treated in a manner analogous to what was done for ODE initial-value problems in Chap. 4.

Recall that for such problems, the easiest scheme to construct is the forward-Euler method, which consists of replacing the time derivative with a forward difference, and evaluating the right-hand side at the previous time level. Thus, (5.7) becomes

$$\frac{u_m^{n+1} - u_m^n}{k} = (u_{xx})_m^n$$

or

$$u_m^{n+1} = u_m^n + k(u_{xx})_m^n \ . \tag{5.9}$$

In (5.9) the m-subscripts denote spatial grid points, while the n-superscripts indicate the time level. To complete the discretization we replace u_{xx} with a centered-difference approximation and obtain

$$u_m^{n+1} = u_m^n + \frac{k}{h^2}\left(u_{m-1}^n - 2u_m^n + u_{m+1}^n\right) \ , \quad m = 1, 2, \ldots, M \ . \tag{5.10}$$

As indicated in Fig. 5.2, k denotes the time step, and h is the space step. Recall that in the context of ODEs, Euler's method is a single-step procedure; in the PDE context we use the terminology *two-level scheme* since contributions from two time levels, n and $n + 1$, appear in the discrete formula.

It is easily seen that (5.10) is an *explicit method*, since the new time level values at each grid point can be directly calculated from previous ones. However, it should be apparent that we cannot compute all values at time level $n+1$ from (5.10) using only the values given at time level n because at each of the endpoints of the computational interval the right-hand side of (5.10) requires one additional point. This is indicated in Fig. 5.2. Here, we show the so-called *grid stencil*, or *mesh star*, for the general m^{th} and final M^{th} grid points. The dashed line in the latter stencil indicates that there is no grid point value available for evaluation of that part of the stencil.

There are several methods by which this situation can be remedied; we will describe only one of them here. It is to calculate only between $m = 2$ and $M-1$ at the second time level, between

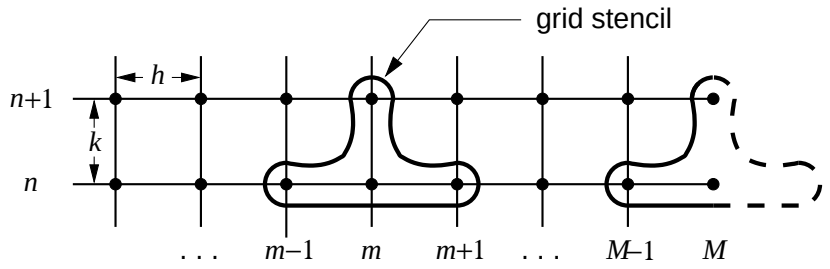


Figure 5.2: Mesh star for forward-Euler method applied to heat equation

$m = 3$ and $M-2$ at the third, *etc.* That is, we lose one spatial grid point from each end of the discrete domain with each new calculated time step. Thus, we must start with a sufficient number of grid function values at the initial time to guarantee that we cover the desired spatial interval at the final time. This approach is computationally inefficient because more grid points are computed than are needed in all but the final step. On the other hand, stability analyses (see below) for such a method are much more straightforward than for alternatives employing so-called "outflow" or "nonreflecting" boundary conditions.

We now provide a pseudo-language algorithm for implementing this method.

Algorithm 5.1 *(Forward-Euler/Centered-Difference Approximation for 1-D Heat Equation)*

1. *Input number of time steps N_t, number of space steps at final time N_x, time step size k, and $[a_x, b_x]$ the solution domain at final time.*

2. *Input initial data:* $u_i^0,\ i = 1, 2, \ldots, N_x$.

3. *Load zeros at additional points needed to obtain solution on required final interval:*

   Do $i = 1, N_t$

   $u_{1-i}^0 = 0$
   $u_{N_x+i}^0 = 0$

   Repeat i

4. *Calculate spatial step size, h:* $h = (b_x - a_x)/(N_x - 1)$.

5. *Begin time stepping:*

   Do $n = 1, N_t$

   $mstrt = n - N_t$
   $mstop = N_x + N_t - n$
   Do $m = mstrt, mstop$
   $u_m^n = u_m^{n-1} + \frac{k}{h^2}\left(u_{m-1}^{n-1} - 2u_m^{n-1} + u_{m+1}^{n-1}\right)$
   Repeat m
   Output results for time step n

   Repeat n

The main reason for introducing the explicit approximation (5.10) is that it is quite simple, and it provides an easy starting point for the discussions of *consistency* and *stability* of difference approximations to partial differential equations. By consistency we mean that the difference approximation converges to the PDE as $h, k \to 0$, and by stability we mean that the solution to the difference equation does not increase with time at a faster rate than does the solution to the differential equation. Consistency and stability are crucial properties for a difference scheme because of the following theorem due to Lax (see Richtmyer and Morton [24]).

Theorem 5.1 *(Lax Equivalence Theorem) Given a well-posed linear initial-value problem, and a corresponding consistent difference approximation, the resulting grid functions converge to the solution of the differential equation(s) as $h, k \to 0$ if and only if the difference approximation is stable.*

It is important to understand the content of this theorem. One can first deduce the less than obvious fact that consistency of a difference approximation is not a sufficient condition for guaranteeing that the grid functions produced by the scheme actually converge to the solution of the original differential equation as discretization step sizes are refined. In particular, both consistency and stability are required. As will be evident in what follows, consistency of a difference approximation is usually very straightforward though sometimes rather tedious to prove, while proof of stability can often be quite difficult.

Consistency of Forward-Euler/Centered-Difference Approximation to the Heat Equation

We will now demonstrate consistency of the difference scheme (5.10). As a byproduct we also obtain the order of accuracy of the difference approximation. For this analysis it is convenient to rewrite (5.10) as

$$u_m^{n+1} - \left(1 - \frac{2k}{h^2}\right) u_m^n - \frac{k}{h^2}\left(u_{m-1}^n + u_{m+1}^n\right) = 0 \ . \tag{5.11}$$

Now from Taylor's theorem,

$$\begin{aligned}
u_m^{n+1} &= u_m^n + k(u_t)_m^n + \frac{k^2}{2}(u_{tt})_m^n + \cdots \\
u_{m-1}^n &= u_m^n - h(u_x)_m^n + \frac{h^2}{2}(u_{xx})_m^n - \frac{h^3}{6}(u_{xxx})_m^n + \frac{h^4}{24}(u_{xxxx})_m^n - \cdots \\
u_{m+1}^n &= u_m^n + h(u_x)_m^n + \frac{h^2}{2}(u_{xx})_m^n + \frac{h^3}{6}(u_{xxx})_m^n + \frac{h^4}{24}(u_{xxxx})_m^n + \cdots \ ,
\end{aligned}$$

and if we substitute these expansions into (5.11) and rearrange the result, we find

$$(u_t - u_{xx})_m^n + \left[\frac{k}{2}(u_{tt})_m^n - \frac{h^2}{12}(u_{xxxx})_m^n\right] + \cdots = 0 \ . \tag{5.12}$$

The first term in this expression is the original differential equation evaluated at the arbitrary space-time point (x_m, t^n). Thus, the difference approximation converges to the differential equation provided the second term $\to 0$ with h, k (under the assumption that all neglected terms are indeed small compared with those retained). This will usually hold (but somewhat pathological exceptions can be constructed) whenever u_{tt} and u_{xxxx} are continuous functions of (x, t); *i.e.*, $u \in C^{2,4}$. Hence, consistency has been shown within this context.

The second term of (5.12) is the *dominant*, or *principal*, term of the *global truncation error*. This is analogous to the ODE case discussed in Chap. 4. In order notation we also have that the local truncation error is $\mathcal{O}\left(k\left(k+h^2\right)\right)$. The global (in time) truncation error, and thus the *order of the method* is $\mathcal{O}\left(k+h^2\right)$. Hence, we have shown that the difference scheme based on forward-Euler time integration and centered spatial differencing is consistent with the heat equation, and it is first order in time and second order in space. This is the expected result.

Stability of Forward-Euler Approximation to the Heat Equation

We next carry out a stability analysis for the difference equation (5.10). Define $r \equiv k/h^2$, and now write (5.10) as

$$u_m^{n+1} = (1-2r)u_m^n + r\left(u_{m-1}^n + u_{m+1}^n\right) \ . \tag{5.13}$$

It is not altogether obvious that difference equations can be unstable, particularly when they are consistent approximations to a well-posed PDE problem, but recall the situation for ODE IVPs discussed in Chap. 4; we will here demonstrate that analogous behavior occurs in the PDE case. It is worth noting that were this not true the consequences of the Lax theorem would be mere triviality.

We have already seen that for a consistent difference approximation the difference between the theoretical solution of the PDE and the theoretical solution of the difference equation is related to the truncation error, and hence for sufficiently small h, k is always bounded. However, the theoretical solution to the difference equation will not be obtained in actual calculations, mainly because of machine rounding errors; so we must define a new error component as the difference between the theoretical solution u_m^n and the computed solution v_m^n of the difference equation: we set

$$z_m^n = u_m^n - v_m^n \, ,$$

where v_m^n is the result of actual computation, again analogous to the ODE IVP treatment of Chap. 4. We wish to investigate the growth of z_m^n as n increases, for if this quantity grows rapidly it follows that the computed solution may bear no resemblance to the true solution, and in such a case Eq. (5.13) will be called *unstable*.

To proceed with this investigation we note, as we have already seen in the ODE case, that for linear difference equations z_m^n must satisfy the same difference equation as does u_m^n and v_m^n. Hence, in the present case, from (5.13) we have

$$z_m^{n+1} = (1-2r)z_m^n + r(z_{m-1}^n + z_{m+1}^n) \ . \tag{5.14}$$

We will now assume that $z(x,t)$, the continuous analog of the grid function $\{z_m^n\}$, has a Fourier representation

$$z(x,t) = \sum_l A_l(t) e^{i\beta_l x}$$

for arbitrary wavenumbers β_l. We observe that this is a rather natural assumption in light of the form of exact solutions to the heat equation (see, *e.g.*, [2]). It is sufficient to consider a single arbitrary term of this series because if any one term is growing unboundedly, then this must also be true, in general, for the series.

Now we want the initial error at $t = 0$ to consist only of error in the spatial representation, so we set

$$z_m^n = e^{\alpha t} e^{i\beta x_m} = e^{\alpha n k} e^{i\beta m h} = \xi^n e^{i\beta m h} \, , \tag{5.15}$$

where the first right-hand equality follows from exact solutions to the heat equation but now with the l index suppressed. Clearly, at $t = 0$, the error is $z_m^0 = e^{i\beta mh}$; furthermore, this will not grow in time, provided

$$|\xi| \leq 1 \ . \tag{5.16}$$

This is the well-known *von Neumann stability criterion* for stability of discretizations of parabolic PDEs.

For two-level methods, and only a single differential equation, it is both necessary and sufficient for stability; but for schemes employing three (or more) time levels, and/or applied to more than one equation, it is only necessary, but not sufficient in general for stability.

We now substitute the right-hand side of (5.15) into (5.14) to obtain

$$\xi^{n+1} e^{i\beta mh} = (1-2r)\xi^n e^{i\beta mh} + r\left(\xi^n e^{i\beta(m-1)h} + \xi^n e^{i\beta(m+1)h}\right) \ .$$

Division of $\xi^n e^{i\beta mh}$ leaves

$$\begin{aligned} \xi &= (1-2r) + r\left(e^{-i\beta h} + e^{i\beta h}\right) \\ &= 1 - 2r(1-\cos\beta h) \\ &= 1 - 4r\sin^2\frac{\beta h}{2} \ . \end{aligned}$$

Thus, in order for the forward-Euler method to satisfy the von Neumann condition (5.16) we must choose r so that

$$-1 \leq 1 - 4r\sin^2\frac{\beta h}{2} \leq 1 \qquad \forall \ \beta h \, .$$

It is easily seen that the right-hand inequality is satisfied $\forall\ r \geq 0$, while the left-hand side implies

$$r \leq \frac{1}{2\sin^2\frac{\beta h}{2}} \ .$$

We must choose βh so that the right-hand side is a minimum in order to place the strongest restrictions on r. This minimum occurs when the denominator is a maximum, namely when $\sin^2\frac{\beta h}{2} = 1$. Thus, the stability requirement for the forward-Euler scheme is

$$0 \leq r \leq \frac{1}{2} \ . \tag{5.17}$$

(Of course we would never take $r = 0$ because this implies a zero time step.)

If we recall that $r = k/h^2$, we see an important consequence of this stability criterion: it is that as the spatial grid is refined to improve accuracy, the time steps must be decreased as the square of the spatial grid size to maintain stability. For example, if $h = 0.1$, then we must have $k \leq 0.005$. This is often a far too restrictive requirement, and it has motivated development of implicit methods such as those that we next consider.

5.3.2 Backward-Euler Method for the Heat Equation

Although the backward-Euler method is only first-order accurate in time, it is nevertheless widely used because of its favorable stability properties and its simplicity. Recall from our discussions for ODEs that the backward-Euler procedure is A-stable. We will see that for the PDE case it is unconditionally stable for linear parabolic problems.

We now consider the heat equation on a bounded domain; *i.e.*, we treat the initial boundary value problem

$$u_t = u_{xx} \ , \quad x \in (0,1) \ , \quad t \in (0,T] \ , \tag{5.18}$$

with initial conditions

$$u(x,0) = u_0(x) \ , \tag{5.19}$$

and boundary conditions

$$u(0,t) = f(t) \ , \quad u(1,t) = g(t) \ . \tag{5.20}$$

Here $u_0(x)$, $f(t)$ and $g(t)$ are known functions. To approximate (5.18) with the backward-Euler method, we employ a backward difference for the time derivative approximation, and then in contrast to (5.10), evaluate everything on the right-hand side at the advanced time level. Hence,

$$\frac{u_m^n - u_m^{n-1}}{k} = (u_{xx})_m^n \ ,$$

or after shifting temporal indexing forward by one and introducing a centered-difference spatial approximation for u_{xx},

$$u_m^{n+1} = u_m^n + \frac{k}{h^2}(u_{m-1}^{n+1} - 2u_m^{n+1} + u_{m+1}^{n+1}) \ . \tag{5.21}$$

Now unknown $n+1$ time level results appear on both sides of the equation, so it is not possible to explicitly solve for u_m^{n+1}; hence, this is an *implicit*, but still two-level, method.

As is standard in treating such difference equations, we first move all $n+1$ time level terms to the left-hand side and rearrange this as

$$-u_{m-1}^{n+1} + \left(\frac{1}{r} + 2\right) u_m^{n+1} - u_{m+1}^{n+1} = \frac{1}{r} u_m^n \ , \qquad m = 2, 3, \ldots, M-1 \, . \tag{5.22}$$

The collection of all such equations results in a tridiagonal system for the grid function $\{u_m^{n+1}\}$. Clearly, for $m = 1$ we have

$$u_1^{n+1} = f^{n+1} \ ,$$

and for $m = M$

$$u_M^{n+1} = g^{n+1} \ .$$

Just as was the case for ODE BVPs, these boundary conditions can be inserted directly into the system for u_m^{n+1}, or they can be used to eliminate u_1^{n+1} and u_M^{n+1} from the equations corresponding, respectively, to $m = 2$ and $m = M-1$ in (5.22). (Recall Fig. 4.9.)

We will not present a detailed pseudo-language algorithm for the backward-Euler method at this time because it is very similar to that for the trapezoidal method to be presented later and, in fact, can be embedded as an option in this latter algorithm.

Consistency and Stability of Backward-Euler Method

Consistency (and order of accuracy) of the backward-Euler method can be demonstrated in precisely the same way as already carried out for the forward-Euler discretization, and the result is the same. Namely, as we should expect, backward Euler/centered-discretizations are (globally) first-order accurate in time and second order in space. We leave demonstration of this as an exercise for the reader.

The von Neumann stability analysis applied to (5.22) shows that the method is stable for any value of $r > 0$; *i.e.*, it is unconditionally stable. We leave this also as an exercise to the reader.

It should be noted, however, that such an analysis does not include the effects of either boundary conditions or variable coefficients. Thus, we very briefly consider a method which does, but one that is rather seldom used because it, in general, requires use of additional numerical methods to obtain sharp predictions regarding stability.

The method is known as the *matrix method*, or *matrix stability analysis*, and it is based on the fact that any two-level method for PDEs can be formally represented as

$$A^{n+1}u^{n+1} = B^n u^n + f^n \ , \tag{5.23}$$

where A and B are both square matrices whose elements depend upon the specific difference approximation employed (and, of course, on the specific PDE being solved), and the superscripts indicate that time-dependent coefficients can be treated; f^n is a given nonhomogeneous function and/or boundary data. (Recall the manner in which boundary conditions entered the matrix structure of finite-difference approximations to ODE BVPs in Chap. 4.) It is of interest to observe that if the scheme is explicit, $A^{n+1} = I \ \forall \ n$. By the same arguments employed in the von Neumann stability analysis, we can show for the linear case that the error vector z^n satisfies (5.23) without the inhomogeneous term. We have

$$A^{n+1}z^{n+1} = B^n z^n \ ,$$

or after formally solving for z^{n+1},

$$z^{n+1} = \left(A^{n+1}\right)^{-1} B^n z^n \ . \tag{5.24}$$

Now if the error is not to grow in time we must have

$$\left\| \left(A^{n+1}\right)^{-1} B^n \right\| \leq 1 \ ,$$

where $\|\cdot\|$ denotes the spectral norm. Checking this requires computation of the largest eigenvalue of $\left[\left(A^{n+1}\right)^{-1} B^n\right] \left[\left(A^{n+1}\right)^{-1} B^n\right]^T$, which can seldom be done analytically, and is usually rather expensive if done numerically. Nevertheless, this approach does have some theoretical applications, as we shall see later, and with ever-increasing hardware performance it is beginning to represent a tractable approach for stability analysis for a wide range of numerical PDE problems.

5.3.3 Second-Order Approximations to the Heat Equation

To this point, the (two) methods considered have been only first-order accurate in time. Often, we wish to have more accuracy to permit taking larger time steps, thus reducing rounding errors, or to obtain better accuracy for any given time step size. Hence, we consider two second-order methods. The first is a famous method, due to Richardson, which is consistent with the heat equation to second order in both space and time, but which is unconditionally unstable, and thus useless as a solution method. The second is probably the most widely-used scheme for solving parabolic equations. It is based on trapezoidal integration; it is also second order in both space and time, and is unconditionally stable for linear constant coefficient problems. It is known as the Crank–Nicolson method.

Richardson's Method for the Heat Equation

Richardson's scheme is the most natural second-order method. It is constructed by replacing all derivatives, in both space and time, with second-order centered differences. Thus, for the heat

equation we have the approximation

$$\frac{u_m^{n+1} - u_m^{n-1}}{2k} = \frac{u_{m-1}^n - 2u_m^n + u_{m+1}^n}{h^2} \ ,$$

or

$$u_m^{n+1} = u_m^{n-1} + \frac{2k}{h^2}(u_{m-1}^n - 2u_m^n + u_{m+1}^n) \ . \tag{5.25}$$

We observe that this is a three-level difference scheme, and as a consequence it can have more than one solution. Moreover, because it is centered in time it has a backward as well as a forward (in time) solution, and this is the underlying reason for its instability. Recall that the backward heat equation problem is ill posed, and in general has no bounded solution. This lack of boundedness in the true solution leads to lack of boundedness in the solution to the difference equation as well for any consistent discretization admitting multiple solutions which can propagate backwards in time. We note that a von Neumann stability analysis can be applied to (5.25); however, it is not as direct as in the case of two-level schemes (it depends on conversion of the multi-level method to a system of two-level difference equations; see *e.g.*, Richtmyer and Morton [24] and treatment of the wave equation in Sec. 5.5 below). Furthermore, the results obtained provide necessary, but not generally sufficient, stability criteria.

We must again emphasize that this second-order discretization should never be used for solving the heat equation, or any other parabolic equation. On the other hand, if one is merely seeking a numerical evaluation of the heat equation (for example, to confirm heat balances in laboratory experiments) at a fixed instant in time, then Eq. (5.25) is a quite satisfactory approximation. The difference between this type of application and using (5.25) to solve the heat equation is that evaluation does not imply evolution, and it is (time-like) evolution of difference approximations that can lead to instability.

Crank–Nicolson Method for the Heat Equation

To derive the Crank–Nicolson method we will employ a procedure that has received wide acceptance in the study of difference approximations of evolution, *i.e.*, time-dependent, equations. Namely, we view the PDE as an ODE in time. Thus, we can write the heat equation as

$$\frac{du}{dt} = u_{xx} \equiv F(u) \ , \tag{5.26}$$

with initial and boundary conditions given, for example, by Eqs. (5.19), (5.20) and develop solution procedures based on ODE methods. It is worth mentioning at this point that it can be shown that the eigenvalues of $F(u)$ are negative and large in magnitude. Hence (5.26), viewed as a system of ODEs, is stiff, and we should seek an A-stable method for its solution.

Thus, we apply the trapezoidal rule to (5.26) and obtain

$$u^{n+1} = u^n + \frac{k}{2}\big[F(u^{n+1}) + F(u^n)\big] = u^n + \frac{k}{2}\big[(u_{xx})^{n+1} + (u_{xx})^n\big] \ .$$

We already know that this is (globally) second order in time, and it will be second order in space if we approximate u_{xx} with a second-order centered difference. Hence, we write

$$u_m^{n+1} = u_m^n + \frac{k}{2h^2}\big[(u_{m-1}^{n+1} - 2u_m^{n+1} + u_{m+1}^{n+1}) + (u_{m-1}^n - 2u_m^n + u_{m+1}^n)\big] \ ,$$

or after setting $r \equiv k/2h^2$ and rearranging, we obtain the *Crank–Nicolson method*,

$$u_{m-1}^{n+1} - \left(2 + \frac{1}{r}\right) u_m^{n+1} + u_{m+1}^{n+1} = -u_{m-1}^n + \left(2 - \frac{1}{r}\right) u_m^n - u_{m+1}^n \ ,$$
$$\forall \ m = 2, 3, \ldots, M-1 \, . \quad (5.27)$$

Since the right-hand side consists only of already known grid-function values at time level n, this constitutes a tridiagonal system for the implicit determination of the solution vector u^{n+1} at time level $n+1$.

We should comment that division by $-r$ to obtain Eq. (5.27) is rather customary (although the factor of $\frac{1}{2}$ is usually not included in the definition of r) and analogous to multiplying by h^2 in the ODE case. Also recall, as observed in Chap. 1, that pivoting is seldom performed in the context of tridiagonal LU decomposition, and that diagonal dominance is required to control round-off errors. Clearly, as $r \to 0$ the left-hand side of (5.27) will become strongly diagonally dominant.

It is of interest to analyze the consistency and stability of this widely-used method. In the process we will obtain the formal truncation error. We note at the outset that (5.27) has been arranged in a different manner than was the case for the explicit Euler formula (5.11). In particular, to obtain (5.27) we have multiplied by $1/r$. Our results will thus be global in time. We also carry out the analysis in terms of the $(n+1)^{th}$ time level, rather than the n^{th} as done previously; but this is an unimportant detail that cannot effect the results. (We leave as an exercise for the reader to repeat the following analysis for time level n.) To begin we write (5.27) as

$$u_{m-1}^{n+1} + u_{m-1}^n - \left(2 + \frac{1}{r}\right) u_m^{n+1} - \left(2 - \frac{1}{r}\right) u_m^n + u_{m+1}^{n+1} + u_{m+1}^n = 0 \ . \quad (5.28)$$

We expand u_{m-1}^n about the $n+1^{th}$ time level to obtain

$$u_{m-1}^{n+1} + u_{m-1}^n = 2u_{m-1}^{n+1} - k(u_t)_{m-1}^{n+1} + \frac{k^2}{2}(u_{tt})_{m-1}^{n+1} - \frac{k^3}{6}(u_{ttt})_{m-1}^{n+1} + \cdots \ .$$

Next, we expand each of the terms appearing on the right in terms of the m^{th} spatial grid point. Thus,

$$\begin{aligned}
u_{m-1}^{n+1} &= u_m^{n+1} - h(u_x)_m^{n+1} + \frac{h^2}{2}(u_{xx})_m^{n+1} - \frac{h^3}{6}(u_{xxx})_m^{n+1} + \frac{h^4}{24}(u_{xxxx})_m^{n+1} - \cdots \\
(u_t)_{m-1}^{n+1} &= (u_t)_m^{n+1} - h(u_{tx})_m^{n+1} + \frac{h^2}{2}(u_{txx})_m^{n+1} - \frac{h^3}{6}(u_{txxx})_m^{n+1} + \cdots \\
(u_{tt})_{m-1}^{n+1} &= (u_{tt})_m^{n+1} - h(u_{ttx})_m^{n+1} + \frac{h^2}{2}(u_{ttxx})_m^{n+1} - \cdots \\
(u_{ttt})_{m-1}^{n+1} &= (u_{ttt})_m^{n+1} - h(u_{tttx})_m^{n+1} + \cdots
\end{aligned}$$

Henceforth we suppress the $(\,\cdot\,)_m^{n+1}$ notation, and only explicitly denote grid-point values having other indices. Then

$$\begin{aligned}
u_{m-1}^{n+1} + u_{m-1}^n = {} & 2\left(u - hu_x + \frac{h^2}{2}u_{xx} - \frac{h^3}{6}u_{xxx} + \frac{h^4}{24}u_{xxxx} - \cdots\right) \\
& - k\left(u_t - hu_{tx} + \frac{h^2}{2}u_{txx} - \frac{h^3}{6}u_{txxx} + \cdots\right) \\
& + \frac{k^2}{2}\left(u_{tt} - hu_{ttx} + \frac{h^2}{2}u_{ttxx} - \cdots\right) \\
& - \frac{k^3}{6}\left(u_{ttt} - hu_{tttx} + \cdots\right) \ .
\end{aligned}$$

After similar manipulations we also obtain

$$\begin{aligned} u_{m+1}^{n+1} + u_{m+1}^{n} = 2 &\left(u + hu_x + \frac{h^2}{2}u_{xx} + \frac{h^3}{6}u_{xxx} + \frac{h^4}{24}u_{xxxx} + \cdots \right) \\ &- k\left(u_t + hu_{tx} + \frac{h^2}{2}u_{txx} + \frac{h^3}{6}u_{txxx} + \cdots \right) \\ &+ \frac{k^2}{2}\left(u_{tt} + hu_{ttx} + \frac{h^2}{2}u_{ttxx} + \cdots \right) \\ &- \frac{k^3}{6}\left(u_{ttt} + hu_{tttx} + \cdots \right) . \end{aligned}$$

In addition, we have

$$-\left(2 + \frac{1}{r}\right)u - \left(2 - \frac{1}{r}\right)u_m^n = -4u + \left(2 - \frac{1}{r}\right)\left(ku_t - \frac{k^2}{2}u_{tt} + \frac{k^3}{6}u_{ttt} - \cdots \right) .$$

We now combine these results and use the definition of r to obtain, after some rearrangement,

$$u_t - u_{xx} - \frac{h^2}{12}u_{xxxx} + \frac{k^2}{6}u_{ttt} - \frac{k^2}{4}u_{ttxx} + \frac{k}{2}(u_{txx} - u_{tt}) + \cdots = 0 .$$

Finally, we observe that since $u_t = u_{xx}$, it follows that $u_{tt} = u_{xxt}$, and if u is sufficiently smooth $u_{xxt} = u_{txx}$. Hence, in this case the $\mathcal{O}(k)$ term is identically zero, and since $u_{ttxx} = u_{ttt}$, the $\mathcal{O}(k^2)$ terms can be combined so that the above becomes

$$(u_t - u_{xx})_m^{n+1} = \frac{1}{12}\left[(u_{xxxx})_m^{n+1}h^2 + (u_{ttt})_m^{n+1}k^2 \right] \sim \mathcal{O}(h^2 + k^2) . \tag{5.29}$$

The right-hand side of Eq. (5.29) is the global truncation error; it clearly goes to zero as $h, k \to 0$, proving consistency of the Crank–Nicolson scheme. It should be clear at this point that our ODE analogy breaks down to some extent because of the additive nature of the different portions of the truncation error. In particular, letting only $k \to 0$, but with h finite, will <u>not</u> result in the exact solution to a PDE problem unless the solution happens to be such that u_{xxxx} and all higher derivatives are identically zero (as, for example, with a low-degree polynomial for the true solution). In fact, we would obtain the exact solution to a different problem,

$$u_t = u_{xx} + T(h) ,$$

where $T(h)$ is the spatial truncation error due to a step size h. It is important to keep this point in mind when performing grid-function convergence tests for PDEs because if h remains fixed while k is decreased, convergence will occur, but not to the true solution of the problem. This can often lead to seemingly ambiguous results. For a method with truncation error $\mathcal{O}(h^2+k^2)$, k and h should be simultaneously reduced by the same factor during convergence tests. Analogous arguments and conclusions apply if $h \to 0$ with k finite.

We next briefly consider a stability analysis for the Crank–Nicolson procedure. We would expect, because trapezoidal integration is A-stable, that the Crank–Nicolson scheme should be unconditionally stable for linear constant-coefficient problems. This is, in fact, the case as we will show. We apply the von Neumann analysis discussed earlier, but once again note that this does not account for boundary conditions. Again letting

$$z_m^n = \xi^n e^{i\beta mh}$$

denote the difference between the true solution and the computed solution to (5.27), we obtain

$$\xi = \frac{2 - \frac{1}{r} - 2\cos\beta h}{-\left(2 + \frac{1}{r}\right) + 2\cos\beta h} \ .$$

Recalling that the von Neumann stability condition (which is both necessary and sufficient in this case) is $|\xi| \leq 1$, we see that the following two inequalities must be satisfied:

$$\frac{2 - \frac{1}{r} - 2\cos\beta h}{-\left(2 + \frac{1}{r}\right) + 2\cos\beta h} \leq 1 \ , \qquad \text{and} \qquad \frac{2 - \frac{1}{r} - 2\cos\beta h}{\left(2 + \frac{1}{r}\right) - 2\cos\beta h} \geq 1 \ .$$

It is a simple exercise, left to the reader, to show that both of these hold, completely independent of the value of r. Hence, the Crank–Nicolson method applied to the linear heat equation is unconditionally stable.

Finally, we remark that this result is not significantly altered by the presence of inhomogeneities; however, in such cases we employ a more general concept of stability. Namely, because the true solutions may grow as fast as exponentially with time, it is too restrictive to require $|\xi| \leq 1$. Instead, generally for nonhomogeneous problems, we only require

$$|\xi| \leq 1 + \mathcal{O}(k)$$

for stability.

We conclude this section on second-order methods for parabolic equations with a pseudo-language algorithm for the trapezoidal method. Before presenting this we provide a slightly generalized version of Eq. (5.27) allowing us to write a single code that embodies all of forward- and backward-Euler, and trapezoidal, methods for initial boundary value problems for the heat equation. This is done by viewing the construction of the trapezoidal method as consisting of a simple average of values between the n and $n+1$ time levels, and replacing this with a weighted average. If we denote this weight by θ, $(0 \leq \theta \leq 1)$, then the basic trapezoidal integration formula is replaced by the more general form

$$u_m^{n+1} = u_m^n + \frac{k}{h^2}\left[\theta\left(u_{m-1} - 2u_m + u_{m+1}\right)^{n+1} + (1-\theta)\left(u_{m-1} - 2u_m + u_{m+1}\right)^n\right] \ .$$

It is easily seen that $\theta = \frac{1}{2}$ yields the usual trapezoidal formula. If we set $\theta = 1$, we obtain the backward-Euler method, while $\theta = 0$ results in the forward-Euler approximation.

One often encounters the terminology "semi-implicit", "implicit" and "explicit" for these three cases, but the first of these is misleading because as will be evident from the pseudo-languange algorithm that follows, the trapezoidal method is no less implicit than is the backward-Euler method: the entire spatial grid function is computed simultaneously (implicitly) at each time step in both approaches. They do, however, exhibit different behavior for large values of time step k. In particular, despite the fact that trapezoidal integration is A-stable, if k is too large solutions will oscillate (boundedly) from one time step to the next. For this reason backward Euler is sometimes preferred even though it is only first-order accurate in time. An instance of this is the case of calculating a steady-state solution by integrating a formally time-dependent equation to steady state, a widely-used practice in computational fluid dynamics and heat transfer.

If we define

$$r_1 \equiv \frac{\theta k}{h^2} \ , \qquad r_2 \equiv \frac{(1-\theta)k}{h^2} \ ,$$

then we can express the above equation in a form analogous to that of Eq. (5.27):

$$u_{m-1}^{n+1} - \left(2 + \frac{1}{r_1}\right) u_m^{n+1} + u_{m+1}^{n+1} = -\frac{r_2}{r_1} u_{m-1}^n + \left(\frac{2r_2}{r_1} - \frac{1}{r_1}\right) u_m^n - \frac{r_2}{r_1} u_{m+1}^n \ . \tag{5.30}$$

We now present the pseudo-language algorithm for implementing this generalized trapezoidal method.

Algorithm 5.2 *(Generalized Trapezoidal Method)*

1. *Input number of time steps N_t, number of spatial grid points N_x, time step size k, and endpoints of spatial domain, a_x, b_x, time integration weight, θ.*
2. *Input initial data, $u_m^0,\ m = 1, \ldots, N_x$, and initial time, t^0.*
3. *Calculate grid spacing: $h = (b_x - a_x)/(N_x - 1)$, time integration parameters, $r_1 = \theta * k/h^2$, $r_2 = (1-\theta) * k/h^2$.*
4. *Begin time stepping*

   Do $n = 1, N_t$
   $t^n = k * n + t^0$
   [Load tridiagonal matrix and right-hand side]
   $A_{1,1} = 0$
   $A_{2,1} = 1$
   $A_{3,1} = 0$
   $B_1 = Lftbndry(t^n)$
   Do $m = 2, N_x - 1$
   $A_{1,m} = 1$
   $A_{2,m} = -\left(2 + \frac{1}{r_1}\right)$
   $A_{3,m} = 1$
   $B_m = -\frac{r_2}{r_1}\left(u_{m-1}^{n-1} + u_{m+1}^{n-1}\right) + \left(\frac{2r_2}{r_1} - \frac{1}{r_1}\right) u_m^{n-1}$
   Repeat m
   $A_{1,N_x} = 0$
   $A_{2,N_x} = 1$
   $A_{3,N_x} = 0$
   $B_{N_x} = Rhtbndry(t^n)$
   [Solve tridiagonal system]
   Call $LUDCMP(A, B, N_x)$
   [Store current time level solution]
   Do $m = 1, N_x$
   $u_m^n = B_m$
   Repeat m
   [Go to next time step] Repeat n
   End

   FUNCTION Lftbndry(t)

Lftbndry = formula for time-dependent left boundary condition
Return
End

FUNCTION Rhtbndry(t)

Rhtbndry = formula for time-dependent right boundary condition
Return
End

5.3.4 Peaceman–Rachford Alternating-Direction-Implicit Scheme

Our final topic in this study of parabolic equations is treatment of the two-dimensional heat equation. We first point out that, generally, explicit methods should not be used because the stability requirements are even more stringent for 2-D algorithms than in the 1-D case, leading to usually unacceptable increases in required arithmetic to integrate to a desired final time. Thus, we will restrict attention to implicit methods. In fact, we shall consider only the 2-D analogue of the Crank–Nicolson method although a similar approach based on backward-Euler time integration is also widely used.

The problem we study is the 2-D heat equation on the unit square,

$$u_t = u_{xx} + u_{yy} + f(x,y,t) \ , \quad (x,y) \in (0,1) \times (0,1) \equiv \Omega \ , \quad t \in (0,T] \ , \tag{5.31}$$

with prescribed Dirichlet conditions on the boundaries, $\partial\Omega$, and given initial data on $\overline{\Omega} \equiv \Omega \cup \partial\Omega$. If we apply the trapezoidal integration to (5.31) we obtain for interior grid points

$$u_{l,m}^{n+1} = u_{l,m}^n + \frac{k}{2}\left[(u_{xx}+u_{yy})_{l,m}^{n+1} + (u_{xx}+u_{yy})_{l,m}^n + f_{l,m}^{n+1} + f_{l,m}^n\right] + \mathcal{O}(k^3) \, , \tag{5.32}$$

which is the 2-D Crank–Nicolson method.

It is worthwhile to construct and examine the matrix of difference equation coefficients for this 2-D problem because this will highlight some of the difficulties that arise in the solution process, and motivate a specific treatment. We rewrite (5.32) as

$$u_{l,m}^{n+1} - \frac{k}{2}(u_{xx}+u_{yy})_{l,m}^{n+1} = u_{l,m}^n + \frac{k}{2}(u_{xx}+u_{yy})_{l,m}^n + \frac{k}{2}\left(f_{l,m}^{n+1} + f_{l,m}^n\right) \ ,$$

and then replace spatial derivatives with centered differences, constructed with uniform grid spacing h in both x and y directions, to obtain

$$\begin{aligned} u_{l,m}^{n+1} - \frac{k}{2h^2}\left(u_{l-1,m}^{n+1} + u_{l,m-1}^{n+1} - 4u_{l,m}^{n+1} + u_{l,m+1}^{n+1} + u_{l+1,m}^{n+1}\right) \\ = u_{l,m}^n + \frac{k}{2h^2}\left(u_{l-1,m}^n + u_{l,m-1}^n - 4u_{l,m}^n + u_{l,m+1}^n + u_{l+1,m}^n\right) + \frac{k}{2}(f_{l,m}^{n+1} + f_{l,m}^n). \end{aligned} \tag{5.33}$$

If we suppose that the spatial grid consists of N_x points in the x direction, and N_y in the y direction (with $N_x = N_y$ so that $h_x = h_y = h$), then (5.33) holds for all $l = 2,3,\ldots,N_x - 1$ and $m = 2,3,\ldots,N_y - 1$. For a Dirichlet problem, if $l = 1$ or N_x, or $m = 1$ or N_y, we replace (5.33) with the appropriate Dirichlet condition,

$$u_{l,m} = \sigma(x_l, y_m, t) \ , \tag{5.34}$$

where $\sigma(x,y,t)$ is a prescribed boundary function. We note that boundary condition evaluation for the left-hand side of (5.33) must be at time level $n+1$, while that on the right-hand side must be at time level n. Failure to adhere to this will result in degradation of formal—and observed—accuracy.

Before writing the final form of the difference equations we again introduce the notation $r \equiv k/2h^2$. We now divide by $-r$ in (5.33) and obtain

$$u^{n+1}_{l-1,m} + u^{n+1}_{l,m-1} - \left(4 + \frac{1}{r}\right) u^{n+1}_{l,m} + u^{n+1}_{l,m+1} + u^{n+1}_{l+1,m}$$
$$= -u^n_{l-1,m} - u^n_{l,m-1} + \left(4 - \frac{1}{r}\right) u^n_{l,m} - u^n_{l,m+1} - u^n_{l+1,m} - h^2(f^{n+1}_{l,m} + f^n_{l,m}).$$

We observe that everything on the right-hand side is known prior to the start of calculations for time level $n+1$, so we define

$$F^n_{l,m} \equiv -u^n_{l-1,m} - u^n_{l,m-1} + \left(4 - \frac{1}{r}\right) u^n_{l,m} - u^n_{l,m+1} - u^n_{l+1,m} - h^2(f^{n+1}_{l,m} + f^n_{l,m}).$$

Hence, the difference equations at interior grid points can be expressed simply as

$$u^{n+1}_{l-1,m} + u^{n+1}_{l,m-1} - \left(4 + \frac{1}{r}\right) u^{n+1}_{l,m} + u^{n+1}_{l,m+1} + u^{n+1}_{l+1,m} = F^n_{l,m} \ , \tag{5.35}$$

$\forall\ l = 2,3,\ldots,N_x - 1$ and $m = 2,3,\ldots,N_y - 1$. Clearly, (5.35) is of the form

$$A_{1_{l,m}} u^{n+1}_{l-1,m} + A_{2_{l,m}} u^{n+1}_{l,m-1} + A_{3_{l,m}} u^{n+1}_{l,m} + A_{4_{l,m}} u^{n+1}_{l,m+1} + A_{5_{l,m}} u^{n+1}_{l+1,m} = b_{l,m} \ . \tag{5.36}$$

Moreover, (5.34) is also of this form if we set $A_{3_{l,m}} = 1, A_{i_{l,m}} = 0, i \neq 3$, and $b_{l,m} = \sigma^{n+1}_{l,m}$. The complete matrix associated with this system of difference equations has the structure displayed in Fig. 5.3.

This matrix consists of only five nonzero bands. However, the matrix structure shown is such that this high degree of sparsity cannot be readily exploited. Thus, usual direct elimination cannot be applied. It is the presence of the two outer bands, symmetrically spaced N_y elements from the main diagonal, that prevents use of sparse LU decomposition. It turns out, however, that with a little manipulation, the original system of difference equations can be converted to a system of two sets of equations, each of which involves only one spatial direction, and requires solution of only tridiagonal systems. Such procedures are described generically as alternating-direction-implicit (ADI) methods. They require only $\mathcal{O}(N)$ ($N = N_x \times N_y$) arithmetic operations per time step, in contrast to the $\mathcal{O}\left(N^3\right)$ required by direct Gaussian elimination applied to the entire system.

To derive the ADI scheme to be used here, we observe that the approximation (5.33) can be rearranged and represented as

$$\left[I - \frac{k}{2}(D^2_{0,x} + D^2_{0,y})\right] u^{n+1}_{l,m} = \left[I + \frac{k}{2}(D^2_{0,x} + D^2_{0,y})\right] u^n_{l,m} + \frac{k}{2}(f^{n+1}_{l,m} + f^n_{l,m}) \tag{5.37}$$

$\forall\ l = 2,\ldots,N_x - 1,\ m = 2,\ldots,N_y - 1$. Now each of $D^2_{0,x}u_{l,m}$ and $D^2_{0,y}u_{l,m}$ is just a tridiagonal matrix times the vector grid function $\{u_{l,m}\}$, as we have already seen in the study of ODE BVPs. For notational simplicity, label these matrices as A_x and A_y, respectively, and write the above as

$$\left[I - \frac{k}{2}(A_x + A_y)\right] u^{n+1} = \left[I + \frac{k}{2}(A_x + A_y)\right] u^n + \frac{k}{2}(f^{n+1} + f^n)\,. \tag{5.38}$$

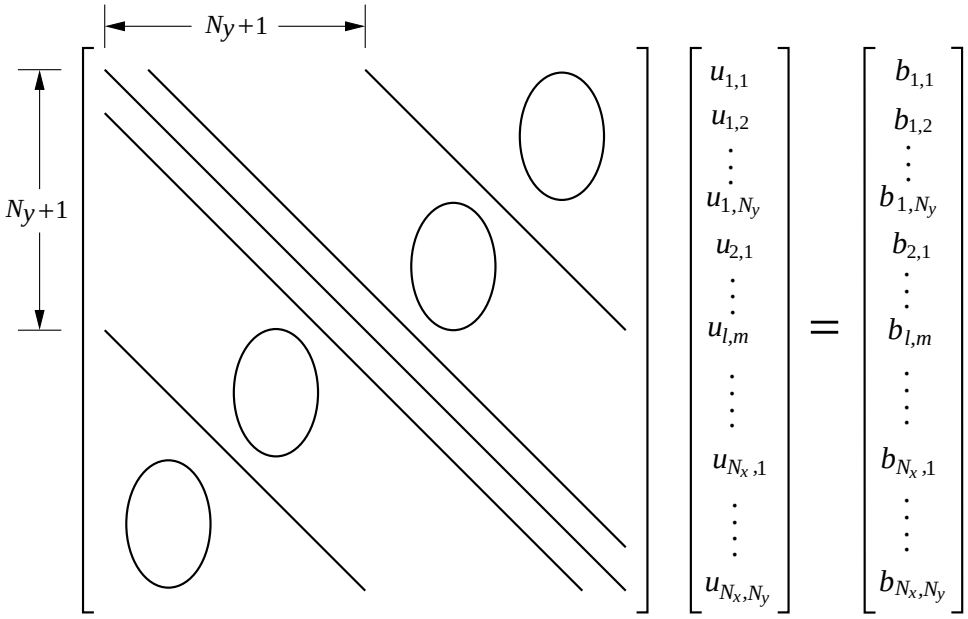


Figure 5.3: Matrix structure for 2-D Crank–Nicolson method.

We next observe that the left-hand side matrix can be approximately factored as

$$\left(I - \frac{k}{2}A_x\right)\left(I - \frac{k}{2}A_y\right) = I - \frac{k}{2}(A_x + A_y) + \frac{k^2}{4}A_xA_y \ ,$$

and similarly for the right-hand side,

$$\left(I + \frac{k}{2}A_x\right)\left(I + \frac{k}{2}A_y\right) = I + \frac{k}{2}(A_x + A_y) + \frac{k^2}{4}A_xA_y \ ,$$

with the last term on the right in each of these expressions being additional truncation error arising from the approximate factorization. Notice that each of the two factored matrices on the left of these expressions is tridiagonal. Moreover, their products are within $\mathcal{O}\left(k^2\right)$ of the original, unfactored matrices.

We now substitute these into (5.38) obtaining the result

$$\left(I - \frac{k}{2}A_x\right)\left(I - \frac{k}{2}A_y\right)u^{n+1} = \left(I + \frac{k}{2}A_x\right)\left(I + \frac{k}{2}A_y\right)u^n + \frac{k}{2}(f^{n+1} + f^n) + \frac{k^2}{4}A_xA_y(u^{n+1} - u^n) \ .$$

Finally, we observe that if $u(x, y, t)$ is sufficiently smooth,

$$u^{n+1}_{l,m} - u^n_{l,m} = \mathcal{O}(k) \qquad \forall \ \ l, m$$

by the mean value theorem. Hence, the factorization,

$$\left(I - \frac{k}{2}A_x\right)\left(I - \frac{k}{2}A_y\right)u^{n+1} = \left(I + \frac{k}{2}A_x\right)\left(I + \frac{k}{2}A_y\right)u^n + \frac{k}{2}(f^{n+1} + f^n) \tag{5.39}$$

is within $\mathcal{O}\left(k^3\right)$ of the original 2-D Crank–Nicolson scheme. That is, no significant formal error has been introduced by the factorization. However, we must also observe that our goal of replacing

the Crank–Nicolson procedure with a system involving only tridiagonal matrices has not yet been achieved because (5.39) contains matrix products—not single tridiagonal matrices.

The standard remedy is to split (5.39) into two equations as follows:

$$\left(I - \frac{k}{2}A_x\right) u^{n+1^*} = \left(I + \frac{k}{2}A_y\right) u^n + \frac{k}{2} f^n \,, \tag{5.40a}$$

$$\left(I - \frac{k}{2}A_y\right) u^{n+1} = \left(I + \frac{k}{2}A_x\right) u^{n+1^*} + \frac{k}{2} f^{n+1} \,. \tag{5.40b}$$

Thus, we calculate the advanced time step grid-function values in two consecutive steps. The first involves only x derivatives at the advanced time level (*i.e.*, at $n+1^*$), while the second involves only y derivatives. In each case the advanced time values are calculated implicitly, thus giving rise to the name "alternating direction implicit," or ADI. There are many such procedures, and the one presented here is among the oldest, but still widely used, due to Peaceman and Rachford [23].

It is rather common practice to view u^{n+1^*} as being equivalent to $u^{n+\frac{1}{2}}$. In general, this is not true for ADI-like schemes. In particular, for problems involving time-dependent boundary conditions, second-order temporal accuracy can be lost if boundary conditions for the first split step are evaluated at $t = t^{n+\frac{1}{2}}$. The correct implementation of time-dependent Dirichlet conditions is presented in Mitchell and Griffiths [21]; we shall not pursue this topic further herein except to mention that in the specific case of Peaceman–Rachford ADI presented here, numerical tests generally indicate no loss of formal accuracy when $u^{n+1^*} = u^{n+\frac{1}{2}}$ is used.

Formal Accuracy of Peaceman–Rachford ADI

We will now demonstrate formal accuracy of the Peaceman–Rachford scheme for the case $f \equiv 0$. Thus, we consider

$$\left(I - \frac{k}{2}A_x\right) u^{n+1^*} = \left(I + \frac{k}{2}A_y\right) u^n \,, \tag{5.41a}$$

$$\left(I - \frac{k}{2}A_y\right) u^{n+1} = \left(I + \frac{k}{2}A_x\right) u^{n+1^*} \,. \tag{5.41b}$$

If we "solve" the first of these for u^{n+1^*}, and substitute the result into the second, we obtain

$$\left(I - \frac{k}{2}A_y\right) u^{n+1} = \left(I + \frac{k}{2}A_x\right) \left(I - \frac{k}{2}A_x\right)^{-1} \left(I + \frac{k}{2}A_y\right) u^n \,.$$

We now multiply by $I - \frac{k}{2}A_x$, and note that the matrices $\left(I + \frac{k}{2}A_x\right)$ and $\left(I - \frac{k}{2}A_x\right)$ commute, leading to

$$\left(I - \frac{k}{2}A_x\right) \left(I - \frac{k}{2}A_y\right) u^{n+1} = \left(I + \frac{k}{2}A_x\right) \left(I + \frac{k}{2}A_y\right) u^n \,.$$

This is exactly the unsplit scheme, (5.39), with $f \equiv 0$. Hence, for constant coefficient homogeneous problems, second-order global accuracy is achieved up to implementation of boundary conditions. In fact, this is still true for $f \not\equiv 0$ (and even time-dependent); but this is somewhat more tedious to demonstrate, and we leave this to the ambitious reader.

Stability of Peaceman–Rachford ADI

We next consider stability of the Peaceman–Rachford scheme. We shall not carry out a rigorous analysis, but instead give an heuristic treatment that is somewhat specific to the 2-D heat equation.

For this particular case, it is easily seen that $A_x = A_y$ up to multiplication by a permutation matrix. We then recognize that each step of (5.41) corresponds to a 1-D Crank–Nicolson procedure, which previously was shown to be unconditionally stable in the sense of von Neumann. Hence, if we ignore boundary conditions, we can infer that for $z_{l,m}^n$ defined as in (5.15),

$$\left|z_{l,m}^{n+1^*}\right| = |\xi_x| \left|z_{l,m}^n\right| \ ,$$

and we have $|\xi_x| \leq 1$ for the x-direction step. Similarly, for the y direction we have

$$\left|z_{l,m}^{n+1}\right| = |\xi_y| \left|z_{l,m}^{n+1^*}\right|$$

with $|\xi_y| \leq 1$. Thus, it follows that

$$\begin{aligned} \left|z_{l,m}^{n+1}\right| &= |\xi_y| \, |\xi_x| \left|z_{l,m}^n\right| \\ &= |\xi| \left|z_{l,m}^n\right| \ , \end{aligned}$$

with $|\xi| = |\xi_x \xi_y| \leq 1$. Hence, the von Neumann stability criterion is satisfied unconditionally for the Peaceman–Rachford ADI method in the form considered here. It is clear for constant Dirichlet conditions considered alone (without the differential operator of the problem) that $\xi = 1$, so boundary points are also stable for this case. Moreover, it can be shown via Taylor expansion that $|\xi| = 1 + \mathcal{O}(k)$ in the nonconstant Dirichlet case provided the boundary function is in C^1.

In summary, we have argued that for the homogeneous heat equation with Dirichlet boundary conditions, Peaceman–Rachford ADI is second-order accurate in both space and time (for constant boundary conditions—in fact, with some effort it can be proven for time-dependent boundary conditions) and unconditionally stable.

Implementation of Peaceman–Rachford ADI

We will now consider some implementational details. Since $A_x = D_{0,x}^2$ and $A_y = D_{0,y}^2$, Eq. (5.40a) written at the (l, m) grid point is

$$u_{l,m}^{n+1^*} - \frac{k}{2h^2}\left(u_{l-1,m}^{n+1^*} - 2u_{l,m}^{n+1^*} + u_{l+1,m}^{n+1^*}\right) = u_{l,m}^n + \frac{k}{2h^2}\left(u_{l,m-1}^n - 2u_{l,m}^n + u_{l,m+1}^n\right) + \frac{k}{2} f_{l,m}^n \ .$$

As we have done previously for the Crank–Nicolson scheme, we set

$$r = \frac{k}{2h^2}$$

and divide by $-r$ to obtain

$$u_{l-1,m}^{n+1^*} - \left(2 + \frac{1}{r}\right) u_{l,m}^{n+1^*} + u_{l+1,m}^{n+1^*} = -u_{l,m-1}^n + \left(2 - \frac{1}{r}\right) u_{l,m}^n - u_{l,m+1}^n - h^2 f_{l,m}^n \, . \tag{5.42}$$

Analogous rearrangement of (5.40b) leads to

$$u_{l,m-1}^{n+1} - \left(2 + \frac{1}{r}\right) u_{l,m}^{n+1} + u_{l,m+1}^{n+1} = -u_{l-1,m}^{n+1^*} + \left(2 - \frac{1}{r}\right) u_{l,m}^{n+1^*} - u_{l+1,m}^{n+1^*} - h^2 f_{l,m}^{n+1} \, . \tag{5.43}$$

We remark that a similar treatment can be carried out when the grid spacing h is different in the two different directions, say h_x and h_y, but the resulting equations are considerably more tedious. We leave analysis of this as an exercise for the reader.

There are several things to notice regarding Eqs. (5.42) and (5.43). The first is that each of these separate steps is formally very similar to the 1-D Crank–Nicolson scheme, a fact we have already used to make plausible the unconditional stability of the Peaceman–Rachford procedure. The second is that (5.42) holds for each $m = 1, 2, \ldots, N_y$ except when Dirichlet conditions are imposed at one, or both, y-direction boundaries. In particular, for each value of the index m, (5.42) results in a tridiagonal system of N_x equations for the grid function values $\{u_{1,m}^{n+1^*}, u_{2,m}^{n+1^*}, \ldots, u_{l,m}^{n+1^*}, \ldots, u_{N_x,m}^{n+1^*}\}$. Hence, to compute the complete grid function $\left\{u_{l,m}^{n+1^*}\right\}$ we use $\mathcal{O}(N_x N_y) \sim \mathcal{O}(N)$ arithmetic operations.

Construction of the right-hand sides of (5.42) deserves some special attention. We first note that all quantities are known either from previous computation or from given data; but some care must be exercised in loading results from previous calculations. In particular, we must be sure to use only results from time level n and <u>not</u> recently-calculated grid-function values from $n + 1^*$ in constructing the right-hand side of (5.42). Figure 5.4 indicates why this is a concern. At the m^{th}

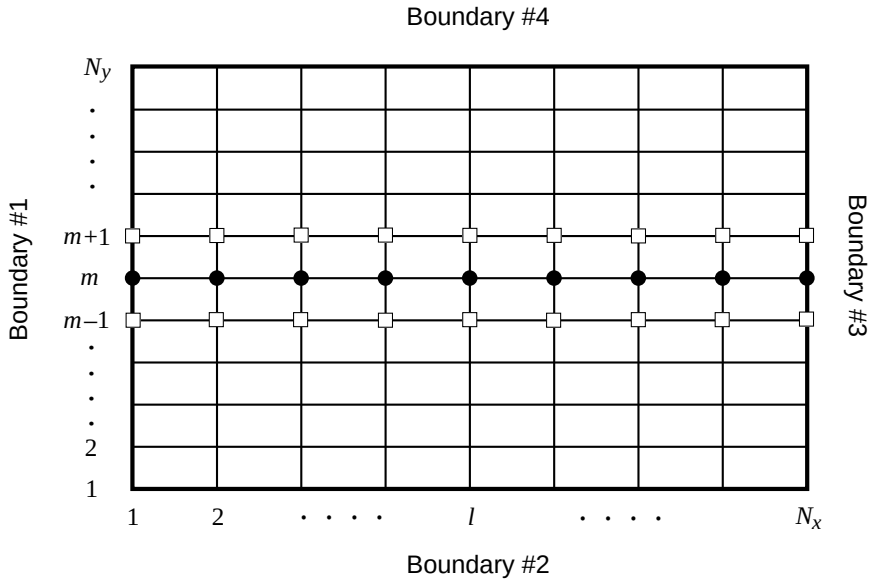


Figure 5.4: Implementation of Peaceman–Rachford ADI.

row of grid points we will solve for grid function values at time level $n+1^*$ at locations indicated by dots. At each of these points we must evaluate Eq. (5.42). But the right-hand side of this equation requires points, indicated by squares corresponding to time level n. Thus, values $\left\{u_{l,m-1}^{n}\right\}$ and $\left\{u_{l,m+1}^{n}\right\}$ are needed. In computer implementations we would almost always load the $n+1^*$ values over the old n-level ones to reduce memory requirements. But this means that memory locations corresponding to $u_{l,m-1}^{n}$ will actually contain $u_{l,m-1}^{n+1^*}$ leading to incorrect right-hand sides in (5.42). This implies that we must keep a temporary storage vector of length N_x (actually, $\max(N_x, N_y)$) so that $u_{l,m-1}^{n}$ can be saved for use at row m when results for the $n + 1^*$ level are loaded over $u_{l,m-1}^{n}$. Failure to do this will result in loss of temporal accuracy.

Precisely the same treatment is employed in solving for the $n + 1$ level values $\left\{u_{l,m}^{n+1}\right\}$. In this case, again the right-hand side contains only known information, and all of the grid function values are obtained from N_x systems, Eq. (5.43), containing N_y equations each. From this we

conclude that the total arithmetic per time step is $\mathcal{O}(N_x N_y)$, and since $N = N_x N_y$ is the length of the solution vector, the Peaceman–Rachford scheme requires only $\mathcal{O}(N)$ arithmetic operations per time step.

We now present a pseudo-language algorithm for implementing the Peaceman–Rachford scheme.

Algorithm 5.3 *(Peaceman–Rachford ADI Method for Dirichlet Problems)*

1. *Read inputs:* $N_t,\ N_x,\ N_y,\ k,\ h,$ *and initial data*
2. *Calculate* $r = \frac{k}{2h^2}$*; initiate time step index:* $n = 0$
3. *Print inputs*
4. *Increment time:* $t^n = t^0 + n * k$
5. *Calculate* $n + 1^*$ *intermediate time level results*
   Do $m = 1, N_y$
   If $m = 1$ *then*

   Do $l = 1, N_x$
   $u_{sv,l} = u^n_{l,m}$
   $u^*_{l,m} = Bndry\left(x_l, y_m, t^n + \frac{k}{2}, 2\right)$
   Repeat l

   Else if $m = N_y$

   Do $l = 1, N_x$
   $u^*_{l,m} = Bndry\left(x_l, y_m, t^n + \frac{k}{2}, 4\right)$
   Repeat l

   Else

   Do $l = 1, N_x$
   If $l = 1$ *or* $l = N_x$ *then*

   $A_{1,l} = 0$
   $A_2, l = 1$
   $A_3, l = 0$
   If $l = 1, B_1 = Bndry\left(x_l, y_m, t^n + \frac{k}{2}, 1\right)$
   If $l = N_x, B_{N_x} = Bndry\left(x_l, y_m, t^n + \frac{k}{2}, 3\right)$

   Else

   $A_{1,l} = 1$
   $A_{2,l} = -\left(2 + \frac{1}{r}\right)$
   $A_{3,l} = 1$
   $B_l = -u_{sv,l} + \left(2 - \frac{1}{r}\right) u^n_{l,m} - u^n_{l,m+1} - h^2 fn\left(x_l, y_m, t^n\right)$

   End if
   Repeat l
   Call $LUDCMP\ (A, B, N_x)$
   Do $l = 1, N_x$

   $u_{sv,l} = u^n_{l,m}$
   $u^{n+1^*}_{l,m} = B_l$

Repeat l

End if
Repeat m

6. *Calculate* $n+1$ *time level*
Do $l = 1, N_x$
If $l = 1$ *then*

$Do\ m = 1, N_y$
$u_{sv,m} = u_{l,m}^{n+1^*}$
$u_{l,m}^{n+1} = Bndry\left(x_l, y_m, t^n + k, 1\right)$
Repeat m

Else If $l = N_x$ *then*

$Do\ m = 1, N_y$
$u_{l,m}^{n+1} = Bndry\left(x_l, y_m, t^n + k, 3\right)$
Repeat m

Else

$Do\ m = 1, N_y$
If $m = 1$ *or* $m = N_y$ *then*

$A_{1,m} = 0$
$A_{2,m} = 1$
$A_{3,m} = 0$
If $m = 1$, $B_1 = Bndry\left(x_l, y_m, t^n + k, 2\right)$
If $m = N_y$, $B_{N_y} = Bndry\left(x_l, y_m, t^n + k, 4\right)$

Else

$A_{1,m} = 1$
$A_{2,m} = -\left(2 + \frac{1}{r}\right)$
$A_{3,m} = 1$
$B_m = -u_{sv,m} + \left(2 - \frac{1}{r}\right) u_{l,m}^{n+1^*} - u_{l+1,m}^{n+1^*} - h^2 fn\left(x_l, y_m, t^n + k\right)$

End if
Repeat m
Call $LUDCMP\ (A, B, N_y)$
$Do\ m = 1, N_y$

$u_{sv,m} = u_{l,m}^{n+1^*}$
$u_{l,m}^{n+1} = B_m$

Repeat m

End if
Repeat l
Print results for $t = t^{n+1}$

7. *Increment* n

$n = n + 1$
If $n \le N_t$, *then go to* 4
else stop

Although the preceding algorithm has been constructed specifically for the heat equation, the extension to more general 2-D parabolic problems is straightforward. On the other hand, the Peaceman–Rachford procedure cannot be implemented for 3-D problems in any direct manner. Nevertheless, there are numerous related methods that can be applied in this case. We refer the reader to Douglas and Gunn [7] for a very general treatment.

In closing this section on the Peaceman–Rachford ADI method we should point out that in recent years iterative techniques have often been employed to solve the five-band system shown in Fig. 5.3, especially in FEMs. It is important to recognize, however, that such approaches cannot possibly compete with the Peaceman–Rachford scheme (or other similar "time-splitting" methods) in terms of total required arithmetic. Moreover, ADI schemes in general are highly parallelizable, so they present several distinct advantages on modern multiprocessors that are difficult to realize with various alternative approaches.

5.4 Elliptic Equations

Recall that, as discussed earlier, constant coefficient elliptic PDEs in two space dimensions can always be reduced to the canonical form

$$u_{xx} + u_{yy} + Au = f(x, y) \ , \tag{5.44}$$

where $A = -1, 0$ or $+1$. In the present treatment we will consider only the case $A = 0$, so that (5.44) becomes a Poisson equation. Moreover, we will restrict attention to Dirichlet problems on rectangular domains. We remark that the cases $A = \pm 1$ can be treated completely analogously to that considered here, and other types of boundary conditions can be implemented as described in Chap. 4 for ODE problems.

Two classical methods will be applied to the solution of

$$u_{xx} + u_{yy} = f(x, y) \ , \quad (x, y) \in (a_x, b_x) \times (a_y, b_y) \equiv \Omega \tag{5.45}$$

with

$$u(x, y) = g(x, y) \ , \quad (x, y) \in \partial\Omega \ . \tag{5.46}$$

These methods are: *i*) successive overrelaxation (SOR) and *ii*) alternating direction implicit (ADI). We have already encountered forms of both of these methods in our earlier studies. It must be noted that although many other more efficient procedures for solving elliptic problems are now in existence, SOR and ADI still are widely used, and understanding them helps to form the basis for developing more sophisticated modern methods.

We shall begin by discretizing (5.44) in our usual manner, *viz.*, by replacing derivatives with second-order centered differences. Thus, at grid point (i, j) we have

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} = f_{ij} \ ,$$

or (with $h_x = h_y = h$)

$$u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1} = h^2 f_{ij} \tag{5.47}$$

$\forall\ i = 2, 3, \ldots, N_x - 1$ and $j = 2, 3, \ldots, N_y - 1$. Boundary conditions can be handled exactly as they are for ODE two-point BVPs. Namely, at each boundary point, instead of (5.47) we have

$$\begin{aligned} u_{ij} = g_{ij}\ , \quad & i = 1, N_x \ \text{ with } \ j = 1, 2, \ldots, N_y\ , \\ & j = 1, N_y \ \text{ with } \ i = 1, 2, \ldots, N_x\ . \end{aligned} \tag{5.48}$$

This set of equations can be formally solved with the system (5.47), or it can be used to eliminate terms from those equations in (5.47) corresponding to points adjacent to boundaries. For general codes, written to permit other boundary condition types as well as Dirichlet conditions, the former is preferred, as has been noted previously.

We leave formal truncation error analysis of this discretization as an exercise for the reader. Our intuition should suggest (correctly) that the dominant truncation error is $\mathcal{O}(h^2)$ for sufficiently-smooth solutions $u(x, y)$. Moreover, because the problem being considered is independent of time, and correspondingly the difference approximation is not of evolution type, stability of the approximation is not an issue.

The matrix form of (5.47), (5.48) is presented in Fig. 5.5. The figure shows that the system matrix is sparse, and banded; but as we have previously noted, matrices of this form do not admit a sparse LU decomposition. Thus, in general, if direct elimination methods are employed, obtaining solutions may require as much as $\mathcal{O}(N^3)$ arithmetic operations, where $N = N_x N_y$. In practical problems N can easily be 10^6, or even larger, and it is clear from this that iterative methods usually must be employed. We shall first consider application of SOR, and we then present use of ADI in a pseudo-time marching procedure.

5.4.1 Successive Overrelaxation

Recall from Chap. 1 that SOR is an accelerated form of Gauss–Seidel iteration, which in turn is just a modification of Jacobi iteration. Because of the sparsity of our system matrix, it is better to derive the SOR iteration equations specifically for (5.47), rather than to use the general SOR formula of Chap. 1. Thus, for the $(i, j)^{th}$ equation of (5.47) "solving for u_{ij}" leads to the Jacobi iteration equation

$$u_{ij}^{(m+1)} = \frac{1}{4}\left(u_{i-1,j}^{(m)} + u_{i,j-1}^{(m)} + u_{i,j+1}^{(m)} + u_{i+1,j}^{(m)} - h^2 f_{ij}\right)\ .$$

It is interesting to observe that u_{ij} is computed as simply the average of its four nearest neighbors whenever $f_{ij} = 0$.

Now we modify this iteration by always using the most recent iterate. From the ordering of the u_{ij} shown in Fig. 5.5 we see that when $u_{ij}^{(m+1)}$ is to be computed, $u_{i-1,j}^{(m+1)}$ and $u_{i,j-1}^{(m+1)}$ will already be known. Thus, we arrive at the Gauss–Seidel iteration formula

$$u_{ij}^{(m+1)} = \frac{1}{4}\left(u_{i-1,j}^{(m+1)} + u_{i,j-1}^{(m+1)} + u_{i,j+1}^{(m)} + u_{i+1,j}^{(m)} - h^2 f_{ij}\right)\ .$$

If we denote this result by u_{ij}^*, the SOR formula can be expressed as

$$\begin{aligned} u_{ij}^{(m+1)} &= \omega u_{ij}^* + (1 - \omega) u_{ij}^{(m)} \\ &= u_{ij}^{(m)} + \omega\left(u_{ij}^* - u_{ij}^{(m)}\right) \equiv u_{ij}^{(m)} + \omega \Delta u_{ij}\ , \end{aligned}$$

where ω is the relaxation parameter. (For problems of the type treated herein, $0 < \omega < 2$ must hold.)

$$
\begin{bmatrix}
1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
0 & 1 & 0 & & & & & \ddots & & & & & & & & & & \vdots \\
\vdots & \ddots & \ddots & \ddots & & & & & \ddots & & & & & & & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & & & & & \ddots & & & & & & & & \vdots \\
\vdots & & & 0 & 1 & 0 & & & & & 0 & & & & & & & \vdots \\
\vdots & & & & 0 & 1 & 0 & & & & & 0 & & & & & & \vdots \\
1 & & & & & 1 & -4 & 1 & & & & & 1 & & & & & \vdots \\
\vdots & \ddots & & & & & \ddots & \ddots & \ddots & & & & & \ddots & & & & \vdots \\
\vdots & & \ddots & & & & & \ddots & \ddots & \ddots & & & & & \ddots & & & \vdots \\
\vdots & & & 1 & & & & & 1 & -4 & 1 & & & & & 1 & & \vdots \\
\vdots & & & & \ddots & & & & & \ddots & \ddots & \ddots & & & & & \ddots & \vdots \\
\vdots & & & & & \ddots & & & & & \ddots & \ddots & \ddots & & & & & \vdots \\
\vdots & & & & & & 1 & & & & & 1 & -4 & 1 & & & & \vdots \\
\vdots & & & & & & & 0 & & & & & 0 & 1 & 0 & & & \vdots \\
\vdots & & & & & & & & \ddots & & & & & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & & & & & & & \ddots & & & & & \ddots & \ddots & \ddots & \vdots \\
\vdots & & & & & & & & & & \ddots & & & & & \ddots & \ddots & \vdots \\
\vdots & & & & & & & & & & & \ddots & & & & & \ddots & 0 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1
\end{bmatrix}
\begin{bmatrix} u_{11} \\ \vdots \\ \vdots \\ \vdots \\ u_{1,N_y} \\ u_{21} \\ u_{22} \\ \vdots \\ \vdots \\ \vdots \\ u_{ij} \\ \vdots \\ \vdots \\ \vdots \\ u_{N_x,1} \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_{N_x,N_y} \end{bmatrix}
= h^2
\begin{bmatrix} g_{11}/h^2 \\ \vdots \\ \vdots \\ \vdots \\ g_{1,N_y}/h^2 \\ g_{21}/h^2 \\ f_{22} \\ \vdots \\ \vdots \\ \vdots \\ f_{ij} \\ \vdots \\ \vdots \\ \vdots \\ g_{N_x,1}/h^2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ g_{N_x,N_y}/h^2 \end{bmatrix}
$$

Figure 5.5: Matrix structure of centered discretization of Poisson/Dirichlet problem.

It is well known that performance of SOR is quite sensitive to the value employed for the iteration parameter ω. Optimal values of this parameter can be derived analytically for Poisson/Dirichlet problems posed on rectangular domains (see, *e.g.*, Young [39]). In fact, if the domain is the unit square and the discretization employs uniform grid spacing h in both directions, it can be shown that the optimal value of ω, denoted ω_b, is given as

$$\omega_b = \frac{2}{1 + \sqrt{1 - \cos^2 \pi h}} .$$

The SOR algorithm for solving Poisson equations is the following.

Algorithm 5.4 *(SOR for Elliptic Dirichlet Problems)*

1. *Input* h, *maxit*, ϵ, ω, N_x, N_y
2. *Initialize solution array,* u_{ij}, *with initial guesses. Set iteration counter,* $m = 1$.
3. *Set* $errmx = 0$
4. *Update solution at all grid points*

   Do $i = 1, N_x$

   Do $j = 1, N_y$

   If $m = 1$, *then*

   If $i = 1$ *or* $i = N_x$, $u_{ij} = g_{ij}$

   If $j = 1$ *or* $j = N_y$, $u_{ij} = g_{ij}$

   Else

   $\Delta u = \frac{1}{4}(u_{i-1,j} + u_{i,j-1} + u_{i,j+1} + u_{i+1,j} - h^2 f_{ij}) - u_{i,j}$

   If $|\Delta u| > errmx$, $errmx = |\Delta u|$

   $u_{i,j} = u_{i,j} + \omega \Delta u$

   End if

   Repeat j

   Repeat i
5. *Test convergence*

   If $errmx < \epsilon$, *then stop,*

   else If $m < maxit$, *then* $m = m + 1$

   goto 3

   else stop

We conclude this treatment of SOR by remarking that it is efficient mainly only for constant-coefficient Dirichlet problems. Other problems are readily coded, as can be seen from the simplicity of the algorithm; but convergence can be very slow, if it occurs at all. The convergence rate of SOR depends strongly on the value of the relaxation parameter ω, as already noted. Clearly, this is a disadvantage. For optimal ω, only $\mathcal{O}\left(N^{1.5}\right)$ arithmetic operations are needed in 2D. But for other than optimal ω, as many as $\mathcal{O}\left(N^2\right)$ operations may be required. On the other hand, it is important to recognize that there are more sophistcated versions of SOR (see, *e.g.*, [39]) in which

such difficulties are mitigated to some extent. Furthermore, the operation count per iteration for SOR is far lower than that for essentially any other method, so although the convergence rate of SOR may be inferior to that of other methods, it still is often used because its required total arithmetic is often competitive, and it is easily coded.

5.4.2 The Alternating-Direction-Implicit Scheme

Because of the convergence difficulties often experienced when SOR is applied to more realistic problems, various alternatives have been studied. One of the early more successful of these was the alternating-direction-implicit (ADI) algorithm. The treatment we shall present is very similar to that given earlier for parabolic equations. In fact, we introduce a temporal operator, and formally integrate the resulting time-dependent problem to steady state. Such an approach is sometimes called the method of "false transients." Its efficiency depends upon use of unconditionally stable time-stepping procedures, so that relatively large time steps may be taken. While this would lead to great inaccuracy in an actual transient problem, it is of no concern when only a steady solution is required because all contributions to temporal truncation error vanish at steady state.

We begin again with Eqs. (5.45) and (5.46). But we now introduce a temporal operator corresponding to a "pseudo time," τ, and write (5.45) as

$$u_\tau = u_{xx} + u_{yy} - f(x,y) \ .$$

We already know that the Peaceman–Rachford method is unconditionally stable for linear problems of this type. Thus, we can approximate the above with time-split formulas

$$\left[I - \frac{k}{2}D^2_{0,x}\right] u^*_{l,m} = \left[I + \frac{k}{2}D^2_{0,y}\right] u^{(n)}_{l,m} - \frac{k}{2}f_{l,m}, \tag{5.49a}$$

$$\left[I - \frac{k}{2}D^2_{0,y}\right] u^{(n+1)}_{l,m} = \left[I + \frac{k}{2}D^2_{0,x}\right] u^*_{l,m} - \frac{k}{2}f_{l,m}, \tag{5.49b}$$

$\forall\ l,m$, and use any desired value of time step, k. We observe that (5.49a,b) is identical to (5.40a,b) except for the sign of f (which has been changed for notational consistency with (5.45)), and our altered notation for the time step indices. The latter is due to viewing pseudo-time stepping as being equivalent to iteration in the present case. In this regard, we also note that use of an unconditionally stable time-stepping procedure implies convergence of the iteration procedure. In particular, for the linear problems treated here, the connection between the amplification matrix of the time-stepping algorithm and the Lipschitz constant of the iteration scheme is quite direct. As a consequence, our only real concern with (5.49) is consistency. We wish to show that as $n \to \infty$, (5.49) becomes a consistent approximation to (5.44).

To do this we note that from stability of the time stepping, and hence convergence of the iteration procedure, we have

$$u^{(n)}_{l,m} = u^*_{l,m} = u^{(n+1)}_{l,m}$$

as $n \to \infty$, provided a steady solution to (5.45) exists. It then follows that (5.49) collapses to

$$\left(D^2_{0,x} + D^2_{0,y}\right) u_{l,m} = f_{l,m} \ ,$$

which is the consistent difference approximation to (5.45). Details of proving this are left to the interested reader.

Implementation of ADI for the steady case is only slightly different from that for transient calculations. The first, and more obvious, difference is that a convergence test must now be incorporated into the algorithm, since convergence to steady state must be tested. Second, the effort

expended in coding temporary storage for time-accurate ADI is no longer needed; new iterates are typically used as soon as they are available (*a la* Gauss–Seidel iteration) since temporal accuracy is no longer an issue for steady BVP solutions.

Finally, the fact that the time-step parameter, $r = k/2h^2$, of the transient calculations is now a somewhat arbitrary iteration parameter. It has been found that iterations converge much more rapidly if this parameter is varied from one iteration to the next. In general, only a finite (and usually fairly small) set of different iteration parameters is used; one application of each of these on successive iterations until each has been used once is called a *cycle*. An algorithm implemented so as to use a different iteration parameter on each iteration is said to constitute a *nonstationary* iteration procedure. When only a finite set of parameters is used repeatedly, the method is said to be *cyclic*. Thus, the ADI method just described is often termed *cyclic ADI*.

It should be noted that the cyclic ADI parameters for optimal performance are not easily predicted for any but the simplest problems. Here we present a particular sequence applicable for solving the Poisson equation on the unit square. We also point out that our definition of r is one-half the usual definition. Thus, values of r obtained from the usual sequence, due to Wachspress [7] must be divided by two before insertion into the preceding formulas. We first determine the number of parameters making up the sequence. This is given as the smallest integer n_0 such that

$$\left(\sqrt{2}-1\right)^{(n_0-1)} \leq \tan\frac{\pi}{2(N_x-1)} \; .$$

Then the iteration parameter sequence is given by

$$r_{n+1} = \frac{1}{2\cos^2(\frac{\pi}{2(N_x-1)})}\left[\cot^2\left(\frac{\pi}{2(N_x-1)}\right)\right]^{\frac{n}{n_0-1}}$$

for $n = 0, 1, \ldots, n_0 - 1$. Note that it is assumed that $N_x = N_y$ in applying these formulas.

We end this discussion of elliptic equations by commenting that in recent years a great variety of new and more efficient solution procedures has appeared. Some of these are of restricted applicability, such as the fast Poisson solvers, while others such as the incomplete factorization methods, conjugate gradient, multigrid and domain decomposition can be used to solve problems for which neither SOR nor ADI performs well. Some of these constitute nonstationary iteration schemes, while others often represent more sophisticated forms of SOR. Despite the effort that has gone into developing such methods, genuinely robust and efficient methods still do not exist, and numerical solution of elliptic equations remains an important area of research in computational PDEs.

5.5 Hyperbolic Equations

We begin this section with a fairly complete, but elementary, treatment of the classical second-order wave equation. We discuss some of the basic mathematics associated with this equation, provide a second-order accurate difference approximation for solving it, and then analyze the consistency and stability of this scheme. We then present an introduction to the first-order wave equation, and extend this treatment to first-order hyperbolic systems. The reader is referred to Strikwerda [32] for a comprehensive treatment of these topics.

5.5.1 The Wave Equation

Hyperbolic equations are typified by the second-order wave equation,

$$u_{tt} = c^2 u_{xx} \, , \tag{5.50}$$

where c is the *wave speed*. Here we will mainly consider the Cauchy problem for this equation. That is, we let $x \in (-\infty, \infty)$ and prescribe the initial conditions

$$u(x,0) = f(x) \ , \tag{5.51a}$$
$$u_t(x,0) = g(x) \ . \tag{5.51b}$$

For this case, (5.50) has the exact solution (known as the *d'Lambert solution*),

$$u(x,t) = \frac{1}{2}\left[f(x+ct) + f(x-ct) + \int_{x-ct}^{x+ct} g(\xi)\, d\xi\right] \ . \tag{5.52}$$

An important property of this solution is that for any fixed and finite (x,t), the value of u is determined from values of the initial data restricted to the finite interval $[x-ct, x+ct]$. This interval is called the *domain of dependence* of the point (x,t) and is depicted in Fig. 5.6.

The lines running between $(x-ct,0)$ and (x,t), and $(x+ct,0)$ and (x,t), are respectively the right-running and left-running *characteristics* through the point (x,t). These are shown extended beyond (x,t) in the figure to indicate the *region of influence* of the point (x,t). The value of u at any point in this shaded region depends on the value at (x,t), as well as elsewhere, of course, and the solution at any point outside this region will be independent of $u(x,t)$. The slope of the characteristics is determined from the wave speed c as

$$\theta = \pm \tan^{-1}\left(\frac{1}{c}\right) \ .$$

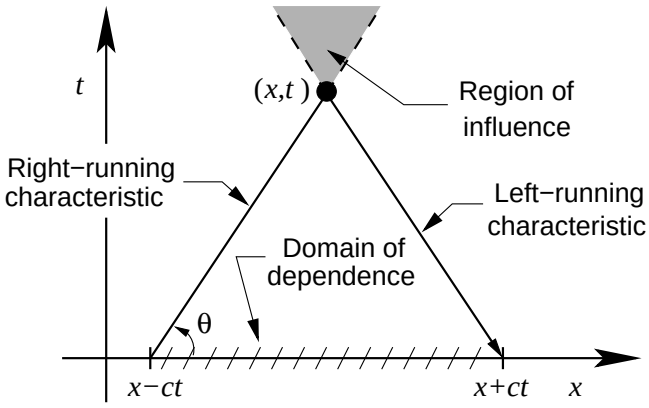


Figure 5.6: Analytical domain of dependence for the point (x,t).

Second-Order Discretization of Wave Equation

We now consider numerical solution of Eq. (5.50) with $c = 1$ and initial conditions (5.51a,b). The most natural explicit difference approximation results from replacing both differential operators in (5.50) with centered differences. Thus, we obtain

$$\frac{1}{k^2}\left(u_m^{n-1} - 2u_m^n + u_m^{n+1}\right) = \frac{1}{h^2}\left(u_{m-1}^n - 2u_m^n + u_{m+1}^n\right) \ .$$

This can be explicitly solved for u at the $(n+1)^{th}$ time level:

$$u_m^{n+1} = 2u_m^n + \frac{k^2}{h^2}\left(u_{m-1}^n - 2u_m^n + u_{m+1}^n\right) - u_m^{n-1}.$$

Now let $\rho = k/h$, and rearrange this to the form

$$u_m^{n+1} = 2(1-\rho^2)u_m^n + \rho^2\left(u_{m+1}^n + u_{m-1}^n\right) - u_m^{n-1}. \tag{5.53}$$

We immediately observe that this is a *three-level scheme*. Hence, when $n = 1$, the left-hand side of (5.53) will provide results at $n = 2$. Moreover, use of $n = 1$ leads to only one term containing initial data—the last term on the right-hand side. Thus, we must have an independent means to produce results for $n = 1$. This can be done using the initial data as follows. First, the $n = 0$ time level is specified by the initial condition $u(x, 0) = f(x)$. Now, from a Taylor expansion we have

$$u_m^1 = u_m^0 + k\left(\frac{\partial u}{\partial t}\right)_m^0 + \frac{k^2}{2}\left(\frac{\partial^2 u}{\partial t^2}\right)_m^0 + \mathcal{O}(k^3)\,. \tag{5.54}$$

But from (5.51b), $(\partial u/\partial t)_m^0$ is given as

$$\frac{\partial u}{\partial t}(x, 0) = g(x)\,. \tag{5.55}$$

Furthermore, from the differential equation we have

$$\left(\frac{\partial^2 u}{\partial t^2}\right)_m^0 = \left(\frac{\partial^2 u}{\partial x^2}\right)_m^0 = \left(\frac{d^2 f}{dx^2}\right)_m = \frac{1}{h^2}(f_{m-1} - 2f_m + f_{m+1}) + \mathcal{O}(h^2)\,. \tag{5.56}$$

Substitution of (5.55) and (5.56) into (5.54) results in an equation for u_m^1 in terms of prescribed initial data $f(x)$ and $g(x)$:

$$u_m^1 = (1-\rho^2)f_m + \frac{1}{2}\rho^2(f_{m+1} + f_{m-1}) + kg_m\,. \tag{5.57}$$

This provides the additional information needed to start calculations that are then continued using (5.53).

We next discuss implementation of (5.53). It should be noted that in practice the initial data, $f(x)$ and $g(x)$ will only be specified on a finite interval, rather than for $x \in (-\infty, \infty)$. As Fig. 5.7 indicates, the length of this interval must be set by the amount of information (number of spatial grid points in the solution) required at the final time. Clearly, as can be seen from the form of the mesh star, one grid point is lost from each end of the grid at each time step just as occurred for the Cauchy problem in the parabolic case discussed earlier. Thus, once k and h are set, the number of grid points on the initial interval required to obtain a desired number on the final interval can be determined. The algorithm in this case is quite similar to Algorithm 5.1, and we leave its development as an exercise for the reader.

Consistency and Stability of Wave Equation Discretization

Now that we have considered calculation using the formula (5.53), it is worthwhile to analyze it with respect to consistency and stability. To check consistency, we require the following Taylor

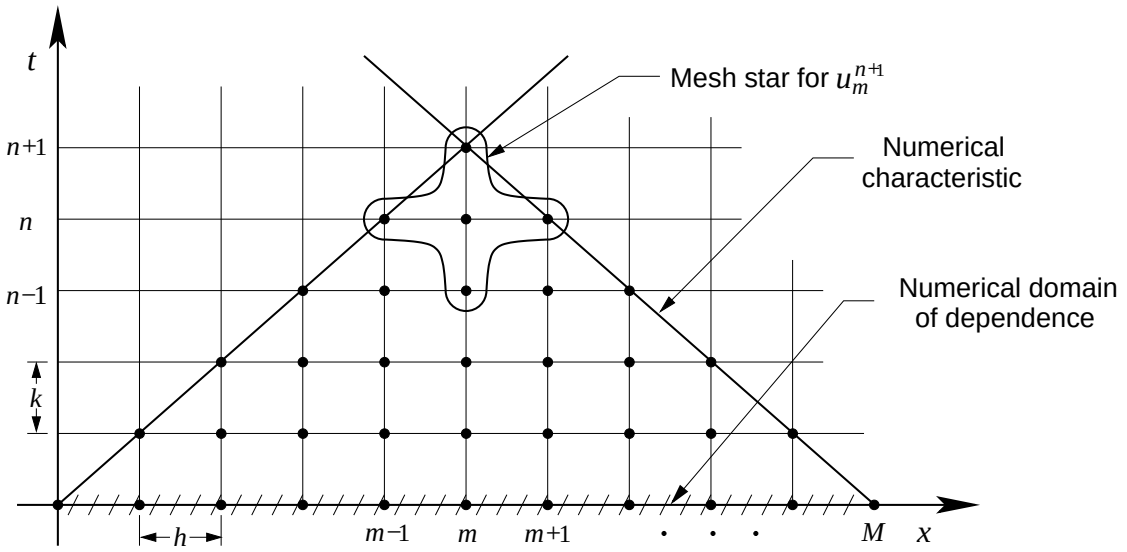


Figure 5.7: Numerical domain of dependence of the grid point $(m, n+1)$.

expansions:

$$u_m^{n+1} = u_m^n + k(u_t)_m^n + \frac{k^2}{2}(u_{tt})_m^n + \frac{k^3}{6}(u_{ttt})_m^n + \frac{k^4}{24}(u_{tttt})_m^n + \cdots$$
$$u_m^{n-1} = u_m^n - k(u_t)_m^n + \frac{k^2}{2}(u_{tt})_m^n - \frac{k^3}{6}(u_{ttt})_m^n + \frac{k^4}{24}(u_{tttt})_m^n - \cdots$$
$$u_{m+1}^n = u_m^n + h(u_x)_m^n + \frac{h^2}{2}(u_{xx})_m^n + \frac{h^3}{6}(u_{xxx})_m^n + \frac{h^4}{24}(u_{xxxx})_m^n + \cdots$$
$$u_{m-1}^n = u_m^n - h(u_x)_m^n + \frac{h^2}{2}(u_{xx})_m^n - \frac{h^3}{6}(u_{xxx})_m^n + \frac{h^4}{24}(u_{xxxx})_m^n - \cdots$$

The first two of these combine to yield

$$u_m^{n+1} + u_m^{n-1} = 2u_m^n + k^2(u_{tt})_m^n + \frac{k^4}{12}(u_{tttt})_m^n + \cdots \ ,$$

and the latter two produce

$$u_{m+1}^n + u_{m-1}^n = 2u_m^n + h^2(u_{xx})_m^n + \frac{h^4}{12}(u_{xxxx})_m^n + \cdots \ .$$

We next rewrite (5.49) as

$$2(1-\rho^2)u_m^n + \rho^2\left(u_{m+1}^n + u_{m-1}^n\right) - \left(u_m^{n+1} + u_m^{n-1}\right) = 0 \ ,$$

and substitute the two preceding results to obtain

$$2(1-\rho^2)u_m^n + 2\rho^2 u_m^n + \rho^2\left[(u_{xx})_m^n h^2 + \frac{1}{12}(u_{xxxx})_m^n h^4\right] - 2u_m^n$$
$$- \left[(u_{tt})_m^n k^2 + \frac{1}{12}(u_{tttt})_m^n k^4\right] + \cdots = 0 \ .$$

After rearrangement this collapses to

$$(u_{tt})_m^n - (u_{xx})_m^n + \frac{1}{12}\left[(u_{tttt})_m^n k^2 - (u_{xxxx})_m^n h^2\right] + \cdots = 0 \ . \tag{5.58}$$

Clearly, as $h, k \to 0$, the dominant truncation error goes to zero, and the differential equation is recovered; hence, the scheme (5.53) is consistent with the wave equation. Moreover, the method is second-order accurate in both space and time.

We analyze the stability of (5.53) via a von Neumann analysis. This might seem particularly appropriate in the present case because there are no boundary conditions. On the other hand, because (5.49) is a three-level difference scheme, it must be recognized that the von Neumann condition supplies only a necessary (and not sufficient) stability requirement. The preferred way to carry out analysis of a multi-level difference equation is to reduce it to a system of two-level equations. For the present problem we define

$$v_m^{n+1} = u_m^n \ .$$

Then (5.53) is replaced by the system

$$u_m^{n+1} = 2(1-\rho^2)u_m^n + \rho^2(u_{m+1}^n + u_{m-1}^n) - v_m \ , \tag{5.59a}$$

$$v_m^{n+1} = u_m^n \ . \tag{5.59b}$$

For $\beta \in \mathbb{R}$ we can write

$$u_{m+1}^n = e^{i\beta h}u_m^n \quad \text{and} \quad u_{m-1}^n = e^{-i\beta h}u_m^n \ ,$$

which holds because, for example,

$$\begin{aligned} u_{m+1}^n &= \left[I + h\frac{\partial}{\partial x} + \frac{h^2}{2}\frac{\partial^2}{\partial x^2} + \cdots\right] u_m^n \\ &\overset{\mathcal{F}}{\longmapsto} \left[I + h(i\beta) - \frac{h^2}{2}\beta^2 + \cdots\right] u_m^n = e^{i\beta h}u_m^n \ , \end{aligned}$$

where $\mathcal{F}$ denotes Fourier transform. Then Eqs. (5.54) become

$$\begin{aligned} u_m^{n+1} &= 2\left(1 - 2\rho^2 \sin^2\frac{\beta h}{2}\right) u_m^n - v_m^n \\ v_m^{n+1} &= u_m^n \ , \end{aligned}$$

which in matrix form is

$$\begin{bmatrix} u_m^{n+1} \\ \\ v_m^{n+1} \end{bmatrix} = \begin{bmatrix} 2\left(1 - 2\rho^2 \sin^2\frac{\beta h}{2}\right) & -1 \\ \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_m^n \\ \\ v_m^n \end{bmatrix} \ .$$

Now using arguments similar to those employed earlier when studying stability for difference approximations to parabolic equations, we know that the error vector must satisfy this same equation. Denoting this by $z_m^n \in \mathbb{R}^2$, we see that

$$z_m^{n+1} = C z_m^n \ ,$$

where

$$C \equiv \begin{bmatrix} 2\left(1 - 2\rho^2 \sin^2 \frac{\beta h}{2}\right) & -1 \\ 1 & 0 \end{bmatrix}$$

is the amplification matrix. As we noted earlier, the von Neumann necessary condition for stability is

$$\|C\| \leq 1 \ ,$$

where $\|\cdot\|$ is the spectral norm. Hence, we need to calculate the eigenvalues of C. The characteristic polynomial is

$$\lambda^2 - 2\left(1 - 2\rho^2 \sin^2 \frac{\beta h}{2}\right)\lambda + 1 = 0 \ ,$$

which has roots

$$\lambda_\pm = 1 - 2\rho^2 \sin^2 \frac{\beta h}{2} \pm 2\rho \sin \frac{\beta h}{2} \left[\rho^2 \sin^2 \frac{\beta h}{2} - 1\right]^{\frac{1}{2}} \ . \tag{5.60}$$

We must determine the larger of these, and from the requirement

$$\max\left(|\lambda_+|, |\lambda_-|\right) \leq 1 \ ,$$

establish permissible bounds on $\rho = k/h$.

There are two cases to consider. First, if $\rho^2 \sin^2 \frac{\beta h}{2} \leq 1$ we have

$$\lambda_\pm = 1 - 2\rho^2 \sin^2 \frac{\beta h}{2} \pm i \left\{2\rho \sin \frac{\beta h}{2} \left[1 - \rho^2 \sin^2 \frac{\beta h}{2}\right]^{\frac{1}{2}}\right\} \ .$$

which implies $|\lambda_+| = |\lambda_-| = 1$. Second, if $\rho^2 \sin^2 \frac{\beta h}{2} > 1$, it immediately follows that $|\lambda_-| > 1$, *i.e.*, instability. Hence, we must require

$$\rho^2 \sin^2 \frac{\beta h}{2} \leq 1$$

for stability. This implies that

$$\rho^2 \leq \frac{1}{\sin^2 \frac{\beta h}{2}} \ ,$$

and thus $\rho^2 \leq 1, \Rightarrow \ \rho \leq 1$. It follows that $k \leq h$ (or $k \leq h/c$ for $c \neq 1$) must hold for stability of (5.54). We again emphasize that this is a necessary, but not sufficient, condition in this case.

The CFL Condition

We now discuss an aspect of hyperbolic equations that does not arise in either parabolic or elliptic cases. Recall that we earlier, in Fig. 5.6, introduced the notion of domain of dependence of a point value of the solution to a hyperbolic equation. Although we did not mention it previously, there is an important relationship between the domain of dependence of the differential equation and that of its difference approximation, shown in Fig. 5.7. These are <u>not</u> the same, in general. For each domain of dependence, the respective solution is influenced only by the points included in its domain. Thus, it follows that if the domain of dependence of the numerical approximation is contained within the domain of dependence of the differential equation, initial data of the latter could be arbitrarily changed outside the domain of dependence of the difference scheme (but inside that of the differential equation), and the numerical solution would remain unchanged. In such a

case, the solution to the difference equation would not converge to that of the differential equation even though it might be a consistent approximation. This is the content of the well-known *Courant-Friedrichs-Lewy (CFL) condition*, often simply called the *Courant condition.*

Theorem 5.2 *(CFL condition) In order for a difference approximation to a Cauchy problem for a hyperbolic equation to converge (to the solution of the DE) as $h, k \to 0$, the domain of dependence of the difference equation must include that of the differential equation.*

For the wave equation we have been discussing, this implies that $\rho \leq 1$ must hold, which coincides with the von Neumann condition.

Although we have not thus far treated implicit methods for hyperbolic problems, we note that the CFL condition is satisfied, independent of the value of ρ for such methods because the domain of dependence of the difference equation is the entire spatial domain under consideration. We shall not give any further treatment herein to the initial boundary value problem for the wave equation except to note that implicit methods can be constructed quite readily merely by evaluating terms from the spatial approximation at the advanced time. For example, we would have

$$\frac{1}{k^2}\left(u_m^{n-1} - 2u_m^n + u_m^{n+1}\right) = \frac{1}{h^2}\left(u_{m-1}^{n+1} - 2u_m^{n+1} + u_{m+1}^{n+1}\right) \ .$$

It is easily seen that after rearrangement this leads to a tridiagonal system to be solved at each time step. Initial conditions are treated just as in the explicit case discussed above, and boundary conditions are handled in the usual way. We note, however, that implicit methods are not so widely used in solving hyperbolic equations for two main reasons. First, the Courant condition for stability is not overly restrictive, at least for 1-D problems (but in multidimensions stability requirements for explicit schemes can become burdensome), and second, a CFL number ($\rho = ck/h$) close to unity is generally required for accurate simulaton of time-dependent wave propagation in any case.

5.5.2 First-Order Hyperbolic Equations and Systems

The last topic we shall treat here is one that is less classical than those discussed previously, but one of great importance in practical computational physics: first-order hyperbolic equations. We will first present several methods that are applicable to single equations, and then use one of these, the Lax–Wendroff method, to solve a first-order hyperbolic system corresponding to the second-order wave equation. Again, we will treat only the pure Cauchy problem; analysis of first-order initial boundary value problems is well beyond the intended scope of these lectures. We refer the interested reader to the monograph by Kreiss and Oliger [18] for a more thorough introduction to this topic.

The First-Order Wave Equation

The first-order equation we shall study here is

$$u_t + au_x = 0 \ , \quad a > 0 \ , \text{ constant} \ , \tag{5.61}$$

with the initial condition

$$u(x, 0) = f(x) \ . \tag{5.62}$$

The exact solution to this problem is

$$u(x, t) = f(x - at) \ , \tag{5.63}$$

as is easily checked. Associated with (5.63) is a single family of right-running characteristics with slope

$$\theta = \tan^{-1}\left(\frac{1}{a}\right) \ .$$

We require knowledge of this for construction of difference schemes which satisfy the CFL condition. For the first-order equation (5.61), the CFL condition requires that the characteristics passing through the advanced time level point pass inside the points of the mesh star at the current time level. This is demonstrated in Fig. 5.8, and is equivalent to satisfaction of a CFL condition in the sense of the CFL Theorem.

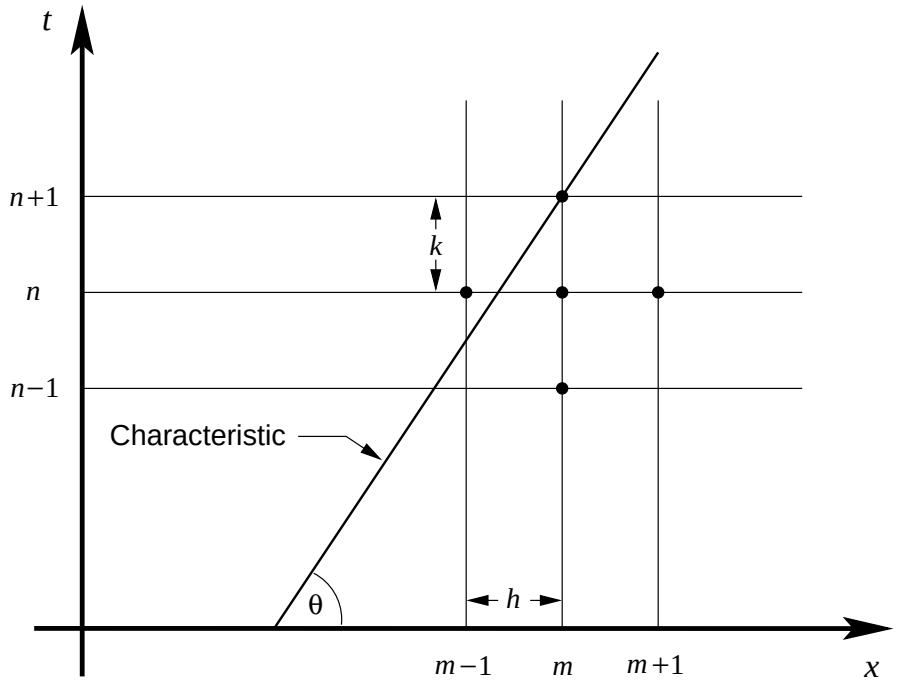


Figure 5.8: Difference approximation satisfying CFL condition.

The simplest approximation to (5.61) satisfying the CFL condition is the first-order approximation

$$\frac{u_m^{n+1} - u_m^n}{k} = -a\frac{u_m^n - u_{m-1}^n}{h} \ ,$$

or

$$u_m^{n+1} = u_m^n - \rho a(u_m^n - u_{m-1}^n) = (1 - \rho a)u_m^n + \rho a u_{m-1}^n \ , \tag{5.64}$$

where $\rho = k/h$. This scheme is explicit, and it is easily checked that the CFL condition is satisfied, provided $\rho a \leq 1$. We defer the stability analysis to what will be developed below for another method. An obvious disadvantage of this method is its first-order accuracy.

We now provide an approach that is second order in both space and time, known as the "leap frog" scheme because it "leaps over", *i.e.*, does not include, the center point of its mesh star. It is constructed by replacing derivatives in both space and time by centered differences. Thus, (5.61) is approximated by

$$\frac{u_m^{n+1} - u_m^{n-1}}{2k} + a\frac{u_{m+1}^n - u_{m-1}^n}{2h} = 0 \ ,$$

or

$$u_m^{n+1} = u_m^{n-1} - \rho a \left(u_{m+1}^n - u_{m-1}^n\right) \ . \tag{5.65}$$

It is easily checked from the preceding figure that the CFL condition again requires $\rho a \leq 1$. The stability analysis for (5.64) is identical in form to the von Neumann analysis earlier applied to the second-order wave equation approximation; and the result is similar. Namely, von Neumann stability coincides with satisfaction of the CFL condition. It is also easily shown by standard methods (Taylor expansion) that the scheme is second-order accurate in both space and time.

The final method we shall consider for single first-order hyperbolic equations is the widely-used Lax–Wendroff [20] scheme. To derive this method we expand u_m^{n+1} in a Taylor series about u_m^n to second order, and then use the differential equation (5.65), to replace derivatives with respect to t with those with respect to x. The derivation is completed by approximating the spatial derivatives with centered differences. We have

$$u_m^{n+1} = u_m^n + k(u_t)_m^n + \frac{k^2}{2}(u_{tt})_m^n + \cdots \ .$$

But from (5.65),

$$(u_t)_m^n = -a(u_x)_m^n \, .$$

From this it follows that

$$(u_{tt})_m^n = -a(u_{xt})_m^n = -a(u_{tx})_m^n = a^2(u_{xx})_m^n \, ,$$

provided u is sufficiently smooth. Substitution of these results into the Taylor series yields

$$u_m^{n+1} = u_m^n - ak(u_x)_m^n + \frac{a^2k^2}{2}(u_{xx})_m^n + \cdots \ .$$

Then replacement of u_x and u_{xx} with centered differences leads to

$$u_m^{n+1} = u_m^n - \frac{\rho a}{2}(u_{m+1}^n - u_{m-1}^n) + \frac{\rho^2 a^2}{2}(u_{m-1}^n - 2u_m^n + u_{m+1}^n) \, ,$$

or

$$u_m^{n+1} = \frac{\rho a}{2}(1+\rho a)u_{m-1}^n + (1-\rho^2 a^2)u_m^n - \frac{\rho a}{2}(1-\rho a)u_{m+1}^n \, . \tag{5.66}$$

Since this is only a two-level scheme, the stability analysis is analogous to that used earlier for parabolic equations. We again note that the error in calculating the solution to (5.66) also satisfies (5.65), and we represent this as

$$z_m^n = \xi^n e^{i\beta mh} \, .$$

Substitution of this quantity into (5.66) followed by division by z_m^n yields the result:

$$\begin{aligned} \xi &= (1-r^2) + \frac{1}{2}r^2\left(e^{i\beta h} + e^{-i\beta h}\right) - \frac{1}{2}r\left(e^{i\beta h} - e^{-i\beta h}\right) \\ &= \left(1 - 2r^2 \sin^2 \frac{\beta h}{2}\right) - ir \sin \beta h \, , \end{aligned}$$

where $r \equiv \rho a$. The modulus of ξ is then

$$|\xi| = \left[1 - 4r^2(1-r^2)\sin^4 \frac{\beta h}{2}\right]^{\frac{1}{2}} \ ,$$

which shows that $|\xi| \leq 1$ if $0 < r \leq 1$. That is, for stability we must have $\rho a \leq 1$, which again coincides with the CFL condition.

First-Order Hyperbolic Systems

We now consider application of the Lax–Wendroff scheme just studied to the first-order hyperbolic system corresponding to the second-order wave equation. Starting with the wave equation,

$$u_{tt} - u_{xx} = 0\,,$$

we set

$$u_1 = u_t \quad \text{and} \quad u_2 = u_x\,.$$

Then we have

$$(u_1)_t - (u_2)_x = 0$$

from the original equation, and

$$(u_2)_t - (u_1)_x = 0$$

from the transformation. These two equations can be represented as the first-order system

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}_t + \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}_x = 0\,, \tag{5.67}$$

or, equivalently,

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}_t - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}_x = 0\ .$$

Next set

$$A \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \text{and} \qquad U \equiv (u_1, u_2)^T\ ,$$

to obtain

$$U_t - AU_x = 0\ . \tag{5.68}$$

We now derive the Lax–Wendroff formula for this system. We have

$$U_m^{n+1} = U_m^n + k(U_t)_m^n + \frac{k^2}{2}(U_{tt})_m^n + \cdots\ .$$

But from (5.64)

$$(U_t)_m^n = A(U_x)_m^n\ ,$$

and analogous to what was found previously in the scalar case,

$$(U_{tt})_m^n = -A^2(U_{xx})_m^n\ .$$

Thus,

$$U_m^{n+1} = U_m^n + Ak(U_x)_m^n - \frac{1}{2}A^2k^2(U_{xx})_m^n + \cdots\ .$$

Finally, we can write this in terms of centered-difference operators as follows:

$$U_m^{n+1} = \left[I + kAD_0 - \frac{k^2}{2}A^2D_0^2\right] U_m^n\ . \tag{5.69}$$

This same procedure can be applied to various other hyperbolic systems with system matrices that are different from the matrix A given here. Moreover, although it is not straightforward, the Lax–Wendroff method can also be applied to variable-coefficient problems.

5.6 Summary

In this chapter we have studied elementary numerical methods, based on the finite-difference approximation, for solving the partial differential equations of classical mathematical physics, as well as for first-order wave equations and first-order hyperbolic systems.

For parabolic equations typified by the heat equation we presented both explicit and implicit methods. We should recall that explicit methods are generally less stable than implicit ones, and this is especially true in the parabolic case. Particularly for initial boundary value problems we recommend use of the Crank–Nicolson method in most situations even though there is considerably more required arithmetic per time step in comparison with explicit techniques because of the need to solve tridiagonal linear systems at each time step. But far larger time steps can be used, so required arithmetic is essentially always less than that for an explicit method. In the 2-D case we gave a fairly detailed treatment of the Peaceman–Rachford ADI scheme applied to the heat equation. This is a particular instance of a wide class of methods often referred to as "time splitting." They are extremely effective for solving time-dependent problems because of their favorable stability properties, and because of the efficiency with which they can be solved, especially on modern symmetric multi-processor (SMP) and cluster architectures.

We discussed only two very basic techniques for solving elliptic equations such as Laplace's equation, namely SOR and ADI. We noted that there are by now many more sophisticated methods, but the majority of these tend to be of somewhat limited applicability. Variants of SOR and ADI are naturally highly parallelizable, and because of this they may yet prove to be as effective as many of the modern procedures such as conjugate gradient, incomplete LU decompositions and generalized minimum residual (GMRES) to name a few.

In any case, it is important to recognize that iterative methods (at least in the guise of pseudo-time marching) are essentially always used for solving the sparse, banded linear systems arising from discretization of elliptic PDEs. Such systems can be extremely large ($10^6 \times 10^6$, or larger, matrices are now common), and direct methods requiring $\mathcal{O}(N^3)$ arithmetic are not usually viable techniques. But it must be mentioned that direct methods exist that require only $\mathcal{O}(N^2)$ arithmetic operations when applied to the sparse systems considered here; if these can be parallelized, then they may prove to be effective alternatives to iteration in some cases.

Our discussions of hyperbolic PDEs included numerical treatment of the classical second-order wave equation as well as first-order wave equations in scalar and vector form. Explicit methods were emphasized because they are very efficient and easily implemented, and in the hyperbolic case the requirements for stability (generally satisfaction of a Courant condition) tend to impose far less restriction on the time step than is true for parabolic problems. We mainly treated only the Cauchy problem. This was done to avoid the technicalities of numerical boundary conditions for hyperbolic problems. This is still a very active area of research, particularly in the context of computational electromagnetics, but also in computational fluid dynamics.

The main computational techniques we presented include straightforward centered differencing (in both space and time) for the second-order wave equation and the leap-frog method, the equivalent scheme for first-order equations. In addition, we provided a fairly complete treatment of the Lax–Wendroff scheme for first-order equations and systems. Although these are among the most widely-used methods for hyperbolic equations, there are numerous other possibilities, as discussed in Strikwerda [32] and elsewhere. The interested reader is encouraged to consult the extant literature.

References

[1] T. M. Apostol. *Mathematical Analysis.* Addison-Wesley Pub. Co., Reading, MA, second edition, 1974.

[2] P. W. Berg and J. L. McGregor. *Elementary Parial Differential Equations.* Holden-Day, San Francisco, 1966.

[3] G. Dahlquist. *A Special Stability Problem for Linear Multistep Methods.* BIT 3, New York, 1963.

[4] P. J. Davis and P. Rabinowitz. *Methods of Numerical Integration.* Academic Press, New York, NY, 1975.

[5] C. deBoor. *A Practical Guide to Splines.* V. A. Barker (ed.), Springer-Verlag, New York, NY, 1978.

[6] C. B. Moler J. J. Dongarra, J. R. Bunch and G. W. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, PA, 1979.

[7] J. Douglas, Jr. and J. E. Gunn. A general formulation of alternating direction methods, part i. parabolic and hyperbolic problems. *Numer. Math.* **6**, 428–453, 1964.

[8] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[9] G. Forsythe and C. B. Moler. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1967.

[10] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971.

[11] K. E. Gustafson. *Introduction to Partial Differential Equations and Hilbert Space Methods.* John Wiley and Sons, New York, 1980.

[12] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations.* Springer-Verlag, NY, 1994.

[13] P. Henrici. *Elements of Numerical Analysis.* John Wiley & Sons, New York, 1964.

[14] R. W. Hornbeck. *Numerical Methods.* Quantum Publishers, Inc., New York, NY, 1975.

[15] E. Isaacson and H. B. Keller. *Analysis of Numerical Methods.* John Wiley & Sons, Inc., New York, NY, 1966 (now available as the Dover paperback first published in 1994).

[16] L. W. Johnson and R. D. Riess. *Numerical Analysis*. Addison-Wesley Pub. Co., Reading, MA, second edition, 1982.

[17] H. B. Keller. *Numerical Methods for Two-Point Boundary-Value Problems*. Dover Pub., Inc., New York, NY, 1992.

[18] H. Kreiss and J. Oliger. *Methods for the Approximate Solution of Time Dependent Problems*. GARP Pub, 1973.

[19] L. Lapidus and J. H. Seinfield. *Numerical Solution of Ordinary Differential Equations*. Academic Press, New York, NY, 1971.

[20] P. D. Lax and B. Wendroff. *Systems of Conservation Laws*, volume 13. Can. Pure Appl. Math., 1960.

[21] A. R. Mitchell and D. F. Griffiths. *The Finite Difference Method in Partial Differential Equations*. John Wiley & Sons, Inc., Chichester, 1980.

[22] A. M. Ostrowski. *Solution of Equations and Systems of Equations*. Academic Press, New York, second edition, 1966.

[23] D. W. Peaceman and H. H. Rachford, Jr. "The numerical solution of Parabolic and Elliptic Differential Equations", *SIAM J.* bf 3, 28–41, 1955.

[24] R. D. Richtmyer and K. W. Morton. *Difference Methods for Initial-Value Problems*. John Wiley & Sons, Inc., New York, NY, second edition, 1967.

[25] A. Ruhe. "Computation of Eigenvalues and Eigenvectors" *in Sparse Matrix Techniques*. V. A. Barker (ed.), Springer-Verlag, Berlin, 1977.

[26] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Co., Boston, 1996.

[27] B. T. Smith, P. Bjørstad and W. Gropp. *Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, Cambridge, 1996.

[28] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema and C. B. Moler. *Matrix Eigensystem Routines - EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, second edition, 1976.

[29] I. Stakgold. *Boundary Value Problems of Mathematical Physics* Volume I. Society for Industrial and Applied Mathematics, Philadelphia, 2000.

[30] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1980.

[31] G. Strang and G. J. Fix. *An Analysis of the Finite Element Method*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[32] J. C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. Wadsworth & Brooks/Cole, Pacific Groove, CA, 1989.

[33] A. H. Stroud. *Approximate Calculation of Multiple Integrals*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971.

[34] L. H. Thomas. "Elliptic Problems in Linear Difference Equations over a Network," *Watson Sci. Comput. Lab. Rept.*, Columbia University, NY, 1949.

[35] J. F. Thompson, B. K. Soni and N. P. Weatherill (Eds.) *Handbook of GRID GENERATION*. CRC Press, Boca Raton, 1999.

[36] J. F. Traub. *Iterative Methods for the Solution of Equations*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1964.

[37] J. H. Wilkinson. *The Algeibraic Eigenvalue Problem*. University Press, London, 1965.

[38] J. H. Wilkinson and C. Reinsch. *Linear Algebra*. Springer-Verlag, New York, NY, 1971.

[39] D. M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, New York, NY, 1971.