Numerical Methods in Science and Engineering

Thomas R. Bewley
UC San Diego

Contents

Preface

The present text provides a brief (one quarter) introduction to efficient and effective numerical methods for solving typical problems in scientific and engineering applications. It is intended to provide a succinct and modern guide for a senior or first-quarter masters level course on this subject, assuming only a prior exposure to linear algebra, a knowledge of complex variables, and rudimentary skills in computer programming.

I am indebted to several sources for the material compiled in this text, which draws from class notes from ME200 by Prof. Parviz Moin at Stanford University, class notes from ME214 by Prof. Harv Lomax at Stanford University, and material presented in *Numerical Recipes* by Press *et al.* (1988-1992) and *Matrix Computations* by Golub & van Loan (1989). The latter two textbooks are highly recommended as supplemental texts to the present notes. We do not attempt to duplicate these excellent texts, but rather attempt to build up to and introduce the subjects discussed at greater lengths in these more exhaustive texts at a metered pace.

The present text was prepared as supplemental material for MAE107 and MAE290a at UC San Diego.

Chapter 1

A short review of linear algebra

Linear algebra forms the foundation upon which efficient numerical methods may be built to solve both linear and nonlinear systems of equations. Consequently, it is useful to review briefly some relevant concepts from linear algebra.

1.1 Notation

Over the years, a fairly standard notation has evolved for problems in linear algebra. For clarity, this notation is now reviewed.

1.1.1 Vectors

A vector is defined as an ordered collection of numbers or algebraic variables:

$$\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}.$$

All vectors in the present notes will be assumed to be arranged in a column unless indicated otherwise. Vectors are represented with lower-case letters and denoted in writing (*i.e.*, on the blackboard) with an arrow above the letter ($\vec{c}$) and in print (as in these notes) with boldface ($\mathbf{c}$). The vector $\mathbf{c}$ shown above is n-dimensional, and its i'th element is referred to as c_i. For simplicity, the elements of all vectors and matrices in these notes will be assumed to be real unless indicated otherwise. However, all of the numerical tools we will develop extend to complex systems in a straightforward manner.

1.1.2 Vector addition

Two vectors of the same size are added by adding their individual elements:

$$\mathbf{c} + \mathbf{d} = \begin{pmatrix} c_1 + d_1 \\ c_2 + d_2 \\ \vdots \\ c_n + d_n \end{pmatrix}.$$

1.1.3 Vector multiplication

In order to multiply a vector with a scalar, operations are performed on each element:

$$\alpha\,\mathbf{c} = \begin{pmatrix} \alpha\, c_1 \\ \alpha\, c_2 \\ \vdots \\ \alpha\, c_n \end{pmatrix}.$$

The **inner product** of two real vectors of the same size, also known as **dot product**, is defined as the sum of the products of the corresponding elements:

$$(\mathbf{u}, \mathbf{v}) = \mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i\, v_i = u_1\, v_1 + u_2\, v_2 + \ldots + u_n\, v_n.$$

The **2-norm of a vector**, also known as the **Euclidean norm** or the **vector length**, is defined by the square root of the inner product of the vector with itself:

$$\|\mathbf{u}\| = \sqrt{(\mathbf{u}, \mathbf{u})} = \sqrt{u_1^2 + u_2^2 + \ldots + u_n^2}.$$

The **angle between two vectors** may be defined using the inner product such that

$$\cos \measuredangle(\mathbf{u}, \mathbf{v}) = \frac{(\mathbf{u}, \mathbf{v})}{\|\mathbf{u}\|\, \|\mathbf{v}\|}.$$

In **summation notation**, any term in an equation with lower-case English letter indices repeated twice implies summation over all values of that index. Using this notation, the inner product is written simply as $u_i\, v_i$. To avoid implying summation notation, Greek indices are usually used. Thus, $u_\alpha\, v_\alpha$ does *not* imply summation over α.

1.1.4 Matrices

A matrix is defined as a two-dimensional ordered array of numbers or algebraic variables:

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}.$$

The matrix above has m rows and n columns, and is referred to as an $m \times n$ matrix. Matrices are represented with uppercase letters, with their elements represented with lowercase letters. The element of the matrix A in the i'th row and the j'th column is referred to as a_{ij}.

1.1.5 Matrix addition

Two matrices of the same size are added by adding their individual elements. Thus, if $C = A + B$, then $c_{ij} = a_{ij} + b_{ij}$.

1.1.6 Matrix/vector multiplication

The product of a matrix A with a vector $\mathbf{x}$, which results in another vector $\mathbf{b}$, is denoted $A\mathbf{x} = \mathbf{b}$. It may be defined in index notation as:

$$b_i = \sum_{j=1}^{n} a_{ij}\, x_j.$$

In summation notation, it is written:

$$b_i = a_{ij}\, x_j.$$

Recall that, as the j index is repeated in the above expression, summation over all values of j is implied without being explicitly stated. The first few elements of the vector $\mathbf{b}$ are given by:

$$\begin{aligned} b_1 &= a_{11}\, x_1 + a_{12}\, x_2 + \ldots + a_{1n}\, x_n, \\ b_2 &= a_{21}\, x_1 + a_{22}\, x_2 + \ldots + a_{2n}\, x_n, \end{aligned}$$

etc. The vector $\mathbf{b}$ may be written:

$$\mathbf{b} = x_1 \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} + x_2 \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} + \ldots + x_n \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix}.$$

Thus, $\mathbf{b}$ is simply a linear combination of the columns of A with the elements of $\mathbf{x}$ as weights.

1.1.7 Matrix multiplication

Given two matrices A and B, where the number of columns of A is the same as the number of rows of B, the product $C = A\,B$ is defined in summation notation, for the (i,j)'th element of the matrix C, as

$$c_{ij} = a_{ik}\, b_{kj}.$$

Again, as the index k is repeated in this expression, summation is implied over the index k. In other words, c_{ij} is just the inner product of row i of A with column j of B. For example, if we write:

$$\underbrace{\begin{pmatrix} c_{11} & \boxed{c_{12}} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \ldots & c_{mn} \end{pmatrix}}_{C} = \underbrace{\begin{pmatrix} \boxed{a_{11} \quad a_{12} \quad \ldots \quad a_{1l}} \\ a_{21} \quad a_{22} \quad \ldots \quad a_{2l} \\ \vdots \quad\;\; \vdots \quad\;\; \ddots \quad\;\; \vdots \\ a_{m1} \quad a_{m2} \quad \ldots \quad a_{ml} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} b_{11} & \boxed{\begin{matrix} b_{12} \\ b_{22} \\ \vdots \\ b_{l2} \end{matrix}} & \ldots & b_{1n} \\ b_{21} & & \ldots & b_{2n} \\ \vdots & & \ddots & \vdots \\ b_{l1} & & \ldots & b_{ln} \end{pmatrix}}_{B},$$

then we can see that c_{12} is the inner product of row 1 of A with column 2 of B. Note that usually $AB \neq BA$; matrix multiplication usually does not commute.

1.1.8 Identity matrix

The identity matrix is a square matrix with ones on the diagonal and zeros off the diagonal.

$$I = \begin{pmatrix} 1 & & & 0 \\ & 1 & & \\ & & \ddots & \\ 0 & & & 1 \end{pmatrix} \qquad \Rightarrow \qquad I\mathbf{x} = \mathbf{x}, \qquad IA = AI = A$$

In the notation for I used at left, in which there are several blank spots in the matrix, the zeros are assumed to act like "paint" and fill up all unmarked entries. Note that a matrix or a vector is not changed when multiplied by I. The elements of the identity matrix are equal to the Kronecker delta:

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise.} \end{cases}$$

1.1.9 Inverse of a square matrix

If $BA = I$, we may refer to B as A^{-1}. (Note, however, that for a given square matrix A, it is not always possible to compute its inverse; when such a computation is possible, we refer to the matrix A as being "nonsingular" or "invertible".) If we take $A\mathbf{x} = \mathbf{b}$, we may multiply this equation from the left by A^{-1}, which results in

$$A^{-1}\Big[A\mathbf{x} = \mathbf{b}\Big] \qquad \Rightarrow \qquad \mathbf{x} = A^{-1}\mathbf{b}.$$

Computation of the inverse of a matrix thus leads to one method for determining $\mathbf{x}$ given A and $\mathbf{b}$; unfortunately, this method is extremely inefficient. Note that, since matrix multiplication does not commute, one always has to be careful when multiplying an equation by a matrix to multiply out all terms consistently (either from the left, as illustrated above, or from the right).

If we take $AB = I$ and $CA = I$, then we may premultiply the former equation by C, leading to

$$C\Big[AB = I\Big] \qquad \Rightarrow \qquad \underbrace{CA}_{I}B = C \qquad \Rightarrow \qquad B = C.$$

Thus, **the left and right inverses are identical**.

If we take $AX = I$ and $AY = I$ (noting by the above argument that $YA = I$), it follows that

$$Y\Big[AX = I\Big] \qquad \Rightarrow \qquad \underbrace{YA}_{I}X = Y \qquad \Rightarrow \qquad X = Y.$$

Thus, **the inverse is unique**.

1.1.10 Other definitions

The **transpose of a matrix** A, denoted A^T, is found by swapping the rows and the columns:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad \Rightarrow \qquad A^T = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}.$$

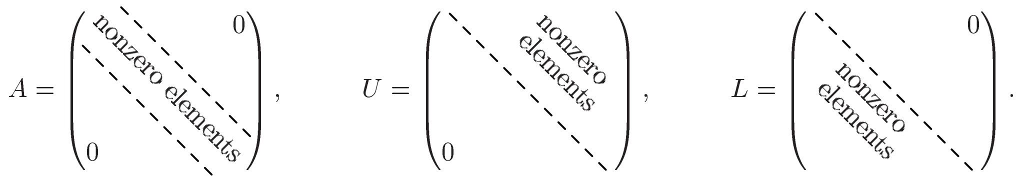
In index notation, we say that $b_{ij} = a_{ji}$, where $B = A^T$.

The **adjoint of a matrix** A, denoted A^*, is found by taking the conjugate transpose of A:

$$A = \begin{pmatrix} 1 & 2i \\ 1+3i & 0 \end{pmatrix} \qquad \Rightarrow \qquad A^* = \begin{pmatrix} 1 & 1-3i \\ -2i & 0 \end{pmatrix}.$$

The **main diagonal** of a matrix A is the collection of elements along the line from a_{11} to a_{nn}. The **first subdiagonal** is immediately below the main diagonal (from a_{21} to $a_{n,n-1}$), the **second subdiagonal** is immediately below the first subdiagonal, etc.; the **first superdiagonal** is immediately above the main diagonal (from a_{12} to $a_{n-1,n}$), the **second superdiagonal** is immediately above the first superdiagonal, etc.

A **banded matrix** has nonzero elements only near the main diagonal. Such matrices arise in discretization of differential equations. As we will show, the narrower the width of the band of nonzero elements, the easier it is to solve the problem $A\mathbf{x} = \mathbf{b}$ with an efficient numerical algorithm. A **diagonal matrix** is one in which only the main diagonal of the matrix is nonzero, a **tridiagonal matrix** is one in which only the main diagonal and the first subdiagonal and superdiagonal are nonzero, etc. An **upper triangular matrix** is one for which all subdiagonals are zero, and a **lower triangular matrix** is one for which all superdiagonals are zero. Generically, such matrices look like:



Banded matrix Upper triangular matrix Lower triangular matrix

A **block banded matrix** is a banded matrix in which the nonzero elements themselves are naturally grouped into smaller submatrices. Such matrices arise when discretizing systems of partial differential equations in more than one direction. For example, as shown in class, the following is the block tridiagonal matrix that arises when discretizing the Laplacian operator in two dimensions on a uniform grid:

$$M = \begin{pmatrix} B & C & & & 0 \\ A & B & C & & \\ & \ddots & \ddots & \ddots & \\ & & A & B & C \\ 0 & & & A & B \end{pmatrix} \qquad \text{with} \qquad B = \begin{pmatrix} -4 & 1 & & & 0 \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ 0 & & & 1 & -4 \end{pmatrix}, \qquad A = C = I.$$

We have, so far, reviewed some of the notation of linear algebra that will be essential in the development of numerical methods. In §2, we will discuss various methods of solution of nonsingular square systems of the form $A\,\mathbf{x} = \mathbf{b}$ for the unknown vector $\mathbf{x}$. We will need to solve systems of this type repeatedly in the numerical algorithms we will develop, so we will devote a lot of attention to this problem. With this machinery, and a bit of analysis, we will see in the first homework that we are already able to analyze important systems of engineering interest with a reasonable degree of accuracy. In homework #1, we analyze of the static forces in a truss subject to some significant simplifying assumptions.

In order to further our understanding of the statics and dynamics of phenomena important in physical systems, we need to review a few more elements from linear algebra: determinants, eigenvalues, matrix norms, and the condition number.

1.2 Determinants

1.2.1 Definition of the determinant

An extremely useful method to characterize a square matrix is by making use of a scalar quantity called the **determinant**, denoted $|A|$. The determinant may be defined by induction as follows:

1) The determinant of a 1×1 matrix $A = [a_{11}]$ is just $|A| = a_{11}$.

2) The determinant of a 2×2 matrix $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ is $|A| = a_{11}\, a_{22} - a_{12}\, a_{21}$.

n) The determinant of an $n \times n$ matrix is defined as a function of the determinant of several $(n-1) \times (n-1)$ matrices as follows: the determinant of A is a linear combination of the elements of row α (for any α such that $1 \leq \alpha \leq n$) and their corresponding cofactors:

$$|A| = a_{\alpha 1}\, A_{\alpha 1} + a_{\alpha 2}\, A_{\alpha 2} + \cdots a_{\alpha n}\, A_{\alpha n},$$

where the cofactor $A_{\alpha\beta}$ is defined as the determinant of $M_{\alpha\beta}$ with the correct sign:

$$A_{\alpha\beta} = (-1)^{\alpha+\beta} |M_{\alpha\beta}|,$$

where the minor $M_{\alpha\beta}$ is the matrix formed by deleting row α and column β of the matrix A.

1.2.2 Properties of the determinant

When defined in this manner, the determinant has several important properties:

1. Adding a multiple of one row of the matrix to another row leaves the determinant unchanged:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} a & b \\ 0 & d - \frac{c}{a} b \end{vmatrix}$$

2. Exchanging two rows of the matrix flips the sign of the determinant:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = - \begin{vmatrix} c & d \\ a & b \end{vmatrix}$$

3. If A is triangular (or diagonal), then $|A|$ is the product $a_{11}\, a_{22} \cdots a_{nn}$ of the elements on the main diagonal. In particular, the determinant of the identity matrix is $|I| = 1$.

4. If A is nonsingular (*i.e.*, if $A\mathbf{x} = \mathbf{b}$ has a unique solution), then $|A| \neq 0$.
   If A is singular (*i.e.*, if $A\mathbf{x} = \mathbf{b}$ does not have a unique solution), then $|A| = 0$.

1.2.3 Computing the determinant

For a large matrix A, the determinant is most easily computed by performing the row operations mentioned in properties #1 and 2 discussed in the previous section to reduce A to an upper triangular matrix U. (In fact, this is the heart of Gaussian elimination procedure, which will be described in detail §2.) Taking properties #1, 2, and 3 together, if follows that

$$|A| = (-1)^r\, |U| = (-1)^r\, u_{11}\, u_{22} \cdots u_{nn},$$

where r is the number of row exchanges performed, and the $u_{\alpha\alpha}$ are the elements of U which are on the main diagonal. By property #4, we see that whether or not the determinant of a matrix is zero is the litmus test for whether or not that matrix is singular.

The command `det(A)` is used to compute the determinant in Matlab.

1.3 Eigenvalues and Eigenvectors

Consider the equation

$$A\boldsymbol{\xi} = \lambda\boldsymbol{\xi}.$$

We want to solve for both a scalar λ and some corresponding vector $\boldsymbol{\xi}$ (other than the trivial solution $\boldsymbol{\xi} = \mathbf{0}$) such that, when $\boldsymbol{\xi}$ is multiplied from the left by A, it is equivalent to simply scaling $\boldsymbol{\xi}$ by the factor λ. Such a situation has the important physical interpretation as a natural mode of a system when A represents the "system matrix" for a given dynamical system, as will be illustrated in §1.3.1.

The easiest way to determine for which λ it is possible to solve the equation $A\boldsymbol{\xi} = \lambda\boldsymbol{\xi}$ for $\boldsymbol{\xi} \neq \mathbf{0}$ is to rewrite this equation as

$$(A - \lambda I)\boldsymbol{\xi} = \mathbf{0}.$$

If $(A - \lambda I)$ is a nonsingular matrix, then this equation has a unique solution, and since the right-hand side is zero, that solution must be $\boldsymbol{\xi} = \mathbf{0}$. However, for those values of λ for which $(A - \lambda I)$ is singular, this equation admits other solutions with $\boldsymbol{\xi} \neq \mathbf{0}$. The values of λ for which $(A - \lambda I)$ is singular are called the **eigenvalues** of the matrix A, and the corresponding vectors $\boldsymbol{\xi}$ are called the **eigenvectors**.

Making use of property #4 of the determinant, we see that the eigenvalues must therefore be exactly those values of λ for which

$$|A - \lambda I| = 0.$$

This expression, when multiplied out, turns out to be a polynomial in λ of degree n for an $n \times n$ matrix A; this is referred to as the **characteristic polynomial** of A. By the fundamental theorem of algebra, there are exactly n roots to this equation, though these roots need not be distinct. Once the eigenvalues λ are found by finding the roots of the characteristic polynomial of A, the eigenvectors $\boldsymbol{\xi}$ may be found by solving the equation $(A - \lambda I)\boldsymbol{\xi} = \mathbf{0}$. Note that the $\boldsymbol{\xi}$ in this equation may be determined only up to an arbitrary constant, which can not be determined because $(A - \lambda I)$ is singular. In other words, if $\boldsymbol{\xi}$ is an eigenvector corresponding to a particular eigenvalue λ, then $c\boldsymbol{\xi}$ is also an eigenvector for any scalar c. Note also that, if all of the eigenvalues of A are distinct (different), all of the eigenvectors of A are linearly independent (*i.e.*, $\measuredangle(\boldsymbol{\xi}^i, \boldsymbol{\xi}^j) \neq 0$ for $i \neq j$).

The command `[V,D] = eig(A)` is used to compute eigenvalues and eigenvectors in Matlab.

1.3.1 Physical motivation for eigenvalues and eigenvectors

In order to realize the significance of eigenvalues and eigenvectors for characterizing physical systems, and to foreshadow some of the developments in later chapters, it is enlightening at this point to diverge for a bit and discuss the time evolution of a taught wire which has just been struck (as with a piano wire) or plucked (as with the wire of a guitar or a harp). Neglecting damping, the deflection of the wire, $f(x,t)$, obeys the linear partial differential equation (PDE)

$$\frac{\partial^2 f}{\partial t^2} = \alpha^2 \frac{\partial^2 f}{\partial x^2} \tag{1.1}$$

subject to

$$\text{boundary conditions:} \quad \begin{cases} f = 0 & \text{at } x = 0, \\ f = 0 & \text{at } x = L, \end{cases}$$

$$\text{and initial conditions:} \quad \begin{cases} f = c(x) & \text{at } t = 0, \\ \frac{\partial f}{\partial t} = d(x) & \text{at } t = 0. \end{cases}$$

We will solve this system using the separation of variables (SOV) approach. With this approach, we seek "modes" of the solution, f_ι, which satisfy the boundary conditions on f and which decouple into the form

$$f_\iota = X_\iota(x) T_\iota(t). \tag{1.2}$$

(No summation is implied over the Greek index ι, pronounced "iota".) If we can find enough nontrivial (nonzero) solutions of (1.1) which fit this form, we will be able to reconstruct a solution of (1.1) which also satisfies the initial conditions as a superposition of these modes. Inserting (1.2) into (1.1), we find that

$$X_\iota T_\iota'' = \alpha^2 X_\iota'' T_\iota \quad \Rightarrow \quad \frac{T_\iota''}{T_\iota} = \alpha^2 \frac{X_\iota''}{X_\iota} \triangleq -\omega_\iota^2 \quad \Rightarrow \quad \begin{aligned} X_\iota'' &= -\frac{\omega_\iota^2}{\alpha^2} X_\iota \\ T_\iota'' &= -\omega_\iota^2 T_\iota, \end{aligned}$$

where the constant ω_ι must be independent of both x and t due to the center equation combined with the facts that $X_\iota = X_\iota(x)$ and $T_\iota = T_\iota(t)$. The two systems at right are solved with:

$$\begin{aligned} X_\iota &= A_\iota \cos\left(\frac{\omega_\iota x}{\alpha}\right) + B_\iota \sin\left(\frac{\omega_\iota x}{\alpha}\right) \\ T_\iota &= C_\iota \cos(\omega_\iota t) \quad + D_\iota \sin(\omega_\iota t) \end{aligned}$$

Due to the boundary condition at $x = 0$, it must follow that $A_\iota = 0$. Due to the boundary condition at $x = L$, it follows for most ω_ι that $B_\iota = 0$ as well, and thus $f_\iota(x,t) = 0 \;\forall\; x, t$. However, for certain specific values of ω_ι (specifically, for $\omega_\iota L/\alpha = \iota\pi$ for integer values of ι), X_ι satisfies the homogeneous boundary condition at $x = L$ even for nonzero values of B_ι.

We now attempt to form a superposition of the nontrivial f_ι that solves the initial conditions given for f. Defining $\hat{c}_\iota = B_\iota C_\iota$ and $\hat{d}_\iota = B_\iota D_\iota$, we take

$$f = \sum_{\iota=1}^{\infty} f_\iota = \sum_{\iota=1}^{\infty} \left[\hat{c}_\iota \sin\left(\frac{\omega_\iota x}{\alpha}\right) \cos(\omega_\iota t) + \hat{d}_\iota \sin\left(\frac{\omega_\iota x}{\alpha}\right) \sin(\omega_\iota t) \right],$$

where $\omega_\iota \triangleq \iota\pi\alpha/L$. The coefficients $\hat{c}_\iota$ and $\hat{d}_\iota$ may be determined by enforcing the initial conditions:

$$f(x, t=0) = c(x) = \sum_{i=1}^{\infty} \hat{c}_\iota \sin\left(\frac{\omega_\iota x}{\alpha}\right), \qquad \frac{\partial f}{\partial t}(x, t=0) = d(x) = \sum_{i=1}^{\infty} \hat{d}_\iota \omega_\iota \sin\left(\frac{\omega_\iota x}{\alpha}\right).$$

Noting the orthogonality of the sine functions[1], we multiply both of the above equations by $\sin(\omega_\kappa x/\alpha)$ and integrate over the domain $x \in [0, L]$, which results in:

$$\left. \begin{aligned} \int_0^L c(x) \sin\left(\frac{\omega_\kappa x}{\alpha}\right) dx = \hat{c}_\kappa \frac{L}{2} \quad &\Rightarrow \quad \hat{c}_\kappa = \frac{2}{L} \int_0^L c(x) \sin\left(\frac{\omega_\kappa x}{\alpha}\right) dx \\ \int_0^L d(x) \sin\left(\frac{\omega_\kappa x}{\alpha}\right) dx = \hat{d}_\kappa \omega_\kappa \frac{L}{2} \quad &\Rightarrow \quad \hat{d}_\kappa = \frac{2}{\kappa\pi\alpha} \int_0^L d(x) \sin\left(\frac{\omega_\kappa x}{\alpha}\right) dx \end{aligned} \right\} \quad \text{for } \kappa = 1, 2, 3, \ldots$$

The $\hat{c}_\iota$ and $\hat{d}_\iota$ are referred to as the discrete sine transforms of $c(x)$ and $d(x)$ on the interval $x \in [0, L]$.

[1] This orthogonality principle states that, for ι, κ integers:

$$\int_0^L \sin\left(\frac{\iota\pi x}{L}\right) \sin\left(\frac{\kappa\pi x}{L}\right) dx = \begin{cases} L/2 & \iota = \kappa \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the solutions of (1.1) which satisfies the boundary conditions and initial conditions may be found as a linear combination of modes of the simple decoupled form given in (1.2), which may be determined analytically. But what if, for example, α is a function of x? Then we can no longer represent the mode shapes analytically with sines and cosines. In such cases, we can still seek decoupled modes of the form $f = X(x)T(t)$, but we now must determine the $X(x)$ numerically. Consider again the equation of the form:

$$X'' = -\frac{\omega^2}{\alpha^2}X \qquad \text{with} \qquad X = 0 \quad \text{at } x = 0 \text{ and } x = L.$$

Consider now the values of X only at $N+1$ discrete locations ("grid points") located at $x = j\,\Delta x$ for $j = 0\ldots N$, where $\Delta x = L/N$. Note that, at these grid points, the second derivative may be approximated by:

$$\left.\frac{\partial^2 X}{\partial x^2}\right|_{x_j} \approx \left(\frac{X_{j+1}-X_j}{\Delta x} - \frac{X_j - X_{j-1}}{\Delta x}\right)\Big/\Delta x = \frac{X_{j+1}-2X_j+X_{j-1}}{(\Delta x)^2},$$

where, for clarity, we have switched to the notation that $X_j \triangleq X(x_j)$. By the boundary conditions, $X_0 = X_N = 0$. The differential equation at each of the $N-1$ grid points on the interior may be approximated by the relation

$$\alpha_j^2 \frac{X_{j+1}-2X_j+X_{j-1}}{(\Delta x)^2} = -\omega^2 X_j,$$

which may be written in the matrix form:

$$\frac{1}{(\Delta x)^2}\begin{pmatrix} -2\alpha_1^2 & \alpha_1^2 & & & & 0 \\ \alpha_2^2 & -2\alpha_2^2 & \alpha_2^2 & & & \\ & \alpha_3^2 & -2\alpha_3^2 & \alpha_3^2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \alpha_{N-2}^2 & -2\alpha_{N-2}^2 & \alpha_{N-2}^2 \\ 0 & & & & \alpha_{N-1}^2 & -2\alpha_{N-1}^2 \end{pmatrix}\begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N-2} \\ X_{N-1}\end{pmatrix} = \Big[-\omega^2\Big]\begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N-2} \\ X_{N-1}\end{pmatrix},$$

or, more simply, as

$$A\boldsymbol{\xi} = \lambda\boldsymbol{\xi},$$

where $\lambda \triangleq -\omega^2$. This is exactly the matrix eigenvalue problem discussed at the beginning of this section, and can be solved in Matlab for the eigenvalues λ_i and the corresponding mode shapes $\boldsymbol{\xi}_i$ using the `eig` command, as illustrated in the code `wire.m` provided at the class web site. Note that, for constant α and a sufficiently large number of gridpoints, the first several eigenvalues returned by `wire.m` closely match the analytic solution $\omega_\iota = \iota\pi\alpha/L$, and the first several eigenvectors $\boldsymbol{\xi}_\iota$ are of the same shape as the analytic mode shapes $\sin(\omega_\iota x/\alpha)$.

1.3.2 Eigenvector decomposition

If all of the eigenvectors $\boldsymbol{\xi}^i$ of a given matrix A are linearly independent, then any vector $\mathbf{x}$ may be uniquely decomposed in terms of contributions parallel to each eigenvector such that

$$\mathbf{x} = S\boldsymbol{\chi}, \qquad \text{where} \qquad S = \begin{pmatrix} | & | & | & \\ \boldsymbol{\xi}^1 & \boldsymbol{\xi}^2 & \boldsymbol{\xi}^3 & \ldots \\ | & | & | & \end{pmatrix}.$$

Such change of variables often simplifies a dynamical equation significantly. For example, if a given dynamical system may be written in the form $\dot{\mathbf{x}} = A\mathbf{x}$ where the eigenvalues of A are distinct, then (by substitution of $\mathbf{x} = S\boldsymbol{\chi}$ and multiplication from the left by S^{-1}) we may write:

$$\dot{\boldsymbol{\chi}} = \Lambda\boldsymbol{\chi} \qquad \text{where} \qquad \Lambda = S^{-1}AS = \begin{pmatrix} \lambda_1 & & & 0 \\ & \lambda_2 & & \\ & & \ddots & \\ 0 & & & \lambda_n \end{pmatrix}.$$

In this representation, as Λ is diagonal, the dynamical evolution of each mode of the system is completely decoupled (*i.e.*, $\dot{\chi}_1 = \lambda_1\chi_1$, $\dot{\chi}_2 = \lambda_2\chi_2$, etc.).

1.4 Matrix norms

The norm of A, denoted $||A||$, is defined by

$$||A|| = \max_{\mathbf{x}\neq\mathbf{0}} \frac{||A\mathbf{x}||}{||\mathbf{x}||},$$

where $||\mathbf{x}||$ is the Euclidean norm of the vector $\mathbf{x}$. In other words, $||A||$ is an upper bound on the amount the matrix A can "amplify" the vector $\mathbf{x}$:

$$||A\mathbf{x}|| \leq ||A||\,||\mathbf{x}|| \qquad \forall\mathbf{x}.$$

In order to compute the norm of A, we square both sides of the expression for $||A||$, which results in

$$||A||^2 = \max_{\mathbf{x}\neq\mathbf{0}} \frac{||A\mathbf{x}||^2}{||\mathbf{x}||^2} = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\mathbf{x}^T(A^TA)\,\mathbf{x}}{\mathbf{x}^T\,\mathbf{x}}.$$

The peak value of the expression on the right is attained when $(A^TA)\,\mathbf{x} = \lambda_{\max}\,\mathbf{x}$. In other words, the matrix norm may be computed by taking the maximum eigenvalue of the matrix (A^TA). In order to compute a matrix norm in Matlab, one may type in `sqrt(max(eig(A' * A)))`, or, more simply, just call the command `norm(A)`.

1.5 Condition number

Let $A\mathbf{x} = \mathbf{b}$ and consider a small perturbation to the right-hand side. The perturbed system is written

$$A(\mathbf{x} + \delta\mathbf{x}) = (\mathbf{b} + \delta\mathbf{b}) \qquad \Rightarrow \qquad A\,\delta\mathbf{x} = \delta\mathbf{b}.$$

We are interested in bounding the change $\delta\mathbf{x}$ in the solution $\mathbf{x}$ resulting from the change $\delta\mathbf{b}$ to the right-hand side $\mathbf{b}$. Note by the definition of the matrix norm that

$$||\mathbf{x}|| \geq ||\mathbf{b}||/||A|| \qquad \text{and} \qquad ||\delta\mathbf{x}|| \leq ||A^{-1}||\,||\delta\mathbf{b}||.$$

Dividing the equation on the right by the equation on the left, we see that the relative change in $\mathbf{x}$ is bounded by the relative change in $\mathbf{b}$ according to the following:

$$\frac{||\delta\mathbf{x}||}{||\mathbf{x}||} \leq c\frac{||\delta\mathbf{b}||}{||\mathbf{b}||},$$

where $c = \|A\|\,\|A^{-1}\|$ is known as the condition number of the matrix A. If the condition number is small (say, $\leq O(10^3)$), the the matrix is referred to as well conditioned, meaning that small errors in the values of $\mathbf{b}$ on the right-hand side will result in "small" errors in the computed values of $\mathbf{x}$. However, if the condition number is large (say, $> O(10^3)$), then the matrix is poorly conditioned, and the solution $\mathbf{x}$ computed for the problem $A\mathbf{x} = \mathbf{b}$ is often unreliable.

In order to compute the condition number of a matrix in Matlab, one may type in `norm(A) * norm(inv(A))` or, more simply, just call the command `cond(A)`.

Chapter 2

Solving linear equations

Systems of linear algebraic equations may be represented efficiently in the form $A\mathbf{x} = \mathbf{b}$. For example:

$$\begin{aligned} 2u + 3v - 4w &= 0 \\ u \qquad - 2w &= 7 \\ u + \ v + \ w &= 12 \end{aligned} \qquad \Rightarrow \qquad \underbrace{\begin{pmatrix} 2 & 3 & -4 \\ 1 & 0 & -2 \\ 1 & 1 & 1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} u \\ v \\ w \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} 0 \\ 7 \\ 12 \end{pmatrix}}_{\mathbf{b}}.$$

Given an A and $\mathbf{b}$, one often needs to solve such a system for $\mathbf{x}$. Systems of this form need to be solved frequently, so these notes will devote substantial attention to numerical methods which solve this type of problem efficiently.

2.1 Introduction to the solution of $A\mathbf{x} = \mathbf{b}$

If A is diagonal, the solution may be found by inspection:

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 7 \end{pmatrix} \qquad \Rightarrow \qquad \begin{aligned} x_1 &= 5/2 \\ x_2 &= 2 \\ x_3 &= 7/4 \end{aligned}$$

If A is upper triangular, the problem is almost as easy. Consider the following:

$$\begin{pmatrix} 3 & 4 & 5 \\ 0 & 6 & 7 \\ 0 & 0 & 8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 19 \\ 8 \end{pmatrix}$$

The solution for x_3 may be found by by inspection: $x_3 = 8/8 = 1$.
Substituting this result into the equation implied by the second row, the solution for x_2 may then be found:

$$6\,x_2 + 7\underbrace{x_3}_{1} = 19 \qquad \Rightarrow \qquad x_2 = 12/6 = 2.$$

Finally, substituting the resulting values for x_2 and x_3 into the equation implied by the first row, the solution for x_1 may then be found:

$$3\,x_1 + 4\underbrace{x_2}_{2} + 5\underbrace{x_3}_{1} = 1 \qquad \Rightarrow \qquad x_1 = -12/3 = -4.$$

Thus, upper triangular matrices naturally lend themselves to solution via a march up from the bottom row. Similarly, lower triangular matrices naturally lend themselves to solution via a march down from the top row.

Note that if there is a zero in the i'th element on the main diagonal when attempting to solve a triangular system, we are in trouble. There are either:

- *zero solutions* (if, when solving the i'th equation, one reaches an equation like $1 = 0$, which cannot be made true for any value of x_i), or there are
- *infinitely many solutions* (if, when solving the i'th equation, one reaches the truism $0 = 0$, in which case the corresponding element x_i can take any value).

The matrix A is called **singular** in such cases. When studying science and engineering problems on a computer, generally one should first identify nonsingular problems before attempting to solve them numerically.

To solve a general nonsingular matrix problem $A\mathbf{x} = \mathbf{b}$, we would like to reduce the problem to a triangular form, from which the solution may be found by the marching procedure illustrated above. Such a reduction to triangular form is called Gaussian elimination. We first illustrate the Gaussian elimination procedure by example, then present the general algorithm.

2.1.1 Example of solution approach

Consider the problem

$$\begin{pmatrix} 0 & 4 & -1 \\ 1 & 1 & 1 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 1 \end{pmatrix}$$

Considering this matrix equation as a collection of rows, each representing a separate equation, we can perform simple linear combinations of the rows and still have the same system. For example, we can perform the following manipulations:

1. Interchange the first two rows:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & -1 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ 1 \end{pmatrix}$$

2. Multiply the first row by 2 and subtract from the last row:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & -1 \\ 0 & -4 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ -11 \end{pmatrix}$$

3. Add second row to third:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & -1 \\ 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ -6 \end{pmatrix}$$

This is an upper triangular matrix, so we can solve this by inspection (as discussed earlier). Alternatively (and equivalently), we continue to combine rows until the matrix becomes the identity; this is referred to as the Gauss-Jordan process:

4. Divide the last row by -2, then add the result to the second row:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \\ 3 \end{pmatrix}$$

5. Divide the second row by 4:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 2 \\ 3 \end{pmatrix}$$

6. Subtract second and third rows from the first:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad \Rightarrow \qquad \mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

The letters x_1, x_2, and x_3 clutter this process, so we may devise a shorthand **augmented matrix** in which we can conduct the same series of operations without the extraneous symbols:

$$\left(\begin{array}{ccc|c} 0 & 4 & -1 & 5 \\ 1 & 1 & 1 & 6 \\ 2 & -2 & 1 & 1 \end{array}\right) \Rightarrow \left(\begin{array}{ccc|c} 1 & 1 & 1 & 6 \\ 0 & 4 & -1 & 5 \\ 2 & -2 & 1 & 1 \end{array}\right) \Rightarrow \ldots \Rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array}\right)$$

$$\underbrace{}_{\text{A}}\ \underbrace{}_{\mathbf{b}} \qquad\qquad \underbrace{}_{\mathbf{x}}$$

An advantage of this notation is that we can solve it simultaneously for several right hand sides $\mathbf{b}^i$ comprising a right-hand-side matrix B. A particular case of interest is the several columns that make up the identity matrix. Example: construct three vectors $\mathbf{x}^1$, $\mathbf{x}^2$, and $\mathbf{x}^3$ such that

$$A\mathbf{x}^1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad A\mathbf{x}^2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad A\mathbf{x}^3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

This problem is solved as follows:

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 2 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right) \Rightarrow \left(\begin{array}{ccc|ccc} 1 & 0 & 2 & 1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right) \Rightarrow \left(\begin{array}{ccc|ccc} 1 & 0 & 2 & 1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 & 0 \\ 0 & 0 & 2 & 1 & -1 & 1 \end{array}\right) \Rightarrow$$

$$\underbrace{}_{\text{A}}\ \underbrace{}_{\text{B}}$$

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 2 & 1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 & 0 \\ 0 & 0 & 1 & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{array}\right) \Rightarrow \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{array}\right)$$

$$\underbrace{}_{\text{X}}$$

Defining $X = \begin{pmatrix} | & | & | \\ \mathbf{x}^1 & \mathbf{x}^2 & \mathbf{x}^3 \\ | & | & | \end{pmatrix}$, we have $AX = I$ by construction, and thus $X = A^{-1}$.

The above procedure is time consuming, but is just a sequence of mechanical steps. In the following section, the procedure is generalized so that we can teach the computer to do the work for us.

2.2 Gaussian elimination algorithm

This section discusses the Gaussian elimination algorithm to find the solution $\mathbf{x}$ of the system $A\,\mathbf{x} = \mathbf{b}$, where A and $\mathbf{b}$ are given. The following notation is used for the augmented matrix:

$$(A\,|\,\mathbf{b}) = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & & a_{nn} & b_n \end{array}\right).$$

2.2.1 Forward sweep

1. Eliminate everything below a_{11} (the first "pivot") in the first column:
   Let $m_{21} = -a_{21}/a_{11}$. Multiply the first row by m_{21} and add to the second row.
   Let $m_{31} = -a_{31}/a_{11}$. Multiply the first row by m_{31} and add to the third row.
   ... etc. The modified augmented matrix soon has the form

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ \hdashline 0 & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right)$$

   where all elements except those in the first row have been changed.

2. Repeat step 1 for the new (smaller) augmented matrix (highlighted by the dashed box in the last equation). The pivot for the second column is a_{22}.

... etc. The modified augmented matrix eventually takes the form

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & a_{nn} & b_n \end{array}\right)$$

Note that at each stage we need to divide by the "pivot", so it is pivotal that the pivot is nonzero. If it is not, exchange the row with the zero pivot with one of the lower rows that has a nonzero element in the pivot column. Such a procedure is referred to as "partial pivoting". We can always complete the Gaussian elimination procedure with partial pivoting if the matrix we are solving is nonsingular, *i.e.*, if the problem we are solving has a unique solution.

2.2.2 Back substitution

The process of back substitution is straightforward. Initiate with:

$$b_n \leftarrow b_n/a_{nn}.$$

Starting from $i = n-1$ and working back to $i = 1$, update the other b_i as follows:

$$b_i \leftarrow \Big(b_i - \sum_{k=i+1}^{n} a_{ik}\, b_k\Big)/a_{ii}$$

where summation notation is *not* implied. Once finished, the vector $\mathbf{b}$ contains the solution $\mathbf{x}$ of the original system $A\,\mathbf{x} = \mathbf{b}$.

2.2.3 Operation count

Let's now determine how expensive the Gaussian elimination algorithm is.

Operation count for the forward sweep:

	$\div$	$\times$	$+$
To eliminate a_{21}:	1	n	n
To eliminate entire first column:	$(n-1)$	$n(n-1)$	$n(n-1)$
To eliminate a_{32}:	1	$(n-1)$	$(n-1)$
To eliminate entire second column:	$(n-2)$	$(n-1)(n-2)$	$(n-1)(n-2)$
...etc.			

The total number of divisions is thus: $\displaystyle\sum_{k=1}^{n-1}(n-k)$

The total number of multiplications is: $\displaystyle\sum_{k=1}^{n-1}(n-k+1)(n-k)$

The total number of additions is: $\displaystyle\sum_{k=1}^{n-1}(n-k+1)(n-k)$

Two useful identities here are

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \qquad \text{and} \qquad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$

both of which may be verified by induction. Applying these identities, we see that:

The total number of divisions is: $n(n-1)/2$

The total number of multiplications is: $(n^3-n)/3$

The total number of additions is: $(n^3-n)/3$

$\Rightarrow$ For large n, the total number of flops for the forward sweep is thus $O(2n^3/3)$.

Operation count for the back substitution:

The total number of divisions is: n

The total number of multiplications is: $\displaystyle\sum_{k=1}^{n-1}(n-k) = n(n-1)/2$

The total number of additions is: $\displaystyle\sum_{k=1}^{n-1}(n-k) = n(n-1)/2$

$\Rightarrow$ For large n, the total number of flops for the back substitution is thus $O(n^2)$.

Thus, we see that the forward sweep is *much* more expensive than the back substitution for large n.

2.2.4 Matlab implementation

The following code is an efficient Matlab implementation of Gaussian elimination. The "partial pivoting" checks necessary to insure success of the approach have been omitted for simplicity, and are left as an exercise for the motivated reader. Thus, the following algorithm may fail even on nonsingular problems if pivoting is required. Note that, unfortunately, Matlab refers to the elements of `A` as `A(i,j)`, though the accepted convention is to use lowercase letters for the elements of matrices.

```
% gauss.m
% Solves the system Ax=b for x using Gaussian elimination without
% pivoting.  The matrix A is replaced by the m_ij and U on exit, and
% the vector b is replaced by the solution x of the original system.

%  -------------- FORWARD SWEEP --------------

for j = 1:n-1,   % For each column j<n,
   for i=j+1:n,  % loop through the elements a_ij below the pivot a_jj.

      % Compute m_ij.  Note that we can store m_ij in the location
      % (below the diagonal!) that a_ij used to sit without disrupting
      % the rest of the algorithm, as a_ij is set to zero by construction
      % during this iteration.

      A(i,j)     = - A(i,j) / A(j,j);

      % Add m_ij times the upper triangular part of the j'th row of
      % the augmented matrix to the i'th row of the augmented matrix.

      A(i,j+1:n) = A(i,j+1:n) + A(i,j) * A(j,j+1:n);
      b(i)       = b(i)       + A(i,j) * b(j);
   end
end

%  ------------ BACK SUBSTITUTION ------------

b(n) = b(n) / A(n,n);   % Initialize the backwards march
for i = n-1:-1:1,

   % Note that an inner product is performed at the multiplication
   % sign here, accounting for all values of x already determined:

   b(i) = ( b(i) - A(i,i+1:n) * b(i+1:n) ) / A(i,i);
end
% end gauss.m
```

2.2.5 *LU* decomposition

We now show that the forward sweep of the Gaussian elimination algorithm inherently constructs an LU decomposition of A. Through several row operations, the matrix A is transformed by the Gaussian elimination procedure into an upper triangular form, which we will call U. Furthermore, each row operation (which is simply the multiplication of one row by a number and adding the result to another row) may also be denoted by the premultiplication of A by a simple transformation matrix E_{ij}. It turns out that the transformation matrix which does the trick at each step is simply an identity matrix with the (i,j)'th component replaced by m_{ij}. For example, if we define

$$E_{21} = \begin{pmatrix} 1 & & & 0 \\ m_{21} & 1 & & \\ & & \ddots & \\ 0 & & & 1 \end{pmatrix},$$

then $E_{21}A$ means simply to multiply the first row of A by m_{21} and add it to the second row, which is exactly the first step of the Gaussian elimination process. To "undo" the multiplication of a matrix by E_{21}, we simply multiply the first row of the resulting matrix by $-m_{21}$ and add it to the second row, so that

$$E_{21}^{-1} = \begin{pmatrix} 1 & & & 0 \\ -m_{21} & 1 & & \\ & & \ddots & \\ 0 & & & 1 \end{pmatrix}.$$

The forward sweep of Gaussian elimination (without pivoting) involves simply the premultiplication of A by several such matrices:

$$\underbrace{(E_{n,n-1})(E_{n,n-2}\,E_{n-1,n-2})\cdots(E_{n2}\cdots E_{42}\,E_{32})(E_{n1}\cdots E_{31}\,E_{21})}_{E}\,A = U.$$

To "undo" the effect of this whole string of multiplications, we may simply multiply by the inverse of E, which, it is easily verified, is given by

$$E^{-1} = \begin{pmatrix} 1 & & & & 0 \\ -m_{21} & 1 & & & \\ -m_{31} & -m_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ -m_{n1} & -m_{n2} & \cdots & -m_{n,n-1} & 1 \end{pmatrix}.$$

Defining $L = E^{-1}$ and noting that $EA = U$, it follows at once that $A = LU$.

We thus see that both L and U may be extracted from the matrix that has replaced A after the forward sweep of the Gaussian elimination procedure. The following Matlab code constructs these two matrices from the value of `A` returned by `gauss.m`:

```
% extract_LU.m
% Extract the LU decomposition of A from the modified version of A
% returned by gauss.m.  Note that this routine does not make efficient
% use of memory.  It is for demonstration purposes only.

% First, construct L with 1's on the diagonal and the negative of the
% factors m_ij used during the Gaussian elimination below the diagonal.
L=eye(n);
for j=1:n-1,
   for i=j+1:n,
      L(i,j)=-A(i,j);
   end
end

% U is simply the upper-triangular part of the modified A.
U=zeros(n);
for i=1:n,
   for j=i:n,
      U(i,j)=A(i,j);
   end
end
% end extract_LU.m
```

As opposed to the careful implementation of the Gaussian elimination procedure in `gauss.m`, in which the entire operation is done "in place" in memory, the code `extract_LU.m` is not efficient with memory. It takes the information stored in the array `A` and spreads it out over two arrays `L` and `U`, constructing the LU decomposition of A. In codes for which memory storage is a limiting factor, this is probably not a good idea. Leaving the nontrivial components of `L` and `U` in a single array `A`, though it makes the code a bit difficult to interpret, is an effective method of saving memory space.

Once we have the LU decomposition of A (e.g., once we run the full Gaussian elimination procedure once), we can solve a system with a new right hand side $A\,\mathbf{x} = \mathbf{b}'$ with a very inexpensive algorithm. We note that we may first solve an intermediate problem

$$L\,\mathbf{y} = \mathbf{b}'$$

for the vector $\mathbf{y}$. As L is (lower) triangular, this system can be solved inexpensively ($O(n^2)$ flops). Once $\mathbf{y}$ is found, we may then solve the system

$$U\,\mathbf{x} = \mathbf{y}$$

for the vector $\mathbf{x}$. As U is (upper) triangular, this system can also be solved inexpensively ($O(n^2)$ flops). Substituting the second equation into the first, and noting that $A = LU$, we see that what we have solved by this two-step process is equivalent to solving the desired problem $A\,\mathbf{x} = \mathbf{b}'$, but at a significantly lower cost ($O(2n^2)$ flops instead of $O(2n^3/3)$ flops) because we were able to leverage the LU decomposition of the matrix A. Thus, if you are going to get several right-hand-side vectors $\mathbf{b}'$ with A remaining fixed, it is a very good idea to reuse the LU decomposition of A rather than repeatedly running the Gaussian elimination routine from scratch.

2.2.6 Testing the Gaussian elimination code

The following code tests `gauss.m` and `extract_LU.m` with random A and $\mathbf{b}$.

```
% test_gauss.m
echo on
% This code tests the Gaussian elimination and LU decomposition
% routines.  First, create a random A and b.

clear, n=4;  A=rand(n),  b=rand(n,1),  pause

% Recall that A and b are destroyed in gauss.m.
% Let's hang on to them here.

Asave=A;  bsave=b;

% Run the Gaussian elimination code to find the solution x of
% Ax=b, and extract the LU decomposition of A.

echo off, gauss,  extract_LU,  echo on,  pause

% Now let's see how good x is.  If we did well, the value of A*x
% should be about the same as the value of b.
% Recall that the solution x is returned in b by gauss.m.

x=b,  Ax = Asave*x, b=bsave, pause

% Now let's see how good L and U are.  If we did well, L should be
% lower triangular with 1's on the diagonal, U should be upper
% triangular, and the value of L*U should be about the same as
% the value of A.
% Note that the product L*U is not done efficiently below, as both
% L and U have structure which is not being leveraged.

L, U, LU=L*U, A=Asave
% end test_gauss.m
```

2.2.7 Pivoting

As you run the code `test_gauss.m` on several random matrices A, you may be lulled into a false sense of security that pivoting isn't all that important. I will shatter this dream for you with homework #1. Just because a routine works well on several random matrices does not mean it will work well in general!

As mentioned earlier, any nonsingular system may be solved by the Gaussian elimination procedure if partial pivoting is implemented. Recall that partial pivoting involves simply swapping rows whenever a zero pivot is encountered. This can sometimes lead to numerical inaccuracies, as *small* (but nonzero) pivots may be encountered by this algorithm. This can lead to subsequent row combinations which involve the difference of two large numbers which are almost equal. On a computer, in which all numbers are represented with only finite precision, taking the difference of two numbers which are almost equal can lead to significant magnification of round-off error. To alleviate this situation, we can develop a procedure which swaps *columns*, in addition to swapping the rows, to maximize the size of the pivot at each step. One has to bookkeep carefully when swapping columns because the elements of the solution vector are also swapped.

2.3 Thomas algorithm

This section discusses the Thomas algorithm that finds the solution $\mathbf{x}$ of the system $A\,\mathbf{x} = \mathbf{g}$, where A and $\mathbf{g}$ are given with A assumed to be tridiagonal and diagonally dominant[1]. The algorithm is based on the Gaussian elimination algorithm of the previous section but capitalizes on the structure of A. The following notation is used for the augmented matrix:

$$(A\,|\,\mathbf{g}) = \left(\begin{array}{cccccc|c} b_1 & c_1 & & & & 0 & g_1 \\ a_2 & b_2 & c_2 & & & & g_2 \\ & a_3 & b_3 & c_3 & & & g_3 \\ & & \ddots & \ddots & \ddots & & \vdots \\ & & & a_{n-1} & b_{n-1} & c_{n-1} & g_{n-1} \\ 0 & & & & a_n & b_n & g_n \end{array}\right).$$

2.3.1 Forward sweep

1. Eliminate everything below b_1 (the first pivot) in the first column:
   Let $m_2 = -a_2/b_1$. Multiply the first row by m_2 and add to the second row.

2. Repeat step 1 for the new (smaller) augmented matrix, as in the Gaussian elimination procedure. The pivot for the second column is b_2.

... Continue iterating to eliminate the a_i until the modified augmented matrix takes the form

$$\left(\begin{array}{cccccc|c} b_1 & c_1 & & & & 0 & g_1 \\ 0 & b_2 & c_2 & & & & g_2 \\ & 0 & b_3 & c_3 & & & g_3 \\ & & \ddots & \ddots & \ddots & & \vdots \\ & & & 0 & b_{n-1} & c_{n-1} & g_{n-1} \\ 0 & & & & 0 & b_n & g_n \end{array}\right)$$

Again, at each stage it is pivotal that the pivot is nonzero. A good numerical discretization of a differential equation will result in matrices A which are diagonally dominant, in which case the pivots are always nonzero and we may proceed without worrying about the tedious (and numerically expensive) chore of pivoting.

[1]Diagonal dominance means that the magnitude of the element on the main diagonal in each row is larger than the sum of the magnitudes of the other elements in that row.

2.3.2 Back substitution

As before, initiate the back substitution with:

$$g_n \leftarrow g_n/b_n.$$

Starting from $i = n-1$ and working back to $i = 1$, update the other g_i as follows:

$$g_i \leftarrow \Big(g_i - c_i\, g_{i+1}\Big)/b_i$$

where summation notation is *not* implied. Once finished, the vector $\mathbf{g}$ contains the solution $\mathbf{x}$ of the original system $A\,\mathbf{x} = \mathbf{g}$.

2.3.3 Operation count

Let's now determine how expensive the Thomas algorithm is.

Operation count for the forward sweep:

	$\div$	$\times$	$+$
To eliminate a_2:	1	2	2
To eliminate entire subdiagonal:	$(n-1)$	$2(n-1)$	$2(n-1)$

$\Rightarrow$ For large n, the total number of flops for the forward sweep is thus $O(5n)$.

Operation count for the back substitution:

The total number of divisions is:	n
The total number of multiplications is:	$(n-1)$
The total number of additions is:	$(n-1)$

$\Rightarrow$ For large n, the total number of flops for the back substitution is thus $O(3n)$.

This is a lot cheaper than Gaussian elimination!

2.3.4 Matlab implementation

The following code is an efficient Matlab implementation of the Thomas algorithm. The matrix A is assumed to be diagonally dominant so that the need for pivoting is obviated.

```
% thomas.m
% Solves the system Ax=g for x using the Thomas algorithm,
% assuming A is tridiagonal and diagonally dominant.  It is
% assumed that (a,b,c,g) are previously-defined vectors of
% length n, where a is the subdiagonal, b is the main diagonal,
% and c is the superdiagonal of the matrix A.  The vectors
% (a,b,c) are replaced by the m_i and U on exit, and the vector
% g is replaced by the solution x of the original system.

%  -------------- FORWARD SWEEP --------------

for j = 1:n-1,   % For each column j<n,

   % Compute m_(j+1).  Note that we can put m_(j+1) in the location
   % (below the diagonal!) that a_(j+1) used to sit without disrupting
   % the rest of the algorithm, as a_(j+1) is set to zero by construction
   % during this iteration.

   a(j+1)     = - a(j+1) / b(j);

   % Add m_(j+1) times the upper triangular part of the j'th row of
   % the augmented matrix to the (j+1)'th row of the augmented
   % matrix.

   b(j+1) = b(j+1) + a(j+1) * c(j);
   g(j+1) = g(j+1) + a(j+1) * g(j);
end

%  ------------ BACK SUBSTITUTION ------------

g(n) = g(n) / b(n);
for i = n-1:-1:1,
   g(i) = ( g(i) - c(i) * g(i+1) ) / b(i);
end
% end thomas.m
```

2.3.5 *LU* decomposition

The forward sweep of the Thomas algorithm again inherently constructs an LU decomposition of A. This decomposition may (inefficiently) be constructed with the code `extract_LU.m` used for the Gaussian elimination procedure. Note that most of the elements of both L and U are zero.

Once we have the LU decomposition of A, we can solve a system with a new right hand side $A\,\mathbf{x} = \mathbf{g}'$ with a two-step procedure as before. The cost of efficiently solving

$$L\,\mathbf{y} = \mathbf{g}'$$

for the vector $\mathbf{y}$ is $O(2n)$ flops (similar to the cost of the back substitution in the Thomas algorithm, but noting that the divisions are not required because the diagonal elements are unity), and the cost of efficiently solving

$$U\,\mathbf{x} = \mathbf{y}$$

for the vector $\mathbf{x}$ is $O(3n)$ flops (the same as the cost of back substitution in the Thomas algorithm). Thus, solving $A\,\mathbf{x} = \mathbf{g}'$ by reusing the LU decomposition of A costs $O(5n)$ flops, whereas solving it by the Thomas algorithm costs $O(8n)$ flops. This is a measurable savings which should not be discounted.

2.3.6 Testing the Thomas code

The following code tests `thomas.m` with `extract_LU.m` for random A and $\mathbf{g}$.

```
% test_thomas.m
echo on
% This code tests the implementation of the Thomas algorithm.
% First, create a random a, b, c, and g.
clear, n=4;  a=rand(n,1);  b=rand(n,1);  c=rand(n,1);  g=rand(n,1);

% Construct A.  Note that this is an insanely inefficient way of
% storing the nonzero elements of A, and is done for demonstration
% purposes only.
A = diag(a(2:n),-1) + diag(b,0) + diag(c(1:n-1),1)

% Hang on to A and g for later use.
Asave=A;  gsave=g;

% Run the Thomas algorithm to find the solution x of Ax=g,
% put the diagonals back into matrix form, and extract L and U.
echo off,  thomas
A = diag(a(2:n),-1) + diag(b,0) + diag(c(1:n-1),1);
extract_LU,  echo on,  pause

% Now let's see how good x is.  If we did well, the value of A*x
% should be about the same as the value of g.
% Recall that the solution x is returned in g by thomas.m.
x=g,  Ax = Asave*x, g=gsave, pause

% Now let's see how good L and U are.  If we did well, L should be
% lower triangular with 1's on the diagonal, U should be upper
% triangular, and the value of L*U should be about the same as
% the value of A.  Note the structure of L and U.

L, U, LU=L*U, A=Asave
% end test_thomas.m
```

2.3.7 Parallelization

The Thomas algorithm, which is $O(8n)$, is very efficient and, thus, widely used.

Many modern computers achieve their speed by finding many things to do at the same time. This is referred to as parallelization of an algorithm. However, each step of the Thomas algorithm depends upon the previous step. For example, iteration `j=7` of the forward sweep must be complete before iteration `j=8` can begin. Thus, the Thomas algorithm does not parallelize, which is unfortunate. In many problems, however, one can find several different systems of the form $A\,\mathbf{x} = \mathbf{g}$ and work on them all simultaneously to achieve the proper "load balancing" of a well-parallelized code. One must always use great caution when parallelizing a code to not accidentally reference a variable before it is actually calculated.

2.4 Review

We have seen that Gaussian elimination is a general tool that may be used whenever A is nonsingular (regardless of the structure of A), and that partial pivoting (exchanging rows) is sometimes required to make it work. Gaussian elimination is also a means of obtaining an LU decomposition of A, with which more efficient solution algorithms may be developed if several systems of the form $A\,\mathbf{x} = \mathbf{b}$ must be solved where A is fixed and several values of $\mathbf{b}$ will be encountered.

Most of the systems we will encounter in our numerical algorithms, however, will *not* be full. When the equations and unknowns of the system are enumerated in such a manner that the nonzero elements lie only near the main diagonal, resulting in what we call a banded matrix, more efficient solution techniques are available. A particular example of interest is tridiagonal systems, which are amenable to very efficient solution via the Thomas algorithm.

Chapter 3

Solving nonlinear equations

Many problems in engineering require solution of nonlinear algebraic equations. In what follows, we will show that the most popular numerical methods for solving such equations involve linearization, which leads to repeatedly solving linear systems of the form $A\mathbf{x} = \mathbf{b}$. Solution of nonlinear equations always requires iterations. That is, unlike linear systems where, if solutions exist, they can be obtained exactly with Gaussian elimination, with nonlinear equations, only approximate solutions are obtained. However, in principle, the approximations can be improved by increasing the number of iterations.

A single nonlinear equation may be written in the form $f(x) = 0$. The objective is to find the value(s) of x for which f is zero. In terms of a geometrical interpretation, we want to find the crossing point(s), $x = x^{\text{opt}}$, where $f(x)$ crosses the x-axis in a plot of f vs. x. Unfortunately, there are no systematic methods to determine when a nonlinear equation $f(x) = 0$ will have zero, one, or several values of x which satisfy $f(x) = 0$.

3.1 The Newton-Raphson method for nonlinear root finding

3.1.1 Scalar case

Iterative techniques start from an initial "guess" for the solution and seek successive improvements to this guess. Suppose the initial guess is $x = x^{(0)}$. Linearize $f(x)$ about $x^{(0)}$ using the Taylor series expansion

$$f(x) = f(x^{(0)}) + (x - x^{(0)})f'(x^{(0)}) + \dots \tag{3.1}$$

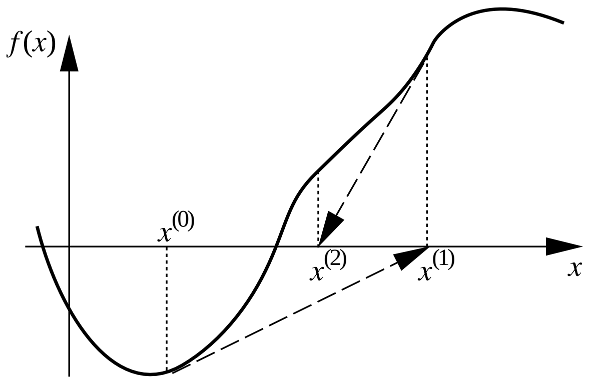
Instead of finding the roots of the resulting polynomial of degree ∞, we settle for the root of the approximate polynomial consisting of the first two terms on the right-hand side of (3.1). Let $x^{(1)}$ be this root, which can be easily obtained (when $f'(x^{(0)}) \neq 0$) by taking $f(x) = 0$ and solving (3.1) for x, which results in

$$x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})}.$$

Thus, starting with $x^{(0)}$, the next approximation is $x^{(1)}$. In general, successive approximations are obtained from

$$x^{(j+1)} = x^{(j)} - \frac{f(x^{(j)})}{f'(x^{(j)})} \qquad \text{for} \qquad j = 0, 1, 2, \dots \tag{3.2}$$

until (hopefully) we obtain a solution that is sufficiently accurate. The geometrical interpretation of the method is shown below.



The function f at point $x^{(0)}$ is approximated by a straight line which is tangent to f with slope $f'(x^{(0)})$. The intersection of this line with the x-axis gives $x^{(1)}$, the function at $x^{(1)}$ is approximated by a tangent straight line which intersects the x-axis at $x^{(2)}$, etc.

3.1.2 Quadratic convergence

We now show that, once the solution is near the exact value $x = x^{\text{opt}}$, the Newton-Raphson method converges quadratically. Let $x^{(j)}$ indicate the solution at the j^{th} iteration. Consider the Taylor series expansion

$$f(x^{\text{opt}}) = 0 \approx f(x^{(j)}) + (x^{\text{opt}} - x^{(j)})f'(x^{(j)}) + \frac{(x^{\text{opt}} - x^{(j)})^2}{2} f''(x^{(j)}).$$

If $f'(x^{(j)}) \neq 0$, divide by $f'(x^{(j)})$,

$$x^{(j)} - x^{\text{opt}} \approx \frac{f(x^{(j)})}{f'(x^{(j)})} + \frac{(x^{\text{opt}} - x^{(j)})^2}{2} \frac{f''(x^{(j)})}{f'(x^{(j)})}.$$

Combining this with the Newton-Raphson formula (3.2) leads to

$$x^{(j+1)} - x^{\text{opt}} \approx \frac{(x^{(j)} - x^{\text{opt}})^2}{2} \frac{f''(x^{(j)})}{f'(x^{(j)})}. \tag{3.3}$$

Defining the error at iteration j as $\epsilon^{(j)} = \left| x^{(j)} - x^{\text{opt}} \right|$, the error at the iteration $j+1$ is related to the error at the iteration j by

$$\epsilon^{(j+1)} \approx \frac{1}{2} \left| \frac{f''(x^{(j)})}{f'(x^{(j)})} \right| \left(\epsilon^{(j)} \right)^2.$$

That is, convergence is quadratic.

Note that convergence is guaranteed only if the initial guess is fairly close to the exact root; otherwise, the neglected higher-order terms dominate the above expression and we may encouter divergence. If f' is near zero at the root, we will also have problems.

3.1.3 Multivariable case—systems of nonlinear equations

A system of n nonlinear equations in n unknowns can be written in the general form:

$$f_i(x_1, x_2, \ldots, x_n) = 0 \qquad \text{for} \qquad i = 1, 2, \ldots, n,$$

or, more compactly, as $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. As usual, subscripts denote vector components and boldface denotes whole vectors. Generalization of the Newton-Raphson method to such systems is achieved by using the multi-dimensional Taylor series expansion:

$$\begin{aligned} f_i(x_1, x_2, \ldots, x_n) &= f_i(x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}) + (x_1 - x_1^{(0)}) \left.\frac{\partial f_i}{\partial x_1}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} + (x_2 - x_2^{(0)}) \left.\frac{\partial f_i}{\partial x_2}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} + \ldots \\ &= f_i(x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}) + \sum_{j=1}^{n} (x_j - x_j^{(0)}) \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} + \ldots \end{aligned}$$

where, as in the previous section, superscripts denote iteration number. Thus, for example, the components of the initial guess vector $\mathbf{x}^{(0)}$ are $x_i^{(0)}$ for $i = 1, 2, \ldots, n$. Let $\mathbf{x}^{(1)}$ be the solution of the above equation with only the linear terms retained and with $f_i(x_1, x_2, \ldots, x_n)$ set to zero as desired. Then

$$\sum_{j=1}^{n} \underbrace{\left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}}}_{a_{ij}^{(0)}} \underbrace{(x_j^{(1)} - x_j^{(0)})}_{h_j^{(1)}} = -f_i(\mathbf{x}^{(0)}) \qquad \text{for} \qquad i = 1, 2, \ldots, n. \tag{3.4}$$

Equation (3.4) constitutes n linear equations for the n components of $\mathbf{h}^{(1)} = \mathbf{x}^{(1)} - \mathbf{x}^{(0)}$, which may be considered as the desired update to $\mathbf{x}^{(0)}$. In matrix notation, we have

$$A^{(0)} \mathbf{h}^{(1)} = -\mathbf{f}(\mathbf{x}^{(0)}) \qquad \text{where} \qquad a_{ij}^{(0)} = \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}}.$$

Note that the elemenets of the Jacobian matrix $A^{(0)}$ are evaluated at $\mathbf{x} = \mathbf{x}^{(0)}$. The solution at the first iteration, $\mathbf{x}^{(1)}$, is obtained from

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{h}^{(1)}.$$

Successive approximations are obtained from

$$\begin{aligned} A^{(k)} \mathbf{h}^{(k+1)} &= -\mathbf{f}(\mathbf{x}^{(k)}) \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \mathbf{h}^{(k+1)} \end{aligned} \qquad \text{where} \qquad a_{ij}^{(k)} = \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(k)}}.$$

Note that the elements of the Jacobian matrix $A^{(k)}$ are evaluated at $\mathbf{x} = \mathbf{x}^{(k)}$.

As an example, consider the nonlinear system of equations

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^2 + 3 \cos x_2 - 1 \\ x_2 + 2 \sin x_1 - 2 \end{pmatrix} = 0.$$

The Jacobian matrix is given by

$$A^{(k)} = \left.\begin{pmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} \end{pmatrix}\right|_{\mathbf{x}=\mathbf{x}^{(k)}} = \begin{pmatrix} 2x_1^{(k)} & -3 \sin x_2^{(k)} \\ 2 \cos x_1^{(k)} & 1 \end{pmatrix}.$$

Let the initial guess be:

$$\mathbf{x}^{(0)} = \begin{pmatrix} x_1^{(0)} \\ \\ x_2^{(0)} \end{pmatrix}.$$

The function $\mathbf{f}$ and Jacobian A are first evaluated at $\mathbf{x} = \mathbf{x}^{(0)}$. Next, the following system of equations is solved for $\mathbf{h}^{(1)}$

$$A^{(0)}\mathbf{h}^{(1)} = -\mathbf{f}(\mathbf{x}^{(0)}).$$

We then update $\mathbf{x}$ according to

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{h}^{(1)}.$$

The function $\mathbf{f}$ and Jacobian A are then evaluated at $\mathbf{x} = \mathbf{x}^{(1)}$, and we solve

$$\begin{aligned} A^{(1)}\mathbf{h}^{(2)} &= -\mathbf{f}(\mathbf{x}^{(1)}) \\ \mathbf{x}^{(2)} &= \mathbf{x}^{(1)} + \mathbf{h}^{(2)}. \end{aligned}$$

The process continues in an iterative fashion until convergence. A numerical solution to this example nonlinear system is found in the following sections.

3.1.4 Matlab implementation

The following code is a modular Matlab implementation of the Newton-Raphson method. By using modular programming style, we are most easily able to adapt segments of code written here for later use on different nonlinear systems.

```
% newt.m
% Given an initial guess for the solution x and the auxiliary functions
% compute_f.m and compute_A.m to compute a function and the corresponding
% Jacobian, solve a nonlinear system using Newton-Raphson.  Note that f may
% be a scalar function or a system of nonlinear functions of any dimension.
res=1;  i=1;
while (res>1e-10)
   f=compute_f(x);  A=compute_A(x);        % Compute function and Jacobian
   res=norm(f);                            % Compute residual
   x_save(i,:)=x';                         % Save x, f, and the residual
   f_save(i,:)=f';  res_save(i,1)=res;
   x=x-(A\f);                              % Solve system for next x
   i=i+1;                                  % Increment index
end
evals=i-1;
% end newt.m
```

The auxiliary functions (listed below) are written as functions rather than as simple Matlab scripts, and all variables are passed in as arguments and pass back out via the function calls. Such handshaking between subprograms eases debugging considerably when your programs become large.

```
function [f] = compute_f(x)
f=[x(1)*x(1)+3*cos(x(2))-1;          % Evaluate the function f(x).
      x(2)+2*sin(x(1))-2   ];
% end function compute_f.m

function [A] = compute_A(x)
A=[  2*x(1)      -3*sin(x(2));       % Evaluate the Jacobian A
     2*cos(x(1))      1       ];     % of the function in compute_f
% end function compute_A.m
```

To prevent confusion with other cases presented in these notes, these case-specific functions are stored at the class web site as `compute_f.m.0` and `compute_A.m.0`, and must be saved as `compute_f.m` and `compute_A.m` before the test code in the following section will run.

3.1.5 Dependence of Newton-Raphson on a good initial guess

The following code tests our implementation of the Newton-Raphson algorithm.

```
% test_newt.m
% Tests Newton's algorithm for nonlinear root finding.
% First, provide a sufficiently accurate initial guess for the root
clear;  format long;  x=init_x;
% Find the root with Newton-Raphson and print the convergence
newt,  x_save,  res_save
% end test_newt.m

function [x] = init_x
x=[0; 1];                            % Initial guess for x
% end function init_x.m
```

As before, the case-specific function to compute the initial guess is stored at the class web site as `init_x.m.0`, and must be saved as `init_x.m` before the test code will run.

Typical results when applying the Newton-Raphson approach to solve a nonlinear system of equations are shown below. It is seen that a "good" (lucky?) choice of initial conditions will converge very rapidly to the solution with this scheme. A poor choice will not converge smoothly, and may not converge at all—this is a major drawback to the Newton-Raphson approach. As nonlinear systems do not necessarily have unique solutions, the final root to which the system converges is dependent on the choice of initial conditions. It is apparent in the results shown that, in this example, the solution is not unique.

A good initial guess

iteration	x_1	x_2	residual
1	0.0000	1.0000	1.17708342963828
2	0.3770	1.2460	0.10116915143065
3	0.3689	1.2788	0.00043489424427
4	0.3690	1.2787	0.00000000250477
5	0.3690	1.2787	0.00000000000000

A poor initial guess

iteration	x_1	x_2
1	0.00	-1.00
2	1.62	-1.25
3	-0.11	-0.18
4	2.42	-2.82
5	1.57	-0.60
6	-0.01	0.00
7	553.55	-1105.01
8	279.87	443.99
9	143.16	-261.08
10	72.58	34.77
11	36.75	-65.29
⋮	⋮	⋮
23	-1.82	4.39
24	-1.79	3.96
25	-1.74	3.97
26	-1.74	3.97

3.2 Bracketing approaches for scalar root finding

It was shown in the previous section that the Newton-Raphson technique is very efficient for finding the solution of both scalar nonlinear equations of the form $f(x) = 0$ and multivariable nonlinear systems of equations of the form $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ when good initial guesses are available. Unfortunately, good initial guesses are not always readily available. When they are not, it is desirable to seek the roots of such equations by more pedestrian means which guarantee success. The techniques we will present in this section, though they can not achieve quadratic convergence, are guaranteed to converge to a solution of a scalar nonlinear equation so long as:

a) the function is continuous, and
b) an initial bracketing pair may be found.

They also have the added benefit that they are based on function evaluations alone (*i.e.*, you don't need to write `compute_A.m`), which makes them simple to implement. Unfortunately, these techniques are based on a bracketing principle that does not extend readily to multi-dimensional functions.

3.2.1 Bracketing a root

The class of scalar nonlinear functions we will consider are continuous. Our first task is to find a pair of values for x which bracket the minimum, *i.e.*, we want to find an $x^{(\text{lower})}$ and an $x^{(\text{upper})}$ such that $f(x^{(\text{lower})})$ and $f(x^{(\text{upper})})$ have opposite signs. This may often be done by hand with a minor amount of trial and error. At times, however, it is convenient to have an automatic procedure to find such a bracketing pair. For example, for functions which have opposite sign for sufficiently large and small arguments, a simple approach is to start with an initial guess for the bracket and then geometrically increase the distance between these points until a bracketing pair is found. This may be implemented with the following codes (or variants thereof):

```
function [x_lower,x_upper] = find_brack
[x_lower,x_upper] = init_brack;
while compute_f(x_lower) * compute_f(x_upper) > 0
    interval=x_upper-x_lower;
    x_upper =x_upper+0.5*interval;  x_lower =x_lower-0.5*interval;
end
% end function find_brack.m
```

The auxiliary functions used by this code may be written as

```
function [x_lower,x_upper] = init_brack
x_lower=-1;    x_upper=1;       % Initialize a guess for the bracket.
% end function init_brack.m

function [f] = compute_f(x)
f=x^3 + x^2 - 20*x + 50;        % Evaluate the function f(x).
% end function compute_f.m
```

To prevent confusion with other cases presented in these notes, these case-specific functions are stored at the class web site as `init_brack.m.1` and `compute_f.m.1`.

3.2.2 Refining the bracket - bisection

Once a bracketing pair is found, the task that remains is simply to refine this bracket until a desired degree of precision is obtained. The most straightforward approach is to repeatedly chop the interval in half, keeping those two values of x that bracket the root. The convergence of such an algorithm is linear; at each iteration, the bounds on the solution are reduced by exactly a factor of 2.

The bisection algorithm may be implemented with the following code:

```
% bisection.m
f_lower=compute_f(x_lower);  f_upper=compute_f(x_upper);   evals=2;
interval=x_upper-x_lower;    i=1;
while interval > x_tol
    x = (x_upper + x_lower)/2;
    f = compute_f(x);    evals=evals+1;
    res_save(i,1)=norm(f);
    plot(x,f,'ko');
    if f_lower*f < 0
        x_upper = x;
        f_upper = f;
    else
        x_lower = x;
        f_lower = f;
    end
    interval=interval/2;
    i=i+1;  pause;
end
x = (x_upper + x_lower)/2;
% end bisection.m
```

3.2.3 Refining the bracket - false position

A technique that is usually slightly faster than the bisection technique, but that retains the safety of maintaining and refining a bracketing pair, is to compute each new point by a numerical approximation to the Newton-Raphson formula (3.2) such that

$$x^{(\text{new})} = x^{(\text{lower})} - \frac{f(x^{(\text{lower})})}{\delta f/\delta x}, \tag{3.5}$$

where the quantity $\delta f/\delta x$ is a simple difference approximation to the slope of the function f

$$\frac{\delta f}{\delta x} = \frac{f(x^{(\text{upper})}) - f(x^{(\text{lower})})}{x^{(\text{upper})} - x^{(\text{lower})}}.$$

As shown in class, this approach sometimes stalls, so it is useful to put in an ad hoc check to keep the progress moving.

```
% false_pos.m
f_lower=compute_f(x_lower);  f_upper=compute_f(x_upper);   evals=2;
interval=x_upper-x_lower;    i=1;
while interval > x_tol
   fprime = (f_upper-f_lower)/interval;
   x = x_lower - f_lower / fprime;

   % Ad hoc check:  stick with bisection technique if updates
   % provided by false position technique are too small.
   tol1 = interval/10;
   if ((x-x_lower) < tol1 | (x_upper-x) < tol1)
      x = (x_lower+x_upper)/2;
   end

   f = compute_f(x);  evals = evals+1;
   res_save(i,1)=norm(f);
   plot([x_lower x_upper],[f_lower f_upper],'k--',x,f,'ko');
   if f_lower*f < 0
       x_upper = x;
       f_upper = f;
   else
       x_lower = x;
       f_lower = f;
   end
   interval=x_upper-x_lower;
   i=i+1;     pause;
end
x = (x_upper + x_lower)/2;
% end false_pos.m
```

3.2.4 Testing the bracketing algorithms

The following code tests the bracketing algorithms for scalar nonlinear rootfinding and compares with the Newton-Raphson algorithm. Convergence of the false position algorithm is usually found to be faster than bisection, and both are safer (and slower) than Newton-Raphson. Don't forget to update `init_x.m` and `compute_A.m` appropriately (*i.e.*, with `init_x.m.1` and `compute_A.m.1`) in order to get Newton-Raphson to work. You might have to fiddle with `init_x.m` in order to obtain convergence of the Newton-Raphson algorithm.

```
% test_brack.m
% Tests the bisection, false position, and Newton routines for
% finding the root of a scalar nonlinear function.

% First, compute a bracket of the root.
clear;  [x_lower,x_upper] = find_brack;

% Prepare to make some plots of the function on this interval
xx=[x_lower : (x_upper-x_lower)/100 : x_upper];
for i=1:101, yy(i)=compute_f(xx(i)); end
x1=[x_lower x_upper];  y1=[0 0];

x_tol = 0.0001;           % Set tolerance desired for x.
x_lower_save=x_lower;  x_upper_save=x_upper;

fprintf('\n Now testing the bisection algorithm.\n');
figure(1); clf; plot(xx,yy,'k-',x1,y1,'b-');  hold on; grid;
title('Convergence of the bisection routine')
bisection
evals,  hold off

fprintf(' Now testing the false position algorithm.\n');
figure(2); clf; plot(xx,yy,'k-',x1,y1,'b-');  hold on; grid;
title('Convergence of false position routine')
x_lower=x_lower_save;
x_upper=x_upper_save;
false_pos
evals,  hold off

fprintf(' Now testing the Newton-Raphson algorithm.\n');
figure(3); clf; plot(xx,yy,'k-');  hold on; grid;
title('Convergence of the Newton-Raphson routine')
x=init_x;
newt
% Finally, an extra bit of plotting stuff to show what happens:
x_lower_t=min(x_save);  x_upper_t=max(x_save);
xx=[x_lower_t : (x_upper_t-x_lower_t)/100 : x_upper_t];
for i=1:101, yy(i)=compute_f(xx(i)); end
plot(xx,yy,'k-',x_save(1),f_save(1),'ko'); pause
for i=1:size(x_save)-1
```

```
  plot([x_save(i) x_save(i+1)],[f_save(i) 0],'k--', ...
       x_save(i+1),f_save(i+1),'ko')
  pause
end;
evals,  hold off

% end test_brack.m
```

Chapter 4

Interpolation

One often encounters the problem of constructing a smooth curve which passes through discrete data points. This situation arises when developing differentiation and integration strategies, as we will discuss in the following chapters, as well as when simply estimating the value of a function between known values. In this handout, we will present two techniques for this procedure: polynomial (Lagrange) interpolation and cubic spline interpolation.

Note that the process of interpolation passes a curve exactly through each data point. This is sometimes exactly what is desired. However, if the data is from an experiment and has any appreciable amount of uncertainty, it is best to use a least-squares technique to fit a low-order curve in the general vicinity of several data points. This technique minimizes a weighted sum of the square distance from each data points to this curve without forcing the curve to pass through each data point individually. Such an approach generally produces a much smoother curve (and a more physically-meaningful result) when the measured data is noisy.

4.1 Lagrange interpolation

Suppose we have a set of $n + 1$ data points $\{x_i, y_i\}$. The process of Lagrange interpolation involves simply fitting an n'th degree polynomial (*i.e.*, a polynomial with $n + 1$ degrees of freedom) exactly through this data. There are two ways of accomplishing this: solving a system of $n+1$ simultaneous equations for the $n + 1$ coefficients of this polynomial, and constructing the polynomial directly in factored form. We will present both of these techniques.

4.1.1 Solving $n + 1$ equations for the $n + 1$ coefficients

Consider the polynomial

$$P(x) = a_o + a_1\, x + a_2\, x^2 + \ldots + a_n\, x^n.$$

At each point x_i, this polynomial must take the value y_i, *i.e.*,

$$y_i = P(x_i) = a_o + a_1\, x_i + a_2\, x_i^2 + \ldots + a_n\, x_i^n \qquad \text{for} \qquad i = 0, 1, 2, \ldots, n.$$

In matrix form, we may write this system as

$$\underbrace{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}}_{V} \underbrace{\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix}}_{\mathbf{a}} = \underbrace{\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}}_{\mathbf{y}}.$$

This system is of the form $V\mathbf{a} = \mathbf{y}$, where V is commonly referred to as Vandermonde's matrix, and may be solved for the vector $\mathbf{a}$ containing the coefficients a_i of the desired polynomial. Unfortunately, Vandermonde's matrix is usually quite poorly conditioned, and thus this technique of finding an interpolating polynomial is unreliable at best.

4.1.2 Constructing the polynomial directly

Consider the n'th degree polynomial given by the factored expression

$$L_\kappa(x) = \alpha_\kappa (x - x_0)(x - x_1)\cdots(x - x_{\kappa-1})(x - x_{\kappa+1})\cdots(x - x_n) = \alpha_\kappa \prod_{\substack{i=0 \\ i\neq\kappa}}^{n} (x - x_i),$$

where α_κ is a constant yet to be determined. This expression (by construction) is equal to zero if x is equal to any of the data points except x_κ. In other words, $L_\kappa(x_i) = 0$ for $i \neq \kappa$. Choosing α_κ to normalize this polynomial at $x = x_\kappa$, we define

$$\alpha_\kappa = \left[\prod_{\substack{i=0 \\ i\neq\kappa}}^{n} (x_\kappa - x_i) \right]^{-1},$$

which results in the very useful relationship

$$L_\kappa(x_i) = \delta_{i\kappa} = \begin{cases} 1 & i = \kappa, \\ 0 & i \neq \kappa. \end{cases}$$

Scaling this result, the polynomial $y_\kappa L_\kappa(x)$ (no summation implied) passes through zero at every data point $x = x_i$ except at $x = x_\kappa$, where it has the value y_κ. Finally, a linear combination of $n+1$ of these polynomials

$$P(x) = \sum_{k=0}^{n} y_k L_k(x)$$

provides an n'th degree polynomial which exactly passes through *all* of the data points, by construction.

A code which implements this constructive technique to determine the interpolating polynomial is given in the following subsection.

Unfortunately, high-order polynomials tend to wander wildly between the data points even if the data appears to be fairly regular, as shown in Figure 4.1. Thus, Lagrange interpolation should be thought of as dangerous for anything more than a few data points and avoided in favor of other techniques, such as spline interpolation.

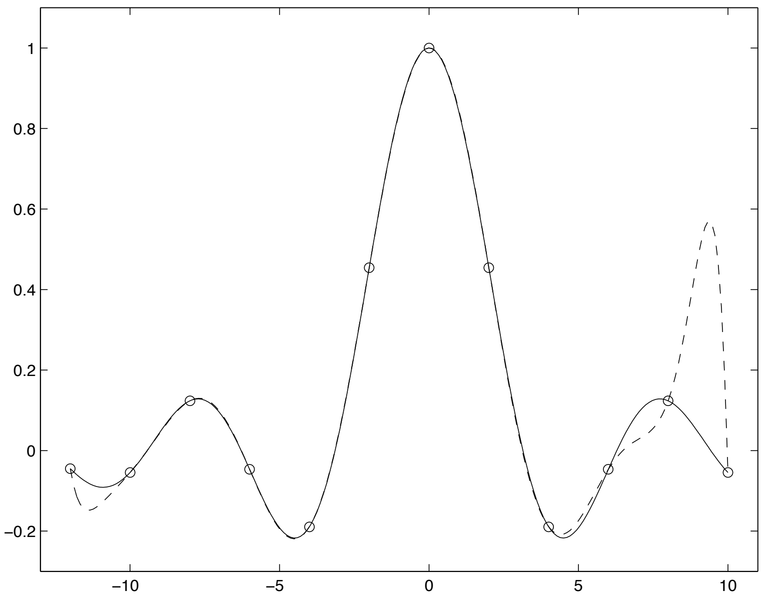


Figure 4.1: The interpolation problem: ——— a continuous curve representing the function of interest; ∘ several points on this curve; ---- the Lagrange interpolation of these points.

4.1.3 Matlab implementation

```
function [p] = lagrange(x,x_data,y_data)
% Computes the Lagrange polynomial p(x) that passes through the given
% data {x_data,y_data}.

n=size(x_data,1);
p=0;
for k=1:n;                % For each data point {x_data(k),y_data(k)}
   L=1;
   for i=1:k-1
      L=L*(x-x_data(i))/(x_data(k)-x_data(i));    % Compute L_k
   end
   for i=k+1:n
      L=L*(x-x_data(i))/(x_data(k)-x_data(i));
   end

   p = p + y_data(k) * L;   % Add L_k's contribution to p(x)
end
% end lagrange.m
```

4.2 Cubic spline interpolation

Instead of forcing a high-order polynomial through the entire dataset (which often has the spurious effect shown in Figure 4.1), we may instead construct a continuous, smooth, piecewise cubic function through the data. We will construct this function to be smooth in the sense of having continuous first and second derivatives at each data point. These conditions, together with the appropriate conditions at each end, uniquely determine a piecewise cubic function through the data which is usually reasonably smooth; we will call this function the cubic spline interpolant.

Defining the interpolant in this manner is akin to deforming a thin piece of wood or metal to pass over all of the data points plotted on a large block of wood and marked with thin nails. In fact, the first definition of the word "spline" in Webster's dictionary is "a thin wood or metal strip used in building construction"—this is where the method takes its name. The elasticity equation governing the deformation f of the spline is

$$f'''' = G, \tag{4.1}$$

where G is a force localized near each nail (*i.e.*, with a delta function) which is sufficient to pass the spline through the data. As G is nonzero only in the immediate vicinity of each nail, such a spline takes an approximately piecewise cubic shape between the data points. Thus, *between the data points*, we have:

$$f'''' = 0, \qquad f''' = C_1, \qquad f'' = C_1\,x + C_2,$$
$$f' = \frac{C_1}{2}x^2 + C_2\,x + C_3, \quad \text{and} \quad f = \frac{C_1}{6}x^3 + \frac{C_2}{2}x^2 + C_3\,x + C_4.$$

4.2.1 Constructing the cubic spline interpolant

Let $f_i(x)$ denote the cubic in the interval $x_i \le x \le x_{i+1}$ and let $f(x)$ denote the collection of all the cubics for the entire range $x_0 \le x \le x_n$. As noted above, f_i'' varies linearly with x between each data point. *At* each data point, by (4.1) with G being a linear combination of delta functions, we have:

(a) continuity of the function f, *i.e.*, $f_{i-1}(x_i) = f_i(x_i) = f(x_i) = y_i$,
(b) continuity of the first derivative f', *i.e.*, $f'_{i-1}(x_i) = f'_i(x_i) = f'(x_i), \textit{and}$
(c) continuity of the second derivative f'', *i.e.*, $f''_{i-1}(x_i) = f''_i(x_i) = f''(x_i)$.

We now describe a procedure to determine an f which satisfies conditions (a) and (c) by *construction*, in a manner analogous to the construction of the Lagrange interpolant in §4.1.2, and which satisfies condition (b) by setting up and solving the appropriate system of equations for the value of f'' at each data point x_i.

To begin the constructive procedure for determining f, note that on each interval $x_i \le x \le x_{i+1}$ for $i = 0, 1, \ldots, n-1$, we may write a linear equation for $f_i''(x)$ as a function of its value at the endpoints, $f''(x_i)$ and $f''(x_{i+1})$, which are (as yet) undetermined. The following form (which is linear in x) fits the bill by construction:

$$f_i''(x) = f''(x_i)\frac{x - x_{i+1}}{x_i - x_{i+1}} + f''(x_{i+1})\frac{x - x_i}{x_{i+1} - x_i}.$$

Note that this first degree polynomial is in fact just a Lagrange interpolation of the two data points $\{x_i, f''(x_i)\}$ and $\{x_{i+1}, f''(x_{i+1})\}$. By construction, condition (c) is satisfied. Integrating this equation twice and defining $\Delta_i = x_{i+1} - x_i$, it follows that

$$f_i'(x) = -\frac{f''(x_i)}{2}\frac{(x_{i+1} - x)^2}{\Delta_i} + \frac{f''(x_{i+1})}{2}\frac{(x - x_i)^2}{\Delta_i} + C_1.$$
$$f_i(x) = \frac{f''(x_i)}{6}\frac{(x_{i+1} - x)^3}{\Delta_i} + \frac{f''(x_{i+1})}{6}\frac{(x - x_i)^3}{\Delta_i} + C_1 x + C_2.$$

The undetermined constants of integration are obtained by matching the end conditions

$$f_i(x_i) = y_i \qquad \text{and} \qquad f_i(x_{i+1}) = y_{i+1}.$$

A convenient way of constructing the linear and constant terms in the expression for $f_i(x)$ in such a way that the desired end conditions are met is by writing $f_i(x)$ in the form

$$\begin{aligned} f_i(x) = {} & \frac{f''(x_i)}{6}\left(\frac{(x_{i+1}-x)^3}{\Delta_i} - \Delta_i(x_{i+1}-x)\right) + \frac{f''(x_{i+1})}{6}\left(\frac{(x-x_i)^3}{\Delta_i} - \Delta_i(x-x_i)\right) \\ & + y_i\frac{(x_{i+1}-x)}{\Delta_i} + y_{i+1}\frac{(x-x_i)}{\Delta_i}, \qquad \text{where} \qquad x_i \le x \le x_{i+1}. \end{aligned} \tag{4.2}$$

By construction, condition (a) is satisfied. Finally, an expression for $f_i'(x)$ may now be found by differentiating this expression for $f_i(x)$, which gives

$$f_i'(x) = \frac{f''(x_i)}{6}\left(-3\frac{(x_{i+1}-x)^2}{\Delta_i} + \Delta_i\right) + \frac{f''(x_{i+1})}{6}\left(3\frac{(x-x_i)^2}{\Delta_i} - \Delta_i\right) + \frac{y_{i+1}}{\Delta_i} - \frac{y_i}{\Delta_i}.$$

The second derivative of f at each node, $f''(x_i)$, is still undetermined. A system of equations from which the $f''(x_i)$ may be found is obtained by imposing condition (b), which is achieved by setting

$$f_i'(x_i) = f_{i-1}'(x_i) \qquad \text{for} \qquad i = 1, 2, \ldots, n-1.$$

Substituting appropriately from the above expression for $f_i'(x)$, noting that $\Delta_i = x_{i+1} - x_i$, leads to

$$\frac{\Delta_{i-1}}{6} f''(x_{i-1}) + \frac{\Delta_{i-1}+\Delta_i}{3} f''(x_i) + \frac{\Delta_i}{6} f''(x_{i+1}) = \frac{y_{i+1}-y_i}{\Delta_i} - \frac{y_i - y_{i-1}}{\Delta_{i-1}} \tag{4.3}$$

for $i = 1, 2, \ldots, n-1$. This is a diagonally-dominant tridiagonal system of $n-1$ equations for the $n+1$ unknowns $f''(x_0)$, $f''(x_1)$, $\ldots$, $f''(x_n)$. We find the two remaining equations by prescribing conditions on the interpolating function at each end. We will consider three types of end conditions:

- Parabolic run-out:

$$\begin{aligned} f''(x_0) &= f''(x_1) \\ f''(x_n) &= f''(x_{n-1}) \end{aligned} \tag{4.4}$$

- Free run-out (also known as natural splines):

$$\begin{aligned} f''(x_0) &= 0 \\ f''(x_n) &= 0 \end{aligned} \tag{4.5}$$

- Periodic end-conditions:

$$\begin{aligned} f''(x_0) &= f''(x_{n-1}) \\ f''(x_1) &= f''(x_n) \end{aligned} \tag{4.6}$$

Equation (4.3) may be taken together with equation (4.4), (4.5), or (4.6) (making the appropriate choice for the end conditions depending upon the problem at hand) to give us $n+1$ equations for the $n+1$ unknowns $f''(x_i)$. This set of equations is then solved for the $f''(x_i)$, which thereby ensures that condition (b) is satisfied. Once this system is solved for the $f''(x_i)$, the cubic spline interpolant follows immediately from equation (4.2).

x_i	-12	-10	-8	-6	-4	-2	0	2	4	6	8	10
y_i	-0.05	-0.05	0.12	-0.05	-0.19	0.46	1	0.46	-0.19	-0.05	0.12	-0.05

Table 4.1: Data points from which we want to reproduce (via interpolation) a continuous function. These data points all lie near the familiar curve generated by the function $\sin(x)/x$.

Note that, when equation (4.3) is taken together with parabolic or free run-out at the ends, a tridiagonal system results which can be solved efficiently with the Thomas algorithm. When equation (4.3) is taken together periodic end conditions, a tridiagonal circulant system (which is not diagonally dominant) results. A code which solves these systems to determine the cubic spline interpolation with any of the above three end-conditions is given in the following subsection.

The results of applying the cubic spline interpolation formula to the set of data given in Table 4.1 is shown in Figure 4.2. All three end conditions on the cubic spline do reasonably well, but the Lagrange interpolation is again spurious. Note that applying periodic end conditions on the spline in this case (which is not periodic) leads to a non-physical "wiggle" in the interpolated curve near the left end; in general, the periodic end conditions should be reserved for those cases for which the function being interpolated is indeed periodic. The parabolic run-out simply places a parabola between x_0 and x_1, whereas the free run-out tapers the curvature down to zero near x_0. Both end conditions provide reasonably smooth interpolations.

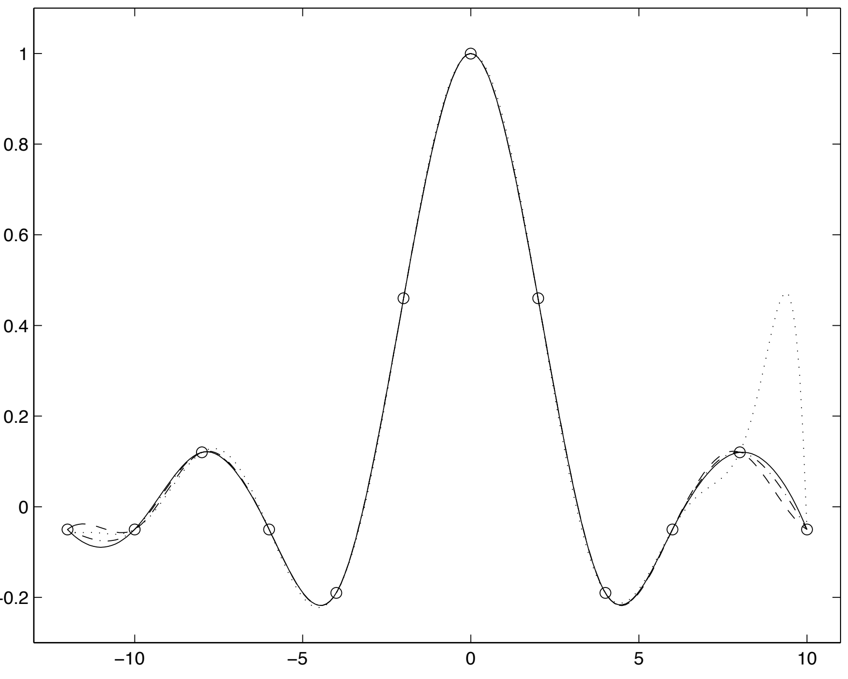


Figure 4.2: Various interpolations of the data of Table 1: ∘ data; ········ Lagrange interpolation; ——— cubic spline (parabolic run-out); —·— cubic spline (free run-out); – – – – cubic spline (periodic).

4.2.2 Matlab implementation

```
% spline_setup.m
% Determines the quantity g=f'' for constructing a cubic spline
% interpolation of a function.  Note that this algorithm calls
% thomas.m, which was developed earlier in the course, and assumes
% the x_data is already sorted in ascending order.

% Compute the size of the problem.
n=size(x_data,1);

% Set up the delta_i = x_(i+1)-x_i for i=1:n-1.
delta(1:n-1)=x_data(2:n)-x_data(1:n-1);

% Set up and solve a tridiagonal system for the g at each data point.
for i=2:n-1
   a(i)=delta(i-1)/6;
   b(i)=(delta(i-1)+delta(i))/3;
   c(i)=delta(i)/6;
   g(i)=(y_data(i+1)-y_data(i))/delta(i) -...
        (y_data(i)-y_data(i-1))/delta(i-1);
end
if end_conditions==1             % Parabolic run-out
   b(1)=1;  c(1)=-1;  g(1)=0;
   b(n)=1;  a(n)=-1;  g(n)=0;
   thomas;     % <--- solve system

elseif end_conditions==2         % Free run-out ("natural" spline)
   b(1)=1;  c(1)=0;   g(1)=0;
   b(n)=1;  a(n)=0;   g(n)=0;
   thomas;     % <--- solve system

elseif end_conditions==3         % Periodic end conditions
   a(1)=-1;  b(1)=0;  c(1)=1;   g(1)=0;
   a(n)=-1;  b(n)=0;  c(n)=1;   g(n)=0;
   % Note:  the following is an inefficient way to solve this circulant
   % (but NOT diagonally dominant!) system via full-blown Gaussian
   % elimination.
   A = diag(a(2:n),-1) + diag(b,0) + diag(c(1:n-1),1);
   A(1,n)=a(1);  A(n,1)=c(1);
   g=A\g';  % <--- solve system

end
% end spline_setup.m

-----------------------------------------------------------------
```

```
function [f] = spline_interp(x,x_data,y_data,g,delta)
n=size(x_data,1);
% Find that value of i such that x_data(i) <= x <= x_data(i+1)
for i=1:n-1
   if x_data(i+1) > x, break, end
end
% compute the cubic spline approximation of the function.
f=g(i)  /6 *((x_data(i+1)-x)^3/delta(i)-delta(i)*(x_data(i+1)-x))+...
  g(i+1)/6 *((x - x_data(i))^3/delta(i)-delta(i)*(x - x_data(i)))+...
  (y_data(i)*(x_data(i+1)-x) + y_data(i+1)*(x-x_data(i))) / delta(i);
% end spline_interp.m

-----------------------------------------------------------------

% test_interp.m
clear; clf;
[x_data, y_data]=input_data;
plot(x_data,y_data,'ko');  axis([-13 11 -0.3 1.1]);
hold on;  pause;

x_lower=min(x_data);  x_upper=max(x_data);
xx=[x_lower : (x_upper-x_lower)/200 : x_upper];

for i=1:201;
  ylagrange(i)=lagrange(xx(i),x_data,y_data);
end
plot(xx,ylagrange,'b:');  pause;

for end_conditions=1:3;     % Set the end conditions you want to try here.
  spline_setup;
  for i=1:201;
    y_spline(i)=spline_interp(xx(i),x_data,y_data,g,delta);
  end
  plot(xx,y_spline);  pause;
end
% end test_interp.m

-----------------------------------------------------------------

function [x_data,y_data] = input_data
x_data=[-12:2:10]';
y_data=sin(x_data+eps)./(x_data+eps);
% end input_data.m
```

4.2.3 Tension splines

For some special cases, cubic splines aren't even smooth enough. In such cases, it is helpful to put "tension" on the spline to straighten out the curvature between the "nails" (datapoints). In the limit of large tension, the interpolant becomes almost piecewise linear.

Tensioned splines obey the differential equation:

$$f'''' - \sigma^2 f'' = G$$

where σ is the tension of the spline. This leads to the following relationships between the data points:

$$[f'' - \sigma^2 f]'' = 0, \qquad [f'' - \sigma^2 f]' = C_1, \qquad [f'' - \sigma^2 f] = C_1\, x + C_2.$$

Solving the ODE on the right leads to an equation of the form

$$f = -\sigma^{-2}(C_1 x + C_2) + C_3 e^{-\sigma x} + C_4 e^{\sigma x}.$$

Proceeding with a constructive process to satisfy condition (a) as discussed in §4.2.1, we assemble the linear and constant terms of $f'' - \sigma^2 f$ such that

$$\big[f_i''(x) - \sigma^2 f_i(x)\big] = \big[f_i''(x_i) - \sigma^2 y_i\big]\,\frac{x - x_{i+1}}{x_i - x_{i+1}} + \big[f_i''(x_{i+1}) - \sigma^2 y_{i+1}\big]\,\frac{x - x_i}{x_{i+1} - x_i}.$$

Similarly, we assemble the exponential terms in the solution of this ODE for f in a constructive manner such that condition (c) is satisfied. Rewriting the exponentials as sinh functions, the desired solution may be written

$$\begin{aligned} f_i(x) = -\sigma^{-2}\Big\{ &\big[f''(x_i) - \sigma^2 y_i\big]\,\frac{x_{i+1} - x}{\Delta_i} + \big[f''(x_{i+1}) - \sigma^2 y_{i+1}\big]\,\frac{x - x_i}{\Delta_i} - f''(x_i)\frac{\sinh\sigma(x_{i+1} - x)}{\sinh\sigma\Delta_i} \\ &- f''(x_{i+1})\frac{\sinh\sigma(x - x_i)}{\sinh\sigma\Delta_i}\Big\} \qquad \text{where} \qquad x_i \le x \le x_{i+1}. \end{aligned} \tag{4.7}$$

Differentiating once and appling condition (b) leads to the tridiagonal system:

$$\begin{aligned} &\left(\frac{1}{\Delta_{i-1}} - \frac{\sigma}{\sinh\sigma\Delta_{i-1}}\right)\frac{f''(x_{i-1})}{\sigma^2} - \left(\frac{1}{\Delta_{i-1}} - \frac{\sigma\cosh\sigma\Delta_{i-1}}{\sinh\sigma\Delta_{i-1}} + \frac{1}{\Delta_i} - \frac{\sigma\cosh\sigma\Delta_i}{\sinh\sigma\Delta_i}\right)\frac{f''(x_i)}{\sigma^2} \\ &\qquad + \left(\frac{1}{\Delta_i} - \frac{\sigma}{\sinh\sigma\Delta_i}\right)\frac{f''(x_{i+1})}{\sigma^2} = \frac{y_{i+1} - y_i}{\Delta_i} - \frac{y_i - y_{i-1}}{\Delta_{i-1}}. \end{aligned} \tag{4.8}$$

The tridiagonal system (4.8) can be set up and solved exactly as was done with (4.3), even though the coefficients have a slightly more complicated form. The tensioned-spline interpolant is then given by (4.7).

4.2.4 B-splines

It is interesting to note that we may write the cubic spline interpolant in a similar form to how we constructed the Lagrange interpolant, that is,

$$f(x) = \sum_{\kappa=0}^{n} y_\kappa b_\kappa(x),$$

where the $b_\kappa(x)$ correspond to the various cubic spline interpolations of the Kronecker deltas such that $b_\kappa(x_i) = \delta_{i\kappa}$, as discussed in §4.1.2 for the functions $L_\kappa(x)$. The $b_\kappa(x)$ in this representation are referred to as the basis functions, and are found to have **localized support** (in other words, $b_\kappa(x) \to 0$ for large $|x - x_\kappa|$).

By relaxing some of the continuity constraints, we may confine each of the basis functions to have **compact support** (in other words, we can set $b_\kappa(x) = 0$ exactly for $|x - x_\kappa| > R$ for some R). With such functions, it is easier both to compute the interpolations themselves and to project the interpolated function onto a different mesh of gridpoints. The industry of computer animation makes heavy use of efficient algorithms for this type of interpolation extended to three-dimensional problems.

Chapter 5

Minimization

5.0 Motivation

It is often the case that one needs to minimize a scalar function of one or several variables. Three cases of particular interest which motivate the present chapter are described here.

5.0.1 Solution of large linear systems of equations

Consider the $n \times n$ problem $A\mathbf{x} = \mathbf{b}$, where n is so large that an $\sim O(n^3)$ algorithm such as Gaussian elimination is out of the question, and the problem can not be put into such a form that A has a banded structure. In such cases, it is sometimes useful to construct a function $\mathcal{J}(\mathbf{x})$ such that, once (approximately) minimized via an iterative technique, the desired condition $A\mathbf{x} = \mathbf{b}$ is (approximately) satisfied. If A is symmetric positive definite (*i.e.*, if $a_{kj} = a_{jk}$ and all eigenvalues of A are positive) then we can accomplish this by defining

$$\mathcal{J}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} = \frac{1}{2}x_i\, a_{ij}\, x_j - b_i\, x_i.$$

Requiring that A be symmetric positive definite insures that $\mathcal{J} \to \infty$ for $|\mathbf{x}| \to \infty$ in all directions and thus that a minimum point indeed exists. (We will extend this approach to more general problems at the end of this chapter.) Differentiating $\mathcal{J}$ with respect to an arbitrary component of $\mathbf{x}$, we find that

$$\frac{\partial \mathcal{J}}{\partial x_k} = \frac{1}{2}(\delta_{ik}\, a_{ij}\, x_j + x_i\, a_{ij}\, \delta_{jk}) - b_i\, \delta_{ik} = a_{kj}x_j - b_k.$$

The unique minimum of $\mathcal{J}(\mathbf{x})$ is characterized by

$$\frac{\partial \mathcal{J}}{\partial x_k} = a_{kj}x_j - b_k = 0 \qquad \text{or} \qquad \boldsymbol{\nabla}\mathcal{J} = A\mathbf{x} - \mathbf{b} = \mathbf{0}.$$

Thus, solution of large linear systems of the form $A\mathbf{x} = \mathbf{b}$ (where A is symmetric positive definite) may be found by minimization of $\mathcal{J}(\mathbf{x})$.

5.0.2 Solution of nonlinear systems of equations

Recall from §3.1 that the Newton-Raphson method was an effective technique to find the root (when one exists) of a nonlinear system of equations $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ when a sufficiently-accurate initial guess is available. When such a guess is not available, an alternative technique is to examine the square of the norm of the vector $\mathbf{f}$:

$$\mathcal{J}(\mathbf{x}) = [\mathbf{f}(\mathbf{x})]^T \mathbf{f}(\mathbf{x})$$

Note that this quantity is never negative, so any point $\mathbf{x}$ for which $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ minimizes $\mathcal{J}(\mathbf{x})$. Thus, seeking a minimum of this $\mathcal{J}(\mathbf{x})$ with respect to $\mathbf{x}$ <u>might</u> result in an $\mathbf{x}$ such that $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. However, there are quite likely many minimum points of $\mathcal{J}(\mathbf{x})$ (at which $\boldsymbol{\nabla}\mathcal{J} = \mathbf{0}$), only some of which (if any!) will correspond to $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. Root finding in systems of nonlinear equations is very difficult—though this method has significant drawbacks, variants of this method are really about the best one can do when one doesn't have a good initial guess for the solution.

5.0.3 Optimization and control of dynamic systems

In control and optimization problems, one can often represent the desired objective mathematically as a "cost function" to be minimized. When the cost function is formulated properly, its minimization with respect to the control parameters results in an effectively-controlled dynamic system. Such cost functions may often be put in the form

$$\mathcal{J}(\mathbf{u}) = \mathbf{x}^T Q \mathbf{x} + \mathbf{u}^T R \mathbf{u}$$

where $\mathbf{x} = \mathbf{x}(\mathbf{u})$ is the state of the system and $\mathbf{u}$ is the control. The second term in the above expression measures the amount of control used and the first term is an observation of the property of interest of the dynamic system which, in turn, is a function of the applied control. When the control is both efficient (small in magnitude) and has the desired effect on the dynamic system, both of these terms are small. Thus, minimization of this cost function with respect to $\mathbf{u}$ results in the determination of an effective set of control parameters.

5.1 The Newton-Raphson method for nonquadratic minimization

If a good initial guess is available (near the desired minimum point), minimization of $\mathcal{J}(\mathbf{x})$ can be accomplished simply by applying the Newton-Raphson method developed previously to the gradient of $\mathcal{J}(\mathbf{x})$, given by function $\mathbf{f}(\mathbf{x}) = \boldsymbol{\nabla}\mathcal{J}$, in order to find the solution to the equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. This works just as well for scalar or vector $\mathbf{x}$, and converges quite rapidly. Recall from equation (3.4) that the Newton-Raphson method requires both evaluation of the function $\mathbf{f}(\mathbf{x})$, in this case taken to be the gradient of $\mathcal{J}(\mathbf{x})$, and the Jacobian of $\mathbf{f}(\mathbf{x})$, given in this case by

$$a_{ij}^{(k)} = \frac{\partial f_i}{\partial x_j}\Big|_{\mathbf{x}=\mathbf{x}^{(k)}} = \frac{\partial^2 \mathcal{J}}{\partial x_i \partial x_j}\Big|_{\mathbf{x}=\mathbf{x}^{(k)}}.$$

This matrix of second derivatives is referred to as the Hessian of the function $\mathcal{J}$. Unfortunately, for very large N, computation and storage of the Hessian matrix, which has N^2 elements, can be prohibitively expensive.

5.2 Bracketing approaches for minimization of scalar functions

5.2.1 Bracketing a minimum

We now seek a reliable but pedestrian approach to a minimize scalar function $\mathcal{J}(\mathbf{x})$ when a good initial guess for the minimum is not available. To do this, we begin with a "bracketing" approach analogous to that which we used for finding the root of a nonlinear scalar equation. Recall from §3.2 that bracketing a root means finding a pair $\{x^{(\text{lower})}, x^{(\text{upper})}\}$ for which $f(x^{(\text{lower})})$ and $f(x^{(\text{upper})})$ have opposite signs (so a root must exist between $x^{(\text{lower})}$ and $x^{(\text{upper})}$ if the function is continuous and bounded).

Analogously, bracketing a minimum means finding a triplet $\{x_1, x_2, x_3\}$ with x_2 between x_1 and x_3 and for which $\mathcal{J}(x_2) < \mathcal{J}(x_1)$ and $\mathcal{J}(x_2) < \mathcal{J}(x_3)$ (so a minimum must exist between x_1 and x_3 if the function is continuous and bounded). Such an initial bracketing triplet may often be found by a minor amount of trial and error. At times, however, it is convenient to have an automatic procedure to find such a bracketing triplet. For example, for functions which are large and positive for sufficiently large $|x|$, a simple approach is to start with an initial guess for the bracket and then geometrically scale out each end until a bracketing triplet is found.

Matlab implementation

```
% find_triplet.m
% Initialize and expand a triplet until the minimum is bracketed.
% Should work if J -> inf as |x| -> inf.
[x1,x2,x3] = init_triplet;
J1=compute_J(x1);  J2=compute_J(x2);  J3=compute_J(x3);
while (J2>J1)
   % Compute a new point x4 to the left of the triplet
   x4=x1-2.0*(x2-x1);  J4=compute_J(x4);
   % Center new triplet on x1
   x3=x2;      J3=J2;
   x2=x1;      J2=J1;
   x1=x4;      J1=J4;
end
while (J2>J3)
   % Compute new point x4 to the right of the triplet
   x4=x3+2.0*(x3-x2);   J4=compute_J(x4);
   % Center new triplet on x3
   x1=x2;      J1=J2;
   x2=x3;      J2=J3;
   x3=x4;      J3=J4;
end
% end find_triplet.m
```

Case-specific auxiliary functions are given below.

```
function [J] = compute_J(x)
J=3*cos(x)-sin(2*x) + 0.1*x.^2;  % compute function you are trying to minimize.
plot(x,J,'ko');                  % plot a circle at the data point.
% end compute_J.m
```

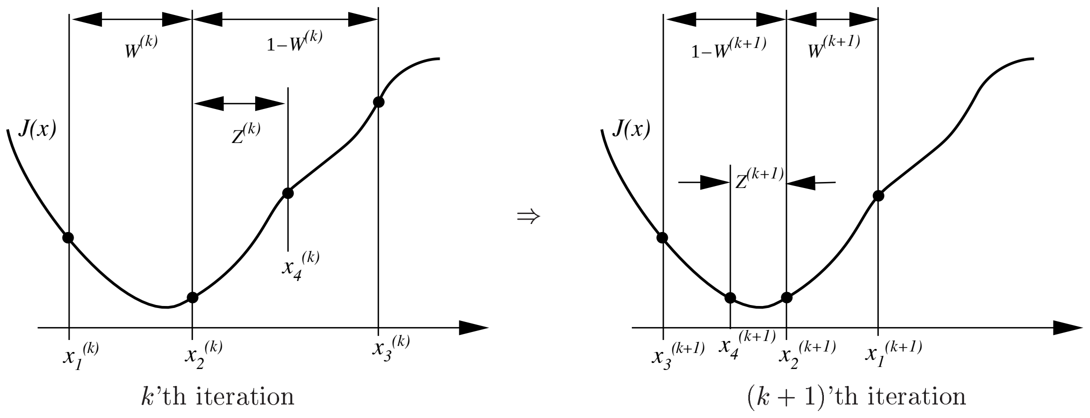


Figure 5.1: Naming of points used in golden section search procedure: $\{x_1^{(k)}, x_2^{(k)}, x_3^{(k)}\}$ is referred to as the bracketing triplet at iteration k, where $x_2^{(k)}$ is between $x_1^{(k)}$ and $x_3^{(k)}$ and $x_2^{(k)}$ is assumed to be closer to $x_1^{(k)}$ than it is to $x_3^{(k)}$. A new guess is made at point $x_4^{(k)}$ and the bracket is refined by retaining those three of the four points which maintain the tightest bracket. The reduction of the interval continues at the following iterations in a self-similar fashion.

```
function [x1,x2,x3] = init_triplet
x1=-5; x2=0; x3=5;              % Initialize guess for the bracketing triplet
% end init_triplet.m
```

5.2.2 Refining the bracket - the golden section search

Once the minimum of the non-quadratic function is bracketed, all that remains to be done is to refine these brackets. A simple algorithm to accomplish this, analogous to the bisection technique developed for scalar root finding, is called the golden section search. As illustrated in Figure 5.1, let W be defined as the ratio of the smaller interval to the width of the bracketing triplet $\{x_1, x_2, x_3\}$ such that

$$W = \frac{x_2 - x_1}{x_3 - x_1} \qquad \Rightarrow \qquad 1 - W = \frac{x_3 - x_2}{x_3 - x_1}.$$

We now pick a new trial point x_4 and define $Z = \dfrac{x_4 - x_2}{x_3 - x_1} \quad \Rightarrow \quad x_4 = x_2 + Z(x_3 - x_1)$.

There are two possibilities:

if $\mathcal{J}(x_4) > \mathcal{J}(x_2)$, then $\{x_4, x_2, x_1\}$ is a new bracketing triplet (as in Figure 5.1), and
if $\mathcal{J}(x_4) < \mathcal{J}(x_2)$, then $\{x_2, x_4, x_3\}$ is a new bracketing triplet.

Minimizing the width of this new (refined) bracketing triplet in the worst case, we should take the width of both of these triplets as identical, so that

$$W + Z = 1 - W \qquad \Rightarrow \qquad Z = 1 - 2W \tag{5.1}$$

If the same algorithm is used for the refinement at each iteration k, then a self-similar situation develops in which the ratio W is constant from one iteration to the next, *i.e.*, $W^{(k)} = W^{(k+1)}$. Note that, in terms of the quantities at iteration k, we have either

$W^{k+1} = Z^{(k)}/(W^{(k)} + Z^{(k)})$ if $\{x_4^{(k)}, x_2^{(k)}, x_1^{(k)}\}$ is the new bracketing triplet, or
$W^{k+1} = Z^{(k)}/(1 - W^{(k)})$ if $\{x_2^{(k)}, x_4^{(k)}, x_3^{(k)}\}$ is the new bracketing triplet.

Dropping the superscripts on W and Z, which we assume to be independent of k, and inserting (5.1), both of these conditions reduce to the relation

$$W^2 - 3W + 1 = 0,$$

which (because $0 < W < 1$) implies that

$$W = \frac{3-\sqrt{5}}{2} \approx 0.381966 \quad \Rightarrow \quad 1 - W = \frac{\sqrt{5}-1}{2} \approx 0.618034 \quad \text{and} \quad Z = \sqrt{5} - 2 \approx 0.236068.$$

These proportions are referred to as the golden section.

To summarize, the golden section algorithm takes an initial bracketing triplet $\{x_1^{(0)}, x_2^{(0)}, x_3^{(0)}\}$, computes a new data point at $x_4^{(0)} = x_2^{(0)} + Z(x_3^{(0)} - x_1^{(0)})$ where $Z = 0.236068$, and then:

if $\mathcal{J}(x_4^{(0)}) > \mathcal{J}(x_2^{(0)})$, the new triplet is $\{x_1^{(1)}, x_2^{(1)}, x_3^{(1)}\} = \{x_4^{(0)}, x_2^{(0)}, x_1^{(0)}\}$, or
if $\mathcal{J}(x_4^{(0)}) < \mathcal{J}(x_2^{(0)})$, the new triplet is $\{x_1^{(1)}, x_2^{(1)}, x_3^{(1)}\} = \{x_2^{(0)}, x_4^{(0)}, x_3^{(0)}\}$.

The process continues on the new (refined) bracketing triplet in an iterative fashion until the desired tolerance is reached such that $|x_3 - x_1| < \epsilon$. Even if the initial bracketing triplet is not in the ratio of the golden section, repeated application of this algorithm quickly brings the triplet into this ratio as it is refined. Note that convergence is attained linearly: each bound on the minimum is 0.618034 times the previous bound. This is slightly slower than the convergence of the bisection algorithm for nonlinear root-finding.

Matlab implementation

```
% golden.m
% Input assumes {x1,x2,x3} are a bracketing triple with function
% values {J1,J2,J3}.  On output, x2 is the best guess of the minimum.

Z=sqrt(5)-2;                       % initialize the golden section ratio
if (abs(x2-x1) > abs(x2-x3))       % insure proper ordering
   swap=x1;  x1=x3;  x3=swap;
   swap=J1;  J1=J3;  J3=swap;
end
pause;
while (abs(x3-x1) > x_tol)
   x4 = x2 + Z*(x3-x1);      % compute and plot new point
   J4 = compute_J(x4);
   evals = evals+1;

   % Note that a couple of lines are commented out below because some
   % of the data is already in the right place!
   if (J4>J2)
      x3=x1;      J3=J1;    % Center new triplet on x2
    % x2=x2;      J2=J2;
      x1=x4;      J1=J4;
   else
```

```
      x1=x2;      J1=J2;    % Center new triplet on x4
      x2=x4;      J2=J4;
    % x3=x3;      J3=J3;
   end
   pause;
end
% end golden.m
```

The implementation of the golden section search algorithm may be tested as follows:

```
% test_golden.m
% Tests the golden search routine
clear;  find_triplet;    % Initialize a bracket of the minimum.

% Prepare to make some plots of the function over the width of the triplet
xx=[x1 : (x3-x1)/100 : x3];
for i=1:101, yy(i)=compute_J(xx(i)); end
figure(1); clf; plot(xx,yy,'k-');  hold on; grid;
title('Convergence of the golden section search')
plot(x1,J1,'ko');  plot(x2,J2,'ko');  plot(x3,J3,'ko');

x_tol = 0.0001;  % Set desired tolerance for x.
evals=0;         % Reset a counter for tracking function evaluations.
golden; hold off
x2, J2, evals
% end test_golden.m
```

5.2.3 Refining the bracket - inverse parabolic interpolation

Recall from §3.2.3 that, when a function $f(x)$ is "locally linear" (meaning that its shape is well-approximated by a linear function), the false position method is an efficient technique to find the root of the function based on function evaluations alone. The false position method is based on the construction of successive linear interpolations of recent function evaluations, taking each new estimate of the root of $f(x)$ as that value of x for which the value of the linear interpolant is zero.

Analogously, when a function $\mathcal{J}(x)$ is "locally quadratic" (meaning that its shape is well-approximated by a quadratic function), the minimum point of the function may be found via an efficient technique based on function evaluations alone. At the heart of this technique is the construction of successive quadratic interpolations based on recent function evaluations, taking each new estimate of the minimum of $\mathcal{J}(x)$ as that value of x for which the value of the quadratic interpolant is minimum. For example, given data points $\{x_0, y_0\}$, $\{x_1, y_1\}$, and $\{x_2, y_2\}$, the quadratic interpolant is given by the Lagrange interpolation formula

$$P(x) = y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + y_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)},$$

as described in §4.1. Setting $dP(x)/dx = 0$ to find the critical point of this quadratic yields

$$0 = y_0 \frac{2\,x-x_1-x_2}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{2\,x-x_0-x_2}{(x_1-x_0)(x_1-x_2)} + y_2 \frac{2\,x-x_0-x_1}{(x_2-x_0)(x_2-x_1)}.$$

Multiplying by $(x_0 - x_1)(x_1 - x_2)(x_2 - x_0)$ and then solving for x gives the desired value of x which is a critical point of the interpolating quadratic:

$$x = \frac{1}{2}\,\frac{y_0(x_1+x_2)(x_1-x_2) + y_1(x_0+x_2)(x_2-x_0) + y_2(x_0+x_1)(x_0-x_1)}{y_0(x_1-x_2) + y_1(x_2-x_0) + y_2(x_0-x_1)}.$$

Note that the critical point found may either be a minimum point or a maximum point depending on the relative orientation of the data, but will always maintain the bracketing triplet.

Matlab implementation

```
% inv_quad.m
% Finds the minimum or maximum of a function by repeatedly moving to the
% critical point of a quadratic interpolant of three recently-computed data points
% in an attempt to home in on the critical point of a nonquadratic function.
res=1; i=1;
J1=compute_J(x1);  J2=compute_J(x2);  J3=compute_J(x3);
while (abs(x3-x1) > x_tol)
   x4 = 0.5*(J1*(x2+x3)*(x2-x3)+ J2*(x3+x1)*(x3-x1)+ J3*(x1+x2)*(x1-x2))/...
            (J1*(x2-x3)+ J2*(x3-x1)+ J3*(x1-x2));   % compute the critical point

   % The following plotting stuff is done for demonstration purposes only.
   x_lower=min([x1 x2 x3]);  x_upper=max([x1 x2 x3]);
   xx=[x_lower : (x_upper-x_lower)/200 : x_upper];
   for i=1:201;
      J_lagrange(i)=lagrange(xx(i),[x1; x2; x3],[J1; J2; J3]);
   end
   plot(xx,J_lagrange,'b-');    % plot a curve through the data
   Jinterp=lagrange(x4,[x1; x2; x3],[J1; J2; J3]);  % plot a * at the critical point
   plot(x4,Jinterp,'r*');                             % of the Lagrange interpolant.
   pause;

   J4 = compute_J(x4);      % Compute function J at new point and the proceed as with
   evals = evals+1;         % the golden section search.

   % Note that a couple of lines are commented out below because some
   % of the data is already in the right place!
   if (J4>J2)
      x3=x1;      J3=J1;    % Center new triplet on x2
    % x2=x2;      J2=J2;
      x1=x4;      J1=J4;
   else
      x1=x2;      J1=J2;    % Center new triplet on x4
      x2=x4;      J2=J4;
    % x3=x3;      J3=J3;
   end
end
% end inv_quad.m
```

The `test_golden.m` code is easily modified (by replacing the call to `golden.m` with a call to `inv_quad.m`) to test the efficiency of the inverse parabolic algorithm.

5.2.4 Refining the bracket - Brent's method

As with the false position technique for accelerated bracket refinement for the problem of scalar root finding, the inverse parabolic technique can also stall for a variety of scalar functions $\mathcal{J}(x)$ one might attempt to minimize.

A hybrid technique, referred to as Brent's method, combines the reliable convergence benefits of the golden section search with the ability of the inverse parabolic interpolation technique to "home in" quickly on the solution when the minimum point is approached. Switching in a reliable fashion from one technique to the other without the possibility for stalling requires a certain degree of heuristics and results in a rather long code which is not very elegant looking. Don't let this dissuade you, however, the algorithm (when called correctly) is reliable, fast, and requires a minimum number of function evaluations. The algorithm was developed by Brent in 1973 and implemented in Fortran and C by the authors of Numerical Recipes (1998), where it is discussed in further detail. (A Matlab implementation of the algorithm is included at the class web site, with the name `brent.m`. It is functionally equivalent to `golden.m` and `inv_quad.m`, discussed above, and can be called in the same manner.) Of the methods discussed in these notes, Brent's method is the best "all purpose" scalar minimimization tool for a wide variety of applications.

5.3 Gradient-based approaches for minimization of multivariable functions

We now seek a reliable technique to minimize a multivariable function $\mathcal{J}(\mathbf{x})$ which a) does not require a good initial guess, b) is efficient for high-dimensional systems, and c) does not require computation and storage of the Hessian matrix. As opposed to the problem of root finding in the multivariable case, some very good techniques are available to solve this problem.

A straightforward approach to minimizing a scalar function $\mathcal{J}$ of an n-dimensional vector $\mathbf{x}$ is to update the vector $\mathbf{x}$ iteratively, proceeding at each step in a downhill direction $\mathbf{p}$ a distance which minimizes $\mathcal{J}(\mathbf{x})$ in this direction. In the simplest such algorithm (referred to as the steepest descent or simple gradient algorithm), the direction $\mathbf{p}$ is taken to be the direction $\mathbf{r}$ of maximum decrease of the function $\mathcal{J}(\mathbf{x})$, *i.e.*, the direction opposite to the gradient vector $\boldsymbol{\nabla}\mathcal{J}(\mathbf{x})$. As the iteration $k \to \infty$, this approach should converge to one of the minima of $\mathcal{J}(\mathbf{x})$ (if such a minimum exists). Note that, if $\mathcal{J}(\mathbf{x})$ has multiple minima, this technique will only find one of the minimum points, and the one it finds (a "local" minimum) might not necessarily be the one with the smallest value of $\mathcal{J}(\mathbf{x})$ (the "global" minimum).

Though the above approach is simple, it is often quite slow. As we will show, it is not always the best idea to proceed in the direction of steepest descent of the cost function. A descent direction $\mathbf{p}^{(k)}$ chosen to be a linear combination of the direction of steepest descent $\mathbf{r}^{(k)}$ and the step taken at the previous iteration $\mathbf{p}^{(k-1)}$ is often much more effective. The "momentum" carried by such an approach allows the iteration to turn more directly down narrow valleys without oscillating between one descent direction and another, a phenomenon often encountered when momentum is lacking. A particular choice of the momentum term results in a remarkable orthogonality property amongst the set of various descent directions (namely, ${\mathbf{p}^{(k)}}^T A\, \mathbf{p}^{(j)} = 0$ for $j \neq k$) and the set of descent directions are referred to as a conjugate set. Searching in a series of mutually conjugate directions leads to *exact* convergence of the iterative algorithm in n iterations, assuming a quadratic cost function and no numerical round-off errors[1].

[1]Note that, for large n, the accumulating round-off error due to the finite-precision arithmetic of the calculations is significant, so exact convergence in n iterations usually can not be obtained.

In the following, we will develop the steepest descent (§5.4.1) and conjugate gradient (§5.4.2) approaches for quadratic functions first, then discuss their extension to nonquadratic functions (§5.4.3). Though the quadratic and nonquadratic cases are handled essentially identically in most regards, the line minimizations required by the algorithms may be done directly for quadratic functions, but should be accomplished by a more reliable bracketing procedure (e.g., Brent's method) for nonquadratic functions. Exact convergence in n iterations (again, neglecting numerical round-off errors) is possible only in the quadratic case, though nonquadratic functions may also be minimized quite effectively with the conjugate gradient algorithm when the appropriate modifications are made.

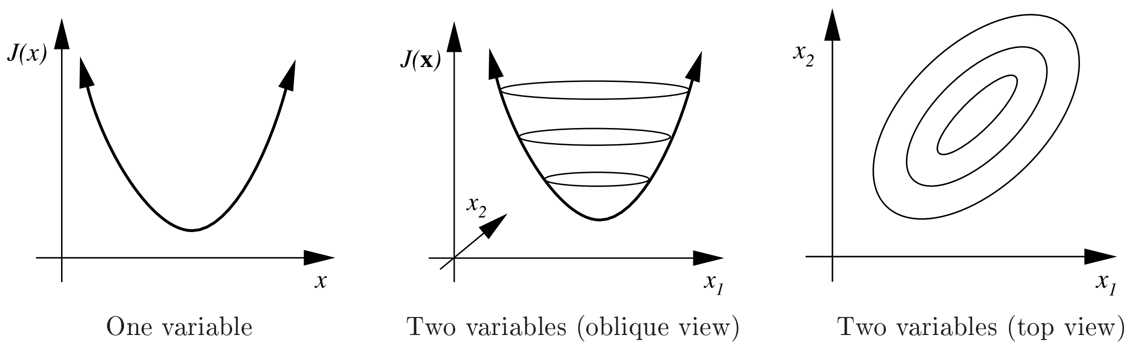


Figure 5.2: Geometry of the minimization problem for quadratic functions. The ellipses in the two figures on the right indicate isosurfaces of constant $\mathcal{J}$.

5.3.1 Steepest descent for quadratic functions

Consider a quadratic function $\mathcal{J}(\mathbf{x})$ of the form

$$\mathcal{J}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$$

where A is positive definite. The geometry of this problem is illustrated in Figure 5.2.

We will begin at some initial guess $\mathbf{x}^{(0)}$ and move at each step of the iteration k in a direction downhill $\mathbf{r}^{(k)}$ such that

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{r}^{(k)},$$

where $\alpha^{(k)}$ is a parameter for the descent which will be determined. In this manner, we proceed iteratively towards the minimum of $\mathcal{J}(\mathbf{x})$. Noting the derivation of $\boldsymbol{\nabla}\mathcal{J}$ in §5.1.1, define $\mathbf{r}^{(k)}$ as the direction of steepest descent such that

$$\mathbf{r}^{(k)} = -\boldsymbol{\nabla}\mathcal{J}(\mathbf{x}^{(k)}) = -(A\mathbf{x}^{(k)} - \mathbf{b}).$$

Now that we have figured out what *direction* we will update $\mathbf{x}^{(k)}$, we need to figure out the parameter of descent $\alpha^{(k)}$, which governs the *distance* we will update $\mathbf{x}^{(k)}$ in this direction. This may be found by minimizing $\mathcal{J}(\mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{r}^{(k)})$ with respect to $\alpha^{(k)}$. Dropping the superscript $()^{(k)}$ for the time being for notational clarity, note first that

$$\mathcal{J}(\mathbf{x} + \alpha\,\mathbf{r}) = \frac{1}{2}(\mathbf{x} + \alpha\,\mathbf{r})^T A(\mathbf{x} + \alpha\,\mathbf{r}) - \mathbf{b}^T(\mathbf{x} + \alpha\,\mathbf{r})$$

and thus

$$\begin{aligned}\frac{\partial\mathcal{J}(\mathbf{x} + \alpha\,\mathbf{r})}{\partial\alpha} &= \frac{1}{2}\mathbf{r}^T A(\mathbf{x} + \alpha\,\mathbf{r}) + \frac{1}{2}(\mathbf{x} + \alpha\,\mathbf{r})^T A\mathbf{r} - \mathbf{b}^T\mathbf{r} \\ &= \alpha\,\mathbf{r}^T A\mathbf{r} + \mathbf{r}^T A\mathbf{x} - \mathbf{r}^T\mathbf{b} \\ &= \alpha\,\mathbf{r}^T A\mathbf{r} + \mathbf{r}^T(A\mathbf{x} - \mathbf{b}) \\ &= \alpha\,\mathbf{r}^T A\mathbf{r} - \mathbf{r}^T\mathbf{r}.\end{aligned}$$

Setting $\partial\mathcal{J}(\mathbf{x} + \alpha\mathbf{r})/\partial\alpha = 0$ yields

$$\alpha = \frac{\mathbf{r}^T\mathbf{r}}{\mathbf{r}^T A\mathbf{r}}.$$

Thus, from the value of $\mathbf{x}^{(k)}$ at each iteration, we can determine explicitly both the direction of descent $\mathbf{r}^{(k)}$ and the parameter $\alpha^{(k)}$ which minimizes $\mathcal{J}$.

Matlab implementation

```
% sd_quad.m
% Minimize a quadratic function J(x) = (1/2) x^T A x - b^T x
% using the steepest descent method
clear res_save x_save;  epsilon=1e-6;  x=zeros(size(b));

for iter=1:itmax
  r=b-A*x;                                     % determine gradient
  res=r'*r;                                    % compute residual
  res_save(iter)=res;  x_save(:,iter)=x;       % save some stuff
  if (res<epsilon | iter==itmax), break; end   % exit yet?
  alpha=res/(r'*A*r);                          % compute alpha
  x=x+alpha*r;                                 % update x
end

% end sd_quad.m
```

Operation count

The operation count for each iteration of the steepest descent algorithm may be determined by inspection of the above code, and is calculated in the following table.

Operation	flops
To compute $A\mathbf{x}-\mathbf{b}$:	$O(2n^2)$
To compute $\mathbf{r}^T\mathbf{r}$:	$O(2n)$
To compute $\mathbf{r}^T A\mathbf{r}$:	$O(2n^2)$
To compute $\mathbf{x}+\alpha\mathbf{r}$:	$O(2n)$
TOTAL:	$O(4n^2)$

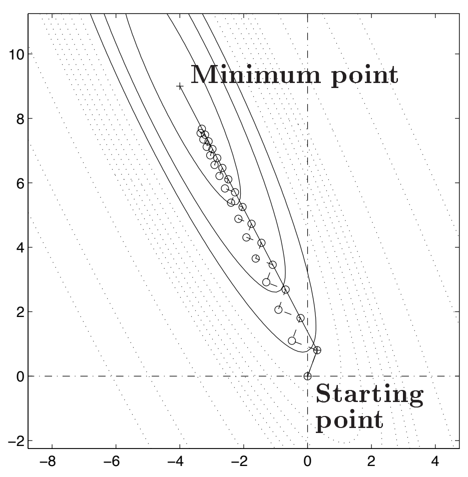
A cheaper technique for computing such an algorithm is discussed at the end of the next section.

5.3.2 Conjugate gradient for quadratic functions

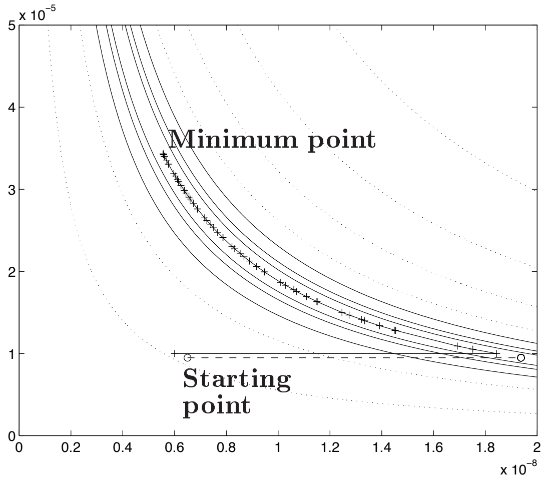
As discussed earlier, and shown in Figure 5.3, proceeding in the direction of steepest descent at each iteration is not necessarily the most efficient strategy. By so doing, the path of the algorithm can be very jagged. Due to the successive line minimizations and the lack of momentum from one iteration to the next, the steepest descent algorithm must tack back and forth 90° at each turn. We now show that, by slight modification of the steepest descent algorithm, we arrive at the vastly improved conjugate gradient algorithm. This improved algorithm retains the correct amount of momentum from one iteration to the next to successfully negotiate functions $\mathcal{J}(\mathbf{x})$ with narrow valleys.

Note that in “easy” cases for which the condition number is approximately unity, the level surfaces of $\mathcal{J}$ are approximately circular, and convergence with either the steepest descent or the conjugate gradient algorithm will be quite rapid. In poorly conditioned problems, the level surfaces become highly elongated ellipses, and the zig-zag behavior is amplified.

Instead of minimizing in a single search direction at each iteration, as we did for the method of steepest descent, now consider searching *simultaneously* in m directions, which we will denote $\mathbf{p}^{(0)}$,



a) Quadratic $\mathcal{J}_3(\phi_1, \phi_2)$.



b) Non-quadratic $\mathcal{J}_4(\phi_1, \phi_2)$.

Figure 5.3: Convergence of: ($\circ$) simple gradient, and (+) the conjugate gradient algorithms when applied to find minima of two test functions of two scalar control variables x_1 and x_2 (horizontal and vertical axes). Contours illustrate the level surfaces of the test functions; contours corresponding to the smallest isovalues are solid, those corresponding to higher isovalues are dotted.

$\mathbf{p}^{(1)}, \ldots\ \mathbf{p}^{(m-1)}$. Take

$$\mathbf{x}^{(m)} = \mathbf{x}^{(0)} + \sum_{j=0}^{m-1} \alpha^{(j)} \mathbf{p}^{(j)},$$

and note that

$$\mathcal{J}(\mathbf{x}^{(m)}) = \frac{1}{2}\Big(\mathbf{x}^{(0)} + \sum_{j=0}^{m-1} \alpha^{(j)} \mathbf{p}^{(j)}\Big)^T A \left(\mathbf{x}^{(0)} + \sum_{j=0}^{m-1} \alpha^{(j)} \mathbf{p}^{(j)}\right) - \mathbf{b}^T \Big(\mathbf{x}^{(0)} + \sum_{j=0}^{m-1} \alpha^{(j)} \mathbf{p}^{(j)}\Big).$$

Taking the derivative of this expression with respect to $\alpha^{(k)}$,

$$\begin{aligned}\frac{\partial \mathcal{J}(\mathbf{x}^{(m)})}{\partial \alpha^{(k)}} &= \frac{1}{2}\Big(\mathbf{p}^{(k)}\Big)^T A \left(\mathbf{x}^{(0)} + \sum_{j=0}^{m-1} \alpha^{(j)} \mathbf{p}^{(j)}\right) + \frac{1}{2}\Big(\mathbf{x}^{(0)} + \sum_{j=0}^{m-1} \alpha^{(j)} \mathbf{p}^{(j)}\Big)^T A \left(\mathbf{p}^{(k)}\right) - \mathbf{b}^T \mathbf{p}^{(k)} \\ &= \alpha^{(k)} \mathbf{p}^{(k)^T} A\, \mathbf{p}^{(k)} + \mathbf{p}^{(k)^T} A\, \mathbf{x}^{(0)} - \mathbf{p}^{(k)^T} \mathbf{b} + \sum_{\substack{j=0 \\ j \neq k}}^{m-1} \alpha^{(j)} \mathbf{p}^{(k)^T} A\, \mathbf{p}^{(j)}.\end{aligned}$$

We seek a technique to select all the $\mathbf{p}^{(j)}$ in such a way that they are orthogonal through A, or conjugate, such that

$$\mathbf{p}^{(k)^T} A\, \mathbf{p}^{(j)} = 0 \quad \text{for} \quad j \neq k.$$

IF we can find such a sequence of $\mathbf{p}^{(j)}$, then we obtain

$$\begin{aligned}\frac{\partial \mathcal{J}(\mathbf{x}^m)}{\partial \alpha^{(k)}} &= \alpha^{(k)} \mathbf{p}^{(k)^T} A\, \mathbf{p}^{(k)} + \mathbf{p}^{(k)^T} \Big(A\, \mathbf{x}^{(0)} - \mathbf{b} \Big) \\ &= \alpha^{(k)} \mathbf{p}^{(k)^T} A\, \mathbf{p}^{(k)} - \mathbf{p}^{(k)^T} \mathbf{r}^{(0)},\end{aligned}$$

and thus setting $\partial \mathcal{J} / \partial \alpha^{(k)} = 0$ results in

$$\alpha^{(k)} = \frac{\mathbf{p}^{(k)^T} \mathbf{r}^{(0)}}{\mathbf{p}^{(k)^T} A\, \mathbf{p}^{(k)}}.$$

The remarkable thing about this result is that it is independent of $\mathbf{p}^{(j)}$ for $j \neq k$! Thus, so long as we can find a way to construct a sequence of $\mathbf{p}^{(k)}$ which are all conjugate, then each of these minimizations may be done separately.

The conjugate gradient technique is simply an efficient technique to construct a sequence of $\mathbf{p}^{(k)}$ which are conjugate. It entails just redefining the descent direction $\mathbf{p}^{(k)}$ at each iteration after the first to be a linear combination of the direction of steepest descent, $\mathbf{r}^{(k)}$, and the descent direction at the previous iteration, $\mathbf{p}^{(k-1)}$, such that

$$\mathbf{p}^{(k)} = \mathbf{r}^{(k)} + \beta\, \mathbf{p}^{(k-1)} \quad \text{and} \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\, \mathbf{p}^{(k)},$$

where β and α are given by

$$\beta = \frac{\mathbf{r}^{(k)^T} \mathbf{r}^{(k)}}{\mathbf{r}^{(k-1)^T} \mathbf{r}^{(k-1)}}, \qquad \alpha = \frac{\mathbf{r}^{(k)^T} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)^T} A\, \mathbf{p}^{(k)}}.$$

Verification that this choice of β results in conjugate directions and that this choice of α is equivalent to the one mentioned previously (minimizing $\mathcal{J}$ in the direction $\mathbf{p}^{(k)}$ from the point $\mathbf{x}^{(k)}$) involves a straightforward (but lengthy) proof by induction. The interested reader may find this proof in Golub and van Loan (1989).

Matlab implementation

```
% cg_quad.m
% Minimize a quadratic function J=(1/2) x^T A x - b^T x
% using the conjugate gradient method
clear res_save x_save;  epsilon=1e-6;  x=zeros(size(b));

for iter=1:itmax
  r=b-A*x;                                   % determine gradient
  res=r'*r;                                  % compute residual
  res_save(iter)=res;  x_save(:,iter)=x;     % save some stuff
  if (res<epsilon | iter==itmax), break; end % exit yet?
  if (iter==1)                               % compute update direction
     p = r;                                  % set up a S.D. step
  else
     beta = res / res_save(iter-1);          % compute momentum term
     p = r + beta * p;                       % set up a C.G. step
  end
  alpha=res/(p'*A*p);                           % compute alpha
  x=x+alpha*p;                                  % update x
end
% end cg_quad.m
```

As seen by comparison of the `cg_quad.m` to `sd_quad.m`, implementation of the conjugate gradient algorithm involves only a slight modification of the steepest descent algorithm. The operation count is virtually the same (`cg_quad.m` requires $2n$ more flops per iteration), and the storage requirements are slightly increased (`cg_quad.m` defines an extra n-dimensional vector $\mathbf{p}$).

Matlab implementation - cheaper version

A cheaper version of `cg_quad` may be written by leveraging the fact that an iterative procedure may developed for the computation of $\mathbf{r}$ by combining the following two equations:

$$\begin{aligned}\mathbf{x}^{(k)} &= \mathbf{x}^{(k-1)} + \alpha^{(k-1)}\mathbf{p}^{(k-1)},\\ \mathbf{r}^{(k)} &= \mathbf{b} - A\mathbf{x}^{(k)}.\end{aligned}$$

Putting these two equations together leads to an update equation for $\mathbf{r}$ at each iteration:

$$\mathbf{r}^{(k)} = \mathbf{b} - A(\mathbf{x}^{(k-1)} + \alpha^{(k-1)}\mathbf{p}^{(k-1)}) = \mathbf{r}^{(k-1)} - \alpha^{(k-1)}A\mathbf{p}^{(k-1)}.$$

The nice thing about this update equation for $\mathbf{r}$, versus the direct calculation of $\mathbf{r}$, is that this update equation depends on the matrix/vector product $A\mathbf{p}^{(k-1)}$, which needs to be computed anyway during the computation of α. Thus, for quadratic functions, an implementation which costs only $O(2n^2)$ per iteration is possible, as given by the following code:

```
% cg_cheap_quad.m
% Minimize a quadratic function J=(1/2) x^T A x - b^T x
% using the conjugate gradient method
clear res_save x_save;  epsilon=1e-6;  x=zeros(size(b));

for iter=1:itmax
  if (iter==1)
     r=b-A*x;        % determine r directly
  else
     r=r-alpha*d;    % determine r by the iterative formula.
  end
  res=r'*r;                                     % compute residual
  res_save(iter)=res;  x_save(:,iter)=x;        % save some stuff
  if (res<epsilon | iter==itmax), break; end    % exit yet?
  if (iter==1)                                  % compute update direction
     p = r;                                     % set up a S.D. step
  else
     beta = res / res_save(iter-1);             % compute momentum term
     p = r + beta * p;                          % set up a C.G. step
  end
  d=A*p;             % perform the (expensive) matrix/vector product
  alpha=res/(p'*d);                             % compute alpha
  x=x+alpha*p;                                  % update x
```

```
end
% end cg_cheap_quad.m
```

A test code named `test_sd_cg.m`, which is available at the class web site, may be used to check to make sure these steepest descent and conjugate gradient algorithms work as advertized. These codes also produce some nice plots to get a graphical interpretation of the operation of the algorithms.

5.3.3 Preconditioned conjugate gradient

Assuming exact arithmetic, the conjugate gradient algorithm converges in exactly N iterations for an N-dimensional quadratic minimization problem. For large N, however, we often can not afford to perform N iterations. We often seek to perform approximate minimization of an N-diminsional problem with a total number of iterations $m \ll N$. Unfortunately, convergence of the conjugate gradient algorithm to the minimum of $\mathcal{J}$, though monotonic, is often highly nonuniform, so large reductions in $\mathcal{J}$ might not occur until iterations well after the iteration m at which we would like to truncate the iteration sequence.

The uniformity of the convergence is governed by the condition number c of the matrix A, which (for symmetric positive-definite A) is just equal to the ratio of its maximum and minimum eigenvalues, $\lambda_{\max}/\lambda_{\min}$. For small c, convergence of the conjugate gradient algorithm is quite rapid even for high-dimensional problems with $N \gg 1$.

We therefore seek to solve a better conditioned but equivalent problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ which, once solved, will allow us to easily extract the solution of the original problem $A\mathbf{x} = \mathbf{b}$ for a poorly-conditioned symmetric positive definite A. To accomplish this, premultiply $A\mathbf{x} = \mathbf{b}$ by P^{-1} for some symmetric preconditioning matrix P:

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b} \qquad \Rightarrow \qquad \underbrace{(P^{-1}AP^{-1})}_{\tilde{A}}\,\underbrace{P\mathbf{x}}_{\tilde{\mathbf{x}}} = \underbrace{P^{-1}\mathbf{b}}_{\tilde{\mathbf{b}}}$$

Note that the matrix $P^{-1}AP^{-1}$ is symmetric positive definite. We will defer discussion of the construction of an appropriate P to the end of the section; suffice it to say for the moment that, if P^2 is somehow "close" to A, then the problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ is a well conditioned problem (because $\tilde{A} = P^{-1}AP^{-1} \approx I$) and can be solved rapidly (with a small number of iterations) using the conjugate gradient approach.

The computation of $\tilde{A}$ might be prohibitively expensive and destroy any sparsity structure of A. We now show that it is not actually necessary to compute $\tilde{A}$ and $\tilde{\mathbf{b}}$ in order to solve the original problem $A\mathbf{x} = \mathbf{b}$ in a well conditioned manner. To begin, we write the conjugate gradient algorithm for the well conditioned problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$. For simplicity, we use a short-hand ("pseudo-code") notation:

```
for $i = 1 : m$
    $\tilde{\mathbf{r}} \leftarrow \begin{cases} \tilde{\mathbf{b}} - \tilde{A}\tilde{\mathbf{x}}, & i = 1 \\ \tilde{\mathbf{r}} - \alpha\tilde{\mathbf{d}}, & i > 1 \end{cases}$
    $res_{\text{old}} = res, \qquad res = \tilde{\mathbf{r}}^T\tilde{\mathbf{r}}$
    $\tilde{\mathbf{p}} \leftarrow \begin{cases} \tilde{\mathbf{r}}, & i = 1 \\ \tilde{\mathbf{r}} + \beta\tilde{\mathbf{p}} \quad \text{where} \quad \beta = res/res_{\text{old}}, & i > 1 \end{cases}$
    $\alpha = res/(\tilde{\mathbf{p}}^T\tilde{\mathbf{d}}) \quad \text{where} \quad \tilde{\mathbf{d}} = \tilde{A}\tilde{\mathbf{p}}$
    $\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}} + \alpha\tilde{\mathbf{p}}$
end
```

For clarity of notation, we have introduced a tilde over each vector involed in this optimization. Note that, in converting the poorly-conditioned problem $A\mathbf{x} = \mathbf{b}$ to the well-conditioned problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$, we made the following definitions: $\tilde{A} = P^{-1}AP^{-1}$, $\tilde{\mathbf{x}} = P\mathbf{x}$, and $\tilde{\mathbf{b}} = P^{-1}\mathbf{b}$. Define now some new intermediate variables $\mathbf{r} = P\tilde{\mathbf{r}}$, $\mathbf{p} = P^{-1}\tilde{\mathbf{p}}$, and $\mathbf{d} = P\tilde{\mathbf{d}}$. With these definitions, we now rewrite *exactly* the above algorithm for solving the well-conditioned problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$, but substitute in the non-tilde variables:

```
for $i = 1 : m$
    $P^{-1}\mathbf{r} \leftarrow \begin{cases} P^{-1}\mathbf{b} - (P^{-1}AP^{-1})P\mathbf{x}, & i = 1 \\ P^{-1}\mathbf{r} - \alpha P^{-1}\mathbf{d}, & i > 1 \end{cases}$
    $res_{\text{old}} = res, \qquad res = (P^{-1}\mathbf{r})^T(P^{-1}\mathbf{r})$
    $P\mathbf{p} \leftarrow \begin{cases} P^{-1}\mathbf{r}, & i = 1 \\ P^{-1}\mathbf{r} + \beta P\mathbf{p} \quad \text{where} \quad \beta = res/res_{\text{old}}, & i > 1 \end{cases}$
    $\alpha = res/[(P\mathbf{p})^T(P^{-1}\mathbf{d})] \quad \text{where} \quad P^{-1}\mathbf{d} = (P^{-1}AP^{-1})P\mathbf{p}$
    $P\mathbf{x} \leftarrow P\mathbf{x} + \alpha P\mathbf{p}$
end
```

Now define $M = P^2$ and simplify:

```
for $i = 1 : m$
    $\mathbf{r} \leftarrow \begin{cases} \mathbf{b} - A\mathbf{x}, & i = 1 \\ \mathbf{r} - \alpha\mathbf{d}, & i > 1 \end{cases}$
    $res_{\text{old}} = res, \qquad res = \mathbf{r}^T\mathbf{s} \quad \text{where} \quad \mathbf{s} = M^{-1}\mathbf{r}$
    $\mathbf{p} \leftarrow \begin{cases} \mathbf{s}, & i = 1 \\ \mathbf{s} + \beta\mathbf{p} \quad \text{where} \quad \beta = res/res_{\text{old}}, & i > 1 \end{cases}$
    $\alpha = res/[\mathbf{p}^T\mathbf{d}] \quad \text{where} \quad \mathbf{d} = A\mathbf{p}$
    $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
end
```

This is practically identical to the original conjugate gradient algorithm for solving the problem $A\mathbf{x} = \mathbf{b}$. The new variable $\mathbf{s} = M^{-1}\mathbf{r}$ may be found by solution of the system $M\mathbf{s} = \mathbf{r}$. Thus, when implementing this method, we seek an M for which we can solve this system quickly (e.g., an M which is the product of sparse triangular matrices). Recall that if $M = P^2$ is somehow "close" to A, the problem here (which is actually the solution of $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ via standard conjugate gradient) is well conditioned and converges in a small number of iterations. There are a variety of heuristic techniques to construct an appropriate M. One of the most popular is **incomplete Cholesky factorization**, which constructs a triangular H with $HH^T = M \approx A$ with the following strategy:

```
$H = A$
for $k = 1 : n$
    $H(k,k) = \sqrt{H(k,k)}$
    for $i = k+1 : n$
        if $H(i,k) \neq 0$ then $H(i,k) = H(i,k)/H(k,k)$
    end
    for $j = k+1 : n$
        for $i = j : n$
            if $H(i,j) \neq 0$ then $H(i,j) = H(i,j) - H(i,k)H(j,k)$
        end
    end
end
```

Once H is obtained with this appraoch such that $HH^T = M$, solving the system $M\mathbf{s} = \mathbf{r}$ for $\mathbf{s}$ is similar to solving Gaussian elimination by leveraging an LU decomposition: one first solves the triangular system $H\mathbf{f} = \mathbf{r}$ for the intermediate variable $\mathbf{f}$, then solves the triangular system $H^T\mathbf{s} = \mathbf{f}$ for the desired quantity $\mathbf{s}$.

Note that the triangular factors H and H^T are zero everywhere A is zero. Thus, if A is sparse, the above algorithm can be rewritten in a manner that leverages the sparsity structure of H (akin to the backsubstitution in the Thomas algorithm). Though it is sometimes takes a bit of effort to write an algorithm that efficiently leverages such sparsity structure, as it usually must be done on a case-by-case basis, the benefits of preconditioning are often quite significant and well worth the coding effort which it necessitates.

Motivation and further discussion of incomplete Cholesky factorization, as well as other preconditioning techniques, is deferred to Golub and van Loan (1989).

5.3.4 Extension non-quadratic functions

At each iteration of the conjugate gradient method, there are five things to be done:

1. Determine the (negative of) the gradient direction, $\mathbf{r} = -\boldsymbol{\nabla}\mathcal{J}$,
2. compute the residual $\mathbf{r}^T\mathbf{r}$,
3. determine the necessary momentum β and the corresponding update direction $\mathbf{p} \leftarrow \mathbf{r} + \beta\mathbf{p}$,
4. determine the (scalar) parameter of descent α which minimizes $\mathcal{J}(\mathbf{x} + \alpha\mathbf{p})$, and
5. update $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$.

In the homework, you will extend the codes developed here for quadratic problems to nonquadratic problems, creating two new routines `sd_nq.m` and `cg_nq.m`. Essentially, the algorithm is the same, but $\mathcal{J}(\mathbf{x})$ now lacks the special quadratic structure we assumed in the previous sections. Generalizing the results of the previous sections to nonquadratic problems entails only a few modifications:

1'. Replace the line which determines the gradient direction with a call to a function, which we will call `compute_grad.m`, which calculates the gradient $\boldsymbol{\nabla}\mathcal{J}$ of the nonquadratic function $\mathcal{J}$.

3'. As the function is not quadratic, but the momentum term in the conjugate gradient algorithm is computed using a local quadratic approximation of $\mathcal{J}$, the momentum sometimes builds up in the wrong direction. Thus, the momentum should be reset to zero (*i.e.*, take $\beta = 0$) every R iterations in `cg_nq.m` ($R = 20$ is often a good choice).

4'. Replace the direct computation of α with a call to an (appropriately modified) version of Brent's method to determine α based on a series of function evaluations.

Finally, when working with nonquadratic functions, it is often adventageous to compute the momentum term β according to the formula

$$\beta = \frac{(\mathbf{r}^{(k)} - \mathbf{r}^{(k-1)})^T \mathbf{r}^{(k)}}{(\mathbf{r}^{(k-1)})^T \mathbf{r}^{(k-1)}}$$

The "correction" term $(\mathbf{r}^{(k-1)})^T \mathbf{r}^{(k)}$ is zero using the conjugate gradient approach when the function $\mathcal{J}$ is quadratic. When the function $\mathcal{J}$ is not quadratic, this additional term often serves to nudge the descent direction towards that of a steepest descent step in regions of the function which are highly nonquadratic. This approach is referred to as the **Polak-Ribiere** variant of the conjugate gradient method for nonquadratic functions.

Chapter 6

Differentiation

6.1 Derivation of finite difference formulae

In the simulation of physical systems, one often needs to compute the derivative of a function $f(x)$ which is known only at a discrete set of grid points $x_0, x_1, \ldots, x_N$. Effective formulae for computing such approximations, known as finite difference formulae, may be derived by combination of one or more Taylor series expansions. For example, defining $f_j = f(x_j)$, the Taylor series expansion for f at point x_{j+1} in terms of f and its derivatives at the point x_j is given by

$$f_{j+1} = f_j + (x_{j+1} - x_j)f'_j + \frac{(x_{j+1} - x_j)^2}{2!} f''_j + \frac{(x_{j+1} - x_j)^3}{3!} f'''_j + \ldots$$

Defining $h_j = x_{j+1} - x_j$, rearrangement of the above equation leads to

$$f'_j = \frac{f_{j+1} - f_j}{h_j} - \frac{h_j}{2} f''_j + \ldots$$

In these notes, we will indicate a uniform mesh by denoting its (constant) grid spacing as h (without a subscript), and we will denote a nonuniform ("stretched") mesh by denoting the grid spacing as h_j (with a subscript). We will assume the mesh is uniform unless indicated otherwise. For a uniform mesh, we may write the above equation as

$$f'_j = \frac{f_{j+1} - f_j}{h} + O(h),$$

where $O(h)$ denotes the contribution from all terms which have a power of h which is greater then or equal to one. Neglecting these "higher-order terms" for sufficiently small h, we can approximate the derivative as

$$\left(\frac{\delta f}{\delta x}\right)_j = \frac{f_{j+1} - f_j}{h},$$

which is referred to as the first-order forward difference formula for the first derivative. The neglected term with the highest power of h (in this case, $-h\, f''_j/2$) is referred to as the leading-order error. The exponent of h in the leading-order error is the order of accuracy of the method. For a sufficiently fine initial grid, if we refine the grid spacing further by a factor of two, the truncation error of this method is also reduced by approximately a factor of 2, indicating a "first-order" behavior.

Similarly, by expanding f_{j-1} about the point x_j and rearranging, we obtain

$$\left(\frac{\delta f}{\delta x}\right)_j = \frac{f_j - f_{j-1}}{h},$$

which is referred to as the first-order backward difference formula for the first derivative. Higher-order (more accurate) schemes can be derived by combining Taylor series of the function f at various points near the point x_j. For example, the widely used second-order central difference formula for the first derivative can be obtained by subtraction of the two Taylor series expansions

$$\begin{aligned} f_{j+1} &= f_j + h\, f_j' + \frac{h^2}{2} f_j'' + \frac{h^3}{6} f_j''' + \ldots \\ f_{j-1} &= f_j - h\, f_j' + \frac{h^2}{2} f_j'' - \frac{h^3}{6} f_j''' + \ldots, \end{aligned}$$

leading to

$$f_{j+1} - f_{j-1} = 2\,h\, f_j' + \frac{h^3}{3} f_j''' + \ldots$$

and thus

$$f_j' = \frac{f_{j+1} - f_{j-1}}{2h} - \frac{h^2}{6} f_j''' + \ldots \qquad \Rightarrow \qquad \left(\frac{\delta f}{\delta x}\right)_j = \frac{f_{j+1} - f_{j-1}}{2h}.$$

Similar formulae can be derived for second-order derivatives (and higher). For example, by adding the above Taylor series expansions instead of subtracting them, we obtain the second-order central difference formula for second derivative given by

$$f_j'' = \frac{f_{j+1} - 2f_j + f_{j-1}}{h^2} - \frac{h^2}{24} f_j^{IV} + \ldots \qquad \Rightarrow \qquad \left(\frac{\delta^2 f}{\delta x^2}\right)_j = \frac{f_{j+1} - 2f_j + f_{j-1}}{h^2}.$$

In general, we can obtain higher accuracy if we include more points. By approximate linear combination of four different Taylor Series expansions, a fourth-order central difference formula for the first derivative may be found, which takes the form

$$\left(\frac{\delta f}{\delta x}\right)_j = \frac{f_{j-2} - 8\, f_{j-1} + 8\, f_{j+1} - f_{j+2}}{12h}.$$

The main difficulty with higher order formulae occurs near boundaries of the domain. They require the functional values at points outside the domain which are not available. Near boundaries one usually resorts to lower order formulae.

6.2 Taylor Tables

There is a general technique for constructing finite difference formulae using a tool known as a Taylor Table. This technique is best illustrated by example. Suppose we want to construct the most accurate finite difference scheme for the first derivative that involves the function values f_j, f_{j+1}, and f_{j+2}. Given this restriction on the information used, we seek the highest order of accuracy that can be achieved. That is, if we take

$$f_j' - \sum_{\kappa=0}^{2} a_\kappa\, f_{j+\kappa} = \epsilon, \tag{6.1}$$

where the a_κ are the coefficients of the finite difference formula sought, then we desire to select the a_κ such that the error ϵ is as large a power of h as possible (and thus will vanish rapidly upon refinement of the grid). It is convenient to organize the Taylor series of the terms in the above formula using a "Taylor Table" of the form

	f_j	f'_j	f''_j	f'''_j
f'_j	0	1	0	0
$-a_0 f_j$	$-a_0$	0	0	0
$-a_1 f_{j+1}$	$-a_1$	$-a_1 h$	$-a_1 \frac{h^2}{2}$	$-a_1 \frac{h^3}{6}$
$-a_2 f_{j+2}$	$-a_2$	$-a_2 (2h)$	$-a_2 \frac{(2h)^2}{2}$	$-a_2 \frac{(2h)^3}{6}$

The leftmost column of this table contains all of the terms on the left-hand side of (6.1). The elements to the right, when multiplied by the corresponding terms at the top of each column and summed, yield the Taylor series expansion of each of the terms to the left. Summing up all of these terms, we get the error ϵ expanded in terms of powers of the grid spacing h. By proper choice of the available degrees of freedom $\{a_0, a_1, a_2\}$, we can set several of the coefficients of this polynomial equal to zero, thereby making ϵ as high a power of h as possible. For small h, this is a good way of making this error small. In the present case, we have three free coefficients, and can set the coefficients of the first three terms to zero:

$$\left.\begin{aligned} -a_0 - a_1 - a_2 &= 0 \\ 1 - a_1 h - a_2 (2h) &= 0 \\ -a_1 \frac{h^2}{2} - a_2 \frac{(2h)^2}{2} &= 0 \end{aligned}\right\} \quad \Rightarrow \quad \begin{pmatrix} -1 & -1 & -1 \\ 0 & -1 & -2 \\ 0 & -1/2 & -2 \end{pmatrix} \begin{pmatrix} a_0 h \\ a_1 h \\ a_2 h \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} -3/(2h) \\ 2/h \\ -1/(2h) \end{pmatrix}$$

This simple linear system may be solved either by hand or with Matlab. The resulting second-order forward difference formula for the first derivative is

$$\left(\frac{\delta f}{\delta x}\right)_j = \frac{-3 f_j + 4 f_{j+1} - f_{j+2}}{2h},$$

and the leading-order error, which can be determined by multiplying the first non-zero column sum by the term at the top of the corresponding column, is

$$\left(-a_1 \frac{h^3}{6} - a_2 \frac{(2h)^3}{6}\right) f'''_j = \frac{h^2}{3} f'''_j,$$

revealing that this scheme is second-order accurate.

6.3 Padé Approximations

By including both nearby function evaluations and nearby gradient approximations on the left hand side of an expression like (6.1), we can derive a banded system of equations which can easily be solved to determine a numerical approximation of the f'_j. This approach is referred to as **Padé approximation**. Again illustrating by example, consider the equation

$$b_{-1} f'_{j-1} + f'_j + b_1 f'_{j+1} - \sum_{k=-1}^{1} a_\kappa f_{j+\kappa} = \epsilon. \tag{6.2}$$

Leveraging the Taylor series expansions

$$f_{j+1} = f_j + h\,f'_j + \frac{h^2}{2}\,f''_j + \frac{h^3}{6}\,f'''_j + \frac{h^4}{24}\,f^{(iv)}_j + \frac{h^5}{120} f^{(v)}_j + \ldots$$
$$f'_{j+1} = f'_j + h\,f''_j + \frac{h^2}{2}\,f'''_j + \frac{h^3}{6}\,f^{(iv)}_j + \frac{h^4}{24}\,f^{(v)}_j + \ldots\,,$$

the corresponding Taylor table is

	f_j	f'_j	f''_j	f'''_j	$f^{(iv)}_j$	$f^{(v)}_j$
$b_{-1}\,f'_{j-1}$	0	b_{-1}	$b_{-1}(-h)$	$b_{-1}\frac{(-h)^2}{2}$	$b_{-1}\frac{(-h)^3}{6}$	$b_{-1}\frac{(-h)^4}{24}$
f'_j	0	1	0	0	0	0
$b_1\,f'_{j+1}$	0	b_1	$b_1 h$	$b_1\frac{h^2}{2}$	$b_1\frac{h^3}{6}$	$b_1\frac{h^4}{24}$
$-a_{-1}\,f_{j-1}$	$-a_{-1}$	$-a_{-1}(-h)$	$-a_{-1}\frac{(-h)^2}{2}$	$-a_{-1}\frac{(-h)^3}{6}$	$-a_{-1}\frac{(-h)^4}{24}$	$-a_{-1}\frac{(-h)^5}{120}$
$-a_0\,f_j$	$-a_0$	0	0	0	0	0
$-a_1\,f_{j+1}$	$-a_1$	$-a_1 h$	$-a_1\frac{h^2}{2}$	$-a_1\frac{h^3}{6}$	$-a_1\frac{h^4}{24}$	$-a_1\frac{h^5}{120}$

We again use the available degrees of freedom $\{b_{-1}, b_1, a_{-1}, a_0, a_1\}$ to obtain the highest possible accuracy in (6.2). Setting the sums of the first five columns equal to zero leads to the linear system

$$\begin{pmatrix} 0 & 0 & -1 & -1 & -1 \\ 1 & 1 & h & 0 & -h \\ -h & h & -h^2/2 & 0 & -h^2/2 \\ h^2/2 & h^2/2 & h^3/6 & 0 & -h^3/6 \\ -h^3/6 & h^3/6 & -h^4/24 & 0 & -h^4/24 \end{pmatrix} \begin{pmatrix} b_{-1} \\ b_1 \\ a_{-1} \\ a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

which is equivalent to

$$\begin{pmatrix} 0 & 0 & -1 & -1 & -1 \\ 1 & 1 & 1 & 0 & -1 \\ -1 & 1 & -1/2 & 0 & -1/2 \\ 1/2 & 1/2 & 1/6 & 0 & -1/6 \\ -1/6 & 1/6 & -1/24 & 0 & -1/24 \end{pmatrix} \begin{pmatrix} b_{-1} \\ b_1 \\ a_{-1}h \\ a_0 h \\ a_1 h \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \Rightarrow \qquad \begin{pmatrix} b_{-1} \\ b_1 \\ a_{-1} \\ a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 1/4 \\ 1/4 \\ 3/(4h) \\ 0 \\ -3/(4h) \end{pmatrix}$$

Thus, the system to be solved to determine the numerical approximation of the derivatives at each grid point has a typical row given by

$$\frac{1}{4}\Big(\frac{\delta f}{\delta x}\Big)_{j+1} + \Big(\frac{\delta f}{\delta x}\Big)_j + \frac{1}{4}\Big(\frac{\delta f}{\delta x}\Big)_{j-1} = \frac{3}{4h}\Big(f_{j+1} - f_{j-1}\Big),$$

and has a leading-order error of $h^4 f^{(v)}_j/30$, and thus is fourth-order accurate. Writing out this equation for all values of j on the interior leads to the tridiagonal, diagonally-dominant system

$$\begin{pmatrix} \ddots & \ddots & \ddots & & \\ & \frac{1}{4} & 1 & \frac{1}{4} & \\ & & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} \vdots \\ \Big(\frac{\delta f}{\delta x}\Big)_{j-1} \\ \Big(\frac{\delta f}{\delta x}\Big)_j \\ \Big(\frac{\delta f}{\delta x}\Big)_{j+1} \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ \vdots \\ \frac{3}{4h}\Big(f_{j+1} - f_{j-1}\Big) \\ \vdots \\ \vdots \end{pmatrix}.$$

At the endpoints, a different treatment is needed; a central expression (Padé or otherwise) cannot be used at x_0 because x_{-1} is outside of our available grid of data. One commonly settles on a lower-order forward (backward) expression at the left (right) boundary in order to close this set of equations in a nonsingular manner. This system can then be solved efficiently for the numerical approximation of the derivative at all of the gridpoints using the Thomas algorithm.

6.4 Modified wavenumber analysis

The order of accuracy is usually the primary indicator of the accuracy of finite-difference formulae; it tells us how mesh refinement improves the accuracy. For example, for a sufficiently fine grid, mesh refinement by a factor of two improves the accuracy of a second-order finite-difference scheme by a factor of four, and improves the accuracy of a fourth-order scheme by a factor of sixteen. Another method for quantifying the accuracy of a finite difference formula that yields further information is called the **modified wavenumber** approach. To illustrate this approach, consider the harmonic function given by

$$f(x) = e^{i\,k\,x} \tag{6.3}$$

(Alternatively, we can also do this derivation with sines and cosines, but complex exponentials tend to make the algebra easier.) The exact derivative of this function is

$$f' = i\,k\,e^{i\,k\,x} = i\,k\,f \tag{6.4}$$

We now ask how accurately the second order central finite-difference scheme, for example, computes the derivative of f. Let us discretize the x axis with a uniform mesh,

$$x_j = h \cdot j \qquad \text{where} \qquad j = 0, 1, 2, \dots N, \qquad \text{and} \qquad h = \frac{L}{N}.$$

The finite-difference approximation for the derivative which we consider here is

$$\left.\frac{\delta f}{\delta x}\right|_j = \frac{f_{j+1} - f_{j-1}}{2h}$$

Substituting for $f_j = e^{i\,k\,x_j}$, noting that $x_{j+1} = x_j + h$ and $x_{j-1} = x_j - h$, we obtain

$$\left.\frac{\delta\,f}{\delta\,x}\right|_j = \frac{e^{i\,k\,(x_j+h)} - e^{i\,k\,(x_j-h)}}{2h} = \frac{e^{i\,k\,h} - e^{-i\,k\,h}}{2h} f_j = i\frac{\sin(h\,k)}{h} f_j \triangleq i\,k'\,f_j \tag{6.5}$$

where

$$k' \triangleq \frac{\sin\,h\,k}{h} \qquad \Rightarrow \qquad h\,k' = \sin(h\,k)$$

By analogy with (6.4), k' is called the modified wave number for this second-order finite difference scheme. In an analogous manner, one can derive modified wave numbers for any finite difference formula. A measure of accuracy of the finite-difference scheme is provided by comparing the modified wavenumber k', which appears in the numerical approximation of the derivative (6.5), with the actual wavenumber k, which appears in the exact expression for the derivative (6.4). For small wavenumbers, the numerical approximation of the derivative on our discrete grid is usually pretty good ($k' \approx k$), but for large wavenumbers, the numerical approximation is degraded. As $k \to \pi/h$, the numerical approximation of the derivative is completely off.

As another example, consider the modified wavenumber for the fourth-order expression of first derivative given by

$$\frac{\delta f}{\delta x}\Big|_j = \frac{f_{j-2} - 8f_{j-1} + 8f_{j+1} - f_{j+2}}{12h} = \frac{2}{3h}(f_{j+1} - f_{j-1}) - \frac{1}{12h}(f_{j+2} - f_{j-2}).$$

Inserting (6.3) and manipulating as before, we obtain:

$$\frac{\delta f}{\delta x}\Big|_j = \frac{2}{3h}(e^{i\,k\,h} - e^{-i\,k\,h})f_j - \frac{1}{12}(e^{i\,k\,2h} - e^{-i\,k\,2h})f_j = i\Big[\frac{4}{3h}\sin(hk) - \frac{1}{6h}\sin(2hk)\Big]f_j \triangleq ik' f_j$$

$$\Rightarrow hk' = \frac{4}{3}\sin(hk) - \frac{1}{6}\sin(2hk)$$

Consider now our fourth-order Padé approximation:

$$\frac{1}{4}\left(\frac{\delta f}{\delta x}\right)_{j+1} + \left(\frac{\delta f}{\delta x}\right)_j + \frac{1}{4}\left(\frac{\delta f}{\delta x}\right)_{j-1} = \frac{3}{4h}\left(f_{j+1} - f_{j-1}\right)$$

Approximating the modified wavenumber at points x_{j+1} and x_{j-1} with their corresponding numerical approximations

$$\left(\frac{\delta f}{\delta x}\right)_{j+1} = i\,k'\,e^{i\,k\,x_{j+1}} = i\,k'\,e^{i\,k\,h}\,f_j \qquad \text{and} \qquad \left(\frac{\delta f}{\delta x}\right)_{j-1} = i\,k'\,e^{i\,k\,x_{j-1}} = i\,k'\,e^{-i\,k\,h}\,f_j$$

and inserting (6.3) and manipulating as before, we obtain:

$$i\,k'\left(\frac{1}{4}e^{i\,k\,h} + 1 + \frac{1}{4}e^{-i\,k\,h}\right)f_j = \frac{3}{4h}(e^{i\,k\,h} - e^{-i\,k\,h})f_j$$

$$i\,k'\left(1 + \frac{1}{2}\cos(hk)\right)f_j = i\frac{3}{2h}\sin(hk)f_j$$

$$\Rightarrow hk' = \frac{\frac{3}{2}\sin(hk)}{1 + \frac{1}{2}\cos(hk)}$$

6.5 Alternative derivation of differentiation formulae

Consider now the Lagrange interpolation of the three points $\{x_{i-1}, f_{i-1}\}$, $\{x_i, f_i\}$, and $\{x_{i+1}, f_{i+1}\}$, given by:

$$f(x) = \frac{(x - x_i)(x - x_{i+1})}{(x_{i-1} - x_i)(x_{i-1} - x_{i+1})}f_{i-1} + \frac{(x - x_{i-1})(x - x_{i+1})}{(x_i - x_{i-1})(x_i - x_{i+1})}f_i + \frac{(x - x_{i-1})(x - x_i)}{(x_{i+1} - x_{i-1})(x_{i+1} - x_i)}f_{i+1}$$

Differentiate this expression with respect to x and then evaluating at $x = x_i$ gives

$$f'(x_i) = \frac{(-h)}{(-h)(-2h)}f_{i-1} + 0 + \frac{(h)}{(2h)(h)}f_{i+1} = \frac{f_{i+1} - f_{i-1}}{2h},$$

which is the same as what we get with Taylor Table.

Chapter 7

Integration

Differentiation and integration are two essential tools of calculus which we need to solve engineering problems. The previous chapter discussed methods to approximate derivatives numerically; we now turn to the problem of numerical integration. In the setting we discuss in the present chapter, in which we approximate the integral of a given function over a specified domain, this procedure is usually referred to as **numerical quadrature**.

7.1 Basic quadrature formulae

7.1.1 Techniques based on Lagrange interpolation

Consider the problem of integrating a function f on the interval $[a, c]$ when the function is evaluated only at a limited number of discrete gridpoints. One approach to approximating the integral of f is to integrate the lowest-order polynomial that passes through a specified number of function evaluations using the formulae of Lagrange interpolation.

For example, if the function is evaluated at the midpoint $b = (a + c)/2$, then (defining $h = c - a$) we can integrate a *constant* approximation of the function over the interval $[a, c]$, leading to the **midpoint rule**:

$$\int_a^c f(x)\,dx \approx \int_a^c \Big[f(b)\Big] dx = h\,f(b) \triangleq M(f). \tag{7.1}$$

If the function is evaluated at the two endpoints a and c, we can integrate a *linear* approximation of the function over the interval $[a, c]$, leading to the **trapezoidal rule**:

$$\int_a^c f(x)\,dx \approx \int_a^c \Big[\frac{(x-c)}{(a-c)}\,f(a) + \frac{(x-a)}{(c-a)}\,f(c)\Big] dx = h\,\frac{f(a)+f(c)}{2} \triangleq T(f). \tag{7.2}$$

If the function is known at all three points a, b, and c, we can integrate a *quadratic* approximation of the function over the interval $[a, c]$, leading to **Simpson's rule**:

$$\begin{aligned}\int_a^c f(x)\,dx &\approx \int_a^c \Big[\frac{(x-b)(x-c)}{(a-b)(a-c)}\,f(a) + \frac{(x-a)(x-c)}{(b-a)(b-c)}\,f(b) + \frac{(x-a)(x-b)}{(c-a)(c-b)}\,f(c)\Big] dx \\ &= \ldots = h\,\frac{f(a)+4f(b)+f(c)}{6} \triangleq S(f).\end{aligned} \tag{7.3}$$

Don't forget that symbolic manipulation packages like Maple (which is included with Matlab) and Mathematica are well suited for this sort of algebraic manipulation.

7.1.2 Extension to several gridpoints

Recall that Lagrange interpolations are ill-behaved near the endpoints when the number of gridpoints is large. Thus, it is generally ill-advised to continue the approach of the previous section to function approximations which are higher order than quadratic. Instead, better results are usually obtained by applying the formulae of §7.1.1 repeatedly over several smaller subintervals. Defining a numerical grid of points $\{x_0, x_1, \dots, x_n\}$ distributed over the interval $[L, R]$, the intermediate gridpoints $x_{i-1/2} = (x_{i-1} + x_i)/2$, the grid spacing $h_i = (x_i - x_{i-1})$, and the function evaluations $f_i = f(x_i)$, the numerical approximation of the integral of $f(x)$ over the interval $[L, R]$ via the midpoint rule takes the form

$$\int_L^R f(x)\,dx \approx \sum_{i=1}^n h_i\, f_{i-1/2}, \tag{7.4}$$

numerical approximation of the integral via the trapezoidal rule takes the form

$$\int_L^R f(x)\,dx \approx \sum_{i=1}^n h_i\, \frac{f_{i-1}+f_i}{2} = \frac{h}{2}\Big[f_0 + f_n + 2\sum_{i=1}^{n-1} f_i\Big], \tag{7.5}$$

and numerical approximation of the integral via Simpson's rule takes the form

$$\int_L^R f(x)\,dx \approx \sum_{i=1}^n h_i\, \frac{f_{i-1}+4f_{i-\frac{1}{2}}+f_i}{6} = \frac{h}{6}\Big[f_0 + f_n + 4\sum_{i=1}^{n} f_{i-\frac{1}{2}} + 2\sum_{i=1}^{n-1} f_i\Big], \tag{7.6}$$

where the rightmost expressions assume a uniform grid in which the grid spacing h is constant.

7.2 Error Analysis of Integration Rules

In order to quantify the accuracy of the integration rules we have proposed so far, we may again turn to Taylor series analysis. For example, replacing $f(x)$ with its Taylor series approximation about b and integrating, we obtain

$$\begin{aligned}
\int_a^c f(x)\,dx &= \int_a^c \Big[f(b) + (x-b)\,f'(b) + \frac{1}{2}\,(x-b)^2\,f''(b) + \frac{1}{3}\,(x-b)^3\,f'''(b) + \dots\Big]\,dx \\
&= h\,f(b) + \frac{1}{2}\,(x-b)^2\Big|_a^c\,f'(b) + \frac{1}{6}\,(x-b)^3\Big|_a^c\,f''(b) + \dots \\
&= h\,f(b) + \frac{h^3}{24}\,f''(b) + \frac{h^5}{1920}\,f^{(iv)}(b) + \dots
\end{aligned} \tag{7.7}$$

Thus, if the integral is approximated by the midpoint rule (7.1), the leading-order error is proportional to h^3, and the approximation of the integral over this single interval is third-order accurate. Note also that if all even derivatives of f happen to be zero (for example, if $f(x)$ is linear in x), the midpoint rule integrates the function *exactly.*

The question of the accuracy of a particular integration rule over a single interval is often not of much interest, however. A more relevant measure is the rate of convergence of the integration rule when applied over several gridpoints on a given interval $[L,R]$ as the numerical grid is refined. For example, consider the formula (7.4) on n subintervals ($n+1$ gridpoints). As $h \propto 1/n$ (the width of each subinterval is inversely proportional to the number of subintervals), the error over the *entire* interval $[L,R]$ of the numerical integration will be proportional to $n\,h^3 = h^2$. Thus, for approximations of the integral on a given interval $[L,R]$ as the computational grid is refined, the **midpoint rule is second-order accurate**.

Consider now the Taylor series approximations of $f(a)$ and $f(c)$ about b:

$$f(a) = f(b) + \left(\frac{-h}{2}\right) f'(b) + \frac{1}{2}\left(\frac{-h}{2}\right)^2 f''(b) + \frac{1}{6}\left(\frac{-h}{2}\right)^3 f'''(b) + \dots$$
$$f(c) = f(b) + \left(\frac{h}{2}\right) f'(b) + \frac{1}{2}\left(\frac{h}{2}\right)^2 f''(b) + \frac{1}{6}\left(\frac{h}{2}\right)^3 f'''(b) + \dots$$

Combining these expressions gives

$$\frac{f(a)+f(c)}{2} = f(b) + \frac{1}{8}h^2 f''(b) + \frac{1}{384} f^{(iv)}(b) + \dots$$

Solve for $f(b)$ and substituting into (7.7) yields

$$\int_a^c f(x)\,dx = h\,\frac{f(a)+f(c)}{2} - \frac{h^3}{12} f''(b) - \frac{h^5}{480} f^{(iv)}(b) + \dots \tag{7.8}$$

As with the midpoint rule, the leading-order error of the trapezoidal rule (7.2) is proportional to h^3, and thus the trapezoidal approximation of the integral over this single interval is third-order accurate. Again, the most relevant measure is the rate of convergence of the integration rule (7.5) when applied over several gridpoints on a given interval $[L,R]$ as the number of gridpoints is increased; in such a setting, as with the midpoint rule, the **trapezoidal rule is second-order accurate**.

Note from (7.1)-(7.3) that $S(f) = \frac{2}{3}M(f) + \frac{1}{3}T(f)$. Adding 2/3 times equation (7.7) plus 1/3 times equation (7.8) gives

$$\int_a^c f(x)\,dx = h\,\frac{f(a)+4f(b)+f(c)}{6} - \frac{h^5}{2880} f^{(iv)}(b) + \dots$$

The leading-order error of Simpson's rule (7.3) is therefore proportional to h^5, and thus the approximation of the integral over this single interval using this rule is fifth-order accurate. Note that if all even derivatives of f higher than three happen to be zero (e.g., if $f(x)$ is cubic in x), then Simpson's rule integrates the function *exactly*. Again, the most relevant measure is the rate of convergence of the integration rule (7.6) when applied over several gridpoints on the given interval $[L,R]$ as the number of gridpoints is increased; in such a setting, **Simpson's rule is fourth-order accurate**.

7.3 Romberg integration

In the previous derivation, it became evident that keeping track of the leading-order error of a particular numerical formula can be a useful thing to do. In fact, Simpson's rule can be constructed simply by determining the specific linear combination of the midpoint rule and the trapezoidal rule for which the leading-order error term vanishes. We now pursue further such a constructive procedure, with a technique known as Richardson extrapolation, in order to determine even higher-order approximations of the integral on the interval $[L,R]$ using linear combinations of several trapezoidal approximations of the integral on a series of successively finer grids. The approach we will present is commonly referred to as Romberg integration.

Recall first that the error of the trapezoidal approximation of the integral (7.5) on the given interval $[L, R]$ may be written

$$I \triangleq \int_L^R f(x)\,d\,x = \frac{h}{2}\Big[f_0 + f_n + 2\sum_{i=1}^{n-1} f_i\Big] + c_1\,h^2 + c_2\,h^4 + c_3\,h^4 + c_3\,h^6 + \dots$$

Let us start with $n_1 = 2$ and $h_1 = (R-L)/n_1$ and iteratively refine the grid. Define the trapezoidal approximation of the integral on a numerical grid with $n = n_l = 2^l$ (e.g., $h = h_l = (R-L)/n_l$) as:

$$I_{l,1} = \frac{h_l}{2}\Big[f_0 + f_{n_l} + 2\sum_{i=1}^{n_l-1} f_i\Big]$$

We now examine the truncation error (in terms of h_1) as the grid is refined. Note that at the first level we have

$$I_{1,1} = I - c_1\,h_1^2 - c_2\,h_1^4 - c_3\,h_1^6 \dots,$$

whereas at the second level we have

$$\begin{aligned} I_{2,1} &= I - c_1\,h_2^2 - c_2\,h_2^4 - c_3\,h_2^6 - \dots \\ &= I - c_1\,\frac{h_1^2}{4} - c_2\,\frac{h_1^4}{16} - c_3\,\frac{h_1^6}{64} - \dots \end{aligned}$$

Assuming the coefficients c_i (which are proportional to the various derivatives of f) vary only slowly in space, we can eliminate the error proportional to h_1^2 by taking a linear combination of $I_{1,1}$ and $I_{2,1}$ to obtain:

$$I_{2,2} = \frac{4\,I_{2,1} - I_{1,1}}{3} = I + \frac{1}{4}\,c_2\,h_1^4 + \frac{5}{16}\,c_3\,h_1^6 + \dots$$

(This results in Simpson's rule, if you do all of the appropriate substitutions.) Continuing to the third level of grid refinement, the trapezoidal approximation of the integral satisfies

$$\begin{aligned} I_{3,1} &= I - c_1\,h_3^2 - c_2\,h_3^4 - c_3\,h_3^6 - \dots \\ &= I - c_1\,\frac{h_1^2}{16} - c_2\,\frac{h_1^4}{256} - c_3\,\frac{h_1^6}{4096} - \dots \end{aligned}$$

First, eliminate terms proportional to h_1^2 by linear combination with $I_{2,1}$:

$$I_{3,2} = \frac{4\,I_{3,1} - I_{2,1}}{3} = I + \frac{1}{64}\,c_2\,h_1^4 + \frac{5}{1024}\,c_3\,h_1^6 + \dots$$

Then, eliminate terms proportional to h_1^4 by linear combination with $I_{2,2}$:

$$I_{3,3} = \frac{16\,I_{3,2} - I_{2,2}}{15} = I - \frac{1}{64}\,c_3\,h_1^6 - \dots$$

This process may be repeated to provide increasingly higher-order approximations to the integral I. The structure of the refinements is:

$$\begin{tikzcd}
\text{Gridpoints} & \text{2^{nd}-Order Approximation} & \text{4^{th}-Order Correction} & \text{6^{th}-Order Correction} & \text{8^{th}-Order Correction} \\
n_1 = 2^1 = 2 & I_{1,1} \arrow[dr] & & & \\
n_2 = 2^2 = 4 & I_{2,1} \arrow[r] \arrow[dr] & I_{2,2} \arrow[dr] & & \\
n_3 = 2^3 = 8 & I_{3,1} \arrow[r] \arrow[dr] & I_{3,2} \arrow[r] \arrow[dr] & I_{3,3} \arrow[dr] & \\
n_4 = 2^4 = 16 & I_{4,1} \arrow[r] & I_{4,2} \arrow[r] & I_{4,3} \arrow[r] & I_{4,4}
\end{tikzcd}$$

The general form for the correction term (for $k \geq 2$) is:

$$I_{l,k} = \frac{4^{(k-1)}\, I_{l,(k-1)} - I_{(l-1),(k-1)}}{4^{(k-1)} - 1}$$

Matlab implementation

A matlab implementation of Romberg integration is given below. A straightforward test code is provided at the class web site.

```
function [int,evals] = int_romb(L,R,refinements)
% Integrate the function defined in compute_f.m from x=L to x=R using
% Romberg integration to provide the maximal order of accuracy with a
% given number of grid refinements.

evals=0;  toplevel=refinements+1;
for level=1:toplevel
   % Approximate the integral with the trapezoidal rule on 2^level subintervals
   n=2^level;
   [I(level,1),evals_temp]= int_trap(L,R,n);
   evals=evals+evals_temp;

   % Perform several corrections based on I at the previous level.
   for k=2:level
      I(level,k) = (4^(k-1)*I(level,k-1) - I(level-1,k-1))/(4^(k-1) -1);
   end
end
int=I(toplevel,toplevel);
% end int_romb.m
```

A simple function to perform the trapezoidal integrations is given by:

```
function [int,evals] = int_trap(L,R,n)
% Integrate the function defined in compute_f.m from x=L to x=R on
% n equal subintervals using the trapezoidal rule.

h=(R-L)/n;
```

```
int=0.5*(compute_f(L) + compute_f(R));
for i=1:n-1
   x=L+h*i;   int=int+compute_f(x);
end
int=h*int;  evals=2+(n-1);
% end int_trap
```

7.4 Adaptive Quadrature

Often, it is wasteful to use the same grid spacing h everywhere in the interval of integration $[L, R]$. Ideally, one would like to use a fine grid in the regions where the integrand varies quickly and a coarse grid where the integrand varies slowly. As we now show, adaptive quadrature techniques automatically adjust the grid spacing in just such a manner.

Suppose we seek a numerical approximation $\tilde{I}$ of the integral I such that

$$|\tilde{I} - I| \leq \epsilon,$$

where ϵ is the error tolerance provided by the user. The idea of adaptive quadrature is to spread out this error in our approximation of the integral proportionally across the subintervals spanning $[L, R]$. To demonstrate this technique, we will use Simpson's rule as the base method. First, divide the interval $[L, R]$ into several subintervals with the numerical grid $\{x_0, x_1, \ldots, x_n\}$. Evaluating the integral on a particular subinterval $[x_{i-1}, x_i]$ with Simpson's rule yields

$$S_i = \frac{h_i}{6}\,[f(x_i - h_i) + 4\,f(x_i - \frac{h_i}{2}) + f(x_i)].$$

Dividing this particular subinterval in half and summing Simpson's approximations of the integral on each of these smaller subintervals yields

$$S_i^{(2)} = \frac{h_i}{12}\,[f(x_i - h_i) + 4f(x_i - \frac{3\,h_i}{4}) + 2f(x_i - \frac{h_i}{2}) + 4f(x_i - \frac{h_i}{4}) + f(x_i)].$$

The essential idea is to compare the two approximations S_i and $S_i^{(2)}$ to obtain an estimate for the accuracy of $S_i^{(2)}$. If the accuracy is acceptable, we will use $S_i^{(2)}$ for the approximation of the integral on this interval; otherwise, the adaptive procedure further subdivides the interval and the process is repeated. Let I_i denote the exact integral on $[x_{i-1}, x_i]$. From our error analysis, we know that

$$I_i - S_i = c\,h_i^5\,f^{(iv)}\,(x_i - \frac{h_i}{2}) + \ldots \tag{7.9}$$

and

$$I_i - S_i^{(2)} = c\left(\frac{h_i}{2}\right)^5\left[f^{(iv)}\left(x_i - \frac{3h_i}{4}\right) + f^{(iv)}\left(x_i - \frac{h_i}{4}\right)\right] + \ldots$$

Each of the terms in the bracket in the last expression can be expanded in a Taylor series about the point $x_i - \frac{h_i}{2}$:

$$\begin{aligned} f^{(iv)}\left(x_i - \frac{3h_i}{4}\right) &= f^{(iv)}\left(x_i - \frac{h_i}{2}\right) - \frac{h_i}{4}\,f^{(v)}\left(x_i - \frac{h_i}{2}\right) + \ldots \\ f^{(iv)}\left(x_i - \frac{h_i}{4}\right) &= f^{(iv)}\left(x_i - \frac{h_i}{2}\right) + \frac{h_i}{4}\,f^{(v)}\left(x_i - \frac{h_i}{2}\right) + \ldots \end{aligned}$$

Thus,

$$I_i - S_i^{(2)} = 2c\Big(\frac{h_i}{2}\Big)^5 \Big[f^{(iv)}\Big(x_i - \frac{h_i}{2}\Big)\Big] + \dots \tag{7.10}$$

Subtracting (7.10) from (7.9), we obtain

$$S_i^{(2)} - S_i = \frac{15}{16}\,c\,h_i^5\,f^{(iv)}\Big(x_i - \frac{h_i}{2}\Big) + \dots,$$

and substituting into the RHS of (7.10) reveals that

$$I - S_i^{(2)} \approx \frac{1}{15}\,(S_i^{(2)} - S_i).$$

Thus, the error in $S_i^{(2)}$ is, to leading order, about $\frac{1}{15}$ of the difference between S_i and $S_i^{(2)}$. The good news is that this difference can easily be computed. If

$$\frac{1}{15}\,|S_i^{(2)} - S_i| \,\le\, \frac{h_i}{R - L}\,\epsilon,$$

then $S_i^{(2)}$ is sufficiently accurate for the subinterval $[x_{i-1}, x_i]$, and we move on to the next subinterval. If this condition is not satisfied, the subinterval $[x_{i-1}, x_i]$ will be subdivided further. The essential idea of adaptive quadrature is thus to spread evenly the error of the numerical approximation of the integral (or, at least, an approximation of this error) over the entire interval $[L, R]$ by selective refinements of the numerical grid. Similar schemes may also be pursued for the other base integration schemes such as the trapezoidal rule. As with Richardson extrapolation, knowledge of the truncation error can be leveraged to estimate the accuracy of the numerical solution without knowing the exact solution.

Chapter 8

Ordinary differential equations

Consider first a scalar, first-order ordinary differential equation (ODE) of the form

$$\frac{dy}{dt} = f(y,t) \quad \text{with} \quad y(t_0) = y_0. \tag{8.1}$$

The problem we address now is the advancement of such a system in time by integration of this differential equation. As the quantity being integrated, f, is itself a function of the result of the integration, y, the problem of integration of an ODE is fundamentally different than the problem of numerical quadrature discussed in §7, in which the function being integrated was given. Note that ODEs with higher-order derivatives and systems of ODEs present a straightforward generalization of the present discussion, as will be shown in due course. Note also that we refer to the independent variable in this chapter as time, t, but this is done without loss of generality and other interpretations of the independent variable are also possible.

The ODE given above may be "solved" numerically by marching it forward in time, step by step. In other words, we seek to approximate the solution y to (8.1) at timestep $t_{n+1} = t_n + h_n$ given the solution at the initial time t_0 and the solution at the previously-computed timesteps t_1 to t_n. For simplicity of notation, we will focus our discussion initially on the case with constant stepsize h; generalization to the case with nonconstant h_n is straightforward.

8.1 Taylor-series methods

One of the simplest approaches to march the ODE (8.1) forward in time is to appeal to a Taylor series expansion in time, such as

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \frac{h^3}{6}y'''(t_n) + \dots\,. \tag{8.2}$$

From our ODE, we have:

$$\begin{aligned}
y' &= \frac{dy}{dt} = f \\
y'' &= \frac{dy'}{dt} = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{dy}{dt} = f_t + f f_y \\
y''' &= \frac{dy''}{dt} = \frac{d}{dt}(f_t + ff_y) = \frac{\partial}{\partial t}(f_t + ff_y) + \frac{\partial}{\partial y}(f_t + ff_y)\frac{dy}{dt} = f_{tt} + f_t f_y + 2ff_{yt} + f_y^2 f + f^2 f_{yy},
\end{aligned}$$

etc. Denoting the numerical approximation of $y(t_n)$ as y_n, the time integration method based on the first two terms of (8.2) is given by

$$\boxed{y_{n+1} = y_n + hf(y_n, t_n).} \tag{8.3}$$

This is referred to as the **explicit Euler** method, and is the simplest of all time integration schemes. Note that this method neglects terms which are proportional to h^2, and thus is "second-order" accurate over a single time step. As with the problem of numerical quadrature, however, a more relevant measure is the accuracy achieved when marching the ODE over a given time interval $(t_0, t_0 + T)$ as the timesteps h are made smaller. In such a setting, we lose one in the order of accuracy (as in the quadrature problem discussed in §7) and thus, over a specified time interval (t_0, t_0+T), **explicit Euler is first-order accurate**.

We can also base a time integration scheme on the first three terms of (8.2):

$$y_{n+1} = y_n + hf(y_n, t_n) + \frac{h^2}{2}[f_t(y_n, t_n) + f(y_n, t_n)f_y(y_n, t_n)].$$

Even higher-order Taylor series methods are also possible. We do not pursue such high-order Taylor series approaches in the present text, however, as their computational expense is relatively high (due to all of the cross derivatives required) and their stability and accuracy is not as good as some of the other methods which we will develop.

Note that a Taylor series expansion in time may also be written around t_{n+1}:

$$y(t_n) = y(t_{n+1}) - hy'(t_{n+1}) + \frac{h^2}{2}y''(t_{n+1}) - \frac{h^3}{6}y'''(t_{n+1}) + \dots$$

The time integration method based on the first two terms of this Taylor series is given by

$$\boxed{y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}).} \tag{8.4}$$

This is referred to as the **implicit Euler** method. It also neglects terms which are proportional to h^2, and thus is "second-order" accurate over a single time step. As with explicit Euler, over a specified time interval $(t_0, t_0 + T)$, **implicit Euler is first-order accurate**.

If f is nonlinear in y, implicit methods such as the implicit Euler method given above are difficult to use, because knowledge of y_{n+1} is needed (before it is computed!) to compute f in order to advance from y_n to y_{n+1}. Typically, such problems are approximated by some type of linearization or iteration, as will be discussed further in class. On the other hand, if f is linear in y, implicit strategies such as (8.4) are easily solved for y_{n+1}.

8.2 The trapezoidal method

The formal solution of the ODE (8.1) over the interval $[t_n, t_{n+1}]$ is given by

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(y, t)dt.$$

Approximating this integral with the trapezoidal rule from §7.1.1 gives

$$\boxed{y_{n+1} = y_n + \frac{h}{2}[f(y_n, t_n) + f(y_{n+1}, t_{n+1})].} \tag{8.5}$$

This is referred to as the **trapezoidal** or **Crank-Nicholson** method. We defer discussion of the accuracy of this method to §8.4, after we discuss first an illustrative model problem.

8.3 A model problem

A scalar model problem which is very useful for characterizing various time integration strategies is

$$y' = \lambda y \quad \text{with} \quad y(t_0) = y_0, \tag{8.6}$$

where λ is, in general, allowed to be complex. The exact solution of this problem is $y = y_0 e^{\lambda(t-t_0)}$. The utility of this model problem is that the exact solution is available, so we can compare the numerical approximation using a particular numerical method to the exact solution in order to quantify the pros and cons of the numerical method. The insight we gain by studying the application of the numerical method we choose to this simple model problem allows us to predict how this method will work on more difficult problems for which the exact solution is not available.

Note that, for $\Re(\lambda) > 0$, the magnitude of the exact solution grows without bound. We thus refer to the exact solution as being **unstable** if $\Re(\lambda) > 0$ and **stable** if $\Re(\lambda) \leq 0$. Graphically, we denote the region of stability of the exact solution in the complex plane λ by the shaded region shown in Figure 8.1.

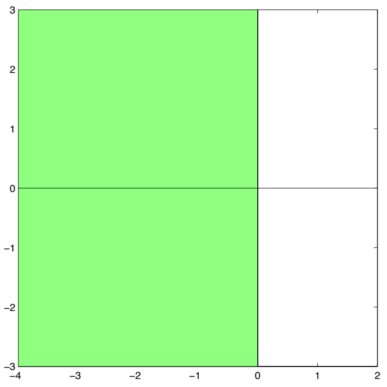


Figure 8.1: Stability of the exact solution to the model problem $y' = \lambda y$ in the complex plane λ.

8.3.1 Simulation of an exponentially-decaying system

Consider now the model problem (8.6) with $\lambda = -1$. The exact solution of this system is simply a decaying exponential. In Figure 8.2, we show the application of the explicit Euler method, the implicit Euler method, and the trapezoidal method to this problem. Note that the explicit Euler method appear to be unstable for the large values of h. Note also that all three methods are more accurate as h is refined, with the trapezoidal method appearing to be the most accurate.

8.3.2 Simulation of an undamped oscillating system

Consider now the second-order ODE for a simple mass/spring system given by

$$y'' = -\omega^2 y \quad \text{with} \quad y(t_0) = y_0, \;\; y'(t_0) = 0, \tag{8.7}$$

where $\omega = 1$. The exact solution is $y = y_0 \cos[\omega(t - t_0)] = (y_0/2)[e^{i\omega(t-t_0)} + e^{-i\omega(t-t_0)}]$.

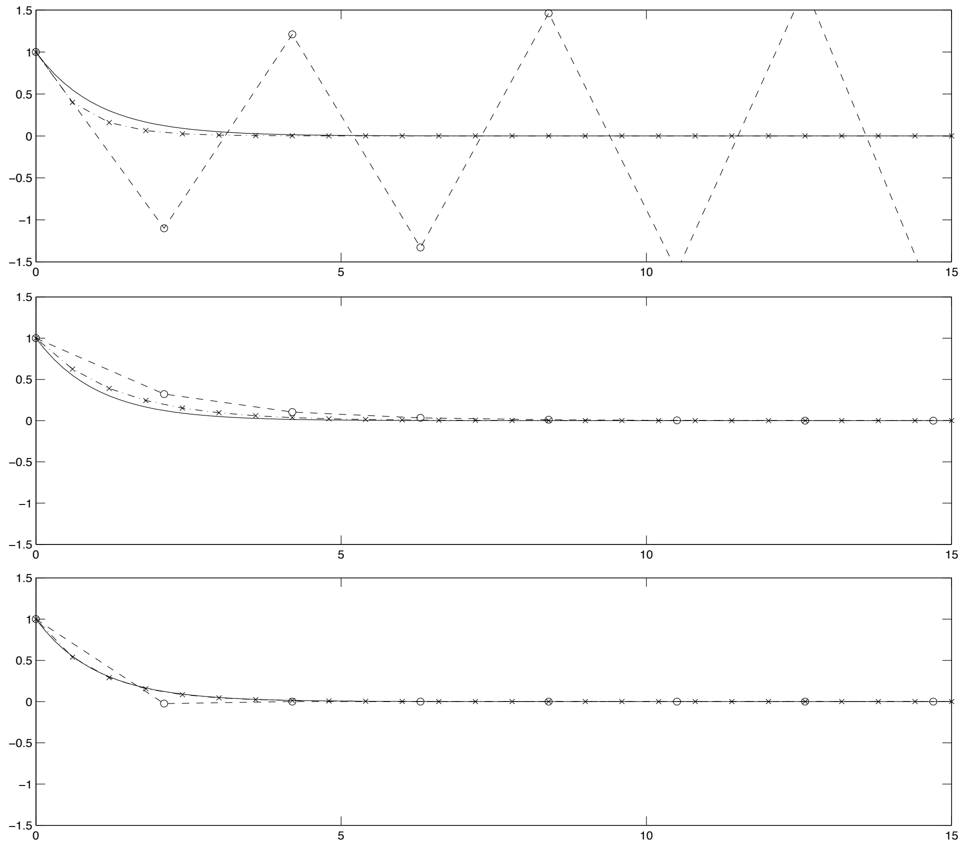


Figure 8.2: Simulation of the model problem $y' = \lambda y$ with $\lambda = -1$ using the explicit Euler method (top), the implicit Euler method (middle), and the trapezoidal method (bottom). Symbols denote: $\circ$, $h = 2.1$; $\times$, $h = 0.6$; ——— , exact solution.

We may easily write this second-order ODE as a first-order system of ODEs by defining $y_1 = y$ and $y_2 = y'$ and writing:

$$\underbrace{\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}'}_{\mathbf{y}'} = \underbrace{\begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}}_{\mathbf{y}}. \tag{8.8}$$

The eigenvalues of A are $\pm i\omega$. Note that the eigenvalues are imaginary; if we has started with the equation for a damped oscillator, the eigenvalues would have a negative real part as well. Note also that A may be diagonalized by its matrix of eigenvectors:

$$A = S\Lambda S^{-1} \quad \text{where} \quad \Lambda = \begin{pmatrix} i\omega & 0 \\ 0 & -i\omega \end{pmatrix}.$$

Thus, we have

$$\mathbf{y}' = S\Lambda S^{-1}\mathbf{y} \quad \Rightarrow \quad S^{-1}\mathbf{y}' = \Lambda S^{-1}\mathbf{y} \quad \Rightarrow \quad \mathbf{z}' = \Lambda \mathbf{z},$$

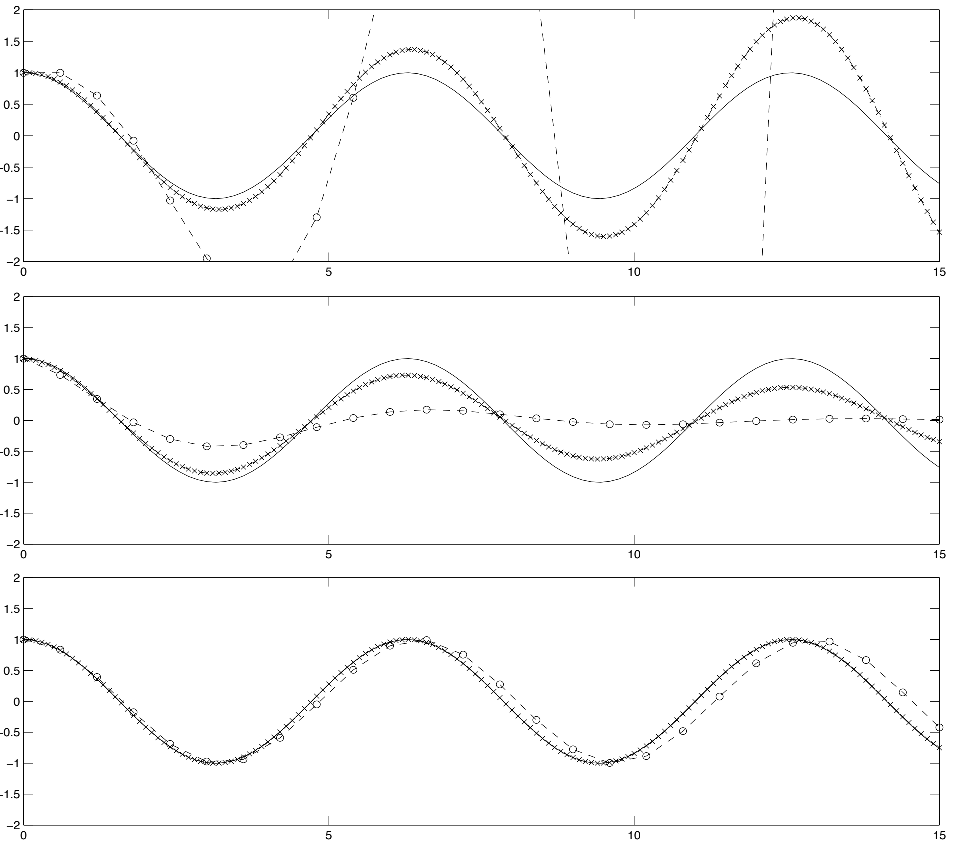


Figure 8.3: Simulation of the oscillatory system $y'' = -\omega^2 y$ with $\omega = 1$ using the explicit Euler method (top), the implicit Euler method (middle), and the trapezoidal method (bottom). Symbols denote: $\circ$, $h = 0.6$; $\times$, $h = 0.1$; ——— , exact solution.

where we have defined $\mathbf{z} = S^{-1}\mathbf{y}$. In terms of the components of $\mathbf{z}$, we have decoupled the dynamics of the system:

$$
\begin{aligned}
z_1' &= i\omega z_1 \\
z_2' &= -i\omega z_2.
\end{aligned}
$$

Each of these systems is exactly the same form as our scalar model problem (8.6) with complex (in this case, pure imaginary) values for λ. Thus, eigenmode decompositions of physical systems (like mass/spring systems) motivate us to look at the scalar model problem (8.6) over the complex plane λ. In fact, our original second-order system (8.7), as re-expressed in (8.8), will be stable iff there are no eigenvalues of A with $\Re(\lambda) > 0$.

In Figure 8.3, we show the application of the explicit Euler method, the implicit Euler method, and the trapezoidal method to the first-order system of equations (8.8). Note that the explicit Euler method appears to be unstable for both large and small values of h. Note also that all three methods are more accurate as h is refined, with the trapezoidal method appearing to be the most accurate.

We see that some numerical methods for time integration of ODEs are more accurate than others, and some numerical techniques are sometimes unstable, even for ODEs with stable exact solutions.

In the next two sections, we develop techniques to quantify both the stability and the accuracy of numerical methods for time integration of ODEs by application of these numerical methods to the model problem (8.6).

8.4 Stability

For stability of a numerical method for time integration of an ODE, we want to insure that, if the exact solution is bounded, the numerical solution is also bounded. We often need to restrict the timestep h in order to insure this. To make this discussion concrete, consider a system whose exact solution is bounded and define:
1) a stable numerical scheme: one which does not blow up for any h,
2) an unstable numerical scheme: one which blows up for any h, and
3) a conditionally stable numerical scheme: one which blows up for some h.

8.4.1 Stability of the explicit Euler method

Applying the explicit Euler method (8.3) to the model problem (8.6), we see that

$$y_{n+1} = y_n + \lambda h y_n = (1 + \lambda h) y_n.$$

Thus, assuming constant h, the solution at time step n is:

$$y_n = (1 + \lambda h)^n y_0 \triangleq \sigma^n y_0 \qquad \Rightarrow \qquad \sigma = 1 + \lambda h.$$

For large n, the numerical solution remains stable iff

$$|\sigma| \leq 1 \qquad \Rightarrow \qquad (1 + \lambda_R h)^2 + (\lambda_I h)^2 \leq 1.$$

The region of the complex plane which satisfies this stability constraint is shown in Figure 8.4. Note that this region of stability in the complex plane λh is consistent with the numerical simulations shown in Figure 8.2a and 8.3a: for real, negative λ, this numerical method is conditionally stable (*i.e.*, it is stable for sufficiently small h), whereas for pure imaginary λ, this numerical method is unstable for any h, though the instability is mild for small h.

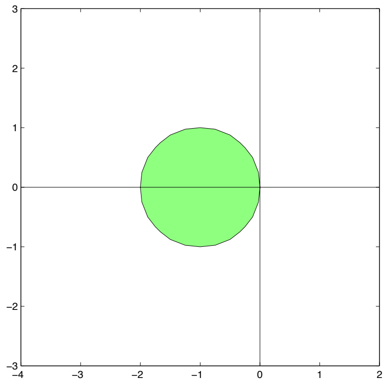


Figure 8.4: Stability of the numerical solution to $y' = \lambda y$ in the complex plane λh using the explict Euler method.

8.4.2 Stability of the implicit Euler method

Applying the implicit Euler method (8.4) to the model problem (8.6), we see that

$$y_{n+1} = y_n + \lambda h y_{n+1} \qquad \Rightarrow \qquad y_{n+1} = (1-\lambda h)^{-1} y_n.$$

Thus, assuming constant h, the solution at time step n is:

$$y_n = \left(\frac{1}{1-\lambda h}\right)^n y_0 \triangleq \sigma^n y_0 \qquad \Rightarrow \qquad \sigma = \frac{1}{1-\lambda h}.$$

For large n, the numerical solution remains stable iff

$$|\sigma| \le 1 \qquad \Rightarrow \qquad (1-\lambda_R h)^2 + (\lambda_I h)^2 \ge 1.$$

The region of the complex plane which satisfies this stability constraint is shown in Figure 8.5. Note that this region of stability in the complex plane λh is consistent with the numerical simulations shown in Figure 8.2b and 8.3b: this method is stable for any stable ODE for any h, and is even stable for some cases in which the ODE itself is unstable.

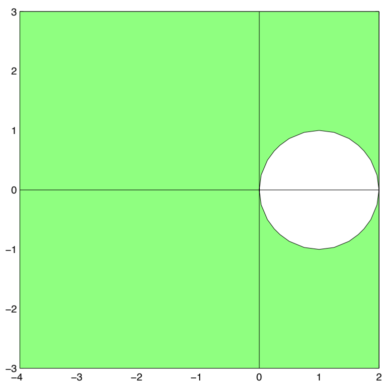


Figure 8.5: Stability of the numerical solution to $y' = \lambda y$ in the complex plane λh using the implicit Euler method.

8.4.3 Stability of the trapezoidal method

Applying the trapezoidal method (8.5) to the model problem (8.6), we see that

$$y_{n+1} = y_n + \frac{\lambda h}{2}(y_n + y_{n+1}) \qquad \Rightarrow \qquad y_{n+1} = \left(\frac{1+\frac{\lambda h}{2}}{1-\frac{\lambda h}{2}}\right) y_n.$$

Thus, assuming constant h, the solution at time step n is:

$$y_n = \left(\frac{1+\frac{\lambda h}{2}}{1-\frac{\lambda h}{2}}\right)^n y_0 \triangleq \sigma^n y_0 \qquad \Rightarrow \qquad \sigma = \frac{1+\frac{\lambda h}{2}}{1-\frac{\lambda h}{2}}.$$

For large n, the numerical solution remains stable iff

$$|\sigma| \le 1 \qquad \Rightarrow \quad \dots \quad \Rightarrow \qquad \Re(\lambda h) \le 0.$$

The region of the complex plane which satisfies this stability constraint coincides exactly with the region of stability of the exact solution, as shown in Figure 8.6. Note that this region of stability in the complex plane λh is consistent with the numerical simulations shown in Figure 8.2c and 8.3c, which are stable for systems with $\Re(\lambda) < 0$ and marginally stable for systems with $\Re(\lambda) = 0$.

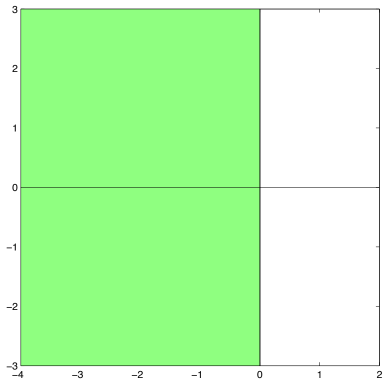


Figure 8.6: Stability of the numerical solution to $y' = \lambda y$ in the complex plane λh using the trapezoidal method.

8.5 Accuracy

Revisiting the model problem $y' = \lambda y$, the exact solution (assuming $t_0 = 0$ and $h =$ constant) is

$$y(t_n) = e^{\lambda t_n} y_0 = (e^{\lambda h})^n y_0 = \left(1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \dots\right)^n y_0.$$

On the other hand, solving the model problem with explicit Euler led to

$$y_n = (1 + \lambda h)^n y_0 \triangleq \sigma^n y_0,$$

solving the model problem with implicit Euler led to

$$y_n = \left(\frac{1}{1 - \lambda h}\right)^n y_0 = \left(1 + \lambda h + \lambda^2 h^2 + \lambda^3 h^3 + \dots\right)^n y_0 \triangleq \sigma^n y_0,$$

and solving the model problem with trapezoidal led to

$$y_n = \left(\frac{1 + \frac{\lambda h}{2}}{1 - \frac{\lambda h}{2}}\right)^n y_0 = \left(1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{4} + \dots\right)^n y_0 \triangleq \sigma^n y_0.$$

To quantify the accuracy of these three methods, we can compare the amplification factor σ in each of the numerical approximations to the exact value $e^{\lambda h}$. The leading order error of the explicit Euler and implicit Euler methods are seen to be proportional to h^2, as noted in §8.1, and the leading order error of the trapezoidal method is proportional to h^3. Thus, over a specified time interval $(t_0, t_0 + T)$, **explicit Euler and implicit Euler are first-order accurate** and **trapezoidal is second-order accurate**. The higher order of accuracy of the trapezoidal method implies an improved rate of convergence of this scheme to the exact solution as the timestep h is refined, as observed in Figures 8.2 and 8.3.

8.6 Runge-Kutta methods

An important class of explicit methods, called Runge-Kutta methods, is given by the general form:

$$\begin{aligned}
k_1 &= f\Big(y_n, t_n\Big)\\
k_2 &= f\Big(y_n + \beta_1\, h\, k_1, t_n + \alpha_1\, h\Big)\\
k_3 &= f\Big(y_n + \beta_2\, h\, k_1 + \beta_3\, h\, k_2, t_n + \alpha_2\, h\Big)\\
&\vdots\\
y_{n+1} &= y_n + \gamma_1\, h\, k_1 + \gamma_2\, h\, k_2 + \gamma_3\, h\, k_3 + \dots,
\end{aligned} \tag{8.9}$$

where the constants α_i, β_i, and γ_i are selected to match as many terms as possible of the exact solution:

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \frac{h^3}{6}y'''(t_n) + \dots$$

where

$$\begin{aligned}
y' &= f\\
y'' &= f_t + f f_y\\
y''' &= f_{tt} + f_t f_y + 2 f f_{yt} + f_y^2 f + f^2 f_{yy},
\end{aligned}$$

etc. Runge-Kutta methods are explicit and "self starting", as they don't require any information about the numerical approximation of the solution before time t_n; this typically makes them quite easy to use. As the number of intermediate steps k_i in the Runge-Kutta method is increased, the order of accuracy of the method can also be increased. The stability properties of higher-order Runge-Kutta methods are also generally quite favorable, as will be shown.

8.6.1 The class of second-order Runge-Kutta methods (RK2)

Consider first the family of two-step schemes of the form (8.9):

$$\begin{aligned}
k_1 &= f(y_n, t_n),\\
k_2 &= f(y_n + \beta_1\, h\, k_1, t_n + \alpha_1\, h)\\
&\approx f(y_n, t_n) + f_y(y_n, t_n)\Big(\beta_1\, h\, f(y_n, t_n)\Big) + f_t(y_n, t_n)\Big(\alpha_1\, h\Big),\\
y_{n+1} &= y_n + \gamma_1\, h\, k_1 + \gamma_2\, h\, k_2\\
&\approx y_n + \gamma_1\, h\, f(y_n, t_n) + \gamma_2\, h\Big(f(y_n, t_n) + \beta_1\, h\, f_y(y_n, t_n)\, f(y_n, t_n) + \alpha_1\, h\, f_t(y_n, t_n)\Big)\\
&\approx y_n + (\gamma_1 + \gamma_2)\, h\, f(y_n, t_n) + \gamma_2\, h^2\, \beta_1\, f_y(y_n, t_n)\, f(y_n, t_n) + \gamma_2\, h^2\, \alpha_1\, f_t(y_n, t_n).
\end{aligned}$$

Note that the approximations given above are exact if f is linear in y and t, as it is in our model problem. The exact solution we seek to match with this scheme is given by

$$y(t_{n+1}) = y(t_n) + hf(y_n, t_n) + \frac{h^2}{2}\Big(f_t(y_n, t_n) + f(y_n, t_n) f_y(y_n, t_n)\Big) + \dots$$

Matching coefficients to as high an order as possible, we require that

$$\left.\begin{aligned} \gamma_1 + \gamma_2 &= 1 \\ \gamma_2\, h^2\, \beta_1 &= \frac{h^2}{2} \\ \gamma_2\, h^2\, \alpha_1 &= \frac{h^2}{2} \end{aligned}\right\} \quad \Rightarrow \quad \alpha_1 = \beta_1, \quad \gamma_2 = \frac{1}{2\alpha_1}, \quad \gamma_1 = 1 - \frac{1}{2\alpha_1}.$$

Thus, the general form of the two-step second-order Runge-Kutta method (RK2) is

$$\begin{aligned} k_1 &= f\Big(y_n, t_n\Big) \\ k_2 &= f\Big(y_n + \alpha\, h\, k_1, t_n + \alpha\, h\Big) \\ y_{n+1} &= y_n + \Big(1 - \frac{1}{2\alpha}\Big) h\, k_1 + \Big(\frac{1}{2\alpha}\Big) h\, k_2, \end{aligned} \tag{8.10}$$

where α is a free parameter. A popular choice is $\alpha = 1/2$, which is known as the midpoint method and has a clear geometric interpretation of approximating a central difference formula in the integration of the ODE from t_n to t_{n+1}. Another popular choice is $\alpha = 1$, which is equivalent to perhaps the most common so-called "predictor-corrector" scheme, and may be computed in the following order:

$$\begin{aligned} \text{predictor}: \quad y^*_{n+1} &= y_n + hf(y_n, t_n) \\ \text{corrector}: \quad y_{n+1} &= y_n + \frac{h}{2}\Big[f(y_n, t_n) + f(y^*_{n+1}, t_{n+1})\Big]. \end{aligned}$$

The "predictor" (which is simply an explicit Euler estimate of y_{n+1}) is only "stepwise 2nd-order accurate". However, as we shown below, calculation of the "corrector" (which looks roughly like a recalculation of y_{n+1} with a trapezoidal rule) results in a value for y_{n+1} which is "stepwise 3rd-order accurate" (and thus the scheme is globally 2nd-order accurate).

Applying an RK2 method (for some value of the free parameter α) to the model problem $y' = \lambda y$ yields

$$\begin{aligned} y_{n+1} &= y_n + \Big(1 - \frac{1}{2\alpha}\Big) h\, \lambda y_n + \Big(\frac{1}{2\alpha}\Big) h\, \lambda (1 + \alpha\, h\, \lambda) y_n \\ &= \Big(1 + \lambda h + \frac{\lambda^2\, h^2}{2}\Big) y_n \triangleq \sigma y_n \quad \Rightarrow \quad \sigma = 1 + \lambda h + \frac{\lambda^2\, h^2}{2}. \end{aligned}$$

The amplification factor σ is seen to be a truncation of the Taylor series of the exact value $e^{\lambda h} = 1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \ldots$ We thus see that the leading order error of this method (for any value of α) is proportional to h^3 and, over a specified time interval $(t_0, t_0 + T)$, **an RK2 method is second-order accurate**. Over a large number of timesteps, the method is stable iff $|\sigma| \le 1$; the domain of stability of this method is illustrated in Figure 8.7.

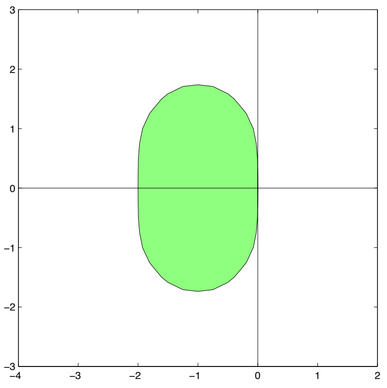


Figure 8.7: Stability of the numerical solution to $y' = \lambda y$ in the complex plane λh using RK2.

8.6.2 A popular fourth-order Runge-Kutta method (RK4)

The most popular fourth-order Runge-Kutta method is

$$
\begin{aligned}
k_1 &= f\Big(y_n, t_n\Big) \\
k_2 &= f\Big(y_n + \frac{h}{2}\,k_1, t_{n+1/2}\Big) \\
k_3 &= f\Big(y_n + \frac{h}{2}\,k_2, t_{n+1/2}\Big) \\
k_4 &= f\Big(y_n + h\,k_3, t_{n+1}\Big) \\
y_{n+1} &= y_n + \frac{h}{6}k_1 + \frac{h}{3}\Big(k_2 + k_3\Big) + \frac{h}{6}k_4
\end{aligned}
\tag{8.11}
$$

This scheme usually performs very well, and is the workhorse of many ODE solvers. This particular RK4 scheme also has a reasonably-clear geometric interpretation, as discussed further in class.

A derivation similar to that in the previous section confirms that the constants chosen in (8.11) indeed provide fourth-order accuracy, with the $\lambda - \sigma$ relationship again given by a truncated Taylor series of the exact value:

$$\sigma = 1 + \lambda h + \frac{\lambda^2\,h^2}{2} + \frac{\lambda^3\,h^3}{6} + \frac{\lambda^4\,h^4}{24}.$$

Over a large number of timesteps, the method is stable iff $|\sigma| \leq 1$; the domain of stability of this method is illustrated in Figure 8.8.

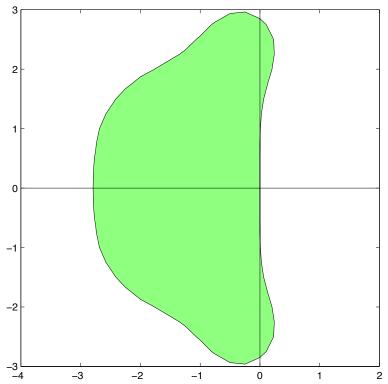


Figure 8.8: Stability of the numerical solution to $y' = \lambda y$ in the complex plane λh using RK4.

8.6.3 An adaptive Runge-Kutta method (RKM4)

Another popular fourth-order scheme, known as the Runge-Kutta-Merson method, is

$$\begin{aligned}
k_1 &= f\Big(y_n, t_n\Big) \\
k_2 &= f\Big(y_n + \frac{h}{3}\,k_1, t_{n+1/3}\Big) \\
k_3 &= f\Big(y_n + \frac{h}{6}(k_1 + k_2), t_{n+1/3}\Big) \\
k_4 &= f\Big(y_n + \frac{h}{8}(k_1 + 3k_3), t_{n+1/2}\Big) \\
y^*_{n+1} &= y_n + \frac{h}{2}k_1 - \frac{3h}{2}k_3 + 2hk_4 \\
k_5 &= f\Big(y^*_{n+1}, t_{n+1}\Big) \\
y_{n+1} &= y_n + \frac{h}{6}k_1 + \frac{2h}{3}k_3 + \frac{h}{6}k_5.
\end{aligned} \tag{8.12}$$

Note that one extra computation of f is required in this method as compared with the method given in (8.11). With the same sort of analysis as we did for RK2, it may be shown that both y^*_{n+1} and y_{n+1} are "stepwise 5th-order accurate", meaning that using either to advance in time over a given interval gives global 4th-order accuracy. In fact, if $\tilde{y}(t)$ is the exact solution to an ODE and y_n takes this exact value of $\tilde{y}(t_n)$ at $t = t_n$, then it follows after a bit of analysis that the errors in y^*_{n+1} and y_{n+1} are

$$y^*_{n+1} - \tilde{y}(t_{n+1}) = -\frac{h^5}{120}\tilde{y}^{(v)} + O(h^6) \tag{8.13}$$

$$y_{n+1} - \tilde{y}(t_{n+1}) = -\frac{h^5}{720}\tilde{y}^{(v)} + O(h^6). \tag{8.14}$$

Subtracting (8.13) from (8.14) gives

$$y_{n+1} - y^*_{n+1} = \frac{h^5}{144}\tilde{y}^{(v)} + O(h^6),$$

which may be substituted on the RHS of (8.14) to give

$$y_{n+1} - \tilde{y}(t_{n+1}) = -\frac{1}{5}(y_{n+1} - y^*_{n+1}) + O(h^6). \tag{8.15}$$

The quantity on the LHS of (8.15) is the error of our current "best guess" for y_{n+1}. The first term on the RHS is something we can compute, even if we don't know the exact solution $\tilde{y}(t)$. Thus, even if the exact solution $\tilde{y}(t)$ is unknown, we can still estimate the error of our best guess of y_{n+1} with quantities which we have computed. We may use this estimate to decide whether or not to refine or coarsen the stepsize h to attain a desired degree of accuracy on the entire interval. As with the procedure of adaptive quadrature, it is straightforward to determine whether or not the error on any particular step is small enough such that, when the entire (global) error is added up, it will be within a predefined acceptable level of tolerance.

8.6.4 A low-storage Runge-Kutta method (RKW3)

Amongst people doing very large simulations with specialized solvers, a third-order scheme which is rapidly gaining popularity, known as the Runge-Kutta-Wray method, is

$$\begin{aligned}
k_1 &= f\Big(y_n, t_n\Big) \\
k_2 &= f\Big(y_n + \beta_1\, h\, k_1, t_n + \alpha_1\, h\Big) \\
k_3 &= f\Big(y_n + \beta_2\, h\, k_1 + \beta_3\, h\, k_2, t_n + \alpha_2\, h\Big) \\
y_{n+1} &= y_n + \gamma_1\, h\, k_1 + \gamma_2\, h\, k_2 + \gamma_3\, h\, k_3,
\end{aligned} \tag{8.16}$$

where

$$\begin{aligned}
&\beta_1 = 8/15, \quad &&\beta_2 = 1/4, \quad &&\beta_3 = 5/12, \\
&\alpha_1 = 8/15, \quad &&\alpha_2 = 2/3, && \\
&\gamma_1 = 1/4, \quad &&\gamma_2 = 0, \quad &&\gamma_3 = 3/4.
\end{aligned}$$

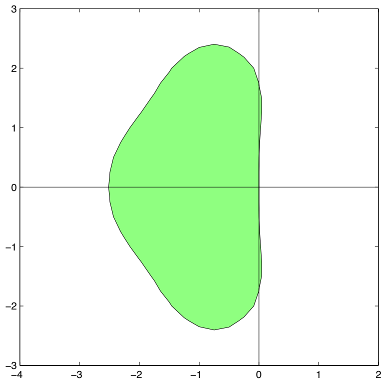


Figure 8.9: Stability of the numerical solution to $y' = \lambda y$ in the complex plane λh using third-order Runge-Kutta.

A derivation similar to that in §8.6.1 confirms that the constants chosen in (8.16) indeed provide third-order accuracy, with the $\lambda - \sigma$ relationship again given by a truncated Taylor series of the exact value:

$$\sigma = 1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6}.$$

Over a large number of timesteps, the method is stable iff $|\sigma| \le 1$; the domain of stability of this method is illustrated in Figure 8.9.

This scheme is particularly useful because it may be computed in the following order

$$\begin{aligned}
k_1 &= f\Big(y_n, t_n\Big) && \\
y^* &= y_n + \beta_1\, h\, k_1 && \text{(overwrite } y_n\text{)} \\[1em]
y^{**} &= y^* + \zeta_1\, h\, k_1 && \text{(overwrite } k_1\text{)} \\
k_2 &= f\Big(y^*, t_n + \alpha_1\, h\Big) && \text{(overwrite } y^*\text{)} \\
y^{**} &= y^{**} + \beta_3\, h\, k_2 && \text{(update } y^{**}\text{)} \\[1em]
y_{n+1} &= y^{**} + \zeta_2\, h\, k_2 && \text{(overwrite } k_2\text{)} \\
k_3 &= f\Big(y^{**}, t_n + \alpha_2\, h\Big) && \text{(overwrite } y^{**}\text{)} \\
y_{n+1} &= y_{n+1} + \gamma_3\, h\, k_3 && \text{(update } y_{n+1}\text{)},
\end{aligned}$$

with $\zeta_1 = -17/60$ and $\zeta_2 = -5/12$. Verification that these two forms of the RKW3 scheme are identical is easily obtained by substitution. The above ordering of the RKW3 scheme is useful when the dimension N of the vector y is huge. In the above scheme, we first compute the vector k_1 (also of dimension N) as we do, for example, for explicit Euler. However, *every operation that follows* either updates an existing vector or may overwrite the memory of an existing vector! (Pointers are very useful to set up multiple variables at the same memory location.) Thus, we only need enough room in the computer for *two* vectors of length N, not the four that one might expect are needed upon examination of (8.16). Amazingly, though this scheme is third-order accurate, it requires no more memory than explicit Euler. Note that one has to be very careful when computing an operation like $f(y^*)$ in place (*i.e.*, in a manner which immediately overwrites y^*) when the function f is a complicated nonlinear function.

Low-storage schemes such as this are essential in both computational mechanics (e.g., the finite element modeling of the stresses and temperatures of an engine block) and computational fluid dynamics (e.g., the finite-difference simulation of the turbulent flow in a jet engine). In these and many other problems of engineering interest, accurate discretization of the governing PDE necessitates big state vectors and efficient numerical methods. It is my hope that, in the past 10 weeks, you have begun to get a flavor for how such numerical techniques can be derived and implemented.

Appendix A

Getting started with Matlab

A.1 What is Matlab?

Matlab, short for Matrix Laboratory, is a high-level language excellent for, among many other things, linear algebra, data analysis, two and three dimensional graphics, and numerical computation of small-scale systems. In addition, extensive "toolboxes" are available which contain useful routines for a wide variety of disciplines, including control design, system identification, optimization, signal processing, and adaptive filters.

In short, Matlab is a very rapid way to solve problems which don't require intensive computations or to create plots of functions or data sets. Because a lot of problems encountered in the engineering world are fairly small, Matlab is an important tool for the engineer to have in his or her arsenal. Matlab is also useful as a simple programming language in which one can experiment (on small problems) with efficient numerical algorithms (designed for big problems) in a user-friendly environment. Problems which do require intensive computations, such as those often encountered in industry, are more efficiently solved in other languages, such as Fortran 90.

A.2 Where to find Matlab

For those who own a personal computer, the "student edition" of Matlab is available at the bookstore for just over what it costs to obtain the manual, which is included with the program itself. This is a bargain price (less than $100) for excellent software, and it is highly recommended that you get your own (legal) copy of it to take with you when you leave UCSD.[1]

For those who don't own a personal computer, or if you choose not to make this investment, you will find that Matlab is available on a wide variety of platforms around campus. The most stable versions are on various Unix machines around campus, including the ACS Unix boxes in the student labs (accessible by undergrads) and most of the Unix machines in the research groups of the MAE department. Unix machines running Matlab can be remotely accessed with any computer running X-windows on campus. There are also several Macs and PCs in the department (including those run by ACS) with Matlab already loaded.

[1]The student edition is actually an almost full-fledged version of the latest release of the professional version of Matlab, limited only in its printing capabilities and the maximum matrix size allowed. Note that the use of m-files (described at the end of this chapter) allows you to rerun finished Matlab codes at high resolution on the larger Unix machines to get report-quality printouts, if so desired, after debugging your code on your PC.

A.3 How to start Matlab

Running Matlab is the same as running any software on the respective platform. For example, on a Mac or PC, just find the Matlab icon and double click on it.

Instructions on how to open an account on one of the ACS Unix machines will be discussed in class. To run Matlab once logged in to one of these machines, just type `matlab`. The PATH variable should be set correctly so it will start on the first try—if it doesn't, that machine probably doesn't have Matlab. If running Matlab remotely, the DISPLAY variable must be set to the local computer before starting Matlab for the subsequent plots to appear on your screen. For example, if you are sitting at a machine named turbulence, you need to:

A) log in to turbulence,

B) open a window and telnet to an ACS machine on which you have an account

```
telnet iacs5
```

C) once logged in, set the DISPLAY environmental variable, with

```
setenv DISPLAY turbulence:0.0
```

D) run Matlab, with

```
matlab
```

If you purchase the student edition, of course, full instructions on how to install the program will be included in the manual.

A.4 How to run Matlab—the basics

Now that you found Matlab and got it running, you are probably on a roll and will stop reading the manual and this handout (you are, after all, an engineer!) The following section is designed to give you just enough knowledge of the language to be dangerous. From there, you can:

A) read the manual, available with the student edition or on its own at the bookstore, or

B) use the online help, by typing `help` <*command name*>, for a fairly thorough description of how any particular command works. Don't be timid to use this approach, as the online help in Matlab is very extensive.

To begin with, Matlab can function as an ordinary (but expensive!) calculator. At the `>>` prompt, try typing

```
>> 1+1
```

Matlab will reassure you that the universe is still in good order

```
ans =
     2
```

To enter a matrix, type

```
>> A = [1 2 3; 4 5 6; 7 8 0]
```

Matlab responds with

```
A =
     1     2     3
     4     5     6
     7     8     0
```

By default, Matlab operates in an echo mode; that is, the elements of a matrix or vector will be printed out as it is created. This may become tedious for large operations—to suppress it, type a semicolon after entering commands, such as:

```
>> A = [1 2 3; 4 5 6; 7 8 0];
```

Matrix elements are separated by spaces or commas, and a semicolon indicates the end of a row. Three periods in a row means that the present command is continued on the following line, as in:

```
>> A = [1 2 3;    ...
        4 5 6;    ...
        7 8 0];
```

When typing in a long expression, this can greatly improve the legibility of what you just typed. Elements of a matrix can also be arithmetic expressions, such as `3*pi`, `5*sqrt(3)`, etc. A column vector may be constructed with

```
>> y = [7 8 9]'
```

resulting in

```
y =
     7
     8
     9
```

To automatically build a row vector, a statement such as

```
>> z = [0:2:10]
```

results in

```
z =
     0     2     4     6     8    10
```

Vector `y` can be premultiplied by matrix `A` and the result stored in vector `z` with

```
>> z = A*y
```

Multiplication of a vector by a scalar may be accomplished with

```
>> z = 3.0*y
```

Matrix A may be transposed as

```
>> C = A'
```

The inverse of a matrix is obtained by typing

```
>> D = inv(A)
```

A 5×5 identity matrix may be constructed with

```
>> E = eye(5)
```

Tridiagonal matrices may be constructed by the following command and variations thereof:

```
>> 1*diag(ones(m-1,1),-1) - 4*diag(ones(m,1),0) + 1*diag(ones(m-1,1),1)
```

There are two "matrix division" symbols in Matlab, `\` and `/` — if `A` is a nonsingular square matrix, then `A\B` and `B/A` correspond formally to left and right multiplication of `B` (which must be of the appropriate size) by the inverse of `A`, that is `inv(A)*B` and `B*inv(A)`, but the result is obtained directly (via Gaussian elimination with full pivoting) without the computation of the inverse (which is a very expensive computation). Thus, to solve a system `A*x=b` for the unknown vector `x`, type

```
>> A=[1 2 3; 4 5 6; 7 8 0];
>> b=[5 8 -7]';
>> x=A\b
```

which results in

```
x =
   -1
    0
    2
```

To check this result, just type

```
>> A*x
```

which verifies that

```
ans =
    5
    8
   -7
```

Starting with the innermost group of operations nested in parentheses and working outward, the usual precedence rules are observed by Matlab. First, all the exponentials are calculated. Then, all the multiplications and divisions are calculated. Finally, all the additions and subtractions are calculated. In each of these three catagories, the calculation proceeds from left to right through the expression. Thus

```
>> 5/5*3
ans =
     3
```

and

```
>> 5/(5*3)
ans = 0.3333
```

It is best to make frequent use of parentheses to insure the order of operations is as you intend. Note that the matrix sizes must be correct for the requested operations to be defined or an error will result.

Suppose we have two vectors $\mathbf{x}$ and $\mathbf{y}$ and we wish to perform the componentwise operations:

$$z_\alpha = x_\alpha * y_\alpha \text{ for } \alpha = 1, n$$

The Matlab command to execute this is

```
>> x = [1:5]';
>> y = [6:10]';
>> z = x.*y
```

Note that `z = x*y` will result in an error, since this implies matrix multiplication and is undefined in this case. (A row vector times a column vector, however, is a well defined operation, so `z = x'*y` is successful. Try it!) The period distinguishes matrix operations (`* ^` and `/`) from component-wise operations (`.* .^` and `./`). One of the most common bugs when starting out with Matlab is defining and using row vectors where column vectors are in fact needed.

Matlab also has control flow statements, similar to many programming languages:

```
>> for i = 1:6, x(i) = i; end, x
```

creates,

```
x =
    1     2     3     4     5     6
```

Each `for` must be matched by an `end`. Note in this example that commas are used to include several commands on a single line, and the trailing `x` is used to write the final value of `x` to the screen. Note also that this `for` loop builds a row vector, not a column vector. An `if` statement may be used as follows:

```
>> n = 7;
>> if n > 0, j = 1, elseif n < 0, j = -1, else j = 0, end
```

The format of a `while` statement is similar to that of the `for`, but exits at the control of a logical condition:

```
>> m = 0;
>> while m < 7, m = m+2; end, m
```

Matlab has several advanced matrix functions which will become useful to you as your knowledge of linear algebra grows. These are summarized in the following section—don't be concerned if most of these functions are unfamiliar to you now. One command which we will encounter early in the quarter is LU decomposition:

```
>> A=[1 2 3; 4 5 6; 7 8 0];
>> [L,U,P]=lu(A)
```

This results in a lower triangular matrix `L`, an upper triangular matrix `U`, and a permutation matrix `P` such that `P*X = L*U`

```
L =
   1.0000         0         0
   0.1429    1.0000         0
   0.5714    0.5000    1.0000

U =
   7.0000    8.0000         0
        0    0.8571    3.0000
        0         0    4.5000

P =
    0     0     1
    1     0     0
    0     1     0
```

Checking this with

```
 >> P\L*U
```

confirms that the original matrix `A` is recovered.

A.5 Commands for matrix factoring and decomposition

The following commands are but a small subset of what Matlab has to offer:

```
>> R = chol(X)
```

produces an upper triangular `R` so that `R'*R = X`. If `X` is not positive definite, an error message is printed.

```
>> [V,D] = eig(X)
```

produces a diagonal matrix `D` of eigenvalues and a full matrix `V` whose columns are the corresponding eigenvectors so that `X*V = V*D`.

```
>> p = poly(A)
```

If `A` is an N by N matrix, `poly(A)` is a row vector with N+1 elements which are the coefficients of the characteristic polynomial, `det(lambda*eye(A) - A)`.
If `A` is a vector, `poly(A)` is a vector whose elements are the coefficients of the polynomial whose roots are the elements of `A`.

```
>> [P,H] = hess(A)
```

produces a unitary matrix `P` and a Hessenberg matrix `H` so that `A = P*H*P'` and `P'*P = eye(size(P))`.

```
>> [L,U,P] = lu(X)
```

produces a lower triangular matrix `L`, an upper triangular matrix `U`, and a permutation matrix `P` so that `P*X = L*U`.

```
>> Q = orth(A)
```

produces an orthonormal basis for the range of `A`. Note that `Q'*Q = I`, the columns of `Q` span the same space as the columns of `A` and the number of columns of `Q` is the rank of `A`.

```
>> [Q,R] = qr(X)
```

produces an upper triangular matrix `R` of the same dimension as `X` and a unitary matrix `Q` so that `X = Q*R`.

```
>> [U,S,V] = svd(X)
```

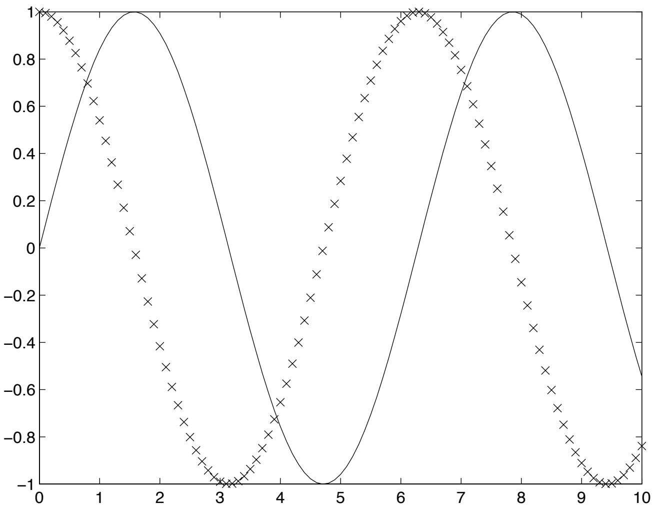
produces a diagonal matrix S, of the same dimension as `X`, with nonnegative diagonal elements in decreasing order, and unitary matrices `U` and `V` so that `X = U*S*V'`.

A.6 Commands used in plotting

For a two dimensional plot, try for example:

```
>> x=[0:.1:10];
>> y1=sin(x);
>> y2=cos(x);
>> plot(x,y1,'-',x,y2,'x')
```

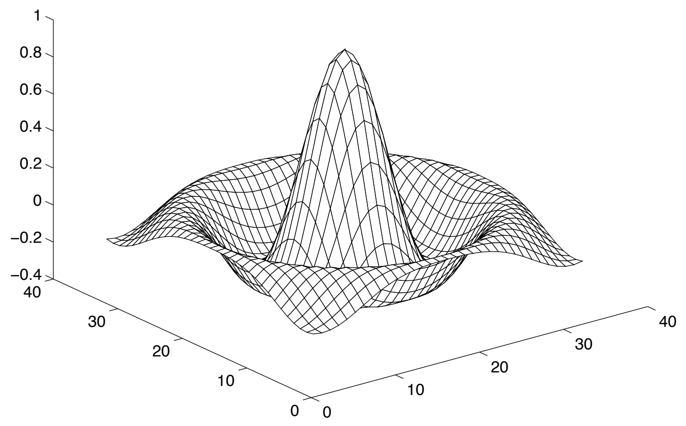
This results in the plot:



Three dimensional plots are also possible:

```
>> [x,y] = meshgrid(-8:.5:8,-8:.5:8);
>> R = sqrt(x.^2 + y.^2) + eps;
>> Z = sin(R)./R;
>> mesh(Z)
```

This results in the plot:



Axis rescaling and labelling can be controlled with `loglog`, `semilogx`, `semilogy`, `title`, `xlabel`, and `ylabel`—see the help pages for more information.

A.7 Other Matlab commands

Typing `who` lists all the variables created up to that point. Typing `whos` gives detailed information showing sizes of arrays and vectors.

In order to save the entire variable set computed in a session, type `save session` prior to quitting. All variables will be saved in a file named session.mat. At a later time the session may be resumed by typing `load session`. Save and load commands are menu-driven on the Mac and PC.

Some additional useful functions, which are for the most part self-explanatory (check the help page if not), are: `abs, conj, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exp, log, log10, eps`.

Matlab also has many special functions built in to aid in linear problem-solving; an index of these along with their usage is in the Matlab manual. Note that many of the 'built-in' functions are just prewritten m-files stored in subdirectories, and can easily be opened and accessed by the user for examination.

A.8 Hardcopies

Hardcopies of graphs on the Unix stations are best achieved using the `print` command. This creates postscript files which may then be sent directly to a postscript printer using the local print command (usually `lp` or `lpr`) at the Unix level. If using the `-deps` option of the `print` command in Matlab, then the encapulated postscript file (*e.g.*, `figure1.eps`) may be included in TEX documents as done here.

Hardcopies of graphs on a Mac may be obtained from the options available under the 'File' menu. On the student edition for the PC, the `prtscr` command may be used for a screen dump.

Text hardcopy on all platforms is best achieved by copying the text in the Matlab window and pasting it in to the editor of your choosing and then printing from there.

A.9 Matlab programming procedures: m-files

In addition to the interactive, or "Command" mode, you can execute a series of Matlab commands that are stored in 'm-files', which have a file extension '.m'. An example m-file is shown in the following section. When working on more complicated problems, it is a very good idea to work from m-files so that set-up commands don't need to be retyped repeatedly.

Any text editor may be used to generate m-files. It is important to note that these should be plain ASCI text files—type in the sequence of commands exactly as you would if running interactively. The `%` symbol is used in these files to indicate that the rest of that line is a comment—comment all m-files clearly, sufficiently, and succinctly, so that you can come back to the code later and understand how it works. To execute the commands in an m-file called sample.m, simply type `sample` at the Matlab `>>` prompt. Note that an m-file may call other m-files. Note that there are two types of m-files: scripts and functions. A script is just a list of matlab commands which run just as if you had typed them in one at a time (though it sounds pedestrian, this simple method is a useful and straightforward mode of operation in Matlab). A function is a set of commands that is "called" (as in a real programming language), only inheriting those variables in the argument list with which it is called and only returning the variables specified in the function declaration. These two types of m-files will be illustrated thoroughly by example as the course proceeds.

On Unix machines, Matlab should be started from the directory containing your m-files so that Matlab can find these files. On all platforms, I recommend making a new directory for each new problem you work on, to keep things organized and to keep from overwriting previously-written m-files. On the Mac and the PC, the search path must be updated using the `path` command after starting Matlab so that Matlab can find your m-files (see help page for details).

A.10 Sample m-file

```
% sample.m
echo on, clc
% This code uses MATLAB's eig program to find eigenvalues of
% a few random matrices which we will construct with special
% structure.
%
% press any key to begin
pause
R = rand(4)
eig(R)

% As R has random entries, it may have real or complex eigenvalues.
% press any key
pause
eig(R + R')

% Notice that R + R' is symmetric, with real eigenvalues.
% press any key
pause
eig(R - R')

% The matrix R - R' is skew-symmetric, with imaginary eigenvalues.
% press any key
pause
[V,D] = eig(R'*R)

% This matrix R'*R is symmetric, with real POSITIVE eigenvalues.
% press any key
pause
[V,D] = eig(R*R')

% R*R' has the same eigenvalues as R'*R. But different eigenvectors!
% press any key
pause
% You can create matrices with desired eigenvalues 1,2,3,4 from any
% invertible S times lambda times S inverse.

S = rand(4);
lambda = diag([1 2 3 4]);
A = S * lambda * inv(S)
eig(A)
echo off
% end sample.m
```