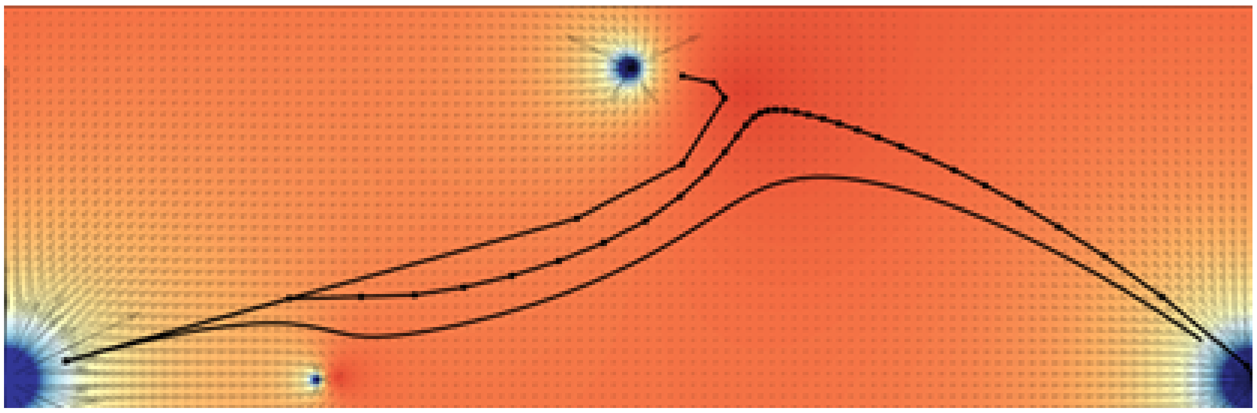
Numerical Methods in Practice

An Engineer's Guide to Numerical Computing Using C/C++ and Python

David Mayerich
University of Houston



"Mathematical science shows what is. It is the language of unseen relations between things. But to use and apply that language, we must be able fully to appreciate, to feel, to seize the unseen..."
-**Ada Lovelace** (1815-1852)

"The purpose of computing is insight, not numbers."
-**Richard Hamming** (1915-1998)

Last Updated: January 20, 2026

Contents

Chapter 1

Prerequisites: Mathematics

Since this textbook focuses on the fundamentals of numerical computation and therefore expects the reader to be familiar with several of the underlying mathematical concepts. An overview is given in this section to refresh the reader's memory and provide a picture of what will be discussed throughout the rest of the book.

In general, the background material amounts to college math classes up to and including calculus and differential equations. The reader is expected to understand the following general concepts:

- Complex numbers, including their relationships with trigonometric functions and basic arithmetic operations.
- Taylor series and how they can be used to approximate special functions.
- Linear algebra, including matrix multiplication and solving linear systems.
- Analytical methods for calculating derivatives and antiderivatives of functions, including the chain rule.
- Analytical methods for finding solutions to differential equations, including separation of variables.

1.1 Complex Variables

A complex number is a value that allows us to solve a certain subset of mathematical problems that would otherwise not have a solution. For example, consider the polynomial:

$$x^2 + 1 = 0 \tag{1.1}$$

No real value exists for x that can provide a solution to this equation. In practice, this equation does not have a solution. However, this equation forms the basis for a set of purely theoretical values that make several practical problems *much* simpler. While no real value for x can satisfy Equation 1.1, we can define a value i such that

$$i^2 = -1 \tag{1.2}$$

that would provide a mathematical solution. This value only exists on paper, and only because we precisely define its behavior based on Equation 1.2. Outside of this one case, i behaves like any other variable. For example, it can be combined with real numbers using any standard algebraic rule:

$$\begin{gathered} b(a+i) = ba + bi \\ (a+bi) + (c+di) = (a+c) + (b+d)i \end{gathered}$$

Any time real and imaginary numbers enter the same expression, that expression is said to be *complex*. When working with complex numbers, the only time the uniqueness of i comes into play using standard algebra is when it is multiplied by another factor of i:

$$\begin{gathered} bi(a+i) = abi + bi^2 = abi - b \\ (a+bi)(c+di) = ac + adi + bci + bdi^2 = ac - bd + (ad+bc)i \end{gathered}$$

While the existence of i seems to add a layer of complexity to theoretical mathematics, we will find that it actually makes many extremely complex equations much *much* simpler. This is because of the one fundamental principal: **the only rule we have to remember when working with this imaginary value is Equation 1.2.**

Defining i as $\sqrt{-1}$

It is possible that the reader has seen the imaginary value i expressed as $i = \sqrt{-1}$. I generally consider this definition misleading, because it implies that i can be used in some practical sense, or that it exists outside of the theoretical framework established in Equation 1.2.

The formulation of $i = \sqrt{-1}$ is useful in that it allows a more fundamental approach towards solving problems like:

$$x = \sqrt{-9}$$

In this case, the solution can be directly expressed as:

$$\begin{aligned} x &= \sqrt{9}\sqrt{-1} \\ &= 3i \end{aligned}$$

This is fine. However, it is important to note that this is simply a re-formulation of the definition that we have already established in Equation 1.2, and the solution can be readily calculated using Equation 1.2, albeit with more work:

$$\begin{aligned} x^2 &= -9 \\ &= 3^2(-1) \\ &= 3^2 i^2 = (3i)^2 \\ x &= 3i \end{aligned}$$

> In short, i **only has an impact when it is multiplied by itself.** Any other operations are simply a matter of record-keeping.

1.1.1 Converting Complex Numbers

In order to bridge the gap between the theoretical complex values and the real numbers used in practice, we often have to convert a complex value to a real one. Depending on the application, this will generally involve retrieving the real or imaginary part. We use the following notation to return the real or imaginary components of a complex value $z = a + bi$:

$$x = Re\{a + bi\} = a$$
$$y = Im\{a + bi\} = b$$

1.1.2 Complex Plane

We traditionally visualize real numbers using function plots that show the relationship between an independent variable (usually x) and a dependent variables (usually y). This relationship is generally expressed as some function of the independent variable, such as $x = f(x)$. However, functions that deal with complex variables are more difficult to visualize, because the real and imaginary components cannot be readily combined.

One option for visualizing complex functions is to plot the real and imaginary components in the same graph as two real functions:

$$y_1 = Re\{f(x)\}$$
$$y_2 = Im\{f(x)\}$$

Alternatively, complex values can be plotted in a two-dimensional *complex plane*, where both axes are dependent on the input value x and correspond to $Re\{f(x)\}$ and $Im\{f(x)\}$. One interesting aspect of the complex plane is that it helps us visualize another concept commonly encountered in complex analysis, which is the calculation of the absolute value of a complex variable $z = a + bi$:

$$|a + bi| = \sqrt{a^2 + b^2} \tag{1.3}$$

which corresponds to the distance of the value $z = a + bi$ from the origin in the complex plane. The absolute value (or magnitude) of a complex function is also a common way to visualize $f(x)$.

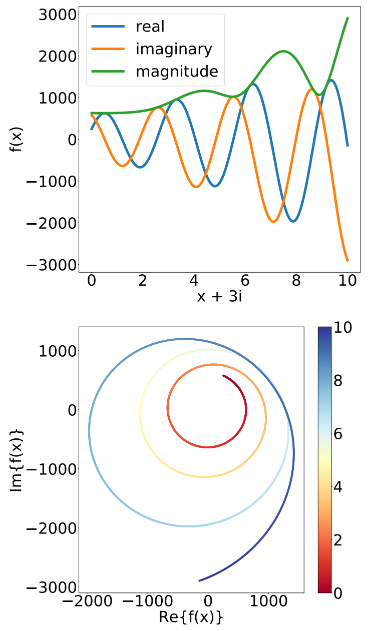


Figure 1.1 Plots of the function $f(x) = \sin(2x + 2)x - x^3$ on the interval $x = [0 + 3i, 10 + 3i]$. (top) Complex functions can be plotted independently using their real and imaginary components, or by calculating the absolute value. (bottom) The complex plane is also useful for displaying the relationship between the real and imaginary components.

1.1.3 Euler's Formula

The reason for using complex numbers in engineering primarily stems from their usefulness in representing signals. In particular, they can be used to algebraically represent trigonometric functions using Euler's formula:

$$e^{ix} = \cos x + i \sin x \tag{1.4}$$

This allows the representation of complex trigonometric systems using relatively simple algebra. For example, the following set of trigonometric operations:

$$x = -\sin a \sin c + \cos a \cos c - \sin a \sin d + \cos a \cos d \\ - \sin b \sin c + \cos b \cos c - \sin b \sin d + \cos b \cos d$$

can be dramatically simplified using complex variables:

$$x = Re\{(e^{ia} + e^{ib})(e^{ic} + e^{id})\}$$

1.1.4 Complex Conjugate

The final definition that we will cover is the concept of the *complex conjugate*, which is calculated by swapping the sign preceding the imaginary part of a complex variable:

$$z^* = (a + bi)^* = a - bi \tag{1.5}$$

While the complex conjugate is used extensively in signal processing, as well as finding roots of polynomials with complex variables.

1.2 Hilbert Spaces

A Hilbert space is the generalization of a two-dimensional or three-dimensional vector space to any number of dimensions. The two important concepts for the reader to understand are notational and mathematical. Several algorithms that we address in this book require working with linear systems and representing points and vectors in higher dimensional spaces. In addition, Hilbert spaces provide several mathematical notations that can significantly simplify how values are described later on.

1.2.1 Notation

A common notation will be adopted in this text to define how values are represented mathematically. The main goal is to allow the reader to quickly understand how defined values are represented and minimize any confusion. In addition, these concepts are widely used in mathematics and will simplify the any attempts to do further research.

Tensors. The first concept to address is that of a *tensor*, which is simply a generalization of a value. A single, vector, or matrix are all tensors of a different *order*. The order of a tensor specifies how many indices are required to access all elements.

A *scalar* value x is a 0th-order tensor. It is a single value that requires no indexing. Scalars form the basis of all higher-order tensors.

A first order tensor consists of multiple scalars that can be individually identified by a single index (ex. $[i]$). A *vector*

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

is a 1st-order tensor made up of individual scalar components x_i.

A *matrix* is a 2nd-order tensor. The matrix can be thought of as either:

- a set of individual scalar values accessed using two indices (ex. $[i, j]$)
- a set of vectors accessed using one index (ex. $[j]$), with each vector accessed using a single index (ex. $[i]$)

Both explanations are useful in the abstract. In general, tensors are easier to implement as individually-indexed n-dimensional arrays. However, all of the necessary mathematical definitions are applied recursively.

Variable Names. The following standards will be adopted for variable names in mathematical expressions:

- Scalar variables will be expressed as lower-case letters: x, a, b.
- Scalar constants will be expressed as upper-case letters: N, M.
- In general, tensors with non-zero order will be expressed in **bold**. More specifically:
  - Vectors will be expressed using lower-case letters: $\mathbf{x}$, $\mathbf{y}$.
  - Higher-order tensors (such as matrices) will be expressed using upper-case letters: $\mathbf{A}$, $\mathbf{M}$

Scalar variables will be expressed as lower-case letters, for example: x, a, b. TensoHigher-order tensors will generally be expressed in **bold**.

Number Sets. It is often convenient to limit the types of values that a particular scalar can represent. For example if we are representing an index, we generally don't have to consider fractional values and can therefore limit our representation to whole numbers. This can simplify both the mathematical expression of an algorithm, as well as its implementation. In this book, we will encounter scalar values that belong to the following sets:

- **real numbers** $\mathbb{R}$ **-** This is the most common set of numbers used in calculations. Take on any whole or fractional value
- **complex numbers** $\mathbb{C}$ **-** Complex values are composed of two real components, where one is a factor of the imaginary value $i^2 = -1$. For example, the complex value $z = a + bi$ is composed of the two real values a and b (Section 1.1).
- **integers** $\mathbb{Z}$ **-** Integers are real numbers that have no fractional component.
- **natural numbers** $\mathbb{N}$ **-** Natural numbers are positive integer values (including zero). These numbers are most frequently used as indices for tensors.

The most common number sets encountered in numerical methods are real numbers $\mathbb{R}$, natural numbers $\mathbb{N}$, and complex numbers $\mathbb{C}$. Real values are used in most calculations and returned by most functions. Natural numbers are most frequently used as indices for keeping track of series or identifying scalar components of tensors. Complex numbers are used in calculations involving complex variables, such as signal processing.

When specifying which set a variable belongs to, the following notation is used to denote membership:

$$x \in \mathbb{R} \quad z \in \mathbb{C} \quad i \in \mathbb{N}$$

indicating that the variable x is a real number (member of $\mathbb{R}$), the variable z is complex, and the variable i is a natural number.

Defining Tensors. When defining a non-scalar variable, have to know its order and size. We also have to know the type of value used to express its scalar components. *Note: tensors are generally represented using scalar values that are the same type.*

The size and shape of a tensor is specified by adding a superscript to a number set. For example, a vector composed of 3 real values is specified as:

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^3$$

whereas a 4×4 matrix of complex values can be specified as:

$$\mathbf{A} = \begin{bmatrix} a_{00} + b_{00}i & a_{01} + b_{01}i & a_{02} + b_{02}i & a_{03} + b_{03}i \\ a_{10} + b_{10}i & a_{11} + b_{11}i & a_{12} + b_{12}i & a_{13} + b_{13}i \\ a_{20} + b_{20}i & a_{21} + b_{21}i & a_{22} + b_{22}i & a_{23} + b_{23}i \\ a_{30} + b_{30}i & a_{31} + b_{31}i & a_{32} + b_{32}i & a_{33} + b_{33}i \end{bmatrix} \in \mathbb{C}^{4\times 4}$$

1.2.2 Operations

Hilbert spaces are particularly important in numerical methods because they provide a general extension of mathematics to higher-dimensional spaces. All major aspects of algebra and calculus can be extended to a Hilbert space. The reader will likely be familiar with many of these concepts from linear algebra. For example, the linear system

$$\mathbf{Ax} = \mathbf{b}$$

demonstrates the multiplication of an $N \times N$ matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ with $\mathbf{b} \in \mathbb{R}^N$ in order to produce the result $\mathbf{b} \in \mathbb{R}^N$. In particular, note that the dimensions of the variables have to match in order for the operation to be valid. For example, consider the following expression:

$$c\mathbf{Ax} = \mathbf{y}$$

where $c \in \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^N$, and $\mathbf{A} \in \mathbb{R}^{M\times N}$. The second dimension of $\mathbf{A}$ must equal the dimension of $\mathbf{x}$, and this produces the value $\mathbf{y} \in \mathbb{R}^M$. A scalar value (ex. c) can be multiplied by any tensor, and is applied to each scalar component.

While the division operation is not defined for tensors, the analogue is multiplication by the inverse:

$$c\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$$

where $\mathbf{A}^{-1} \in \mathbb{R}^{N\times M}$ if $\mathbf{A} \in \mathbb{R}^{M\times N}$. We will discuss methods for calculating matrix inverses in Chapter 8.

Addition and subtraction are element-wise operations requiring both tensors to be the same shape:

$$\begin{bmatrix} A_{00} & \dots & A_{0N} \\ A_{10} & \dots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{M0} & \dots & A_{MN} \end{bmatrix} + \begin{bmatrix} B_{00} & \dots & B_{0N} \\ B_{10} & \dots & B_{1N} \\ \vdots & \ddots & \vdots \\ B_{M0} & \dots & B_{MN} \end{bmatrix} = \begin{bmatrix} A_{00}+B_{00} & \dots & A_{0N}+B_{0N} \\ A_{10}+B_{10} & \dots & A_{1N}+B_{1N} \\ \vdots & \ddots & \vdots \\ A_{M0}+B_{M0} & \dots & A_{MN}+B_{MN} \end{bmatrix}$$

A common operation when dealing with linear systems is to calculate a matrix *transpose*:

$$\mathbf{A}^T\mathbf{x} + \mathbf{y} = \mathbf{z}$$

where $\mathbf{A}^T \in \mathbb{R}^{M\times N}$ is the same as $\mathbf{A}^T \in \mathbb{R}^{M\times N}$ with the rows and columns swapped. The scalars values in the transpose are therefore calculated as $A^T_{ij} = A_{ji}$. Note that the diagonal elements, where $i = j$, remain unchanged. Other common operations related to the transpose are the complex conjugate and the Hermetian transpose (Figure 1.2).

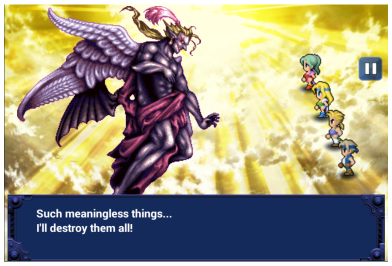


Figure 1.2 Hermetian Transpose.

Such meaningless things...
I'll destroy them all!

One type of multiplication is extremely common between two N-dimensional vectors in a Hilbert space: $\mathbf{x}^T\mathbf{y} = c$. This is generally referred to as the *inner product* or *dot product* and produces a scalar value. The term "dot product" comes from the dot operator used to simplify the notation:

$$\mathbf{x}\cdot\mathbf{y} = \mathbf{x}^T\mathbf{y} = \begin{bmatrix} x_0 & x_1 & \dots & x_N \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_N \end{bmatrix} = c$$

and has the interesting property:

$$\mathbf{x}\cdot\mathbf{y} = \|\mathbf{x}\|\,\|\mathbf{y}\|\cos\theta \tag{1.6}$$

where θ is the angle between the vectors in the corresponding N-dimensional space and $\|\mathbf{x}\|$ is the length of the vector as defined by the Euclidean norm:

$$\|\mathbf{x}\| = \sqrt{x_0^2 + x_1^2 + \cdots + x_N^2} \tag{1.7}$$

Note that the Euclidean norm can only be calculated for 1st-order tensors. We will discuss extensions of the norm to higher order tensors in Chapter 8.

Transposing the second vector produces the *outer product* or *tensor product*:

$$\mathbf{x}\otimes\mathbf{y} = \mathbf{x}\mathbf{y}^T = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_N \end{bmatrix} \begin{bmatrix} y_0 & y_1 & \dots & y_N \end{bmatrix} = \begin{bmatrix} x_0y_0 & x_0y_1 & \dots & x_0y_N \\ x_1y_0 & x_1y_1 & \dots & x_1y_N \\ \vdots & \ddots & \ddots & \vdots \\ x_Ny_0 & x_Ny_1 & \dots & x_Ny_N \end{bmatrix}$$

Covariance Matrix

The covariance matrix is commonly used in multi-variate analysis in order to understand how a vector of variables behave relative to each other. Calculation of the covariance matrix is important for several numerical applications, including principal component analysis (PCA). The definition also provides an example of the notations and operations discussed in this section.

Assume that we are given a set of N multi-variate measurements $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_N]$. These can be any signal, such as audio from a microphone or a row of pixels from a digital camera. We first compute the mean vector μ by averaging all of the measurements:

$$\mu = \frac{1}{N}\sum_{n=0}^{N-1} \mathbf{x}_n$$

The covariance matrix $\mathbf{C}$ is calculated using:

$$\mathbf{C} = \frac{1}{N}\sum_{n=0}^{N-1} (\mathbf{x}_n - \mu) \otimes (\mathbf{x}_n - \mu)$$

1.3 Solving Linear Systems

Linear systems are frequently encountered in science and engineering and generally take the form

$$\mathbf{Ax} = \mathbf{b}$$

where $\mathbf{A} \in \mathbb{R}^{N\times N}$ is a matrix of coefficients, $\mathbf{b} \in \mathbb{R}^N$, $\mathbf{b} \in \mathbb{R}^N$ is a vector of known values, and $\mathbf{x} \in \mathbb{R}^N$ is a matrix of unknown values.

The naive method for solving this system is to calculate the matrix inverse $\mathbf{A}^{-1}$ and apply it to solve for $\mathbf{x}$:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ \mathbf{A}^{-1}\mathbf{Ax} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \end{aligned}$$

The more common approach is to apply *Gaussian Elimination* (GE) followed by *backsubstitution* to iteratively solve for each value in $\mathbf{x}$.

1.3.1 Gaussian Elimination

Gaussian Elimination is the process of performing a series of elementary row operations on an augmented matrix to reduce it to row-echelon form. The augmented matrix $\bar{\mathbf{A}}$ is generated by appending the coefficient matrix to the right hand side vector $\mathbf{b}$:

$$\bar{\mathbf{A}} = \left[\begin{array}{cccc|c} A_{00} & A_{01} & \dots & A_{0N} & b_0 \\ A_{10} & A_{11} & \dots & A_{1N} & b_1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ A_{N0} & A_{N1} & \dots & A_{NN} & b_N \end{array}\right]$$

Operations allowed by GE are performed on rows of the augmented matrix. It is therefore easier to consider these operations by represented the augmented matrix as a set of row vectors:

$$\bar{\mathbf{A}} = \left[\begin{array}{c|c} \mathbf{a}_0^T & b_0 \\ \mathbf{a}_1^T & b_1 \\ \vdots & \vdots \\ \mathbf{a}_N^T & b_N \end{array}\right]$$

where

$$\mathbf{a}_n^T = \left[\begin{array}{cccc} A_{n0} & A_{n1} & \dots & A_{nN} \end{array}\right]$$

Gaussian elimination applies a series of row operations, generating a new augmented matrix $\hat{\mathbf{A}}_i$ after each step. The operations allowed by Gaussian elimination are:

1. Adding a scalar multiple of one row to another:

$$\bar{\mathbf{A}}_i = \left[\begin{array}{c|c} \mathbf{a}_0^T & b_0 \\ \mathbf{a}_1^T & b_1 \\ \vdots & \vdots \\ \mathbf{a}_N^T & b_N \end{array}\right] \quad \Rightarrow \quad \bar{\mathbf{A}}_{i+1} = \left[\begin{array}{c|c} \mathbf{a}_0^T & b_0 \\ \mathbf{a}_1^T + c\mathbf{a}_0 & b_1 + cb_0 \\ \vdots & \vdots \\ \mathbf{a}_N^T & b_N \end{array}\right]$$

2. Scaling a row by some factor c:

$$\bar{\mathbf{A}}_i = \left[\begin{array}{c|c} \mathbf{a}_0^T & b_0 \\ \mathbf{a}_1^T & b_1 \\ \vdots & \vdots \\ \mathbf{a}_N^T & b_N \end{array}\right] \quad \Rightarrow \quad \bar{\mathbf{A}}_i = \left[\begin{array}{c|c} \mathbf{a}_0^T & b_0 \\ c\mathbf{a}_1^T & cb_1 \\ \vdots & \vdots \\ \mathbf{a}_N^T & b_N \end{array}\right]$$

3. Swap two rows:

$$\bar{\mathbf{A}}_i = \left[\begin{array}{c|c} \mathbf{a}_0^T & b_0 \\ \mathbf{a}_1^T & b_1 \\ \vdots & \vdots \\ \mathbf{a}_N^T & b_N \end{array}\right] \quad \Rightarrow \quad \bar{\mathbf{A}}_i = \left[\begin{array}{c|c} \mathbf{a}_1^T & b_1 \\ \mathbf{a}_0^T & b_0 \\ \vdots & \vdots \\ \mathbf{a}_N^T & b_N \end{array}\right]$$

Row operations are applied iteratively until the final matrix is in row-echelon form, defined by:

- Any rows containing only zeros are at the bottom of the matrix.
- The first non-zero coefficient in any row is to the right of the first non-zero coefficient of the row above it.

If GE is used to solve a linear system, examination of the final row-echelon matrix will determine whether or not there is a unique solution. Specifically, the linear system can be solved if:

- The left portion of the augmented matrix is upper triangular.
- No row contains only zeros.

1.3.2 Backsubstitution

If the linear system can be solved, the final augmented matrix after M iterations of GE will have the form:

$$\bar{\mathbf{A}}_M = \left[\begin{array}{cccc|c} A_{00} & A_{01} & \dots & A_{0N} & b_0 \\ 0 & A_{11} & \dots & A_{1N} & b_1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & A_{NN} & b_N \end{array}\right]$$

Backsubstitution is the solution of each of the linear equations embedded in the matrix. The calculation starts with the last row:

$$\begin{aligned} A_{NN}x_N &= b_N \\ x_N &= \frac{b_N}{A_{NN}} \end{aligned}$$

which provides the solution for the last element in $\mathbf{x}$, x_N. That value is substituted into the next equation (moving up):

$$\begin{aligned} A_{N-1,N-1}x_{N-1} + A_{N-1,N}x_N = b_{N-1} \\ A_{N-1,N-1}x_{N-1} &= b_{N-1} - A_{N-1,N}x_N \\ x_{N-1} &= \frac{b_{N-1} - A_{N-1,N}x_N}{A_{N-1,N-1}} \end{aligned}$$

This process continues until all values of $\mathbf{x}$ are calculated.

1.3.3 Algorithms

In this subsection, we provide algorithms in pseudocode for both Gaussian elimination and backsubstitution. Implementation in C/C++ is left as an exercise for the user and may be particularly helpful for reviewing the material in Chapter 2.

Example 1.1

Solve the following system of linear equations:

$$\begin{aligned} 9x^3 + 3z &= 3 \\ 18x^3 + 2y^2 + z &= 7 \\ 27x^3 + y^2 + 4z &= 9 \end{aligned}$$

Solution: This linear system in matrix form is:

$$\begin{bmatrix} 9 & 0 & 3 \\ 18 & 2 & 1 \\ 27 & 1 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ 7 \\ 9 \end{bmatrix}$$

Algorithm 1 Gaussian elimination is an $O(n^3)$ algorithm for solving linear systems of equations by iteratively refining a matrix until it is in row-echelon form.

```
Gaussian Elimination
    input: $\mathbf{A} \in \mathbb{R}^{N\times N}$ and $\mathbf{b} \in \mathbb{R}^N$
    initialize: $i, j, k \in \mathbb{N}$ and $m \in \mathbb{R}$
    for $k = 0$ to $N-2$:
        for $i = k+1$ to $N-1$:
            $m = \frac{A_{ik}}{A_{kk}}$
            for $j = k$ to $N-1$:
                $A_{ij} = A_{ij} - mA_{kj}$
            $b_i = b_i - mb_k$
    output: A and b
```

Algorithm 2 After a matrix representing a linear system is in row-echelon form, backsubstitution is used to iteratively solve for each unknown variable.

```
Backsubstitution
    input: $\mathbf{A} \in \mathbb{R}^{N\times N}$ in row-echelon form and $\mathbf{b} \in \mathbb{R}^N$
    initialize: $i, j \in \mathbb{N}$ and $s \in \mathbb{R}$
    for $i = N-1$ to 0 (step $-1$):
        $s = b_i$
        for $j = i$ to $N-1$:
            $s = s - A_{ij}x_j$
        $x_i = \frac{s}{A_{ii}}$
    output: A and b
```

which can be combined into the augmented matrix:

$$\bar{\mathbf{A}}_0 = \left[\begin{array}{ccc|c} 8 & 0 & 3 & 4 \\ 16 & 2 & 1 & 7 \\ 24 & 1 & 4 & 9 \end{array}\right]$$

Applying the following operations reduce the matrix to row-echelon form:

$$\overset{\mathbf{R}_1 = \mathbf{R}_1 - 2\mathbf{R}_0}{\bar{\mathbf{A}}_1 = \left[\begin{array}{ccc|c} 8 & 0 & 3 & 4 \\ 0 & 2 & -5 & -1 \\ 24 & 1 & 4 & 9 \end{array}\right]} \quad \overset{\mathbf{R}_2 = \mathbf{R}_2 - 3\mathbf{R}_0}{\bar{\mathbf{A}}_2 = \left[\begin{array}{ccc|c} 9 & 0 & 3 & 3 \\ 0 & 2 & -5 & -1 \\ 0 & 1 & -5 & -3 \end{array}\right]} \quad \overset{\mathbf{R}_2 = \mathbf{R}_2 - \frac{1}{2}\mathbf{R}_1}{\bar{\mathbf{A}}_3 = \left[\begin{array}{ccc|c} 9 & 0 & 3 & 3 \\ 0 & 2 & -5 & -1 \\ 0 & 0 & -\frac{5}{2} & -\frac{5}{2} \end{array}\right]}$$

Converting out of matrix form, we now have a new set of linear equations that can be solved using back-substitution:

$$\begin{aligned}8x+3z&=4\\2y-5z&=-1\\-\frac{5}{2}z&=-\frac{5}{2}\end{aligned}$$

Solving for each value individually yields:

$$\begin{aligned}z&=-\frac{5}{2}\left(-\frac{2}{5}\right)\\&=\mathbf{1}\\2y-5(1)&=-1\\y&=\frac{-1+5}{2}\\&=\mathbf{2}\\8x+3(1)&=4\\x&=\frac{4-3}{8}\\&=\mathbf{\frac{1}{8}}\end{aligned}$$

1.4 Derivatives

1.4.1 Notation

In this text, we will most frequently express differentiation Leibniz's notation. The derivative of a function $f(x)$ with respect to a dependent variable x is expressed as:

$$\frac{df(x)}{dx} \tag{1.8}$$

Higher order derivatives are expressed as:

$$\frac{d^n f(x)}{dx^n} \tag{1.9}$$

for the nth derivative.

The advantage of this notation is that it explicitly identifies the dependent variable with respect to which the derivative is taken. This is particularly useful for higher-dimensional functions and vector calculations.

1.4.2 Polynomials

Polynomial differentiation is based on several fundamental rules:

1. **The derivative of any constant function $f(x) = c$ is zero.** Specifically, a function is considered constant if the variable with respect to which the derivative is taken is not expressed:

$$\frac{d}{dx}c=0$$

2. **The derivative operator is distributive:**

$$\frac{d}{dx}\left(g(x)+h(x)-k(x)\right)=\frac{d}{dx}g(x)+\frac{d}{dx}h(x)-\frac{d}{dx}k(x)$$
$$\frac{d}{dx}\left(af(x)+bg(x)\right)=a\left(\frac{d}{dx}f(x)\right)+b\left(\frac{d}{dx}g(x)\right)$$

3. **The derivative of a dependent variable in an exponential expression is defined as:**

$$\frac{d}{dx}x^n=nx^{n-1}$$

Example 1.2

Calculate the derivative of the polynomial

$$p(x)=3x^5+x^2+7$$

Solution: Based on the distributive property:

$$\frac{dp(x)}{dx}=3\frac{d}{dx}x^5+\frac{d}{dx}x^2+7\frac{d}{dx}$$

Since the derivative of a constant is zero, the last term:

$$7\frac{d}{dx}=0$$

Finally, solving the derivatives for the exponential expressions gives:

$$\begin{aligned}\frac{d}{dx}p(x)&=3\left(5x^{5-1}\right)+\left(2x^{2-1}\right)+0\\&=15x^4+2x\end{aligned}$$

1.4.3 Special Functions

Trigonometric functions have corresponding analytical derivatives. Derivatives of the forward trigonometric functions are:

$$\frac{d}{dx}\sin x=\cos x \qquad \frac{d}{dx}\cos x=-\sin x \qquad \frac{d}{dx}\tan x=\frac{1}{\cos^2 x}$$

Derivatives of the inverse trigonometric functions are:

$$\frac{d}{dx}\arcsin x=\frac{1}{\sqrt{1-x^2}} \qquad \frac{d}{dx}\arccos x=\frac{-1}{\sqrt{1-x^2}} \qquad \frac{d}{dx}\arctan x=\frac{1}{x^2+1}$$

Derivatives for other common functions include:

$$\frac{d}{dx}e^x=e^x \qquad \frac{d}{dx}\ln x=\frac{1}{x} \qquad \frac{d}{dx}\log_b x=\frac{1}{x\ln b}$$

Example 1.3

Calculate the derivative of the following function:

$$f(x) = \cos x + 7e^x - 6x^3 + \ln x$$

Solution: The distributive property of the differentiation operation allows the derivative to be expressed as:

$$\frac{d}{dx}f(x) = \frac{d}{dx}\cos x + 7\frac{d}{dx}e^x - 6\frac{d}{dx}x^3 + \frac{d}{dx}\ln x$$

Applying the necessary operations for each terms yields:

$$\frac{d}{dx}f(x) = -\sin x + 7e^x - 18x^2 + \frac{1}{x}$$

1.4.4 Product and Chain Rules

The *product rule* and *chain rule* are particularly powerful tools that allow us to differentiate most complex functions by breaking them down into simple parts. The fundamental rules for differentiation of polynomials and special functions can then be used to differentiate these individual parts.

The product rules is defined as:

$$\frac{d}{dx}\left(f(x)g(x)\right) = f(x)\frac{d}{dx}g(x) + g(x)\frac{d}{dx}f(x)$$

and often stated as **the *first* times the derivative of the *second* plus the *second* times the derivative of the *first*.**

The chain rule is defined as:

$$\frac{d}{dx}f\left(g(x)\right) = \frac{df}{dx}\left(g(x)\right)\frac{d}{dx}g(x)$$

and is often stated as **the derivative of the *outside* times the derivative of the *inside*.** The chain rule is actually more easily stated using prime notation for the derivative:

$$\frac{d}{dx}f\left(g(x)\right) = f'\left(g(x)\right)g'(x)$$

Example 1.4

Use the product and chain rules to calculate the derivative of:

$$f(x) = \cos(2x^3)x^4$$

Solution: Applying the chain and product rules yields:

$$\begin{aligned}\frac{d}{dx}f(x) &= \cos\left(2x^3\right)\frac{d}{dx}x^4 + x^4\frac{d}{dx}\left(\cos\left(2x^3\right)\right)\\ &= \cos\left(2x^3\right)\frac{d}{dx}x^4 + x^4\frac{d\cos}{dx}\left(2x^3\right)\frac{d}{dx}\left(2x^3\right)\\ &= 4\cos\left(2x^3\right)x^3 - 6x^6\sin\left(2x^3\right)\end{aligned}$$

1.5 Taylor Series

A Taylor series is a polynomial representation for a function composed of an infinite sum of terms based on the derivatives of the original function. Given a differentiable function $f(x)$, we can calculate a polynomial approximation $T(x)$ using the generating function:

$$T(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{1.10}$$

where $f^{(n)}(x)$ is the n-th derivative of $f(x)$ and a is a parameter for which we know the value of $f(a)$ and all of its derivatives. The denominator $n!$ in the generating function is the factorial of n, defined as:

$$n! = \prod_{k=1}^{n} k \tag{1.11}$$

The first practical feature of the Taylor series that should be understood by the reader is that successive terms of $T(x)$ become smaller. This is because the factorial term $n!$ increases very quickly with n. For large n, $n! \gg x^n$ for any constant value of x. This feature is important because it suggests that the Taylor series can approximate a function with a finite number of terms is some error is acceptable.

We can control the accuracy of the approximation by controlling the number of terms used to represent $T(x)$:

$$T_N(x) = \sum_{n=0}^{N} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{1.12}$$

such that N, generally referred to as the *order*, controls the number of terms in the polynomial. A 3rd order Taylor series approximation ($N = 3$) would take the form:

$$T_3(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f''(a)}{6}(x-a)^3$$

If we wish to approximate the function $f(x)$ using a Taylor series polynomial $T(x)$, we can select some acceptable absolute error ϵ such that $\left|f(x) - T(x)\right| \le \epsilon$. An Nth order Taylor polynomial $T_N(x)$ can then be calculated such that the order is sufficient to accommodate the error constraints. We will discuss methods for selecting the appropriate order and reducing the error in our approximations throughout this text.

1.5.1 Taylor Series Divergence

One important aspect of Taylor series approximations is that the approximating polynomial $T(x)$ only converges for values near the expansion point a, where $a \pm 1$. For values outside of this region, most Taylor series approximations diverge, resulting in greater error for higher orders of N.

We will discuss the reasons for this in later sections, however it is important to be generally aware that the ability for a Taylor series approximation to effectively represent a function is limited to the region around which $T(x)$ is expanded. **As $|x-a|$ becomes large, $T(x)$ becomes a bad approximation.**

1.5.2 Maclaurin Series

A Maclaurin series is a special case of the Taylor series where $a = 0$, and is specified by the simplified generating function:

$$M(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!}(x)^n \tag{1.13}$$

In numerical methods, we routinely encounter situations where algorithms behave poorly for small values of x. Since $M(x)$ provides a better approximation as $x \to 0$, the Maclaurin approximation is particularly useful.

1.6 Differential Equations

Analytical solutions for differential equations are significantly more difficult to find, and many problems in science and engineering do not have a known analytical solution. Since these problems form the basis of many physically-based models, numerical tools are in high demand and are a major focus in numerical computing. It is useful to have some understanding of analytical methods for solving differential equations.

1.6.1 Separation of Variables

The most common is *separation of variables* followed by calculation of the antiderivative. Given a simple differential equation:

$$\frac{dy}{dx} = y^2\left(x^2 + x\right)$$

we first separate the variables x and y:

$$\frac{1}{y^2}dy = \left(x^2 + x\right)dx$$

followed by calculation of the antiderivatives of both sides and simplification:

$$\begin{aligned}
\int \frac{1}{y^2}dy &= \int \left(x^2 + x\right)dx \\
-\frac{1}{y} + C_1 &= \frac{x^3}{3} + \frac{x^2}{2} + C_2 \\
\frac{1}{y} &= -\frac{2x^3 + 3x^2}{6} + C \\
y &= \frac{6}{3x^2 + 2x^3} + C
\end{aligned}$$

This provides an infinite number of solutions y, given any value for C. If one solution is desired, additional information is required to solve for C. This is usually provide as an *initial value problem* where $y(a) = b$, which can be used to determine the unknown constant:

$$b = \frac{6}{3a^2 - 2a^3} + C$$
$$C = b - \frac{6}{3a^2 - 2a^3}$$

which yields the final solution

$$y = \frac{6}{3x^2 - 2x^3} + b - \frac{6}{3a^2 - 2a^3}$$

Exponential Growth and Decay

One of the simplest differential equations describes the rate of change of a mathematical value over time, when the future value is proportional to the current value. This concept is extremely common in many fields, and can be used to describe a variety of observations including the reproduction of microorganisms, the rate of nuclear decay, and compound interest. This function can be described differentially as:

$$\frac{dy}{dt} = ay$$

where a is the growth rate. The solution can be calculated using separation of variables:

$$\frac{1}{y}dy = a\,dt \tag{1.14}$$
$$\int \frac{1}{y}dy = \int a\,dt \tag{1.15}$$
$$\ln y + C_1 = at + C_2 \tag{1.16}$$
$$\ln y = at + C \tag{1.17}$$

Given an initial value y_0 at time $t = 0$, we can solve for C:

$$\ln y_0 = a(0) + C$$
$$C = \ln y_0$$

Substituting into the original function yields the exponential:

$$\ln y = at + \ln y_0$$
$$y = e^{at + \ln y_0}$$
$$y = y_0 e^{at}$$

where y_0 defines the initial function value (substance), a defines the growth rate, and $y = f(t)$ describes the amount of the substance at any future time t.

Exercises

Use the following problems to asses your background and determine if any additional study is needed before proceeding.

Exercises for 1.1 Complex Variables

P1.1 Calculate $y = i^3$

P1.2 Calculate $y = i^{631}$

P1.3 Calculate $y = z^2 z^*$ where $z = (2+3i)$

P1.4 Calculate$y = e^{i\pi}$

P1.5 Show that $e^{\frac{2i\pi}{3}} = \frac{-1+i\sqrt{3}}{2}$

Exercises for 1.2 Hilbert Spaces

P1.6 Describe the Hilbert space of $\mathbf{y} = \mathbf{A}\mathbf{B}\mathbf{x}$ if $\mathbf{A} \in \mathbb{R}^{5\times4}$ and $\mathbf{x} \in \mathbb{C}^7$. What is $\mathbf{B}$?

P1.7 Calculate $\mathbf{x}\cdot\mathbf{y}$ and $\mathbf{x}\otimes\mathbf{y}$ where

$$\mathbf{x} = \begin{bmatrix} 2 \\ -7 \\ 3 \\ 1 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} -1 \\ 3 \\ 2 \\ 2 \end{bmatrix}$$

P1.8 Calculate the angle θ between the two vectors $\mathbf{x} = [2 \quad 7 \quad -1]^T$ and $\mathbf{y} = [-1 \quad 2 \quad 3]^T$.

P1.9 The inner product $\mathbf{x}\cdot\mathbf{y}$ for complex vectors is defined as $\mathbf{x}^T\bar{\mathbf{y}}$ (where $\bar{\mathbf{y}}$ is the complex conjugate of $\mathbf{y}$). Calculate $\mathbf{x}\cdot\mathbf{y}$ for the complex vectors:

$$\mathbf{x} = \begin{bmatrix} 2i \\ -7+2i \\ 3-i \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} -1+2i \\ 1-3i \\ i \end{bmatrix}$$

Exercises for 1.3 Solving Linear Systems

P1.10 Solve the following linear system:

$$\begin{aligned} x - 2y + 3z &= 7 \\ 2x + y + z &= 4 \\ -3x + 2y - 2z &= -10 \end{aligned}$$

P1.11 Solve the following linear system using Gaussian Elimination and back-substitution:

$$\begin{aligned} a+b-2c+d+3e-f &= 4\\ 2a-b+c+2d+e-3f &= 20\\ a+3b-3c-d+2e+f &= -15\\ 5a+2b-c-d+2e+f &= -3\\ -3a-b+2c+3d+e+3f &= 16\\ 4a+3b+c-6d-3e-2f &= -27 \end{aligned}$$

Exercises for 1.4 Derivatives

P1.12 Calculate the derivative of $f(x) = \frac{\sqrt{x}}{x+3}$

P1.13 Calculate the second derivative of $f(x) = \frac{\sin(3x)}{4+5\cos(2x)}$

P1.14 Calculate the derivative of $f(x) = \frac{x\csc x}{3-\csc x}$

Exercises for 1.5 Taylor Series

P1.15 Calculate a 4th order Maclaurin series for $f(x) = e^x \cos x^2$

P1.16 Calculate a 4th order polynomial approximation to $f(x) = e^{-4x}$ near $x = -4$

P1.17 Calculate a 4th order polynomial approximation to $f(x) = \sqrt{3}$ near $x = 3$

P1.18 Use a Taylor series expansion to calculate:

$$\lim_{x\to 0} \frac{1-\cos x}{2x^2}$$

Exercises for 1.6 Differential Equations

P1.19 Find the general solution $y(x)$ that satisfies $\frac{dy}{dx}2y = (2x+1)$

P1.20 Find the general solution $y(x)$ that satisfies $\frac{dy}{dx} = 2\cos(2x)$

P1.21 Find the function $y(x)$ that satisfies $\frac{dy}{dx} = 7y^2x^3$ with the boundary condition $y(2) = 3$

P1.22 Find the general solution $y(x)$ that satisfies $x\sin^2 y\frac{dy}{dx} = (x+1)^2$

P1.23 Find the function $y(x)$ that satisfies $\frac{dy}{dx} = -2x \tan y$ given the boundary condition $y(0) = \frac{\pi}{2}$

P1.24 Find the function $y(x)$ that satisfies $\left(1 + x^2\right) \frac{dy}{dx} + xy = 0$ given the boundary condition $y(0) = 2$

Chapter 2

Prerequisites: Programming

This textbook assumes experience with C/C++ that is comparable to an undergraduate programming course. Several code examples, assignments, and implementation discussions rely on the reader's familiarity with C/C++ language. In particular, this textbook assumes that the reader understands:

- Basic data types used to allocate variables and memory, as well as how those data types behave in mathematical operations through promotion and casting.
- Methods for input and output (I/O), including command-line arguments, reading and writing data via the console, and reading and writing binary and text data using files.
- Dynamic memory allocation, including preventing memory leaks and segmentation faults as well as allocating multidimensional arrays.

High-level examples, which focus on the use of numerical methods rather than their implementation, will be provided in Python. These are designed to be easy to follow given some experience with a scripting language such as Python and MATLAB.

2.1 Data Types

C/C++ require that data types be specified for variables at compile time. These data types dictate variable behavior and limitations. The most common data types we will use are:

- `double` - a 64-bit value capable of representing both whole and fractional numbers
- `size_t` - a value capable of representing a natural number and addressing all of the memory available on the host system (usually an `unsigned long long`)
- `int` - an value capable of representing 32-bit positive and negative integers

These values stand out because they generally provide the greatest precision for numerical calculations. Other common data types include:

- `char` and `unsigned char` - values used to store a single byte or character (usually as part of an input string)
- `float` - a 32-bit value capable of representing whole and fractional numbers that is less precise than a `double` but usually faster to calculate

2.1.1 Promotion

It is important to understand how variables of different types behave during arithmetic operations. When a standard mathematical operation is performed (ex. "+", "-", "/", "*"), all operands are always the *same* data type. If two operands are of *different* types, one of them is *promoted* (Figure 2.1).

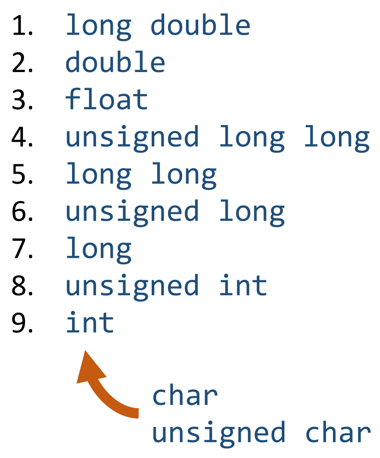


Figure 2.1 Promotion order in C/C++. If an operation has operands of multiple data types, all operands are promoted by being cast to the data type with the highest priority. Any `char` is always promoted to an `int`.

2.1.2 Casting

The assignment of a value of one data type to a variable of another, including promotion, is generally referred to as *casting*. Implicit casting is generally transparent to the user:

```
...
double d = 1.0/3.0;     //64-bit representation of an irrational number
double x = 10*d;        //10 (an integer) is promoted to a double
int i1 = x;             //x is implicitly cast to an integer, i1 = 33
int i2 = (int)x;        //x is explicitly cast to an integer, i2 = 33
...
```

In most cases, explicit casting is preferred and it is possible to set most compilers to require it. This acts as a "sanity check" for the programmer since a cast can result in a significant loss of data, such as when a floating point value with a fractional component is cast to an integer. Explicit casting forces the programmer to acknowledge the potential for data loss.

2.1.3 Overflow and Underflow

Integral data types (ex. `int`) can only represent a limited number of values. Details for this will be described in Chapter 5. **In C/C++, numerical overflow and underflow are defined for unsigned integral types only.** For all other data types, overflow/underflow are undefined. Overflow is defined for unsigned integral values because it can lead to extremely efficient calculations of particular values, such as the *modulus* operation.

Assigning a negative value to an `unsigned int` will result in an *underflow* while assigning a value higher than the maximum supported will result in an *overflow*:

```
...
#include <limits.h>
...
unsigned int a = -1;            //underflow
unsigned int b = UINT_MAX * 2;  //overflow
...
```

IEEE standard behavior is defined the case of overflow for unsigned integers in C/C++. The values *wrap around* back to zero:

```
...
#include <limits.h>
...
unsigned int a = UINT_MAX;      //a is assigned the maximum value
unsigned int b = UINT_MAX + 1;  //values "wrap around": b = 0
unsigned int c = UINT_MAX + 2;  //c = 1
unsigned int d = UINT_MAX + 10; //d = 9
...
```

This behavior is similar to what one would see in an *odometer*. However, this behavior is more mathematically useful when seen as a way to quickly perform a modulus operation.

Modulus Operation Using Overflow

In C/C++, overflow of an unsigned integer results in the application of the *modulus* operation, with an operand based on the number of bits assigned to the destination variable. If a value x is cast to an n-bit unsigned integral data type and stored in variable y, the result is equivalent to applying the operation:

$$y = x \mod 2^n$$

Since an `unsigned char` is an 8-bit value, the following code:

```
    ...
    unsigned int y = x;     //calculate y = x % 256
```

is the equivalent of calculating $y = \lfloor x \rfloor \mod 2^8$, where $\lfloor x \rfloor$ is the floor operation, removing any fractional part from x.

2.2 Input and Output

2.2.1 Command-Line Arguments

Command-line arguments are extremely useful for sending input to an executable for processing. These can be parameters for algorithms, file names, or even raw numerical input. C/C++ supports command-line arguments natively as parameters to the `main()` function:

```
int main(int argc, char* argv){
    ...
    for(i = 1; i < argc; i++){      //for each argument
        ...                         //process argv[i]
    }
}
```

where the first parameter `argc` provides the number of arguments and the second parameter `argv` is an array of `argc` strings storing each argument. Note that the first argument stored in `argv[0]` is the command used to call the executable and is generally ignored. Command-line arguments are separated by spaces.

The following string-processing functions, defined in `stdlib.h`, are useful for managing numerical input:

- `atof(char* str)` - returns a `double` value represented by `str`
- `atoi(char* str)` - returns an `int` value represented by `str`

String processing is time-consuming, so **always convert text data to native data types as soon as possible**.

2.2.2 Console and Piping

The first input/output method taught in most C/C++ programming classes involve the *console*.

2.2.3 Files

Most numerical methods that use file I/O will operate directly on binary files. Reading and writing binary data directly is far more efficient than string processing. However, if data is converted to/from text, it should be done right before writing or after reading. String processing is extremely slow and should always be minimized.

Reading a binary file requires knowledge of its structure, since variables must be appropriately allocated for storage in memory. Native functions in C/C++ will generally be sufficient:

```
int main(int argc, char* argv){
    FILE* in = fopen(argv[1], "rb");    //open a file for binary reading
    int x;                              //allocate space for an integer
    double y[3];                        //allocate an array of double values
    fread(&x, sizeof(int), 1, in);      //read an integer
    fread(y, sizeof(double), 3, in);    //read an array of 3 double values
    fclose(f);                          //close the file
    ...                                 //data processing
    FILE* out = fopen(argv[2], "wb");   //open a file for binary writing
    fwrite(&x, sizeof(int), 1, out);    //write an integer
    fwrite(y, sizeof(double), 3, out);  //write an array of double values
    fclose(f);                          //close the file
}
```

The beginning of a binary file often contains a *header* that specifies file type and formatting data. In particular, this usually includes the size of data blocks, allowing the programmer to allocate sufficient memory and effectively read the data. Images are good examples of data blocks that can vary in size and format (color, grayscale). An open and common image format containing a simple header is the Portable Network Graphics (PNG) format (Figure 2.2). MATLAB arrays are also often used to store blocks of data that vary in size and data type (Figure 2.3). If an unknown file format is encountered, it is often useful to browse for a header using a *hex dump* (Chapter 5).

2.3 Memory Allocation

Many numerical methods are designed to produce or operate on large blocks of data. These data sets often vary in size based on user-specified parameters. For example, the user may specify the number of samples required for a function, or the size of an input matrix.

2.3.1 Stack and Heap

The memory structure of an active program can be broadly split into two sub-structures:

- **call stack** - stores all active global and local variables as well as pointers to all active functions
- **heap** - stores all data that has been dynamically allocated for use by the active program

In C/C++, these memory substructures are exposed to the user. This allows a programmer to more effectively optimize how an algorithm performs.

The first important point to be aware of is that variables stored on the call stack are relatively faster to access, however storage space is limited. All locally declared variables, including arrays allocated with constant size, are placed on the stack. Most compilers require that the size of the call stack be known at compile time in order to minimize the risk of a *stack overflow* error. For example, the following code is **not** allowed by modern compilers:

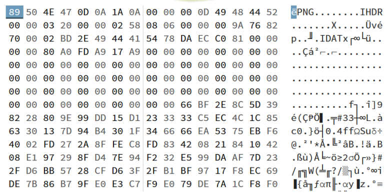
```
...
int N;
N = ...          //ask the user to specify N
double x[N];     //allocate an array of size N on the stack
...
```

This code will cause an error because it attempts to allocate an array of N 64-bit values without knowing the size of N at compile time. In order to compile this code, N must be specified as a constant:

```
...
double x[256];     //allocate an array of size N on the stack
...
```

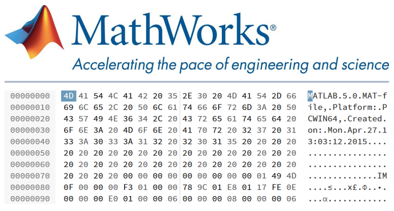
or

```
#define N 256
...
double x[N];     //allocate an array of size N on the stack
...
```



```
89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52   ëPNG........IHDR
00 00 03 20 00 00 02 58 08 06 00 00 00 9A 76 82   .......X.....Üvé
70 00 02 BD 2E 49 44 41 54 78 DA EC C0 81 00 00   p..╜.IDATx┌∞└ü..
00 00 80 A0 FD A9 17 A9 00 00 00 00 00 00 00 00   ..Çá²⌐.⌐........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 66 BF 2E 8C 5D 39   ..........f┐.î]9
82 28 80 9E 99 DD 15 D1 23 33 33 C5 EC 4C 1C 85   é(Ç₧Ö▌.╤#33┼∞L.à
63 30 13 7D 94 B4 30 1F 34 66 66 EA 53 75 EB F6   c0.}ö┤0.4ffΩSuδ÷
40 02 FD 27 2A 8F FE C8 FD 83 42 08 21 84 10 42   @.²'*Å■╚²âB.!ä.B
08 E1 97 29 8F D4 7E 94 F2 32 E5 99 DA AF 7D 23   .ßù)Å╘~ö≥2σÖ┌»}#
2F D6 BB 57 28 CF D6 3F 2F B1 BF 97 17 F8 EC 69   /╓╗W(╧╓?/▒┐ù.°∞i
DE 7B 86 B7 9F E0 E3 C7 F9 E0 79 DE 7A 1C F8 F0   ▐{å╖ƒαπ╟·αy▐z.°≡
```

Figure 2.2 PNG is an open raster-graphics format that supports color images in a variety of pixel formats as well as lossless compression (top). The file contains a header describing the image size and pixel format, followed by binary data containing the actual pixel values (bottom). (Wikipedia)



```
00000000  4D 41 54 4C 41 42 20 35 2E 30 20 4D 41 54 2D 66   MATLAB.5.0.MAT-f
00000010  69 6C 65 2C 20 50 6C 61 74 66 6F 72 6D 3A 20 50   ile,.Platform:.P
00000020  43 57 49 4E 36 34 2C 20 43 72 65 61 74 65 64 20   CWIN64,.Created.
00000030  6F 6E 3A 20 4D 6F 6E 20 41 70 72 20 32 37 20 31   on:.Mon.Apr.27.1
00000040  33 3A 30 33 3A 31 32 20 32 30 31 35 20 20 20 20   3:03:12.2015....
00000050  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   ................
00000060  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   ................
00000070  20 20 20 20 00 00 00 00 00 00 00 00 00 01 49 4D   ..............IM
00000080  0F 00 00 00 F3 01 00 00 78 9C 01 E8 01 17 FE 0E   ....≤...x£.Φ..■.
00000090  00 00 00 E0 01 00 00 06 00 00 00 08 00 00 00 06   ...α............
```

Figure 2.3 The MAT file format is designed and maintained by Mathworks for storing binary data related to MATLAB matrices. The Version-4 specification is generally easy to read and incorporate into custom software for storing and loading two-dimensional matrices of various data types. (Mathworks)

Even if you have access to a C compiler that supports dynamic stack allocation, **never use it**. The stack is relatively small when compared to the size of modern input data sets and a stack overflow is almost certain.

2.3.2 Dynamic Allocation

The alternative to allocating a relatively small amount of data on the call stack is to utilize the *heap*. The memory available on the heap is limited by:

1. the amount of hardware RAM available to the system
2. the size of the memory addresses allowed

Most modern systems allow 64-bit addressing, although many Windows applications are still using 32-bit memory addressing. If you compile your code for a 32-bit system, you are limited to $M = 2^{32} \approx 4GB$ of memory.

Allocation and manipulation of data on the heap is done using *pointers* in C/C++ returned using the `malloc()` function:

```
#include <stdlib.h>     //contains the definition for malloc()
...
int N;
N = ...                 //get data size
double* x;              //allocate a pointer to heap data
x = (double*) malloc(N * sizeof(double));   //allocate
...                     //use data
y = x[i];               //data is accessed with array indexing
...                     //use data
free(x);                //release allocated heap memory
```

The variable x is of data type `double*`. All pointer values are technically the same data type: `size_t`. This is imply a memory address identifying the location of the first element in the allocated memory block. The only reason to specify the `double` data type as an element of the pointer is that it tells the compiler how the data will be accessed when array notation (ex. `x[i]`) is used. A pointer value that doesn't have a data type associated with it is termed a *void pointer* and actually has the data type `void*`. If you look at the function signature for `malloc()`, it returns a value of type `void*`.

When the memory is no longer needed, it is good practice to release it using the `free()` function. Failure to do so can introduce a *memory leak*. If the pointer associated with the heap allocation goes out of scope, the memory is no longer accessible. If this is done repeatedly for large blocks of memory, the program will run out of accessible memory space. However, all memory is released by the operating system (OS) when the program terminates.

2.3.3 Segmentation Faults

Perhaps the most common error encountered when designing numerical algorithms is a *segmentation fault* (or *segfault*). A segmentation fault is a symptom of an inappropriate memory access:

```
#include <stdlib.h>     //contains the definition for malloc()
```

```
#define N  256
...
double* x;              //allocate a pointer to heap data
x = (double*) malloc(N * sizeof(double));   //allocate
for(int i=0; i <= N; i++){  //loop runs 257 times: [0, 256]
    x[i] = i;               //assign a value
}
...
```

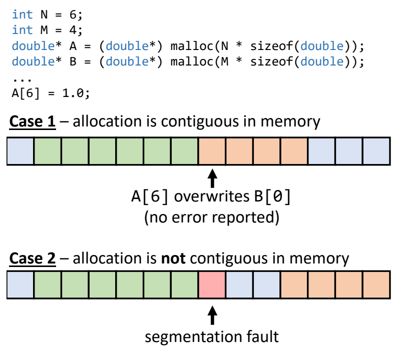
This code demonstrates common errors that can lead to a segmentation fault. First of all, an array is allocated to hold $N = 256$ elements. A loop then assigns a unique value to each element. However, this loop executes 257 times, causing the program to write to element `x[256]` which **has not been allocated**.

Since the program is simply following addressing instructions, the command `x[257] = 257` translates to "write the value 257 to memory location `x + 256`", where `x` is a memory address. This will result in one of the following:

1. **segmentation fault** - if the memory **has not** been allocated to the program, the attempted access will trigger an error from the operating system
2. **no fault** - if the memory **has** been allocated to the program, this will simply overwrite whatever is at that memory location

The second problem (2) is actually worse because it can go undetected by the user and result in erroneous output. However, one symptom of (2) is that the program provides different output on different systems or at different times. This is because the memory address of the allocation will vary depending on what addresses are available to the OS. This will change over time.

It is highly recommended that you fix segmentation faults immediately. Adding additional code, particularly additional memory allocations, makes the error harder to detect later.



```
int N = 6;
int M = 4;
double* A = (double*) malloc(N * sizeof(double));
double* B = (double*) malloc(M * sizeof(double));
...
A[6] = 1.0;
```

Case 1 – allocation is contiguous in memory
A[6] overwrites B[0]
(no error reported)
Case 2 – allocation is not contiguous in memory
segmentation fault

Figure 2.4 A segmentation fault is a reported error that occurs when unallocated memory is accessed. This error can be intermittent, depending on how memory is allocated at run-time. If two dynamic allocations are adjacent, the same incorrect array access will not result in a reported error and will instead unexpectedly change array values. **When a segmentation fault is encountered, it should be found and fixed as soon as possible.**

2.3.4 Multidimensional Arrays

Many numerical methods are applied on high-dimensional data, including tensors and grids. If the size of the array is known and small enough to fit on the heap, the notation is simple:

```
#define N 20
...
double x[20][20];   //allocate a 20x20 matrix on the stack
...                 //initialize values in the matrix
double y = x[i][j]; //access an element at index (i,j)
...
```

When the data size is unknown or the data is large, dynamic allocation becomes necessary. In order to achieve similar indexing behavior to the previous stack-allocated case, a series of recursive pointer arrays can be used:

```
int M, N;               //allocate variables to store matrix size
...                     //initialize/retrieve matrix size
double** x;             //generate a pointer to an array of pointers
x = (double**) malloc(M * sizeof(double*));     //allocate pointer array
```

```
for(int i = 0; i < M; i++){      //for each value in the pointer array
    x[i] = (double*) malloc(N * sizeof(double));//allocate a value array
}
...                         //initialize values in the matrix
double y = x[i][j];         //access an element at index (i,j)
...
```

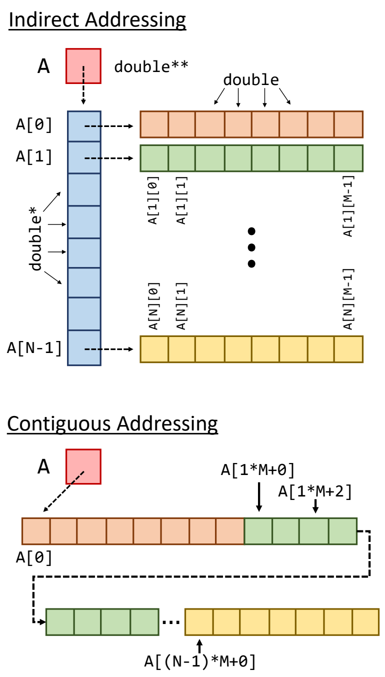


Figure 2.5 Both indirect and contiguous addressing are viable for working with multidimensional arrays. For performance reasons, contiguous addressing is generally preferred since only one memory fetch is required to access an array element.

This method allows more traditional indexing by using an *indirect lookup* via an array of M pointers to different arrays of size N. The initialization is clearly more complex, however the real penalty comes in the form of slower memory access due to the indirect look-up. The use of a primary array to access a secondary array requires one additional memory access and reduces the effectiveness of caching.

The alternative is to create a linear array, which guarantees that all elements are allocated in a single contiguous block:

```
int M, N;                   //allocate variables to store matrix size
...                         //initialize/retrieve matrix size
double* x;                  //allocate space for a single memory address
x = (double*) malloc(M * N * sizeof(double));  //allocate pointer array
...                         //initialize values in the matrix
double y = x[i*N + j];      //access an element at index (i,j)
...
```

The required indexing is more complicated, however the allocation is simple and does not require an indirect address fetch. In addition, contiguous blocks are much easier to copy and save. **In almost all cases, contiguous allocation is recommended.**

Exercises

Exercises for 2.1 Data Types

P2.1 What are the values stored in the registers represented by x, y, and z in the following code?

```
...
unsigned char a = 200;
unsigned char b = 2;
unsigned int c = 128;
unsigned int x = a + c;
unsigned int y = (unsigned char)(a + c);
unsigned int z = a * b;
...
```

P2.2 Write a program that takes a series of numerical values as command line parameters and outputs their sum.

P2.3 How would you send the output from P2.2 to a file named `output.txt`?

Exercises for 2.3 Memory Allocation

P2.4 Allocate space for a matrix **A**, right-hand-side vector **b**, and a variable vector **x** for a system of N equations and N unknowns. Initialize **x** to zero.

Chapter 3

Discrete Mathematics

In this chapter we will study a variety of tools used to objectively measure the difference in run time and hardware requirements between algorithms. Combined with the measurements for error provided earlier, we will be able to objectively compare algorithms and select ones that are appropriate for various applications. We will also look at a basic example of designing an algorithm that runs exponentially faster than a brute-force implementation for evaluating polynomials.

In this chapter, you should learn:

- How computational complexity is defined and how it can be used to compare algorithms
- How to determine the computational complexity of an algorithm based on its expression or source code
- How to implement Horner's method for calculating the value of a polynomial and its derivative

3.1 Computational Complexity

When designing algorithms, it is critical to understand how they behave as the size of the input increases or decreases. This behavior is called *computational complexity* and is closely related to asymptotic analysis (Section 4.6). We are interested in how the time and memory requirements for an algorithm are effected by changing the size of the input. For example, assume that we test our algorithm using n scalar values and it takes 1 day to complete. By understanding the computational complexity of our algorithm, we can answer questions like:

- How much time would be required if we doubled the size of the input?
- Would we have sufficient memory to solve the problem?

Asymptotic analysis is used to describe the behavior of a process as a function of the input based on its dominant components. Consider an algorithm that performs the following operations:

1. Load an ordered set of $N+1$ real coefficients $\mathbf{c} = [c_0,\ c_1,\ \cdots,\ c_N]$ where $c_i \in \mathbb{R}$
2. Load a value $x \in \mathbb{C}$
3. Evaluate the polynomial:

$$
\begin{aligned}
p(x) &= c_0 + c_1 x + c_2 x^2 + \cdots + c_N x^N \\
&= \sum_{n=0}^{N} c_n x^n
\end{aligned}
$$

The actual time required to complete each step will depend on the hardware and methods used. However, we can analytically characterize the behavior of this algorithm such that we can predict how it will perform for varying numbers of coefficients $N+1$.

Step (1) requires loading $N+1$ coefficients. If they are transferred from another memory location, each coefficient may require a few micro-seconds (μs) to copy. If the values are stored on a hard drive or remotely, the each coefficient may require several milliseconds (ms). If the values are entered manually, the user may require a second or more for each coefficient. However, we have an expression for the number of values (N) and can therefore characterize the time required for (1) as:

$$T_1 = C_1(N+1)$$

where C_1 is the amount of time required to load each value.

Step (2) requires only 1 value. Since x may or may not use the same loading mechanism as $\mathbf{c}$, we will use a separate constant:

$$T_2 = C_2$$

Step (3) is more complex, requiring that all N values be read from memory and multiplied by x^n where n is the coefficient index. In this example, we will perform a brute-force evaluation of x^n:

$$x^n = \prod_{n=1}^{N} x$$

where each multiplication requires some amount of time C_3. Since the number of required multiplications increases with the coefficient index:

$$
\begin{aligned}
T\{c_0\} &= 0 \\
T\{c_1 x\} &= C_3 \\
T\left\{c_2 x^2\right\} &= 2C_3 \\
T\left\{c_3 x^3\right\} &= 3C_3 \\
&\cdots \\
T\left\{c_N x^N\right\} &= NC_3
\end{aligned}
$$

the time required for step (3) is:

$$T_3 = \frac{1}{2} C_3 N^2$$

The total time required to perform this algorithm is:

$$T = C_1(N+1) + C_2 + \frac{1}{2} C_3 N^2$$

Based on our previous discussion of asymptotic analysis (Section 4.6), we know that for higher-order polynomial terms will always exceed lower-order terms independent of their coefficients in the asymptotic case. Step (3) will dominate the total time for large N, even if $C_1 \gg C_3$ or $C_2 \gg C_3$. When considering which algorithms to use or how an algorithm scales, the constants are irrelevant. Because of this, we simply say that the algorithm scales quadratically with the number of input values. This is expressed using "big O" notation as:

$$O\left(N^2\right)$$

which translates to "this algorithm behaves on the order of N^2" or "this algorithm has a *time complexity* of order-N^2".

Big-O Notation

Big-O notation is used to describe the upper-bound asymptotic behavior as the argument to a function approaches infinity. Formally, $f(n) = O\big(g(n)\big)$ if there is some constant M for which $|f(n)| \leq M|g(n)|$ if $n \geq n_0$. We generally expect n_0 to be less than the expected input size of the function.

An algorithm classified as $O\left(N^2\right)$ is said to be "on the order of N^2" as $N \to \infty$. Intuitively, this expression says that an algorithm or function behaves *like* the function expressed in big-O notation scaled by some constant value. The lower-bound of n_0 addresses the fact that algorithms can have complex behaviors for small inputs due to hardware or software, such as caching, overhead, and memory allocation. n_0 is assumed to be large enough that these features average out and can be incorporated into the constant M.

Complexity analysis is extremely useful when designing and selecting algorithms by providing us with tools to:

- Estimate the amount of time required for a large input based on measurements of a smaller input.
- Predict the complexity of algorithms that we want to develop.
- Determine where to focus optimization efforts.

3.1.1 Estimating Complexity

Determining the computational complexity of an algorithm becomes easier with experience, as you build up the library of algorithms that you have worked with.

The easiest way to determine the computational complexity of an algorithm is to consider the number of *nested loops* required in the implementation. The brute-force polynomial evaluation described previously could be expressed as:

$$p(x) = \sum_{n=0}^{N} \left(c_n \prod_{k=0}^{n} x \right)$$

which contains two nested iterative operators (summation and product) that are both dependent on the size of the input N. The number of nested loops can be counted in an implementation of the algorithm as well:

```
...
unsigned N;                     //number of coefficients
double* C;                      //array of coefficients
double x;                       //parameter value
double y = 0;                   //polynomial result
...                             //initialize values
int n, k;                       //declare loop iterators
double x_n;                     //stores the value of pow(x, n)
for(n = 0; n < N; n++){         //for each coefficient
    x_n = 1;
    for(k = 0; k < n; k++)      //for each exponent
        x_n *= x;               //calculate the exponential
    y += c[n] * x_n;            //add to the summation
}
return y;                       //return the result
```

When using nested loops as an indication of the computational complexity, note that loops with a constant number of iterations are rolled into the constant portion of the implementation time. Consider an algorithm that calculates the average value of an $N \times 3$ matrix $\mathbf{A} \in \mathbb{R}^{N\times 3}$. The output can be represented using two nested iterative operators:

$$y = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{1}{3} \sum_{j=1}^{3} A_{ij} \right)$$

The complexity of this algorithm is $O(N)$, since the inner loop does not change as a function of input size and is therefore constant.

When the size of the input is dependent on multiple variables, such as averaging the values of an $N \times M$ matrix $\mathbf{A} \in \mathbb{R}^{N\times M}$:

$$y = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{1}{M} \sum_{j=1}^{M} A_{ij} \right)$$

the $O(MN)$ complexity can be expressed as a function of all input sizes. For most problems, the input size actually changes in very few ways and we often know ahead of time how they are related to each other. In the previous example, we may know that $M \ll N$ and can therefore approximate the algorithm as $O(N)$.

Here are a list of the most common values for computational complexity, as well algorithms associated with each:

- **O(1) - constant time** - This expression is used for algorithms that have a constant execution time independent of the input. Determining if the value $x \in \mathbb{I}$ is even or odd is a constant time operation, since it requires looking at a single digit and is independent of the number of digits used to represent x.
- **O(N log N) - logarithmic time** - Many searching algorithms fall into this category, including bracketing algorithms used for calculating roots of equations (Chapter 7).
- **O(N) - linear time** - Calculate the average of N values or finding the smallest value in a list of N unsorted values.
- **O(N²)** - **quadratic time** - Many algorithms designed to sort lists of N values, most operations that occur on $N \times N$ matrices (including backsubstitution).
- **O(N³)** - **cubic time** - Algorithms for solving systems of N linear equations. This includes Gaussian elimination, eigendecomposition, or singlular-value decomposition. **O(N!)** - **factorial time** - This is an example of a non-polynomial (NP) class algorithm. Algorithms in this category are generally so slow as to be unusable for large input. A well-known algorithm of this class is the brute-force solution to the Traveling Salesman Problem.

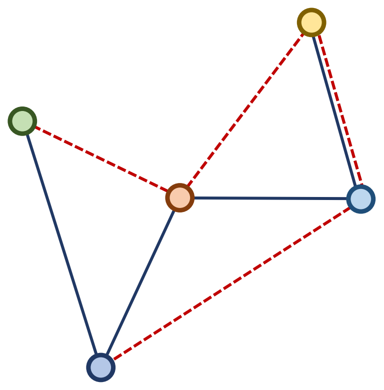


Figure 3.1 The traveling salesman problem (TSP) is well-known in theoretical computer science as an example of *combinatorial optimization*. The problem is formally stated as "Given a set of input cities and distances between each pair of cities, find the shortest cyclical path that visits all cities." The brute force solution requires testing all possible paths, requiring $O(n!)$ time. Other methods, such as the Held-Karp algorithm, can compute the exact solution in $O(n^2 2^n)$ time and requiring $O(n2^n)$ memory. (Wikipedia)

3.1.2 Horner's Method

Earlier in this chapter we discussed an $O(N^2)$ algorithm for calculating the value of a polynomial for a given input $x \in \mathbb{C}$ and set of C coefficients $\mathbf{C} \in \mathbb{R}^C$. This is based on the traditional formulation of a polynomial, which is generally expressed in descending order:

$$p(x) = c_N x^N + \cdots + c_2 x^2 + c_1 x + c_0 = \sum_{n=0}^{N} \left(c_n \prod_{k=0}^{n} x \right) \tag{3.1}$$

We can represent the same polynomial in nested form:

$$p(x) = ((((c_N)\, x \cdots)\, x + c_2)\, x + c_1)\, x + c_0 \tag{3.2}$$

Implementation of the first expression will yield the described $O(N^2)$ algorithm. However, implementation of Equation 3.2 requires significantly fewer operations, resulting in an algorithm that has a complexity of $O(N)$.

This linear-time algorithm for evaluating polynomials is known as Horner's method.

Synthetic Division

Horner's method can also be derived from the polynomial remainder theorem, implying that the result of a polynomial can be calculated using *synthetic division*. The synthetic division algorithm is identical to Horner's method.

Polynomial Remainder Theorem

The remainder of the division of a polynomial $p(x)$ by the first-order polynomial $x - a$ is equal to $p(a)$.

This is proven through Euclidean division:

$$\frac{p(x)}{d(x)} = q(x) + \frac{r(x)}{d(x)}$$

where dividing the polynomial $p(x)$ by a divisor $d(x)$ results in a new polynomial quotient $q(x)$ and remainder $r(x)$. Re-arranging terms gives us:

$$p(x) = d(x)q(x) + r(x)$$

If the divisor is the linear polynomial $d(x) = x - a$:

$$p(x) = (x - a)q(x) + r(x)$$

the quotient $q(x)$ will be eliminated, leaving only the remainder. Since $\deg(r) < \deg(q)$, the remainder term must be degree-zero and therefore is not a function of a:

$$p(a) = r$$

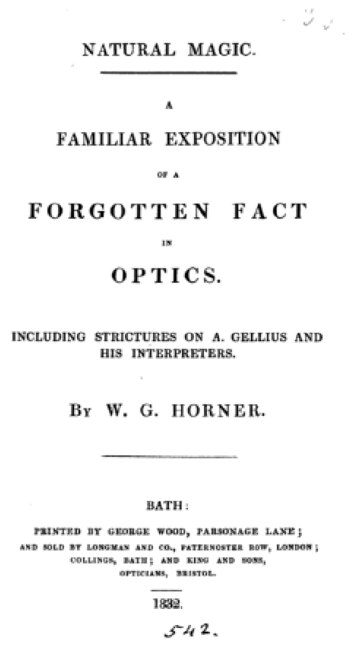
Synthetic division by $x - a$ is performed by writing down the coefficients **c** of the polynomial $p(x)$, starting from the highest order term:

$$\begin{array}{r|cccc} a & c_N & \cdots & c_1 & c_0 \\ \hline \end{array}$$

and then iteratively performing the following steps:

1. add each coefficient to the value underneath it
2. multiply the result of the sum by a
3. carry the result of the multiplication over to the next column

$$\begin{array}{r|cllcc} a & c_N & c_{N-1} & \cdots & c_1 & c_0 \\ \hline & & ac_N & a(c_{N-1} + ac_N) & \cdots & \cdots \\ \hline & c_N & c_{N-1} + ac_N & \cdots & \cdots & p(a) \end{array}$$


NATURAL MAGIC.

A

FAMILIAR EXPOSITION

OF A

FORGOTTEN FACT

IN

OPTICS.

INCLUDING STRICTURES ON A. GELLIUS AND HIS INTERPRETERS.

By W. G. HORNER.

BATH:

PRINTED BY GEORGE WOOD, PARSONAGE LANE; AND SOLD BY LONGMAN AND CO., PATERNOSTER ROW, LONDON; COLLINGS, BATH; AND KING AND SONS, OPTICIANS, BRISTOL.

1832.

542.

Horner's Method (William George Horner, 1786-1837) This method for solving polynomials in linear time is named after William Horner, however the algorithm was known previously by the mathematician's Paolo Ruffini and Qin Jiushao. (Wikipedia)

Example 3.1

Use Horner's method to calculate the value of the polynomial:

$$p(x) = 2x^5 + 4x^4 - 7x^3 + 5x^2 + 3x - 6$$

at $x = 2$.

Solution: Horner's method can be expressed in two ways. The polynomial can be written in nested form:

$$p(x) = (((((2)\,x + 4)\,x - 7)\,x + 5)\,x + 3)\,x + 6$$

and evaluated from the inner-most term:

$$\begin{aligned}
p(2) &= (((((2)\,2+4)\,2-7)\,2+5)\,2+3)\,2+6 \\
&= ((((8)\,2-7)\,2+5)\,2+3)\,2+6 \\
&= (((9)\,2+5)\,2+3)\,2+6 \\
&= ((23)\,2+3)\,2+6 \\
&= (49)\,2+6 \\
&= 104
\end{aligned}$$

Synthetic division can also be used to calculate the remainder term of $\dfrac{p(x)}{x-2}$:

$$\begin{array}{r|rrrrrr}
2 & 2 & 4 & -7 & 5 & 3 & 6 \\
\hline
 & & 4 & 16 & 18 & 46 & 98 \\
\hline
 & 2 & 8 & 9 & 23 & 49 & 104
\end{array}$$

In either case, the sequence of calculations is the same and the value of $p(2) = 104$ is calculated in $O(N)$ time.

Derivative Calculation

Given a degree-N polynomial $p(x)$ and a scalar value a, synthetic division allows us to calculate the quotient $q(x)$ and remainder r such that:

$$p(x) = (x-a)\,q(x) + r$$

The derivative of the polynomial using this expression is:

$$p'(x) = (x-a)\,q'(x) + q(x)$$

where the unknown $q'(x)$ term becomes zero when $x = a$, meaning that:

$$p'(a) = q(a)$$

This suggests that Horner's method can also be used to solve for the derivative of $p(x)$ at any point $x = a$ using two iterations of the algorithm: the first solves $p(a)$ and provides $q(x)$, while the second solves $p'(a)$ by calculating $q(a)$.

Example 3.2

Calculate the value of the following polynomial $p(x)$ and its derivative at $x = 3$:

$$p(x) = x^4 - 2x^3 + x + 5$$

Solution: Using two iterations of synthetic division, we can calculate the remainder term, giving $p(a) = r$ and the derivative $p'(a) = q(a)$:

$$\begin{array}{r|rrrrr}
3 & 1 & -2 & 0 & 1 & 5 \\
\hline
 & & 3 & 3 & 9 & 30 \\
\hline
 & 1 & 1 & 3 & 10 & 35 \\
\hline
 & & 3 & 12 & 45 & \\
\hline
 & 1 & 4 & 15 & 55 &
\end{array}$$

The resulting calculation gives $p(3) = 35$ and $p'(3) = 55$. We can verify the value of the derivative at $x = 3$ by computing the analytical expression and solving:

$$\begin{aligned} p'(x) &= 4x^3 - 6x^2 + 1 \\ p'(3) &= 4(3)^3 - 6(3)^2 + 1 \\ &= 108 - 54 + 1 \\ &= 55 \end{aligned}$$

Horner's Algorithm

3.2 Combinatorics

3.2.1 Permutations

3.2.2 Combinations

Exercises

Exercises for 3.1 Computational Complexity

P3.1 Calculate the time complexity of the following algorithms:

- $\mathbf{Av}$ where $\mathbf{A} \in \mathbb{R}^{N\times N}$
- Multiply N vectors $\mathbf{v}_i \in \mathbf{R}^3$ by a transformation matrix $T \in \mathbb{R}^{3\times 3}$
- $\mathbf{Y} = \mathbf{ABx}$ where $\mathbf{A} \in \mathbb{R}^{N\times M}$, $\mathbf{B} \in \mathbb{R}^{M\times N}$ and $\mathbf{x} \in \mathbb{R}^N$
- $\mathbf{x}\cdot\mathbf{y}$ where $\mathbf{x},\ \mathbf{y} \in \mathbb{R}^N$
- $\mathbf{x}\otimes\mathbf{y}$ where $\mathbf{x},\ \mathbf{y} \in \mathbb{R}^N$

Exercises for 3.1.2 Horner's Method

P3.2 Use synthetic division to calculate the result of each polynomial and its derivative:

- $x^4 - 4x^3 + 7x^2 - 5x - 2$ for $x = 3$
- $3x^4 - x^2 + 2x + 13$ for $x = 5$
- $2x^4 + 3x^3 - x^2 - 1$ for $x = -2$

Programming Assignment

Implement Horner's method in C/C++ to calculate the result of any polynomial $p(x)$. Provide the option to also return the quotient polynomial $q(x)$, which can be used to calculate the derivative of $p(x)$ for the programming assignment in Chapter 7.

To test this algorithm, take a list of coefficients **c** along with a value for x as command-line arguments and print the result to the console:

```
>> horner x c0 c1 c2 c3 ... cn
```

Chapter 4

Numerical Error

4.1 International Organization for Standardization

Throughout this book, we will consistently encounter measurements, processes, and terminology that are fundamental to engineering. We rely on standard terminology to communicate and convey engineering principals. For example, measurements are critical for providing instructions on how to build a mechanical device, such as a bridge. In order to communicate efficiently, we rely on standard terminology to define how measurements are taken processes are performed.

The primary source of standardization used in this book is the International Organization for Standards (ISO), which sets several standards commonly encountered in engineering and computation. Various standards that will be encountered within coming chapters include:

- **terminology** - Various terms, such as *precision* and *accuracy*, have specific definitions set by the ISO. While many of these terms are common in day-to-day conversation, this use is often "messy" and can cause confusion if the standard definition isn't clearly understood.
- **representation** - The ISO also sets standards for representing numerical values, as well as describing their behavior. For example, the most common method for representing numbers in digital systems is based on the *floating point* standard (**ISO/IEC/IEEE 60559:2011**).
- **programming/algorithms** - Programming languages frequently used in numerical computing are also set by the ISO. This includes C (**ISO/IEC 9899**), C++ (**ISO/IEC 14882**), and FORTRAN (**ISO/IEC 1539**). While the Python and Matlab scripting languages are not covered by the ISO, they rely heavily on libraries written in FORTRAN.

International Organization for Standardization (1947-present, Geneva, Switzerland) The ISO is responsible for setting voluntary standards that facilitate trade of goods and services between 163 member countries. The ISO covers a broad range of standards that include manufacturing, technology, and healthcare. While the ISO officially began operations in 1947, it was preceded by the National Standardizing Associations (ISA), which was established in 1926 and later suspended during World War II. The ISO is funded primarily by the subscriptions from member countries and the sale of standards to individuals and companies. (Wikipedia)

4.2 Accuracy: Precision and Trueness

In order to evaluate error in numerical processes, we must first understand the fundamental types of error that can occur. The **accuracy** of any process that measures or calculates a value can be expressed using the following two terms (ISO 5725):

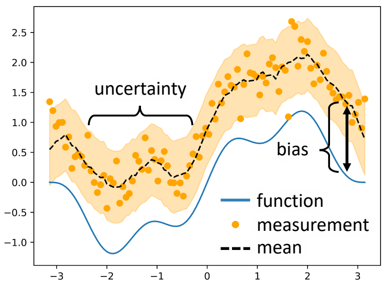


Figure 4.1 The accuracy of a process is defined by its *precision*, or statistical variability, and *trueness*, or average deviation. The precision of a measurement reflects its uncertainty or variance (orange), while a bias affects the trueness of a measurement by introducing a systematic shift.

- **Trueness** describes how well a series of values compare to the truth. A high degree of trueness implies that the arithmetic mean of several measurements lies close to a true or accepted reference value. Deviation of the mean from the true value is the result of a *bias* in the underlying process by which the values were measured or calculated.
  - **Example (measurement)**: A scale that is incorrectly calibrated (tared or zeroed) will introduce a systematic bias when used to measure weight.
  - **Example (computing)**: An algorithm that consistently rounds the results of operations up or down will produce biased results.
- **Precision** describes how well a series of values compare to each other. Imprecise values are the result of *uncertainty* in the process by which they were measured or calculated.
  - **Example (general)**: General statements such as "cold" or "hot" are imprecise because there is variability in how these terms can be interpreted. An exact temperature provides a more precise value.
  - **Example (measurement)**: A picture taken in low-light is often noisy. The precision of the image can be improved by increasing illumination, using a more sensitive camera, or averaging multiple images together.
  - **Example (computing)**: It is impossible to represent every real number using a digital system. Therefore every calculation must "clamp" the result to the nearest value that can be represented. The spacing between representable values in a digital system define the precision of the resulting calculation.

Note that it is common in engineering to use the term *accuracy* as a substitute for trueness. We have chosen to adhere to the ISO standard, however readers should be aware that it is common practice to evaluate the quality of a process that produces numerical results using "accuracy and precision" rather than "trueness and precision."

4.3 Calculating Error

When confronted with a numerical problem, our goal is to select the most appropriate method. However, we are always constrained by the tools that are available.

For example, extremely accurate calculations are time consuming and require a significant amount of processing power. We could hardly expect an autonomous drone to have access to the resources of a supercomputer.

We will therefore always resort to approximations, and must determine if a given approximation is sufficient to solve our problem. We rely on several metrics to determine the quality of an approximation. The simplest metric is the *error* ϵ, which is the deviation of our calculated or measured value c from the true value t:

$$\epsilon = t - c \tag{4.1}$$

4.3.1 Mean Squared Error

Our goal is often to minimize the magnitude of this error. When dealing with multiple measurements, ϵ is insufficient because errors above and below the target value can cancel each other out. We therefore generally determine the optimal method based on the *absolute error*:

$$|\epsilon| = |t - c| \tag{4.2}$$

When multiple measurements are being considered, this metric can be summed across all outputs to determine the *sum of absolute errors* (SAE):

$$\text{SAE} = \sum_{i=0}^{N} |\epsilon_i| \tag{4.3}$$

While selecting the method that minimizes the SAE is preferred, there are some disadvantages to this method. In particular, the SAE is difficult to optimize. As we will discuss later (Chapter 7), it is easiest to optimize functions that are differentiable. Because of this, one of the most common error metrics used is the *mean squared error* (MSE):

$$\text{MSE} = \frac{1}{N}\sum_{i=0}^{N} \epsilon_i^2 \tag{4.4}$$

This error function is differentiable, and minimizing the MSE is often the equivalent to minimizing the SAE.

4.3.2 Relative Error

Another metric commonly used in engineering and computing is the *relative error*:

$$E = \left|\frac{\epsilon}{t}\right| \tag{4.5}$$

which represents the ratio of the error to the true value. The relative error is frequently given as the percentage error:

$$E = 100 \times \left|\frac{\epsilon}{t}\right| \% \tag{4.6}$$

This relative and percentage errors are often the most important metrics *in practice*. For example, when we are measuring the voltage drop across a resistor, a difference of $\pm 1mV$ is extremely important. However, this value is irrelevant when measured across a mile-long high-voltage power line. In most practical applications, we are interested in minimizing the error with respect to the size of the result.

There are cases where the relative error cannot be directly used:

- When the correct or true measurement is zero, the relative error is undefined. When the true value is zero, the absolute error can be used as a substitute.
- Relative error is only valid for a ratio *level of measurement*. The value being measured or calculated must have a meaningful zero (Example 4.1).

Level of Measurement

A level of measurement, or *scale of measure*, is an abstract scheme for characterizing the type of data associated with a value. The most common scale consists of four types, each providing increasing specificity:

(a) **nominal** - Values specifying a class and allowing the basic comparisons of equality or inequality. Nominal measurements do not have an ordering: red, green; fruit, vegetable; car, truck.
(b) **ordinal** - Values that specify an order, allowing comparisons such as *greater than* and *less than*, but not the degree of difference: first place, second place, third place; highest, lowest; any list of items sorted by weight, volume, or other measurement.
(c) **interval** - Values that provide a degree of difference between measurements: year (in CE or BC), temperature (in F° or C°), longitude.
(d) **ratio** - Values that have a meaningful ratio or meaningful zero. Statements such as "half as long" or "twice as heavy" have meaning: temperature (in Kelvin), mass, length, age.

Example 4.1

A weather model based on satellite images calculates a predicted temperature of $10^\circ C$. The actual temperature measured on the ground is $11^\circ C$. What is the relative error of this prediction?

Solution: The measurements in degrees Celsius are *interval* measurements. They describe the degree of difference between other measurements in Celsius. Taking a ratio measurement is not meaningful: $20^\circ C$ is not *twice as hot* as $10^\circ C$. A temperature of $0^\circ C$ does not mean that there is no heat. The zero value for the Celsius scale is arbitrarily placed at the freezing point of water.

The calculation of relative error requires the use of a *ratio* scale. Fortunately, Celsius is an interval measurement that can be readily converted to the ratio scale Kelvin:

$$K = C + 273.15$$

Using the Kelvin scale, the true measurement is 283.15 K and the modeled value is 284.15 K. Calculating the relative error gives us:

$$E_R = \left| \frac{283.15 - 284.15}{283.15} \right| = 0.353\%$$

4.4 Rounding Error

Numerical algorithms almost never produce an exact result. At the very least, a digital system must use a finite number of digits to represent any value. That eliminates our ability to represent irrational numbers, such as π, e, and $\frac{1}{3}$. In calculations, these numbers are instead approximated using the closest representable value that can be stored in a numerical register. Errors like these are introduced at almost every step of a numerical algorithm. In fact, mitigation of these errors can be thought of as our primary goal when designing numerical algorithms.

Rounding is the act of replacing a value with a less complex, and often less precise, representation. This is most commonly done to reduce the number of digits used to represent a number. For example:

$$\begin{aligned} \pi &\approx 3.14159 \\ e &\approx 2.14 \\ \frac{1}{3} &\approx 0.3333 \end{aligned}$$

A rounding operation can be specified using a variety of rules. The most common rule for rounding in traditional arithmetic follows the following algorithm:

- **INPUT:** the mantissa x of a value represented using scientific notation
- **INPUT:** $N \in \mathbb{N}$ number of digits in the output
- **OUTPUT:** new mantissa y containing only N digits

1. consider digit d in the $N+1$ position of x
   (a) if $d < 5$:
   $$y = \frac{\lfloor x \times 10^d \rfloor}{10^d}$$
   (b) if $d \geq 5$:
   $$y = \frac{\lceil x \times 10^d \rceil}{10^d}$$

Note that the act of rounding introduces error into the output mantissa because digits, containing information, are being thrown away. This effectively reduces the *precision* of any calculation relying on the rounded value.

In computing, rounding is consistently performed by chopping digits that exceed the representable length of the mantissa. This occurs because there is no way to determine the value of the $N+1$ digit without using a larger register. It is impossible to access digits that cannot be stored in order to determine a more advanced rounding behavior.

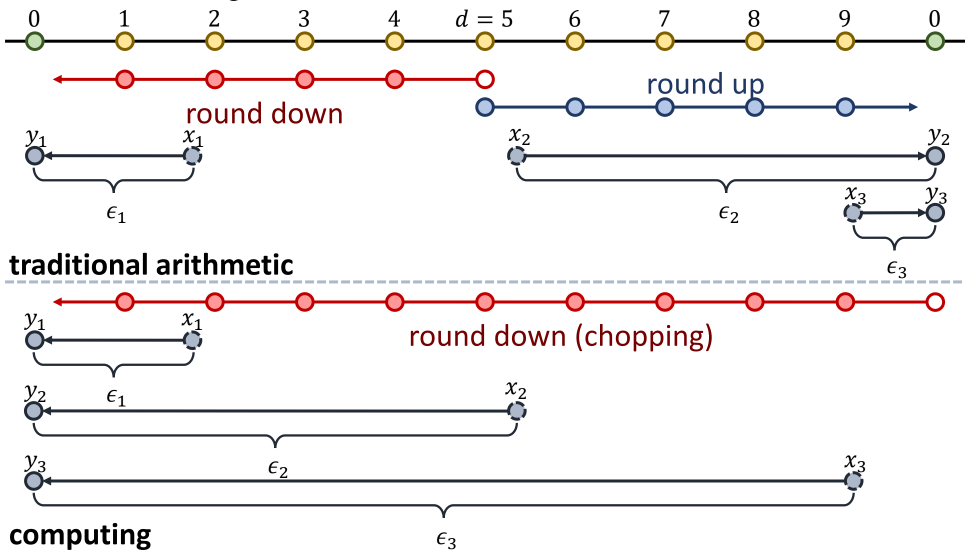


Figure 4.2 Application of rounding for traditional arithmetic (top) and chopping (bottom). Traditional rounding places the output at the nearest compatible value, resulting in an error ϵ that is *at most* half of the spacing between representable values (top, ϵ_1 to ϵ_3). In computing, rounding is performed by chopping insignificant values, resulting in a maximum error bounded by the spacing between representable values.

In addition to a loss of precision, rounding also introduces a bias. Note that the chopping procedure used in digital computing (Figure 4.2) will consistently *under*-estimate the correct result. Calculating the result of a function using a digital system will therefore generate a plot consistently lower than the result calculated by hand.

Interestingly, one should also note that traditional rounding *also* introduces a bias. There is some ambiguity when $d = 5$, in which case it is equidistant from both the lower and upper representable value. By consistently selecting the upper value, we are introducing a small upward bias. This bias could be theoretically eliminated by randomly selecting the higher or lower values when $d = 5$. Besides being difficult to implement in practice, this would also generate different results for the same calculation. We generally prefer this small bias to inconsistent results.

4.5 Truncation Error

Special functions, such as trigonometric functions and logarithms, are often computed using iterative series. Consider the calculation of $\sin x$ using a Maclaurin polynomial:

$$= \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1}$$

$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} - \ldots$$

Clearly it is impossible to evaluate an infinite series on a digital system, so the function must be evaluated with a finite number of terms. In the case of the Taylor series

$$T(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{4.7}$$

the denominator of each term contains the factorial of increasing value n. As the parameter increases, the factorial function $n!$ increases faster than any polynomial or exponential function. Provided that the denominator increases faster for successive terms than the numerator $f^{(n)}(a)(x-a)^n$, **as n increases successive terms will become smaller**.

We can therefore represent a truncated Taylor series as a finite summation and a *residual* or *remainder* term:

$$T(x) = \sum_{n=0}^{N} \frac{f^{(n)}(a)}{n!}(x-a)^n + R(x) \tag{4.8}$$

Alternatively, the residual can be expressed as a difference between the truncated Taylor polynomial $T_N(x)$ and the original function:

$$R(x) = f(x) - T_N(x) \tag{4.9}$$

If we want to calculate any function using a finite Taylor series, our goal is to select an order N such that the residual term $R(x)$ is acceptably small. This requires determining two constraints:

- What value of $R(x)$ is considered *acceptably small* in an algorithm?
- What is the value of the residual term $R(x)$?

While the definition of an acceptable residual may differ between applications, a good place to start is the value that can be represented given the numerical precision. In other words, we want to select $R(x) \leq \epsilon$ such that ϵ is the error introduced due to rounding (Section 4.4). Again, this is a general specification. For any given application, **if the speed of the calculation is more important than the accuracy, the order N of the series can and should be reduced.**

Calculating a value for the residual $R(x)$ is a little more difficult. In fact, this can only be done if we have some way to calculate the correct value of $f(x)$ (Equation 4.9). However, we can generally describe the *behavior* of the remainder term and, given some information about $f(x)$, we can restrict the magnitude of the residual $|R(x)|$.

4.6 Bounding Error

Calculating an acceptable bound on the residual $R(x)$ requires a couple of useful mathematical tools, the first of which is *asymptotic analysis*.

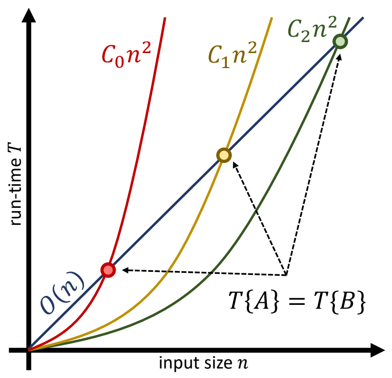


Figure 4.3 Asymptotic analysis is used to simplify the expression of algorithm run-times for large input. This plot shows the run-time of an $O(n)$ (linear) algorithm compared to the run-time of several $O(n^2)$ (quadratic) algorithms with different constants $C_0 > C_1 > C_2$. In this case, the constant could reflect the processing speed of the host machine. However large the leading constant, the quadratic algorithm will always exceed the run-time of the linear algorithm as n gets large. This method of analysis is extremely helpful for understanding how algorithms behave as the input size changes.

Asymptotic Analysis

Given any two exponential terms C_1x^n and C_2x^m where $n > m$, two statements can be made about the behavior of these functions with respect to x:

- As x becomes large, $C_1x^m < C_2x^n$. No matter how large C_1 or how small C_2, there will always be a value x for which the size of C_2x^n begins to exceed C_1x^m.
- As x becomes small, $C_1x^m > C_2x^n$.

Both of these points rely on the fact that exponential functions x^n get larger for increasing n if $x > 1$ and smaller if $x < 1$. Therefore, in the asymptotic case where $x \to 0$ or $x \to \infty$, we can say that any function "behaves like" x^n where n is the largest exponential in the function. This is usually expressed using "big-O" notation, where a function

$$f(x) = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n$$

is said to be $O(x^n)$ or *on the order of* x^n. This simply means that in the asymptotic cases where x gets really large or really small, the function $f(x)$ "behaves like" the polynomial x^n.

Based on asymptotic analysis, we can determine some interesting things about the residual term $R(x)$. Specifically, if we generate an nth-order Taylor series approximation to a function, we know that the *lowest* order term in $R(x)$ is $(x-a)^{n+1}$. This enforces a concept we already knew from Taylor series: the residual term (or error) in the approximation $T_n(x)$ gets smaller as $x \to a$. The difference is that we now have an idea of how quickly that happens since we know that the residual will be dominated by the $(x-a)^{n+1}$ term. The error will behave like h^{n+1} where $h = x - a$ is the distance from the expansion point of the Taylor polynomial. **If we reduce the spacing between x and a by some factor $\frac{1}{k}$, the error in our approximation is reduced by $\frac{1}{k^{n+1}}$.**

For example, consider a 5th order Maclaurin approximation $T_5(x)$ of the function $f(x)$. If we know the truncation error ϵ at some point x_0, we know that evaluating $T_5(x)$ at $\frac{x_0}{2}$ will reduce the error by $\frac{1}{2^5} = \frac{1}{32}$ to $\frac{\epsilon}{32}$. Alternatively, doubling the argument will increase the error by a factor of 32: $2^5\epsilon = 32\epsilon$

This is a particularly helpful result if we know the error for some value x. Some choices can be made to control that error:

- Re-evaluate the approximation for a closer expansion point $b \to x$
- Add additional terms to the expansion

In either case, we can measure how much of an adjustment is required by knowing the error for some value x.

The second tool that will be useful for bounding the residual term is the Remainder Theorem for Taylor Series. Specifically, the residual can be expressed using the Lagrange remainder.

Lagrange Remainder

Given an Nth order Taylor series approximation $T_N(x)$ for a function $f(x)$, the Lagrange remainder is expressed as:

$$R_N(x) = f(x) - T_N(x) = \frac{f^{(N+1)}(c)}{(N+1)!}(x-a)^{N+1} \tag{4.10}$$

where $f^{(N+1)}$ is the $N+1$ derivative of $f(x)$ and c is some value between x and a.

The Lagrange remainder can be used to apply a tight bound around the error term if some information about the *magnitude* of the derivative is known. If we know that $|f(x)^{(N+1)}| < K$, then the Lagrange remainder states that:

$$R_N(x) < \frac{K}{(N+1)!}(x-a)^{N+1}$$

Note that the Lagrange remainder is simply an alternative expression of the residual. This formulation is generally useful if we know the behavior of the function $f(x)$ that we are trying to approximate. If we have a means of calculating the maximum magnitude of the $N+1$ derivative of a function, the Lagrange remainder provides us with the largest possible constant

$$K < \frac{f^{(N+1)}(c)}{(N+1)!}$$

and asymptotic analysis provides us with the behavior of the function, $O\left(h^{n+1}\right)$, as a function of the spacing between a and x.

Example 4.2

What order Maclaurin series will be necessary to calculate the value of $\sin x$ for $x \le 1$ with an error of $\epsilon < 0.00014$?

Solution: We are looking for an order N approximation $T_N(x)$ of $\sin x$ such that the residual term $R(x) < 0.00014$. Since the approximation is a Maclaurin series, we know that the expansion is about $a = 0$ and can therefore express this residual using the Lagrange remainder:

$$R_N(x) < \frac{f^{(N+1)}(c)}{(N+1)!}(x)^{N+1}$$

Since we know that $x \le 1$, asymptotic analysis implies that the error will only get *smaller* as $x \to 0$. Therefore, if $R_N(1) < 0.00014$ we can guarantee that all values $x \le 1$ will also be below the error threshold:

$$R_N(x) < \frac{f^{(N+1)}(c)}{(N+1)!}$$

Finally, note that $\sin x$ is a periodic function and all of its derivatives are also periodic trigonometric functions: $\frac{d\sin x}{dx} = \cos x$, $\frac{d^2 \sin x}{dx^2} = -\sin x$, etc. The maximum magnitude of any of these functions is 1. Therefore, the maximum possible value for the numerator in the remainder term is $f^{(N+1)}(c) = 1$. This means that we can calculate an order N such that:

$$R_N(x) < \frac{1}{(N+1)!} \leq 0.00014$$

The first viable value of $N = 7$ where

$$\frac{1}{(7+1)!} = \frac{1}{40320} \approx 0.0000248 \leq 0.00014$$

Therefore the 7th order Maclaurin expansion of $\sin x$ meets the appropriate requirements:

$$T_N(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} \approx \sin x \tag{4.11}$$

Exercises

Exercises for 4.3 Calculating Error

P4.1 The irrational value π can be approximated using the generating function:

$$\pi = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

What are the absolute and relative errors using a 0th, 1st, and 2nd order approximation?

P4.2 You are developing a new radiocarbon dating method. Your test sample is an Egyptian papyrus document that has been officially dated at 2560BCE. Your method estimates the date as 2830BCE. What is the relative error of your method (in percent), assuming that the official date is correct?

Exercises for 4.4 Rounding Error

P4.3 Consider calculating the following summation:

$$x = \sum_{n=0}^{N} \frac{n\pi}{N}\left(\sin^2 x\right)$$

If $N = 100$, what is the result if x stores the running total of the summation and is limited to 3 digits of precision?

Exercises for 4.5 Truncation Error

P4.4 What error would we expect for $x = 0.5$ or $x = 2$ given the implementation of $\sin x$ in Equation 4.11?

P4.5 How many Taylor series terms are required to calculate e^x for $x \leq 1$ with an error bound of $\epsilon \leq 1 \times 10^{-12}$?

P4.6 Design an algorithm that returns an Nth order Taylor series approximation of $\cos x$ for any x (in radians). Make it as simple as you can.

Chapter 5

Numbers in Digital Systems

A *numerical system* is a general system used for writing and expressing numbers. Numerical systems can be summarized by their *basis*, which is the number of different values that can be represented with a single *digit*. Humans have generally adopted *decimal* numbers for common arithmetic operations. Decimal is a base-10 number system, since a single decimal digit can have 10 distinct states: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

In computing, we often use multiple numerical bases as a matter of convenience. Most output is given in decimal for human interpretation. All values are internally represented using base-2, or *binary*. This is a requirement of the underlying digital hardware. There are also specialized uses for hexadecimal (base-16) and octal (base-8).

5.1 Binary Numbers

Binary is the most common numerical basis encountered in computing. A single binary digit is referred to as a *bit* and can take on two values: 0 and 1. This representation is convenient because it allows digital hardware to store values and perform calculations.

Since we expect the reader to be comfortable with decimal representation, we will generally use it to illustrate concepts in digital systems. However, it is important to understand that the fundamental representation used by these systems is binary, since side-effects of this representation will creep up from time to time in our studies. For now, it is important to understand that all of the necessary calculations can be performed in both systems, and that any analogies that we propose between the two are trustworthy.

In order to avoid confusion when working with multiple number systems, the basis is often specified explicitly as a subscript. The binary number 1101 and its corresponding decimal value are therefore represented as $1101_2 = 13_{10}$.

5.1.1 Convert to Decimal

Converting between a numerical basis b_0 and a destination basis b_1 can be performed using the following steps:

1. Initialize a base-b_1 value $x = 0$
2. For each digit d in the input base-b_0 value:
   (a) Convert d to base b_1
   (b) Multiply x by b_0 and add d

This algorithm is implemented in Listing 5.1

Listing 5.1 (to-decimal.h)

```
#include <cmath.h>
int bin2dec(int v){                   //binary value v
    int N = (int) log10(v) + 1;       //get the number of binary digits
    int b = 2;                        //set the base
    int x = 0;                        //initialize the base-10 value
    for(int i = 0; i < N; i++)        //for each digit
        x = x * b + D[i];             //multiply base and add the digit
    return x;                         //return the decimal value
}
```

In the early 1960s, applying this algorithm to binary numbers used to be referred to as *double dabble*: Starting with a "register" value $x = 0$, visit each binary digit. If the digit is a 0, double x. If the digit is a 1, dabble x (double and add +1). For example, the binary value 10110_2 is sequentially evaluated at each digit:

$$\begin{array}{ccccc} 1 & 0 & 1 & 1 & 0 \\ 0\times2+1=\mathbf{1} & 1\times2=\mathbf{2} & 2\times2+1=\mathbf{5} & 5\times2+1=\mathbf{11} & 11\times2=\mathbf{22} \end{array}$$

Example 5.1

Convert the binary value 1100011 to base-10.

Apply the double-dabble algorithm:

$$\begin{array}{ccccccc} 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 3 & 6 & 12 & 24 & 49 & 99 \end{array}$$

$$1100011_2 = 99_{10}$$

Example 5.2

Convert the octal (base-8) value 11753 to base-10.

Apply the conversion algorithm from Listing 5.1:

$$\begin{array}{ccccc} 1 & 1 & 7 & 5 & 3 \\ 1 & 1\times8+1=9 & 9\times8+7=79 & 79\times8+5=637 & 637\times8+3=5099 \end{array}$$

$$11753_8 = 5099_{10}$$

In order to understand how numbers are represented in digital systems, it is also important to understand fractional binary values. Just as in decimal, binary values can be used to represent numbers less than 1. The decimal value 6.25 is expressed fractionally as $6\frac{1}{4}$, where the integer and fractional components are separated by a *decimal point*. The same value in binary can be expressed as $6.25_{10} = 110.01_2$.

First of all, the term "decimal point" would obviously be confusing when working outside of the base-10 number system. The more general term for the "dot" separating integer and fractional values is a *radix point*. When specifically applied to binary numbers, the radix point can naturally be referred to as a *binary point*.

Conveniently, the conversion algorithm described previously can be used to convert fractional binary to decimal values with minimal modification. We simply continue evaluating all of the digits and divide by b^n where n is the number of values after the radix point:

$$\begin{array}{cccccc} 1 & 1 & 0 & . & 0 & 1 \\ 1 & 3 & 6 & & \frac{12}{2} & \frac{25}{4} \end{array}$$

$$110.01_2 = \left(\frac{25}{4}\right)_{10} = 6.25_{10}$$

When doing this calculation mentally (or on paper), it is actually more convenient to evaluate the integer and fractional terms separately. For example, converting 1010.1011_2 to decimal yields an integer part of:

$$\begin{array}{cccc} 1 & 0 & 1 & 0 \\ 1 & 2 & 5 & \mathbf{10} \end{array}$$

and a fractional part of:

$$\begin{array}{cccc} .1 & 0 & 1 & 1 \\ \frac{1}{2} & \frac{2}{4} & \frac{5}{8} & \mathbf{\frac{11}{16}} \end{array}$$

$$1010.1011_2 = \left(10\frac{11}{16}\right)_{10} = 10.6875_{10}$$

5.1.2 Convert from Decimal

When considering the conversion of decimal values to binary, the first option is to use the conversion algorithm from the previous section (5.1.1). While the answer to this is "yes," it is difficult to apply the algorithm by hand. This is because the average student is not trained in binary arithmetic. For example, the first step in converting the value 35_{10} to binary is the conversion of $3_{10} = 11_2$. The second step involves multiplying by $b = 10$, which is $10_{10} = 1010_2$ in binary. Calculating $10_2 \times 1010_2$ is not something that most students are comfortable doing by hand. We will cover this type of calculation briefly in the next section.

In the mean time, we will discuss one easier method to perform a conversion from binary. This method, known as the *remainder method*, relies on performing iterative division of the original value, in base b_0 by the target base b_1:

```
1. Assign the input to a base-$b_0$ value $x$
2. Initialize an empty list of base-$b_1$ digits $D$
3. While $x \neq 0$
    (a) Calculate $x \div b_1$
    (b) Store the quotient in $x$
    (c) Add the remainder to $D$
4. Output the digits of $D$ in reverse order
```

This algorithm is implemented in Listing 5.2 for the conversion of a binary value to base-10.

Listing 5.2 (from-decimal.h)

```
#include <cstdlib.h>
int dec2bin(int x){                  //decimal value x
    int b = 2;                       //output base
    int v = 0;                       //initialize the target result
    int i = 0;                       //current binary digit
    div_t r;                         //structure stores quotient and remainder
    while(x != 0){                   //loop for all binary digits
        r = div(x, b);               //calculate quotient and remainder
        x = r.quot;                  //assign the new quotient to x
        v += r.rem * pow(10, i);     //add the remainder to the binary value
        i++;                         //increment the digit counter
    }
    return v;                        //return the binary result
}
```

Example 5.3

Convert the decimal value 817 to binary.

Apply the remainder method:

$$\begin{array}{rl}
2\,\overline{)817} & 1 \\
2\,\overline{)408} & 0 \\
2\,\overline{)204} & 0 \\
2\,\overline{)102} & 0 \\
2\,\overline{)51} & 1 \\
2\,\overline{)25} & 1 \\
2\,\overline{)12} & 0 \\
2\,\overline{)6} & 0 \\
2\,\overline{)3} & 1 \\
2\,\overline{)1} & 1
\end{array}$$

$$817_{10} = 1100110001_2$$

We can validate this result using the double dabble method:

1	1	0	0	1	1	0	0	0	1
1	3	6	12	25	51	102	204	408	**817**

5.1.3 Binary Calculations

This final section on binary numbers provides a few examples of binary arithmetic. Mathematical operations in binary follow the same process as for decimal numbers.

Addition and subtraction are performed using the same rules, with the understanding that 0, 1, and 2 in decimal correspond to 0, 1, and 10 in binary. For example, adding $1 + 1 = 10$ in binary results in "carrying the one" to the more significant position (Figure 5.2).

Fractional numbers using a radix point are are manipulated the same way. Values are lined up at the radix point and added. Finally, note that all operations on binary values provide the same results as the corresponding decimal calculation (Figure 5.1). One could just as readily calculate the values in decimal and convert the final result to binary.

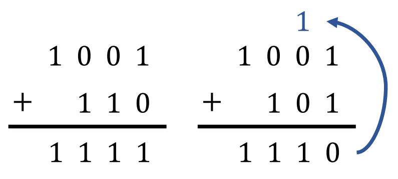


Figure 5.1 Binary addition (left) obeys the same rules as decimal addition. When two 1 values are added (right), the result of $1+1 = 10$ results in "carrying the 1" to a more significant position.

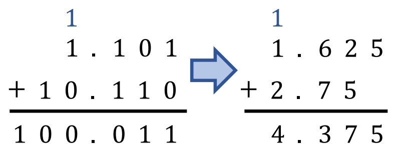


Figure 5.2 When a radix point is used (left) values are lined up at the point and added. The operands and result represent values identical to the corresponding decimal numbers.

5.2 Integer Registers

Now that we understand how to effectively "translate" between our own base-10 numerical system and that of a binary computer, we will look at how numerical values are actually stored.

Every value in a digital system is stored in a *register*, which is made up of a series of electrical components (bits) capable of representing two distinct values (mapped to 0 and 1). The size of the register indicates how many individual bits of binary data can be stored. In most modern electronics, register sizes are in increments of 8 bits - often referred to as a *byte*. For example, a 32-bit register can store 32 binary values amounting to $32/8 = 4$ bytes of data.

5.2.1 Unsigned Registers

The simplest way to store a numerical value inside of a register is to directly map bits to the corresponding binary number. A 16-bit register therefore provides 16 "slots" that can be used to store individual binary bits. In this case, the maximum *decimal* value that can be stored in a 16-bit register is represented by the case where all 16 bits are set to 1:

$$1111111111111111_2 = 65535_{10}$$

A 16-bit register can therefore represent every decimal value in the range $[0, 65535]$. More generally, an n-bit register can represent 2^n integer values in the range of $[0, 2^n - 1]$. When using this type of representation, we are limited to the set of natural numbers $\mathbb{N}$ (positive integers) within the specified range. Representing values that are outside of the set of natural numbers $\mathbb{N}$ requires a more complex representation.

There are several applications for natural numbers in algorithms. In programming languages, such as C/C++, the user can declare data types that use this representation to store values. Common data types that use this format are:

- `unsigned char` - 8-bit integer in the range $[0, 255]$
- `unsigned short` - a 16-bit register (at least) that can store a value in the range $[0, 65535]$ (calculated above)
- `unsigned` (or `unsigned int`) - at least a 16-bit register, but usually a 32-bit register on modern desktop systems
- `unsigned long` - at least a 32-bit register
- `unsigned long long` - at least a 64-bit register
- `size_t` - data type used for addressing, so the size depends on the maximum addressable memory location on the system

5.2.2 Signed Registers

A simple modification of this representation allows us to also represent negative values. We do this by reserving a single bit that will be used to identify the stored value as negative (1) or positive (0). This bit is generally referred to as a *sign bit*, and is most commonly found in the most significant position. The register uses the remaining bits to represent the magnitude of the numerical value using one of two formats: *one's complement* or *two's complement*.

One's complement - A signed bit of 1 indicates that the number is negative and the magnitude is calculated by apply an OR operation to each remaining bit:

$$\begin{aligned} 11111010_2 &= & -0001010_2 &= -5_{10} \\ 11011101_2 &= & -100010_2 &= -34_{10} \end{aligned}$$

While one's complement is relatively easy to understand and convert by hand, it is less frequently used in digital systems because it introduces additional challenges when performing mathematical operations, such as addition, and has a double representation for zero (a *negative zero*):

$$\begin{aligned} 00000000_2 &= & 0000000_2 &= 0_{10} \\ 11111111_2 &= & -0000000_2 &= -0_{10} \end{aligned}$$

Two's complement - The two's complement is computed by taking the one's complement and adding 1_2 to the magnitude:

$$\begin{aligned} 11111010_2 &= & -(000101_2 + 1_2) &= -000110_2 &= -6_{10} \\ 11011101_2 &= & -(100010_2 + 1_2) &= -100011_2 &= -35_{10} \end{aligned}$$

This is by far the most common digital representation for signed integers, since all fundamental operations (addition, subtraction, multiplication) are identical to signed arithmetic. This method also eliminates the negative zero, enabling use of the full range of numbers $\left[-2^{n-1}, 2^{n-1} - 1\right]$.

Integer data types in C/C++ that use sign bits are generally referred to as *signed integers*. This is the default type allocated when the variable is not explicitly declared as `unsigned`. The supported data types for signed values are `char`, `short`, `int`, `long`, and `long long`. Their register sizes correspond to the same number of bits specified in Section 5.2.1.

5.3 Floating Point

Integers $\mathbb{I}$ and natural numbers $\mathbb{N}$ serve several useful purposes in numerical algorithms. However, most calculations require the use of fractional values in larger sets, such as real $\mathbb{R}$ and complex $\mathbb{C}$ numbers. These values are represented digitally in fixed-sized registers using the *floating point* representation.

Floating point numbers are very closely related to scientific notation, which provides a way to represent extremely large and extremely small values that would be impractical to represent using a standard radix point. Floating point values are composed of two major components: a *mantissa* and an *exponent* (Figure 5.3).

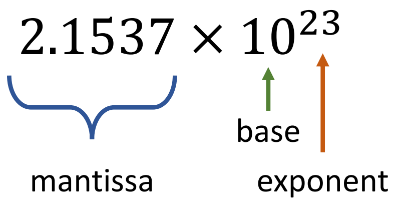


Figure 5.3 Floating point numbers are composed of a normalized mantissa m multiplied by a numerical base b raised to the power of an exponent x. This allows the representation of both very large and very small values using a fixed number of digits. In this example, the 4-digit mantissa and 2-digit exponent can represent values in the range of $[0.1111, 0.9999 \times 10^{99}]$.

Floating point values are represented on digital systems internally using base-2, however this is not a requirement in general. We will therefore explore the basic concepts behind floating point using base-10 mathematics and reserve some of the base-2 idiosyncrasies for later.

5.3.1 Normalization

The first constraint that we apply is a requirement that the floating-point mantissa be *normalized*. The general expression for a floating point value can be expressed as:

$$m \times b^x$$

where m is the mantissa, b is the base, and x is the exponent. The normalization constraint forces the mantissa to be within the range:

$$1 \leq m < b$$

In general terms, this means that:

- The radix point always follows the first digit of the mantissa.
- The first digit of the mantissa is always non-zero.

This constraint forces a unique representation for every real value. Without the normalization constraint the value $f = 3.14_{10}$ could be represented using multiple values of m and x:

$$f = 3.14 \times 10^0 \quad f = 31.4 \times 10^{-1} \quad f = 0.0314 \times 10^2$$

Normalization limits the floating point representation to one possible mantissa and exponent: $m = 3.14$ and $x = 0$.

5.3.2 Spacing

Storing a floating point value digitally requires the mantissa and exponent to be of finite length. We therefore assign a fixed number of digits to m and x. For example, if we are using a base-10 floating point representation and assign 3 digits to the mantissa and 2 digits to the exponent, the largest possible value that can be represented with this format is:

$$9.99 \times 10^{99}$$

In addition to limiting the range of possible values, the fixed mantissa digits also limits the spacing between values. If we wanted to represent the value $\pi \approx 3.14159$ with a 3-digit mantissa, the closest possible values are:

$$\begin{aligned} 3.14 \times 10^0 &= \quad 3.14 \\ 3.15 \times 10^0 &= \quad 3.15 \end{aligned}$$

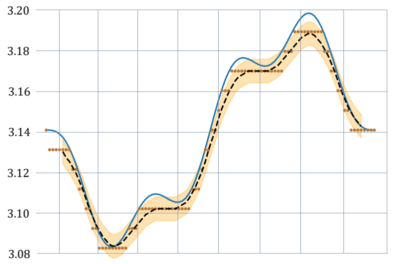


Figure 5.4 A function (blue) sampled using a floating point representation with a 3-digit mantissa (orange). The black line shows the mean of the sampled values, indicating a downward bias caused by rounding. The orange band shows the uncertainty in represented values. The use of floating point in digital systems requires allocating a fixed number of digits to both the mantissa m and exponent x. The number of digits devoted to the exponent set the range of representable values, while the number of digits in the mantissa determine the spacing between values. When calculated values do not fit into the mantissa, the lowest precision digits are chopped off. This results in a loss of accuracy caused by both the limited precision and bias introduced due to chopping.

This discrepancy is generally resolved by *chopping* any digits that do not fit within the mantissa, starting with those that are least significant. This introduces two errors into our calculations:

1. **imprecision:** the spacing between floating point numbers means that we can often only represent values on either side of the desired result

2. **bias:** chopping the least significant figures to force them into the mantissa introduces a downward bias to all of our calculations

Another feature of the fixed mantissa size combined with a variable exponent is that the spacing between floating point values is not constant. Consider the case of a 3-digit mantissa. If $x = 1$, the spacing between sequential numbers is:

$$1.01 \times 10^1 - 1.00 \times 10^1 = 0.1$$

However, if the exponent is changed to $x = 4$, the spacing increases dramatically:

$$1.01 \times 10^4 - 1.00 \times 10^4 = 100$$

If we calculate the difference between any two adjacent numbers with the same exponent, we will find that the spacing is equal. The spacing between numbers is therefore dependent only on the value of x and the number of digits in the mantissa.

The fact that floating point values change precision with the exponent is actually one of the primary advantages of the format. Floating point provides us with high precision when working with small numbers and lower precision when working with large numbers. Floating point is a numerical representation designed to minimize the *relative error* of our numerical calculations (Section 4.3.2).

This variable spacing produces some artifacts that we must be aware of when designing algorithms. The first is that there will be a significant change in spacing between values whenever the exponent changes. Consider the number lines shown in Figure 5.5. The spacing between base-10 values represented using base-b floating point will increase by a factor of b every time the exponent is increased. In general, all of the necessary tools for handling these shifts are taken care of by the hardware. However, there is one extremely important problem that can cause serious problems in our algorithms if we aren't careful: chopping.

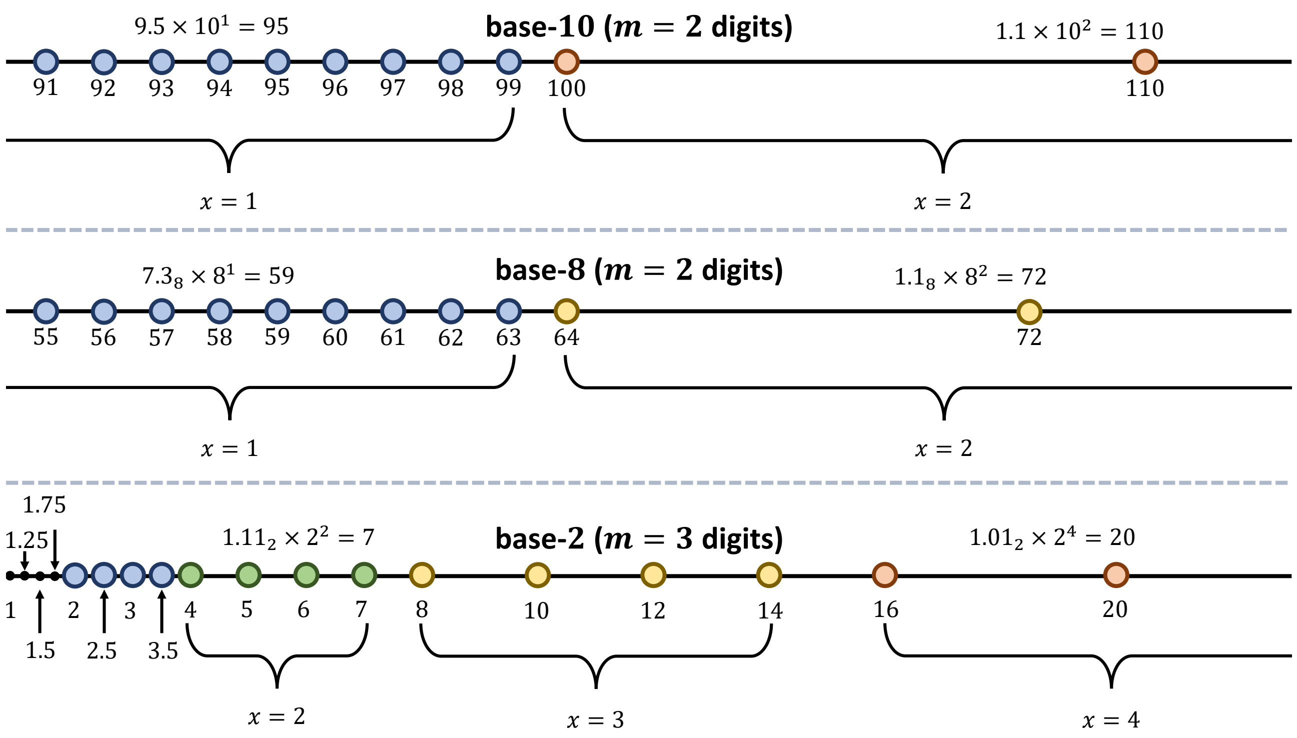


Figure 5.5 Number lines showing the spacing between values in floating point representations in base-10, base-8, and base-2 (binary). Note that the transition between exponents is more gradual for smaller bases. The binary floating point format shown in this figure, using a 3-bit mantissa, is a traditional 8-bit floating point implemented in some digital systems.

5.3.3 Rounding

When two floating point values are added or subtracted, they must be lined up at the same radix point. Essentially, they must be briefly transformed to the same exponent during the operation. This almost always results in a loss of information when the two numbers have different exponents. Consider the addition $c = a + b$ using base-10 floating point and a 3-digit mantissa, where $a = 237$ and $b = 23.6$. The corresponding floating point representation for these two values are:

$$a = 2.37 \times 10^2 \quad b = 2.36 \times 10^1$$

Lining up the radix point produces a result that has greater precision than could be represented using our floating point representation:

$$\begin{array}{rl} & 237.0 \\ + & 23.6 \\ & 250.6 = 2.506 \times 10^2 \end{array}$$

While both operands a and b require 3-digit precision, the result c requires a 4-digit mantissa. Digital systems will address this by simply chopping the least-significant digits until the result can be sufficiently represented. In this case, the result of the subtraction will be $c = 2.50 \times 10^2$.

The chopping of least significant digits is the rounding method used in digital systems. In fact, it's the only method that can be used. Traditional rounding methods used by hand - most commonly rounding up if the least significant digit is ≥ 5 and rounding down otherwise - would actually require the calculation of an additional digit. In other words, we would have to use larger registers to perform the calculation and store the result after rounding. The use of a larger intermediate register, often called an *accumulator*, can be found in some types of computing hardware.

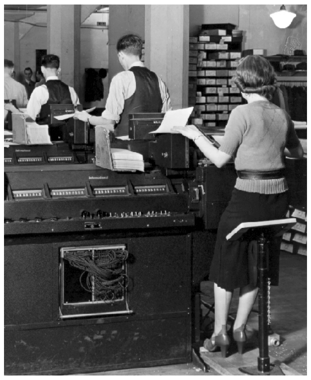


Figure 5.6 Accumulators were often used to store intermediate results of a calculation before it was offloaded to a register for further calculation or output. This method allows increased precision in a sequential set of arithmetic operations when rounding would cause significant error. (Wikipedia).

On most numerical systems, this chopping is generally difficult to notice with a single calculation since current floating point formats provide a fairly large mantissa. One exception, discussed in Chapter 6, is the significant loss of precision that can occur with certain types of subtraction. Iterative algorithms also tend to compound these small arithmetic errors. Therefore particular care must be taken when designing loops, particularly when the data is modified every iteration. The following example provides an example of a particularly common mistake when using floating point numbers in loops.

Example 5.4

Assume that we are using a base-10 floating point representation with a 4-digit mantissa. How many times will the following loop execute?

```
$x = 0$
$h = 0.01$
while $x < 200$
    ...perform some operation using $x$
    $x = x + h$
```

Intuitively, we would expect $200/0.01 = 20000$ iterations. This is likely what the programmer intends. However, consider what happens when the value of x reaches $x = 100$. In the specified floating point representation, the next iteration of the loop will require the following sum:

$$\begin{array}{rl} & 100.00 \\ + & 0.01 \\ & 100.01 = .10001 \times 10^3 \rightarrow .1000 \times 10^3 \end{array}$$

In fact, this loop will execute forever and the value of x will never exceed $x = 100$.

Because of this trivial example, the authors recommend against making a loop termination conditional on a floating point test.

In addition, here are some options for addressing this common problem:

(a) Calculate the number of iterations before entering the loop and calculate the value of x every iteration:

```
$h = 0.01$
$N = 200/h$
$i = 0$
```

```
        while $i < 200$
            ...perform some operation using $i \times h$
            $i = i + 1$
```

(b) In cases where (1) cannot be used, another option is to detect the error and exit. In that case you can add a conditional before x is incremented to exit the loop if $x = x + h$.

Finally, it is often useful to quickly calculate the spacing between floating point values, as well as determine error bounds for any arithmetic operation. These are useful tools for determining exactly how much numerical precision is required to solve a problem so that an appropriate floating point format can be selected. One simple method for characterizing these features for a given floating point system is by determining the *machine epsilon*.

Machine Epsilon

The machine epsilon, represented as ϵ, is a value associated with a floating point system that gives the maximum relative error introduced by rounding.

Specifically, the machine epsilon is the smallest value for which $1 + \epsilon \neq 1$ and can be calculated as:

$$\epsilon = b^{-(p-1)} \tag{5.1}$$

where b is the numerical base for the floating point representation and p is the number of base-b digits in the mantissa.

The machine epsilon can be used to calculate the *maximum* spacing h between any floating point value n and its neighbor:

$$h_{max} = 2\epsilon|n| \tag{5.2}$$

Note that the error due to precision and rounding for values near n will be:

$$h = \frac{2\epsilon|n|}{2} \tag{5.3}$$

Example 5.5

Calculate the spacing between values for a base-2 floating point number with a 3 bit mantissa and an exponent of $x = 3$.

This can be accomplished two ways. Both require selecting a value in the specified range. We'll select the smallest representable number with $x = 3$:

$$y_0 = 1.00_2 \times 2^3 = 1000_2 = 8_{10}$$

(a) One method is to simply select the next adjacent value:

$$y_1 = 1.01_2 \times 2^3 = 1010_2 = 10_{10}$$

and calculate the difference:

$$h = 10_{10} - 8_{10} = 2_{10}$$

(b) The second method requires calculating the machine epsilon:

$$\epsilon = b^{-(p-1)} = 2^{-2} = 0.25$$

followed by the spacing (Equation 5.3):

$$h = \frac{2\epsilon|y_0|}{2} = \frac{2(0.25)|8|}{2} = 2_{10}$$

5.4 Floating Point Registers

Now that you understand the underlying principals behind the floating point representation, we will confront a few specifics used in practical applications of this standard. As mentioned previously, all numerical values are stored in binary registers of a specified size. Therefore all floating point values implemented on digital hardware use a base-2 representation.

Anyone who has made extensive use of floating point as a programmer may have noticed that rounding doesn't actually occur the way one would expect. In fact, the authors have frequently had to consider "how many digits of precision" are offered by a 32-bit floating point value. It's important to note that this question doesn't actually make sense in a normal context of base-10 arithmetic because all rounding occurs in binary. Likewise, the "digits of precision" are also represented in base-2. However, once you understand how floating point works "under the hood" you can readily answer these types of questions in a meaningful way. In addition, this representation provides us with several practical advantages.

- **Implied 1s:** The underlying binary representation provides us with a unique benefit that could not be leveraged in any other floating point system. Recall that the normalization constraint described in Section 5.3.1 requires that the leading digit be non-zero. One interesting consequence of base-2 is that there is only one non-zero value. We can therefore assume that the first digit in the mantissa is always 1 and thereby gain an additional bit of precision! This binary shortcut is generally referred to as an *implied 1*, and should be taken into consideration when calculating the number of values that can be represented in the mantissa.

- **Signed Bits:** Most programming languages expect floating point numbers to be able to take on both positive and negative values. A signed bit is therefore introduced as the most significant bit to designate the value as negative (1) or positive (0).

- **Exponent Bias:** In order to represent small fractional values, it is critical to be able to allow the exponent x to drop below 0. In order to allow this, floating point formats apply an internal bias to the exponent in the form of a subtracted constant. As an alternative, an additional signed bit could also be used. However, the bias option is more efficient in practice and allows us to use the full range of the exponent value.

With these modifications, a more practical binary floating point representation can be specified as:

$$(-1)^s 1.m_2 \times 2^{x-c} \tag{5.4}$$

where s is the signed bit value, m is a series of bits storing the mantissa, x is the exponent, and c is a bias constant specified in the representation. Any future exercises related to binary floating point should assume this format.

There are several floating point standards that are set by the ISO, with three data types generally supported by C/C++:

- `float` - usually a 32-bit register, storing 1 signed bit, an 8-bit exponent with a bias of $c = 128$, and 23 explicit bits for the mantissa (making it a 24-bit mantissa with the implied 1).
- `double` - usually a 64-bit register storing 1 signed bit, an 11-bit exponent with a bias of $c = 1023$, and 52 explicit bits for the mantissa.
- `long double` - if implemented, it is usually a 128-bit register storing 1 signed bit, an 15-bit exponent with a bias of $c = 262143$, and 112 explicit bits for the mantissa.

There is also an IEEE-754 standard supporting 16-bit floating point. While not generally implemented in C/C++, it is used on some architectures (such as the nVIDIA CUDA programming language):

- `half` - 16-bit register storing 1 signed bit, an 5-bit exponent with a bias of $c = 15$, and 10 explicit bits for the mantissa.

The main advantage to using a smaller floating point representation is speed. Larger registers usually require computing power proportional to their size. To a first approximation, it requires twice as much time to evaluate a 64-bit floating point value when compared to a 32-bit value.

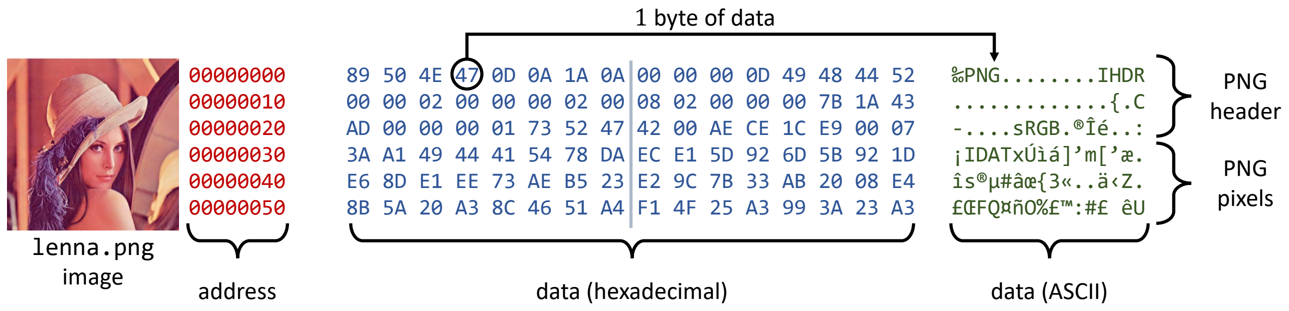
5.5 Hexadecimal

Hexadecimal, or base-16 is frequently encountered in computing when visualizing binary data, since it enables easier interpretation of individual bytes. Unlike previous bases that we have discussed, which can be represented using traditional base-10 digits, hexadecimal digits can have 16 different values. In order to represent these additional values, we incorporate the numeric symbols $0-9$ and the alphabetic symbols $A-F$. Therefore:

$$5_{10} = 5_{16}$$
$$10_{10} = A_{16}$$
$$28_{10} = 1C_{16}$$

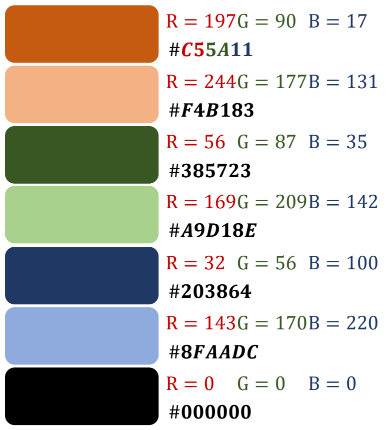
Note that the largest two-digit value in hexidecimal:

$$FF_{16} = 255_{10} = 11111111_2$$



```
00000000  89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52  ‰PNG........IHDR
00000010  00 00 02 00 00 00 02 00 08 02 00 00 00 7B 1A 43  .............{.C
00000020  AD 00 00 00 01 73 52 47 42 00 AE CE 1C E9 00 07  -....sRGB.®Îé..:
00000030  3A A1 49 44 41 54 78 DA EC E1 5D 92 6D 5B 92 1D  ¡IDATxÚìá]’m[’æ.
00000040  E6 8D E1 EE 73 AE B5 23 E2 9C 7B 33 AB 20 08 E4  îs®µ#âœ{3«..ä‹Z.
00000050  8B 5A 20 A3 8C 46 51 A4 F1 4F 25 A3 99 3A 23 A3  £ŒFQ¤ñO%£™:#£ êU
```

Figure 5.7 Hexadecimal data dump for a portable network graphics (PNG) image. Most hex dumps display the memory address (red) along with the data in both hexadecimal (blue) and ASCII (green). The hexadecimal values represent each byte in the data as two digits. The corresponding ASCII (with control characters represented as '.') makes it easier to identify strings embedded in the data. For example, the "PNG", "IHDR", and "sRGB" components of the header file can be seen in the ASCII code, while the command codes 89 (at byte 00000000) and the cross-platform line ending sequence ($0D$ $0A$ $1A$ $0A$) starting at 00000004. Software to explore hex dumps of files is available online at hexed.it.



🟧	R = 197	G = 90	B = 17	**#*C*55*A*11**
🟧	R = 244	G = 177	B = 131	**#*F*4*B*183**
🟩	R = 56	G = 87	B = 35	**#385723**
🟩	R = 169	G = 209	B = 142	**#*A*9*D*18*E***
🟦	R = 32	G = 56	B = 100	**#203864**
🟦	R = 143	G = 170	B = 220	**#8*FAADC***
⬛	R = 0	G = 0	B = 0	**#000000**

Figure 5.8 Hexadecimal offers a convenient representation for 24-bit color images using a 2-digit representation for each channel (red, green, blue). Every pair of digits represents a single bit, and can take a value in the range 0_{10} to 255_{10}. HTML Color Codes

corresponds to 1 byte (8 binary bits). Two-digit symbols in hexidecimal are often used to represent sequences of bytes in binary. So, an array of binary values:

$$1101 \quad 1011 \quad 0010 \quad 1101 \quad 1010 \quad 0000 \quad 0111 \quad 0101 \quad 1100 \quad 1111$$

is most often written using the corresponding series of hexadecimal pairs:

$$DB \quad 2D \quad A0 \quad 75 \quad CF$$

The hexadecimal representation is more generally preferred because humans can more readily adapt to patterns in this format - English speakers are already used a single digit representing a larger number of alphanumeric values.

Hexadecimal Color Codes. One of the most common uses of hexadecimal values in programming is the specification of color values in several programming languages, such as HTML (Figure 5.8).

Hexadecimal Data Dumps. Hex dumps are often used in computing to look at the contents of memory, registers, or files. They provide a convenient means of looking at raw data to make sure that values are being stored or interpreted correctly (Figure 5.7).

5.6 Endianness

The *endianness* of a digital system specifies how values are stored in memory, transferred during communication, or saved to disk. The two most important endian formats are:

- **Big-Endian** - The bytes are stored in the system in the same order that they would be interpreted by a user. The hexadecimal value $1A$ $2B$ $3C$ $4D$ is interpreted such that $1A$ is the most significant byte and $4D$ is the least significant byte.

- **Little-Endian** - Bytes are stored with the least significant digit in the first. The hexadecimal value $1A$ $2B$ $3C$ $4D$ would be stored in memory as:

$$\begin{array}{rcccc} \text{address:} & 00 & 01 & 02 & 03 \\ \text{value:} & 4D & 3C & 2B & 1A \end{array} \tag{5.5}$$

Endianness is an extremely common practical problem, and is often a major source of confusion the first time a programmer attempts to process binary data at the byte level. Interestingly, **the only reason we have to talk about endianness is because most systems that you will encounter will store data in the little-endian format.** Which is to say that most values you look at in a raw data dump will appear backwards. Note that the *binary* data does not appear backwards. The byte order is reversed. For example, given the binary version of the example above:

hexadecimal:	$1A$	$2B$	$3C$	$4D$
binary:	0001 1010	0010 1011	0011 1100	0100 1101

will appear in memory as:

address:	00	01	02	03
value:	0100 1101	0011 1100	0010 1011	0001 1010

In addition, this byte swapping only occurs for each independent numerical value. For example, consider two adjacent values: a 32-bit floating point number ($0A$ $0B$ $0C$ $0D$) and a short integer ($1E$ $2F$). In a little-endian system, these two values would be stored in memory as:

$$\begin{array}{rcccccc} \text{address:} & 00 & 01 & 02 & 03 & 04 & 05 \\ \text{value:} & 0D & 0C & 0B & 0A & 2F & 1E \end{array}$$

The individual values are in the expected order, however the bytes representing each value are reversed.

For a more practical example, see the implementation in Program 5.3. This C code generates a series of values of different sizes and writes them directly to a binary file. If the file is read using a hex dump or any debugging software, the values will appear as shown in Figure 5.9.

Most digital systems, including desktop computers and cluster computers, all use the little-endian format. Supercomputers vary depending on manufacturer. Most networking protocols, such as TCP, IPv4, and IPv6 use big-endian to transfer data. If you expect to be working with data on widely varying platforms, it is recommended that you test for endianness and reverse the byte order when necessary. This can be done by implementing a header that writes a single value (ex. `short`). The value can then be read as individual bytes (ex. 2x `char`) on the destination system. Depending on the byte order of this *test value*, the byte order of stored values can be read directly or swapped.

```
00000000     7B 39 30 D2 02 96 49 15|81 E9 7D F4 10 22 11 DB   {90Ò.–I.é.}ô.“.Û
00000010     0F 49 40 CD 3B 7F 66 9E|A0 F6 3F                  .I@Í;.fž.ö?
```

`00000000: 7B` $7B_{16} = 0111\ 1011_2 = \mathbf{123_{10}}$

`00000001: 39 30` $30\ 39_{16} = 0011\ 0000\ 0011\ 1001 = \mathbf{12345_{10}}$

`00000003: D2 02 96 49` $49\ 96\ 02\ D2_{16} = 0100\ 1001\ 1001\ 0110\ 0000\ 0010\ 1101\ 0010_2 = \mathbf{1234567890_{10}}$

`00000007: 15 81 E9 7D F4 10 22 11`
$$\begin{aligned}11\ 22\ 10\ F4\ 7D\ E9\ 81\ 15_{16} &= 0001\ 0001\ 0010\ 0010\ 0001\ 0000\ 1111\ 0100\\ &\quad 0111\ 1101\ 1110\ 1001\ 1000\ 0001\ 0001\ 0101_2\\ &= \mathbf{1234567890123456789_{10}}\end{aligned}$$

`0000000F: DB 0F 49 40`
$$\begin{aligned}40\ 49\ 0F\ DB_{16} &= 0100\ 0000\ 0100\ 1001\ 0000\ 1111\ 1101\ 1011\\ &= \mathbf{3.1415927}\end{aligned}$$

`00000013: CD 3B 7F 66 9E A0 F6 3F`
$$\begin{aligned}3F\ F6\ A0\ 9E\ 66\ 7F\ 3B\ CD_{16} &= 0011\ 1111\ 1111\ 0110\ 1010\ 0000\ 1001\ 1110\\ &\quad 0110\ 0110\ 0111\ 1111\ 0011\ 1011\ 1100\ 1101_2\\ &= \mathbf{1.414213562373095}\end{aligned}$$

Figure 5.9 Hex dump produced by Program 5.3 showing the values at each memory location in little-endian format, along with the correct hexadecimal, binary, and decimal values. A detailed description of the 32-bit floating point conversion is given in Example 5.6 and the 64-bit floating point conversion is left as an exercise for the reader.

Listing 5.3 (binary-out.c)

```c
#include <stdio.h>                        //file I/O
#include <math.h>                         //required for sqrt() and M_PI

int main(int argc, char* argv[]){
        char a = 123;                         //set an 8-bit value
        short b = 12345;                      //set a 16-bit value
        int c = 1234567890;                   //set a 32-bit value
        long long d = 1234567890123456789;    //set a 64-bit value
        float e = M_PI;                       //assign PI to a 32-bit float
        double f = sqrt(2.0);                 //assign sqrt(2) to a 64-bit float
        FILE* file = fopen("output.dat", "wb"); //create a binary output file
        fwrite(&a, sizeof(char), 1, file);      //write each value
        fwrite(&b, sizeof(short), 1, file);
        fwrite(&c, sizeof(int), 1, file);
        fwrite(&d, sizeof(long long), 1, file);
        fwrite(&e, sizeof(float), 1, file);
        fwrite(&f, sizeof(double), 1, file);
        fclose(file);                           //close the output file
}
```

Example 5.6

A 32-bit region of memory stores the following value (see Figure 5.9):

$$DB \quad 0F \quad 49 \quad 40$$

This memory address is the location of an IEEE 32-bit floating point value on an Intel desktop system. What is the decimal value that it represents?

Solution: Since a personal computer (like most digital systems) uses little endian to store values in memory, the sequence of bytes, from most to least significant, is:

$$40 \quad 49 \quad 0F \quad DB$$

which corresponds to the following binary sequence:

$$0100\ 0000\ 0100\ 1001\ 0000\ 1111\ 1101\ 1011$$

The IEEE 32-bit format has the following specification:

- 1 sign bit
- 8-bit exponent with a bias of 127
- 23-bit mantissa with an implied 1

The bits associated with each component are:

$$0100\ 0000\ 0100\ 1001\ 0000\ 1111\ 1101\ 1011$$

- sign: 0 (+) the number is positive
- exponent: $10000000_2 = 128_{10} - 127 = 1$
- mantissa: $100\ 1001\ 0000\ 1111\ 1101\ 1011_2$

Adding the implied 1 and exponent gives us the following binary floating point value:

$$1.10010010000111111011011 \times 2^1 = 11.0010010000111111011011$$

Converting to decimal, we find that the register stores the base-10 value:

$$3\frac{593883}{2^{22}} = 3.1415927410125732421875 \approx \pi$$

Note that this value is only accurate out to 7 decimal digits (3.1415927), even though directly converting the fractional value *appears* to provide significantly more. This is because the machine epsilon for a 32-bit IEEE floating point value is:

$$\epsilon_{32} = \frac{1}{2^{23}} = 1.1920928955078125 \times 10^{-7}$$

Since the exponent for this value is 1, this is proportional to the spacing between adjacent values and defines the limit of our precision.

Exercises

Exercises for 5.2 Integer Registers

P5.1 Convert the following signed (twos-complement) integers to decimal:

1. 0111
2. 1011 0101
3. 1100 0101 1011

P5.2 Calculate the results of the following arithmetic operations using unsigned integers (bit depth is specified):

1. 4-bit registers: 0101 × 0010
2. 4-bit registers: 1011 + 0110
3. 8-bit registers: 1101 0101 × 0000 0101

P5.3 What is the smallest value possible for a 5-bit signed integer?

P5.4 A Nintendo Entertainment System (NES) used a pixel processing unit (PPU) with 246 bytes of addressable sprite memory that could be used to process motion on the screen. How many bits were required to address this memory space?

P5.5 The NeoGeo had 64 kilobytes (kB) of main VRAM. How many bits were required to address this memory space? What is the maximum amount of memory (in bytes) that the NeoGeo could have had with this addressing scheme?

- **Optional:** There is a lot of confusion surrounding different standards for the term *kilobyte*. Look up and describe the difference between the ISO and JEDEC standards. Which do you think is used in this case?

P5.6 The number of seconds given to complete each level L in the video game Donkey Kong is $s = \min(10L + 40,\ 80)$. Why has nobody gotten past level 22?

Exercises for 5.3 Floating Point

P5.7 Convert the following 8-bit floating point values to decimal assuming the following format (with a bias of 7):

0001 0101 →

sign	exponent				mantissa		
0	0	0	1	0	1	0	1

1. 0001 0101
2. 1000 0100
3. 1011 0100

Exercises for 5.6 Endianness

P5.8 Assuming a processor that stores values in the little-endian format, translate the following register values to their digital representation (*Hints are provided to check your answers*):

1.	`unsigned char`	41	53	43	49	49	(ASCII)
2.	`char`	E2					(Cleopatra)
3.	`char`	9C					(Julius Caesar)
4.	`short`	C3	01				(Bradbury)
5.	`short`	D4	05				(Columbus)
6.	`short`	C3	FA				(Tutankhamun)
7.	`int`	4A	78	DE	11		(Light Speed)
8.	`float`	CD	CC	1C	C1		(Gravity)

Chapter 6

Precision Loss

General Notes:

- Loss of precision is due to the limited number of digits available for representing numbers.
- One of the most destructive forms of precision loss is *catastrophic cancellation*, caused by subtracting two numbers that are almost equal.
- Catastrophic cancellation can be mitigated by reformulating expressions or utilizing approximations such as Taylor series or the Mean Value Theorem.

6.1 Catastrophic Cancellation

One of the fundamental challenges encountered in numerical computing is loss of significance during the calculation of numerical expressions. Particular sequences of operations can cause surprisingly severe losses of precision. The most common of these is *catastrophic cancellation*, which occurs when two similar values are subtracted.

Consider the following calculation:

$$f(x) = 1 - \cos x \tag{6.1}$$

implemented on a digital system using IEEE 32-bit floating point precision. Each step of this calculation can be expressed as an algorithm using the following sequence of instructions:

Listing 6.1 (cancellation.h)

```
#include <math.h>
float f(float x){
    float a = 1.0f;             //store 1.0 as a floating point value
    float b = cosf(x);          //calculate a 32-bit cosine
    float c = a - b;            //perform the subtraction
    return c;                   //return the result
}
```

Based on the ISO standard, the machine epsilon for an IEEE 32-bit value is $\epsilon = 2^{-23} \approx 0.1 \times 10^{-8}$. We therefore expect the result of this calculation to have approximately 7 to 8 digits of precision. What we find is that the value returned by this function is significantly less accurate as $x \to 0$.

Let's look at each step in this calculation for two different values of x:

$$x_1 = 1.0$$

$$x_2 = 0.00112$$

Analytically calculating $f(x) = 1 - \cos x$ results in the following correct values up to 8 significant figures:

$$f(x_1) = .45969769 \qquad \text{correct value for } 1 - \cos(1.0) \tag{6.2}$$

$$f(x_2) = .62719993 \times 10^{-6} \qquad \text{correct value for } 1 - \cos(0.00112) \tag{6.3}$$

Stepping through the process of calculation on a digital system using Algorithm 6.1 yields the following register values for a, b, and c when $x = x_1 = 1.0$:

$$a_1 = 1.0000000 \tag{6.4}$$

$$b_1 = .54030228 \tag{6.5}$$

$$c_1 = .45969772 \tag{6.6}$$

Note that c_1 differs from the real value of $f(x_1)$ in the last digit (and the second to last is rounded up). This is within range of what we would expect for our result. However, using Algorithm 6.1 to calculate $f(0.00112)$ results in the following register values:

$$a_2 = 1.0000000 \tag{6.7}$$

$$b_2 = .99999934 \tag{6.8}$$

$$c_2 = .00000066 \tag{6.9}$$

which provide only one correct digit of precision when compared to the expected value of $f(x_2) = .62719993 \times 10^{-6}$. This results in a relative error of

$$E = \left| \frac{0.66 \times 10^{-6} - .62719993 \times 10^{-6}}{.62719993 \times 10^{-6}} \right| \approx 5.2\% \tag{6.10}$$

The reason for this discrepancy is clear when thinking in terms of a digital computer. The register b can only store a finite number of digits from the calculation of $\cos x$. During the final subtraction $c = a - b$, most of those leading digits are cancelled out. At that point there is no mechanism to retrieve additional digits of precision from the $\cos x$ calculation. The subtraction forces previously insignificant digits from the calculation of $\cos x$ to become significant when calculating $1 - \cos x$.

6.2 Quantifying Precision Loss

Catastrophic cancellation using floating numbers can be tightly quantified using the Loss of Precision Theorem. This theorem defines how many digits are cancelled when two numbers are subtracted.

Loss of Precision Theorem

The number of digits lost in a base-b floating-point operation can be quantified using the Loss of Precision Theorem.

Loss of Precision Theorem. If x and y are two positive base-b floating point numbers $x > y > 0$, there are two adjacent natural numbers p and q such that

$$b^{-p} \leq 1 - \frac{y}{x} \leq b^{-q} \tag{6.11}$$

At most p and at least q significant digits are lost in the calculation of $x - y$.

Example 6.1

How many digits of precision are lost in the following subtraction?

$$c = .37593621 \times 10^2 - .37584216 \times 10^2 \tag{6.12}$$

Solution: We are interested in the values of p and q specified in the Loss of Precision Theorem. First, we reformulate the Loss of Precision Theorem such that we can determine values for p and q:

$$b^{-p} \leq 1 - \frac{y}{x} \leq b^{-q} \tag{6.13}$$

$$\log(b^{-p}) \leq \log\left(1 - \frac{y}{x}\right) \leq \log(b^{-q}) \tag{6.14}$$

$$-p \log b \leq \log\left(1 - \frac{y}{x}\right) \leq -q \log b \tag{6.15}$$

$$p \log b \geq -\log\left(1 - \frac{y}{x}\right) \geq q \log b \tag{6.16}$$

$$p \geq \frac{-\log\left(1 - \frac{y}{x}\right)}{\log b} \geq q \tag{6.17}$$

We also know that p and q are (1) natural numbers $(p, q \in \mathbb{N})$, also known as positive integers, and (2) adjacent, such that $p = q + 1$. By evaluating the central term in Equation 6.17, we can determine p and q by calculating the two integers adjacent to the resulting value:

$$q = \left\lfloor \frac{-\log\left(1 - \frac{y}{x}\right)}{\log b} \right\rfloor \tag{6.18}$$

$$p = \left\lceil \frac{-\log\left(1 - \frac{y}{x}\right)}{\log b} \right\rceil \tag{6.19}$$

We then substitute the values from the original problem to determine the number of base-10 digits lost

$$q \le \frac{-\log\left(1 - \frac{.37584216\times10^2}{.37593621\times10^2}\right)}{\log 10} \le p \tag{6.20}$$

$$q \le \frac{-\log(1 - 0.99974982)}{\log 10} \le p \tag{6.21}$$

$$q \le \frac{8.2933482}{2.3025851} \le p \tag{6.22}$$

$$q \le 3.6017553 \le p \tag{6.23}$$

Calculating the floor and ceiling values indicates that at least $q = 3$ and at most $p = 4$ base-10 digits are lost. We can verify this by lining up each of the values, to see that 3 – and almost 4 – leading digits are cancelled in the subtraction:

$$x = .37593621 \times 10^2 \tag{6.24}$$

$$y = .37584216 \times 10^2 \tag{6.25}$$

Since the floating point numbers in a digital system have a mantissa with a fixed number of base-2 bits, the Loss of Precision Theorem can be used to determine how many bits of precision can be expected after a subtraction is performed. Recalculating Example 6.1 for base-2 floating point yields values of $q = 11$ and $p = 12$. If we perform this subtraction on a system using IEEE 32-bit floating point registers, this effectively reduces the mantissa size by half, from 24 bits to about 12.

6.3 Reformulation

The key to mitigating precision loss is to remove operations that result in the cancellation of significant digits. In the case of catastrophic cancellation, the focus should be removing subtractions that can lead to significant precision loss. Consider the example given in Equation 6.1. We can devise a formulation that is far more stable for small values of x:

$$f(x) = 1 - \cos x \qquad \text{(precision loss for small } x\text{)} \tag{6.26}$$

$$= (1 - \cos x)\left(\frac{1+\cos x}{1+\cos x}\right) \qquad \text{(multiply by the conjugate)} \tag{6.27}$$

$$= \frac{1-\cos^2 x}{1+\cos x} \qquad \text{(simplify)} \tag{6.28}$$

$$= \frac{\sin^2 x}{1+\cos x} \qquad \text{(apply trigonometry)} \tag{6.29}$$

In this example, we first multiply the original expression by its conjugate (Equation 6.27). This allows us to replace the expression in the numerator using a trigonometric identity (Equation 6.29).

This final expression eliminates subtractions that could result in cancellation. We can implement and test the new formulation using the following algorithm:

Listing 6.2 (reformulation.h)

```
#include <math.h>
float f(float x){
    float a = sinf(x);          //calculate a 32-bit sine
    float b = cosf(x);          //calculate a 32-bit cosine
    float c = 1.0f;             //store 1.0 as a floating point value
    float d = a * a;            //calculate sin^2(x)
    float e = c + b;            //calculate 1 + cos(x)
    float f = d / e;            //calculate the final result
    return f;                   //return the result
}
```

Stepping through this algorithm, the following results are calculated for each operation:

$$a = .11199997 \times 10^{-2} \tag{6.30}$$

$$b = .99999934 \tag{6.31}$$

$$c = 1.0000000 \tag{6.32}$$

$$d = .12543995 \times 10^{-5} \tag{6.33}$$

$$e = 1.9999992 \tag{6.34}$$

$$f = .62719999 \times 10^{-6} \tag{6.35}$$

The final value in register f is significantly closer to the actual result, with a difference in only the least significant digit and a relative error of

$$E = \left| \frac{.62719999 \times 10^{-6} - .62719993 \times 10^{-6}}{.62719993 \times 10^{-6}} \right| \approx 0.00001\% \tag{6.36}$$

Example 6.2

The quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{6.37}$$

is an extremely useful numerical algorithm for analytically calculating the roots of quadratic equations of the form $ax^2 + bx + c = 0$. When is this formula unstable as a result of precision loss? Can a better algorithm be provided?

Solution: Although subtraction of similar values can occur in the $b^2 - 4ac$ term when $b^2 \approx 4ac$, the resulting value will be extremely small relative to b:

$$\sqrt{b^2 - 4ac} \ll b \quad \text{if} \quad b^2 \approx 4ac \tag{6.38}$$

and will therefore be insignificant in the numerator.

Catastrophic cancellation can occur when $4ac \ll b^2$:

- $-b + \sqrt{b^2 - 4ac}$, when $b > 0$
- $-b - \sqrt{b^2 - 4ac}$, when $b < 0$

The root associated with the cancelled term is extremely small.

Consider the polynomial:

$$x^2 + 182^3 x - 1 = 0 \tag{6.39}$$

which has roots at $x_1 = -.60286 \times 10^7$ and $-.165786 \times 10^{-8}$. However, applying the quadratic formula using 32-bit registers produces the following values:

$$x_1 = -.60286 \times 10^7 \tag{6.40}$$

$$x_2 = -.15 \times 10^{-8} \tag{6.41}$$

resulting in a relative error of

$$E = \left| \frac{-.15 \times 10^{-8} + .165786 \times 10^{-8}}{-.165786 \times 10^{-8}} \right| \approx 9.5\% \tag{6.42}$$

When $b > 0$, the smallest root (x_2) can be reformulated to eliminate the catastrophic cancellation:

$$x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{6.43}$$

$$= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \left(\frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} \right) \tag{6.44}$$

$$= \frac{4ac}{-2a\left(b + \sqrt{b^2 - 4ac}\right)} \tag{6.45}$$

$$= \frac{-2c}{b + \sqrt{b^2 - 4ac}} \tag{6.46}$$

The same reformulation can be applied to x_2 when $b < 0$. Reformulating both possible cases results in a stable conditional formulation that requires testing the sign of b:

- If $b > 0$

$$x_1 = -\frac{1}{2} \left(\frac{b + \sqrt{b^2 - 4ac}}{a} \right) \qquad x_2 = \frac{c}{a x_1} \tag{6.47}$$

- If $b < 0$

$$x_1 = -\frac{1}{2} \left(\frac{b - \sqrt{b^2 - 4ac}}{a} \right) \qquad x_2 = \frac{c}{a x_1} \tag{6.48}$$

The final algorithm used in most software implementations of the quadratic formula further optimizes to reduce the number of required operations:

$$q = -\frac{1}{2} \left[b + \operatorname{sgn}(b) \sqrt{b^2 - 4ac} \right] \tag{6.49}$$

$$x_1 = \frac{q}{a} \tag{6.50}$$

$$x_2 = \frac{c}{q} \tag{6.51}$$

(6.52)

where

$$\operatorname{sgn}(x) = \begin{cases} -1, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \tag{6.53}$$

6.4 Reformulation Using Taylor Series

Taylor series expansion are particularly useful for addressing catastrophic cancellation, since a large part of the eliminated operands can be analytically removed. This effectively replaces precision loss from catastrophic cancellation with truncation error, which can be tightly controlled.

Maclaurin series expansions are the most common, since catastrophic cancellation often occurs when the parameter $x \to 0$. For example, consider the case:

$$1 - e^x$$

where loss of precision will occur when $x \to 0$. Replacing e^x with the corresponding Maclaurin expansion yields:

$$1 - \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 - \left(1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots\right)$$

which allows the first operand to be eliminated analytically. The resulting expression can be calculated to the desired precision without loss of significance:

$$1 - e^x = \sum_{n=1}^{\infty} \frac{x^n}{n!}$$

When calculating a Taylor series expansion to simplify a formula, note that the entire formula does not have to be expanded. It is perfectly reasonable to perform the expansion of a single expression and simplify the result.

Example 6.3

Remove the catastrophic cancellation from the following formula:

$$\cos x - e^{2x}$$

Solution: Loss of precision occurs when $x \to 0$, where both operands approach 1. Independently expanding the two expressions $\cos x$ and e^y (where $y = 2x$) and substituting them into the formula yields:

$$\left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots\right) - \left(1 + 2x + \frac{4x^2}{2!} + \frac{8x^3}{3!} + \frac{16x^4}{4!} + \frac{32x^5}{5!} + \frac{64x^6}{6!} + \cdots\right)$$

which allows us to analytically remove the leading 1 from both expansions and solve to the desired precision:

$$-\left(2x + \frac{5x^2}{2!} + \frac{8x^3}{3!} + \frac{15x^4}{4!} + \frac{32x^5}{5!} + \frac{65x^6}{6!} + \cdots\right)$$

6.5 Mean Value Theorem

If both operands to a subtraction can be drawn from the same function $f(x)$, and the derivative $f'(x)$ is known, the result of the subtraction can be approximated using the *Mean Value Theorem* (MVT).

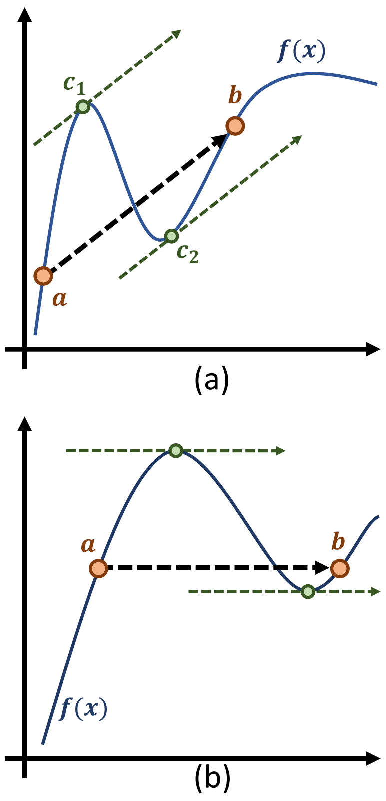


Figure 6.1 Evidence for the Mean Value Theorem (MVT) is easy to see graphically for any continuous function $f(x)$. (a) Two tangent lines exist for points c_1 and c_2 on the interval $[a, b]$ that are parallel to the line connecting a and b. (b) Rolle's Theorem, which says that there is at least one point c in $[a, b]$ such that $f'(c) = 0$ if $f(a) = f(b)$, is a useful extension of the MVT.

Mean Value Theorem

If a function $f(x)$ is:

- continuous on $[a, b]$
- differentiable on (a, b)

then there exists some point $c \in (a, b)$ such that:

$$f'(c) = \frac{f(b) - f(a)}{b - a} \tag{6.54}$$

In order to deal with catastrophic cancellation, the MVT can be rearranged in two ways:

$$f(b) - f(a) = f'(c)(b - a)$$

which suggests that the difference between two values returned by the same function $f(x)$ can be calculated using the difference between their arguments a and b, and the derivative $f'(x)$ at some point $c \in (a, b)$. If the function is well-behaved, we can approximate this subtraction by selecting the midpoint:

$$c = \frac{a + b}{2}$$

Consider the expression $e^x - 1$, which can also be expressed as:

$$f(b) - f(a)$$

where

$$f(x) = e^x \quad f'(x) = e^x \quad a = 0 \quad b = x \quad c = \tfrac{x}{2}$$

We can then reformulate the expression:

$$e^x - 1 \approx x e^{\frac{x}{2}}$$

Given that the value for c is an approximation, we should evaluate the expression to determine if the approximation error provides a better result.

Example 6.4

Mitigate the loss of precision in the expression $e^x - 1$ and evaluate the improvement for $x = 0.05$ using 3 digits of precision.

Solution: Using the Mean Value Theorem, we can reformulate the expression:

$$e^x - 1 \approx xe^{\frac{x}{2}}$$

We should compare the improvement using the relative error. This requires a *true* value T representing the expected value for $x = 0.01$, which can be obtained using a calculator and rounding to three digits of precision:

$$\begin{aligned} e^x - 1 &= 0.051271 \\ T &= 5.13 \times 10^{-2} \end{aligned}$$

Next, evaluate the original expression while chopping to 3 significant figures at each operation. This simulates a computer performing the calculation at the specified precision:

$$\begin{aligned} e^x - 1 &= 1.05127 - 1.00000 \\ &= 1.05 - 1.00 \\ &= 5 \times 10^{-2} \end{aligned}$$

The original formulation provides a relative error of:

$$\left| \frac{5.13 \times 10^{-2} - 5.00 \times 10^{-2}}{5.13 \times 10^{-2}} \right| \approx 2.53\%$$

Using the new formulation (chopping to 3 significant figures at each step) yields:

$$\begin{aligned} xe^{\frac{x}{2}} &= 0.05e^{0.025} \\ &= 0.05(1.0253) \\ &= 0.05(1.02) \\ &= 0.051 = 5.1 \times 10^{-2} \end{aligned}$$

which has a relative error of:

$$\left| \frac{5.13 \times 10^{-2} - 5.10 \times 10^{-2}}{5.13 \times 10^{-2}} \right| \approx 0.58\%$$

Exercises

Identify where the loss of precision occurs in the following expressions and reformulate them to mitigate it.

Verify the improvement in precision by testing the given numerical value using 3 digits of precision and calculating the relative error before and after reformulation. **All angles are expressed in radians.**

Exercises for 6.3 Reformulation

P6.1	$f(x)=\dfrac{\sqrt{1+x^2}-1}{x^2}$	$f(0.14)=.498$
P6.2	$f(x)=\sqrt{x^4+4}-2$	$f(0.4)=.639\times10^{-2}$
P6.3	$f(x)=\sqrt{x+2}-\sqrt{x}$	$f(138)=.848\times10^{-1}$
P6.4	$f(x)=\sqrt[4]{x+4}-\sqrt[4]{x}$	$f(14)=.125$
P6.5	$f(x)=\dfrac{\sin x}{x-\sqrt{x^2-1}}$	$f(4)=-5.96$
P6.6	$f(x)=1-\sin x$	$f(1.5)=.251\times10^{-2}$
P6.7	$f(x)=\dfrac{1-\cos x}{2}$	$f(.15)=.561\times10^{-2}$
P6.8	$f(x)=\cos(\pi+x)+1$	$f(.07)=.245\times10^{-2}$
P6.9	$f(x)=1-\sin\left(\dfrac{\pi}{2}+x\right)$	$f(.09)=.405\times10^{-2}$
P6.10	$f(x,y)=\cos(x-y)-\sin x\sin y$	$f(1.57,0.52)=.691\times10^{-3}$
P6.11	$f(x,y)=1-\sqrt{\dfrac{\cos(x-y)-\cos x\cos y}{\sin x}}$	$f(.6,1.5)=.125\times10^{-2}$

Use series expansions to mitigate catastrophic cancellation. You can limit your expansion to a third-order approximation.

Exercises for 6.4 Reformulation Using Taylor Series

P6.12	$f(x)=\dfrac{1-\cos x}{x^2}$	$f(0.01)=5.0\times10^{-1}$
P6.13	$f(x)=\sqrt{\dfrac{e^{2x}-1}{e^x-1}}$	$f(0.01)=1.42$
P6.14	$f(x)=\sqrt[3]{x+1}-\sqrt[3]{x}$	$f(0.01)=0.788$

Provide a better approximation for the following expressions using the Mean Value Theorem. Compare your results to a high-precision calculation (computer or calculator).

Exercises for 6.5 Mean Value Theorem

P6.15	$f(x)=\sqrt{100}-\sqrt{101}$
P6.16	$f(x)=1.1e^{-1.1}-1.11e^{-1.11}$
P6.17	$f(x)=\cos 1.1-\cos 1.11$

Chapter 7

Roots of Equations

Many physical systems can be written in the form of an equation. For example, consider solving for the dependent variable x given the independent value y and their relationship through the function $f(x)$:

$$y = f(x)$$

If we have an analytical expression for $f(x)$, we may be able to find the inverse function

$$x = f^{-1}(y)$$

However, the relationship specified in $f(x)$ may be unknown or extremely complex, making an analytical solution impractical. We may be able to find a numerical solution by using an algorithm to calculate the value of x within some defined precision.

We can solve this numerical problem by reformulating the expression such that the solution lies at a *root*. A *root* of an expression $g(x)$ is a value r such that $g(r) = 0$. To do this, we first reformulate the expression:

$$y - f(x) = 0 \tag{7.1}$$

and then use a *root finding* algorithm to identify values for x that satisfy this equation.

7.1 Simple and Multiple Roots

Roots of expressions can be classified based on their *multiplicity*.

Multiplicity

For a given polynomial $p(x)$, the root r is said to have *multiplicity* k if $p(x)$ can be expressed as:

$$p(x) = (x - r)^k s(x) \tag{7.2}$$

where $s(x)$ is another polynomial.

Intuitively speaking, the multiplicity is the number of polynomial factors that are zero at the root. For example, the polynomial

$$p(x) = (x-1)^7(x-3)^2(x+2)$$

will have three roots: $r_0 = -2$, $r_1 = 1$, and $r_2 = 3$. The first root $r_0 = -2$ has multiplicity $m_0 = 1$ and is therefore referred to as a *simple root*. The other two roots have multiplicity $m_1 = 7$ for $r_1 = 1$ and $m_2 = 2$ for $r_2 = 3$. The multiplicity also affects several features of the root:

- If a root r of $p(x)$ has multiplicity k, then r is also a root for the first $k-1$ derivatives of $p(x)$.
- If the multiplicity is odd, the values to the left and right of the root have opposite sign.
- Higher multiplicity can affect the convergence of root-finding algorithms. For example, Newton's Method relies on the derivative of the function, which approaches zero as the algorithm approaches the root.

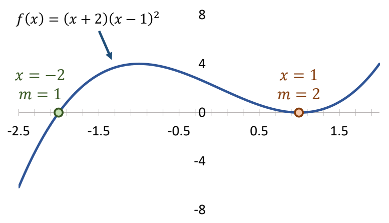


Figure 7.1 The multiplicity of a polynomial specifies how many of its factors are zero at the root. Roots with odd multiplicity (green) have values of opposite sign surrounding the root and can be calculated numerically using bracketing methods. Roots with even multiplicity (orange) require other approaches, such as Newton's method.

A *simple root* is a root that has multiplicity $m = 1$. By definition, a function $g(x)$ with a simple root r_0 has a derivative $g'(r_0) \neq 0$. These roots are generally easier to find because the value of the adjacent points $f(r_0 - \epsilon)$ and $f(r_0 + \epsilon)$ will have opposite sign and the derivative will be non-zero. Simple roots, and roots with odd multiplicity, can be reliably located using *bracketing methods*. Bracketing methods require that the user provide an initial interval surrounding the root. The most common bracketing method is the Bisection Method.

7.2 Bisection Method

The bisection method is a bracketing method that finds a simple root r of a function $g(x)$ at a specified precision. The function $g(x)$ does not have to be differentiable, or even continuous. The algorithm does not necessarily have to evaluate $g(x)$, however it must be able to determine the sign (positive or negative) for any value x. For the purposes of describing the algorithm, we will assume that $g(x)$ can be evaluated.

We are given, as input, the function $g(x)$ as well as an interval $[a, b]$ that bounds the root. The bisection method iteratively subdivides this interval to converge to the root position:

Bisection Method Algorithm

```
    input:  $g(x)$
            $a, b$ such that $r \in [a, b]$
            maximum number of iterations $N$
            acceptable error $\epsilon$
    $n = 0$
    while $n < N$:
        $c = \frac{a+b}{2}$
        **if** $g(c) = 0$ or $\frac{b-a}{2} \le \epsilon$:
            **output** $c$
        $n = n + 1$
        **if** $g(a)g(c) > 0$ **then** $a = c$ **else** $b = c$
    **output** FAIL
```

When implementing this algorithm on a digital system, it is important to note that **the resulting root** r **may not yield** $g(r) = 0$. This algorithm will calculate r to the specified precision, which will not necessarily be the exact root.

The bisection method is performing a binary search of the interval $[a,\ b]$, where each iteration reduces the range of values that *could* be the root by a factor of 2. Convergence is therefore given by the Bisection Method Theorem, allowing us to calculate the number of iterations required to achieve a given precision ϵ.

Bisection Method Theorem

If the bisection algorithm is applied to a continuous function g on an interval $[a,\ b]$, where $f(a)f(b) < 0$, then after n steps, an approximate root will have been computed with an error at most

$$\frac{b-a}{2^n} \tag{7.3}$$

This theorem suggests that the bisection method effectively provides one additional binary bit of precision to the result for every iteration.

Example 7.1

How many iterations are required to achieve an error less than ϵ?

Solution: According to the bisection method theorem:

$$\frac{b-a}{2^n} < \epsilon$$

Solving for n yields:

$$\begin{aligned}
\epsilon &> \frac{b-a}{2^n} \\
\epsilon 2^n &> b-a \\
\ln\left(\epsilon 2^n\right) &> \ln(b-a) \\
\ln(\epsilon) + \ln\left(2^n\right) &> \ln(b-a) \\
n\ln(2) &> \ln(b-a) - \ln(\epsilon) \\
n &> \frac{\ln(b-a) - \ln(\epsilon)}{\ln(2)}
\end{aligned}$$

The number of iterations required to achieve a precision of ϵ is:

$$n = \left\lceil \frac{\ln(b-a) - \ln(\epsilon)}{\ln(2)} \right\rceil$$

However, achieving a precision of ϵ also requires that the value can be represented using the implemented floating point format.

Example 7.2

Find a value of $x \in [0,\ 1]$ that satisfies the following equation:

$$e^x = 3x$$

to an accuracy of $\epsilon = 0.05$.

Solution: The expression can be formulated into a simple root-solving problem:

$$e^x - 3x = 0$$

We can then calculate the number of necessary iterations to solve this problem:

$$\begin{aligned} n &= \left\lceil \frac{\ln(b-a) - \ln(\epsilon)}{\ln(2)} \right\rceil \\ &= \left\lceil \frac{\ln(1) - \ln(0.05)}{\ln(2)} \right\rceil \\ &= \left\lceil \frac{0 - (-2.995)}{0.6931} \right\rceil \\ &= \lceil 4.322 \rceil \\ &= 5 \end{aligned}$$

Calculating the relevant parameters for each iterations gives the following results:

a	b	$f(a)$	$f(b)$	c	$f(c)$
0	1	1	−0.2817	0.5	0.1487
0.5	1	0.1487	−0.2817	0.75	−0.1330
0.5	0.75	0.1487	−0.2817	0.625	−0.006754
0.5	0.625	0.1487	−0.006754	0.5625	0.06755
0.5625	0.625	−0.006754	.06755	0.5938	0.02946

The result after 5 iterations is $r \approx 0.5938$, which is within the specified range of the actual result: $r = 0.6191 \pm 0.05$.

7.3 Newton's Method

Newton's method is a technique often learned early in calculus for finding roots of equations given an initial estimate of the root x_0. The method computes an iterative sequence of values $x_0,\ x_1, \cdots,\ x_n$ that converge to the root based on the update function:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7.4}$$

Iteration continues until a desired level of accuracy is achieved.

7.3.1 Convergence

The main advantage of Newton's method is its extremely fast quadratic convergence. For most applications, you can expect the number of *digits* of accuracy to double every iteration.

Newton's Method Convergence

Consider the first-order Taylor series expansion of a function $f(x)$ about a point x_n that is close to the root r:

$$f(r) = f(x_n) + f'(x_n)(r - x_n) + R_1(r) \tag{7.5}$$

where R_1 is the Lagrange remainder (Equation 4.10), which can be calculated at the root r:

$$R_1(r) = \frac{f''(c)}{2!}(r - x_n)^2 \tag{7.6}$$

Substituting in this value for R_1 in the expansion yields:

$$f(r) = f(x_n) + f'(x_n)(r - x_n) + \frac{f''(c)}{2}(r - x_n)^2$$

Since we know that r is a root, and therefore $f(r) = 0$, we can modify this expression:

$$\begin{aligned}
f(r) &= f(x_n) + f'(x_n)(r - x_n) + \frac{f''(c)}{2}(r - x_n)^2 \\
0 &= f(x_n) + f'(x_n)(r - x_n) + \frac{f''(c)}{2}(r - x_n)^2 \\
f(x_n) + f'(x_n)(r - x_n) &= -\frac{f''(c)}{2}(r - x_n)^2 \\
\frac{f(x_n)}{f'(x_n)} + (r - x_n) &= -\frac{f''(c)}{2f'(x_n)}(r - x_n)^2
\end{aligned}$$

By re-arranging Newton's method (Equation 7.4):

$$\begin{aligned}
x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \\
\frac{f(x_n)}{f'(x_n)} &= x_n - x_{n+1}
\end{aligned}$$

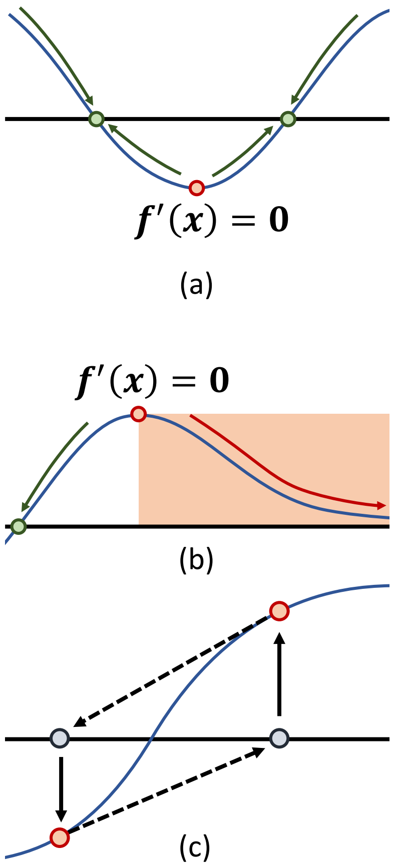


Figure 7.2 Newton's method always moves "uphill" or "downhill" towards a root, and can fail if any iteration x_n encounters a point with zero slope where $f'(x) = 0$ (a). Newton's method can also diverge when the slope of the function at x_n moves away from the root (b). Finally, Newton's method can encounter infinite loops without convergence (c). This final case is common when x_n approaches the root within machine precision, since the value of x_{n+1} gets chopped.

we can substitute the ratio on the left hand side:

$$x_n - x_{n+1} + r - x_n = -\frac{f''(c)}{2f'(x_n)}(r - x_n)^2$$
$$r - x_{n+1} = -\frac{f''(c)}{2f'(x_n)}(r - x_n)^2$$

This gives us the error of two sequential estimates of the root in the same expression:

$$\epsilon_n = r - x_n$$
$$\epsilon_{n+1} = r - x_{n+1}$$
$$\epsilon_{n+1} = K(r - x_n)^2$$

where K is a constant - in this case proportional to half of the second derivative of $f(x)$ near the root. From our discussion on asymptotics, we know that the error in the estimate x_{n+1} is proportional to the square of the error in the estimate x_n. We can therefore say that the error drops quadratically for each iteration of Newton's method. If the error is $\frac{K}{4}$ at iteration n, we would expect it to be $\frac{K}{16}$ at iteration $n+1$.

7.3.2 Pitfalls

The main disadvantage to Newton's method is the reliance on the initial guess x_0, which must be close enough to the root for convergence. One intuitive way to think of Newton's method is that it always rolls "uphill" or "downhill" towards the root. This makes it prone to getting stuck in a local minimum or diverging (Figure 7.2).

One of the most common errors encountered in a digital implementation is an infinite loop. This can frequently occur when x_n approaches r within machine precision. Recall from our study of floating point that an exact representation of r is unlikely. If x_n is close to r, the next approximation x_{n+1} may get chopped to a nearby value. This can result in iteration between two adjacent values $r \pm \epsilon$, or x_{n+1} getting chopped to the same value such that $x_{n+1} = x_n$. It is important to test for this case in a digital implementation, because it will frequently be encountered.

Finally, if r is a multiple root (i.e. has multiplicity $m > 1$), then $f'(r) = 0$. This results in $f'(x_n) \to 0$ as $x_n \to r$ and therefore produces a smaller step size as the algorithm converges. In practical terms, this results in linear convergence for roots of multiplicity $m > 1$. If the multiplicity is known *a priori*, this can be mitigated by swapping the the numerator and denominator and scaling by the multiplicity:

$$x_{n+1} = x_n - m\frac{f(x_n)}{f'(x_n)} \tag{7.7}$$

7.4 Secant Method

Another constraint on Newton's method is the need to calculate the derivative $f'(x)$. In many practical cases, the user may not have access to an analytical representation of the function f, making calculation of f' impractical. However, it is possible to approximate the derivative of a function using only f using the *finite difference method*.

Consider first order Taylor series expansion of the function f about some point a:

$$f(x) \approx f(a) + f'(a)(x-a)$$

Solving for the derivative yields an approximation to the derivative:

$$f'(a) \approx \frac{f(x)-f(a)}{x-a} \tag{7.8}$$

that becomes more accurate as $x \to a$. If we can evaluate f at two nearby points x_{i-1} and x_i, we can use this formulation to approximate the derivative:

$$f'(x_i) \approx \frac{f(x_{i-1})-f(x_i)}{x_{i-1}-x_i} \tag{7.9}$$

We will discuss finite differences in detail in Chapter **??**.

Substituting this approximation for the derivative into Newton's method (Equation 7.4) yields the *secant method*:

$$x_{n+1} = x_n - \frac{f(x_n)(x_{n-1}-x_n)}{f(x_{n-1})-f(x_n)} \tag{7.10}$$

The convergence of the secant method is slightly worse than Newton's method, due to the lower-order approximation of the derivative. Specifically, convergence is on the order of:

$$O\left(n^{\frac{1+\sqrt{5}}{2}}\right) \approx O\left(n^{1.618}\right)$$

compared to quadratic convergence provided by Newton's method. Both are significantly faster than the bisection method, which is linear: $O(n)$. The advantage provided by the bisection method is its stability and guaranteed convergence for simple roots.

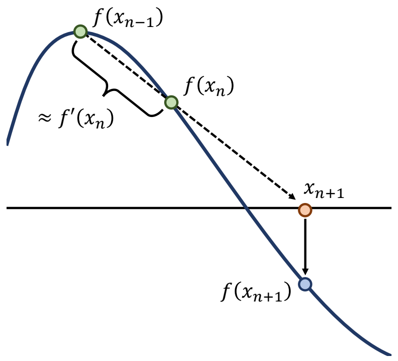


Figure 7.3 The *secant method* approximates the derivative of f by using finite differences based on the previous two root approximations. Alternatively, the secant method can be thought of as a linear extrapolation method, since the new root approximation x_{n+1} lies on the line extrapolated from $f(x_{n-1})$ and $f(x_n)$.

7.5 Inverse Quadratic Interpolation

While the secant method relies on extrapolation of the next value x_{n+1} along a line, *inverse quadratic interpolation* (IQI) fits a second-order polynomial to three consecutive values of the function f. This root-finding algorithm relies on polynomial interpolation, which we will discuss in detail in Chapter **??**.

Given three preceding values $[x_{n-1},\ x_{n-1},\ xn]$, the next value is calculated as a weighted sum of these values:

$$x_{n+1} = Ax_{n-2} + Bx_{n-1} + Cx_n$$

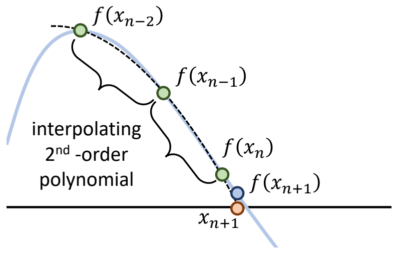


Figure 7.4 Inverse quadratic interpolation is used to calculate an estimate of the root based on three previous values. A quadratic function is fit to the three preceding root estimates, and the intersection with the x-axis is used as the next estimate x_{n+1}.

where the coefficients are determined by fitting a quadratic polynomial to the resulting data (Figure 7.4). We will discuss methods for polynomial fitting in later chapters, so for now we will provide the appropriate weights:

$$A = \frac{f(x_{n-1})f(x_n)}{\big(f(x_{n-2}) - f(x_{n-1})\big)\big(f(x_{n-2}) - f(x_n)\big)}$$
$$B = \frac{f(x_{n-2})f(x_n)}{\big(f(x_{n-1}) - f(x_{n-2})\big)\big(f(x_{n-1}) - f(x_n)\big)}$$
$$C = \frac{f(x_{n-2})f(x_{n-1})}{\big(f(x_n) - f(x_{n-2})\big)\big(f(x_n) - f(x_{n-1})\big)}$$

The inverse quadratic interpolation method has convergence of approximately $O\left(n^{1.8}\right)$, making it faster than the secant method but still slower than Newton's method.

7.6 Hybrid Algorithms

Given the strengths and weaknesses of various algorithms, a more practical approach to finding the roots of arbitrary functions relies on multiple methods. Dekker's method, proposed by Theodorus Dekker, relies on both the secant and bisection methods to create a stable algorithm with fast convergence (Algorithm 3).

The most common hybrid approach is *Brent's method*, which combines inverse quadratic interpolation, the secant method, and the bisection method.

Both of these methods provide several advantages over each of the individual algorithms discussed, including:

- guaranteed convergence provided by using the bisection method
- near-quadratic convergence speed provided by the open methods (secant or inverse quadratic interpolation)
- no need for an analytical derivative

Exercises

Answer the following using 5 iterations of the bisection method.

Exercises for 7.2 Bisection Method

P7.1 Find the root of the polynomial $p(x) = (x+5)(x-2.5)(x+1)(x-1)$ given the interval $[-4.0, 0.75]$. What is the relative error?

P7.2 Solve the equation for x given that $1 \leq x \leq 5$: $\ln x = \cos x$

Answer the following using 5 iterations of Newton's method:

Algorithm 3 Dekker's method, developed in 1969, describes an efficient and stable root-finding algorithm with near-quadratic convergence.

```
Dekker's Method
    input: function $f$ and interval $[a,\ b]$
    initialize: $b_{-1} = a_0 = a$, $b_0 = b$
    for $n = 1$ to $N$:
        $m = \frac{a_{n-1} + b_{n-1}}{2}$
        if $f(b_{n-1}) = f(b_{n-2})$ then $s = m$:
        else $s = b_{n-1} - \frac{b_{n-1} - b_{n-2}}{f(b_{n-1}) - f(b_{n-2})} f(b_{n-1})$
        if $s \in [b_{n-1},\ m]$ then $b_{n+2} = s$ else $b_{n+2} = m$
        if $f(a_{n-1}) f(b_{n+2}) < 0$ then $a_n = a_{n-1}$ else $a_n = b_{n-1}$
    output: $b_N$
```

Exercises for 7.3 Newton's Method

P7.3 Solve the equation for x given that $1 \le x \le 8$: $\ln x = \sin x$

Programming Assignment - Newton-Horner Method

Implement an algorithm that is guaranteed to find the root r of a polynomial $p(x)$ within a user-specified range $[a,\ b]$. Calculate r to the highest possible precision using 64-bit floating point values. Your method should converge quadratically.

This programming assignment will require a hybrid approach (similar to Brent's method). You will primily rely on Newton's method to calculate the zero. Use Horner's method to calculate the value of the polynomial and its derivative as necessary. In cases where Newton's method does not converge, use the bisection method to refine the interval. Attempt to optimize this algorithm by minimizing the number of overall iterations required to achieve machine precision.

To test this algorithm, take a list of coefficients **c** along with values for $[a,\ b]$ as command-line arguments and print the result to the console:

```
>> nhroot c0 c1 c2 c3 ... cn a b
```

Your console output should include:

- the estimated root, showing 15 digits of precision
- the value of $f(x)$ at the calculated root
- the number of bisection iterations used
- the number of Newton's method iterations used

Chapter 8

Solving Linear Systems

One of the most common calculations performed in numerical computing is solving linear systems of equations. Linear systems are routinely encountered in science and engineering. These problems range from computing the properties of electrical circuits to predicting the complex behaviors of fluids. However, the limited precision provided by floating point numbers can introduce significant errors in solutions to relatively simple problems. In this chapter, we will discuss:

- Standard methods for solving linear systems using Gaussian elimination.
- How numerical errors are introduced when Gaussian elimination is directly applied to linear systems.
- Methods for mitigating numerical error, including *pivoting* and *scaling*.
- Methods for calculating the properties of linear systems so that errors can be estimated and bounded.
- Algorithms for performing *LU decomposition*, which forms the basis for many common algorithms applied to linear systems.

8.1 Gaussian Elimination and Backsubstitution

Linear equations take the form:

$$\begin{aligned}
6x_1 - 2x_2 + 2x_3 + 4x_4 &= 16 \\
12x_1 - 8x_2 + 6x_3 + 10x_4 &= 26 \\
3x_1 - 13x_2 + 9x_3 + 3x_4 &= -19 \\
-6x_1 + 4x_2 + x_3 - 18x_4 &= -34
\end{aligned}$$

and are more easily expressed using matrix and vector operations:

$$\mathbf{Ax} = \mathbf{b}$$

where

$$\mathbf{A} = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 16 \\ 26 \\ -19 \\ -34 \end{bmatrix}$$

This system can be solved by computing the matrix inverse $\mathbf{A}^{-1}$ and solving for $\mathbf{x}$:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

However, calculating a matrix inverse is a computationally expensive approach that is usually unnecessary. The most common method for solving a single linear system is to apply Gaussian elimination (GE) (see Section 1.3) followed by backsubstitution. Gaussian elimination is used to iteratively reduce the matrix $\mathbf{A}$ and right hand side $\mathbf{b}$ into row-echelon form:

$$\mathbf{A} = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & -3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 16 \\ -6 \\ -9 \\ -3 \end{bmatrix}$$

At this point, the entire system can be solved using backsubstitution:

$$\begin{aligned} -3x_4 &= -3 & & & x_4 &= 1 \\ 2x_3 - 5(1) &= -9 & & & x_3 &= -2 \\ -4x_2 + 2(-2) + 2(1) &= -6 & &\Longleftrightarrow & x_2 &= 1 \\ 6x_1 - 2(1) + 2(-2) + 4(1) &= 16 & & & x_1 &= 3 \end{aligned}$$

Gaussian elimination is an extremely common computational problem, and is one of the key methods currently used to benchmark supercomputer performance. However, we will find that the method as analytically derived is extremely unstable on a digital system. In this chapter, we will discuss methods used to analyze and solve linear systems while maximizing performance and minimizing error.

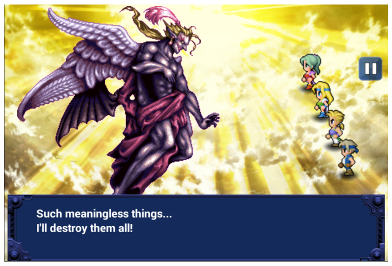


Figure 8.1 Geometric Series.

8.2 Robustness Testing

It is often useful to design a problem that can be used to test the *robustness*, or stability, of an algorithm. In the case of Gaussian elimination, we are interested in designing a system of linear equations that will be difficult for a computer to solve using GE, but for which we know (or can readily calculate) the correct answer. This will allow us to evaluate the accuracy of our GE implementation using the relative error (see Section 4.3.2).

We can create such a problem using the *geometric series*:

$$p(t) = a + at + at^2 + at^3 + \cdots + at^{N-2} + at^{N-1} \tag{8.1}$$

$$= \sum_{j=0}^{N-1} at^j \tag{8.2}$$

which allows us to specify the number of terms N in the linear system. This geometric series can also be compressed to a relatively simple analytical formulation:

$$p(t) = \sum_{j=0}^{N-1} a t^j = a\frac{t^N - 1}{t-1}$$

We can therefore generate a set of N unique linear equations using $p(n)$ where $n \in [1,\ N]$:

$$\begin{aligned}
a + a(1+1) + a(1+1)^2 + \cdots + a(1+1)^{N-1} &= a\frac{(1+1)^N - 1}{1} \\
a + a(1+2) + a(1+2)^2 + \cdots + a(1+2)^{N-1} &= a\frac{(1+2)^N - 1}{2} \\
&\vdots \\
a + a(1+N) + a(1+N)^2 + \cdots + a(1+N)^{N-1} &= a\frac{(1+N)^N - 1}{N}
\end{aligned}$$

By selecting a value for a, we can substitute it into the right hand side and compare the result of GE to the known value for a. In most cases we will use $a = 1$. For example, selecting $N = 3$ or $N = 5$ generates the following matrix equations:

$$\begin{bmatrix} 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{bmatrix} \cdot \begin{bmatrix} a \\ a \\ a \end{bmatrix} = \begin{bmatrix} 7 \\ 13 \\ 21 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \\ 1 & 5 & 25 & 125 & 625 \\ 1 & 6 & 36 & 216 & 1296 \end{bmatrix} \cdot \begin{bmatrix} a \\ a \\ a \\ a \\ a \end{bmatrix} = \begin{bmatrix} 31 \\ 121 \\ 341 \\ 781 \\ 1555 \end{bmatrix}$$

By solving for the vector **a**, we then calculate the maximum relative error in the result to see which values for N the standard GE algorithm fails. We test this using three different IEEE standard floating point formats (Figure 8.2), and find that the largest order for which can can achieve any level of accuracy using the `float64` format is $N = 14$. From an engineering or scientific perspective, this is **not** a large matrix. In addition, the values in the linear system corresponding to $N = 15$ range from the lowest $A_{11} = 1$ to the highest $b_{15} = 7.686 \times 10^{16}$. Since the IEEE `float64` format uses an 11-bit exponent and bias of 1023, this is well within the allowable exponent range of $\left[2^{-1023},\ 2^{1024}\right] = \left[1.1125 \times 10^{-308},\ 1.7980 \times 10^{308}\right]$.

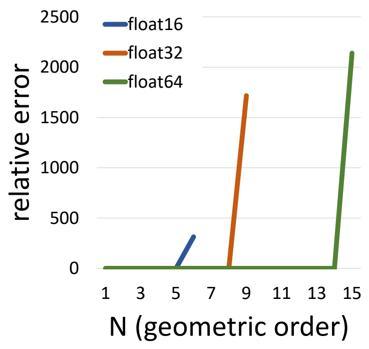


Figure 8.2 The relative error using Gaussian elimination to solve a system of linear equations composed of an Nth-order geometric series varies based on the floating point format. Increased floating point precision allows for a larger set of unknown values, however Gaussian elimination becomes impractical for double precision (`float64`) at $N = 15$.

8.3 Precision Loss

Consider the following linear system and corresponding matrix equation

$$\begin{aligned} x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned} \quad \longrightarrow \quad \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

where the first step in Gaussian elimination results in division by zero. Students are generally taught that this *degenerate case* is addressed by swapping the order of equations. However, this example provides a convenient lesson: **if a numerical method fails for some value, it is probably unstable *near* that value**.

Now consider the almost identical example:

$$\begin{aligned} \epsilon x_1 + x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned} \quad \rightarrow \quad \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

where $\epsilon \approx 0$ is an extremely small number. The solution to this linear system is clearly $x_1 \approx x_2 \approx 1$. If we symbolically perform Gaussian elimination, the resulting linear system in row-echelon form is:

$$\begin{aligned} \epsilon x_1 + x_2 &= 1 \\ \left(1 - \frac{1}{\epsilon}\right) x_2 &= 2 - \frac{1}{\epsilon} \end{aligned} \quad \rightarrow \quad \begin{bmatrix} \epsilon & 1 \\ 0 & (1 - \frac{1}{\epsilon}) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 - \frac{1}{\epsilon} \end{bmatrix}$$

Applying the backsubstitution algorithm yields the correct answer for x_2:

$$\begin{aligned} \left(1 - \frac{1}{\epsilon}\right) x_2 &= 2 - \frac{1}{\epsilon} \\ x_2 &= \frac{2 - \frac{1}{\epsilon}}{1 - \frac{1}{\epsilon}} \\ x_2 &\approx 1 \end{aligned}$$

However, calculating the value for x_1:

$$\begin{aligned} \epsilon x_1 + x_2 &= 1 \\ x_1 &= \frac{1 - x_2}{\epsilon} \quad \text{where } x_2 \approx 1 \end{aligned}$$

will result in catastrophic cancellation in the numerator. Small round-off errors due to reduced significance can result in large errors in the final division by ϵ.

It is tempting to assume that the reason for this catastrophic cancellation is the small leading coefficient in the first equation. However, it is important to understand that this cancellation results from a coefficient being small **relative to other coefficients in the same equation**. For example, consider modifying the equations:

$$\begin{aligned} x_1 + C x_2 &= C + \epsilon \\ x_1 + x_2 &= 2 \end{aligned} \quad \rightarrow \quad \begin{bmatrix} 1 & C \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} C + \epsilon \\ 2 \end{bmatrix}$$

where $C = \frac{1}{\epsilon}$ is a very large number. Applying GE results in the following system in row-echelon form:

$$\begin{aligned} x_1 + C x_2 &= C + \epsilon \\ (1 - C) x_2 &= 2 - C - \epsilon \end{aligned} \quad \rightarrow \quad \begin{bmatrix} 1 & C \\ 0 & 1 - C \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} C + \epsilon \\ 2 - C - \epsilon \end{bmatrix}$$

Because C is much larger than all other values, they get rounded and chopped when stored in a digital register:

$$\begin{aligned} x_1 + C x_2 &= C \\ -C x_2 &= -C \end{aligned} \quad \rightarrow \quad \begin{bmatrix} 1 & C \\ 0 & C \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} C \\ -C \end{bmatrix}$$

Applying backsubstitution to the rounded result yields:

$$\begin{aligned} -Cx_2 &= -C \\ x_2 &= \frac{-C}{-C} \\ x_2 &\approx 1 \end{aligned} \quad \implies \quad \begin{aligned} x_1 + Cx_2 &= C \\ x_1 &= C - C \\ x_1 &\approx 0 \end{aligned}$$

which again results in catastrophic cancellation since $(C + \epsilon) \approx Cx_2$ when $x_2 \approx 1$ and $|C|$ is large.

The critical step in GE is the scaling of the pivot equation $p(x)$ by a factor m (see Algorithm 1). If the scaled pivot equation $mp(x)$ produces coefficients much larger than others in the system, roundoff will occur when $mp(x)$ is subtracted from the other equations. When the resulting equations have coefficients and right-hand-side values that are large, subtraction during backsubstitution will introduce catastrophic cancellation.

We have seen two cases where GE can result in catastrophic cancellation:

1. **A small leading coefficient** ϵ results in a large scaling factor $m = \frac{1}{\epsilon}$, producing a large scaled pivot $mp(x)$.
2. **Large non-leading coefficients** C **with large right-hand-side components** result in large $mp(x)$ when m is insufficiently small to compensate.

Both of these cases have the same root problem: **the leading coefficients in a row are small relative to other coefficients and the right-hand-side**.

8.4 Partial Pivoting

The most common method for addressing this issue is to select the pivot equation $p(x)$ such that roundoff error is *postponed* in order to prevent iterative amplification of these errors. For each iteration of GE, we select the pivot equation based on the *magnitude*, or absolute value, of the leading coefficient. Once the optimal pivot equation $p(x)$ is identified, a row-swap is applied to move that equation into the appropriate position to put the matrix into row-echelon form. This method is known as *partial pivoting*.

Consider the simple example from the previous section:

$$\begin{aligned} \epsilon x_1 + x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned} \quad \rightarrow \quad \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

The resulting cancellation can be mitigated by selecting the **second** equation as the pivot and swapping it into the first position:

$$\begin{aligned} x_1 + x_2 &= 2 \\ \epsilon x_1 + x_2 &= 1 \end{aligned} \quad \rightarrow \quad \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

This results in a small scaling factor m:

$$\begin{aligned} x_1 + x_2 &= 2 \\ (1-\epsilon)x_2 &= 1 - 2\epsilon \end{aligned} \quad \rightarrow \quad \begin{bmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{bmatrix} \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1-2\epsilon \end{bmatrix}$$

Applying backsubstitution provides an improved result:

$$\begin{aligned}(1-\epsilon)x_2 &= 1-2\epsilon \\ x_2 &= \frac{1-2\epsilon}{1-\epsilon} \\ x_2 &\approx 1\end{aligned} \quad \Longrightarrow \quad \begin{aligned}x_1 + x_2 &= 2 \\ x_1 &= 2 + x_2 \\ x_1 &\approx 1\end{aligned}$$

Example 8.1

Use partial pivoting to solve the following system of linear equations **using 3 digits of precision**:

$$\begin{aligned}0.1x_1 + 300x_2 - 2x_3 &= 7 \\ 2x_1 + 10x_2 - x_3 &= 10 \\ 3x_1 + 6x_2 + 12x_3 &= 15\end{aligned}$$

Solution: Reformulating the system to a matrix equation:

$$\begin{array}{c} r_1 \\ r_2 \\ r_3 \end{array}\begin{bmatrix}0.1 & 300 & -2 \\ 2 & 10 & -1 \\ 3 & 6 & 12\end{bmatrix}\begin{bmatrix}x_1 \\ x_2 \\ x_3\end{bmatrix} = \begin{bmatrix}7 \\ 10 \\ 15\end{bmatrix}$$

Select the pivot equation by finding the largest leading coefficient (r_3) and swap rows to put it in the correct position for elimination:

$$\begin{array}{c} r_3 \\ r_2 \\ r_1 \end{array}\begin{bmatrix}3 & 6 & 12 \\ 2 & 10 & -1 \\ 0.1 & 300 & -2\end{bmatrix}\begin{bmatrix}x_1 \\ x_2 \\ x_3\end{bmatrix} = \begin{bmatrix}15 \\ 10 \\ 7\end{bmatrix}$$

Perform elimination by calculating the appropriate scaling factors and subtracting:

$$\begin{array}{cc} & r_3 \\ m = 2/3 & r_2 \\ m = 0.1/3 & r_1 \end{array}\begin{bmatrix}3 & 6 & 12 \\ 0 & 6 & -91 \\ 0 & 300 & -2.4\end{bmatrix}\begin{bmatrix}x_1 \\ x_2 \\ x_3\end{bmatrix} = \begin{bmatrix}15 \\ 0 \\ 6.5\end{bmatrix}$$

Note that the result 299.8 is rounded back to 300, because we are only maintaining 3 digits of precision. The next pivot equation (r_1) is selected, swapped, and eliminated:

$$\begin{array}{cc} & r_3 \\ & r_1 \\ m = 6/300 & r_2 \end{array}\begin{bmatrix}3 & 6 & 12 \\ 0 & 300 & -2.4 \\ 0 & 0 & -8.95\end{bmatrix}\begin{bmatrix}x_1 \\ x_2 \\ x_3\end{bmatrix} = \begin{bmatrix}15 \\ 6.5 \\ -0.13\end{bmatrix}$$

Finally, solve using backsubstitution:

$$\begin{aligned}-8.95x_3 &= -0.13 \\ x_3 &= 1.45 \times 10^{-2}\end{aligned} \quad \Longrightarrow \quad \begin{aligned}300x_2 - 2.4x_3 &= 6.5 \\ 300x_2 - 2.4(0.0145) &= 6.5 \\ 300x_2 - 0.0348 &= 6.5 \\ 300x_2 &= 6.47 \\ x_2 &= 2.16 \times 10^{-2}\end{aligned}$$

$$\begin{aligned}
3x_1 + 6x_2 + 12x_3 &= 15 \\
3x_1 + 6(0.0216) + 12(0.0145) &= 15 \\
3x_1 + 0.130 + 0.174 &= 15 \\
3x_1 + 0.304 &= 15 \\
3x_1 &= 14.7 \\
x_1 &= 4.9
\end{aligned}$$

Example 8.2

Compare the error between Gaussian elimination with and without partial pivoting for the following linear system **using 3 digits of precision**:

$$\begin{bmatrix} 0.30 & -1.00 & 2.00 \\ 1.00 & 0 & -1.00 \\ 4.00 & 2.00 & -3.00 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 8.00 \\ -1.00 \\ -4.00 \end{bmatrix} \quad \text{where} \quad \begin{aligned} x &= 1.96 \\ y &= -1.48 \\ z &= 2.96 \end{aligned}$$

Solution: Reducing this linear system using normal Gaussian elimination results in:

$$\begin{bmatrix} 0.30 & -1.00 & 2.00 \\ 0 & 3.33 & -7.66 \\ 0 & 0 & 5.60 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 8 \\ -27.6 \\ 17 \end{bmatrix} \quad \text{where} \quad \begin{aligned} x &= 2.10 \\ y &= -1.29 \\ z &= 3.04 \end{aligned}$$

giving a relative error for each term of $E_x = 6.6\%$, $E_y = 12.8\%$, and $E_z = 2.7\%$.

Applying partial pivoting results in:

$$\begin{bmatrix} 4.00 & 2.00 & -3.00 \\ 0 & -1.15 & 2.23 \\ 0 & 0 & -1.22 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -4.00 \\ 8.30 \\ 17.0 \end{bmatrix} \quad \text{where} \quad \begin{aligned} x &= 2.1 \\ y &= -1.29 \\ z &= 3.04 \end{aligned}$$

which provides a significant improvement in relative error, providing the expected result for y and z and a relative error in x of $E_x = 0.5\%$.

8.5 Scaling

Partial pivoting relies on an underlying assumption of the linear system: that the leading coefficient provides a good representation of the magnitude of the equation. This is not necessarily the case:

$$\begin{bmatrix} 9 & 2000 & 16700 \\ 8 & 1 & 1 \\ 7 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 18700 \\ 10 \\ 11 \end{bmatrix} \quad \text{where} \quad \begin{aligned} x_1 &= 1 \\ x_2 &= 1 \\ x_3 &= 1 \end{aligned}$$

Applying partial pivoting to this matrix would select the first equation as the pivot. However, this will result in significant roundoff since the other coefficients in this row are large relative to the coefficients in other equations. Applying partial pivoting results in the row-echelon system:

$$\begin{bmatrix} 9 & 2000 & 16700 \\ 0 & -1780 & -14800 \\ 0 & 0 & 100 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 18700 \\ -16600 \\ 0 \end{bmatrix} \quad \text{yielding} \quad \begin{array}{l} x_1 = 0 \\ x_2 = 9.32 \\ x_3 = 11.1 \end{array}$$

which produces errors exceeding 100% for three significant figures.

The reason for this becomes clear when we re-scale the matrix. Note that multiplying any single equation in a linear system does not fundamentally change the system. We can therefore *scale* each row by the inverse of its largest (highest magnitude) coefficient:

$$\begin{bmatrix} 5.36 \times 10^{-3} & 0.119 & 1 \\ 1 & 0.125 & 0.125 \\ 1 & 0.143 & 0.429 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.11 \\ 1.25 \\ 1.57 \end{bmatrix}$$

Since these systems are fundamentally the same, we were clearly selecting a pivot equation with a relatively small leading coefficient.

While we can identify a better pivot equation by *pre-scaling* a matrix, this is almost never used as an initial step because it introduces additional roundoff. For exampale, note that every element in the above matrix lost a small amount of precision when the scaling factor was applied. The good news is that pre-scaling is also completely unnecessary! In fact, the operation is completely redundant since every iteration of GE applies a new scale value m to each row of the matrix - pre-scaling would simply result in different m values. **Scaling is only used to identify the correct pivot equation as the row with the largest *scaled* leading coefficient**.

Given the original linear system

$$\begin{array}{c} r_1 \\ r_2 \\ r_3 \end{array} \begin{bmatrix} 9 & 2000 & 16700 \\ 8 & 1 & 1 \\ 7 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 18700 \\ 10 \\ 11 \end{bmatrix}$$

we select the pivot equation based on the *scaled* leading coefficient: $\frac{9}{16700} = 5.3 \times 10^{-3}$ for r_1, 1 for both r_2 and r_3. Since r_2 and r_3 have the same scaled leading coefficient, either are a valid pivot so we will select r_2:

$$\begin{array}{c} r_2 \\ r_1 \\ r_3 \end{array} \begin{bmatrix} 8 & 1 & 1 \\ 9 & 2000 & 16700 \\ 7 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 18700 \\ 11 \end{bmatrix}$$

$$\begin{array}{c} r_2 \\ r_1 \\ r_3 \end{array} \begin{bmatrix} 8 & 1 & 1 \\ 0 & 2000 & 16700 \\ 0 & 0.125 & 2.13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 18700 \\ 2.25 \end{bmatrix}$$

The options for the second pivot equation have scaled leading coefficients of $\frac{2000}{16700} = 0.120$ for r_1 and 0.0587 for r_3. Selecting r_1 as the pivot doesn't require a row swap:

$$\begin{array}{c} r_2 \\ r_1 \\ r_3 \end{array} \begin{bmatrix} 8 & 1 & 1 \\ 0 & 2000 & 16700 \\ 0 & 0 & 1.09 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 18700 \\ 1.08 \end{bmatrix}$$

Backsubstitution gives us the result:

$$\begin{aligned} 1.09x_3 &= 1.08 \\ x_3 &= 0.991 \end{aligned} \implies \begin{aligned} 2000x_2 + 16700x_3 &= 18700 \\ 2000x_2 + 16500 &= 18700 \\ 2000x_2 &= 2200 \\ x_2 &= 1.1 \end{aligned}$$

$$\begin{aligned} 8x_1 + x_2 + x_3 &= 10 \\ 8x_1 + 1.10 + 0.991 &= 10 \\ 8x_1 + 2.1 &= 10 \\ 8x_1 &= 7.9 \\ x_1 &= 0.988 \end{aligned}$$

with relative errors of 0.9%, 10%, and 0.2%. This provides a significant improvement over partial pivoting alone.

8.6 Complete Pivoting

Our final consideration will be the application of *complete pivoting*, which selects both the optimal pivot equation *and* leading coefficient. This is implemented using both row and column swaps. Starting from the previous linear system:

$$\begin{array}{c} r_1 \\ r_2 \\ r_3 \end{array} \begin{bmatrix} 9 & 2000 & 16700 \\ 8 & 1 & 1 \\ 7 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 18700 \\ 10 \\ 11 \end{bmatrix}$$

The first pivot equation is r_1, however the leading coefficient is associated with x_3. We apply a column swap to bring the largest coefficient into the leading position. Note that this also changes the order of variables in the $\mathbf{x}$ vector:

$$\begin{array}{c} r_1 \\ r_2 \\ r_3 \end{array} \begin{bmatrix} 16700 & 9 & 2000 \\ 1 & 8 & 1 \\ 3 & 7 & 1 \end{bmatrix} \begin{bmatrix} x_3 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 18700 \\ 10 \\ 11 \end{bmatrix}$$

Applying GE to eliminate the leading coefficients in the remaining equations gives us:

$$\begin{array}{c} r_1 \\ r_2 \\ r_3 \end{array} \begin{bmatrix} 16700 & 9 & 2000 \\ 0 & 8 & 0.88 \\ 0 & 7 & 0.64 \end{bmatrix} \begin{bmatrix} x_3 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 18700 \\ 8.88 \\ 7.63 \end{bmatrix}$$

The leading coefficient of r_2 is the largest, so no further swaps are necessary:

$$\begin{array}{c} r_1 \\ r_2 \\ r_3 \end{array} \begin{bmatrix} 16700 & 9 & 2000 \\ 0 & 8 & 0.88 \\ 0 & 0 & -0.13 \end{bmatrix} \begin{bmatrix} x_3 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 18700 \\ 8.88 \\ -0.14 \end{bmatrix}$$

Using backsubstitution to solve for the unknowns gives us:

$$\begin{aligned} -0.13x_3 &= -0.14 \\ x_3 &= 1.08 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} 8x_1 + 0.88x_2 &= 8.88 \\ 8x_1 + 0.95 &= 8.88 \\ 8x_1 &= 7.93 \\ x_1 &= 0.991 \end{aligned}$$

$$\begin{aligned} 16700x_2 + 9x_1 + 2000x_3 &= 18700 \\ 16700x_2 + 8.92 + 2160 &= 18700 \\ 16700x_2 + 2170 &= 18700 \\ 16700x_2 &= 16500 \\ x_2 &= 0.988 \end{aligned}$$

with relative errors of 0.9%, 1.2%, and 8.0%. Note that using the largest value in the matrix to calculate m at each iteration makes scaling unnecessary.

8.7 Implementation

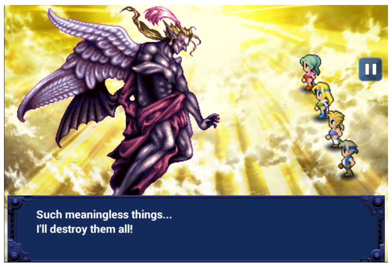


Figure 8.3 Results for partial pivoting using various floating point formats.

One of the biggest challenges for solving linear systems, particularly large linear systems, is finding a robust method for dealing with error propagation due to roundoff. We have discussed the three most fundamental methods used in computing: partial pivoting, scaling, and complete pivoting. For practical applications, it is important to understand the advantages and tradeoffs for each as well as tips for implementing these algorithms.

8.7.1 Index Vectors

All three cases use *matrix permutations*, such as row and column swaps. As a matter of efficiency, **one never copies rows or columns to perform swapping**. Copying matrix data requires unnecessary reading and writing of matrix data, which can be unexpectedly time consuming due to cache misses. Instead, you will always use an *index vector* to look up rows and columns. Consider a matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, which can be indexed in C or Python using the notation `A[r][c]`, where $r, c \in [0, N)$ are indices for rows and columns.

An index vector $\ell \in \mathbb{N}^N$ is used to perform column and row swaps by initializing the vector:

$$\ell = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ N-2 \\ N-1 \end{bmatrix}$$

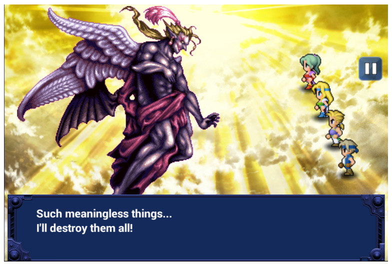
and using it as a proxy for row or column accesses: `A[l[r]][c]`. Two rows a and b are swapped switching the indices in `l[a]` and `l[b]`.

8.7.2 Performance and Complexity

Gaussian elimination is an $O\left(n^3\right)$ operation, which may be clear by looking at Algorithm 1 (Chapter 1). This algorithm is composed of three loops, with each dependent on the input size n. The backsubstitution algorithm (Algorithm 2) is $O\left(n^2\right)$.

Implementing partial pivoting requires an additional linear $O(n)$ search during every iteration of the outer loop. This search identifies the pivot equation by finding the largest leading coefficient. Since this search occurs inside the first loop, it requires an additional $\left(n^2\right)$ accesses. Both scaling and complete pivoting require identifying the largest coefficient in the remaining subset of the matrix, which is an $O\left(n^2\right)$ during each iteration of the outer loop - adding $O\left(n^3\right)$ operations to the total algorithm.

Partial pivoting is generally considered a requirement for stability. Given the minimal gains in accuracy demonstrated over partial pivoting, **scaling or complete pivoting are usually not implemented in practice**. Of course, exceptions to this rule can easily be found: if the user has prior knowledge that a linear system is poorly scaled, complete pivoting may provide critical improvements in accuracy that are worth the additional processing time. However, most pre-packaged algorithms perform GE using partial pivoting as the default.

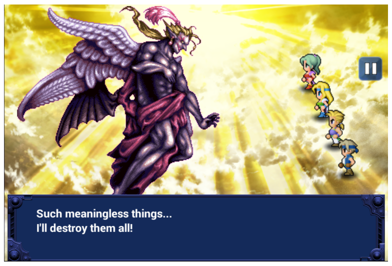


BLAS (1979-present, FORTRAN) The Basic Linear Algebra Subprograms library is one of the most common packages for performing low-level calculations using matrices. Functions are divided into three levels: level-1 operations are performed on vectors, level-2 operations include matrix-vector functions, and level-3 operations can take two matrices as operands. While the specification covers function parameters and output, the current reference libraries are open source and implemented in FORTRAN. (Wikipedia)

8.7.3 Software Packages

The two most common software packages used to for manipulating linear systems of equations are the Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) libraries. These libraries form the basis for both MATLAB and Python (through Numpy), and have C/C++ bindings available.

Both libraries are open source and written in FORTRAN, although highly optimized versions can be purchased for individual chipsets. They are commonly optimized and distributed with compilers designed by processor manufacturers. For example, the Intel compiler comes with pre-compiled versions of BLAS and LAPACK that are highly optimized for their chipsets. Alternative packages have also been designed for other platforms. For example, nVidia has a cuBLAS library that is optimized for processing large matrices using their graphics processing unit (GPU) architecture.



LAPACK (1992-present, FORTRAN) LAPACK is a popular and highly-optimized linear algebra library designed to perform more complex functions such as matrix factorization, eigendecomposition, and inversion. These libraries make extensive use of BLAS, and form the basis for many linear algebra routines in Matlab and Python. Bindings are also available for C/C++ using LAPACKE. (Wikipedia)

Individual package functions are often referenced when discussing common linear algebra tasks. For example, the function for performing Gaussian elimination is `DGETRF`:

- **D**: double (64-bit) precision - other letters identify functions for 32-bit (S), 32-bit complex (C), and 64-bit complex (Z) values
- **GE**: specifies that the input is a general matrix
- **TRF**: specifies the algorithm - in this case *triangular factorization*, which is a more general term for Gaussian elimination

The `TRF` functions implemented in LAPACK use partial pivoting.

Exercises

Exercises for 8.4 Partial Pivoting

P8.1 Solve the following linear system using (1) Gaussian Elimination and (2) Partial Pivoting using 3 digits of precision. What is the worst relative error?

$$\mathbf{A} = \begin{bmatrix} .0133 & 1.24 \\ 2.34 & 1.77 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

where

$$\mathbf{x} = \begin{bmatrix} -0.184 \\ 0.808 \end{bmatrix}$$

P8.2 Solve the following linear system using partial pivoting and provide the final indexing vector.

$$\begin{bmatrix} 0 & 4 & 4 & 4 \\ 2 & 3 & 2 & 7 \\ 0 & 2 & 4 & 8 \\ 0 & 1 & 1 & 4 \end{bmatrix} \mathbf{x} = \begin{bmatrix} -4 \\ -20 \\ -10 \\ -6 \end{bmatrix} \tag{8.3}$$

Chapter 9

Properties of Linear Systems

Solving linear systems is an extremely common technique used in numerical computing for engineering and scientific applications. Gaussian elimination also plays an important role in many fundamental algorithms in linear algebra, including LU factorization and the calculation of matrix inverses. While techniques such as pivoting and scaling can help reduce roundoff errors when solving linear systems, some error propagation is unavoidable when using floating point. However, not all matrices are created equal and some linear systems are significantly more susceptible to error propagation. These are generally referred to as **ill conditioned** linear systems, and reflect cases where small variations in matrix coefficients can produce large changes in the calculated output.

In this chapter, we will learn to identify matrix properties that allow us to determine how sensitive linear systems are to perturbations. We will discuss:

- Methods for identifying singular matrices, which don't have a unique solution.
- Matrix factorization methods, which can be used to calculate the solution for any set of right-hand-side values.
- Matrix inversion and pseudoinversion.
- Methods for quantifying matrix scaling, including matrix and vector norms.
- Methods for determining matrix conditioning and identifying ill-conditioned linear systems.

9.1 LU Decomposition

When working with linear systems, it is often useful to *decompose* or *factorize* a matrix into a product of multiple matrices (ex. $\mathbf{A} = \mathbf{BC}$). LU-decomposition is the most common factorization method, producing a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$ such that:

$$\mathbf{LU} = \mathbf{A}$$

The simplest methods for performing LU decomposition utilize Gaussian elimination. In fact, the upper-triangular matrix generated with GE is a valid matrix for **U**.

9.1.1 Doolittle Algorithm

The *Doolittle algorithm* generates both **L** and **U** during the normal process of GE. Consider the matrix:

$$\mathbf{A} = \begin{bmatrix} 9 & 0 & 3 \\ 18 & 2 & 1 \\ 27 & 1 & 4 \end{bmatrix}$$

Applying Gaussian elimination produces a new matrix by:

1. multiplying the first row by $m_{1,2} = 2$ and subtracting it from the second row
2. multiplying the first row by $m_{1,3} = 3$ and subtracting it from the third row
3. multiplying the second row by $m_{2,3} = \frac{1}{2}$ and subtracting it from the third row

This process can also be expressed in matrix form:

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & -\frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 9 & 0 & 3 \\ 18 & 2 & 1 \\ 27 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 9 & 0 & 3 \\ 0 & 2 & -4 \\ 0 & 0 & -3 \end{bmatrix}$$

where the the lower-triangular matrix is *unitriangular* (the diagonal coefficients are all 1) and the off-diagonal components correspond to $-m_{1,2}$, $-m_{1,3}$, and $-m_{2,3}$. The inverse of a unitriangular matrix is:

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{T} & \mathbf{I} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{T} & \mathbf{I} \end{bmatrix} \tag{9.1}$$

which implies that this first step of GE can be *reversed* by changing the signs of the lower-triangular components:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 9 & 0 & 3 \\ 0 & 2 & -5 \\ 0 & 0 & -3 \end{bmatrix} = \begin{bmatrix} 9 & 0 & 3 \\ 18 & 2 & 1 \\ 27 & 1 & 4 \end{bmatrix}$$

The Doolittle algorithm decomposes a matrix **A** into **L** and **U** using the following steps:

1. Initialize $\mathbf{L} = \mathbf{I}$
2. Generate **L** using Gaussian elimination, storing the scale factors $L_{ij} = m_{ij}$
3. Return the final matrix in row-eschelon form as **U**

Note that this algorithm uses Gaussian elimination at its core and has a computational complexity of $O\left(n^3\right)$.

Example 9.1

Perform LU decomposition on the following matrix:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & -2 \\ 6 & 10 & -10 \\ 4 & 10 & -16 \end{bmatrix}$$

Solution: LU decomposition will be performed iteratively using Gaussian elimination. The first iteration of the output matrices are initialized as $\mathbf{U}_0 = \mathbf{A}$ and $\mathbf{L} = \mathbf{0}$:

$$\mathbf{U}_0 = \begin{bmatrix} 2 & 3 & -2 \\ 6 & 10 & -10 \\ 4 & 10 & -16 \end{bmatrix} \quad \mathbf{L}_0 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The value in $\mathbf{L}$ corresponding to the leading coefficient of the pivot is always 1. Elimination of rows 2 and 3 are performed by scaling the pivot row 1 by $m_{1\to 2} = 3$ and $m_{1\to 3} = 2$:

$$\mathbf{U}_1 = \begin{bmatrix} 2 & 3 & -2 \\ 0 & 1 & -4 \\ 0 & 4 & -12 \end{bmatrix} \quad \mathbf{L}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix}$$

Elimination of the final leading coefficient in row 3 is performed by scaling the pivot row 2 by $m_{2\to 3} = 4$: The final entry in $\mathbf{L}$ corresponding to the last coefficient is also set to 1:

$$\mathbf{U}_2 = \begin{bmatrix} 2 & 3 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 4 \end{bmatrix} \quad \mathbf{L}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 4 & 0 \end{bmatrix}$$

The final entry in $\mathbf{L}$ corresponding to the last coefficient is also set to 1:

$$\mathbf{U} = \begin{bmatrix} 2 & 3 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 4 \end{bmatrix} \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 4 & 1 \end{bmatrix}$$

We can validate the factorization by verifying that $\mathbf{LU} = \mathbf{A}$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 4 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 3 & -2 \\ 6 & 10 & -10 \\ 4 & 10 & -16 \end{bmatrix}$$

9.1.2 LUP Decomposition

Based on our previous discussion of the instability of Gaussian elimination (Chapter 8), the most obvious problem we face is implementing a stable version of LU decomposition. As demonstrated previously, we can achieve a significant increase in stability by implementing partial pivoting. If rows are swapped during decomposition, we cannot guarantee that $\mathbf{LU} = \mathbf{A}$. This is addressed by using a permutation matrix $\mathbf{P}$ such that:

$$\mathbf{LU} = \mathbf{PA}$$

where **P** can be computed by recording the row ordering of the new matrix. The resulting *LUP decomposition* algorithm is most often used in linear algebra software, including LAPACK, SciPy, and MATLAB.

Example 9.2

Perform LUP decomposition on the following matrix:

$$\mathbf{A} = \begin{bmatrix} -4 & 4 & -5 \\ 4 & -1 & 4 \\ -16 & 4 & -20 \end{bmatrix}$$

Solution: LU decomposition will be performed iteratively using Gaussian elimination with partial pivoting. The first iteration of the output matrices are initialized as $\mathbf{U}_0 = \mathbf{A}$ and $\mathbf{L} = \mathbf{0}$, and the index vector is initialized to reference the non-permuted input matrix:

$$\mathbf{U}_0 = \begin{bmatrix} -4 & 4 & -5 \\ 4 & -1 & 4 \\ -16 & 4 & -20 \end{bmatrix} \quad \mathbf{L}_0 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \ell_0 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

The first pivot equation is selecting by finding the row with the largest leading coefficient (row 3). The row-swap is performed in ℓ:

$$\mathbf{U}_0 = \begin{bmatrix} -4 & 4 & -5 \\ 4 & -1 & 4 \\ -16 & 4 & -20 \end{bmatrix} \quad \mathbf{L}_0 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \ell_1 = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

The coefficients in the first row of **L** correspond to the scale factors used to eliminate the leading coefficients. For the pivot equation $\ell[1]$, the corresponding value is $m = 1$. For rows $\ell[2]$ and $\ell[3]$, the values are $m = -\frac{1}{4}$ and $m = \frac{1}{4}$ respectively:

$$\mathbf{U}_1 = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & -1 \\ -16 & 4 & -20 \end{bmatrix} \quad \mathbf{L}_1 = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{4} & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \ell_1 = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

The next pivot equation is selected as row 1, so the ℓ vector is once again updated:

$$\mathbf{U}_1 = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & -1 \\ -16 & 4 & -20 \end{bmatrix} \quad \mathbf{L}_1 = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{4} & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \ell_2 = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$$

The scale value corresponding to the leading coefficient in the pivot equation is always $m = 1$. Since the leading coefficient for row $\ell[3]$ is already zero, the scale value corresponding to this value is also zero and no changes are made to the upper triangular matrix array:

$$\mathbf{U}_2 = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & -1 \\ -16 & 4 & -20 \end{bmatrix} \quad \mathbf{L}_2 = \begin{bmatrix} \frac{1}{4} & 1 & 0 \\ -\frac{1}{4} & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \ell_2 = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$$

For the final step, $m = 1$ is inserted into **L** corresponding to the leading coefficient of the final pivot equation:

$$\mathbf{U}_3 = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & -1 \\ -16 & 4 & -20 \end{bmatrix} \quad \mathbf{L}_3 = \begin{bmatrix} \frac{1}{4} & 1 & 0 \\ -\frac{1}{4} & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \ell = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$$

This produces the final **L** and **U** matrices which, given the row ordering in ℓ are:

$$\mathbf{U} = \begin{bmatrix} -16 & 4 & -20 \\ 0 & 3 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{4} & 1 & 0 \\ -\frac{1}{4} & 0 & 1 \end{bmatrix}$$

The permutation matrix can be calculated by inserting a 1 into the column corresponding to the value of each row in ℓ:

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

We can validate this decomposition by verifying that $\mathbf{LU} = \mathbf{PA}$:

$$\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{4} & 1 & 0 \\ -\frac{1}{4} & 0 & 1 \end{bmatrix} \begin{bmatrix} -16 & 4 & -20 \\ 0 & 3 & 0 \\ 0 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -4 & 4 & -5 \\ 4 & -1 & 4 \\ -16 & 4 & -20 \end{bmatrix}$$

$$\begin{bmatrix} -16 & 4 & -20 \\ -4 & 4 & -5 \\ 4 & -1 & 4 \end{bmatrix} = \begin{bmatrix} -16 & 4 & -20 \\ -4 & 4 & -5 \\ 4 & -1 & 4 \end{bmatrix}$$

9.1.3 Solving Linear Systems Using LU Decomposition

Consider a set of linear systems using the same coefficients **A** with different right-hand-side values:

$$\mathbf{Ax}_1 = \mathbf{b}_1 \qquad \mathbf{Ax}_2 = \mathbf{b}_2 \qquad \mathbf{Ax}_3 = \mathbf{b}_3$$

Performing LU decomposition allows us to modify the traditional expression:

$$\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{LUx} = \mathbf{b}$$

This can be expressed as two separate equations:

$$\mathbf{Lz} = \mathbf{b}$$
$$\mathbf{Ux} = \mathbf{z}$$

where each equation involves a triangular matrix. The solution to $\mathbf{Lz} = \mathbf{b}$ can be calculated using forward substitution, and the final result $\mathbf{Ux} = \mathbf{z}$ can be calculated using back-substitution. This process is significantly faster when there are multiple right-hand-side vectors, since the $O\left(n^3\right)$ LU decomposition algorithm is only required to generate **L** and **U**. Each independent set of right-hand-side values $[\mathbf{b}_1, \mathbf{b}_2, \cdots]$ can be calculated using $O\left(n^2\right)$ forward and back-substitution.

9.2 Matrix Inversion

LU Decomposition also plays a fundamental role in calculating the inverse matrix $\mathbf{A}^{-1}$. Matrix inversion is generally unnecessary for solving linear systems, since LU decomposition is usually sufficient. However, the matrix inverse has a few practical and theoretical applications.

Consider the previously described problem involving multiple linear systems that use the same set of coefficients $\mathbf{A}$ with different right-hand-side values:

$$\mathbf{A}\mathbf{x}_1 = \mathbf{b}_1 \qquad \mathbf{A}\mathbf{x}_2 = \mathbf{b}_2 \qquad \mathbf{A}\mathbf{x}_3 = \mathbf{b}_3$$

These linear systems can be combined into a single expression:

$$\mathbf{A}\mathbf{X} = \mathbf{B} \quad \text{where} \quad \mathbf{X} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \\ | & | & \cdots & | \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_n \\ | & | & \cdots & | \end{bmatrix}$$

where each column in $\mathbf{X}$ can be solved using forward and back-substitution (Section 9.1.3). Since the matrix inverse is defined such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$, we can solve the linear system:

$$\mathbf{A}\mathbf{X} = \mathbf{I}$$

to find the solution $\mathbf{X} = \mathbf{A}^{-1}$.

Example 9.3

Calculate $\mathbf{A}^{-1}$ for the matrix:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & -2 \\ 6 & 10 & -10 \\ 4 & 10 & -16 \end{bmatrix}$$

Solution: We know from Example 9.1 that the corresponding $\mathbf{L}$ and $\mathbf{U}$ matrices are:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 4 & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} 2 & 3 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 4 \end{bmatrix}$$

Since we are solving the linear system:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

we can solve for each column of $\mathbf{A}^{-1}$ using the corresponding column of $\mathbf{I}$. Given $\mathbf{L}\mathbf{U} = \mathbf{A}$, this can be done using forward substitution to solve $\mathbf{L}\mathbf{z} = \mathbf{b}$ and backsubstitution to solve $\mathbf{U}\mathbf{x} = \mathbf{z}$. For the first column, we solve for $\mathbf{z}$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 4 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$z_1 = 1 \quad \Longrightarrow \quad \begin{aligned} 3z_1 + 1z_2 &= 0 \\ 3 + z_2 &= 0 \\ z_2 &= -3 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} 2z_1 + 4z_2 + 1z_3 &= 0 \\ 2 - 12 + z_3 &= 0 \\ z_3 &= 10 \end{aligned}$$

and then solve for **x**:

$$\begin{bmatrix} 2 & 3 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \\ 10 \end{bmatrix}$$

$$\begin{aligned} 4x_3 &= 10 \\ x_3 &= \frac{5}{2} \end{aligned} \quad \Longrightarrow \quad \begin{aligned} 1x_2 - 4x_3 &= -3 \\ x_2 - 10 &= -3 \\ x_2 &= 7 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} 2x_1 + 3x_2 - 2x_3 &= 1 \\ 2x_1 + 21 - 5 &= 1 \\ 2x_1 &= -15 \\ x_1 &= -\frac{15}{2} \end{aligned}$$

which yields the first column of the matrix inverse:

$$\mathbf{A}^{-1} = \begin{bmatrix} -\frac{15}{2} & ? & ? \\ 7 & ? & ? \\ \frac{5}{2} & ? & ? \end{bmatrix}$$

Solving for the remaining columns yields:

$$\mathbf{A}^{-1} = \begin{bmatrix} -\frac{15}{2} & \frac{7}{2} & -\frac{5}{4} \\ 7 & -3 & 1 \\ \frac{5}{2} & -1 & \frac{1}{4} \end{bmatrix}$$

9.3 Singular Systems and Determinants

A linear system is a set of equations representing hyperplanes in an n-dimensional space. The solution to the system is a single point where all of these planes intersect. This visualization makes it easier to understand that there are cases where finding a single solution would be impossible. In particular, if two equations represented planes that were perfectly parallel, this could cause two possible errors:

1. If the planes are separated by some value ϵ, they will never intersect and therefore no solution could be found incorporating both equations.
2. If $\epsilon = 0$, the planes represent the exact same equation, resulting in an infinite set of points where the linear system is satisfied.

In both cases, the linear system is said to be *singular* and has no unique solution. The standard test to determine if a matrix is singular is to calculate the *determinant*. The determinant of a matrix **A** is specified by the notation:

$$\det(\mathbf{A}) \qquad \text{or} \qquad |\mathbf{A}|$$

where $|\mathbf{A}| = 0$ if and only if the matrix is singular. A non-zero determinant also implies that the matrix inverse $\mathbf{A}^{-1}$ exists, and consequently **the determinant can only be calculated for square matrices**.

The determinant has several properties that are used in theoretical applications, and also provide us with options for calculating determinants:

1. $\det(\mathbf{I}) = 1$
2. $\det\left(\mathbf{A}^T\right) = \det(\mathbf{A})$
3. $\det\left(\mathbf{A}^{-1}\right) = \det(\mathbf{A})^{-1} = \frac{1}{\det(\mathbf{A})}$
4. $\det(c\mathbf{A}) = c^n \det(\mathbf{A})$ for $\mathbf{A} \in \mathbb{C}^{n\times n}$
5. $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B})$, note $\mathbf{A}$ and $\mathbf{B}$ must be square and the same size
6. If $\mathbf{A}$ is a triangular matrix, $\det(\mathbf{A})$ is the product of the coefficients on the diagonal:
$$\det(\mathbf{A}) = \prod_{i=1}^{n} A_{ii} \quad \text{if } \mathbf{A} \text{ is triangular}$$
7. If a single row swap is applied to $\mathbf{A}$ to produce a permuted matrix $\mathbf{A}'$, then $\det\left(\mathbf{A}'\right) = -\det(\mathbf{A})$

Properties (5) and (6) allow us to calculate the determinant of a matrix $\mathbf{A}$ using LU decomposition:
$$\det(\mathbf{A}) = \det(\mathbf{L})\det(\mathbf{U})$$

Since both $\mathbf{L}$ and $\mathbf{U}$ are triangular:
$$\det(\mathbf{A}) = \left(\prod_{i=1}^{n} L_{ii}\right)\cdot\left(\prod_{i=1}^{n} U_{ii}\right)$$

Finally, property (7) allows us to implement this algorithm using partial pivoting:
$$\det(\mathbf{A}) = (-1)^s \left(\prod_{i=1}^{n} L_{ii}\right)\cdot\left(\prod_{i=1}^{n} U_{ii}\right) \tag{9.2}$$

where s is the number of row swaps that occur during LU decomposition.

Note: Since $\det(\mathbf{A}) = 0$ indicates that $\mathbf{A}$ is singular, it is tempting to assume that a small determinant $\det(\mathbf{A}) = \epsilon$ means that a matrix is difficult to solve. **This is absolutely not the true.** For example, a linear system involving any diagonal matrix $\mathbf{D}$ can be trivially solved even if the product of the diagonal is small. Consider the simple case:
$$\epsilon\mathbf{Ix} = \mathbf{b}$$

where ϵ is small. Based on properties (1) and (4), the determinant is extremely small: $\det(\epsilon\mathbf{I}) = \epsilon^n$. However, this linear system can be trivially solved without loss of precision:
$$x_i = \frac{b_i}{\epsilon}$$

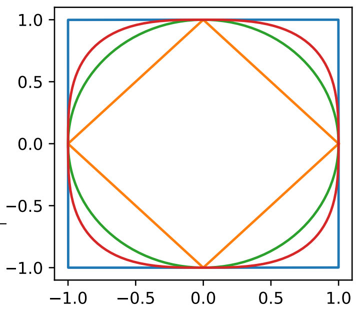


Figure 9.1 Coordinates where $\|\mathbf{x}\|_p = 1$ for $\mathbf{x} \in \mathbb{R}^2$ and $p = 1$ (orange), $p = 2$ (green), and $p = 3$ (red), and $p = \infty$ (blue).

9.4 Vector and Matrix Norms

A *norm* is a scalar value describing the "scale" or "size" of a vector or matrix. The common notation describing the norm n of a vector $\mathbf{x} \in \mathbb{C}^n$ is given by:

$$n = \|\mathbf{x}\|$$

where the norm is always real and positive:

$$\|\mathbf{x}\| \in \mathbb{R} \qquad \|\mathbf{x}\| \geq 0$$

9.4.1 Vector Norms

A *vector norm* is a scalar measure of the length of a vector. The most common set of metrics for measuring vector length are given by the p-norm:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}} = \sqrt[p]{\left(\sum_{i=1}^{n} |x_i|^p\right)} \tag{9.3}$$

While different values of p have a variety of applications, the most common norms you will encounter are:

- $\|\mathbf{x}\|_2$ is generally known as the *Euclidean norm*, and is most common distance measure. The calculation provides the length of the vector $\mathbf{x}$ based on the Pythagorean theorem: $d = \sqrt{x_1^2 + \cdots + x_n^2}$.
- $\|\mathbf{x}\|_1$ is generally known as the *Manhattan distance* or *Taxicab norm* because it reflects the distance between two points when only considering paths oriented along the Cartesian grid (ex. city streets).
- $\|\mathbf{x}\|_\infty$ is the *infinity norm* or *uniform norm*, which converges to the maximum value in $\mathbf{x}$, $\max(x_1, x_2, \cdots, x_n)$, as $p \to \infty$.
- $\|\mathbf{x}\|_0$ has gained prominence in the area of *compressive sensing*, where this norm provides the number of non-zero elements in $\mathbf{x}$ by defining $0^0 = 0$.

9.4.2 Matrix Norms

A *matrix norm* generalizes the concept of vector length to a matrix. The norm $\|\mathbf{A}\|$ specifies the "size" of $\mathbf{A}$ by quantifying how much an input vector $\mathbf{x}$ is scaled by the operation $\mathbf{y} = \mathbf{A}\mathbf{x}$.

For p-norms, the value $\|\mathbf{A}\|_p$ is defined by the *operator norm* that is *induced* by the corresponding vector p-norm. While the mathematical details of how the operator norm is calculated are only briefly described, it is sufficient to understand that the p-norm of a matrix is dependent on the corresponding vector p-norm given by Equation 9.3. However, the matrix and vector p-norm are not necessarily computed in the same way. The most common p-norm is the 2-norm, or Euclidean vector norm, which is simple to calculate for a vector $\mathbf{x}$:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

However, the matrix 2-norm induced by this vector norm is given by:

$$\|\mathbf{A}\|_2 = \sqrt{\lambda_n} \tag{9.4}$$

where λ_n is the largest eigenvalue of $\mathbf{A}^\dagger\mathbf{A}$ and $\mathbf{A}^\dagger$ is the conjugate transpose of $\mathbf{A}$. This 2-norm of a matrix is also known as the *spectral norm* and is the default norm calculated using most numerical algorithms.

Other common matrix norms include:

- The 1-norm for a matrix can be calculated by finding the column that has the largest absolute sum:

$$\|\mathbf{A}\|_1 = \max_{1\le j\le n} \sum_{i=1}^{n} |a_{ij}|$$

- The ∞-norm for a matrix can be calculated by finding the row with the largest absolute sum:

$$\|\mathbf{A}\|_\infty = \max_{1\le i\le n} \sum_{j=1}^{n} |a_{ij}|$$

- The *Frobenius norm*, which is often used as an upper-bound of the spectral norm, is calculated similar to the Euclidean distance across all matrix coefficients:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}|^2} \ge \|\mathbf{A}\|_2$$

9.4.3 Properties of Norms

Both vector and matrix norms have properties that are helpful for bounding error and deriving more specific properties:

absolutely homogeneous:	$\|\alpha\mathbf{A}\| = \|\alpha\| \, \|\mathbf{A}\|$
sub-additive:	$\|\mathbf{A}+\mathbf{B}\| \le \|\mathbf{A}\| + \|\mathbf{B}\|$
positive-valued:	$\|\mathbf{A}\| \ge 0$
definite:	$\|\mathbf{A}\| = 0$ only if all elements of $\mathbf{A}$ are zero

9.5 Matrix Conditioning

In numerical analysis, we are often interested in how sensitive an algorithm is with respect to small changes in the input arguments. If small deviations in the input parameters causes a large change in the result, the function can be described as unstable or *ill-conditioned*. If an algorithm is ill-conditioned, a small change caused by noise or round-off error can produce unusable results.

Linear systems provide excellent examples for both well-conditioned and ill-conditioned problems. While singular systems (Section 9.3) are uncommon, we frequently encounter cases where two linear equations result in nearly parallel hyperplanes separated by a very small angle. In this case, a small change in the coefficients of the linear system or right-hand-side can produce large changes in the solution to the system.

We can quantify the conditioning of an algorithm using its *condition number*, which measures how much the output is expected to change due to a small deviation in the input. For a linear system represented by a matrix **A**, the condition number is given by:

$$\kappa(\mathbf{A}) = \|\mathbf{A}^{-1}\|\,\|\mathbf{A}\| \tag{9.5}$$

The condition number can be used to quantify precision loss when implementing a numerical method. Consider a function $y = f(x)$ implemented in a digital computer. If we increment the input by some value ϵ, the result can be approximated using the condition number:

$$y + \epsilon\kappa(x) \approx f(x+\epsilon)$$

If ϵ is the smallest possible increment using floating point, no output values between y and $y+\epsilon\kappa(x)$ in the range $[x,\ x+\epsilon]$ will be encountered. **Our output precision is limited to a spacing of $\epsilon\kappa(x)$.**

For example, a condition number of $\kappa(\mathbf{A}) = 10^k$ suggests that an input deviation of ϵ will produce an output deviation of $\epsilon 10^k$, resulting in a loss of up to k digits of precision. Similarly, a result of $\kappa(\mathbf{A}) = 2^b$ will result in a loss of between $\lfloor b \rfloor$ and $\lceil b \rceil$ bits of precision in the mantissa (see Chapter 6).

Example 9.4

Calculate the expected precision of the solution to a linear system where the matrix **A** is a 5×5 Hilbert matrix:

$$H_{ij} = \frac{1}{i+j-1}$$

Assume that the calculations are performed using 32-bit IEEE floating point hardware.

Solution: The 5×5 Hilbert matrix and its inverse are shown below:

$$\mathbf{A} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix} \quad \mathbf{A}^{-1} = 5\cdot \begin{bmatrix} 5 & -60 & 210 & -280 & 126 \\ -60 & 960 & -3780 & 5376 & -2520 \\ 210 & -3780 & 15876 & -23520 & 11340 \\ -280 & 5376 & -23520 & 35840 & -17640 \\ 126 & -2520 & 11340 & -17640 & 8820 \end{bmatrix}$$

We can calculate the condition number using several norms:

- Since both matrices are symmetric, the 1-norm and ∞-norm are identical:

$$\begin{aligned}\kappa_1(\mathbf{A}) = \kappa_\infty(\mathbf{A}) &= \left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}\right)\cdot 5\cdot(280+5376+23520+35840+17640) \\ &= 943656\end{aligned}$$

- Frobenius norm:

$$\begin{aligned}\kappa_F(\mathbf{A}) &= \sqrt{\sum_{i=1}^{5}\sum_{j=1}^{5} A_{ij}} \cdot \sqrt{\sum_{i=1}^{5}\sum_{j=1}^{5} A_{ij}^{-1}} \\ &\approx 1.5809 \cdot 304160 \\ &\approx 480849\end{aligned}$$

- The tightest bound on the condition number is provided by the 2-norm, which is the default norm for calculating the condition number using Python:

```
import numpy
import scipy.linalg
n=5                          #specify the matrix size
A = scipy.linalg.hilbert(n) #create a Hilbert matrix
k = numpy.linalg.cond(A)    #calculate the condition number
```

which gives us:

```
k = 476607.250241
```

We use the condition number to estimate the number of bits lost b:

$$\kappa(\mathbf{A}) = 2^b$$

$$b = \frac{\log \kappa(\mathbf{A})}{\log 2}$$

Since the 2-norm provides the tightest bound for the condition number, we can estimate the number of bits lost as:

$$b = 18.86 \quad \lfloor b \rfloor = 18 \quad \lceil b \rceil = 19$$

Since the IEEE 32-bit floating format uses a 24-bit mantissa (23-bits with 1 implied bit), **we would expect any solution to this linear system to have between** 5 **and** 6 **significant bits**. This is quite a significant drop in precision from the 24-bits supplied by the hardware, and is something that you should keep in mind when formulating problems and designing algorithms to solve them.

Exercises

Perform LUP decomposition on the following matrices. You can validate your results using a calculator capable of solving linear algebra problems. LUP decomposition is also implemented in Python using SciPy's `scipy.linalg.lu` function and MATLAB using the built-in `lu` function. **Note:** Some linear algebra packages, such as the SciPy, return $\mathbf{P}^{-1}$ such that $\mathbf{A} = \mathbf{P}^{-1}\mathbf{LU}$. Since the permutation matrix is orthonormal, either version can be readily calculated using $\mathbf{P}^{-1} = \mathbf{P}^T$.

Exercises for 9.1 LU Decomposition

P9.1 Calculate **L**, **U**, and **P** for the following 2 × 2 matrices and give their determinants:

$$\mathbf{A} = \begin{bmatrix} 1 & 7 \\ -2 & -6 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 4 & -9 \\ 3 & 2 \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} 3 & -5 \\ -1 & 6 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} -2 & -3 \\ 4 & -8 \end{bmatrix}$$

P9.2 Calculate **L**, **U**, and **P** for the following 3 × 3 matrices and give their determinants:

$$\mathbf{A} = \begin{bmatrix} -6 & -10 & 6 \\ 3 & 5 & 1 \\ -9 & 2 & -5 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -4 & -2 & 0 \\ 6 & -6 & -7 \\ -4 & 2 & -7 \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} -2 & -2 & -6 \\ 1 & 1 & 9 \\ 3 & 5 & 2 \end{bmatrix}$$

P9.3 Calculate **L**, **U**, and **P** for the following 4 × 4 matrices and give their determinants:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 2 & 0 & -6 \\ 4 & -11 & -6 & 28 \\ -1 & 5 & 11 & -12 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 4 & -3 & 0 & -2 \\ 0 & 3 & 2 & -5 \\ 4 & 3 & 0 & -16 \\ -4 & 6 & 2 & -7 \end{bmatrix}$$

P9.4 Calculate **L**, **U**, and **P** for the following 5 × 5 matrices and give their determinants:

$$\mathbf{A} = \begin{bmatrix} 2 & -5 & -4 & 2 & 2 \\ -2 & 9 & -1 & -5 & -1 \\ 2 & 3 & -12 & -7 & 8 \\ 0 & 4 & -5 & -5 & -4 \\ -2 & 13 & -4 & -5 & 20 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -3 & -5 & -1 & 3 & -2 \\ 6 & 5 & 0 & -10 & 8 \\ -3 & -5 & -6 & 1 & 2 \\ 12 & 5 & 3 & -23 & 11 \\ -3 & 5 & 13 & 17 & -7 \end{bmatrix}$$

Exercises for 9.4 Vector and Matrix Norms

P9.5 Approximate the $p = 2$ norm using the $p = 1$ and $p = \infty$ norms for the following matrices:

$$\mathbf{A} = \begin{bmatrix} 2 & -5 & -4 & 2 & 2 \\ -2 & 9 & -1 & -5 & -1 \\ 2 & 3 & -12 & -7 & 8 \\ 0 & 4 & -5 & -5 & -4 \\ -2 & 13 & -4 & -5 & 20 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -3 & -5 & -1 & 3 & -2 \\ 6 & 5 & 0 & -10 & 8 \\ -3 & -5 & -6 & 1 & 2 \\ 12 & 5 & 3 & -23 & 11 \\ -3 & 5 & 13 & 17 & -7 \end{bmatrix}$$

Exercises for 9.5 Matrix Conditioning

P9.6 Consider solving $\epsilon\mathbf{I}\mathbf{x} = \mathbf{b}$, where ϵ is small. Compare the determinant of $\epsilon\mathbf{I}$ to its condition number.

P9.7 Calculate the condition number of a matrix generated using the 3×3 geometric series (Section 8.2).

P9.8 The condition number of a Hilbert matrix scales as $O\left(\frac{(1+\sqrt{2})^{4n}}{\sqrt{n}}\right)$. What is the largest Hilbert matrix that can be numerically solved using IEEE octuple (256-bit) precision (1 signed bit, 236-bit mantissa, 19-bit exponent) while still providing at least a single bit of precision.

Chapter 10

Calculus

10.1 Finite Difference Methods

10.1.1 Forward and Backward Differences

10.1.2 Central Differences

10.2 Higher-Order Methods

Exercises

Exercises for 10.1 Finite Difference Methods

P10.1 Compare the relative error using the central difference method to approximate

$$\frac{d}{dx}e^{-x}\sin\left(\frac{x^2}{2}\right)$$

at $x = 2$ using spacing $h = 1, 0.1,$ and 0.01 keeping 3 significant digits.

P10.2 The `sinc` function is defined as:

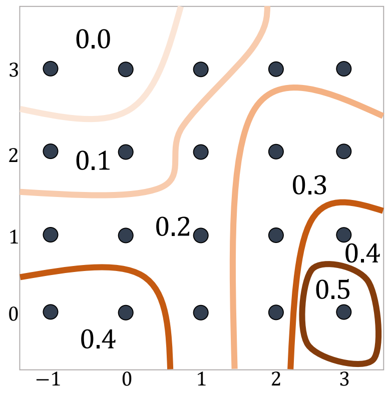
$$\text{sinc}(x) = \frac{\sin x}{x}$$

where $\text{sinc}(0) = 1$. Compare the relative error of the forward, backward, and central difference methods for approximating

$$\frac{d}{dx}\text{sinc}(x)$$

at $x = 0.01$ using a spacing of $h = 0.01$ and keeping 3 significant digits.

P10.3 Calculate the gradient $\nabla I = \begin{bmatrix} \frac{dI}{dx} & \frac{dI}{dy} \end{bmatrix}^T$ of the function $I(x, y)$, represented using the following contour map, at $(3, 1)$ using the best possible approximation (forward, backward, or central differences):



10.3 Newton-Cotes Quadrature Rules

10.3.1 Trapezoid Rule

10.3.2 Simpson's Rule

10.4 Gaussian Quadrature

10.5 Monte-Carlo Integration

Exercises

Compare the relative errors for the following integrals using two iterations of the trapezoid rule and one iteration of Simpson's rule. Use three digits of precision where appropriate.

Exercises for 10.3 Newton-Cotes Quadrature Rules

P10.4 $\int_0^1 x(x-1)(x-2)\,dx$

P10.5 $\int_0^1 \sqrt{1-x^4}\,dx = 0.874$

P10.6 $\int_0^1 \sin x^2\,dx = 0.310$

P10.7 $\int_0^1 e^{e^x}\,dx = 6.32$

P10.8 $\displaystyle\int_5^{10} \frac{1}{\ln x}\, dx = 2.53$

P10.9 $\displaystyle\int_{0.1}^{1} \frac{e^x}{x}\, dx = 3.52$

P10.10 $\displaystyle\int_0^{\pi} \frac{\sin x}{x}\, dx = 1.85$ (`sinc` function)

P10.11 $\displaystyle\int_0^{2} e^{-\frac{x^2}{2}}\, dx = 1.20$ (Gaussian function)

Chapter 11

Differential Equations

Differential equations are frequently encountered in science and engineering. For example, Newton's equations of motion define the velocity of an object over time t as the derivative of its position $\mathbf{p}(t)$:

$$\mathbf{v} = \frac{d}{dt}\mathbf{p}(t) = \mathbf{p}'(t)$$

If the velocity of the object is known, its position at any time can be analytically calculated by integrating both sides:

$$\mathbf{p}(t) = \int \mathbf{v} = \mathbf{v}t + C$$

The unknown value C reflects the fact that we require additional information: the position of the object at any time. If we we know that the object is at $\mathbf{p}_0$ at time $t = 0$:

$$\begin{aligned} \mathbf{p}_0 &= C \\ \mathbf{p}(t) &= \mathbf{v}t + \mathbf{p}_0 \end{aligned}$$

Unfortunately, the vast majority of differential equations encountered in practice have *no known analytical solution.* We must therefore rely on finding these solutions *numerically* using a set of computational tools. This chapter will focus on several common numerical techniques for solving differential equations, including:

- Visualizing differential equations as slope fields
- Using Euler's method to solve differential equations
- Analyzing the numerical error introduced by explicit methods
- Reducing error by using higher-order approximations
- Designing programs to solve problems involving differential equations

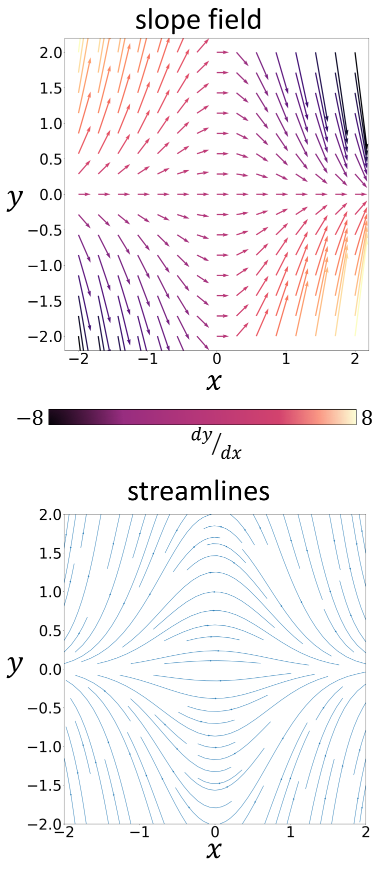


Figure 11.1 The slope field (top) corresponding to the differential equation $\frac{dy}{dx} = -2xy$ is shown, along with streamlines created for various initial values (x_0, y_0) (bottom). Note that there are an infinite number of functions that satisfy this equation, each with a different shape depending on the position of the initial point.

11.1 Slope Fields

First-order differential equations of two variables can be readily visualized graphically using a *slope field* by expressing the derivative as a function of both variables. For example, the values for the differential equation

$$\frac{dy}{dx} = -2xy$$

can be plotted using a vector field to show the slope for various values of x and y (Figure 11.1, **top**). This provides a useful visualization for understanding the shape of equations that satisfy this differential equation.

One could imagine placing a particle at some initial value (x_0, y_0) and moving it through the Cartesian space based on the direction of the slope field. By tracking its position as a *streamline*, we could plot a function that satisfies the differential equation for various initial values (Figure 11.1, **bottom**). This *particle tracking* concept forms the foundation for the set of *explicit* methods that we will discuss in the following sections.

11.2 Euler Method

One of the most fundamental differential equations is given by:

$$y' = y \quad \text{(Lagrange notation)} \qquad \frac{dy}{dx} = y \quad \text{(Lebnitz's notation)} \tag{11.1}$$

where the solution is the exponential function

$$\begin{aligned} \frac{1}{y} dy &= dx \\ \int \frac{1}{y} dy &= \int dx \\ \ln y &= x + C \\ y &= e^{x+C} \end{aligned}$$

This solution represents a family of functions specified by the value of C. For a unique solution, also need a known pair of values (x_0, y_0) such that $y(x_0) = y_0$. This allows us to solve for C:

$$\begin{aligned} \ln y_0 &= x_0 + C \\ C &= \ln y_0 - x_0 \end{aligned}$$

and replace the unknown constant in the solution:

$$\begin{aligned} y &= e^{x + \ln y_0 - x_0} \\ y &= y_0 e^{x - x_0} \end{aligned}$$

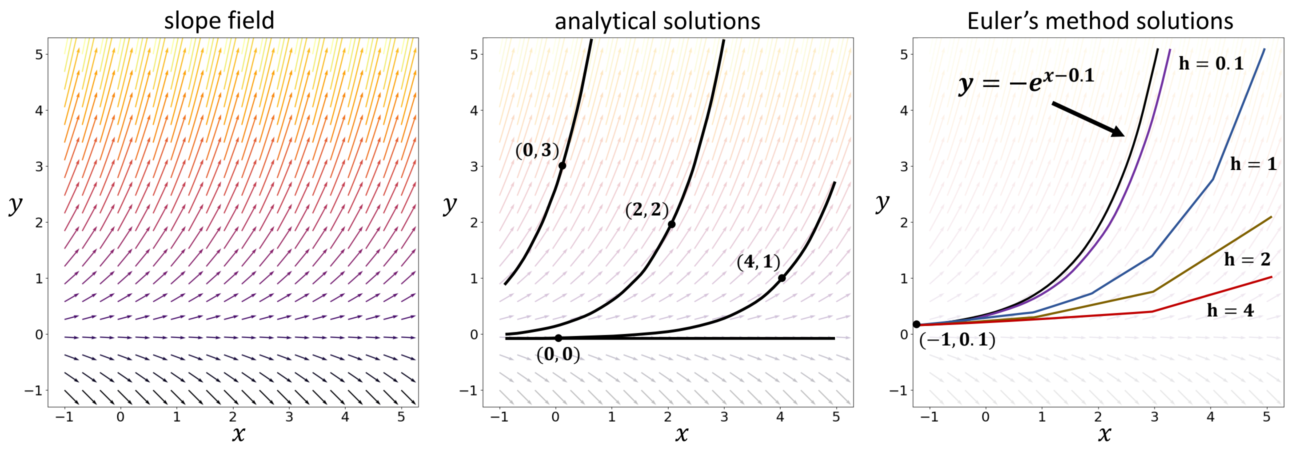


Figure 11.2 Analytical and numerical solutions are shown for the exponential function. The slope field for $\frac{dy}{dx} = y$ is shown in the range $x, y \in [-1, 5]$ (left). Note that this field captures the shape given by the analytical solutions for various initial values (center). Euler's method is compared to the analytical solution for $(-1, 0.1)$ using various values of h (right). Note that (1) iterative application of Euler's method results in cumulative error as x increases and (2) this error gets smaller as $h \to 0$.

This allows us to plot the solution for any pair of initial values (x_0, y_0) and compare it to the slope field for the original differential equation. Note how the slope field implicitly encodes the shape of the solution for any initial value (Figure 11.2).

We can use the differential equation and initial value to evaluate the shape of the solution using a first order Taylor series approximation:

$$y(x) = y(a) + y'(a)(x - a) + O\left[(x-a)^2\right]$$
$$y(x) \approx y(a) + y'(a)(x - a)$$

If we know the initial value $y(x_0) = y_0$, assigning $a = x_0$ yields:

$$y(x) \approx y_0 + y'(x_0)(x - x_0)$$

where the derivative $y'(x_0)$ is the value of the slope field and can be directly calculated using the provided differential equation. We can therefore estimate a nearby value $x = x_0 + h$ using:

$$y(x_0 + h) \approx y_0 + y'(x_0)h \tag{11.2}$$

where a smaller value h yields a better approximation to $y(x)$, and the error of this approximation scales with $O\left(h^2\right)$.

After recording the value of the solution at $x_0 + h$, we can perform the same approximation to estimate the next value at $x_0 + 2h$. This iterative approximation scheme is known as *Euler's method*, and can be concisely represented using the recursive expression:

$$y_{n+1} \approx y_n + f(x_n, y_n)h \tag{11.3}$$
$$x_{n+1} \approx x_n + h \tag{11.4}$$

where the function $f(x, y(x))$ is the differential equation and $y(x)$ is the desired solution. **While this function can be iteratively applied, the error is cumulative.** Each step of Euler's method introduces an additional error term proportional to $O(h^2)$.

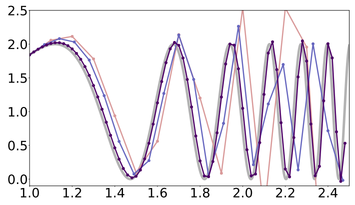


Figure 11.3 Numerical solution for $y' = 4x^3 \cos x^4$ using Euler's method for step sizes of $h = 0.1$ (red), $h = 0.07$ (blue), and $h = 0.02$ (purple). The analytical solution $y = \sin x^4$ is shown.

11.3 Heun's Method

A relatively simple optimization can be applied to Euler's method by noting that the error behaves consistently with respect to the curvature (Figure 11.3). Specifically:

1. If the solution $y(x)$ has curvature $\frac{d^2y}{dx^2} < 0$ - the function is **concave down** and Euler's method **overestimates** $y(x)$.
2. If the solution $y(x)$ has curvature $\frac{d^2y}{dx^2} > 0$ - the function is **concave up** and Euler's method **underestimates** $y(x)$.

In order to find a better approximation, it is best to re-consider the problem. Remember that our goal is to find the solution $y(x)$ to the ordinary differential equation $f(x, y(x)) = y'(x)$, which can be expressed as:

$$y(x) = \int f\left(x, y(x)\right) dx$$

where some initial value $y(x_0) = y_0$ is known. Explicit methods iteratively solve the definite integral:

$$y_{n+1} = y_n + \int_{x_n}^{x_n+h} f\left(x, y(x)\right) dx \tag{11.5}$$

starting with the initial condition $y(x_0) = y_0$. Euler's method uses a *Riemann sum* of rectangles (Figure 11.4) to solve this definite integral.

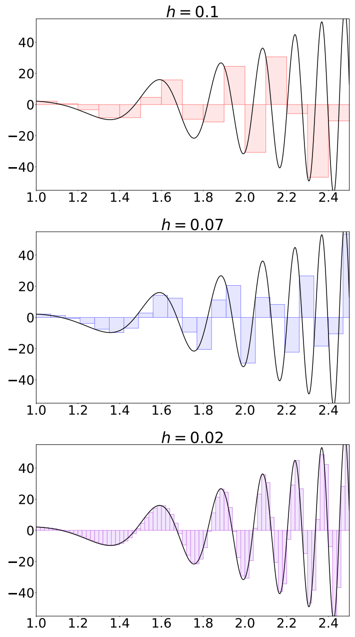


Figure 11.4 Numerical integration of $y' = 4x^3 \cos x^4$ using a Riemann sum for step sizes of $h = 0.1$, $h = 0.07$, and $h = 0.02$. Euler's method is the equivalent to calculating a Riemann sum using rectangles. This error results in over-estimation of the integral when the the solution $y(x)$ is concave down and underestimation when $y(x)$ is concave up.

We have already discussed higher order methods for solving definite integrals (Chapter **??**). By reformulating the indefinite integral using the trapezoid rule, we could theoretically improve the error term to $O(h^3)$:

$$\begin{aligned} y_{n+1} &= y_n + \int_{x_n}^{x_n+h} f\left(x, y(x)\right) ds \\ &= y_n + \frac{h}{2}\left[f\left(x_n, y_n\right) + f\left(x_n + h, y(x_n + h)\right)\right] \\ &= y_n + \frac{h}{2}\left[f\left(x_n, y_n\right) + f\left(x_{n+1}, y_{n+1}\right)\right] \end{aligned}$$

Of course, we run into an obvious problem: we do not have access to the value for y_{n+1}. In fact, this is what we are trying to calculate! We instead calculate an approximation $\hat{y}_{n+1}$ by **taking one step of Euler's method** and using the approximation to implement the trapezoid rule. This results in the following algorithm, known as *Heun's method*:

1. Calculate an initial *guess* $\hat{y}(x_{n+1})$ using Euler's method.
2. Use the average of $f\left(x_n, y_n\right)$ and $f\left(x_{n+1}, \hat{y}_{n+1}\right)$ for the next explicit step:

$$y_{n+1} = y_n + \frac{h}{2}\left[f\left(x_n, y_n\right) + f\left(x_{n+1}, \hat{y}_{n+1}\right)\right] \tag{11.6}$$

The intuition behind Heun's method is easier to understand by returning to the context of a slope field (Figure 11.5). An initial guess $\hat{y}_{n+1}$ is made for the solution $y(x)$. Based on the information encoded in $f\left(x, y(x)\right)$, this guess is *corrected* to provide the more accurate value y_{n+1}. Because of these prediction and correction steps, algorithms like Heun's method are often referred to as *predictor-corrector* algorithms.

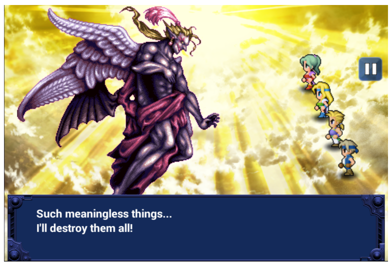

Such meaningless things...
I'll destroy them all!

Figure 11.5 Heun's method as a predictor-corrector algorithm. An initial guess $\hat{y}_{n+1}$ is made for the solution $y(x)$. Based on the slope field at the new location, y_{n+1} is calculated by *correcting* the guess.

Heun's Method from Taylor Series

Like Euler's method, Heun's method can also be derived using a Taylor series approximation. This is particularly useful when deriving higher-order methods. Consider the ordinary differential equation:

$$\frac{d}{dx}y(x) = f(x, y(x)) \tag{11.7}$$

We can calculate a second-order approximation to y as a Taylor series:

$$y(x+h) = y(x) + hy'(x) + \frac{h^2 y''(x)}{2} + O\left(h^3\right)$$

The second derivative term can be calculated from Equation 11.7 using the chain rule:

$$\begin{aligned}\frac{d^2}{dx^2}y(x) &= \frac{d}{dx}f(x,y) + \frac{d}{dy}f(x,y)\frac{d}{dx}y(x)\\ &= \frac{d}{dx}f(x,y) + \frac{d}{dy}f(x,y)f(x,y)\end{aligned}$$

Inserting this solution into the Taylor series yields:

$$\begin{aligned}y(x+h) &= y(x) + hf(x,y) + \frac{h^2}{2}\frac{d^2}{dx^2}y(x) + O\left(h^3\right) &(11.8)\\ &= y(x) + hf(x,y) + \frac{h^2}{2}\left[\frac{d}{dx}f(x,y) + \frac{d}{dy}f(x,y)f(x,y)\right] + O\left(h^3\right) &(11.9)\\ &= y(x) + \frac{h}{2}f(x,y) + \frac{h}{2}\left[f(x,y) + h\frac{d}{dx}f(x,y) + h\frac{d}{dy}f(x,y)f(x,y)\right] + O\left(h^3\right) &(11.10)\end{aligned}$$

The second term can be further simplified using a first-order multidimensional Taylor expansion:

$$t(x,y) = t(a,b) + \frac{d}{dx}f(a,b)(x-a) + \frac{d}{dy}f(a,b)(y-b)$$

By substituting in the associated values for x, y, and h, and adding a new term $k = y - b$ (for the spacing between values in y), this can be reformulated as:

$$t(x+h, y+k) = t(x, y) + \frac{d}{dx}t(x, y)h + \frac{d}{dy}t(x, y)k \tag{11.11}$$

The $y + k$ term can be derived from Euler's method:

$$y(x+h) = y(x) + k \qquad \text{where} \qquad k = hf(x, y)$$

Substituting everything into the Taylor series in Equation 11.11, we get the expression:

$$f\left(x+h, y+hf(t, y)\right) = f(t, y) + h\frac{d}{dx}f(x, y) + h\frac{d}{dy}f(x, y)f(x, y) + O\left(h^2\right)$$

Note that the right-hand-side term in this expression is identical to the bracketed term in Equation 11.10, which can therefore be reformulated as:

$$y(x+h) = y(x) + \frac{h}{2}f(x, y) + \frac{h}{2}\left[f(x, y) + h\frac{d}{dx}f(x, y) + h\frac{d}{dy}f(x, y)f(x, y)\right] + O\left(h^3\right) \tag{11.12}$$

$$= y(x) + \frac{h}{2}f(x, y) + \frac{h}{2}f\left(x+h, y+hf(x, y)\right) + O\left(h^3\right) \tag{11.13}$$

This expression gives us another common expression for Heun's method:

$$y_{n+1} \approx y_n + \frac{h}{2}f(x_n, y_n) + \frac{h}{2}f\left(x_n, y_n + hf(x_n, y_n)\right) \tag{11.14}$$

$$\approx y_n + \frac{h}{2}(k_1 + k_2) \tag{11.15}$$

where

$$k_1 = f\left(x_n, y_n\right)$$
$$k_2 = f\left(x_n + h, y_n + hk_1\right)$$

Compare Equation 11.15 to its counterpart Equation 11.6. This is identical to Heun's method, where k_1 is the slope at x_n and k_2 is our initial guess of the slope at x_{n+1}. This formulation is the second-order formulation for a family of techniques known as *Runge-Kutta methods*, which we will discuss in the next section. The terms k_1 and k_2 are sometimes referred to as *Runge-Kutta terms*.

11.4 The Runge-Kutta Method

Runge-Kutta methods, named after Carl David Tolme Runge and Martin Wilhelm Kutta, form a family of popular numerical techniques for solving differential equations of the form:

$$y'(x) = f\left(x, y(x)\right)$$

We have already discussed the two most basic Runge-Kutta methods. Euler's method, or first-order Runge-Kutta, is given by:

$$y_{n+1} = y_n + hk_1$$

where

$$k_1 = f(x, y)$$

Heun's method, or second-order Runge-Kutta, is given by:

$$y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2)$$

where

$$k_1 = f(x, y)$$
$$k_2 = f\left(x_n + h, y_n + hk_1\right)$$

The most popular Runge-Kutta method, referred to simply as *The* Runge-Kutta method, is fourth-order:

$$y_{n+1} \approx y_n + h\left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\right) \tag{11.16}$$

where

$$k_1 = f\left(x_n, y_n\right) \tag{11.17}$$
$$k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \tag{11.18}$$
$$k_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \tag{11.19}$$
$$k_4 = f\left(x_n + h, y_n + hk_3\right) \tag{11.20}$$

11.5 Error and Time Complexity

Euler's method, or first-order Runge-Kutta, relies on a first-order Taylor series approximation and therefore has an error of $O\left(n^2\right)$ while Heun's method has an error term proportional to $O\left(n^3\right)$ (see Equation 11.10).

The time required to approximate the solution to f must consider the number of times f is evaluated. Fourth-order Runge-Kutta, or RK4, has an error of $O\left(h^5\right)$ and requires four function evaluations for each step of size h in order to calculate the associated Runge-Kutta terms k_1 through k_4. **While higher-order methods can be derived, they are generally considered inefficient because an increasing number of evaluations for $f(x, y)$ are required**. For example, the 5th order method requires 6 evaluations and the 8th order method requires 11.

Example 11.1

Calculate the solution to $\frac{dy}{dx} = \sin\left(y + x^2\right)$ at $y(1)$ given an initial value of $y(0) = 2$.

Solution: This differential equation does not have a numerical solution. We must therefore use a numerical integration method. The primary limitation of explicit integration is computation time, we we will impose a limit of 4 evaluations for the differential equation.

Using Euler's method, or RK1, 4 evaluations allows us to approximate $y(1)$ with a step size of $h = 0.25$. The results at each iteration are given in the following table:

x_n	y_n	$k_1 = f(x_n, y_n) = \frac{dy}{dx}$	$y_{n+1} = y_n + hk_1$
0.00	2.000	.9093	$2 + 0.25 \times .9093 = 2.227$
0.25	2.227	.7523	$2.227 + 0.25 \times .7523 = 2.345$
0.50	2.345	2.595	$2.345 + 0.25 \times 2.595 = 2.994$
0.75	2.994	3.557	$2.994 + 0.25 \times 3.557 = 3.884$
1.00	3.884		

Using Heun's method, or RK2, allows us to approximate $y(1)$ with a step size of $h = 0.50$ since it requires 2 evaluations per step. The results at each iteration are given in the following table:

x_n	y_n	k_1	$k_2 = f(x_n + h, y_n + hk_1)$	$y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2)$
0.00	2.000	.9093	.4229	2.333
0.50	2.333	.5300	−.4407	2.022
1.00	2.022			

Since RK4 requires 4 evaluations, we must use a single step of $h = 1.0$. Evaluating each of the Runge-Kutta constants yields:

$$k_1 = f(x_n, y_n) = \sin\left(0^2 + 2\right) = 0.9093$$
$$k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) = \sin\left(0.5^2 + 2 + 0.5k_1\right) = .4232$$
$$k_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) = \sin\left(0.5^2 + 2 + 0.5k_2\right) = .6288$$
$$k_4 = f\left(x_n + h, y_n + hk_3\right) = \sin\left(1.0^2 + 2 + 1.0k_3\right) = -.4682$$
$$y(1) \approx y_n + \frac{h}{3}\left(\frac{1}{2}k_1 + k_2 + k_3 + \frac{1}{2}k_4\right) = 2.0 + (.1516 + .1411 + .2096 - .0780) = 2.424$$

An extremely accurate numerical solution implemented in Python with a time step $h = .0001$ yields a result of $y(1) = 2.431$. Calculating the relative error for each

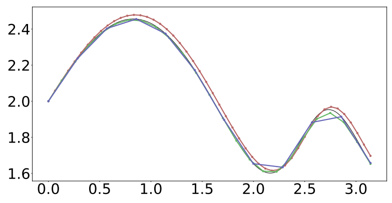


Figure 11.6 A comparison of the numerical solution to $\frac{dy}{dx} = \sin\left(y + x^2\right)$ using Euler's method (red), Heun's method (green), and 4th-order Runge-Kutta (blue). Comparable step sizes are used to ensure that the same number of evaluations are used to compute each solution.

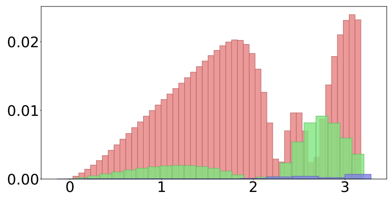


Figure 11.7 The relative error is plotted for each of the numerical evaluations in Figure 11.6. The thickness of each bar reflects the time step used for each algorithm.

algorithm gives:

$$\begin{aligned}
\text{RK1:} \quad & \frac{|3.884 - 2.431|}{2.431} = 59.8\% \\
\text{RK2:} \quad & \frac{|2.022 - 2.431|}{2.431} = 16.8\% \\
\text{RK4:} \quad & \frac{|2.424 - 2.431|}{2.431} = 0.28\%
\end{aligned}$$

11.6 Partial Differential Equations

Explicit integration methods, such as Runge-Kutta, can be readily applied to multivariate partial differential equations by considering the differential equation for each coordinate independently. These methods are often used to calculate *streamlines* indicating the local orientation of vector fields describing physical phenomena such as fluid flow and electromagnetic fields.

Consider the following set of partial differential equations that define the path of a mass-less particle over time:

$$\frac{dx}{dt} = V_x(x, y)$$
$$\frac{dy}{dt} = V_y(x, y)$$

While it may be easier to think of each variable independently, it is generally more convenient to express systems of PDEs in a vector format:

$$\frac{d\mathbf{p}}{dt} = \mathbf{V}(\mathbf{p}) \quad \text{where} \quad \mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{and} \quad \mathbf{V}(\mathbf{p}) = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{bmatrix}$$

The vector field **V** can be constructed using a variety of physical models. For example, consider an electric field generated by a set of point charges at the following locations and Coulomb force:

$$\mathbf{e}_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \quad \text{with charge} \quad c_1 = 1C$$
$$\mathbf{e}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{with charge} \quad c_2 = -1C$$
$$\mathbf{e}_3 = \begin{bmatrix} -\frac{1}{2} \\ 0 \end{bmatrix} \quad \text{with charge} \quad c_3 = -\frac{1}{10}C$$
$$\mathbf{e}_4 = \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix} \quad \text{with charge} \quad c_4 = -\frac{3}{10}C$$

The corresponding electric field is given by:

$$\mathbf{E}(\mathbf{p}) = \sum_{i=1}^{N} \frac{c_i}{4\pi\epsilon_0 |\mathbf{e}_i - \mathbf{p}|^2} \mathbf{v} \quad \text{where} \quad \mathbf{v} = \frac{\mathbf{e}_i - \mathbf{p}}{|\mathbf{e}_i - \mathbf{p}|}$$

where ϵ_0 is the *electrical constant* and $\mathbf{v}$ is the unit vector pointing from $\mathbf{p}$ to the corresponding charge at $\mathbf{e}_i$. Vector fields of this type are extremely common in physics. Like many physical forces, electrical fields obey the *inverse square law*: the force applied decreases with the square of the distance from the source. Many physical phenomena, including electrostatic forces, gravity, and sound, behave this way.

We can track a mass-less particle through the field using any of explicit Runge-Kutta methods described previously. Figure 11.8 shows the result of using Euler integration for various step sizes, where each coordinate is solved independently.

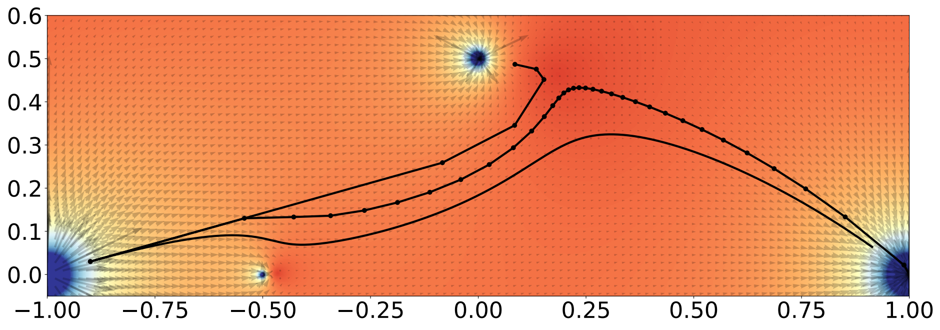


Figure 11.8 Tracking the path of a mass-less particle through an electromagnetic field with multiple positive and negative charges. The field vector is shown (arrows), along with the field magnitude (color). Euler integration is used with step sizes of $h = 0.08$, $h = 0.04$, and $h = 0.0001$. Points where the PDE is evaluated are shown (dots). Note that the step size effects both the path and final particle position.

11.7 Higher-Order Differential Equations

Runge-Kutta methods can also be used to solve higher (2+) order differential and partial differential equations. Consider the electrical field example provided in the previous section. This approach is purely theoretical - allowing us to visualize the structure of the field by plotting the path of mass-less particles. In a more realistic problem, the particle being plotted has mass (as well as velocity and potentially its own ability to accelerate). How can we numerically approximate the path of an object under a set of complex realistic forces?

Consider a relatively simple gravitational problem: determining the path of a satellite in orbit. The path of our satellite over time t is given by the function $\mathbf{p}(t)$. For simplicity, we will just consider the two-dimensional problem where

$$\mathbf{p}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$$

Newton's second law of motion allows us to define two other important properties of our satellite - its velocity and acceleration over time:

$$\mathbf{v}(t) = \frac{d\mathbf{p}}{dt}(t) \quad \text{and} \quad \mathbf{a}(t) = \frac{d\mathbf{v}}{dt}(t) = \frac{d\mathbf{p}^2}{d^2t}(t)$$

Finally, we know that the force on an object is proportional to its mass and acceleration:

$$\mathbf{F}(t) = m\mathbf{a}(t) \tag{11.21}$$

Given an initial position $\mathbf{p}_0 = \mathbf{p}(0)$, velocity $\mathbf{v}_0 = \mathbf{v}(0)$, and a physical model of the forces $\mathbf{F}$ applied to the satellite, we will approximate the position of the satellite $\mathbf{p}(t)$ at any time t.

Assuming that we had the velocity $\mathbf{v}(t)$ available to us, we could approximate the solution using Euler's method:

$$\begin{aligned}\mathbf{p}_n &= \mathbf{p}_{n-1} + h\mathbf{v}(t_{n-1})\\ t_n &= t_{n-1} + h\end{aligned}$$

This is a first-order PDE, since the known velocity $\mathbf{v}(t)$ is the derivative of the position $\mathbf{p}(t)$. While we don't have access to $\mathbf{v}(t)$, we do have an initial velocity $\mathbf{v}_0$, and could therefore approximate it using the same explicit technique given a known acceleration $\mathbf{a}(t)$:

$$\begin{aligned}\mathbf{p}_n &= \mathbf{p}_{n-1} + h\mathbf{v}_{n-1}\\ \mathbf{v}_n &= \mathbf{v}_{n-1} + h\mathbf{a}(t_{n-1})\\ t_n &= t_{n-1} + h\end{aligned}$$

This is a second-order PDE, solved recursively using Euler's method. We first use the known velocity to update the position. We then update the velocity $\mathbf{v}(t) = \mathbf{p}'(t)$ using the known acceleration $\mathbf{a}(t) = \mathbf{v}'(t) = \mathbf{p}''(t)$.

This exhausts our initial values, and we are now left with finding a way to calculate the acceleration $\mathbf{a}(t)$. We can reformulate Newton's second law of motion (Equation 11.21) to provide the acceleration:

$$\mathbf{a}(t) = \frac{1}{m}\mathbf{F}(t)$$

In our simple model, the only force acting on the satellite is the gravitational force applied by the Earth. One simple approach is to define a Cartesian coordinate system with the Earth at the origin: $\mathbf{e} = [0,0]^T$. Newton's law of universal gravitation, like an electric field, obeys the inverse square law:

$$F = G\frac{m_1 m_2}{r^2}$$

where F is the force acting between two objects with masses m_1 and m_2, r is the distance between these objects, and G is a gravitational constant:

$$G = 6.674 \times 10^{-11}\frac{Nm^2}{kg^2}$$

The force applied to the satellite is therefore dependent on the *position* of the satellite through the value r. Since the Earth is at the origin, $r = |\mathbf{p} - \mathbf{0}| = |\mathbf{p}|$. This force occurs in the direction of the Earth (which is pulling on the satellite):

$$\mathbf{F} = G\frac{m_1 m_2}{r^2}\hat{\mathbf{r}} \quad \text{where} \quad \hat{\mathbf{r}} = \frac{\mathbf{p}}{|\mathbf{p}|}$$

The net force on the satellite is therefore a function of the spatial position of the satellite $\mathbf{p}$ relative to the Earth:

$$\mathbf{F}(\mathbf{p}) = G\frac{m_s m_e}{|\mathbf{p}|^2} \cdot \frac{\mathbf{p}}{|\mathbf{p}|}$$

where m_s is the mass of the satellite and m_e is the mass of the Earth. Substituting this into Equation 11.7, we can create an analytical expression for the acceleration:

$$\begin{aligned}\mathbf{a}(t) &= \frac{1}{m_s} G\frac{m_s m_e}{|\mathbf{p}|^2} \cdot \frac{\mathbf{p}}{|\mathbf{p}|} \\ &= G\frac{m_e}{|\mathbf{p}(t)|^3} \cdot \mathbf{p}(t)\end{aligned}$$

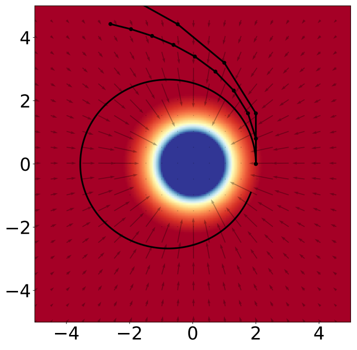


Figure 11.9 2-body simulation of a satellite path around a gravity well using various step sizes.

Using this result in our Euler approximation for this second-order PDE gives us:

$$\begin{aligned}\mathbf{p}_n &= \mathbf{p}_{n-1} + h\mathbf{v}_{n-1} \\ \mathbf{v}_n &= \mathbf{v}_{n-1} + hG\frac{m_e}{|\mathbf{p}_{n-1}|^3} \cdot \mathbf{p}_{n-1} \\ t_n &= t_{n-1} + h\end{aligned}$$

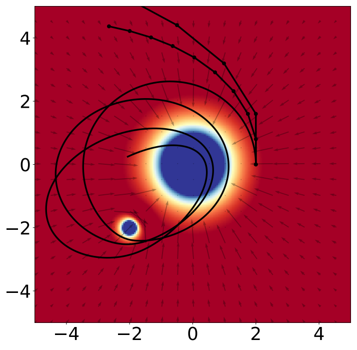


Figure 11.10 3-body simulation (ex. planet and moon).

What we have described is the numerical solution to the *n-body problem*, which describe the motion of n objects that exert gravitational force on each other. Our numerical solution provides an approximation to the 2-body problem. However, we are ignoring the force that the satellite applies to the Earth, which is negligible. An analytical solution to the 2-body problem was derived by Johann Bernoulli, however **cases for $n \geq 3$ rely on numerical methods like Runge-Kutta**.

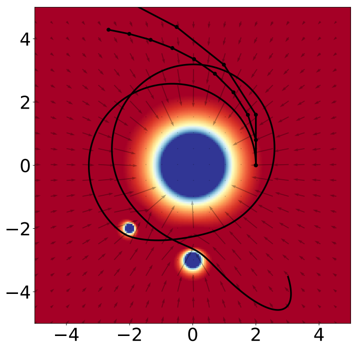


Figure 11.11 4-body simulation (ex. planet and two moons).

Recursive Solutions to Differential Equations

The 2-body problem described above can be rigorously expressed as the second-order partial differential equation:

$$\frac{d\mathbf{p}^2}{d^2 t} = G\frac{m_e}{|\mathbf{p}|^3} \cdot \mathbf{p}$$

This can be expressed using a more general notation, similar to our previous discussion on Runge-Kutta methods:

$$\mathbf{p}''(t) = \mathbf{f}(\mathbf{p}, t)$$

The solution can therefore be calculated using a recursive approach.

Euler's Method for High-Order Differential Equations

input:	slope field $f(\mathbf{x}, t)$
	maximum number of iterations N
	differential equation order $P \geq 2$
	array of initial values $\mathbf{v}[P]$
	step size h

```
output:  approximate solution $\mathbf{r}[N]$

$i = 0$
allocate $\mathbf{v}[N]$
$\mathbf{v}[0] = \mathbf{v}[0]$                          initialize using the initial value
while $i < N$:
    $\mathbf{r}[i] = \mathbf{v}[i-1] + h\mathbf{v}[1]$              store the next step in the solution
    for $p = 1$ to $P-2$
        $\mathbf{v}[p] = \mathbf{v}[p] + h\mathbf{v}[p+1]$          update intermediate values
    $\mathbf{v}[P-1] = \mathbf{v}[P-1] + h\mathbf{f}(\cdots, t)$    update using the PDE $\mathbf{f}(\cdots, t)$
output r
```

Exercises

Compare the relative error approximating solutions to the following differential equations after 5 steps of Euler's method using step sizes of $h = 0.1$ and 0.01.

Exercises for 11.2 Euler Method

P11.1 $y' = 2x\left(\cos x^2 - \sin x^2\right)$ for $y(0) = 1$
where the solution is given by $y = \cos x^2 + \sin x^2$

P11.2 $x^2 dy = (x\cos x - \sin x)dx$ for $y(0) = 1$
where the solution is given by $y(x) = \text{sinc}(x) = \dfrac{\sin x}{x}$.

P11.3 $y' = x\left(1 + 2\log|x|\right)$ for $y(0) = 0$
where the solution is given by $y = x^2 \log|x|$

P11.4 $dy = e^{-x}\left(2x\cos x^2 - \sin x^2\right)dx$ for $y(0) = 0$
where the solution is given by $y = \dfrac{\sin x^2}{e^x}\log|x|$

Write an algorithm to solve the following ordinary differential equations for several values in the given range. Plot the solutions and show how changes in h affect the result.

Exercises for 11.2 Euler Method

P11.5 $\dfrac{dy}{dx} = \sin\left(y + x^2\right)$ in the range $[-3,\ 4]$

P11.6 $\dfrac{dy}{dx} = y^2 - x$ in the range $[-4,\ -1]$

Programming Project - The Kessel Run

Problem: Design an algorithm to plot a path across a cluster of gravity wells.

The Kessel Sector is an area of space 20 parsecs (65.2 light years) wide and 10 parsecs (32.6 light years) across. For simplicity, we will align a compass such that traveling south-to-north will lead you across the thinnest region of this space (Figure 11.12). The entire region is interspersed with gravity wells with various masses. Your goal is to cross this sector from north to south by traversing as little space as possible while avoiding gravity wells.

Physics in the Kessel Sector obey Newton's laws of motion. You can use the following physical parameters to model this region of space:

- Your ship can start at any point within 5 parsecs of the center of the southern boundary.
- A successful path through the Kessel Sector will arrive at the northern boundary without leaving the sector.
- You can select any starting velocity, however the maximum speed of your ship under its own engine power is $1\,\mathrm{pc\,d^{-1}}$.
- Paths through the sector should be selected based on length, **not** time. Shorter paths are better than longer paths.
- It is too dangerous to operate your engines within the Kessel Sector. You have to shut them off and drift once you pass the southern border.
- The gravitational constant is $6.674 \times 10^{-11}\,\mathrm{m^3kg^{-1}s^{-2}}$, however it will be easier to frame this in parsecs and days: $G = 1.696 \times 10^{-50}\,\mathrm{pc^3kg^{-1}d^{-2}}$
- The maximum acceleration your ship can withstand is $1\,\mathrm{pc/d^2}$.
- The gravity well masses in this region range from 10^{49} to 10^{50}kg.

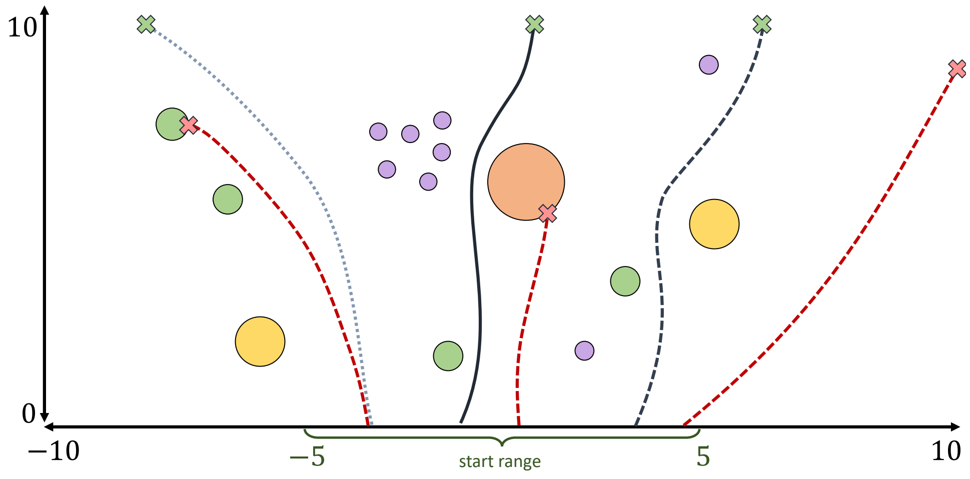


Figure 11.12 An example of traversing the Kessel Sector. A path is unsuccessful if it (a) exceeds the acceleration threshold for your ship or (b) leaves the sector. While there are an infinite number of paths through the Kessel Sector, you should select paths that traverse the least amount of space.

Chapter 12

Probability and Random Numbers

Many physical models and numerical methods can take advantage of random numbers to improve performance and increase accuracy. However, the deterministic nature of digital computers requires a unique set of algorithms for generating numbers that *behave* randomly. In this chapter, we will provide some background in probability theory and describe some basic methods used to simulate random numbers on deterministic machines.

12.1 Probability Theory

The *probability* of an event is a measure of how certain it is to occur. The probability of some event T is generally specified using the function $P(T) \in [0, 1]$, where a probability of 1 (100%) indicates absolute certainty that the event will occur and a probability of 0 (0%) indicates absolute certainty that it will **not** occur. While probabilities of 0 and 1 occur in theory, they rarely occur in practice. For example, consider the proposition: **There is a mouse living in a tea cup on the dark side of the moon.** Nobody that understands probability theory will say that this proposition has probability *zero*, however it is so minuscule as to be irrelevant. For a more practical example, consider **rolling a 7 on a six-sided die**. While the probability theoretically exists, it is generally unnecessary to model it because the cases in which it can happen (ex. the number on the die was misprinted) are extremely specific and generally not of interest.

12.2 Random Variables

We will often need to use an unknown random value in a mathematical expression. This value will be represented using a capital letter (ex. X). An *event* occurs when a random variable X is assigned some known value: $X = x$. The probability of that event occurring is given by $P(X = x)$. Assume that we assign the following binary values to a fair coin flip:

- 1 if the result is *heads*
- 0 if the result is *tails*

The probability of the coin flip event resulting in *heads* is $P(X = 1) = \frac{1}{2}$, or 50%.

A probability function can take multiple arguments, corresponding to the outcome of multiple variables. If we were to flip two coins: a nickel (X) and a quarter (Y), the probability that both land on *heads* is given by $P(X = 1, Y = 1)$. This is referred to as the *joint probability* of $X = x$ and $Y = y$.

Given that a probability function can take multiple variables, it is common to ask questions about the probability of events if one of the variables is constrained. Given two coins: a nickel (X) and a quarter (Y), we may express the probability of the nickel coming up *heads* if we know that the quarter came up *tails* as $P(X = 1 | Y = 0)$. This is known as the *conditional probability* and is frequently phrased as "the probability that $X = 1$ given that $Y = 0$."

12.2.1 Probability Mass Functions

12.2.2 Continuous Probability Density Functions

12.3 Linear Congruential Generators

12.3.1 Maximizing Period

12.3.2 Implementations

Exercises

Exercises for 12.3 Linear Congruential Generators

P12.1 Determine the period of a 5-bit MLCG with a modulus of 31 and a multiplier of 4 by selecting a seed and evaluating.

P12.2 Select the missing parameters to maximize the period for the following LCGs, where m is the modulus, a is the multiplier, and c is the increment:

1. $m = 24 \quad c = 7 \quad a =$??
2. $m =$?? $\quad c = 11 \quad a = 9$ (using a 5-bit register)
3. $m = 8 \quad c =$?? $\quad a = 5$ (list all possible values)

Chapter 13

Regression and Prediction

Exercises

Compare the relative errors for the following integrals using one iteration of the trapezoid rule and Simpson's rule. Use three digits of precision where appropriate.

Exercises for 10.3 Newton-Cotes Quadrature Rules

P13.1 $\int_0^1 x(x-1)(x-2)\,dx$

P13.2 $\int_0^1 \sqrt{1-x^4}\,dx = 0.874$

P13.3 $\int_0^1 \sin x^2\,dx = 0.310$

P13.4 $\int_0^1 e^{e^x}\,dx = 6.32$

P13.5 $\int_5^{10} \frac{1}{\ln x}\,dx = 2.53$

P13.6 $\int_{0.1}^1 \frac{e^x}{x}\,dx = 3.52$

P13.7 $\int_0^{\pi} \frac{\sin x}{x}\,dx = 1.85$ (`sinc` function)

P13.8 $\int_0^2 e^{-\frac{x^2}{2}}\,dx = 1.20$ (Gaussian function)

Index