COMPUTATIONAL ACOUSTICS
<table>
<thead>
<tr>
<th>Wiley Series in Acoustics, Noise and Vibration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computational Acoustics</strong></td>
</tr>
<tr>
<td>Wind Farm Noise</td>
</tr>
<tr>
<td>The Effects of Sound on People</td>
</tr>
<tr>
<td>Engineering Vibroacoustic Analysis:</td>
</tr>
<tr>
<td>Methods and Applications</td>
</tr>
<tr>
<td>Formulas for Dynamics, Vibration and Acoustics</td>
</tr>
</tbody>
</table>
## Contents

**Series Preface** ix

1 **Introduction** 1

2 **Computation and Related Topics** 5
   2.1 Floating-Point Numbers 5
      2.1.1 Representations of Numbers 5
      2.1.2 Floating-Point Numbers 7
   2.2 Computational Cost 9
   2.3 Fidelity 11
   2.4 Code Development 12
   2.5 List of Open-Source Tools 16
   2.6 Exercises 17
   References 17

3 **Derivation of the Wave Equation** 19
   3.1 Introduction 19
   3.2 General Properties of Waves 20
   3.3 One-Dimensional Waves on a String 23
   3.4 Waves in Elastic Solids 26
   3.5 Waves in Ideal Fluids 29
      3.5.1 Setting Up the Derivation 29
      3.5.2 A Simple Example 30
      3.5.3 Linearized Equations 31
      3.5.4 A Second-Order Equation from Differentiation 33
      3.5.5 A Second-Order Equation from a Velocity Potential 34
      3.5.6 Second-Order Equation without Perturbations 36
      3.5.7 Special Form of the Operator 36
      3.5.8 Discussion Regarding Fluid Acoustics 40
3.6 Thin Rods and Plates
3.7 Phonons
3.8 Tensors Lite
3.9 Exercises
References

4 Methods for Solving the Wave Equation
4.1 Introduction
4.2 Method of Characteristics
4.3 Separation of Variables
4.4 Homogeneous Solution in Separable Coordinates
  4.4.1 Cartesian Coordinates
  4.4.2 Cylindrical Coordinates
  4.4.3 Spherical Coordinates
4.5 Boundary Conditions
4.6 Representing Functions with the Homogeneous Solutions
4.7 Green’s Function
  4.7.1 Green’s Function in Free Space
  4.7.2 Mode Expansion of Green’s Functions
4.8 Method of Images
4.9 Comparison of Modes to Images
4.10 Exercises
References

5 Wave Propagation
5.1 Introduction
5.2 Fourier Decomposition and Synthesis
5.3 Dispersion
5.4 Transmission and Reflection
5.5 Attenuation
5.6 Exercises
References

6 Normal Modes
6.1 Introduction
6.2 Mode Theory
6.3 Profile Models
6.4 Analytic Examples
  6.4.1 Example 1: Harmonic Oscillator
  6.4.2 Example 2: Linear
6.5 Perturbation Theory
6.6 Multidimensional Problems and Degeneracy
6.7 Numerical Approach to Modes
  6.7.1 Derivation of the Relaxation Equation
  6.7.2 Boundary Conditions in the Relaxation Method
  6.7.3 Initializing the Relaxation
References
### Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7.4</td>
<td>Stopping the Relaxation</td>
<td>128</td>
</tr>
<tr>
<td>6.8</td>
<td>Coupled Modes and the Pekeris Waveguide</td>
<td>129</td>
</tr>
<tr>
<td>6.8.1</td>
<td>Pekeris Waveguide</td>
<td>129</td>
</tr>
<tr>
<td>6.8.2</td>
<td>Coupled Modes</td>
<td>131</td>
</tr>
<tr>
<td>6.9</td>
<td>Exercises</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>135</td>
</tr>
</tbody>
</table>

7 | Ray Theory | 137   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>137</td>
</tr>
<tr>
<td>7.2</td>
<td>High Frequency Expansion of the Wave Equation</td>
<td>138</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Eikonal Equation and Ray Paths</td>
<td>139</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Paraxial Rays</td>
<td>140</td>
</tr>
<tr>
<td>7.3</td>
<td>Amplitude</td>
<td>144</td>
</tr>
<tr>
<td>7.4</td>
<td>Ray Path Integrals</td>
<td>145</td>
</tr>
<tr>
<td>7.5</td>
<td>Building a Field from Rays</td>
<td>160</td>
</tr>
<tr>
<td>7.6</td>
<td>Numerical Approach to Ray Tracing</td>
<td>162</td>
</tr>
<tr>
<td>7.7</td>
<td>Complete Paraxial Ray Trace</td>
<td>168</td>
</tr>
<tr>
<td>7.8</td>
<td>Implementation Notes</td>
<td>170</td>
</tr>
<tr>
<td>7.9</td>
<td>Gaussian Beam Tracing</td>
<td>171</td>
</tr>
<tr>
<td>7.10</td>
<td>Exercises</td>
<td>173</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>174</td>
</tr>
</tbody>
</table>

8 | Finite Difference and Finite Difference Time Domain | 177   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>177</td>
</tr>
<tr>
<td>8.2</td>
<td>Finite Difference</td>
<td>178</td>
</tr>
<tr>
<td>8.3</td>
<td>Time Domain</td>
<td>188</td>
</tr>
<tr>
<td>8.4</td>
<td>FDTD Representation of the Linear Wave Equation</td>
<td>193</td>
</tr>
<tr>
<td>8.5</td>
<td>Exercises</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>197</td>
</tr>
</tbody>
</table>

9 | Parabolic Equation | 199   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>199</td>
</tr>
<tr>
<td>9.2</td>
<td>The Paraxial Approximation</td>
<td>199</td>
</tr>
<tr>
<td>9.3</td>
<td>Operator Factoring</td>
<td>201</td>
</tr>
<tr>
<td>9.4</td>
<td>Pauli Spin Matrices</td>
<td>204</td>
</tr>
<tr>
<td>9.5</td>
<td>Reduction of Order</td>
<td>205</td>
</tr>
<tr>
<td>9.5.1</td>
<td>The Padé Approximation</td>
<td>207</td>
</tr>
<tr>
<td>9.5.2</td>
<td>Phase Space Representation</td>
<td>208</td>
</tr>
<tr>
<td>9.5.3</td>
<td>Diagonalizing the Hamiltonian</td>
<td>209</td>
</tr>
<tr>
<td>9.6</td>
<td>Numerical Approach</td>
<td>210</td>
</tr>
<tr>
<td>9.7</td>
<td>Exercises</td>
<td>212</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>212</td>
</tr>
</tbody>
</table>

10 | Finite Element Method | 215   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>Introduction</td>
<td>215</td>
</tr>
<tr>
<td>10.2</td>
<td>The Finite Element Technique</td>
<td>216</td>
</tr>
</tbody>
</table>
## 10.3 Discretization of the Domain

10.3.1 One-Dimensional Domains

10.3.2 Two-Dimensional Domains

10.3.3 Three-Dimensional Domains

10.3.4 Using Gmsh

## 10.4 Defining Basis Elements

10.4.1 One-Dimensional Basis Elements

10.4.2 Two-Dimensional Basis Elements

10.4.3 Three-Dimensional Basis Elements

## 10.5 Expressing the Helmholtz Equation in the FEM Basis

## 10.6 Numerical Integration over Triangular and Tetrahedral Domains

10.6.1 Gaussian Quadrature

10.6.2 Integration over Triangular Domains

10.6.3 Integration over Tetrahedral Domains

## 10.7 Implementation Notes

## 10.8 Exercises

## References

## 11 Boundary Element Method

11.1 Introduction

11.2 The Boundary Integral Equations

11.3 Discretization of the BIE

11.4 Basis Elements and Test Functions

11.5 Coupling Integrals

11.5.1 Derivation of Coupling Terms

11.5.2 Singularity Extraction

11.5.3 Evaluation of the Singular Part

11.5.3.1 Closed-Form Expression for the Singular Part of K

11.5.3.2 Method for Partial Analytic Evaluation

11.5.3.3 The Hypersingular Integral

11.6 Scattering from Closed Surfaces

11.7 Implementation Notes

11.8 Comments on Additional Techniques

11.8.1 Higher-Order Methods

11.8.2 Body of Revolution

11.9 Exercises

## References

## Index
This book series will embrace a wide spectrum of acoustics, noise, and vibration topics from theoretical foundations to real-world applications. Individual volumes will range from specialist works of science to advanced undergraduate and graduate student texts. Books in the series will review scientific principles of acoustics, describe special research studies, and discuss solutions for noise and vibration problems in communities, industry, and transportation.

The first books in the series include those on *Biomedical Ultrasound; Effects of Sound on People, Engineering Acoustics, Noise and Vibration Control, Environmental Noise Management;* and *Sound Intensity and Windfarm Noise*. Books on a wide variety of related topics.

The books I edited for Wiley—*Encyclopedia of Acoustics* (1997), *The Handbook of Acoustics* (1998), and *Handbook of Noise and Vibration Control* (2007)—included over 400 chapters written by different authors. Each author had to restrict their chapter length on their special topics to no more than about 10 pages. The books in the current series will allow authors to provide much more in-depth coverage of their topic.

The series will be of interest to senior undergraduate and graduate students, consultants, and researchers in acoustics, noise, and vibration and in particular those involved in engineering and scientific fields, including aerospace, automotive, biomedical, civil/structural, electrical, environmental, industrial, materials, naval architecture, and mechanical systems. In addition the books will be of interest to practitioners and researchers in fields such as audiology, architecture, the environment, physics, signal processing, and speech.

*Malcolm J. Crocker*

Series Editor
1

Introduction

Computers have become an invaluable tool in science and engineering. Over time their use has evolved from a device to aid in complex lengthy calculations to a self-contained discipline or field of study. Coursework in science and engineering often involves learning analytic techniques and exact solutions to a sizable collection of problems. Although educational, these are rarely useful beyond the classroom. On the other side of the spectrum is the practical experimental approach to investigating nature and developing engineering solutions to practical everyday problems. Most readers are familiar with the nonideal nature of things that, in many cases, prevents one from seeing the utility of the theoretical approach. In science theorist and experimentalist see nature from a different perspective in a quest for understanding its laws but agree that the facts can only be found through observation, as patterns in data acquired via well-planned and executed experiments designed to isolate certain degrees of freedom, to replicate an ideal circumstance to the best of our ability. Experiments can be costly but are the only mechanism for determining scientific truth. While the scientist works to create ideal circumstances to verify a fundamental law or hypothesis, an engineer must design and build with non-ideal conditions in mind. In many cases the only approach available is trial and error. This requires the resources to build new versions of a device or invention every time it fails, each prototype being built and used to see what will happen and how it will fail and to learn from the experience. In this regard, computational science offers a path toward testing prototypes in a virtual environment. When executed carefully this approach can save time and money, prevent human injury or loss of life, and reduce impact to the environment.

As computers became larger, faster, and more efficient, the size of the tasks that could be performed also became larger, evolving from modeling the stress on a single beam to that found throughout the structure of a building, ship, or aircraft under dynamic loading. In recent times the use of computer-based modeling and simulation has gained a certain credibility in fields where there is no possible experimental method available and theory does not offer a suitable
path forward in exploring the consequences of natural law, in particular the fields of numerical relativity (NR) and computational fluid mechanics (CFM). In these fields of study, the computer has become the laboratory, offering us the ability to experiment on systems we cannot build in the physical world.

Over the past decade or so, we are perhaps seeing the emergence of a new class of scientist or scientific specialist, along with the experimentalist and theorist, the numericist. Just as an experimentalist needs to be aware of the science behind the inner working of their probes and detectors, the numericist must understand the limitations imposed by working with finite precision, or a discrete representation of the real number system. The computer is the device we used to probe our virtual world, and discrete mathematics imposes constraints on the precision of the probe. In moving from the world of smooth operations on the continuum to discrete representations of the same, we lose some basic kernels of truth we rely on as common sense. Namely, certain operations are abelian. More precisely there are certain procedures that when carried out by hand produce the same results regardless of the order in which the steps are performed but when executed on a computer could lead to different results for different implementations. The development of computational procedures requires attention to this fact, a new burden for the numericist, and understanding of the impact of this behavior on expected results.

This text focuses on the application of computational methods to the fields of linear acoustics. Acoustics is broadly defined as the propagation of mechanical vibration in a medium. Several aspects of this make acoustics an interesting field of study. First is the need for a medium to support the acoustic phenomenon, which unlike light propagates in free space at constant speed relative to all inertial observers. Another point of interest is that there are as many types of acoustic phenomena as there are media, from longitudinal pressure waves in a fluid to S and P waves in seismology. The material properties of the medium determine the number and type of acoustic waves that may be created and observed. We typically think of acoustics as a macro phenomenon, the result of bulk movement of the medium. However, as we probe nature at smaller scales, this type of phenomenon is precisely what is creating the acoustic phenomenon in solids and similarly particle collisions in fluids. The acoustic phenomenon is seen at small scales in lattice vibrations in crystals. Here the acoustic field is quantized and the quanta are referred to as phonons. This model is the result of an attempt to understand a phenomenon that exists at scales too large to be described by the fundamental process and too small to be a purely classical phenomenon.

The goal of this text is to introduce to the reader those numerical methods associated with the development of computational procedures for solving problems in acoustics and understanding linear acoustic propagation and scattering. The intended audience are students and professionals who are interested in the ingredients needed for the development of these procedures. The presentation of the material in this text is unique in the sense that it focuses on modeling paradigms first and introduce the numerical methods appropriate to that modeling paradigm rather than offer them in a preliminary chapter or appendix. Along the way, implementation issues that readers should be aware of are discussed. Examples are provided along with suggested exercises and references. The intent is to be pedagogical in the approach to presenting information so that readers who are new to the subject can begin experimenting. Classic methods and approaches are featured throughout the text while additional comments are included that highlight modern advances and novel modeling approaches that have appeared in the literature.
Since the intended audience consists of upper-level undergraduate students, graduate students, or professionals interested in this discipline, expected prerequisites to this material are:

- An introductory course that covers acoustics or fluid dynamics
- Familiarity with ordinary differential and partial differential equations, perhaps a course in mathematical methods for scientists and engineers
- Some exposure to programming in a high-level language such as Maple, Mathematica, and MATLAB or its open-source counterparts, SCILAB and Octave

The key feature of the presentation contained in this text is that it serves to bridge the gap between theory and implementation. The main focus is on techniques for solving the linear wave equation in homogeneous medium as well as inhomogeneous and anisotropic fluid medium for modeling wave propagation from a source and scattering from objects. Therefore, the starting point for much of this text will be the standard wave equation or the Helmholtz equation.

The transition from equations to computer procedures is not always a straightforward path. High-level programming languages come with easy-to-use interfaces for solving differential equations, matrix equations, and performing signal processing. Beyond these are professional software packages designed to allow users to build and run specific types of simulations using common modeling paradigms. Examples include ANSYS, FEMLAB, and FEKO, just to name a few. An understanding of the math, physics, and numerics is required to evaluate and interpret the results, but low-level programming is not necessary. Why learn these techniques? Specialized software can be very expensive, in fact cost prohibitive for students or those engaging in self-study. Many software companies offer personal or student versions of their software at a severely discounted price and with a restricted user license. If the reader is using this text for coursework in computational acoustics at a college or university, chances are student licenses for some professional software packages are made available through the campus bookstore. If not, it is easy to find this information online. Open-source versions of professional software exist and are worth trying. The downside to this is that bugs exist and due to certain constraints a fix may not be available in a hurry. Also, some open-source tools are not compatible with all operating systems. Readers who like programming and are amenable to the open-source philosophy can always contribute their fixes and upgrades (read the license). Pure curiosity drives most scientists and engineers to want to know what’s going on in any system, and this is a driver for developing homegrown algorithms even when libraries are available.

A brief description of each chapter is provided. Chapter 2 introduces topics related to numerics, computers, and algorithm development. These topics include binary representation of numbers, floating-point numbers, and $O(N)$ analysis, to name a few. Chapter 3 contains a survey of the linear wave equation and its connection to the supporting medium, from elastic bodies to fluids. In this chapter the linear wave equation for acoustics in a moving medium is introduced and discussed in detail. Chapter 4 introduces a variety of mathematical techniques and methods for solving the wave equation and describing the general behavior of the acoustic field. Chapter 5 discusses a variety of topics related to the analysis of acoustic waves: dispersion, refraction, attenuation, and Fourier analysis. After these chapters the structure of the text focuses on specific modeling techniques. In Chapter 6 normal modes are discussed. The wave equation is solved for a variety of 1-dimensional (1-dim) refractive profiles using exact methods, perturbation theory, and the numerical technique of relaxation. The chapter closes
with a brief description of coupled modes and their use in modeling acoustics in realistic environments. Chapter 7 provides an introduction to ray theory and ray tracing techniques. Exact solutions to 1-dim problems are discussed along with methods of developing ray trace procedures that account for 3-dim propagation without simplifying assumptions. Numerical techniques are also discussed and the Runge–Kutta method is introduced. In Chapter 8 the finite difference (FD) and finite difference time domain (FDTD) technique are discussed in theory and applied to the wave equation in the frequency and time domains. Following the FD method, Chapter 9 discusses the parabolic equation and its application to modeling sound in ducted environments. Chapter 10 provides an introduction to the finite element method (FEM), introducing numerical techniques required for building an FEM model of the acoustic field in the frequency domain. The last chapter, Chapter 11, is dedicated to the boundary element method (BEM). This chapter discusses the integral equation form of the Helmholtz equation and its discretization into a matrix equation. The exterior and interior problems are discussed, but attention is spent on developing models of the scattering cross section of hard bodies. This chapter introduces techniques for dealing with singular integrals.
This chapter introduces a collection of topics related to computation, model and simulation development, and code writing, starting with an introduction to floating-point numbers that introduces representations of numbers in bases other than 10 and floating-point representations of numbers. Following this is an introduction to estimating computational cost using \(O(N)\) analysis. The next section provides a discussion on simulation fidelity and complexity followed by a simple example of converting an equation to pseudo code. The last section provides a compiled list of open-source alternative to professional software and open-source numerical libraries for C/C++.

2.1 Floating-Point Numbers

2.1.1 Representations of Numbers

A number, \(x\), is represented by a power series in powers of a fixed number \(b\) called the base. The power series may be finite or infinite depending on the number:

\[
x = \sum_{n = -N_1}^{N_2} a_n b^n
\]

(2.1)

The coefficients in the expansion are given by \(\{a_n\}\), and \(N_1\) and \(N_2\) are the limits of the expansion. Coefficients obey the inequality \(0 \leq a_n < b\), and count how many of that power are present in the number. For irrational and rational numbers with infinitely repeating patterns, \(N_1 = \infty\). For all other rational numbers, \(N_1\) is finite. One typically denotes the number by
writing the coefficients in a sequence without the base explicitly present. A decimal notation is used to separate positive powers from negative powers:

\[ x = a_{N_2} a_{(N_2-1)} \cdots a_1 a_0, a_{-1} a_{-2} \cdots a_{(1-N_1)} a_{-N_1} \]  

(2.2)

Reading the digit sequence from left to right gives the number of each power of the base contained in the series expansion. We grow up learning base 10 and most readers have likely encountered base 2, or binary, representation of numbers. The following notation is used to keep tabs on which base is being used:

\[ x = \left( a_{N_2} a_{(N_2-1)} \cdots a_1 a_0, a_{-1} a_{-2} \cdots a_{(1-N_1)} a_{-N_1} \right)_b \]  

(2.3)

In some cases the parentheses are omitted. As an example, the number \((237.4631)_{10}\) is represented as a series expansion:

\[ 1 \times 10^{-4} + 3 \times 10^{-3} + 6 \times 10^{-2} + 4 \times 10^{-1} + 7 \times 10^{0} + 3 \times 10^{1} + 2 \times 10^{2} \]

Notice the reversed order of appearance of the coefficients. Now consider the base, \(b = 2\). In terms of a power series expansion, numbers are represented in terms of “ones place,” “twos place,” “fours place,” and so on. Coefficients in the expansion are bound by the inequality \(0 \leq a_n < 2\); hence the coefficients can only be 0 or 1. Table 2.1 provides a list of the integers from 0 to 10 in binary representation.

This example illustrates the value in using notation that references the base. The third row of the right column contains 10, which is not the integer 10 but the binary representation of 2, one in the twos place and zero in the ones place. Using the base notation in (2.3), \(7_{10} = 111_2\), both are representations of the number 7. Numbers between 0 and 1 are represented in terms of negative powers of base 2, that is, a halves place, a quarters place, and so on. A few examples are presented in Table 2.2.

**Table 2.1** Binary representation of integers

<table>
<thead>
<tr>
<th>Integer (base 10)</th>
<th>Expansion in base 2</th>
<th>Base 2 representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0 \times 2^0)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>(1 \times 2^0)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(0 \times 2^0 + 1 \times 2^1)</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>(1 \times 2^0 + 1 \times 2^1)</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>(0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2)</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>(1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2)</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>(0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2)</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>(1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2)</td>
<td>111</td>
</tr>
<tr>
<td>8</td>
<td>(0 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3)</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>(1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3)</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>(0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3)</td>
<td>1010</td>
</tr>
</tbody>
</table>
For the example above, fractions that can be expressed with three or fewer coefficients down to 1/8th are presented. Some of the numbers in the table contain the same number of significant figures in both bases. This is serendipitous.

As one final example, consider $5.625_{10} = 101.101_2$. This example illustrates the fact that a different number of significant figures is required to express a number in different bases. This is an important fact whose consequences cannot be overlooked, especially in the world of finite precision arithmetic [1]. One consequence is that some fractions may have a finite number of coefficients in one representation while producing an infinite repeating sequence in another representation. The maximum number of coefficients for representing a number that can be stored in memory is restricted, which means that error will necessarily exist when approximating such numbers. Recall how the fraction $1/3$ is dealt with in base 10, $0.3333\ldots$, or $0.\overline{3}$ to be exact. The bar notation indicates that the sequence repeats an infinite number of times. When using the number in a calculation, it would be truncated, keeping as many places as necessary to maintain the proper number of significant figures in the final answer, for example, $1/3 \approx 0.3333$. Now consider what would happen if only four significant figures were allowed for a number in any representation. This limitation imposes a new constraint called precision. The last example now reads $5.625_{10} \approx 101.1_2$. Starting with this number in base 10 representation, converting to base 2, truncating to four significant figures, and then converting back to base 10 gives $5.500_{10}$, or $5.625_{10} \approx 5.500_{10}$. This is not a horrible approximation, but can we do better? Not to this level of precision.

Three more bases commonly used in computer science are septal (base 7), octal (base 8), and hexadecimal (base 16). Coefficients in septal and octal can be represented by their integer values in base 10, 0–6, and 0–7, respectively. For hexadecimal, 16 characters are needed for each coefficient. The convention used is that each $a_n$ takes a value in the set \{0, 1, 2, \ldots, 9, A, B, \ldots, F\}. Table 2.3 lists the first 16 whole numbers in all representations introduced in this section.

### 2.1.2 Floating-Point Numbers

The IEEE Std 754-1985 defines a standard for representing various numbers as a sequence of bits called a bit string [2]. To represent arbitrary numbers, a form of base 2 scientific notation is used:

$$N = (-1)^s m \times 2^e$$  \hspace{1cm} (2.4)
Three quantities specify the number $N$. $S$ is the sign and can be either $(0, 1)$ for $+$ or $-$, respectively. The number $m$ is called the mantissa and is a binary fraction of the form $1.F$. The exponent is given by $E$. Normalized numbers have the exponent biased depending on the precision. When represented as a bit string, the binary representation of these numbers is placed in the order $(S, E, m)$. In binary notation, the mantissa digits are denoted $b_i$, and the exponent is denoted $a_j$, where the limits on $i, j$ are related to the type, that is, single precision or double precision. A visual representation of the bit string is given as follows:

$$\pm \ a_1 a_2 \cdots a_N \ b_1 b_2 \cdots b_M$$

In normalized format the leading bit of the mantissa is 1 and the exponent has a bias of $2^{N-1} - 1$. The width of the exponent and mantissa along with the bias is listed in for single-, double-, extended-, and quad-precision floating-point number (Table 2.4).

As an example, the single-precision floating-point representation of 3.7 is given as follows:

$$0 \ 10000000 \ 11011001100110011001100$$
The binary representation of 0.7 produces an infinite repeated sequence, 0.101100, which is truncated. There are a few special numbers in the IEEE standard. When the exponent reaches its maximum value and the mantissa is 0, this is defined as infinity in the floating-point system. Another floating-point number is NaN, or not a number. This arises in situations such as 0/0, \( \infty / \infty \), \( \infty - \infty \), \( -\infty + \infty \), and \( 0 \times \infty \). There are two types of NaNs, quiet and signaling. A signaling NaN raises an exception, whereas the quiet NaN will propagate through a routine without any problem, producing NaN for the output. Associated with floating-point arithmetic is the notion of machine epsilon. Given a base and precision, the width of the mantissa including the implicit bit and the machine epsilon is defined as

\[
\varepsilon = b^{-p-1}
\]  

Machine epsilon in binary and decimal representation is listed in Table 2.5 for the four floating-point types in Table 2.4.

### Table 2.5 Machine epsilon for floating-point types

<table>
<thead>
<tr>
<th>Type</th>
<th>Epsilon (binary)</th>
<th>Epsilon (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>(2^{-23})</td>
<td>(\sim 1.19 \times 10^{-7})</td>
</tr>
<tr>
<td>Double</td>
<td>(2^{-52})</td>
<td>(\sim 2.22 \times 10^{-16})</td>
</tr>
<tr>
<td>Extended</td>
<td>(2^{-63})</td>
<td>(\sim 1.08 \times 10^{-19})</td>
</tr>
<tr>
<td>Quad</td>
<td>(2^{-112})</td>
<td>(\sim 1.93 \times 10^{-112})</td>
</tr>
</tbody>
</table>

2.2 Computational Cost

There are two types of cost to consider in simulation development: processing time and memory. One is typically concerned with estimating how these quantities scale with the size of the simulation input, for example, number of degrees of freedom. For calculations involving a large number of operations or a set of operations acting on a large number of degrees of freedom, the “Big O” notion is useful for estimating the processing time and memory. If an algorithm operates on a large number of array elements, \(N\), Big O refers to the largest power of \(N\) obtained in counting the operations. It represents the limiting behavior of the algorithm. The purpose of the notation is to provide insight into how processes scale. Overall constants are dropped and the order of the process is quoted as \(O(N^\alpha)\), where \(\alpha\) is a real number, usually but not necessarily an integer. Leading order estimates do not have to be a power of \(N\). Examples include \(O(N \log N)\), \(O(2^N)\), and \(O(N!)\). To estimate the cost of an algorithm, values for the cost of various processes are needed. A rough estimate could assume that all calculations are equal in cost for a single execution, but this is not very accurate. The fastest operation on a computer is addition (and subtraction). On current processors, multiplication of floating-point numbers is about the same cost as addition. Division of two numbers requires more than one addition and is a more expensive operation. Denoting the cost of an addition by \(c_a\) for a floating-point operation, division can cost between 3 and 10 times \(c_a\), the square root function between 1 and 18 times, and transcendental functions, trigonometric,
and so on up to 15–50 times the cost of an addition. These ranges were estimated using information from Ref. [3], scaling by the cost of a floating-point addition. The range of values is due to differences in architecture location of data in memory and other factors. The take-away is that more complex operations have a larger individual unit of cost. As an example, consider the cost of evaluating the dot product between two vectors of size \( N \). The operation will require \( N \) multiplications and \( N - 1 \) additions. The total cost of these operations is estimated to be \( C_{\text{Dot}} = 2c_aN - c_a \). Taking the largest power of \( N \) and dropping constant, the dot product is an \( O(N) \) algorithm. Doubling the size of an array would double the processing time for this calculation. Matrix multiplication can be thought of as taking a dot product for every row of one matrix with every column of the other matrix. For two \( N \times N \) matrices, this is \( N^2 \) dot products so the process is \( O(N^3) \). Inverting a matrix by Gauss–Jordan elimination is also \( O(N^3) \). Consider the process of solving the linear system, \( A \cdot x = b \), for \( M \) distinct r.h.s. inputs, \( b \). Once the matrix is inverted, there is no need to invert it again to solve the system with different input vectors. Each multiplication of a matrix times a vector consists of \( N \) dot products, each row of \( A^{-1} \) with \( b \). Solving the system is an \( O(N^2) \). Given \( M \) r.h.s. vectors, the total cost is \( \sim N^3 + MN^2 \). Clearly the inverting is the most expensive part of the procedure. Even for a large number of inputs, say, \( M \sim N \), the entire process is still only \( O(N^3) \). Of course, it would be wrong to conclude that there is no extra cost to adding more instances of \( b \) on the pile. That would be comparing apples to oranges. If the inversion is a one-time task followed by a series of applications to an input, then the cost to solve for multiple inputs scales as \( MN^2 \). Lastly, if one were to run a routine for solving a linear system \( M \) times that only accepted a single \( b \) and did not save the inverse, the cost would be \( O(MN^3) \). This is clearly never a wise move.

The use of Big \( O \) provides an estimate of how algorithms scale, but it is equally important to consider the low-level unit cost of operations. The following example is a modified version of that found in Ref. [3]. Consider the cross term of an interference pattern. Two \( N \)-dimensional (\( N \)-dim) arrays of data, \( A \) and \( B \), are given, and the calculation being done is \( \cos(A_i - B_j) \) for all pairs \( i \neq j \). Since the operation is symmetric, there is no need to calculate this for each pair. The total number of calculations is \( N(N - 1)/2 \):

\[
\begin{align*}
\text{for } i &= 1 \text{ to } N \\
\quad &\text{for } j = i + 1 \text{ to } N \\
&Z_{i,j} = \cos (A_i - B_j)
\end{align*}
\]

The cost of this algorithm is \( c_T N(N - 1)/2 \), where \( c_T \) is the cost of a trigonometric function evaluation, the cost of subtraction being ignored relative to transcendental function evaluation. The same calculation can be done by evaluating the trigonometric functions in a single loop and evaluating the quantity \( Z_{ij} \) using the trig identity \( \cos(a - b) = \cos a \cos b + \sin a \sin b \):

\[
\begin{align*}
\text{for } i &= 1 \text{ to } N \\
&X_1 = \cos A_i \\
&X_2 = \cos B_i \\
&Y_1 = \sin A_i \\
&Y_2 = \sin B_i \\
\text{for } i &= 1 \text{ to } N \\
\quad &\text{for } j = i + 1 \text{ to } N \\
&Z_{i,j} = X_1 X_2 j + Y_1 Y_2 j
\end{align*}
\]
The cost for this process is $4cTN$ for the first loop, evaluation of the trigonometric functions, and $3c_a(N - 1)/2$ for the double loop, for a total of $4cTN + 3c_aN(N - 1)/2$. The leading orders of these algorithms are $O(cTN^2)$ for the first approach and $O(3c_aN^2)$ for the second. For large $N$ the ratio of these is $3c_a/cT$. Based on the values for evaluating transcendental functions provided earlier, the second approach can result in significant time savings.

Another cost estimate is memory, either the amount of memory that is required in RAM for a given function or the amount of memory that will be produced as output and written to a file. The same estimation process used for estimating process time carries over. The difference is in the unit cost, in this case bytes. To get an accurate estimate, one needs to know the cost of various data types. This boils down to knowing the number of bytes that a single type of data will occupy. Table 2.6 lists some common numeric data types, their size in bytes, and the symbol used to refer to them in this text. The size of a double, $\text{sizeof}$ ($\text{double}$) in C, is denoted by $d$.

Consider a memory estimate for an $N \times N$ matrix. Clearly there are $N^2$ numbers to store, and the total size will depend on the type of data in the matrix. The memory required for this is $\text{sizeof(type)}N^2$. A square matrix double-precision numbers with 10,000 rows and columns would occupy $8 \times (10^4)^2$ bytes = 0.8GB, almost one gigabyte of memory. If a single-precision result is sufficient for later use, this can be cut in half. It is helpful to estimate the memory that will be stored by temporary variables in a software program to ensure that it will not grow too large for the available RAM. Avoiding unnecessary copies of data or the use of additional variables and steps can streamline the performance of algorithms and simulation code. Similarly, it is always a good idea to get an estimate of the amount of output data that will be produced by a simulation and written to file to ensure there is enough storage.

### Table 2.6 Size of various numbers in bytes

<table>
<thead>
<tr>
<th>Number type</th>
<th>Size in bytes</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short integer</td>
<td>2</td>
<td>s</td>
</tr>
<tr>
<td>Integer</td>
<td>4</td>
<td>i</td>
</tr>
<tr>
<td>Long integer</td>
<td>8</td>
<td>l</td>
</tr>
<tr>
<td>Single</td>
<td>4</td>
<td>f</td>
</tr>
<tr>
<td>Double</td>
<td>8</td>
<td>d</td>
</tr>
<tr>
<td>Quad</td>
<td>16</td>
<td>q</td>
</tr>
</tbody>
</table>

The purpose of a simulation is to represent the behavior of something in the real world. Fidelity is related to the ability of a simulation to accurately represent the state or behavior of a system. When developing a simulation, it is important to decide ahead of time what is really important. Trying to model a complex system with unknown variables using detailed physics-based simulations may likely not succeed in reproducing data gathered in the field. Under ideal circumstances, carefully measured data under controlled conditions should match results from a well-designed model of the system. Common adjectives like “high,” “medium,” and “low” are frequently used in describing the fidelity of simulations. There is nothing wrong with these descriptions as long as they are meaningful to the person using them. That is to say, when
someone with expertise in a field describes a simulation as being high fidelity, most likely they are applying a series of technical comparisons to the results of the simulation in their mind, possibly recalling past experience using the simulation and a lifetime of knowledge regarding what to expect. Defining a fidelity requirement to drive the scope of a simulation development effort requires a detailed description of what the simulation is meant to model, specific descriptions of allowed errors in outputs, and conditions under which the simulation and its results will be used. Using a term like “high fidelity” or even “low fidelity” to set expectations for a simulation will lead to an endless spiral of rework to meet an undefined, and hence unrealizable, expectation. This topic has been the subject of efforts to develop a quantitative, technically meaningful definition of fidelity. The reader is encouraged to review Ref. [4]. Another factor related to fidelity decisions is complexity. Simulations that are very complex can be difficult to test, debug, and maintain. They can also become unstable, being well behaved and producing reliable results only in a limited range of cases.

For the purposes of this text, the system under study is an equation, the wave equation. True its value only resides in the fact that it can describe physical phenomena like acoustics, optics, and quantum mechanics, but it is this equation and the behavior of its solutions under mathematically ideal circumstances that is of interest. A particular numerical implementation of the wave equation can be considered to have “good” fidelity if it reproduces the expected results to within some tolerance when compared with known exact solutions. This is really a validation criterion. Ensuring a good match between two models does not ensure matching with nature, and it does not mean that the implementation will continue to perform as well when it is used outside the set of validation test cases. Validating a procedure against other known results is used to build confidence in the new implementation and, if they fail to match, uncover unintentional errors. Again, the focus of the text is on methods for modeling the wave equation. How the outputs of this equation are used to model music on a windy day, sonar pings in a noisy environment, and so on is a different matter. The person developing models for these purposes has to take into account the unknowns and their potential impact on fidelity. Through this process reasonable compromises can be made, leading to a reduction in complexity while producing useful results.

2.4 Code Development

This section deals with the process of converting a mathematical expression into some type of computer program. This is done by example to illustrate the kind of issues that can arise. Given some proficiency in at least one language, even a high-level language, the real task is deciding how to approach the problem of turning equations into code. This assumes that a modeling paradigm has been chosen.

Sometimes a set of equations written on paper already looks like pseudo code but this is typically not the case. One equation does not easily translate to one line of code. Points that are often overlooked are; (1) that there are preconditions on data, and (2) constraints required for the answers to make sense. Sometimes simple math equations come with lists of caveats, and these caveats are not usually considered as part of the “equation,” but in fact they are and need to be implemented for things to work. Whether or not a particular condition or constraint requires additional processing steps or data types depends on the language used.
A simple example is provided to illustrate this point. The task is to write a function that calculates the two roots of a quadratic expression given the coefficients in standard form:

\[
 r_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]  

There are a few things hidden in this equation. The first is that it is two equations, one for each of two roots. This isn’t hard to handle but may look different in MATLAB than in C [5]. Sample code for both is provided as follows:

```matlab
function [roots] = quadraticRoots(a, b, c)
    inv_2a = 1.0/(2*a);
    R = sqrt(b^2 - 4*a*c);
    Roots = inv_2a*[−b + R; −b − R];

end example
```

Next here is the same example in C:

```c
double quadraticRoots(double a, double b, double c, double *r){

double R, inv_2a;

    inv_2a = 1.0/(2.0*a);
    R = sqrt(b*b - 4*a*c);
    r[0] = inv_2a*(R-b);
    r[1] = -inv_2a*(R+b);
}
```

An obvious difference is the occurrence of “double.” This is a key word in C and many other languages to indicate what type of number is being used to approximate the variable and how much memory to allocate. In C the type of data each variable represents needs to be specified. In MATLAB and other high-level languages, all variables are assumed to be double unless specified otherwise. In defining a function in C, the output, a pointer to an array, is specified as an input, indicated by the “double *r”. Finally notice that the index
for $r$ starts at 0, whereas in MATLAB it would start at 1. The next thing to notice in the quadratic formula is the possibility of a negative radicand and hence a complex result for the root. A human can understand all the possible cases and what they mean geometrically, but a computer can’t. It needs to be told how to give the user what they want, and the user needs to interpret the results based on an understanding of the original problem. In MATLAB, this is not an issue since MATLAB is capable of handling complex numbers without special instructions. One could say that all numbers in MATLAB are double, complex, 1 by 1 matrices. The number 1 would be $1.000\ldots + 1i\ 0.000\ldots$, $1i$ is the syntax for $\sqrt{-1}$. In C/ C++ one would need to either make each root a 2-dim array containing the real and imaginary parts or declare a complex double data type that requires an additional library. This is the least of our trouble since C will not understand what it means to calculate $\sqrt{-1}$ under normal circumstances. To continue this example, assume the use of a programming language that does not understand complex numbers but avoid C/C++ specific syntax [6]. The goal is to write a function that is capable of producing both roots and expressing all the information required to interpret them. Start by writing down exactly what will be done and what is needed in a detailed set of instructions. In this example as many as four output variables are needed; self-explanatory names are provided as follows:

realRoot1, imagRoot1, realRoot2, imagRoot2

Given the coefficients $a$, $b$, and $c$, calculate the radicand $b^2 - 4ac$ and call it rdcnd. We now have a variable $rdcnd = b^2 - 4ac$.

Check the sign on the radicand, $b^2 - 4ac$, since that determines the type of roots. There are three cases:

Case 1: $rdcnd = 0$, there is one real root not two.
Case 2: $rdcnd > 0$, there are two real roots.
Case 3: $rdcnd < 0$, there are two complex roots.

The third case is the cause of possible issues. Here is a strategy for dealing with case 3. If $rdcnd < 0$, then change the sign, calculate the radical term in the quadratic formula, and store the result as the imaginary part of the output.

Calculate the two parts of the root and give them names:

\[
\begin{align*}
  x &= -b / (2a), \\
  y &= \sqrt{\text{abs}(rdcnd)}/(2a)
\end{align*}
\]

Now store the results depending on which of the three cases is realized.

If case 1 is true, then let

realRoot1 = $x$, imagRoot1 = 0, realRoot2 = $x$, imagRoot2 = 0

If case 2 is true, then let

realRoot1 = $x + y$, imagRoot1 = 0, realRoot2 = $x - y$, imagRoot2 = 0
Finally, if case 3 is true, then let

\[
\text{realRoot1} = x, \text{imagRoot1} = y, \text{realRoot2} = x, \text{imagRoot2} = -y
\]

The previous dialog goes through all the logic behind the three cases. A cleaner version of the pseudo code is provided as follows:

**Input variables**

\(a, b, c\)

**Output variables**

\(\text{realRoot1, imagRoot1, realRoot2, imagRoot2}\)

**Processing instructions**

\[
\text{rdcnd} = b^2 - 4ac \\
x = -b/(2a) \\
y = \sqrt{\text{abs(rdcnd)}}/(2a)
\]

**If** \(\text{rdcnd} = 0\)

\[
\text{realRoot1} = x \\
\text{imagRoot1} = 0 \\
\text{realRoot2} = x \\
\text{imagRoot2} = 0
\]

**If** \(\text{rdcnd} > 0\)

\[
\text{realRoot1} = x + y \\
\text{imagRoot1} = 0 \\
\text{realRoot2} = x - y \\
\text{imagRoot2} = 0
\]

**If** \(\text{rdcnd} < 0\)

\[
\text{realRoot1} = x \\
\text{imagRoot1} = y \\
\text{realRoot2} = x \\
\text{imagRoot2} = -y
\]

Writing things out this way makes the logic flow and the steps clear. And it looks like code in many languages. The virtue of writing pseudo code is that it serves as an organizing tool revealing redundant steps and potential optimizations. It serves as a shorthand notation for a script, and that is exactly what code is, a script. This example was meant to illustrate the steps involved in writing a simple procedure. Since all languages are meant to allow scripts to be turned into binary instructions, it makes sense to start organizing things in terms of this approach. Identify what the inputs and outputs are and if possible what data type is needed. Then write instructions as if you were teaching someone how to perform the calculation. Once the instructions are complete, start organizing those instructions in shorthand. At this point the rest depends on the choice of programming language.
2.5 List of Open-Source Tools

This section offers a small collection of open-source tools that falls into one of three categories: libraries, compilers, and other tools for use with C/C++, open-source equivalents of MATLAB, Maple or Mathematica, and miscellaneous software that is relevant to the subject matter of this text (Tables 2.7, 2.8, and 2.9).

Disclaimer(s)

The author is not endorsing any one package over another or the use of open source as an alternative to professional software packages. These lists are provided as a courtesy for readers interested in using and contributing to open source.

It is the responsibility of the reader to read and understand the license agreement associated with these software libraries and alternatives to professional software. What you can or cannot do is expressed in the license. Read the license.

Table 2.7 Open-source alternatives to MATLAB, MAPLE, or Mathematica

<table>
<thead>
<tr>
<th>Name</th>
<th>Where to find</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCILAB</td>
<td><a href="http://www.scilab.org/">http://www.scilab.org/</a></td>
<td>Open-source version of MATLAB</td>
</tr>
<tr>
<td>Octave</td>
<td><a href="https://www.gnu.org/software/octave/">https://www.gnu.org/software/octave/</a></td>
<td>Open-source version of MATLAB</td>
</tr>
<tr>
<td>Maxima</td>
<td><a href="http://maxima.sourceforge.net/">http://maxima.sourceforge.net/</a></td>
<td>Open-source symbolic math software, similar to Maple or Mathematica</td>
</tr>
<tr>
<td>Euler Math</td>
<td><a href="http://euler.rene-grothmann.de/">http://euler.rene-grothmann.de/</a></td>
<td>Open-source version of MATLAB with symbolic math</td>
</tr>
<tr>
<td>Toolbox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FreeMat</td>
<td><a href="http://freemat.sourceforge.net/">http://freemat.sourceforge.net/</a></td>
<td>Open-source version of MATLAB</td>
</tr>
<tr>
<td>SageMath</td>
<td><a href="http://www.sagemath.org/">http://www.sagemath.org/</a></td>
<td>Open-source alternative to Maple, Mathematica, and MATLAB</td>
</tr>
</tbody>
</table>

Table 2.8 List of C/C++ numeric libraries and other items

<table>
<thead>
<tr>
<th>Name</th>
<th>Where to find</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU Science</td>
<td><a href="https://www.gnu.org/software/gsl/">https://www.gnu.org/software/gsl/</a></td>
<td>Numerical library for C/C++</td>
</tr>
<tr>
<td>Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boost</td>
<td><a href="http://www.boost.org/">http://www.boost.org/</a></td>
<td>Numerical library for C/C++</td>
</tr>
<tr>
<td>Eigen</td>
<td><a href="http://eigen.tuxfamily.org/">http://eigen.tuxfamily.org/</a></td>
<td>Linear algebra library for C/C++</td>
</tr>
<tr>
<td>Armadillo</td>
<td><a href="http://arma.sourceforge.net/">http://arma.sourceforge.net/</a></td>
<td>Open-source linear algebra library for C++</td>
</tr>
<tr>
<td>Visual Studio</td>
<td><a href="https://www.visualstudio.com/vs/">https://www.visualstudio.com/vs/</a></td>
<td>Free C/C++ compiler and IDE</td>
</tr>
<tr>
<td>Express</td>
<td>visual-studio-express/</td>
<td></td>
</tr>
<tr>
<td>GCC</td>
<td><a href="https://gcc.gnu.org/">https://gcc.gnu.org/</a></td>
<td>GNU open-source C/C++ compiler</td>
</tr>
<tr>
<td>FFTW</td>
<td><a href="http://www.fftw.org/">http://www.fftw.org/</a></td>
<td>Free discrete FFT library for C</td>
</tr>
<tr>
<td>Odeint</td>
<td><a href="http://headmyshoulder.github.io/odeint-v2/">http://headmyshoulder.github.io/odeint-v2/</a></td>
<td>Free C++ library of numerical ordinary differential equation solvers</td>
</tr>
</tbody>
</table>
2.6 Exercises

1. For the cos(a − b) evaluation, find the value of N when the two approaches are equal as a function of $r = c_T/c_a$.
2. Write the binary representation of 123.45.
3. For the result of Exercise 2, write the single-precision floating-point representation of this number.
4. Write pseudo code for evaluating Green’s function:

\[
\frac{\exp(ikx)}{x}
\]

where, $x, k$ are real number and $x \geq 0$, $k$ can be either sign. Assume that only real number is possible in the evaluation of the function. You will need to decide what the behavior should be if $x = 0$ is an input.
5. The function in Exercise 4 is used to estimate the interaction among $N$ point line objects.
   (a) How many times will the equation need to be used to calculate all interactions?
   (b) Estimate the computational cost of this calculation including the unit cost of the operations expressed in the equation.
   (c) Estimate the memory required to store all results
   (d) What is the memory requirement for 100,000 objects assuming double precision?

References