Third Edition

NUMERICAL ANALYSIS

Pearson

Timothy Sauer

Numerical Analysis

THIRD EDITION

Timothy Sauer
George Mason University

Pearson

Director, Portfolio Management: Deirdre Lynch
Executive Editor: Jeff Weidenaar
Editorial Assistant: Jennifer Snyder
Content Producer: Tara Corpuz
Managing Producer: Scott Disanno
Producer: Jean Choe
Product Marketing Manager: Yvonne Vannatta
Field Marketing Manager: Evan St. Cyr
Marketing Assistant: Jon Bryant
Senior Author Support/Technology Specialist: Joe Vetere
Manager, Rights and Permissions: Gina Cheselka
Manufacturing Buyer: Carol Melville, LSC Communications
Cover Image: Gyn9037/ Shutterstock
Text and Cover Design, Illustrations, Production Coordination, Composition:
Integra Software Services Pvt. Ltd

Photo Credits: Page 1 Zsolt Biczo/ Shutterstock; Page 26 Polonio Video/ Shutterstock; Page 41 DEA PICTURE LIBRARY / Getty Images; Page 74 Redswept /Shutterstock; Page 144 Rosenfeld Images, Ltd./Photo Researchers, Inc.; Page 196 dolgachov/ 123RF; Page 253 wklzzz / 123RF; Page 293 UPPA/Photoshot; Page 366 Paul Springett 04/Alamy Stock Photo; Page 394 iStock/Getty Images Plus; Page 453 xPACIFICA / Alamy; Page 489 Picture Alliance/Photoshot; Page 518 Chris Rout/Alamy Stock Photo; Pages 528 & 534 Toni Angermayer/Photo Researchers, Inc.; Page 556 Jinx Photography Brands/Alamy Stock Photo; Page 593 Astronoman /Shutterstock.

Text Credits: Page 50 J. H. Wilkinson, The perfidious polynomial, In ed. by Gene H. Golub. Studies in Numerical Analysis. Mathematical Association of America, 24 (1984); Page 153 & Page 188 “Author-created using the software from MATLAB. The MathWorks, Inc., Natick, Massachusetts, USA, http://www.mathworks.com.”; Page 454 Von Neumann, John (1951). “Various techniques used in connection with random digits.” In A. S. Householder, G. E. Forsythe, and H. H. Germond, eds., Proceedings of Symposium on “Monte Carlo Method” held June-July 1949 in Los Angeles. Journal of Research of the National Bureau of Standards, Applied Mathematics Series, no. 12, pp 36–38 (Washington, D.C.: USGPO, 1951) Summary written by George E. Forsythe. Reprinted in von Neumann, John von Neumann Collected Works, ed. A. H. Taub, vol. 5 (New York: Macmillan, 1963) Vol. V, pp 768–770; Page 622 Author-created using the software from MATLAB. The MathWorks, Inc., Natick, Massachusetts, USA, http://www.mathworks.com.; Page 623 Author-created using the software from MATLAB. The MathWorks, Inc., Natick, Massachusetts, USA, http://www.mathworks.com.

Library of Congress Cataloging-in-Publication Data

Names: Sauer, Tim, author.
Title: Numerical analysis / Timothy Sauer, George Mason University.
Description: Third edition. | Hoboken : Pearson, [2019] | Includes
bibliographical references and index.
Identifiers: LCCN 2017028491| ISBN 9780134696454 (alk. paper) |
ISBN 013469645X (alk. paper)
Subjects: LCSH: Numerical analysis. | Mathematical analysis.
Classification: LCC QA297 .S348 2019 | DDC 518–dc23
LC record available at https://lccn.loc.gov/2017028491

1 17

Pearson

ISBN 10: 0-13-469645-X
ISBN 13: 978-0-13-469645-4

Contents

This page intentionally left blank

Preface

Numerical Analysis is a text for students of engineering, science, mathematics, and computer science who have completed elementary calculus and matrix algebra. The primary goal is to construct and explore algorithms for solving science and engineering problems. The not-so-secret secondary mission is to help the reader locate these algorithms in a landscape of some potent and far-reaching principles. These unifying principles, taken together, constitute a dynamic field of current research and development in modern numerical and computational science.

The discipline of numerical analysis is jam-packed with useful ideas. Textbooks run the risk of presenting the subject as a bag of neat but unrelated tricks. For a deep understanding, readers need to learn much more than how to code Newton's Method, Runge–Kutta, and the Fast Fourier Transform. They must absorb the big principles, the ones that permeate numerical analysis and integrate its competing concerns of accuracy and efficiency.

The notions of *convergence, complexity, conditioning, compression*, and *orthogonality* are among the most important of the big ideas. Any approximation method worth its salt must converge to the correct answer as more computational resources are devoted to it, and the complexity of a method is a measure of its use of these resources. The conditioning of a problem, or susceptibility to error magnification, is fundamental to knowing how it can be attacked. Many of the newest applications of numerical analysis strive to realize data in a shorter or compressed way. Finally, orthogonality is crucial for efficiency in many algorithms, and is irreplaceable where conditioning is an issue or compression is a goal.

In this book, the roles of these five concepts in modern numerical analysis are emphasized in short thematic elements labeled ***Spotlight***. They comment on the topic at hand and make informal connections to other expressions of the same concept elsewhere in the book. We hope that highlighting the five concepts in such an explicit way functions as a Greek chorus, accentuating what is really crucial about the theory on the page.

Although it is common knowledge that the ideas of numerical analysis are vital to the practice of modern science and engineering, it never hurts to be obvious. The feature entitled ***Reality Check*** provide concrete examples of the way numerical methods lead to solutions of important scientific and technological problems. These extended applications were chosen to be timely and close to everyday experience. Although it is impossible (and probably undesirable) to present the full details of the problems, the Reality Checks attempt to go deeply enough to show how a technique or algorithm can leverage a small amount of mathematics into a great payoff in technological design and function. The Reality Checks were popular as a source of student projects in previous editions, and they have been extended and amplified in this edition.

NEW TO THIS EDITION

Features of the third edition include:

- Short URLs in the side margin of the text (235 of them in all) take students directly to relevant content that supports their use of the textbook. Specifically:
  - **MATLAB Code**: Longer instances of MATLAB code are available for students in *.m format. The homepage for all of the instances of MATLAB code is goo.gl/VxzXyw.

  - **Solutions to Selected Exercises**: This text used to be supported by a Student Solutions Manual that was available for purchase separately. In this edition we are providing students with access solutions to selected exercises online *at no extra charge*. The homepage for the selected solutions is `goo.gl/2j5gI7`.
  - **Additional Examples**: Each section of the third edition is enhanced with extra new examples, designed to reinforce the text exposition and to ease the reader's transition to active solution of exercises and computer problems. The full worked-out details of these examples, more than one hundred in total, are available online. Some of the solutions are in video format (created by the author). The homepage for the solutions to Additional Examples is `goo.gl/lFQb0B`.
  - NOTE: The homepage for *all* web content supporting the text is `goo.gl/zQNJeP`.
- More detailed discussion of several key concepts has been added in this edition, including theory of polynomial interpolation, multi-step differential equation solvers, boundary value problems, and the singular value decomposition, among others.
- The Reality Check on audio compression in Chapter 11 has been refurbished and simplified, and other MATLAB codes have been added and updated throughout the text.
- Several dozen new exercises and computer problems have been added to the third edition.

TECHNOLOGY

The software package MATLAB is used both for exposition of algorithms and as a suggested platform for student assignments and projects. The amount of MATLAB code provided in the text is carefully modulated, due to the fact that too much tends to be counterproductive. More MATLAB code is found in the early chapters, allowing the reader to gain proficiency in a gradual manner. Where more elaborate code is provided (in the study of interpolation, and ordinary and partial differential equations, for example), the expectation is for the reader to use what is given as a jumping-off point to exploit and extend.

It is not essential that any particular computational platform be used with this textbook, but the growing presence of MATLAB in engineering and science departments shows that a common language can smooth over many potholes. With MATLAB, all of the interface problems—data input/output, plotting, and so on—are solved in one fell swoop. Data structure issues (for example those that arise when studying sparse matrix methods) are standardized by relying on appropriate commands. MATLAB has facilities for audio and image file input and output. Differential equations simulations are simple to realize due to the animation commands built into MATLAB. These goals can all be achieved in other ways. But it is helpful to have one package that will run on almost all operating systems and simplify the details so that students can focus on the real mathematical issues. Appendix B is a MATLAB tutorial that can be used as a first introduction to students, or as a reference for those already familiar.

SUPPLEMENTS

The Instructor's Solutions Manual contains detailed solutions to the odd-numbered exercises, and answers to the even-numbered exercises. The manual also shows how to use MATLAB software as an aid to solving the types of problems that are presented in the Exercises and Computer Problems.

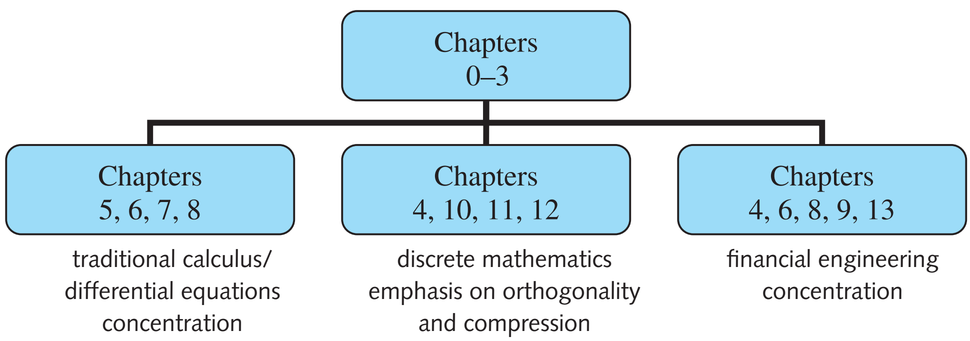
DESIGNING THE COURSE

Numerical Analysis is structured to move from foundational, elementary ideas at the outset to more sophisticated concepts later in the presentation. Chapter 0 provides fundamental building blocks for later use. Some instructors like to start at the beginning; others (including the author) prefer to start at Chapter 1 and fold in topics from Chapter 0 when required. Chapters 1 and 2 cover equation-solving in its various forms. Chapters 3 and 4 primarily treat the fitting of data, interpolation and least squares methods. In chapters 5–8, we return to the classical numerical analysis areas of continuous mathematics: numerical differentiation and integration, and the solution of ordinary and partial differential equations with initial and boundary conditions.

Chapter 9 develops random numbers in order to provide complementary methods to Chapters 5–8: the Monte-Carlo alternative to the standard numerical integration schemes and the counterpoint of stochastic differential equations are necessary when uncertainty is present in the model.

Compression is a core topic of numerical analysis, even though it often hides in plain sight in interpolation, least squares, and Fourier analysis. Modern compression techniques are featured in Chapters 10 and 11. In the former, the Fast Fourier Transform is treated as a device to carry out trigonometric interpolation, both in the exact and least squares sense. Links to audio compression are emphasized, and fully carried out in Chapter 11 on the Discrete Cosine Transform, the standard workhorse for modern audio and image compression. Chapter 12 on eigenvalues and singular values is also written to emphasize its connections to data compression, which are growing in importance in contemporary applications. Chapter 13 provides a short introduction to optimization techniques.

Numerical Analysis can also be used for a one-semester course with judicious choice of topics. Chapters 0–3 are fundamental for any course in the area. Separate one-semester tracks can be designed as follows:



ACKNOWLEDGMENTS

The third edition owes a debt to many people, including the students of many classes who have read and commented on earlier versions. In addition, Paul Lorczak was essential in helping me avoid embarrassing blunders. The resourceful staff at Pearson, including Jeff Weidenaar, Jenn Snyder, Yvonne Vannatta, and Tara Corpuz, made the production of the third edition almost enjoyable. Finally, thanks are due to the helpful readers from other universities for their encouragement of this project and indispensable advice for improvement of earlier versions:

- Eugene Allgower, Colorado State University
- Constantin Bacuta, University of Delaware
- Michele Benzi, Emory University
- Jerry Bona, University of Illinois at Chicago
- George Davis, Georgia State University
- Chris Danforth, University of Vermont
- Alberto Delgado, Illinois State University
- Robert Dillon. Washington State University
- Qiang Du, Columbia University
- Ahmet Duran, University of Michigan
- Gregory Goeckel, Presbyterian College
- Herman Gollwitzer, Drexel University
- Weimin Han, University of Iowa *
- Don Hardcastle, Baylor University
- David R. Hill, Temple University
- Alberto Jimenez, California Polytechnic State University *
- Hideaki Kaneko, Old Dominion University
- Ashwani Kapila, Rensselaer Polytechnic Institute *
- Daniel Kaplan, Macalester College
- Fritz Keinert, Iowa State University
- Akhtar A. Khan, Rochester Institute of Technology
- Lucia M. Kimball, Bentley College
- Colleen M. Kirk, California Polytechnic State University
- Seppo Korpela, Ohio State University
- William Layton, University of Pittsburgh
- Brenton LeMesurier, College of Charleston
- Melvin Leok, University of California, San Diego
- Doron Levy, University of Maryland
- Bo Li, University of California, San Diego *
- Jianguo Liu, University of North Texas *
- Mark Lyon, University of New Hampshire *
- Shankar Mahalingam, University of Alabama, Huntsville
- Amnon Meir, Southern Methodist University
- Peter Monk, University of Delaware
- Joseph E. Pasciak, Texas A&M University
- Jeff Parker, Harvard University
- Jacek Polewczak, California State University
- Jorge Rebaza, Missouri State University
- Jeffrey Scroggs, North Carolina State University
- David Stewart, University of Iowa *
- David Stowell, Brigham Young University *
- Sergei Suslov, Arizona State University
- Daniel Szyld, Temple University
- Ahlam Tannouri, Morgan State University

- Janos Turi, University of Texas, Dallas *
- Jin Wang, Old Dominion University
- Bruno Welfert, Arizona State University
- Nathaniel Whitaker, University of Massachusetts

* Contributed to the current edition

This page intentionally left blank

CHAPTER

0 Fundamentals

This introductory chapter provides basic building blocks necessary for the construction and understanding of the algorithms of the book. They include fundamental ideas of introductory calculus and function evaluation, the details of machine arithmetic as it is carried out on modern computers, and discussion of the loss of significant digits resulting from poorly designed calculations.

After discussing efficient methods for evaluating polynomials, we study the binary number system, the representation of floating point numbers, and the common protocols used for rounding. The effects of the small rounding errors on computations are magnified in ill-conditioned problems. The battle to limit these pernicious effects is a recurring theme throughout the rest of the chapters.

The goal of this book is to present and discuss methods of solving mathematical problems with computers. The most fundamental operations of arithmetic are addition and multiplication. These are also the operations needed to evaluate a polynomial $P(x)$ at a particular value x. It is no coincidence that polynomials are the basic building blocks for many computational techniques we will construct.

Because of this, it is important to know how to evaluate a polynomial. The reader probably already knows how and may consider spending time on such an easy problem slightly ridiculous! But the more basic an operation is, the more we stand to gain by doing it right. Therefore we will think about how to implement polynomial evaluation as efficiently as possible.

0.1 EVALUATING A POLYNOMIAL

What is the best way to evaluate

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1,$$

say, at $x = 1/2$? Assume that the coefficients of the polynomial and the number $1/2$ are stored in memory, and try to minimize the number of additions and multiplications required to get $P(1/2)$. To simplify matters, we will not count time spent storing and fetching numbers to and from memory.

METHOD 1 The first and most straightforward approach is

$$P\left(\frac{1}{2}\right) = 2*\frac{1}{2}*\frac{1}{2}*\frac{1}{2}*\frac{1}{2} + 3*\frac{1}{2}*\frac{1}{2}*\frac{1}{2} - 3*\frac{1}{2}*\frac{1}{2} + 5*\frac{1}{2} - 1 = \frac{5}{4}. \quad (0.1)$$

The number of multiplications required is 10, together with 4 additions. Two of the additions are actually subtractions, but because subtraction can be viewed as adding a negative stored number, we will not worry about the difference.

There surely is a better way than (0.1). Effort is being duplicated—operations can be saved by eliminating the repeated multiplication by the input 1/2. A better strategy is to first compute $(1/2)^4$, storing partial products as we go. That leads to the following method:

METHOD 2 Find the powers of the input number $x = 1/2$ first, and store them for future use:

$$\begin{aligned}
\frac{1}{2}*\frac{1}{2} &= \left(\frac{1}{2}\right)^2 \\
\left(\frac{1}{2}\right)^2*\frac{1}{2} &= \left(\frac{1}{2}\right)^3 \\
\left(\frac{1}{2}\right)^3*\frac{1}{2} &= \left(\frac{1}{2}\right)^4.
\end{aligned}$$

Now we can add up the terms:

$$P\left(\frac{1}{2}\right) = 2*\left(\frac{1}{2}\right)^4 + 3*\left(\frac{1}{2}\right)^3 - 3*\left(\frac{1}{2}\right)^2 + 5*\frac{1}{2} - 1 = \frac{5}{4}.$$

There are now 3 multiplications of 1/2, along with 4 other multiplications. Counting up, we have reduced to 7 multiplications, with the same 4 additions. Is the reduction from 14 to 11 operations a significant improvement? If there is only one evaluation to be done, then probably not. Whether Method 1 or Method 2 is used, the answer will be available before you can lift your fingers from the computer keyboard. However, suppose the polynomial needs to be evaluated at different inputs x several times per second. Then the difference may be crucial to getting the information when it is needed.

Is this the best we can do for a degree 4 polynomial? It may be hard to imagine that we can eliminate three more operations, but we can. The best elementary method is the following one:

METHOD 3 (Nested Multiplication) Rewrite the polynomial so that it can be evaluated from the inside out:

$$\begin{aligned}
P(x) &= -1 + x(5 - 3x + 3x^2 + 2x^3) \\
&= -1 + x(5 + x(-3 + 3x + 2x^2)) \\
&= -1 + x(5 + x(-3 + x(3 + 2x))) \\
&= -1 + x*(5 + x*(-3 + x*(3 + x*2))). \quad (0.2)
\end{aligned}$$

Here the polynomial is written backwards, and powers of x are factored out of the rest of the polynomial. Once you can see to write it this way—no computation is required to do the rewriting—the coefficients are unchanged. Now evaluate from the inside out:

$$
\begin{aligned}
&\text{multiply } \tfrac{1}{2} * 2, \quad &&\text{add } +3 \rightarrow 4 \\
&\text{multiply } \tfrac{1}{2} * 4, \quad &&\text{add } -3 \rightarrow -1 \\
&\text{multiply } \tfrac{1}{2} * -1, \quad &&\text{add } +5 \rightarrow \tfrac{9}{2} \\
&\text{multiply } \tfrac{1}{2} * \tfrac{9}{2}, \quad &&\text{add } -1 \rightarrow \tfrac{5}{4}.
\end{aligned}
\tag{0.3}
$$

This method, called **nested multiplication** or **Horner's method**, evaluates the polynomial in 4 multiplications and 4 additions. A general degree d polynomial can be evaluated in d multiplications and d additions. Nested multiplication is closely related to synthetic division of polynomial arithmetic.

The example of polynomial evaluation is characteristic of the entire topic of computational methods for scientific computing. First, computers are very fast at doing very simple things. Second, it is important to do even simple tasks as efficiently as possible, since they may be executed many times. Third, the best way may not be the obvious way. Over the last half-century, the fields of numerical analysis and scientific computing, hand in hand with computer hardware technology, have developed efficient solution techniques to attack common problems.

While the standard form for a polynomial $c_1 + c_2x + c_3x^2 + c_4x^3 + c_5x^4$ can be written in nested form as

$$c_1 + x(c_2 + x(c_3 + x(c_4 + x(c_5)))), \tag{0.4}$$

some applications require a more general form. In particular, interpolation calculations in Chapter 3 will require the form

$$c_1 + (x - r_1)(c_2 + (x - r_2)(c_3 + (x - r_3)(c_4 + (x - r_4)(c_5)))), \tag{0.5}$$

where we call r_1, r_2, r_3, and r_4 the **base points**. Note that setting $r_1 = r_2 = r_3 = r_4 = 0$ in (0.5) recovers the original nested form (0.4).

The following MATLAB code implements the general form of nested multiplication (compare with (0.3)):

MATLAB code shown here can be found at goo.gl/XjtZ1F

```
%Program 0.1 Nested multiplication
%Evaluates polynomial from nested form using Horner's Method
%Input: degree d of polynomial,
%       array of d+1 coefficients c (constant term first),
%       x-coordinate x at which to evaluate, and
%       array of d base points b, if needed
%Output: value y of polynomial at x
function y=nest(d,c,x,b)
if nargin<4, b=zeros(d,1); end
y=c(d+1);
for i=d:-1:1
  y = y.*(x-b(i))+c(i);
end
```

Running this MATLAB function is a matter of substituting the input data, which consist of the degree, coefficients, evaluation points, and base points. For example, polynomial (0.2) can be evaluated at $x = 1/2$ by the MATLAB command

```
>> nest(4,[-1 5 -3 3 2],1/2,[0 0 0 0])

ans =

    1.2500
```

as we found earlier by hand. The file `nest.m`, as the rest of the MATLAB code shown in this book, must be accessible from the MATLAB path (or in the current directory) when executing the command.

If the `nest` command is to be used with all base points 0 as in (0.2), the abbreviated form

```
>> nest(4,[-1 5 -3 3 2],1/2)
```

may be used with the same result. This is due to the `nargin` statement in `nest.m`. If the number of input arguments is less than 4, the base points are automatically set to zero.

Because of MATLAB's seamless treatment of vector notation, the `nest` command can evaluate an array of x values at once. The following code is illustrative:

```
>> nest(4,[-1 5 -3 3 2],[-2 -1 0 1 2])

ans =

   -15   -10    -1     6    53
```

Finally, the degree 3 interpolating polynomial

$$P(x) = 1 + x\left(\frac{1}{2} + (x-2)\left(\frac{1}{2} + (x-3)\left(-\frac{1}{2}\right)\right)\right)$$

from Chapter 3 has base points $r_1 = 0, r_2 = 2, r_3 = 3$. It can be evaluated at $x = 1$ by

```
>> nest(3,[1 1/2 1/2 -1/2],1,[0 2 3])

ans =

     0
```

▶ EXAMPLE 0.1 Find an efficient method for evaluating the polynomial $P(x) = 4x^5 + 7x^8 - 3x^{11} + 2x^{14}$.

Some rewriting of the polynomial may help reduce the computational effort required for evaluation. The idea is to factor x^5 from each term and write as a polynomial in the quantity x^3:

$$\begin{aligned} P(x) &= x^5(4 + 7x^3 - 3x^6 + 2x^9) \\ &= x^5 * (4 + x^3 * (7 + x^3 * (-3 + x^3 * (2)))). \end{aligned}$$

For each input x, we need to calculate $x * x = x^2$, $x * x^2 = x^3$, and $x^2 * x^3 = x^5$ first. These three multiplications, combined with the multiplication of x^5, and the three multiplications and three additions from the degree 3 polynomial in the quantity x^3 give the total operation count of 7 multiplies and 3 adds per evaluation. ◀

▶ **ADDITIONAL EXAMPLES**

1. Use nested multiplication to evaluate the polynomial $P(x) = x^6 - 2x^5 + 3x^4 - 4x^3 + 5x^2 - 6x + 7$ at $x = 2$.
2. Rewrite the polynomial $P(x) = 3x^{18} - 5x^{15} + 4x^{12} + 2x^6 - x^3 + 4$ in nested form. How many additions and how many multiplications are required for each input x?

Solutions for Additional Examples can be found at `goo.gl/BE9ytE`

0.1 Exercises

Solutions for Exercises numbered in **blue** can be found at `goo.gl/qeVIvL`

1. Rewrite the following polynomials in nested form. Evaluate with and without nested form at $x = 1/3$.

   (a) $P(x) = 6x^4 + x^3 + 5x^2 + x + 1$
   (b) $P(x) = -3x^4 + 4x^3 + 5x^2 - 5x + 1$
   (c) $P(x) = 2x^4 + x^3 - x^2 + 1$

2. Rewrite the following polynomials in nested form and evaluate at $x = -1/2$:

   (a) $P(x) = 6x^3 - 2x^2 - 3x + 7$
   (b) $P(x) = 8x^5 - x^4 - 3x^3 + x^2 - 3x + 1$
   (c) $P(x) = 4x^6 - 2x^4 - 2x + 4$

3. Evaluate $P(x) = x^6 - 4x^4 + 2x^2 + 1$ at $x = 1/2$ by considering $P(x)$ as a polynomial in x^2 and using nested multiplication.
4. Evaluate the nested polynomial with base points $P(x) = 1 + x(1/2 + (x - 2)(1/2 + (x - 3)(-1/2)))$ at (a) $x = 5$ and (b) $x = -1$.
5. Evaluate the nested polynomial with base points $P(x) = 4 + x(4 + (x - 1)(1 + (x - 2)(3 + (x - 3)(2))))$ at (a) $x = 1/2$ and (b) $x = -1/2$.
6. Explain how to evaluate the polynomial for a given input x, using as few operations as possible. How many multiplications and how many additions are required?
   (a) $P(x) = a_0 + a_5x^5 + a_{10}x^{10} + a_{15}x^{15}$
   (b) $P(x) = a_7x^7 + a_{12}x^{12} + a_{17}x^{17} + a_{22}x^{22} + a_{27}x^{27}$.
7. How many additions and multiplications are required to evaluate a degree n polynomial with base points, using the general nested multiplication algorithm?

0.1 Computer Problems

Solutions for Computer Problems numbered in **blue** can be found at `goo.gl/D6YLU2`

1. Use the function `nest` to evaluate $P(x) = 1 + x + \cdots + x^{50}$ at $x = 1.00001$. (Use the MATLAB `ones` command to save typing.) Find the error of the computation by comparing with the equivalent expression $Q(x) = (x^{51} - 1)/(x - 1)$.
2. Use `nest.m` to evaluate $P(x) = 1 - x + x^2 - x^3 + \cdots + x^{98} - x^{99}$ at $x = 1.00001$. Find a simpler, equivalent expression, and use it to estimate the error of the nested multiplication.

0.2 BINARY NUMBERS

In preparation for the detailed study of computer arithmetic in the next section, we need to understand the binary number system. Decimal numbers are converted from base 10 to base 2 in order to store numbers on a computer and to simplify computer operations like addition and multiplication. To give output in decimal notation, the process is reversed. In this section, we discuss ways to convert between decimal and binary numbers.

Binary numbers are expressed as

$$\dots b_2 b_1 b_0 . b_{-1} b_{-2} \dots ,$$

where each binary digit, or **bit**, is 0 or 1. The base 10 equivalent to the number is

$$\dots b_2 2^2 + b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} \dots .$$

For example, the decimal number 4 is expressed as $(100.)_2$ in base 2, and 3/4 is represented as $(0.11)_2$.

0.2.1 Decimal to binary

The decimal number 53 will be represented as $(53)_{10}$ to emphasize that it is to be interpreted as base 10. To convert to binary, it is simplest to break the number into integer and fractional parts and convert each part separately. For the number $(53.7)_{10} = (53)_{10} + (0.7)_{10}$, we will convert each part to binary and combine the results.

Integer part. Convert decimal integers to binary by dividing by 2 successively and recording the remainders. The remainders, 0 or 1, are recorded by starting at the decimal point (or more accurately, **radix**) and moving away (to the left). For $(53)_{10}$, we would have

$$\begin{aligned} 53 \div 2 &= 26 \text{ R } 1 \\ 26 \div 2 &= 13 \text{ R } 0 \\ 13 \div 2 &= \ \ 6 \text{ R } 1 \\ 6 \div 2 &= \ \ 3 \text{ R } 0 \\ 3 \div 2 &= \ \ 1 \text{ R } 1 \\ 1 \div 2 &= \ \ 0 \text{ R } 1. \end{aligned}$$

Therefore, the base 10 number 53 can be written in bits as 110101, denoted as $(53)_{10} = (110101.)_2$. Checking the result, we have $110101 = 2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53$.

Fractional part. Convert $(0.7)_{10}$ to binary by reversing the preceding steps. Multiply by 2 successively and record the integer parts, moving away from the decimal point to the right.

$$\begin{aligned} .7 \times 2 &= .4 + 1 \\ .4 \times 2 &= .8 + 0 \\ .8 \times 2 &= .6 + 1 \\ .6 \times 2 &= .2 + 1 \\ .2 \times 2 &= .4 + 0 \\ .4 \times 2 &= .8 + 0 \\ &\ \ \vdots \end{aligned}$$

Notice that the process repeats after four steps and will repeat indefinitely exactly the same way. Therefore,

$$(0.7)_{10} = (.1011001100110\ldots)_2 = (.1\overline{0110})_2,$$

where overbar notation is used to denote infinitely repeated bits. Putting the two parts together, we conclude that

$$(53.7)_{10} = (110101.1\overline{0110})_2.$$

0.2.2 Binary to decimal

To convert a binary number to decimal, it is again best to separate into integer and fractional parts.

Integer part. Simply add up powers of 2 as we did before. The binary number $(10101)_2$ is simply $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (21)_{10}$.

Fractional part. If the fractional part is finite (a terminating base 2 expansion), proceed the same way. For example,

$$(.1011)_2 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = \left(\frac{11}{16}\right)_{10}.$$

The only complication arises when the fractional part is not a finite base 2 expansion. Converting an infinitely repeating binary expansion to a decimal fraction can be done in several ways. Perhaps the simplest way is to use the shift property of multiplication by 2.

For example, suppose $x = (0.\overline{1011})_2$ is to be converted to decimal. Multiply x by 2^4, which shifts 4 places to the left in binary. Then subtract the original x:

$$\begin{aligned} 2^4 x &= 1011.\overline{1011} \\ x &= 0000.\overline{1011}. \end{aligned}$$

Subtracting yields

$$(2^4 - 1)x = (1011)_2 = (11)_{10}.$$

Then solve for x to find $x = (.\overline{1011})_2 = 11/15$ in base 10.

As another example, assume that the fractional part does not immediately repeat, as in $x = .10\overline{101}$. Multiplying by 2^2 shifts to $y = 2^2 x = 10.\overline{101}$. The fractional part of y, call it $z = .\overline{101}$, is calculated as before:

$$\begin{aligned} 2^3 z &= 101.\overline{101} \\ z &= 000.\overline{101}. \end{aligned}$$

Therefore, $7z = 5$, and $y = 2 + 5/7$, $x = 2^{-2} y = 19/28$ in base 10. It is a good exercise to check this result by converting 19/28 to binary and comparing to the original x.

Binary numbers are the building blocks of machine computations, but they turn out to be long and unwieldy for humans to interpret. It is useful to use base 16 at times just to present numbers more easily. **Hexadecimal numbers** are represented by the 16 numerals $0, 1, 2, \ldots, 9, A, B, C, D, E, F$. Each hex number can be represented by 4 bits. Thus $(1)_{16} = (0001)_2$, $(8)_{16} = (1000)_2$, and $(F)_{16} = (1111)_2 = (15)_{10}$. In the next section, MATLAB's `format hex` for representing machine numbers will be described.

▸ ADDITIONAL EXAMPLES

*1. Convert the decimal number 98.6 to binary.

2. Convert the repeating binary number $0.1\overline{000111}$ to a base 10 fraction.

Solutions for Additional Examples can be found at goo.gl/jVKlKJ
(* example with video solution)

0.2 Exercises

Solutions for Exercises numbered in **blue** can be found at goo.gl/8y092J

1. Find the binary representation of the base 10 integers. (a) 64 (b) 17 (c) 79 (d) 227
2. Find the binary representation of the base 10 numbers. (a) $1/8$ (b) $7/8$ (c) $35/16$ (d) $31/64$
3. Convert the following base 10 numbers to binary. Use overbar notation for nonterminating binary numbers. (a) 10.5 (b) $1/3$ (c) $5/7$ (d) 12.8 (e) 55.4 (f) 0.1
4. Convert the following base 10 numbers to binary. (a) 11.25 (b) $2/3$ (c) $3/5$ (d) 3.2 (e) 30.6 (f) 99.9
5. Find the first 15 bits in the binary representation of π.
6. Find the first 15 bits in the binary representation of e.
7. Convert the following binary numbers to base 10: (a) 1010101 (b) 1011.101 (c) $10111.\overline{01}$ (d) $110.\overline{10}$ (e) $10.\overline{110}$ (f) $110.1\overline{101}$ (g) $10.010\overline{1101}$ (h) $111.\overline{1}$
8. Convert the following binary numbers to base 10: (a) 11011 (b) 110111.001 (c) $111.\overline{001}$ (d) $1010.\overline{01}$ (e) $10111.1\overline{0101}$ (f) $1111.010\overline{001}$

0.3 FLOATING POINT REPRESENTATION OF REAL NUMBERS

There are several models for computer arithmetic of floating point numbers. The models in modern use are based on the IEEE 754 Floating Point Standard. The Institute of Electrical and Electronics Engineers (IEEE) takes an active interest in establishing standards for the industry. Their floating point arithmetic format has become the common standard for single precision and double precision arithmetic throughout the computer industry.

Rounding errors are inevitable when finite-precision computer memory locations are used to represent real, infinite precision numbers. Although we would hope that small errors made during a long calculation have only a minor effect on the answer, this turns out to be wishful thinking in many cases. **Simple algorithms, such as Gaussian elimination or methods for solving differential equations, can magnify microscopic errors to macroscopic size**. In fact, a main theme of this book is to help the reader to recognize when a calculation is at risk of being unreliable due to magnification of the small errors made by digital computers and to know how to avoid or minimize the risk.

0.3.1 Floating point formats

The IEEE standard consists of a set of binary representations of real numbers. A **floating point number** consists of three parts: the **sign** (+ or −), a **mantissa**, which contains the string of significant bits, and an **exponent**. The three parts are stored together in a single computer **word**.

There are three commonly used levels of precision for floating point numbers: single precision, double precision, and extended precision, also known as long-double precision. The number of bits allocated for each floating point number in the three formats is 32, 64, and 80, respectively. The bits are divided among the parts as follows:

precision	sign	exponent	mantissa
single	1	8	23
double	1	11	52
long double	1	15	64

All three types of precision work essentially the same way. The form of a **normalized** IEEE floating point number is

$$\pm 1.bbb\ldots b \times 2^p, \tag{0.6}$$

where each of the N b's is 0 or 1, and p is an M-bit binary number representing the exponent. Normalization means that, as shown in (0.6), the leading (leftmost) bit must be 1.

When a binary number is stored as a normalized floating point number, it is "left-justified," meaning that the leftmost 1 is shifted just to the left of the radix point. The shift is compensated by a change in the exponent. For example, the decimal number 9, which is 1001 in binary, would be stored as

$$+1.001 \times 2^3,$$

because a shift of 3 bits, or multiplication by 2^3, is necessary to move the leftmost one to the correct position.

For concreteness, we will specialize to the double precision format for most of the discussion. The double precision format, common in C compilers, python, and MATLAB, uses exponent length $M = 11$ and mantissa length $N = 52$. Single and long double precision are handled in the same way, but with different choices for M and N as specified above.

The double precision number 1 is

$$+1.\boxed{0000000000000000000000000000000000000000000000000000} \times 2^0,$$

where we have boxed the 52 bits of the mantissa. The next floating point number greater than 1 is

$$+1.\boxed{0000000000000000000000000000000000000000000000000001} \times 2^0,$$

or $1 + 2^{-52}$.

DEFINITION 0.1 The number **machine epsilon**, denoted ϵ_{mach}, is the distance between 1 and the smallest floating point number greater than 1. For the IEEE double precision floating point standard,

$$\epsilon_{\text{mach}} = 2^{-52}.$$

❐

The decimal number $9.4 = (1001.\overline{0110})_2$ is left-justified as

$$+1.\boxed{0010110011001100110011001100110011001100110011001100}\,110\ldots \times 2^3,$$

where we have boxed the first 52 bits of the mantissa. A new question arises: How do we fit the infinite binary number representing 9.4 in a finite number of bits?

We must truncate the number in some way, and in so doing we necessarily make a small error. One method, called **chopping**, is to simply throw away the bits that fall