Chapter 2

Computing with Floating Point Numbers

The first part of this chapter gives an elementary introduction to the representation of computer numbers and computer arithmetic. First, we introduce the computer numbers typically available on modern computers (and available through MATLAB). Next, we discuss the quality of the approximations produced by numeric computing.

In the second part of the chapter we present a number of simple examples that illustrate some of the pitfalls of numerical computing. There are several such examples, each of them suitable for short computer projects. They illustrate a number of pitfalls, with particular emphasis on the effects of catastrophic cancelation, floating–point overflow and underflow, and the accumulation of roundoff errors.

2.1 Numbers

Most scientific programming languages (including MATLAB) provide users with at least two types of numbers: integers and floating–point numbers. In fact, MATLAB supports many other types too but we concentrate on integers and floating–point numbers as those most likely to be used in scientific computing. The floating–point numbers are a subset of the real numbers, so we begin by discussing some basics about numbers.

2.1.1 Decimal and Floating Point Numbers

In math and science courses we typically use the decimal (or base-10) system to denote integer and non-integer numbers, such as 0.1 and 93.75. Note that these specific examples can be written as rational numbers (fractions involving integers),

$$\frac{1}{10} = 0.1 \qquad \text{and} \qquad \frac{750}{8} = 93.75$$

Some numbers, such as

$$\pi = 3.141592653\cdots$$

cannot be written with a finite number of decimal digits. Therefore, when doing hand calculations or calculations on a computer, we are forced to use an approximation of the number by truncating the decimal digits (usually using rounding), for example

$$\pi \approx 3.1415927 \tag{2.1}$$

Note that this means that our representation of π is an approximation; the more digits we use, the better the approximation. This is usually referred to as *precision*. For example, the 8-digit decimal

approximation of π given in (2.1) is more accurate than the 3-digit decimal approximation

$$\pi \approx 3.14$$

Precision is a very important topic in scientific computation!

Computer vendors (and users) need to consider how much precision they want to use when representing numbers on a computer. Before we can discuss this, though, it is essential to have a unique format for numbers. For example, we could represent the rational number 750/8 in many ways,

$$\begin{aligned} \frac{750}{8} &= 93.75 \\ &= 9.375 \times 10^1 \\ &= 0.9375 \times 10^2 \\ &= 937.5 \times 10^{-1} \\ &\vdots \end{aligned}$$

As users of computers, this lack of uniqueness may not be especially important, but to understand numerical precision and how computers do arithmetic, it is important. In this book define the unique, or normalized base-10 number to have the form

$$x = \pm \left(d_1.d_2 \cdots d_t d_{t+1} \cdots\right) \times 10^e \tag{2.2}$$

where d_i are integers, $0 \leq d_i \leq 9$, with $d_1 \neq 0$, and e is an integer exponent.

On a computer, we would not be able to represent an infinite number of decimal digits, so we need to terminate (e.g., by rounding) at some point. In addition, we cannot represent an infinitely large (positive or negative) exponent. Thus, we define a t-digit base-10 (decimal) floating point approximation of x to be:

$$\mathrm{fl}\,(x) = \pm \left(d_1.d_2 \cdots d_t\right) \times 10^e \tag{2.3}$$

where d_i are integers, $0 \leq d_i \leq 9$, with $d_1 \neq 0$, and e is an integer exponent that also must satisfy $L \leq e \leq U$, where L is a lower bound (e.g., a large negative number) and U is an upper bound (e.g., a large positive number) for e. We will often dispense with the notation $\mathrm{fl}\,(\cdot)$ when it is clear that we are using floating point numbers. Obviously, we cannot represent all real numbers on a computer, and thus the set of all floating point numbers is a subset of the real numbers. The number of decimal digits, t, is often referred to as the precision. Notice that the normalized floating point representation defined in (2.3) can also be written as

$$\mathrm{fl}\,(x) = \pm \left(d_1 + \frac{d_2}{10} + \cdots + \frac{d_t}{10^{t-1}}\right) \times 10^e \tag{2.4}$$

This latter form is convenient if we want to generalize the concept of floating point numbers to other bases, as will be done later in this section.

Example 2.1.1. Consider the rational number $x = \frac{750}{8}$.

(a) The normalized base-10 representation of x is

$$\begin{aligned} \frac{750}{8} &= \left(9 + \frac{3}{10} + \frac{7}{100} + \frac{5}{1000}\right) \times 10^1 \\ &= 9.375 \times 10^1 \end{aligned}$$

(b) A 3-digit base 10 normalized floating point approximation of x is

$$\begin{aligned} \mathrm{fl}\left(\frac{750}{8}\right) &= \left(9 + \frac{3}{10} + \frac{8}{100}\right) \times 10^1 \\ &= 9.38 \times 10^1 \end{aligned}$$

2.1.2 Binary Numbers

It is now easy to generalize the base-10 representation of floating point numbers to an arbitrary base β. Specifically, we define a normalized base-β floating point number as

$$\begin{aligned} \mathrm{fl}\left((x)_\beta\right) &= \pm\left(\frac{d_1}{\beta^0}+\frac{d_2}{\beta^1}+\cdots+\frac{d_n}{\beta^{n-1}}\right)\times\beta^e \\ &= \pm\left(d_1.d_2\cdots d_t\right)\times\beta^e \end{aligned}$$

where d_i are integers, $0 \le d_i < \beta$, $d_1 \neq 0$, and e is an integer exponent, $L \le e \le U$, usually also represented as a base-β number. For computers, the most commonly used base is $\beta = 2$, which means

$$\begin{aligned} \mathrm{fl}\left((x)_2\right) &= \pm\left(d_1+\frac{d_2}{2}+\cdots+\frac{d_n}{2^{n-1}}\right)\times 2^e \\ &= \pm\left(d_1.d_2\cdots d_n\right)\times 2^e \end{aligned}$$

where d_i are either 0 or 1 (i.e., binary digits, or bits), $d_1 \neq 0$, and e is a base-2 integer exponent, with $L \le e \le U$. The value n specifies the number of bits allocated in the computer to store the fraction part of the number; this will be discussed below in more detail. There is a relation between this number of bits and the base-10 precision value t.

Example 2.1.2. Consider the rational number $x = \frac{750}{8}$.

(a) Recall that the normalized base-10 representation of x is

$$\begin{aligned} \left(\frac{750}{8}\right)_{10} &= \left(9+\frac{3}{10}+\frac{7}{100}+\frac{5}{1000}\right)\times 10^1 \\ &= 9.375\times 10^1 \end{aligned}$$

(b) The normalized base-2 representation of x is

$$\begin{aligned} \left(\frac{750}{8}\right)_{2} &= \left(1+\frac{0}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{0}{32}+\frac{1}{64}+\frac{1}{128}+\frac{1}{256}\right)\times 2^6 \\ &= (1.01110111)\times 2^6 \end{aligned}$$

Here we are using the notation $(\cdot)_\beta$ to emphasize which basis is being used to represent the number. Usually it is clear from the context, in which case we will dispense from using this notation.

2.1.3 Computer Representation of Numbers

All computer numbers are stored as a sequence of binary digits, or bits. A bit assumes one of two values, 0 or 1. The bits allocated to a number are usually ordered from right to left. For example, if the number v is stored using 8 bits, we may write it

$$\text{stored}(v) = b_7b_6b_5b_4b_3b_2b_1b_0$$

Here, b_i represents the i^{th} bit of v; the indices assigned to these bits start at 0 and increase to the left. Because each bit can assume the value 0 or 1, there are $2^8 = 256$ distinct patterns of bits. However, the value associated with each pattern depends on what *kind* of number v that represents.

Integers

The most commonly used type of integers are the **signed integers**. Most computers store signed integers using *two's complement notation*; in this representation the 8-bit signed integer i is stored as

$$\text{stored}(i) = b_7b_6b_5b_4b_3b_2b_1b_0$$

which is assigned the value

$$\text{value}(i) = b_7(-2^7) + b_6(2^6) + \cdots + b_0(2^0)$$

When we compute with integers we normally use signed integers. So, for example, among the 256 different 8-bit signed integers, the smallest value is -128 and the largest is 127. This asymmetry arises because the total number of possible bit patterns, $2^8 = 256$, is even and one bit pattern, `00000000`, is assigned to zero.

An 8-bit **unsigned integer** j stored as

$$\text{stored}(j) = b_7b_6b_5b_4b_3b_2b_1b_0$$

is assigned the value

$$\text{value}(j) = b_7(2^7) + b_6(2^6) + b_5(2^5) + \cdots + b_0(2^0)$$

Unsigned integers are used mainly in indexing and in the representation of floating–point numbers. Among the 256 different 8-bit unsigned integers, the smallest value, `00000000`, is 0 and the largest, `11111111`, is 255.

Example 2.1.3. Consider the decimal integer 13. Its representation as an unsigned integer is `00001101`, because

$$13 = 0 \cdot (2^7) + 0 \cdot (2^6) + 0 \cdot (2^5) + 0 \cdot (2^4) + 1 \cdot (2^3) + 1 \cdot (2^2) + 0 \cdot (2^1) + 1 \cdot (2^0)$$

The signed integer representation is also `00001101` because

$$13 = 0 \cdot (-2^7) + 0 \cdot (2^6) + 0 \cdot (2^5) + 0 \cdot (2^4) + 1 \cdot (2^3) + 1 \cdot (2^2) + 0 \cdot (2^1) + 1 \cdot (2^0)$$

Now consider the decimal integer -13. It obviously has no representation as an unsigned integer. As a signed integer we first observe that

$$-13 = -128 + 115 = 1 \cdot (-2^7) + 115$$

Now we need to use the remaining seven bits to represent the decimal integer 115:

$$115 = 1 \cdot (2^6) + 1 \cdot (2^5) + 1 \cdot (2^4) + 0 \cdot (2^3) + 0 \cdot (2^2) + 1 \cdot (2^1) + 1 \cdot (2^0)$$

Thus, the two's complement binary form of the decimal integer -13 is `11110011`, or

$$-13 = 1 \cdot (-2^7) + 1 \cdot (2^6) + 1 \cdot (2^5) + 1 \cdot (2^4) + 0 \cdot (2^3) + 0 \cdot (2^2) + 1 \cdot (2^1) + 1 \cdot (2^0)$$

Problem 2.1.1. *Determine a formula for the smallest and largest n-bit unsigned integers. What are the numerical values of the smallest and largest n-bit unsigned integers for each of $n = 8, 11, 16, 32, 64$?*

Problem 2.1.2. *Determine a formula for the smallest and largest n-bit signed integers. What are numerical values of the smallest and largest n-bit signed integers for each of $n = 8, 11, 16, 32, 64$?*

Problem 2.1.3. *List the value of each 8-bit signed integer for which the negative of this value is* not *also an 8-bit signed integer.*

2.1.4 Floating-Point Numbers

In early computers the types of floating–point numbers available depended not only on the programming language but also on the computer. The situation changed dramatically after 1985 with the widespread adoption of the *ANSI/IEEE 754-1985 Standard for Binary Floating–Point Arithmetic*. For brevity we'll call this the *Standard*. It defines several types of floating–point numbers. The most widely implemented is its 64**-bit double precision (DP)** type. In most computer languages the programmer can also use the 32**-bit single precision (SP)** type of floating–point numbers. If a program uses just one type of floating–point number, we refer to that type as the **working precision (WP)** floating–point numbers. (WP is usually either DP or SP. In MATLAB, WP=DP.) The important quantity **machine epsilon**, $\epsilon_{\rm WP}$, is the distance from 1 to the next larger number in the working precision.

An important property of WP numbers is that *the computed result of every sum, difference, product, or quotient of a pair of WP numbers is a valid WP number*. For this property to be possible, the WP numbers in the *Standard* include representations of finite real numbers as well as the special numbers ± 0, $\pm\infty$ and NaN (Not-a-Number). For example, the quotient $1/0$ is the special number ∞ and the product $\infty \times 0$, which is mathematically undefined, has the value NaN.

In the Standard, an SP number x is stored using 32 bits, which are partitioned as

$$\text{stored}(x) = \overbrace{\boxed{b_{31}}}^{s(x)}\overbrace{\boxed{b_{30}b_{29}\cdots b_{23}}}^{e(x)}\overbrace{\boxed{b_{22}b_{21}\quad\cdots\quad b_0}}^{f(x)}$$

and a DP number x is stored using 64 bits, which are partitioned as

$$\text{stored}(y) = \overbrace{\boxed{b_{63}}}^{s(x)}\overbrace{\boxed{b_{62}b_{61}\cdots b_{52}}}^{e(x)}\overbrace{\boxed{b_{51}b_{50}\qquad\cdots\qquad b_0}}^{f(x)}$$

The boxes illustrate how the bits are partitioned into 3 fields:

- One bit is an unsigned integer $s(x)$, the **sign bit of** x; $s(x) = 0$ indicates a positive number, and $s(x) = 1$ indicates a negative number.
- The next partition uses a certain number of bits (8 for single, 11 for double) to represent $e(x)$, as an unsigned integer, and is called the **biased exponent of** x.
- The next partition uses the remaining available bits (23 for single, 52 for double) to represent $f(x)$, as an unsigned integer, and is called the **fraction of** x.

Note that this definition implies that the machine epsilon, that is the distance from 1 to the next larger number in the given working precision is

$$\begin{aligned} \text{single precision:} \qquad & \epsilon_{\rm SP} = 2^{-23} \approx 1.1921 \times 10^{-7} \\ \text{double precision:} \qquad & \epsilon_{\rm DP} = 2^{-52} \approx 2.2204 \times 10^{-16} \end{aligned}$$

The precise details of how to convert the bits into a decimal value, which are not at first obvious, are given in Section 2.1.5, but it is important to notice that there are limitations on what numbers can be represented. Specifically:

Exponent Limitation: This can lead to either *overflow* or *underflow*.

Overflow: This occurs if the exponent is a too large positive number to be represented by the available bits in $e(x)$. In this case, $|x|$ is very large, and the Standard returns a result of either `Inf` or $-$`Inf`.

- **Underflow:** This occurs if the exponent is a too large negative number to be represented by the available bits in $e(x)$. In this case, $|x|$ is a tiny number, and the Standard returns a result of 0.

Fraction Limitation: Some fraction parts cannot be represented by a finite number of bits. This is most easily seen for irrational numbers, such as π or $\sqrt{2}$. Obviously these cannot be written with a finite number of decimal digits, and hence cannot be represented with a finite number of bits. But the situation can occur with relatively simple rational numbers. For example, the decimal number 1/10 does not have a finite binary representation, but instead must be written as the repeating (but infinite) series

$$\left(\frac{1}{10}\right)_2 = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \frac{0}{2^{14}} + \cdots$$

In these situations, the number of bits must be "cutoff" at some point, leading to **roundoff error**.

Section 2.2 discusses in more detail issues of underflow, overflow, and propagation of roundoff errors in scientific computing.

2.1.5 Additional Details on Floating Point Numbers

Double Precision Numbers

As stated in Section 2.1.4 In the Standard, a DP number y is stored as

$$\text{stored}(y) = \overbrace{\boxed{b_{63}}}^{s(y)}\overbrace{\boxed{b_{62}b_{61}\cdots b_{52}}}^{e(y)}\overbrace{\boxed{b_{51}b_{50}\quad\cdots\quad b_0}}^{f(y)}$$

The boxes illustrate how the 64 bits are partitioned into 3 fields: a 1-bit unsigned integer $s(y)$, the **sign bit of** y, an 11-bit unsigned integer $e(y)$, the **biased exponent of** y, and a 52-bit unsigned integer $f(y)$, the **fraction of** y. Because the sign of y is stored explicitly, with $s(y) = 0$ for positive and $s(y) = 1$ for negative y, each DP number can be negated. The 11-bit unsigned integer $e(y)$ represents values from 0 through 2047:

- If $e(y) = 2047$, then y is a special number.
- If $0 < e(y) < 2047$, then y is a **normalized** floating–point number with value

  $$\text{value}(y) = (-1)^{s(y)} \cdot \left\{1 + \frac{f(y)}{2^{52}}\right\} \cdot 2^{E(y)}$$

  where $E(y) = e(y) - 1023$ is the (unbiased) **exponent of** y.
- If $e(y) = 0$ and $f(y) \neq 0$, then y is a **denormalized** floating–point number with value

  $$\text{value}(y) = (-1)^{s(y)} \cdot \left\{0 + \frac{f(y)}{2^{52}}\right\} \cdot 2^{-1022}$$

- If $e(y) = 0$ and $f(y) = 0$, then y has value **zero**.

The **significand of** y is $1 + \frac{f(y)}{2^{52}}$ for normalized numbers, $0 + \frac{f(y)}{2^{52}}$ for denormalized numbers, and 0 for zero. The normalized DP numbers y with exponent $E(y) = k$ belong to **binade** k. For example, binade -1 consists of the numbers y for which $|y| \in \left[\frac{1}{2}, 1\right)$ and binade 1 consists of the numbers y for which $|y| \in [2, 4)$. In general, binade k consists of the numbers y for which $2^k \leq |y| < 2^{k+1}$. Each DP binade contains the same number of positive DP numbers. We define the **DP machine epsilon** ϵ_{DP} as the distance from 1 to the next larger DP number. For a finite nonzero DP number y, we define $\text{ulp}_{\text{DP}}(y) \equiv \epsilon_{\text{DP}} 2^{E(y)}$ as the **DP unit-in-the-last-place of** y.

Example 2.1.4. Consider the real number $y = -1/6$. This number is not directly representable in the Standard but we will compute the nearest Standard floating–point number to it. Note that $-1/6 = -(8/6)2^{-3}$ so $E(y) = -3$, $s(y) = 1$, $e(y) = 1020$ (hence the number y is a normalized DP floating–point number), and $f(y)$ is the closest integer to $2^{52}/3$ (because $8/6 - 1 = 1/3$). This gives $f(y) = 1501199875790165$ as a decimal and 1010101010101010101010101010101010101010101010101010101 in binary.

The above representation as defined in the Standard is rather too cumbersome for our purposes in this text. Below, without significant loss of generality, we will simplify the representation. We use as our **scientific notation** for floating point numbers that any DP number x is represented $S(x) \cdot 2^{e(x)}$ where the exponent $e(x)$ is an integer and the mantissa $S(x)$ satisfies $1 \le |S(x)| < 2$.

Problem 2.1.4. *Write $y = 6.1$ as a normalized DP floating–point number*

Problem 2.1.5. *What are the smallest and largest possible significands for normalized DP numbers?*

Problem 2.1.6. *What are the smallest and largest positive normalized DP numbers?*

Problem 2.1.7. *How many DP numbers are in each binade? Sketch enough of the positive real number line so that you can place marks at the endpoints of each of the DP binades -3, -2, -1, 0, 1, 2 and 3. What does this sketch suggest about the spacing between successive positive DP numbers?*

Problem 2.1.8. *Show that $\epsilon_{DP} = 2^{-52}$.*

Problem 2.1.9. *Show that $y = q \cdot ulp_{DP}(y)$ where q is an integer with $1 \le |q| \le 2^{53} - 1$.*

Problem 2.1.10. *Show that neither $y = \frac{1}{3}$ nor $y = \frac{1}{10}$ is a DP number. Hint: If y is a DP number, then $2^m y$ is a 53-bit unsigned integer for some (positive or negative) integer m.*

Problem 2.1.11. *What is the largest positive integer n such that $2^n - 1$ is a DP number?*

Single Precision Numbers

This section has been included for completeness and may be omitted unless great emphasis is to be given to the definition of computer arithmetic.

In the Standard, a single precision (SP) number x is stored as

$$\text{stored}(x) = \overbrace{\boxed{b_{31}}}^{s(x)}\overbrace{\boxed{b_{30}b_{29}\cdots b_{23}}}^{e(x)}\overbrace{\boxed{b_{22}b_{21}\quad\cdots\quad b_0}}^{f(x)}$$

The boxes illustrate how the 32 bits are partitioned into 3 fields: a 1-bit unsigned integer $s(x)$, the **sign bit of** x, an 8-bit unsigned integer $e(x)$, the **biased exponent of** x, and a 23-bit unsigned integer $f(x)$, the **fraction of** x. The sign bit $s(x) = 0$ for positive and $s(x) = 1$ for negative SP numbers. The 8-bit unsigned integer $e(x)$ represents values in the range from 0 through 255:

- If $e(x) = 255$, then x is a special number.
- If $0 < e(x) < 255$, then x is a **normalized** floating–point number with value

$$\text{value}(x) = (-1)^{s(x)} \cdot \left\{1 + \frac{f(x)}{2^{23}}\right\} \cdot 2^{E(x)}$$

  where $E(x) \equiv e(x) - 127$ is the (unbiased) **exponent of** x.

- If $e(x) = 0$ and $f(x) \neq 0$, then x is a **denormalized** floating–point number with value

$$\text{value}(x) = (-1)^{s(x)} \cdot \left\{ 0 + \frac{f(x)}{2^{23}} \right\} \cdot 2^{-126}$$

- If both $e(x) = 0$ and $f(x) = 0$, then x is **zero**.

The **significand of** x is $1 + \frac{f(x)}{2^{23}}$ for normalized numbers, $0 + \frac{f(x)}{2^{23}}$ for denormalized numbers, and 0 for zero. The normalized SP numbers y with exponent $E(y) = k$ belong to **binade** k. For example, binade -1 consists of the numbers y for which $|y| \in \left[\frac{1}{2}, 1\right)$ and binade 1 consists of the numbers y for which $|y| \in [2, 4)$. In general, binade k consists of the numbers y for which $2^k \leq |y| < 2^{k+1}$. Each SP binade contains the same number of positive SP numbers. We define the **SP machine epsilon** ϵ_{SP} as the distance from 1 to the next larger SP number. For a finite nonzero SP number x, we define $\text{ulp}_{\text{SP}}(x) \equiv \epsilon_{\text{SP}} 2^{E(x)}$ as the **SP unit-in-the-last-place of** x.

Problem 2.1.12. *Write $y = 6.1$ as a normalized SP floating–point number*

Problem 2.1.13. *What are the smallest and largest possible significands for normalized SP numbers?*

Problem 2.1.14. *What are the smallest and largest positive normalized SP numbers?*

Problem 2.1.15. *How many SP numbers are in each binade? Sketch enough of the positive real number line so that you can place marks at the endpoints of each of the SP binades -3, -2, -1, 0, 1, 2 and 3. What does this sketch suggest about the spacing between consecutive positive SP numbers?*

Problem 2.1.16. *Compare the sketches produced in Problems 2.1.7 and 2.1.15. How many DP numbers lie between consecutive positive SP numbers? We say that distinct real numbers x and y are resolved by SP numbers if the SP number nearest x is not the same as the SP number nearest y. Write down the analogous statement for DP numbers. Given two distinct real numbers x and y, are x and y more likely to be resolved by SP numbers or by DP numbers? Justify your answer.*

Problem 2.1.17. *Show that $\epsilon_{SP} = 2^{-23}$.*

Problem 2.1.18. *Show that $x = p \cdot ulp_{SP}(x)$ where p is an integer with $1 \leq |p| \leq 2^{24} - 1$.*

Problem 2.1.19. *If x is a normalized SP number in binade k, what is the distance to the next larger SP number? Express your answer in terms of ϵ_{SP}.*

Problem 2.1.20. *Show that neither $x = \frac{1}{3}$ nor $x = \frac{1}{10}$ is a SP number. Hint: If x is a SP number, then $2^m x$ is an integer for some (positive or negative) integer m.*

Problem 2.1.21. *What is the largest positive integer n such that $2^n - 1$ is a SP number?*

2.2 Computer Arithmetic

What kind of numbers should a program use? Signed integers may be appropriate if the input data are integers and the only operations used are addition, subtraction, and multiplication. Floating–point numbers are appropriate if the data involves fractions, has large or small magnitudes, or if the operations include division, square root, or computing transcendental functions like the sine.

2.2.1 Integer Arithmetic

In integer computer arithmetic the computed sum, difference, or product of integers is the exact result, except when **integer overflow** occurs. This happens when to represent the result of an integer arithmetic operation more bits are needed than are available for the result. Thus, for the 8-bit signed integers $i = 126$ and $j = 124$ the computed value $i + j = 250$ is a number larger than the value of any 8-bit signed integer, so it overflows. So, *in the absence of integer overflow, integer arithmetic performed by the computer is exact*. That you have encountered integer overflow may not always be immediately apparent. The computer may simply return an (incorrect) integer result! For example, in MATLAB, the computation:

```
int8(126) + int8(124)
```

returns an `int8` value of 127. Similarly, the computation

```
int8(-126) + int8(-124)
```

returns an `int8` value of -128. Fortunately the default number of bits used to store integers in most programming languages, including MATLAB, is more than 8, so this kind of situation is unlikely to occur.

2.2.2 Floating-Point Arithmetic

The set of all integers is **closed** under the arithmetic operations of addition, subtraction, and multiplication. That is the sum, difference, or product of two integers is another integer. Similarly, the set of all real numbers is closed under the arithmetic operations of addition, subtraction, and multiplication, and division (except by zero). Unfortunately, *the set of all DP numbers is not closed under any of the operations of add, subtract, multiply, or divide.* For example, both $x = 2^{52} + 1$ and $y = 2^{52} - 1$ are DP numbers and yet their product $xy = 2^{104} - 1$ is not an DP number. To make the DP numbers closed under the arithmetic operations of addition, subtraction, multiplication, and division, the *Standard* modifies slightly the result produced by each operation. Specifically, it defines

$$\begin{aligned} x \oplus y &\equiv \mathrm{fl}_{\mathrm{DP}}(x + y) \\ x \ominus y &\equiv \mathrm{fl}_{\mathrm{DP}}(x - y) \\ x \otimes y &\equiv \mathrm{fl}_{\mathrm{DP}}(x \times y) \\ x \oslash y &\equiv \mathrm{fl}_{\mathrm{DP}}(x / y) \end{aligned}$$

Here, $\mathrm{fl}_{\mathrm{DP}}(z)$ is the DP number *closest* to the real number z; that is, $\mathrm{fl}_{\mathrm{DP}}()$ is the **DP rounding function**[1]. So, for example, the first equality states that $x \oplus y$, the value assigned to the sum of the DP numbers x and y, is the DP number $\mathrm{fl}_{\mathrm{DP}}(x + y)$. In a general sense, no approximate arithmetic on DP numbers can be more accurate than that specified by the *Standard*.

In summary, *floating–point arithmetic is inherently approximate; the computed value of any sum, difference, product, or quotient of DP numbers is equal to the exact value rounded to the nearest floating–point DP number.* In the next section we'll discuss how to measure the quality of this approximate arithmetic.

Problem 2.2.1. *Show that $2^{104} - 1$ is not a DP number. Hint: Recall Problem 2.1.11.*

Problem 2.2.2. *Show that each of $2^{53} - 1$ and 2 is a DP number, but that their sum is not a DP number. So, the set of all DP numbers is not closed under addition. Hint: Recall Problem 2.1.11.*

Problem 2.2.3. *Show that the set of DP numbers is not closed under subtraction; that is, find two DP numbers whose difference is not a DP number.*

[1] When z is midway between two adjacent DP numbers, $\mathrm{fl}_{\mathrm{DP}}(z)$ is the one whose fraction is even.

Problem 2.2.4. *Show that the set of DP numbers is not closed under division, that is find two nonzero DP numbers whose quotient is not a DP number. Hint: Consider Problem 2.1.10.*

Problem 2.2.5. *Assuming floating–point underflow does not occur, why can any DP number x be divided by 2 exactly? Hint: Consider the representation of x as a DP number.*

Problem 2.2.6. *Let x be a DP number. Show that $fl_{DP}(x) = x$.*

Problem 2.2.7. *Let each of x, y and $x + y$ be a DP number. What is the value of the DP number $x \oplus y$? State the extension of your result to the difference, product and quotient of DP numbers.*

Problem 2.2.8. *The real numbers x, y and z satisfy the* ***associative law of addition****:*

$$(x + y) + z = x + (y + z)$$

Consider the DP numbers $a = -2^{60}$, $b = 2^{60}$ and $c = 2^{-60}$. Show that

$$(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$$

So, in general DP addition is not associative. Hint: Show that $b \oplus c = b$.

Problem 2.2.9. *The real numbers x, y and z satisfy the* ***associative law of multiplication****:*

$$(x \times y) \times z = x \times (y \times z),$$

Consider the DP numbers $a = 1 + 2^{-52}$, $b = 1 - 2^{-52}$ and $c = 1.5 + 2^{-52}$. Show that

$$(a \otimes b) \otimes c \neq a \otimes (b \otimes c)$$

So, in general DP multiplication is not associative. Hint: Show that $a \otimes b = 1$ and $b \otimes c = 1.5 - 2^{-52}$.

Problem 2.2.10. *The real numbers x, y and z satisfy the* ***distributive law****:*

$$x \times (y + z) = (x \times y) + (x \times z)$$

Choose values of the DP numbers a, b and c such that

$$a \otimes (b \oplus c) \neq (a \otimes b) \oplus (a \otimes c)$$

So, in general DP arithmetic is not distributive.

Problem 2.2.11. *Define the SP rounding function $fl_{SP}()$ that maps real numbers into SP numbers. Define the values of $x \oplus y$, $x \ominus y$, $x \otimes y$ and $x \oslash y$ for SP arithmetic.*

Problem 2.2.12. *Show that $2^{24} - 1$ and 2 are SP numbers, but their sum is not a SP number. So, the set of all SP numbers is not closed under addition. Hint: Recall Problem 2.1.21.*

2.2.3 Quality of Approximations

To illustrate the use of approximate arithmetic we employ **normalized decimal scientific notation**. The *Standard* represents numbers in binary notation. We use decimal representation to simplify the development since the reader will be more familiar with decimal than binary arithmetic.

Using the notation in equations (2.2) and (2.3), a nonzero real number T is represented as

$$T = \pm\, m(T) \cdot 10^{e(T)}$$

where $1 \le m(T) < 10$ is a real number and $e(T)$ is a positive, negative or zero integer. Here, $m(T)$ is the decimal **significand** and $e(T)$ is the decimal **exponent** of T. For example,

$$\begin{aligned} 120. &= (1.20) \cdot 10^2 \\ \pi &= (3.14159\ldots) \cdot 10^0 \\ -0.01026 &= (-1.026) \cdot 10^{-2} \end{aligned}$$

For the real number $T = 0$, we define $m(T) = 1$ and $e(T) = -\infty$.

For any integer k, **decade** k is the set of real numbers whose exponents $e(T) = k$. So, the decade k is the set of real numbers with values whose magnitudes are in the half open interval $[10^k, 10^{k+1})$.

For a nonzero number T, its i^{th} **significant digit** is the i^{th} digit of $m(T)$, counting to the right starting with the units digit. So, the units digit is the 1^{st} significant digit, the tenths digit is the 2^{nd} significant digit, the hundredths digit is the 3^{rd} significant digit, etc. For the value π listed above, the 1^{st} significant digit is 3, the 2^{nd} significant digit is 1, the 3^{rd} significant digit is 4, etc.

Frequently used measures of the error $A - T$ in A as an approximation to the true value T are

$$\begin{aligned} \text{absolute error} &= |A - T| \\ \text{absolute relative error} &= \left|\frac{A - T}{T}\right| \end{aligned}$$

with the relative error being defined only when $T \neq 0$. The approximation A to T is said to be q–**digits accurate** if the absolute error is less than $\frac{1}{2}$ of one unit in the q^{th} significant digit of T. Since $1 \le m(T) < 10$, A is a q–digits approximation to T if

$$|A - T| \le \frac{1}{2}|m(T)|10^{e(T)}10^{-q} \le \frac{10^{e(T)-q+1}}{2}$$

If $T \neq 0$, then dividing the above inequality by $T = m(T)10^{e(T)}$, we obtain the following statement for relative errors:

If the absolute relative error

$$\frac{|A - T|}{|T|} \le r$$

then A is a q–digits accurate approximation to T provided that $q \le -\log_{10}(2r)$.

Returning now to binary representation, whenever $\text{fl}_{\text{DP}}(z)$ is a normalized DP number,

$$\text{fl}_{\text{DP}}(z) = z(1 + \mu) \quad \text{where} \quad |\mu| \le \frac{\epsilon_{\text{DP}}}{2}$$

The value $|\mu|$, the absolute relative error in $\text{fl}_{\text{DP}}(z)$, depends on the value of z. Then

$$\begin{aligned} x \oplus y &= (x + y)(1 + \mu_a) \\ x \ominus y &= (x - y)(1 + \mu_s) \\ x \otimes y &= (x \times y)(1 + \mu_m) \\ x \oslash y &= (x/y)(1 + \mu_d) \end{aligned}$$

where the absolute relative errors $|\mu_a|$, $|\mu_s|$, $|\mu_m|$ and $|\mu_d|$ are each no larger than $\frac{\epsilon_{\text{DP}}}{2}$.

Problem 2.2.13. *Show that if A is an approximation to T with an absolute relative error less than $0.5 \cdot 10^{-16}$, then A is a 16–digit accurate approximation to T.*

Problem 2.2.14. *Let r be an upper bound on the absolute relative error in the approximation A to T, and let q be an integer that satisfies $q \le -\log_{10}(2r)$. Show that A is a q–digit approximation to T. Hint: Show that $|T| \le 10^{e(T)+1}$, $r \le \frac{10^{-q}}{2}$, and $|A - T| \le \frac{10^{-q}|T|}{2} \le \frac{10^{e(T)-q+1}}{2}$. This last inequality demonstrates that A is q–digits accurate.*

Problem 2.2.15. *Let z be a real number for which $fl_{DP}(z)$ is a normalized DP number. Show that $|fl_{DP}(z) - z|$ is at most half a DP ulp times the value of $fl_{DP}(z)$. Then show that $\mu \equiv \dfrac{fl_{DP}(z) - z}{z}$ satisfies the bound $|\mu| \leq \dfrac{\epsilon_{DP}}{2}$.*

Problem 2.2.16. *Let x and y be DP numbers. The absolute relative error in $x \oplus y$ as an approximation to $x + y$ is no larger than $\dfrac{\epsilon_{DP}}{2}$. Show that $x \oplus y$ is about 15 digits accurate as an approximation to $x + y$. How does the accuracy change if you replace the addition operation by any one of subtraction, multiplication, or division (assuming $y \neq 0$)?*

2.2.4 Propagation of Errors

There are two types of errors in any computed sum, difference, product, or quotient. The first is the error that is inherent in the numbers, and the second is the error introduced by the arithmetic.

Let x' and y' be DP numbers and consider computing $x' \times y'$. Commonly, x' and y' are the result of a previous computation. Let x and y be the values that x' and y' would have been if they had been computed exactly; that is,

$$x' = x(1 + \mu_x), \quad y' = y(1 + \mu_y)$$

where μ_x and μ_y are the relative errors in the approximations of x' to x and of y' to y, respectively. Now, $x' \times y'$ is computed as $x' \otimes y'$, so

$$x' \otimes y' = \mathrm{fl}_{\mathrm{DP}}(x' \times y') = (x' \times y')(1 + \mu_m)$$

where μ_m is the relative error in the approximation $x' \otimes y'$ of $x' \times y'$. How well does $x' \otimes y'$ approximate the exact result $x \times y$? We find that

$$x' \otimes y' = (x \times y)(1 + \mu)$$

where $\mu = (1 + \mu_x)(1 + \mu_y)(1 + \mu_m) - 1$. Expanding the products $\mu = \mu_x + \mu_y + \mu_m + \mu_x\mu_y + \mu_x\mu_m + \mu_y\mu_m + \mu_x\mu_y\mu_m$. Generally, we can drop the terms involving products of the μ's because the magnitude of each value μ is small relative to 1, so the magnitudes of products of the μ's are smaller still. Consequently

$$\mu \approx \mu_x + \mu_y + \mu_m$$

So the relative error in $x' \otimes y'$ is (approximately) equal to the sum of

a. the relative errors *inherent* in each of the values x' and y'

b. the relative error *introduced* when the values x' and y' are multiplied

In an extended computation, we expect the error in the final result to come from the (hopefully slow) accumulation of the errors in the initial data and of the errors from the arithmetic operations on that data. This is a major reason why DP arithmetic is mainly used in general scientific computation. While the final result may be represented adequately by a SP number, we use DP arithmetic in an attempt to reduce the *effect* of the accumulation of errors introduced by the computer arithmetic because the relative errors μ in DP arithmetic are so much smaller than in SP arithmetic.

Relatively few DP computations produce exactly a DP number. However, subtraction of nearly equal DP numbers of the same sign is always exact and so is always a DP number. This **exact cancelation** result is stated mathematically as follows:

$$x' \ominus y' = x' - y'$$

whenever $\dfrac{1}{2} \leq \dfrac{x'}{y'} \leq 2$. So, $\mu_s = 0$ and we expect that $\mu \approx \mu_x + \mu_y$. It is easy to see that

$$x' \ominus y' = x' - y' = (x - y)(1 + \mu)$$

where $\mu = \dfrac{x\mu_x - y\mu_y}{x-y}$. We obtain an upper bound on μ by applying the triangle inequality:

$$\frac{|\mu|}{|\mu_x| + |\mu_y|} \leq g \equiv \frac{|x| + |y|}{|x-y|}$$

The left hand side measures how the relative errors μ_x and μ_y in the values x' and y', respectively, are *magnified* to produce the relative error μ in the value $x' - y'$. The right hand side, g, is an upper bound on this magnification. Observe that $g \geq 1$ and that g grows as $|x-y|$ gets smaller, that is as more cancelation occurs in computing $x - y$. When g is large, the relative error in $x' - y'$ *may be large* because **catastrophic cancelation** has occurred. In the next section we present examples where computed results suffer from the effect of catastrophic cancelation.

Example 2.2.1. We'll give a simple example to simulate how the floating–point addition works on a computer. To simplify, we'll work in three decimal digit floating–point arithmetic, and assume we want to compute an approximation of $1/3 + 8/7$.

- Since neither $x = 1/3$ nor $y = 8/7$ are representable, we must first round to three decimal digits:

  $$\begin{aligned} \text{fl}\,(x) &= \text{fl}\,(1/3) = 3.33 * 10^{-1} \\ \text{fl}\,(y) &= \text{fl}\,(8/7) = 1.14 * 10^{0} \end{aligned}$$

  Notice the errors in just representing the numbers on our computer:

  $$\begin{aligned} \text{fl}\,(x) = x(1+\mu_x) \quad &\Rightarrow \text{fl}\,(1/3) = (1/3)(1-0.001) \quad \Rightarrow |\mu_x| = 0.001 \\ \text{fl}\,(y) = y(1+\mu_y) \quad &\Rightarrow \text{fl}\,(8/7) = (8/7)(1-0.0025) \quad \Rightarrow |\mu_y| = 0.0025 \end{aligned}$$

- Now compute $x \oplus y$:

  $$x \oplus y = \text{fl}\left(3.33 * 10^{-1} + 1.14 * 10^{0}\right) = \text{fl}\left(1.473 * 10^{0}\right) = 1.473 * 10^{0}$$

  Notice that since we simulating 3-digit decimal arithmetic, we need to round after each result.

And, $x \oplus y = (x+y)(1-0.0042)$, and we observe the effects of propagation of rounding error, since $|\mu_a| = 0.0042 > |\mu_x| + |\mu_y| = 0.0035$.

Problem 2.2.17. *Using the previous example, investigate the propagation of errors of $x \oplus y$ and $x \ominus y$ for $x = 1/11$ and $y = 4/3$. Assume 3-digit decimal floating point arithmetic.*

Problem 2.2.18. *Let $x = 1/3$, $y = 8/7$ and $z = 1/13$ and use 3-digit decimal floating point arithmetic to compute*

$$(x \oplus y) \oplus z \quad \text{and} \quad x \oplus (y \oplus z)$$

You should get different results, which illustrates that floating point arithmetic is not associative.

Problem 2.2.19. *Derive the expression*

$$x' \ominus y' = x' - y' = (x-y)(1+\mu)$$

where $\mu \equiv \dfrac{x\mu_x - y\mu_y}{x-y}$. Show that

$$|x\mu_x - y\mu_y| \leq |x||\mu_x| + |y||\mu_y| \leq (|x| + |y|)(|\mu_x| + |\mu_y|)$$

and that

$$\left| \frac{x\mu_x - y\mu_y}{x-y} \right| \leq (|\mu_x| + |\mu_y|)\frac{|x| + |y|}{|x-y|}$$

2.3 Examples

The following examples illustrate some less obvious pitfalls in simple scientific computations.

2.3.1 Plotting a Polynomial

Consider the polynomial $p(x) = (1-x)^{10}$, which can be written in power series form as

$$p(x) = x^{10} - 10\,x^9 + 45\,x^8 - 120\,x^7 + 210\,x^6 - 252\,x^5 + 210\,x^4 - 120\,x^3 + 45\,x^2 - 10\,x + 1$$

Suppose we use this power series to evaluate and plot $p(x)$. For example, if we use DP arithmetic to evaluate $p(x)$ at 101 equally spaced points in the interval $[0, 2]$ (so that the spacing between points is 0.02), and plot the resulting values, we obtain the curve shown on the left in Fig. 2.1. The curve has a shape we might expect. However, if we attempt to zoom in on the region near $x = 1$ by using DP arithmetic to evaluating $p(x)$ at 101 equally spaced points in the interval $[0.99, 1.01]$ (so that the spacing is 0.0002), and plot the resulting values, then we obtain the curve shown on the right in Fig. 2.1. In this case, the plot suggests that $p(x)$ has many zeros in the interval $[0.99, 1.01]$. (Remember, if $p(x)$ changes sign at two points then, by continuity, $p(x)$ has a zero somewhere between those points.) However, the factored form $p(x) = (1-x)^{10}$ implies there is only a single zero, of multiplicity 10, at the point $x = 1$. Roundoff error incurred while evaluating the power series and the effects of cancelation of the rounded values induce this inconsistency.

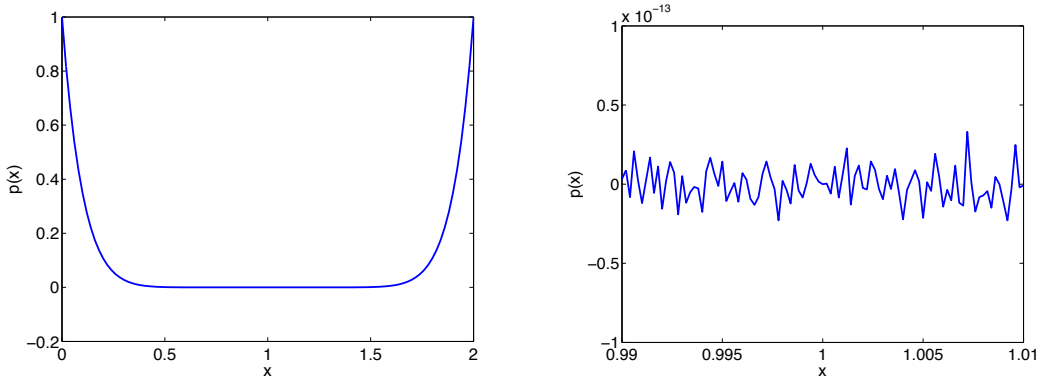

Figure 2.1: Plot of the power form $p(x) = x^{10} - 10\,x^9 + 45\,x^8 - 120\,x^7 + 210\,x^6 - 252\,x^5 + 210\,x^4 - 120\,x^3 + 45\,x^2 - 10\,x + 1$ evaluated in DP arithmetic. The left shows a plot of $p(x)$ using 101 equally spaced points on the interval $0 \le x \le 2$. The right zooms in on the region near $x = 1$ by plotting $p(x)$ using 101 equally spaced points on the interval $0.99 \le x \le 1.01$.

In Fig. 2.1, the maximum amplitudes of the oscillations are larger to the right of $x = 1$ than to the left. Recall that $x = 1$ is the boundary between binade -1 and binade 0. As a result, the magnitude of the maximum error incurred by rounding can be a factor of 2 larger to the right of $x = 1$ than to the left, which is essentially what we observe.

2.3.2 Repeated Square Roots

Let n be a positive integer and let x be a DP number such that $1 \le x < 4$. Consider the following procedure. First, initialize the DP variable $t := x$. Next, take the square root of t a total of n times via the assignment $t := \sqrt{t}$. Then, square the resulting value t a total of n times via the assignment $t := t * t$. Finally, print the value of error, $x - t$. The computed results of an experiment for various values of x, using $n = 100$, are shown in Table 2.1. But for rounding, the value of the error in the third column would be zero, but the results show errors much larger than zero! To understand what is happening, observe that taking the square root of a number discards information. The square

root function maps the interval $[1, 4)$ onto the binade $[1, 2)$. Now $[1, 4)$ is the union of the binades $[1, 2)$ and $[2, 4)$. Furthermore, each binade contains 2^{53} DP numbers. So, the square root function maps each of $2 \cdot 2^{53} = 2^{54}$ arguments into one of 2^{53} possible square roots. On average, then, the square root function maps two DP numbers in $[1, 4)$ into one DP number in $[1, 2)$. So, generally, the DP square root of a DP argument does not contain sufficient information to recover that DP argument, that is taking the DP square root of a DP number usually loses information.

x	t	$x - t$
1.0000	1.0000	0.0000
1.2500	1.0000	0.2500
1.5000	1.0000	0.5000
1.7500	1.0000	0.7500
2.0000	1.0000	1.0000
2.2500	1.0000	1.2500
2.5000	1.0000	1.5000
2.7500	1.0000	1.7500
3.0000	1.0000	2.0000
3.2500	1.0000	2.2500
3.5000	1.0000	2.5000
3.7500	1.0000	2.7500

Table 2.1: Results of the repeated square root experiment. Here x is the exact value, t is the computed result (which in exact arithmetic should be the same as x), and $x - t$ is the error.

2.3.3 Estimating the Derivative

Consider the forward difference estimate

$$\Delta_{\text{E}} f(x) \equiv \frac{f(x+h) - f(x)}{h}$$

of the derivative $f'(x)$ of a continuously differentiable function $f(x)$. Assuming the function $f(x)$ is sufficiently smooth in an interval containing both x and $x + h$, for small values of the increment h, a Taylor series expansion (see Problem 2.3.1) gives the approximation

$$\Delta_{\text{E}} f(x) \approx f'(x) + \frac{h f''(x)}{2}$$

As the increment $h \to 0$ the value of $\Delta_{\text{E}} f(x) \to f'(x)$, a fact familiar from calculus.

Suppose, when computing $\Delta_{\text{E}} f(x)$, that the only errors that occur involve when rounding the exact values of $f(x)$ and $f(x + h)$ to working precision (WP) numbers, that is

$$\text{fl}_{\text{WP}}(f(x)) = f(x)(1 + \mu_1), \quad \text{fl}_{\text{WP}}(f(x+h)) = f(x+h)(1 + \mu_2)$$

where μ_1 and μ_2 account for the errors in the rounding. If $\Delta_{\text{WP}} f(x)$ denotes the computed value of $\Delta_{\text{E}} f(x)$, then

$$\begin{aligned}
\Delta_{\text{WP}} f(x) &\equiv \frac{\text{fl}_{\text{WP}}(f(x+h)) - \text{fl}_{\text{WP}}(f(x))}{h} \\
&= \frac{f(x+h)(1+\mu_2) - f(x)(1+\mu_1)}{h} \\
&= \Delta_{\text{E}} f(x) + \frac{f(x+h)\mu_2 - f(x)\mu_1}{h} \\
&\approx f'(x) + \frac{1}{2} h f''(x) + \frac{f(x+h)\mu_2 - f(x)\mu_1}{h}
\end{aligned}$$

where each absolute relative error $|\mu_i| \leq \frac{\epsilon_{\text{WP}}}{2}$. (For SP arithmetic $\epsilon_{\text{WP}} = 2^{-23}$ and for DP $\epsilon_{\text{WP}} = 2^{-52}$.) Hence, we obtain

$$r = \left|\frac{\Delta_{\text{WP}} f(x) - f'(x)}{f'(x)}\right| \approx h\left|\frac{f''(x)}{2f'(x)}\right| + \frac{1}{h}\left|\frac{f(x+h)\mu_2 - f(x)\mu_1}{f'(x)}\right|$$
$$\approx h\left|\frac{f''(x)}{2f'(x)}\right| + \frac{1}{h}\left|\frac{f(x)(|\mu_2| + |\mu_1|)}{f'(x)}\right| \leq c_1 h + \frac{c_2}{h} \equiv R$$

an approximate upper bound in accepting $\Delta_{\text{WP}} f(x)$ as an approximation of the value $f'(x)$, where $c_1 \equiv \left|\frac{f''(x)}{2f'(x)}\right|$ and $c_2 \equiv \left|\frac{f(x)\epsilon_{\text{WP}}}{f'(x)}\right|$. In deriving this upper bound we have assumed that $f(x+h) \approx f(x)$ which is valid when h is sufficiently small and $f(x)$ is continuous on the interval $[x, x+h]$.

Consider the expression for R. The term $c_1 h \to 0$ as $h \to 0$, accounting for the error in accepting $\Delta_{\text{E}} f(x)$ as an approximation of $f'(x)$. The term $\frac{c_2}{h} \to \infty$ as $h \to 0$, accounting for the error arising from using the computed, rather than the exact, values of $f(x)$ and $f(x+h)$. So, as $h \to 0$, we might expect the absolute relative error in the forward difference estimate $\Delta_{\text{WP}} f(x)$ of the derivative first to decrease and then to increase. Consider using SP arithmetic, the function $f(x) \equiv \sin(x)$ with $x = 1$ radian, and the sequence of increments $h \equiv 2^{-n}$ for $n = 1, 2, 3, \cdots, 22$. Fig. 2.2 uses dots to display $-\log_{10}(2r)$, the digits of accuracy in the computed forward difference estimate of the derivative, and a solid curve to display $-\log_{10}(2R)$, an estimate of the minimum digits of accuracy obtained from the model of rounding. Note, the forward difference estimate becomes more accurate as the curve increases, reaching a maximum accuracy of about 4 digits at $n = 12$, and then becomes less accurate as the curve decreases. The maximum accuracy (that is the minimum value of R) predicted by the model is approximately proportional to the square root of the working-precision.

For h sufficiently small, what we have observed is *catastrophic cancelation* of the values of $f(x)$ and $f(x+h)$ followed by *magnification* of the effect of the resulting loss of significant digits by division by a small number, h.

We will revisit the problem of numerical differentiation in Chapter 5.

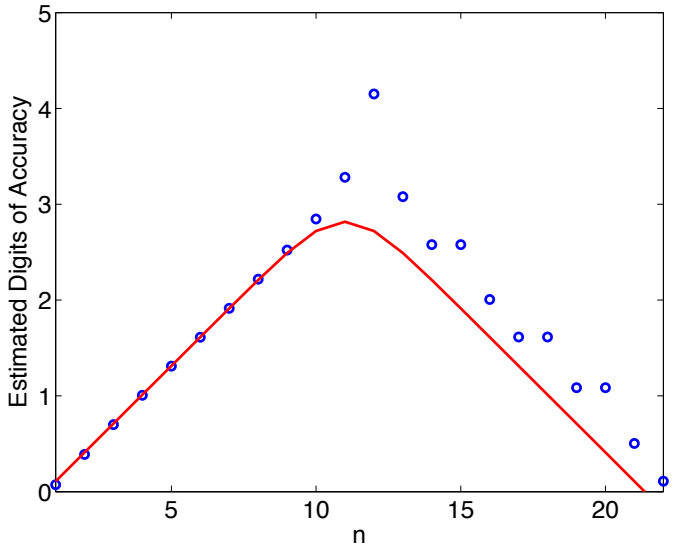

Figure 2.2: Forward difference estimates: $\Delta_{\text{SP}} f(x)$ (dots) and R (solid)

Problem 2.3.1. *A* **Taylor series** *expansion of a function $f(z)$ about the point c is defined to be*

$$f(z) = f(c) + (z-c)f'(c) + (z-c)^2\frac{f''(c)}{2!} + (z-c)^3\frac{f'''(c)}{3!} + \cdots$$

Conditions that guarantee when such a series expansion exists are addressed in a differential calculus course. The function must be sufficiently smooth in the sense that all of the derivatives $f^{(n)}(z)$, $n = 0, 1, 2, \ldots$, must exist in an open interval containing c. In addition, the series must converge, which means the terms of the series must approach zero sufficiently quickly, for all z values in an open interval containing c.
Suppose f is sufficiently smooth in an interval containing $z = x$ and $z = x + h$, where $|h|$ is small. Use the above Taylor series with $z = x + h$ and $c = x$ to show that

$$f(x+h) = f(x) + hf'(x) + h^2\frac{f''(x)}{2!} + h^3\frac{f'''(x)}{3!} + \cdots$$

and argue that

$$\Delta_{\scriptscriptstyle E} f(x) \equiv \frac{f(x+h) - f(x)}{h} \approx f'(x) + \frac{hf''(x)}{2}\,.$$

Problem 2.3.2. *Let the function $R(h) \equiv c_1 h + \dfrac{c_2}{h}$ where c_1, c_2 are positive constants. For what value of h does $R(h)$ attain its smallest value? (Give an expression for this smallest value.) Now, substitute $c_1 \equiv \left|\dfrac{f''(x)}{2f'(x)}\right|$ and $c_2 \equiv \left|\dfrac{f(x)\epsilon_{\scriptscriptstyle WP}}{f'(x)}\right|$ to show how the smallest value of $R(h)$ depends on $\sqrt{\epsilon_{\scriptscriptstyle WP}}$. For $h = 2^{-n}$, what integer n corresponds most closely to this smallest value?*

2.3.4 A Recurrence Relation

Consider the sequence of values $\{V_j\}_{j=0}^{\infty}$ defined by the definite integrals

$$V_j = \int_0^1 e^{x-1}x^j\,dx, \quad j = 0, 1, 2, \cdots$$

It can easily be seen that $0 < V_{j+1} < V_j < \dfrac{1}{j}$ for all $j \geq 1$; see Problem 2.3.4.

Integration by parts demonstrates that the values $\{V_j\}_{j=0}^{\infty}$ satisfy the *recurrence relation*

$$V_j = 1 - jV_{j-1}, \quad j = 1, 2, \cdots$$

Because we can calculate the value of

$$V_0 = \int_0^1 e^{x-1}\,dx = 1 - \frac{1}{e}$$

we can determine the values of $V_1, V_2, \cdots$ using the recurrence starting from the computed estimate for V_0. Table 2.2 displays the values $\hat{V}_j$ for $j = 0, 1, \cdots, 16$ computed by evaluating the recurrence in SP arithmetic.

The first few values $\hat{V}_j$ are positive and form a decreasing sequence. However, the values $\hat{V}_j$ for $j \geq 11$ alternate in sign and increase in magnitude, contradicting the mathematical properties of the sequence. To understand why, suppose that the only rounding error that occurs is in computing V_0. So, instead of the correct initial value V_0 we have used instead the rounded initial value $\hat{V}_0 = V_0 + \epsilon$. Let $\{\hat{V}_j\}_{j=0}^{\infty}$ be the modified sequence determined exactly from the value $\hat{V}_0$. Using the recurrence, the first few terms of this sequence are

$$\begin{array}{lllll}
\hat{V}_1 = 1 - \hat{V}_0 & = & 1 - (V_0 + \epsilon) & = & V_1 - \epsilon \\
\hat{V}_2 = 1 - 2\hat{V}_1 & = & 1 - 2(V_1 - \epsilon) & = & V_2 + 2\epsilon \\
\hat{V}_3 = 1 - 3\hat{V}_2 & = & 1 - 3(V_2 + 2\epsilon) & = & V_3 - 3\cdot 2\epsilon \\
\hat{V}_4 = 1 - 4\hat{V}_3 & = & 1 - 4(V_3 - 3\cdot 2\epsilon) & = & V_4 + 4\cdot 3\cdot 2\epsilon \\
\vdots & & & &
\end{array} \tag{2.5}$$

j	$\hat{V}_j$	$\frac{\hat{V}_j}{j!}$
0	6.3212E-01	6.3212E-01
1	3.6788E-01	3.6788E-01
2	2.6424E-01	1.3212E-01
3	2.0728E-01	3.4546E-02
4	1.7089E-01	7.1205E-03
5	1.4553E-01	1.2128E-03
6	1.2680E-01	1.7611E-04
7	1.1243E-01	2.2307E-05
8	1.0056E-01	2.4941E-06
9	9.4933E-02	2.6161E-07
10	5.0674E-02	1.3965E-08
11	4.4258E-01	1.1088E-08
12	-4.3110E+00	-8.9999E-09
13	5.7043E+01	9.1605E-09
14	-7.9760E+02	-9.1490E-09
15	1.1965E+04	9.1498E-09
16	-1.9144E+05	-9.1498E-09

Table 2.2: Values of V_j determined using SP arithmetic.

In general,

$$\hat{V}_j = V_j + (-1)^j j!\epsilon$$

Now, the computed value $\hat{V}_0$ should have an absolute error ϵ no larger than half a ulp in the SP value of V_0. Because $V_0 \approx 0.632$ is in binade -1, $\epsilon \approx \dfrac{\text{ulp}_{\text{SP}}(V_0)}{2} = 2^{-25} \approx 3 \cdot 10^{-8}$. Substituting this value for ϵ we expect the first few terms of the computed sequence to be positive and decreasing as theory predicts. However, because $j! \cdot (3 \cdot 10^{-8}) > 1$ for all $j \geq 11$ and because $V_j < 1$ for all values of j, the formula for $\hat{V}_j$ leads us to expect that the values $\hat{V}_j$ will ultimately be dominated by $(-1)^j j!\epsilon$ and so will alternate in sign and increase in magnitude. That this analysis gives a reasonable prediction is verified by Table 2.2, where the error grows like $j!$, and $\left|\dfrac{\hat{V}_j}{j!}\right|$ approaches a constant.

What we have observed is the potential for significant *accumulation* and *magnification* of rounding error in a simple process with relatively few steps.

We emphasize that we do not recommend using the recurrence to evaluate the integrals $\hat{V}_j$. They may be evaluated simply either symbolically or numerically using MATLAB integration software; see Chapter 5.

Problem 2.3.3. *Use integration by parts to show the recurrence*

$$V_j = 1 - jV_{j-1}, \quad j = 1, 2, \cdots$$

for evaluating the integral

$$V_j = \int_0^1 e^{x-1} x^j dx, \quad j = 0, 1, 2, \cdots$$

Problem 2.3.4. *In this problem we show that the sequence* $\{V_j\}_{j=0}^{\infty}$ *has positive terms and is strictly decreasing to* 0.

(a) Show that for $0 < x < 1$ *we have* $0 < x^{j+1} < x^j$ *for* $j \geq 0$. *Hence, show that* $0 < V_{j+1} < V_j$.

(b) Show that for $0 < x < 1$ *we have* $0 < e^{x-1} < 1$. *Hence, show that* $0 < V_j < \frac{1}{j}$.

(c) Hence, show that the sequence $\{V_j\}_{j=0}^{\infty}$ *has positive terms and is strictly decreasing to* 0.

Problem 2.3.5. *The error in the terms of the sequence* $\{\hat{V}_j\}_{j=0}^{\infty}$ *grows because* $\hat{V}_{j-1}$ *is multiplied by* j. *To obtain an accurate approximation of* V_j, *we can instead run the recurrence backwards*

$$\hat{V}_j = \frac{1 - \hat{V}_{j+1}}{j+1}, \quad j = M-1, M-2, \cdots, 1, 0$$

Now, the error in $\hat{V}_j$ *is divided by* j. *More specifically, if you want accurate SP approximations of the values* V_j *for* $0 \leq j \leq N$, *start with* $M = N + 12$ *and* $\hat{V}_M = 0$ *and compute the values of* $\hat{V}_j$ *for all* $j = M-1, M-2, \cdots, 1, 0$. *We start* 12 *terms beyond the first value of* $\hat{V}_N$ *so that the error associated with using* $\hat{V}_{N+12} = 0$ *will be divided by at least* $12! \approx 4.8 \cdot 10^8$, *a factor large enough to make the contribution of the initial error in* $\hat{V}_M$ *to the error in* $\hat{V}_N$ *less than a SP ulp in* V_N. *(We know that* $\hat{V}_M$ *is in error – from Problem 2.3.4,* V_M *is a small positive number such that* $0 < V_M < \frac{1}{M}$, *so* $|\hat{V}_M - V_M| = |V_M| < \frac{1}{M}$.*)*

2.3.5 Summing the Exponential Series

From calculus, the Taylor series for the exponential function

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

converges mathematically for all values of x both positive and negative. Let the term

$$T_n \equiv \frac{x^n}{n!}$$

and define the partial sum of the first n terms

$$S_n \equiv 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^{n-1}}{(n-1)!}$$

so that $S_{n+1} = S_n + T_n$ with $S_0 = 0$.

Let's use this series to compute $e^{-20.5} \approx 1.25 \cdot 10^{-9}$. For $x = -20.5$, the terms alternate in sign and their magnitudes increase until we reach the term $T_{20} \approx 7.06 \cdot 10^7$ and then they alternate in sign and their magnitudes steadily decrease to zero. So, for any value $n > 20$, from the theory of alternating series

$$|S_n - e^{-20.5}| < |T_n| .$$

That is, the absolute error in the partial sum S_n as an approximation to the value of $e^{-20.5}$ is less than the absolute value of the first neglected term, $|T_n|$. Note that DP arithmetic is about 16–digits accurate. So, if a value $n \geq 20$ is chosen so that $\frac{|T_n|}{e^{-20.5}} < 10^{-16}$, then S_n should be a 16–digit approximation to $e^{-20.5}$. The smallest value of n for which this inequality is satisfied is $n = 98$, so S_{98} should be a 16–digit accurate approximation to $e^{-20.5}$. Table 2.3 displays a selection of values of the partial sums S_n and the corresponding terms T_n computed using DP arithmetic.

n	T_n	S_{n+1}
0	1.0000e+00	1.0000E+00
5	-3.0171e+04	-2.4057e+04
10	3.6122e+06	2.4009e+06
15	-3.6292e+07	-2.0703e+07
20	7.0624e+07	3.5307e+07
25	-4.0105e+07	-1.7850e+07
30	8.4909e+06	3.4061e+06
35	-7.8914e+05	-2.8817e+05
40	3.6183e+04	1.2126e+04
45	-8.9354e+02	-2.7673e+02
50	1.2724e+01	3.6627e+00
55	-1.1035e-01	-2.9675e-02
60	6.0962e-04	1.5382e-04
65	-2.2267e-06	-5.2412e-07
70	5.5509e-09	6.2893e-09
75	-9.7035e-12	5.0406e-09
80	1.2178e-14	5.0427e-09
85	-1.1201e-17	5.0427e-09
90	7.6897e-21	5.0427e-09
95	-4.0042e-24	5.0427e-09
98	3.7802e-26	5.0427e-09

Table 2.3: Values of S_n and T_n determined using DP arithmetic.

From the table, it appears that the sequence of values S_n is converging. However, $S_{98} \approx 5.04 \cdot 10^{-9}$, and the true value should be $e^{-20.5} \approx 1.25 \cdot 10^{-9}$. Thus, S_{98} is an approximation of $e^{-20.5}$ that is 0–digits accurate! At first glance, it appears that the computation is wrong! However, the computed value of the sum, though very inaccurate, is reasonable. Recall that DP numbers are about 16 digits accurate, so the largest term in magnitude, T_{20}, could be in error by as much as about $|T_{20}| \cdot 10^{-16} \approx 7.06 \cdot 10^7 \cdot 10^{-16} = 7.06 \cdot 10^{-9}$. Of course, changing T_{20} by this amount will change the value of the partial sum S_{98} similarly. So, using DP arithmetic, it is unlikely that the computed value S_{98} will provide an estimate of $e^{-20.5}$ with an absolute error much less than $7.06 \cdot 10^{-9}$.

So, how can we calculate $e^{-20.5}$ accurately using Taylor series? Clearly we must avoid the *catastrophic cancelation* involved in summing an alternating series with large terms when the value of the sum is a much smaller number. There follow two alternative approaches which exploit simple mathematical properties of the exponential function.

1. Consider the relation $e^{-x} = \dfrac{1}{e^x}$. If we use a Taylor series for e^x for $x = 20.5$ it still involves large and small terms T_n but they are all positive and there is no cancellation in evaluating the sum S_n. In fact the terms T_n are the absolute values of those appearing in Table 2.3. In adding this sum we continue until adding further terms can have no impact on the sum. When the terms are smaller then $e^{20.5} \cdot 10^{-16} \approx 8.00 \cdot 10^8 \cdot 10^{-16} = 8.00 \cdot 10^{-8}$ they have no further impact so we stop at the first term smaller than this. Forming this sum and stopping when $|T_n| < 8.00 \cdot 10^{-8}$, it turns out that we need the first 68 terms of the series to give a DP accurate approximation to $e^{20.5}$. Since we are summing a series of positive terms we expect (and get) an approximation for $e^{20.5}$ with a small relative error. Next, we compute $e^{-20.5} = \dfrac{1}{e^{20.5}}$ using this approximation for $e^{20.5}$. The result has at least 15 correct digits in a DP calculation.

2. Another idea is to use a form of *range reduction*. We can use the alternating series as long as we avoid the catastrophic cancellation which leads to a large relative error. We will certainly

avoid this problem if we evaluate e^{-x} only for values of $x \in (0,1)$ since in this case the magnitudes of the terms of the series are monotonically decreasing. Observe that

$$e^{-20.5} = e^{-20}e^{-0.5} = (e^{-1})^{20}e^{-0.5}$$

So, if we can calculate e^{-1} accurately then raise it to the 20^{th} power, then evaluate $e^{-0.5}$ by Taylor series, and finally we can compute $e^{-20.5}$ by multiplying the results. (In the spirit of range reduction, we use, for example, the MATLAB exponential function to evaluate e^{-1} and then its power function to compute its 20^{th} power; this way these quantities are calculated accurately. If we don't have an exponential function available, we could use the series with $x = -1$ to calculate e^{-1}.) Since $e^{-0.5} \approx 0.6$, to get full accuracy we terminate the Taylor series when $|T_n| < 0.6 \cdot 10^{-16}$, that is after 15 terms. This approach delivers at least 15 digits of accuracy for $e^{-20.5}$.

Problem 2.3.6. *Quote and prove the alternating series theorem that shows that $|S_n - e^{-20.5}| < |T_n|$ in exact arithmetic.*

Problem 2.3.7. *Suppose that you use Taylor series to compute e^x for a value of x such that $|x| > 1$. Which is the largest term in the Taylor series? In what circumstances are there two equally sized largest terms in the Taylor series? [Hint: $\dfrac{T_n}{T_{n-1}} = \dfrac{x}{n}$.]*

Problem 2.3.8. *Suppose we are using SP arithmetic (with an accuracy of about 10^{-7}) and say we attempt to compute e^{-15} using the alternating Taylor series. What is the largest value T_n in magnitude. Using the fact that $e^{-15} \approx 3.06 \cdot 10^{-7}$ how many terms of the alternating series are needed to compute e^{-15} to full SP accuracy? What is the approximate absolute error in the sum as an approximation to e^{-15}? [Hint: There is no need to calculate the value of the terms T_n or the partial sums S_n to answer this question.]*

Problem 2.3.9. *For what value of n does the partial sum S_n form a 16–digit accurate approximation of $e^{-21.5}$? Using DP arithmetic, compare the computed value of S_n with the exact value $e^{-21.5} \approx 4.60 \cdot 10^{-10}$. Explain why the computed value of S_n is reasonable. [Hint: Because $\dfrac{T_n}{T_{n-1}} = \dfrac{x}{n}$, if the current value of* `term` *is T_{n-1}, then sequence of assignment statements*

```
term := x*term
term := term/n
```

converts `term` *into the value of T_n.]*

2.3.6 Euclidean Length of a Vector

Many computations require the **Euclidean length**

$$p = \sqrt{a^2 + b^2}$$

of the 2-vector $\begin{bmatrix} a \\ b \end{bmatrix}$. The value p can also be viewed as the **Pythagorean sum** of a and b. Now,

$$\min(|a|, |b|) \le p \le \sqrt{2} \cdot \max(|a|, |b|)$$

so we should be able to compute p in such a way that we never encounter floating–point underflow and we rarely encounter floating–point overflow. However, when computing p via the relation $p = \sqrt{a^2 + b^2}$ we can encounter one or both of *floating–point underflow* and *floating–point overflow* when

computing the squares of a and b. To avoid floating–point overflow, we can choose a value c in such a way that we can scale a and b by this value of c and the resulting scaled quantities $\left[\frac{a}{c}\right]$ and $\left[\frac{b}{c}\right]$ may be safely squared. Using this scaling,

$$p = c \cdot \sqrt{\left[\frac{a}{c}\right]^2 + \left[\frac{b}{c}\right]^2}$$

An obvious choice for the scaling factor is $c = \max(|a|, |b|)$ so that one of $\left[\frac{a}{c}\right]$ and $\left[\frac{b}{c}\right]$ equals 1 and the other has magnitude less than 1. Another sensible choice of scaling factor is c a power of 2 just greater than $\max(|a|, |b|)$; the advantage of this choice is that with *Standard* arithmetic, division by 2 is performed exactly (ignoring underflow). If we choose the scaling factor c in one of these ways it is possible that the smaller squared term in that occurs when computing p will underflow but this is harmless; that is, the computed value of p will be essentially correct.

Another technique that avoids both unnecessary floating–point overflow and floating–point underflow is the Moler-Morrison Pythagorean sum algorithm displayed in the pseudocode in Fig. 2.3. In this algorithm, the value p converges to $\sqrt{a^2 + b^2}$ from below, and the value q converges rapidly to 0 from above. So, floating–point overflow can occur only if the exact value of p overflows. Indeed, only harmless floating–point underflow can occur; that is, the computed value of p will be essentially correct.

Moler-Morrison Pythagorean Sum Algorithm

Input: scalars a and b

Output: $p = \sqrt{a^2 + b^2}$

$p := \max(|a|, |b|)$
$q := \min(|a|, |b|)$
for $i = 1$ to N
  $r := (q/p)^2$
  $s := r/(4 + r)$
  $p := p + 2sp$
  $q := sq$
next i

Figure 2.3: The Moler-Morrison Pythagorean sum algorithm for computing $p \equiv \sqrt{a^2 + b^2}$. Typically $N = 3$ suffices for any SP or DP numbers p and q.

Problem 2.3.10. *For any values a and b show that*

$$\min(|a|, |b|) \leq p \equiv \sqrt{a^2 + b^2} \leq \sqrt{2} \cdot \max(|a|, |b|)$$

Problem 2.3.11. *Suppose $a = 1$ and $b = 10^{-60}$. If $c \approx \max(|a|, |b|)$, floating–point underflow occurs in computing $\left[\frac{b}{c}\right]^2$ in DP arithmetic. In this case, the* Standard *returns the value 0 for $\left[\frac{b}{c}\right]^2$. Why is the computed value of p still accurate?*

Problem 2.3.12. *In the Moler-Morrison Pythagorean sum algorithm, show that though each trip through the for-loop may change the values of p and q, in exact arithmetic it never changes the value of the loop invariant $p^2 + q^2$.*

Problem 2.3.13. *Design a pseudocode to compute $d = \sqrt{x^2 + y^2 + z^2}$. If floating–point underflow occurs it should be harmless, and floating–point overflow should occur only when the exact value of d overflows.*

2.3.7 Roots of a Quadratic Equation

We aim to design a reliable algorithm to determine the roots of the quadratic equation

$$ax^2 - 2bx + c = 0$$

(Note that the factor multiplying x is $-2b$ and not b as in the standard notation.) We assume that the coefficients a, b and c are real numbers, and that the coefficients are chosen so that the roots are real. The familiar quadratic formula for these roots is

$$x_\pm = \frac{b \pm \sqrt{d}}{a}$$

where $d \equiv b^2 - ac$ is the **discriminant**, assumed nonnegative here. The discriminant d is zero if there is a double root. Here are some problems that can arise.

First, the algorithm should check for the special cases $a = 0$, $b = 0$ or $c = 0$. These cases are trivial to eliminate. When $a = 0$ the quadratic equation degenerates to a linear equation whose solution $\frac{c}{2b}$ requires at most a division. When $b = 0$ the roots are $\pm\sqrt{-\frac{c}{a}}$. When $c = 0$ the roots are 0 and $\frac{2b}{a}$. In the latter two cases the roots will normally be calculated more accurately using these special formulas than by using the quadratic formula.

Second, computing d can lead to either *floating–point underflow* or *floating–point overflow*, typically when either b^2 or ac or both are computed. This problem can be eliminated using the technique described in the previous section; scale the coefficients a, b and c by a power of two chosen so that b^2 and ac can be safely computed.

Third, the computation can suffer from *catastrophic cancelation* when either

- the roots are nearly equal, i.e., $ac \approx b^2$
- the roots have significantly different magnitudes, i.e., $|ac| \ll b^2$

When $ac \approx b^2$ there is catastrophic cancelation when computing d. This may be eliminated by computing the discriminant $d = b^2 - ac$ using higher precision arithmetic, when possible. For example, if a, b and c are SP numbers, then we can use DP arithmetic to compute d. If a, b and c are DP numbers, maybe we can use QP (quadruple, extended, precision) arithmetic to compute d; MATLAB does not provide QP. The aim when using higher precision arithmetic is to discard as few digits as possible of b^2 and ac before forming $b^2 - ac$. For many arithmetic processors this is achieved without user intervention. That is, the discriminant is calculated to higher precision automatically, by computing the value of the whole expression $b^2 - ac$ in higher precision before rounding. When cancelation is inevitable, we might first use the quadratic formula to compute approximations to the roots and then using a Newton iteration (see Chapter 6) to improve these approximations.

When $|ac| \ll b^2$, $\sqrt{d} \approx |b|$ and one of the computations $b \pm \sqrt{d}$ suffers from catastrophic cancelation. To eliminate this problem note that

$$\frac{b + \sqrt{d}}{a} = \frac{c}{b - \sqrt{d}}, \quad \frac{b - \sqrt{d}}{a} = \frac{c}{b + \sqrt{d}}$$

So, when $b > 0$ use

$$x_+ = \frac{b + \sqrt{d}}{a}, \quad x_- = \frac{c}{b + \sqrt{d}}$$

and when $b < 0$ use

$$x_+ = \frac{c}{b - \sqrt{d}}, \quad x_- = \frac{b - \sqrt{d}}{a}$$

The following problems are intended as a "pencil and paper" exercises.

Problem 2.3.14. *Show that $a = 2049$, $b = 4097$ and $c = 8192$ are SP numbers. Use* MATLAB *SP arithmetic to compute the roots of $ax^2 - 2bx + c = 0$ using the quadratic formula.*

Problem 2.3.15. *Show that $a = 1$, $b = 4096$ and $c = 1$ are SP numbers. Use* MATLAB *SP arithmetic to compute the roots of $ax^2 - 2bx + c = 0$ using the quadratic formula.*

2.4 Matlab Notes

MATLAB provides a variety of data types, including integers and floating–point numbers. By default, all floating–point variables and constants, and the associated computations, are held in double precision. Indeed, they are all complex numbers but this is normally not visible to the user when using real data. However, there are cases where real data can lead to complex number results, for example the roots of a polynomial with real coefficients can be complex and this is handled seamlessly by MATLAB. The most recent versions of MATLAB (version 7.0 and higher) allow for the creation of single precision variables and constants. MATLAB also defines certain parameters (such as machine epsilon) associated with floating–point computations.

In addition to discussing these issues, we provide implementation details and several exercises associated with the examples discussed in Section 2.3.

2.4.1 Integers in Matlab

MATLAB supports both signed and unsigned integers. For each type, integers of 8, 16, 32 and 64 bits are supported. MATLAB provides functions to convert a numeric object (e.g. an integer of another type or length, or a double precision number) into an integer of the specified type and length; e.g,. `int8` converts numeric objects into 8–bit signed integers and `uint16` converts numeric objects into 16–bit unsigned integers. MATLAB performs arithmetic between integers of the same type and between integers of any type and double precision numbers.

2.4.2 Single Precision Computations in Matlab

By default, MATLAB assumes all floating–point variables and constants, and the associated computations, are double precision. To compute using single precision arithmetic, variables and constants must first be converted using the `single` function. Computations involving a mix of SP and DP variables generally produce SP results. For example,

```
theta1 = 5*single(pi)/6
s1 = sin(theta1)
```

produces the SP values `theta1`= 2.6179941 and `s1`= 0.4999998. Because we specify `single(pi)`, the constants 5 and 6 in `theta1` are assumed SP, and the computations use SP arithmetic.

As a comparison, if we do not specify `single` for any of the variables or constants,

```
theta2 = 5*pi/6
s2 = sin(theta2)
```

then MATLAB produces the DP values `theta2`= 2.617993877991494 and `s2`= 0.50000000000000.

However, if the computations are written

```
theta3 = single(5*pi/6)
s3 = sin(theta3)
```

then MATLAB produces the values `theta3`= 2.6179938, and `s3`= 0.5000001. The computation `5*pi/6` uses default DP arithmetic, then the result is converted to SP.

2.4.3 Special Constants

A nice feature of MATLAB is that many parameters discussed in this chapter can be generated easily. For example, `eps` is a function that can be used to compute ϵ_{DP} and $\text{ulp}_{\text{DP}}(y)$. In particular, `eps(1)`, `eps('double')` and `eps` all produce the same result, namely 2^{-52}. However, if `y` is a DP number, then `eps(y)` computes $\text{ulp}_{\text{DP}}(y)$, which is simply the distance from y to the next largest (in magnitude) DP number. The largest and smallest positive DP numbers can be computed using the functions `realmax` and `realmin`, respectively.

As with DP numbers, the functions `eps`, `realmax` and `realmin` can be used with SP numbers. For example, `eps('single')` and `eps(single(1))` produce the result 2^{-23}. Similarly, `realmax('single')` and `realmin('single')` return, respectively, the largest and smallest SP floating–point numbers.

MATLAB also defines parameters for ∞ (called `inf` or `Inf`) and NaN (called `nan` or `NaN`). It is possible to get, and to use, these quantities in computations. For example:

- The computation `1/0` produces `Inf`.
- The computation `1/Inf` produces 0.
- Computations of indeterminate quantities, such as `0*Inf`, `0/0` and `Inf/Inf` produce `NaN`.

Problem 2.4.1. *In* MATLAB*, consider the anonymous functions:*

```
f = @(x) x ./ (x.*(x-1));
g = @(x) 1 ./ (x-1);
```

What is computed by `f(0)`*,* `f(eps)`*,* `g(0)`*,* `g(eps)`*,* `f(Inf)`*, and* `g(Inf)`*? Explain these results.*

2.4.4 Floating-Point Numbers in Output

When displaying output, MATLAB rounds floating–point numbers to fit the number of digits to be displayed. For example, consider a MATLAB code that sets a DP variable x to the value 1.99 and prints it twice, first using a floating–point format with 1 place after the decimal point and second using a floating–point format with 2 places after the decimal point. MATLAB code for this is:

```
x = 1.99;
fprintf('Printing one decimal point produces %3.1f \n', x)
fprintf('Printing two decimal points produces %4.2f \n', x)
```

The first value printed is 2.0 while the second value printed is 1.99. Of course, 2.0 is not the true value of x. The number 2.0 appears because 2.0 represents the true value rounded to the number of digits displayed. If a printout lists the value of a variable x as precisely 2.0, that is it prints just these digits, then its actual value may be any number in the range $1.95 \leq x < 2.05$.

A similar situation occurs in the MATLAB command window. When MATLAB is started, numbers are displayed on the screen using the default "format" (called `short`) of 5 digits. For example, if we set a DP variable $x = 1.99999$, MATLAB displays the number as

```
2.0000
```

More correct digits can be displayed by changing the format. So, if we execute the MATLAB statement

```
format long
```

then 15 digits are displayed; that is, the number x is displayed as

```
1.99999000000000
```

Other formats can be set; see `doc format` for more information.

Problem 2.4.2. *Using the default* `format short`*, what is displayed in the command window when the following variables are displayed?*

```
x = exp(1)
y = single(exp(1))
z = x - y
```

Do the results make sense? What is displayed when using `format long`*?*

Problem 2.4.3. *Read* MATLAB*'s help documentation on* `fprintf`*. In the example* MATLAB *code given above, why did the first* `fprintf` *command contain* `\n`*? What happens if this is omitted?*

Problem 2.4.4. *Determine the difference between the following* `fprintf` *statements:*

```
fprintf('%6.4f \n',pi)
fprintf('%8.4f \n',pi)
fprintf('%10.4f \n',pi)
fprintf('%10.4e \n',pi)
```

In particular, what is the significance of the numbers `6.4`*,* `8.4`*, and* `10.4`*, and the letters* `f` *and* `e`*?*

2.4.5 Examples

Section 2.3 provided several examples that illustrate difficulties that can arise in scientific computations. Here we provide MATLAB implementation details and several associated exercises.

Plotting a Polynomial

Consider the example from Section 2.3.1, where a plot of

$$\begin{aligned} p(x) &= (1-x)^{10} \\ &= x^{10} - 10\,x^9 + 45\,x^8 - 120\,x^7 + 210\,x^6 - 252\,x^5 + 210\,x^4 - 120\,x^3 + 45\,x^2 - 10\,x + 1 \end{aligned}$$

on the interval $[0.99, 1.01]$ is produced using the power series form of $p(x)$. In MATLAB we use `linspace`, `plot`, and a very useful function called `polyval`, which is used to evaluate polynomials. Specifically, the following MATLAB code is used to produce the plot shown in Fig. 2.1:

```
x = linspace(0.99, 1.01, 101);
c = [1, -10, 45, -120, 210, -252, 210, -120, 45, -10, 1];
p = polyval(c, x);
plot(x, p)
xlabel('x'), ylabel('p(x)')
```

Of course, since we know the factored form of $p(x)$, we can use it to produce an accurate plot:

```
x = linspace(0.99, 1.01, 101);
p = (1 - x).^(10);
plot(x, p)
```

Not all polynomials can be factored easily, and it may be necessary to use the power series form.

Problem 2.4.5. *Use the code given above to sketch $y = p(x)$ for values $x \in [0.99, 1.01]$ using the power form of $p(x)$. Pay particular attention to the scaling of the y-axis. What is the largest value of y that you observe? Now modify the code to plot on the same graph the factored form of $y = p(x)$ and to put axes on the graph. Can you distinguish the plot of the factored form of $y = p(x)$? Explain what you observe.*

Problem 2.4.6. *Construct a figure analogous to Fig. 2.1, but using SP arithmetic rather than DP arithmetic to evaluate $p(x)$. What is the largest value of y that you observe in this case?*

Repeated Square Roots

The following MATLAB code performs the repeated square roots experiment outlined in Section 2.3.2 on a vector of 10 equally spaced values of x using 100 iterations (use a script M–file):

```
x = 1:0.25:3.75;
n = 100
t = x;
for i = 1:n
  t = sqrt(t);
end
for i = 1:n
  t = t .* t;
end
disp('   x         t        x-t ')
disp('==============================')
for k = 1:10
  disp(sprintf('%7.4f   %7.4f   %7.4f', x(k), t(k), x(k)-t(k)))
end
```

When you run this code, the `disp` and `sprintf` commands are used to display, in the command window, the results shown in Table 2.1. The command `sprintf` works just like `fprintf` but prints the result to a MATLAB string (displayed using `disp`) rather than to a file or the screen.

Problem 2.4.7. *Using the MATLAB script M–file above, what is the smallest number of iterations n for which you can reproduce the whole of Table 2.1 exactly?*

Problem 2.4.8. *Modify the MATLAB script M–file above to use SP arithmetic. What is the smallest number of iterations n for which you can reproduce the whole of Table 2.1 exactly?*

Estimating the Derivative

The following problems use MATLAB for experiments related to the methods of Section 2.3.3

Problem 2.4.9. *Use MATLAB (with its default DP arithmetic), $f(x) \equiv \sin(x)$ and $x = 1$ radian. Create a three column table with column headers "n", "$-\log_{10}(2r)$", and "$-\log_{10}(2R)$". Fill the column headed by "n" with the values $1, 2, \cdots, 51$. The remaining entries in each row should be filled with values computed using $h = 2^{-n}$. For what value of n does the forward difference estimate $\Delta_{DP} f(x)$ of the derivative $f'(x)$ achieve its maximum accuracy, and what is this maximum accuracy?*

Problem 2.4.10. *In* MATLAB*, open the help browser, and search for* `single precision mathematics`*. This search can be used to find an example of writing M–files for different data types. Use this example to modify the code from Problem 2.4.9 so that it can be used for either SP or DP arithmetic.*

A Recurrence Relation

The following problems use MATLAB for experiments related to the methods of Section 2.3.4 As mentioned at the end of Section 2.3.4, we emphasize that we do not recommend the use of the recurrence for evaluating the integrals $\hat{V}_j$. These integrals may be almost trivially evaluated using the MATLAB function `integral`; see Chapter 5 for more details.

Problem 2.4.11. *Reproduce the Table 2.2 using* MATLAB *single precision arithmetic.*

Problem 2.4.12. *Repeat the above analysis, but use* MATLAB *with its default DP arithmetic, to compute the sequence* $\{\hat{V}_j\}_{j=0}^{23}$*. Generate a table analogous to Table 2.2 that displays the values of* j*,* $\hat{V}_j$*, and* $\dfrac{\hat{V}_j}{j!}$*. Hint: Assume that* ϵ*, the error in the initial value* $\hat{V}_0$*, has a magnitude equal to half a ulp in the DP value of* V_0*. Using this estimate, show that* $j!\epsilon > 1$ *when* $j \geq 19$*.*

Problem 2.4.13. *Use the* MATLAB *DP integration function* `integral` *to compute the correct values for* V_j *to the number of digits shown in Table 2.2*

Problem 2.4.14. *Consider the sequence of numbers:*

$$x_{k+2} = 2.25x_{k+1} - 0.5x_k\,, \quad k = 1, 2, \ldots \tag{2.6}$$

with starting values of $x_1 = 1/3$ *and* $x_2 = 1/12$*. It can be shown that this sequence of numbers can also be written as:*

$$x_k = \frac{4^{1-k}}{3}\,, \quad k = 1, 2, \ldots \tag{2.7}$$

Looking at (2.7), what can you say about the terms x_k *as* k *increases?*

(a) Write a MATLAB code that will generate the two sequences of numbers given in (2.6) and (2.7). Your code should be written as a function m-file, with input n *(number of points to generate) and output two vectors containing values* $x_1, x_2, \ldots, x_n$ *for each of (2.6) and (2.7).*

(b) Run the code using $n = 60$*, and plot the resulting* x_k *values using MATLAB's* `semilogy` *function. Plot both sets of points on the same axes, using different symbols and colors (e.g., blue circles for (2.6) and red diamonds for (2.7)).*

(c) Do the points generated from your code behave as you expect, considering what you found in part (a)? Can you explain your results?

Summing the Exponential Series

The following problems use MATLAB for experiments related to the methods of Section 2.3.5

Problem 2.4.15. *Implement the scheme described in Section 2.3.5, Note 1, in* MATLAB *DP arithmetic to compute* $e^{-20.5}$*.*

Problem 2.4.16. *Implement the scheme described in Section 2.3.5, Note 2, in* MATLAB *DP arithmetic to compute* $e^{-20.5}$*.*

Euclidean Length of a Vector

The following problem uses MATLAB for experiments related to the methods of Section 2.3.6.

Problem 2.4.17. *Write a* MATLAB *DP program that uses the Moler-Morrison algorithm to compute the length of the vector* $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. *Your code should display the values of p and q initially as well as at the end of each trip through the* ***for*** *loop. It should also display the ratio of the new value of q to the cube of its previous value. Compare the value of these ratios to the value of* $\dfrac{1}{4(a^2+b^2)}$.

Problem 2.4.18. *Use DP arithmetic to compute the lengths of the vectors* $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 10^{60} \\ 10^{61} \end{bmatrix}$ *and* $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 10^{-60} \\ 10^{-61} \end{bmatrix}$ *using both the unscaled and scaled formulas for p. Use the factor* $c = \max(|a|, |b|)$ *in the scaled computation.*

Roots of a Quadratic Equation

The following problem uses MATLAB for experiments related to the methods of Section 2.3.7.

Note that MATLAB does not provide software specifically for computing the roots of a quadratic equation. However, it provides a function `roots` for calculating the roots of general polynomials including quadratics; see Section 6.5 for details.

Problem 2.4.19. *Write a* MATLAB *function to compute the roots of the quadratic equation* $ax^2 - 2bx + c = 0$ *where the coefficients a, b and c are SP numbers. As output produce SP values of the roots. Use DP arithmetic to compute* $d = b^2 - ac$. *Test your program on the cases posed in Problems 2.3.14 and 2.3.15*

Problem 2.4.20. *This problem considers two different approaches to compute the roots of a quadratic polynomial,* $p(x) = ax^2 + bx + c$. *We know that the values r such that* $p(r) = 0$, *are given by the quadratic formula:*

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

(a) A naive MATLAB *implementation to compute the roots might look like the function* `QuadFormula1` *given below. On your computer, create the* MATLAB *function m-file* `QuadFormula1.m`, *using this code. You should type in the comment lines as well.*

(b) Test your code using two simple problems:

$$\begin{aligned} p(x) &= x^2 - 3x + 2 \\ p(x) &= 10^{160}x^2 - 3 * 10^{160}x + 2 * 10^{160} \end{aligned}$$

Does this code compute good approximations of the true roots for this problem?

(c) Briefly explain why you did not get good approximations for the roots in one of the polynomials, and explain how you can fix it.

Copy the code in `QuadFormula1.m` *into a new function called* `QuadFormula2.m`, *and modify the code so that it implements your fix. Use* `QuadFormula2.m` *to compute the roots from the above examples and show that your new code obtains accurate approximations for both of these polynomials.*

```
function r = QuadFormula1(coeffs)
%
%  Given coefficients of a quadratic polynomial, p(x), this function
%  computes the roots of p(x) = 0 using the quadratic formula.
%
%  Input: coeffs - vector containing the coefficents of p(x),
%                  [a, b, c], where p(x) = a*x^2 + b*x + c.
%
%  Output:     r - vector containing the two roots of p(x) = 0.
%
%
r = zeros(2,1);
a = coeffs(1);
b = coeffs(2);
c = coeffs(3);
d = sqrt(b^2 - 4*a*c);
a2 = 2*a;
r(1) = (-b + d)/a2;
r(2) = (-b - d)/a2;
```