Chapter 5

Floating Point Arithmetic

Having studied and implemented a few computational algorithms, we now look at the effects of computer arithmetic on these algorithms. In most cases, this effect is minimal and numerical analysts are happy to program their procedures, run the programs on a computer, and trust the answers that it returns, at least to the extent that they trust their models to be correct and their codes to be bug-free. If a code returns clearly incorrect results, one immediately suspects an error in the program or input data or a misunderstanding of the algorithm or physical model, and in the vast majority of cases, this is indeed the source of the problem.

However, there is another possible source of error when algorithms are implemented on a computer. Unless one uses a symbolic system, which quickly becomes impractical for large computations, the arithmetic that is performed is not exact. It is typically rounded to about 16 decimal places. This seems negligible if one is interested in only a few decimal places of the answer, but sometimes rounding errors can accumulate in a disastrous manner. While such occurrences are rare, they can be extremely difficult to diagnose and understand. In order to do so, one must first understand something about the way in which computers perform arithmetic operations.

Most computers store numbers in binary (base 2) format and since there is limited space in a computer word, not all numbers can be represented exactly; they must be rounded to fit the word size. This means that arithmetic operations are not performed exactly. Although the error made in any one operation is usually negligible (of relative size about $10^{-16}$ using double precision), a poorly designed algorithm may magnify this error to the point that it destroys all accuracy in the computed solution. For this reason it is important to understand the effects of rounding errors on computed results.

To see the effects that roundoff can have, consider the following iteration:

$$x_{k+1} = \begin{cases} 2x_k, & x_k \in [0, \frac{1}{2}], \\ 2x_k - 1 & x_k \in \left(\frac{1}{2}, 1 \right]. \end{cases}$$

(5.1)

Let $x_0 = 1/10$. Then $x_1 = 2/10$, $x_2 = 4/10$, $x_3 = 8/10$, $x_4 = 6/10$, and $x_5 = x_1$. The iteration cycles periodically between these values. When implemented on a computer, however, using floating point arithmetic, this is not the case. As seen in Table 5.1, computed results agree with exact results to at least five decimal places until iteration 40, where accumulated error becomes visible in the
fifth decimal digit. As the iterations continue, the error grows until, at iteration 55, the computed result takes on the value 1 and remains there for all subsequent iterations. Later in the chapter, we will see why such errors occur.

<table>
<thead>
<tr>
<th>( k )</th>
<th>True ( x_k )</th>
<th>Computed ( x_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.10000</td>
<td>0.10000</td>
</tr>
<tr>
<td>1</td>
<td>0.20000</td>
<td>0.20000</td>
</tr>
<tr>
<td>2</td>
<td>0.40000</td>
<td>0.40000</td>
</tr>
<tr>
<td>3</td>
<td>0.80000</td>
<td>0.80000</td>
</tr>
<tr>
<td>4</td>
<td>0.60000</td>
<td>0.60000</td>
</tr>
<tr>
<td>5</td>
<td>0.20000</td>
<td>0.20000</td>
</tr>
<tr>
<td>10</td>
<td>0.40000</td>
<td>0.40000</td>
</tr>
<tr>
<td>20</td>
<td>0.60000</td>
<td>0.60000</td>
</tr>
<tr>
<td>40</td>
<td>0.60000</td>
<td>0.600001</td>
</tr>
<tr>
<td>42</td>
<td>0.40000</td>
<td>0.400002</td>
</tr>
<tr>
<td>44</td>
<td>0.60000</td>
<td>0.60010</td>
</tr>
<tr>
<td>50</td>
<td>0.40000</td>
<td>0.40625</td>
</tr>
<tr>
<td>54</td>
<td>0.40000</td>
<td>0.50000</td>
</tr>
<tr>
<td>55</td>
<td>0.80000</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

Table 5.1: Computed results from iteration (5.1). After 55 iterations, the computed value is 1 and it remains there for all subsequent iterations.

We begin this chapter with two stories that demonstrate the potential cost of overlooking the effects of rounding errors.

5.1 Costly Disasters Caused by Rounding Errors

The Intel Pentium Flaw [?, ?, ?]. In the summer of 1994, Intel anticipated the commercial success of its new Pentium chip. The new chip was twice as fast at division as previous Intel chips running at the same clock rate. Concurrently, Professor Thomas R. Nicely, a mathematician at Lynchburg College in Virginia, was computing the sum of the reciprocals of prime numbers using a computer with the new Pentium chip. The computational and theoretical results differed significantly. However, results run on a computer using an older 486 CPU calculated correct results. In time, Nicely tracked the error to the Intel chip. Having contacted Intel and received little response to his initial queries, Nicely posted a general notice on the Internet asking for others to confirm his findings. The posting (dated October 30, 1994) with subject line Bug in the Pentium FPU [?] began:

It appears that there is a bug in the floating point unit (numeric coprocessor) of many, and perhaps all, Pentium processors.
This email began a furor of activity, so much so that only weeks later on December 13, IBM halted shipment of their Pentium machines, and in late December, Intel agreed to replace all flawed Pentium chips upon request. The company put aside a reserve of $420 million to cover costs, a major investment for a flaw. With a flurry of internet activity between November 29 and December 11, Intel had become a laughingstock on the Internet joke circuit, but it wasn’t funny to Intel. On Friday, December 16, Intel stock closed at $59.50, down $3.25 for the week.

What type of error could the chip make in its arithmetic? The New York Times printed the following example of the Pentium bug: Let $A = 4,195,835.0$ and $B = 3,145,727.0$, and consider the quantity

$$A - \frac{A}{B} \cdot B.$$ 

In exact arithmetic, of course, this would be 0, but the Pentium computed 256, because the quotient $A/B$ was accurate to only about 5 decimal places. Is this close enough? For many applications it probably is, but we will see in later sections that we need to be able to count on computers to do better than this.

While such an example can make one wonder how Intel missed such an error, it should be noted that subsequent analysis confirmed the subtlety of the mistake. Alan Edelman, professor of Mathematics at Massachusetts Institute Technology, writes in his article of 1997 published in SIAM Review:?

We also wish to emphasize that, despite the jokes, the bug is far more subtle than many people realize....The bug in the Pentium was an easy mistake to make, and a difficult one to catch.

Ariane 5 Disaster. The Ariane 5, a giant rocket capable of sending a pair of three-ton satellites into orbit with each launch, took 10 years and 7 billion dollars for the European Space Agency to build. Its maiden launch was met with eager anticipation as the rocket was intended to propel Europe far into the lead of the commercial space business.

On June 4, 1996, the unmanned rocket took off cleanly but veered off course and exploded in just under 40 seconds after liftoff. Why? To answer this question, we must step back into the
During the design of an earlier rocket, programmers decided to implement an additional “feature” that would leave the horizontal positioning function (designed for positioning the rocket on the ground) running after the countdown had started, anticipating the possibility of a delayed take-off. Since the expected deviation while on the ground was minimal, only a small amount of memory (16 bits) was allocated to the storage of this information. After the launch, however, the horizontal deviation was large enough that the number could not be stored correctly with 16-bits, resulting in an exception error. This error instructed the primary unit to shut down. Then, all functions were transferred to a backup unit, created for redundancy in case the primary unit shut down. Unfortunately, the backup system contained the same bug and shut itself down. Suddenly, the rocket was veering off course causing damage between the solid rocket boosters and the main body of the rocket. Detecting the mechanical failure, the master control systems triggered a self-destruct cycle, as had been programmed in the event of serious mechanical failure in flight. Suddenly, the rocket and its expensive payloads were scattered over about 12 square kilometers east of the launch pad. Millions of dollars would have been saved if the data had simply been saved in a larger variable rather than the 16-bit memory location allocated in the program.

As seen from these examples, it is important for numerical analysts to understand the impact of rounding errors on their calculations. This chapter covers the basics of computer arithmetic and the IEEE standard. Later we will see more about how to apply this knowledge to the analysis of algorithms. For an excellent and very readable book on computer arithmetic and the IEEE standard, see [?].
5.2 Binary Representation and Base 2 Arithmetic

Most computers today use binary or base 2 arithmetic. This is natural since on/off gates can represent a 1 (on) or a 0 (off), and these are the only two digits in base 2. In base 10, a natural number is represented by a sequence of digits from 0 to 9, with the right-most digit representing 1’s (or 10^0’s), the next representing 10’s (or 10^1’s), the next representing 100’s (or 10^2’s), etc. In base 2, the digits are 0 and 1, and the right-most digit represents 1’s (or 2^0’s), the next represents 2’s (or 2^1’s), the next 4’s (or 2^2’s), etc. Thus, for example, the decimal number 10 is written in base 2 as 1010: one 2^3 = 8, zero 2^2, one 2^1 = 2, and zero 2^0. The decimal number 27 is 11011_2: one 2^4 = 16, one 2^3 = 8, zero 2^2, one 2^1 = 2, and one 2^0 = 1. The binary representation of a positive integer is determined by first finding the highest power of 2 that is less than or equal to the number; in the case of 27, this is 2^4, so a 1 goes in the fifth position from the right of the number: 1_____. One then subtracts 2^4 from 27 to find that the remainder is 11. Since 2^3 is less than 11, a 1 goes in the next position to the right: 11___. Subtracting 2^3 from 11 leaves 3, which is less than 2^2, so a 0 goes in the next position: 110__. Since 2^1 is less than 3, a 1 goes in the next position, and since 3 − 2^1 = 1, another 1 goes in the right-most position to give 27 = 11011_2.

Binary arithmetic is carried out in a similar way to decimal arithmetic, except that when adding binary numbers one must remember that 1 + 1 is 10_2. To add the two numbers 10 and 27, we align their binary digits and do the addition as below. The top row shows the digits that are carried from one column to the next.

```
  11  1
  1010
+11011
-----
100101
```

You can check that 100101_2 is equal to 37. Subtraction is similar, with borrowing from the next column being necessary when subtracting 1 from 0. Multiplication and division follow similar patterns.

Just as we represent rational numbers using decimal expansions, we can also represent them using binary expansions. The digits to the right of the decimal point in base 10 represent 10\(^{-1}\)'s (tenths), 10\(^{-2}\)'s (hundredths), etc., while those to the right of the binary point in base 2 represent 2\(^{-1}\)'s (halves) 2\(^{-2}\)'s (fourths), etc. For example, the fraction 11/2 is 5.5 in base 10, while it is 10.1_2 in base 2: one 2^2, one 2^0, and one 2\(^{-1}\). Not all rational numbers can be represented with finite decimal expansions. The number 1/3, for example, is .33\(\overline{3}\), with the bar over the 3 meaning that this digit is repeated infinitely many times. The same is true for binary expansions, although the numbers that require an infinite binary expansion may be different from the ones that require an infinite decimal expansion. For example, the number 1/10 = 0.1 in base 10 has the repeating binary expansion: 0.00011001100110011001100110011001100110011001100110011001100110011... to base 10 long division:
Irrational numbers such as \( \pi \approx 3.141592654 \) can only be approximated by decimal expansions, and the same holds for binary expansions.

### 5.3 Floating Point Representation

A computer word consists of a certain number of bits, which can be either on (to represent 1) or off (to represent 0). Some early computers used fixed point representation, where one bit is used to denote the sign of a number, a certain number of the remaining bits are used to store the part of the binary number to the left of the binary point, and the remaining bits are used to store the part to the right of the binary point. The difficulty with this system is that it can store numbers only in a very limited range. If, say, 16 bits are used to store the part of the number to the left of the binary point, then the left-most bit represents \( 2^{15} \), and numbers greater than or equal to \( 2^{16} \) cannot be stored. Similarly, if, say, 15 bits are used to store the part of the number to the right of the binary point, then the right-most bit represents \( 2^{-15} \) and no positive number smaller than \( 2^{-15} \) can be stored.

A more flexible system is floating point representation, which is based on scientific notation. Here a number is written in the form \( \pm m \times 2^E \), where \( 1 \leq m < 2 \). Thus, the number \( 10 = 1010_2 \) would be written as \( 1.010 \times 2^3 \), while \( \frac{1}{16} = 0.00010002 \) would be written as \( 1.1001000 \times 2^{-4} \). The computer word consists of three fields: one for the sign, one for the exponent \( E \), and one for the significand \( m \). A single precision word consists of 32 bits: 1 bit for the sign (0 for +, 1 for –), 8 bits for the exponent, and 23 bits for the significand. Thus the number \( 10 = 1.010 \times 2^3 \) would be stored in the form

\[
0 \quad E=3 \quad 1.010\ldots0
\]

while \( 5.5 = 1.011 \times 2^2 \) would be stored in the form

\[
0 \quad E=2 \quad 1.011\ldots0
\]

[We will explain later how the exponent is actually stored.] A number that can be stored exactly using this scheme is called a floating point number. The number \( 1/10 = 1.1001000_2 \times 2^{-4} \) is not a floating point number since it must be rounded in order to be stored.

Before describing rounding, let us consider one slight improvement on this basic storage scheme. Recall that when we write a binary number in the form \( m \times 2^E \), where \( 1 \leq m < 2 \), the first bit to
the left of the binary point is always 1. Hence there is no need to store it. If the significand is of the form \( b_0.b_1 \ldots b_{23} \), then instead of storing \( b_0, b_1, \ldots, b_{22} \), we can keep one extra place by storing \( b_1, \ldots, b_{23} \), knowing that \( b_0 \) is 1. This is called *hidden bit* representation. Using this scheme, the number \( 10 = 1.0102 \times 2^3 \) is stored as

\[
\begin{array}{c|c}
 0 & E=3 \\
\end{array}
\]

23 bits

The number \( 1/10 = 1.10011001100110011001100 \times 2^{-4} \) is approximated by either

\[
\begin{array}{c|c}
 0 & E=-4 \\
 01011001100110011001100 & \end{array}
\]

or

\[
\begin{array}{c|c}
 0 & E=-4 \\
 10011001100110011001101 & \end{array}
\]

depending on the rounding mode being used. (See Section 5.5.) There is one difficulty with this scheme, and that is how to represent the number 0. Since 0 cannot be written in the form \( 1.b_1b_2 \ldots \times 2^E \), we will need a special way to represent 0.

The gap between 1 and the next larger floating point number is called the *machine precision* and is often denoted by \( \epsilon \). (In MATLAB, this number is called \texttt{eps}.) [Note: Some sources define the machine precision to be \( 1/2 \) times this number.] In single precision, the next floating point number after 1 is \( 1 + 2^{-23} \), which is stored as:

\[
\begin{array}{c|c}
 0 & E=0 \\
 00000000000000000000001 & \end{array}
\]

Thus, for single precision, we have \( \epsilon = 2^{-23} \approx 1.2e-07 \).

The default precision in MATLAB is not single but *double precision*, where a word consists of 64 bits: 1 for the sign, 11 for the exponent, and 52 for the significand. Hence in double precision the next larger floating point number after 1 is \( 1 + 2^{-52} \), so that the machine precision is \( 2^{-52} \approx 2.2e-16 \).

Since single and double precision words contain a large number of bits, it is difficult to get a feel for what numbers can and cannot be represented. It is instructive instead to consider a toy system in which only significands of the form \( 1.b_1b_2 \) can be represented and only exponents 0, 1, and \(-1\) can be stored. This system is described in [1]. What are all the numbers that can be represented in this system? Since \( 1.00_2 = 1 \), we can represent \( 1 \times 2^0 = 1, 1 \times 2^1 = 2, \) and \( 1 \times 2^{-1} = 1/2 \). Since \( 1.01_2 = 5/4 \), we can store the numbers \( 5/4, 5/2, \) and \( 5/8 \). Since \( 1.10_2 = 3/2 \), we can store the numbers \( 3/2, 3, \) and \( 3/4 \). Finally, \( 1.11_2 = 7/4 \) gives us the numbers \( 7/4, 7/2, \) and \( 7/8 \). These numbers are plotted in Figure 5.3.

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
-7/2 & -3 & -5/2 & -2 & -3/2 & -1 & -1/2 & 0 & 1/2 \\
1/2 & 1 & 3/2 & 2 & 5/2 & 3 & 7/2 & \end{array}
\]

Figure 5.3: Numbers represented by a toy system.

There are several things to note about this toy system. First, the machine precision \( \epsilon \) is .25, since the number just right of 1 is \( 5/4 \). Second, in general, the gaps between representable numbers
become larger as we move away from the origin. This is acceptable since the relative gaps, the difference between two consecutive numbers divided by, say, their average, remains of reasonable size. Note, however, that the gap between 0 and the smallest positive number is much larger than the gap between the smallest and next smallest positive number. This is the case with single and double precision floating point numbers as well. The smallest positive (normalized) floating point number in any such system is $1.0 \times 2^{-E}$, where $-E$ is the smallest representable exponent; the next smallest number is $(1 + \epsilon) \times 2^{-E}$, where $\epsilon$ is the machine precision, so the gap between these two numbers is $\epsilon$ times the gap between 0 and the first positive number. This gap can be filled in using subnormal numbers. Subnormal numbers have less precision than normalized floating point numbers and will be described in the next section.

5.4 IEEE Floating Point Arithmetic

In the 1960’s and 1970’s, each computer manufacturer developed its own floating point system, resulting in inconsistent program behavior across machines. Most computers used binary arithmetic, but the IBM 360/70 series used hexadecimal (base 16), and Hewlett-Packard calculators used decimal arithmetic. In the early 1980’s, due largely to the efforts of W. Kahan, computer manufacturers adopted a standard: the IEEE (Institute for Electrical and Electronics Engineers) standard. This standard required:

- Consistent representation of floating point numbers across machines.
- Correctly rounded arithmetic.
- Consistent and sensible treatment of exceptional situations such as divide by 0.

In order to comply with the IEEE standard, computers represent numbers in the way described in the previous section. A special representation is needed for 0 (since it cannot be represented with the standard hidden bit format), and also for $\pm \infty$ (the result of dividing a nonzero number by 0), and also for NaN (Not a Number; e.g. 0/0). This is done with special bits in the exponent field, which slightly reduces the range of possible exponents. Special bits in the exponent field are also used to signal subnormal numbers.

There are 3 standard precisions. As noted previously, a single precision word consists of 32 bits, with 1 bit for the sign, 8 for the exponent, and 23 for the significand. A double precision word consists of 64 bits, with 1 bit for the sign, 11 for the exponent, and 52 for the significand. An extended precision word consists of 80 bits, with 1 bit for the sign, 15 for the exponent, and 64 for the significand. (Note, however, that numbers stored in extended precision do not use hidden bit storage.)

Table 5.2 shows the floating point numbers, subnormal numbers, and exceptional situations that can be represented using IEEE double precision.

It can be seen from the table that the smallest positive normalized floating point number that can be stored is $1.0_2 \times 2^{-1022} \approx 2.2 \times 10^{-308}$, while the largest is $1.1 \ldots 1_2 \times 2^{1023} \approx 1.8 \times 10^{308}$. The exponent field for normalized floating point numbers represents the actual exponent plus 1023, so, using the 11 available exponent bits (and setting aside two possible bit configurations for special
If exponent field is: | Then number is: | Type of number:
--- | --- | ---
000000000000 | ±(0.b₁...b₅₂)₂ × 2⁻¹⁰²² | 0 or subnormal
000000000001 = 1₁₀ | ±(1.b₁...b₅₂)₂ × 2⁻¹⁰²² | Normalized numbers.
000000000010 = 2₁₀ | ±(1.b₁...b₅₂)₂ × 2⁻¹⁰₂₁ | Exponent field is:
: | : | actual exponent + 1023
01111111111 = 1023₁₀ | ±(1.b₁...b₅₂)₂ × 2⁰ | :
: | : | :
11111111110 = 2046₁₀ | ±(1.b₁...b₅₂)₂ × 2¹⁰²₃ | :
11111111111 | ±∞ if b₁ = ... = b₅₂ = 0, NaN otherwise | Exceptions

Table 5.2: IEEE Double Precision

situations) we can represent exponents between −1022 and +1023. The two special exponent field bit patterns are all 0’s and all 1’s. An exponent field consisting of all 0’s signals either 0 or a subnormal number. Note that subnormal numbers have a 0 in front of the binary point instead of a 1 and are always multiplied by 2⁻¹⁰₂₂. Thus the number 1.1₂ × 2⁻¹⁰₂₄ = .011₁₂ × 2⁻¹⁰₂₂ would be represented with an exponent field string of all 0’s and a significand field 0110...0. Subnormal numbers have less precision than normalized floating point numbers since the significand is shifted right, causing fewer of its bits to be stored. The smallest positive subnormal number that can be represented has fifty one 0’s followed by a 1 in its significand field, and its value is 2⁻⁵₂ × 2⁻¹⁰₂₂ = 2⁻¹⁰⁷₄. The number 0 is represented by an exponent field consisting of all 0’s and a significand field of all 0’s. An exponent field consisting of all 1’s signals an exception. If all bits in the significand are 0, then it is ±∞. Otherwise it represents NaN.

**William Kahan** – is an eminent mathematician, numerical analyst and computer scientist who has made important contributions to the study of accurate and efficient methods of solving numerical problems on a computer with finite precision. Among his many contributions, Kahan was the primary architect behind the IEEE 754 standard for floating-point computation. Kahan has received many recognitions for his work, including the Turing Award in 1989 and being named an ACM Fellow in 1994. Kahan was a professor of mathematics and computer science and electrical engineering at the University of California, Berkeley, and he continues his contributions to the ongoing revision of IEEE 754.
5.5 Rounding

There are four rounding modes in the IEEE standard. If \( x \) is a real number that cannot be stored exactly, then it is replaced by a nearby floating point number according to one of the following rules:

- **Round down.** Round(\( x \)) is the largest floating point number that is less than or equal to \( x \).
- **Round up.** Round(\( x \)) is the smallest floating point number that is greater than or equal to \( x \).
- **Round towards 0.** Round(\( x \)) is either round-down(\( x \)) or round-up(\( x \)), whichever lies between 0 and \( x \). Thus if \( x \) is positive then round(\( x \)) = round-down(\( x \)), while if \( x \) is negative then round(\( x \)) = round-up(\( x \)).
- **Round to nearest.** Round(\( x \)) is either round-down(\( x \)) or round-up(\( x \)), whichever is closer. In case of a tie, it is the one whose least significant (rightmost) bit is 0.

The default is round to nearest.

Using double precision, the number \( \frac{1}{10} = 1.10010002 \times 2^{-4} \) is replaced by

\[
\begin{array}{c|cccccccccccc}
0 & 01111111011 & 1001100100110011001100110011001100110011001100110011010
\end{array}
\]

using round down or round towards 0, while it becomes

\[
\begin{array}{c|cccccccccccc}
0 & 01111111011 & 10011001001100110011001100110011001100110011010
\end{array}
\]

using round up or round to nearest. [Note also the exponent field which is the binary representation of 1019, or, 1023 plus the exponent \(-4\).]

The absolute rounding error associated with a number \( x \) is defined as \( |\text{round}(x) - x| \). In double precision, if \( x = \pm (1.b_1 \ldots b_{52} b_{53} \ldots)2^{E} \), where \( E \) is within the range of representable exponents (-1022 to 1023), then the absolute rounding error associated with \( x \) is less than \( 2^{-52} \times 2^{E} \) for any rounding mode; the worst rounding errors occur if, for example, \( b_{53} = b_{54} = \ldots = b_n = 1 \) for some large number \( n \), and round towards 0 is used. For round to nearest, the absolute rounding error is less than or equal to \( 2^{-53} \times 2^{E} \), with the worst case being attained if, say, \( b_{53} = 1 \) and \( b_{54} = \ldots = 0 \); in this case, if \( b_{52} = 0 \), then \( x \) would be replaced by \( 1.b_1 \ldots b_{52} \times 2^{E} \), while if \( b_{52} = 1 \), then \( x \) would be replaced by this number plus \( 2^{-52} \times 2^{E} \). Note that \( 2^{-52} \) is machine \( \epsilon \) for double precision. In single and extended precision we have the analogous result that the absolute rounding error is less than \( \epsilon \times 2^{E} \) for any rounding mode, and less than or equal to \( \frac{\epsilon}{2} \) \times 2^{E} \) for round to nearest.

Usually one is interested not in the absolute rounding error but in the relative rounding error, defined as \( |\text{round}(x) - x|/|x| \). Since we have seen that \( |\text{round}(x) - x| < \epsilon \times 2^{E} \) when \( x \) is of the form \( \pm m \times 2^{E}, \ 1 \leq m < 2 \), it follows that the relative rounding error is less than \( \epsilon \times 2^{E}/(m \times 2^{E}) \leq \epsilon \). For round to nearest, the relative rounding error is less than or equal to \( \frac{\epsilon}{2} \). This means that for any real number \( x \) (in the range of numbers that can be represented by normalized floating point numbers), we can write

\[
\text{round}(x) = x(1 + \delta), \quad \text{where } |\delta| < \epsilon \quad \text{(or } \leq \frac{\epsilon}{2} \text{ for round to nearest).}
\]
The IEEE standard requires that the result of an operation (addition, subtraction, multiplication, or division) on two floating point numbers must be the correctly rounded value of the exact result. For numerical analysts, this is the most important statement in this chapter. It means that if $a$ and $b$ are floating point numbers and $\oplus$, $\ominus$, $\otimes$, and $\oslash$ represent floating point addition, subtraction, multiplication, and division, then we will have

$$
\begin{align*}
    a \oplus b &= \text{round}(a + b) = (a + b)(1 + \delta_1) \\
    a \ominus b &= \text{round}(a - b) = (a - b)(1 + \delta_2) \\
    a \otimes b &= \text{round}(ab) = (ab)(1 + \delta_3) \\
    a \oslash b &= \text{round}(a/b) = (a/b)(1 + \delta_4),
\end{align*}
$$

where $|\delta_i| < \epsilon$ (or $\leq \epsilon/2$ for round to nearest), $i = 1, \ldots, 4$. This is important in the analysis of many algorithms.

**Taking Stock in Vancouver [?, ?, ?]**

On Friday, November 25, 1983, investors and brokers watched as an apparent bear market continued to slowly deplete the Vancouver Stock Exchange (VSE). The following Monday, almost magically, the VSE opened over 550 points higher than it had closed on Friday, even though stock prices were unchanged from Friday’s closing, as no trading had occurred over the weekend. What had transpired between Friday evening and Monday morning was a correction of the index resulting from three weeks of work by consultants from Toronto and California. From the inception of the VSE in January 1982 at a level of 1,000, an error had resulted in a loss of about a point a day or 20 points a month.

Representing 1,500 stocks, the index was recalculated to four decimal places after each recorded transaction and then summarily truncated to only three decimal places. Thus if the index was calculated as 560.9349, it would be truncated to 560.934 instead of rounded to the nearer number 560.935.

With an activity level often at the rate of 3,000 index changes per day, the errors accumulated, until the VSE fell to 520 on November 25, 1983. That next Monday, it was recalculated properly to be 1098.892, thus correcting truncation errors that had been compounding for 22 months.
5.6 Correctly Rounded Floating Point Operations

While the idea of correctly rounded floating point operations sounds quite natural and reasonable (Why would anyone implement *incorrectly* rounded floating point operations?!), it turns out that it is not so easy to accomplish. Here we will give just a flavor of some of the implementation details.

First consider floating point addition and subtraction. Let $a = m \times 2^E$, $1 \leq m < 2$, and $b = p \times 2^F$, $1 \leq p < 2$, be two positive floating point numbers. If $E = F$, then $a + b = (m + p) \times 2^E$. This result may need further normalization if $m + p \geq 2$. Following are two examples, where we retain three digits after the binary point.

1. $1.100_2 \times 2^1 + 1.000_2 \times 2^1 = 10.100_2 \times 2^1 \rightarrow 1.010_2 \times 2^2$,
2. $1.101_2 \times 2^0 + 1.000_2 \times 2^0 = 10.101_2 \times 2^0 \rightarrow 1.010_2 \times 2^1$.

The second example required rounding, and we used round to nearest.

Next suppose that $E \neq F$. In order to add the numbers we must first shift one of the numbers to align the significands, as below:

$1.100_2 \times 2^1 + 1.100_2 \times 2^{-1} = 1.100_2 \times 2^1 + .011_2 \times 2^1 = 1.111_2 \times 2^1$

As another example, consider adding the two single precision numbers $1$ and $1.112 \times 2^{-23}$. This is illustrated below, with the part of the number after the 23rd bit shown on the right of the vertical line:

$1.00000000000000000000000 \times 2^0$
$+.00000000000000000000000 \times 2^0$
$1.00000000000000000000000 \times 2^0$

Using round to nearest, the result becomes $1.0\ldots010_2$.

Now let us consider subtracting the single precision number $1.1\ldots1_2 \times 2^{-24}$ from $1$:

$1.00000000000000000000000 \times 2^0$
$-1.11111111111111111111111 \times 2^0$
$0.00000000000000000000000 \times 2^0$

The result is $1.0_2 \times 2^{-24}$, a perfectly good floating point number, so the IEEE standard requires that we compute this number exactly. In order to do this a *guard bit* is needed to keep track of the 1 to the right of the register after the second number is shifted. Cray computers used to get this wrong because they had no guard bit.

It turns out that correctly rounded arithmetic can be achieved using 2 guard bits and a sticky bit to flag some tricky cases. Following is an example of a tricky case. Suppose we wish to subtract $1.0\ldots01_2 \times 2^{-25}$ from $1$:

$1.00000000000000000000000 \times 2^0$
$-.00000000000000000000000 \times 2^0$
$0.11111111111111111111111 \times 2^0$
Renormalizing and using round to nearest, the result is 1.1\ldots 1_2 \times 2^{-1}. This is the correctly rounded value of the true difference, 1 - 2^{-25} - 2^{-48}. With only 2 guard bits, however, or with any number of guard bits less than 25, the computed result is:

\[
\begin{array}{c|c}
1.00000000000000000000000 & \times 2^0 \\
-0.00000000000000000000000 & 01 \times 2^0 \\
0.11111111111111111111111 & 11 \times 2^0 \\
\end{array}
\]

which, using round to nearest, gives the incorrect answer 1.0_2 \times 2^0. A sticky bit is needed to flag problems of this sort. In practice, floating point operations are often carried out using the 80-bit extended precision registers, in order to minimize the number of special cases that must be flagged.

Floating point multiplication is relatively easy compared to addition and subtraction. The product \((m \times 2^E) \times (p \times 2^F)\) is \((m \times p) \times 2^{E+F}\). This result may need to be renormalized, but no shifting of the factors is required.

### 5.7 Exceptions

Usually when one divides by 0 in a code, it is due to a programming error, but there are occasions when one would actually like to do this and to work with the result as if it were the appropriate mathematical quantity, namely either \(\pm \infty\) if the numerator is nonzero or NaN (not a number) if the numerator is 0. In the past, when a division by 0 occurred, computers would either stop with an error message or continue by setting the result to the largest floating point number. The latter had the unfortunate consequence that two mistakes could cancel each other out: 1/0 - 2/0 = 0. Now 1/0 would be set to \(\infty\), 2/0 would be set to \(\infty\), and the difference, \(\infty - \infty\), would be NaN. The quantities \(\pm \infty\) and NaN obey the standard mathematical rules; for example, \(\infty \times 0 = \text{NaN}\), \(0/0 = \text{NaN}\), \(\infty + a = \infty\) and \(a - \infty = -\infty\) for \(a\) a floating point number.

Over/flow is another type of exceptional situation. This occurs when the true result of an operation is greater than the largest floating point number \((1.1\ldots 1_2 \times 2^{1023} \approx 1.8 \times 10^{308}\) for double precision). How this is handled depends on the rounding mode. Using round up or round to nearest, the result is set to \(\infty\); using round down or round towards 0, it is set to the largest floating point number. Underflow occurs when the true result is less than the smallest floating point number. The result is stored as a subnormal number if it is in the range of the subnormal numbers, and otherwise it is set to 0.
Ch. 5 Exercises

1. Write down the IEEE double precision representation for the following decimal numbers:
   (a) 1.5, using round up.
   (b) 5.1, using round to nearest.
   (c) −5.1, using round towards zero.
   (d) −5.1, using round down.

2. Write down the IEEE double precision representation for the decimal number 50.2, using round to nearest.

3. What is the gap between 2 and the next larger double precision number?

4. What is the gap between 201 and the next larger double precision number?

5. How many different normalized double precision numbers are there? Express your answer using powers of 2.

6. Describe an algorithm to compare two double precision floating point numbers \( a \) and \( b \) to determine whether \( a < b \), \( a = b \), or \( a > b \), by comparing each of their bits from left to right, stopping as soon as a differing bit is encountered.

7. What is the largest decimal number \( x \) that has the following IEEE double precision representation

   \[
   \begin{array}{cccccccccccccccc}
   & 0 & 10000000011 & 111000000000000000000000000000000000000000000000100
   \end{array}
   \]

   using round to nearest. Explain your answer.

8. Consider a very limited system in which significands are only of the form \( 1.b_1b_2b_3 \) and the only exponents are 0, 1, and \(-1\). What is the machine precision \( \epsilon \) for this system? Assuming that subnormal numbers are not used, what is the smallest positive number that can be represented in this system, and what is the largest number that can be represented? Express your answers in decimal form.

9. Consider IEEE double precision floating point arithmetic, using round to nearest. Let \( a, b, \) and \( c \) be normalized double precision floating point numbers, and let \( \oplus, \ominus, \otimes, \) and \( \oslash \) denote correctly rounded floating point addition, subtraction, multiplication, and division.

   (a) Is it necessarily true that \( a \oplus b = b \oplus a \)? Explain why or give an example where this does not hold.

   (b) Is it necessarily true that \( (a \oplus b) \oplus c = a \oplus (b \oplus c) \)? Explain why or give an example where this does not hold.
(c) Determine the maximum possible relative error in the computation \((a \otimes b) \odot c\), assuming that \(c \neq 0\). [You may omit terms of order \(O(\epsilon^2)\) and higher.] Suppose \(c = 0\). What are the possible values that \((a \otimes b) \odot c\) could be assigned?

10. Let \(a\) and \(b\) be two positive floating point numbers with the same exponent. Explain why the computed difference \(a \ominus b\) is always exact using IEEE arithmetic.

11. Explain the behavior seen in example (5.1) at the beginning of this chapter. First note that as long as the exponent in the binary representation of \(x_k\) is less than \(-1\) (so that \(x_k < \frac{1}{2}\)), the new iterate \(x_{k+1}\) is formed just by multiplying \(x_k\) by 2. How is this done using IEEE double precision arithmetic? Are there any rounding errors? Once the exponent of \(x_k\) reaches \(-1\) (assuming that its mantissa has at least one nonzero bit), the new iterate \(x_{k+1}\) is formed by multiplying \(x_k\) by 2 (i.e., increasing the exponent to 0) and then subtracting 1. What does this do to the binary representation of the number when the number is renormalized to have the form \(1.b_1 \ldots b_{52} \times 2^E\)? Based on these observations, can you explain why, starting with any \(x_0 \in (0, 1]\), the computed iterates eventually reach 1 and remain there?

12. The total resistance of an electrical circuit with two resistors connected in parallel, with resistances \(R_1\) and \(R_2\) respectively, is given by the formula

\[
T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}.
\]

If \(R_1 = R_2 = R\), then the total resistance is \(R/2\) since half of the current flows through each resistor. On the other hand, if \(R_2\) is much less than \(R_1\), then most of the current will flow through \(R_2\) but a small amount will still flow through \(R_1\) so the total resistance will be slightly less than \(R_2\). What if \(R_2 = 0\)? Then \(T\) should be 0 since if \(R_1 \neq 0\), then all of the current will flow through \(R_2\), and if \(R_1 = 0\) then there is no resistance anywhere in the circuit. If the above formula is implemented using IEEE arithmetic, will the correct answer be obtained when \(R_2 = 0\)? Explain your answer.

13. In the 7th season episode *Treehouse of Horror VI* of *The Simpsons*, Homer has a nightmare in which the following equation flies past him

\[
1782^{12} + 1841^{12} = 1922^{12}.
\]

(5.2)

Note that this equation, if true, would contradict Fermat’s Last Theorem, which states: For \(n \geq 3\), there do not exist any natural numbers \(x, y, z\) that satisfy the equation \(x^n + y^n = z^n\). Did Homer dream up a counterexample to Fermat’s Last Theorem?
(a) Compute \( \sqrt[12]{1782^{12} + 1841^{12}} \) by typing the following into MATLAB:

```matlab
format short
(1782^{12} + 1841^{12})^{(1/12)}
```

What result does Matlab report? Now look at the answer using format long.

(b) Determine the absolute and relative error in the approximation \( 1782^{12} + 1841^{12} \approx 1922^{12} \). [Such an example is called a Fermat near-miss because of the small relative error. This example was created by The Simpsons writer David S. Cohen with the intent of catching the eye of audience members with a mathematical interest.]

(c) Note that the right-hand side of equation (5.2) is even. Use this to prove that the equation cannot be true.

(d) In a later episode entitled The Wizard of Evergreen Terrace, Homer writes the equation \( 3987^{12} + 4365^{12} = 4472^{12} \). Can you debunk this equation?

14. In the 1999 movie Office Space, a character creates a program that takes fractions of cents that are truncated in a bank’s transactions and deposits them to his own account. This is not a new idea, and hackers who have actually attempted it have been arrested. In this exercise we will simulate the program to determine how long it would take to become a millionaire this way.

Assume that we have access to 50,000 bank accounts. Initially we can take the account balances to be uniformly distributed between, say, $100 and $100,000. The annual interest rate on the accounts is 5%, and interest is compounded daily and added to the accounts, except that fractions of a cent are truncated. These will be deposited to an illegal account that initially has balance $0.

Write a MATLAB program that simulates the Office Space scenario. You can set up the initial accounts with the commands:

```matlab
accounts = 100 + (100000-100)*rand(50000,1); % Sets up 50,000 accounts with balances % between $100 and $100000.
accounts = floor(100*accounts)/100; % Deletes fractions of a cent from % initial balances.
```

(a) Write a MATLAB program that increases the accounts by \( (5/365)\% \) interest each day, truncating each account to the nearest penny and placing the truncated amount into an account, which we will call the illegal account. Assume that the illegal account can hold fractional amounts (that is, do not truncate this account’s values) and let the illegal account also accrue daily interest. Run your code to determine how many days it would take to become a millionaire assuming the illegal account begins with a balance of zero.

(b) Without running your Matlab code, answer the following questions: On average about how much money would you expect to be added to the illegal account each day due to the embezzlement? Suppose you had access to 100,000 accounts, each initially with a
balance of, say $5000. About how much money would be added to the illegal account each day in this case? Explain your answers.

Note that this type of rounding corresponds to fixed point truncation rather than floating point, since only two places are allowed to the right of the decimal point, regardless of how many or few decimal digits appear to the left of the decimal point.

15. In the 1991 Gulf War, the Patriot missile defense system failed due to roundoff error. The troubles stemmed from a computer that performed the tracking calculations with an internal clock whose integer values in tenths of a second were converted to seconds by multiplying by a 24-bit binary approximation to one tenth:

$$0.1_{10} \approx 0.00011001100110011002$$

(a) Convert the binary number in (5.3) to a fraction. Call it $x$.

(b) What is the absolute error in this number; i.e., what is the absolute value of the difference between $x$ and $\frac{1}{10}$?

(c) What is the time error in seconds after 100 hours of operation (i.e., $|360,000 - 3,600,000x|$)?

(d) During the 1991 war, a Scud missile traveled at approximately MACH 5 (3750 miles per hour). Find the distance that a Scud missile would travel during the time error computed in (c).

On February 25, 1991, a Patriot battery system, which was to protect the Dhahran Air Base, had been operating for over 100 consecutive hours. The round-off error caused the system not to track an incoming Scud missile, which slipped through the defense system and detonated on Army barracks, killing 28 American soldiers.
Chapter 6

Conditioning of Problems; Stability of Algorithms

Errors of many sorts are almost always present in scientific computations:

1. Replacing the physical problem by a mathematical model involves approximations.

2. Replacing the mathematical model by a problem that is suitable for numerical solution may involve approximations; e.g., truncating an infinite Taylor series after a finite number of terms.

3. The numerical problem often requires some input data, which may come from measurements or from other sources that are not exact.

4. Once an algorithm is devised for the numerical problem and implemented on a computer, rounding errors will affect the computed result.

It is important to know the effects of such errors on the final result. In this book, we deal mainly with items 2-4, leaving item 1 to the specific application area. In this chapter, we will assume that the mathematical problem has been posed in a way that is amenable to numerical solution and study items 3-4: the effects of errors in the input data on the (exact) solution to the numerical problem, and the effects of errors in computation (e.g., rounding all quantities to 16 decimal places) on the output of an algorithm designed to solve the numerical problem.

There are different ways of measuring error. One can talk about the absolute error in a computed value \( \hat{y} \), which is the absolute value of the difference between \( \hat{y} \) and the true value \( y \): \( |\hat{y} - y| \). One can also measure error in a relative sense, where the quantity of interest is \( |\hat{y} - y|/|y| \). It is usually the relative error that is of interest in applications. In Chapter ??, we will deal with numerical problems and algorithms in which the answer is a vector of values rather than a single number. In this case, we will introduce different norms in which the error can be measured, again in either an absolute or a relative sense.
6.1 Conditioning of Problems

The conditioning of a problem measures how sensitive the answer is to small changes in the input. (Note that this is independent of the algorithm used to compute the answer; it is an intrinsic property of the problem.)

Let \( f \) be a scalar-valued function of a scalar argument \( x \), and suppose that \( \hat{x} \) is close to \( x \). (For example, \( \hat{x} \) might be equal to \( \text{round}(x) \).) How close is \( y = f(x) \) to \( \hat{y} = f(\hat{x}) \)? We can ask this question in an absolute sense: If

\[
|\hat{y} - y| \approx C(x)|\hat{x} - x|,
\]

then \( C(x) \) might be called the absolute condition number of the function \( f \) at the point \( x \). We also can ask the question in a relative sense: If

\[
\left| \frac{\hat{y} - y}{y} \right| \approx \kappa(x) \left| \frac{\hat{x} - x}{x} \right|,
\]

then \( \kappa(x) \) might be called the relative condition number of \( f \) at \( x \).

To determine a possible expression for \( C(x) \), note that

\[
\hat{y} - y = f(\hat{x}) - f(x) = \frac{f(\hat{x}) - f(x)}{\hat{x} - x} \cdot (\hat{x} - x),
\]

and for \( \hat{x} \) very close to \( x \), \( (f(\hat{x}) - f(x))/(\hat{x} - x) \approx f'(x) \). Therefore \( C(x) = |f'(x)| \) is defined to be the absolute condition number. To define the relative condition number \( \kappa(x) \), note that

\[
\frac{\hat{y} - y}{y} = \frac{f(\hat{x}) - f(x)}{\hat{x} - x} \cdot \frac{\hat{x} - x}{x} \cdot \frac{x}{f(x)}.
\]

Again we use the approximation \( (f(\hat{x}) - f(x))/(\hat{x} - x) \approx f'(x) \) to determine

\[
\kappa(x) = \left| \frac{x f'(x)}{f(x)} \right|.
\]

Examples:

- Let \( f(x) = 2x \). Then \( f'(x) = 2 \), so that \( C(x) = 2 \) and \( \kappa(x) = (2x)/(2x) = 1 \). This problem is well-conditioned, in the sense that \( C(x) \) and \( \kappa(x) \) are of moderate size. Exactly what counts as being of “moderate size” depends on the context, but usually one hopes that if \( x \) is changed by an anticipated amount (e.g., \( 10^{-16} \) if the change is due to roundoff in double precision, or perhaps a much larger relative amount if the change is due to measurement error), then the change in \( f \) will be negligible for the application being considered.

- Let \( f(x) = \sqrt{x} \). Then \( f'(x) = \frac{1}{2}x^{-1/2} \), so that \( C(x) = \frac{1}{2}x^{-1/2} \) and \( \kappa(x) = 1/2 \). This problem is well-conditioned in a relative sense. In an absolute sense, it is well-conditioned if \( x \) is not too close to 0, but if, say, \( x = 10^{-16} \), then \( C(x) = 0.5 \times 10^8 \), so a small absolute change in \( x \) (say, changing \( x \) from \( 10^{-16} \) to 0) results in an absolute change in \( \sqrt{x} \) of about \( 10^8 \) times the change in \( x \) (i.e., from \( 10^{-8} \) to 0).
• Let $f(x) = \sin(x)$. Then $f'(x) = \cos(x)$, so that $C(x) = |\cos(x)| \leq 1$ and $\kappa(x) = |x \cot(x)|$.

The relative condition number for this function is large if $x$ is near $\pm \pi$, $\pm 2\pi$, etc., and it also is large if $|x|$ is very large and $|\cot(x)|$ is not extremely small. As $x \to 0$, we find

$$\kappa(x) = \lim_{x \to 0} \left| \frac{x \cos x}{\sin x} \right| = \lim_{x \to 0} \frac{|x - x \sin x|}{\cos x} = 1.$$ 

In Chapter ??? we will consider vector-valued functions of a vector of arguments; e.g., finding the solution to a linear system $Ay = b$. We will need to define norms in order to measure differences in the input vector $\|b - \hat{b}\|/\|b\|$ and in the output vector $\|y - \hat{y}\|/\|y\|$.

### 6.2 Stability of Algorithms

Suppose we have a well-conditioned problem and an algorithm for solving this problem. Will our algorithm give the answer to the expected number of places when implemented in floating point arithmetic? An algorithm that achieves the level of accuracy defined by the conditioning of the problem is called stable, while one that gets unnecessarily inaccurate results due to roundoff is sometimes called unstable. To determine the stability of an algorithm one may do a rounding error analysis.

**Example:** Computing sums: If $x$ and $y$ are two real numbers and they are rounded to floating point numbers and their sum is computed on a machine with unit roundoff $\epsilon$, then

$$\text{fl}(x + y) \equiv \text{round}(x) \oplus \text{round}(y) = (x(1 + \delta_1) + y(1 + \delta_2))(1 + \delta_3), \quad |\delta_i| \leq \epsilon$$

( $\epsilon/2$ for round to nearest),

where fl($\cdot$) denotes the floating point result.

**Forward error analysis:** Here one asks the question: How much does the computed value differ from the exact solution? Multiplying the terms in (6.1), we find

$$\text{fl}(x + y) = x + y + x(\delta_1 + \delta_3 + \delta_1 \delta_3) + y(\delta_2 + \delta_3 + \delta_2 \delta_3).$$

Again one can ask about the absolute error:

$$|\text{fl}(x + y) - (x + y)| \leq (|x| + |y|)(2\epsilon + \epsilon^2),$$

or the relative error:

$$\left| \frac{\text{fl}(x + y) - (x + y)}{x + y} \right| \leq \frac{(|x| + |y|)(2\epsilon + \epsilon^2)}{|x + y|}.$$ 

Note that if $y \approx -x$, then the relative error can be large! The difficulty is due to the initial rounding of the real numbers $x$ and $y$, not the rounding of the sum of the two floating point numbers. (In fact, this sum may be computed exactly; see exercise 10 in Chapter 5.) In the case where $y \approx -x$, this would be considered an ill-conditioned problem, however, since small changes in $x$ and $y$ can...
make a large relative change in their sum; the algorithm for adding the two numbers does as well as one could hope. On the other hand, the subtraction of two nearly equal numbers might be only one step in an algorithm for solving a well-conditioned problem; in this case, the algorithm is likely to be unstable, and one should look for an alternative algorithm that avoids this difficulty.

**Backward error analysis:** Here one tries to show that the computed value is the exact solution to a nearby problem. If the given problem is ill-conditioned (i.e., if a small change in the input data makes a large change in the solution), then probably the best one can hope for is to compute the exact solution of a problem with slightly different input data. For the problem of summing the two numbers \( x \) and \( y \), we have from (6.1)

\[
\text{fl}(x + y) = x(1 + \delta_1)(1 + \delta_3) + y(1 + \delta_2)(1 + \delta_3),
\]

so the computed value is the exact sum of two numbers that differ from \( x \) and \( y \) by relative amounts no greater than \( 2\epsilon + \epsilon^2 \). The algorithm for adding two numbers is **backward stable**.

**Example:** Compute \( \exp(x) \) using the Taylor series expansion

\[
\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots.
\]

The following MATLAB code can be used to compute \( \exp(x) \):

```matlab
oldsum = 0;
newsum = 1;
term = 1;
n = 0;
while newsum ~= oldsum, % Iterate until next term is negligible
    n = n + 1;
term = term * x/n; % x^n/n! = (x^{n-1}/(n-1)!) * x/n
oldsum = newsum;
newsum = newsum + term;
end;
```

This code adds terms in the Taylor series until the next term is so small that adding it to the current sum makes no change in the floating point number that is stored. The code works fine for \( x > 0 \). The computed result for \( x = -20 \), however, is \( 5.6219e - 9 \); the correct result is \( 2.0612e - 9 \). The size of the terms in the series increases to \( 4.3 \times 10^7 \) (for \( n = 20 \)) before starting to decrease, as seen in Table 6.1. This results in double precision rounding errors of size about \( 4.3 \times 10^7 \times 10^{-16} = 4.3 \times 10^{-9} \), which lead to completely wrong results when the true answer is on the order of \( 10^{-9} \).

Note that the problem of computing \( \exp(x) \) for \( x = -20 \) is well-conditioned in both the absolute and relative sense: \( C(x)|_{x=-20} = \frac{d}{dx}\exp(x)|_{x=-20} = \exp(-20) << 1 \) and \( \kappa(x)|_{x=-20} = |x\exp(x)/\exp(x)|_{x=-20} = 20 \). Hence the difficulty here is with the **algorithm**; it is **unstable**.
Table 6.1: Terms added in an unstable algorithm for computing $e^x$.

<table>
<thead>
<tr>
<th>n</th>
<th>$n$th term of series</th>
<th>$n$</th>
<th>$n$th term of series</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-20</td>
<td>25</td>
<td>-2.16e+07</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>30</td>
<td>4.05e+06</td>
</tr>
<tr>
<td>3</td>
<td>-1.33e+03</td>
<td>40</td>
<td>1.35e+04</td>
</tr>
<tr>
<td>4</td>
<td>6.67e+03</td>
<td>50</td>
<td>3.70e+00</td>
</tr>
<tr>
<td>5</td>
<td>-2.67e+04</td>
<td>60</td>
<td>1.39e-04</td>
</tr>
<tr>
<td>10</td>
<td>2.82e+06</td>
<td>70</td>
<td>9.86e-10</td>
</tr>
<tr>
<td>15</td>
<td>-2.51e+07</td>
<td>80</td>
<td>1.69e-15</td>
</tr>
<tr>
<td>20</td>
<td>4.31e+07</td>
<td>90</td>
<td>8.33e-22</td>
</tr>
</tbody>
</table>

Exercise: How can you modify the above code to produce accurate results when $x$ is negative?

Hint: Note that $\exp(-x) = 1/\exp(x)$.

Example: Numerical differentiation: Using Taylor’s theorem with remainder, one can approximate the derivative of a function $f(x)$:

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\xi), \quad \xi \in [x, x+h].$$

$$f'(x) = \frac{f(x + h) - f(x)}{h} - \frac{h}{2}f''(\xi).$$

The term $-\frac{h}{2}f''(\xi)$ is referred to as the truncation error or discretization error when the approximation $(f(x+h) - f(x))/h$ is used for $f'(x)$. The truncation error in this case is of order $h$, denoted $O(h)$, and the approximation is said to be first order accurate.

Suppose one computes this finite difference quotient numerically. In the best possible case, one may be able to store $x$ and $x+h$ exactly, and suppose that the only errors made in evaluating $f(x)$ and $f(x+h)$ come from rounding the results at the end. Then, ignoring any other rounding errors in subtraction or division, one computes:

$$\frac{f(x + h)(1 + \delta_1) - f(x)(1 + \delta_2)}{h} = \frac{f(x + h) - f(x)}{h} + \frac{\delta_1 f(x + h) - \delta_2 f(x)}{h}.$$

Since $|\delta_1|$ and $|\delta_2|$ are less than $\epsilon$, the absolute error due to roundoff is less than or equal to about $2\epsilon |f(x)|/h$, for small $h$. Note that the truncation error is proportional to $h$, while the rounding error is proportional to $1/h$. Decreasing $h$ lowers the truncation error but increases the error due to roundoff.

Suppose $f(x) = \sin(x)$ and $x = \pi/4$. Then $f'(x) = \cos(x)$ and $f''(x) = -\sin(x)$, so the truncation error is about $\sqrt{2}h/4$, while the rounding error is about $\sqrt{2}\epsilon/h$. The most accurate approximation is obtained when the two errors are approximately equal:

$$\frac{\sqrt{2}h}{4} = \frac{\sqrt{2}\epsilon}{h} \Rightarrow h = 2\sqrt{\epsilon}.$$
In this case, the error is on the order of the square root of the machine precision, far less than what one might have hoped for!

Once again, the fault is with the *algorithm*, not the problem of determining \( \frac{d}{dx} \sin(x) |_{x=\pi/4} = \cos(x) |_{x=\pi/4} = \sqrt{2}/2 \). This problem is well-conditioned since if the input argument \( x = \pi/4 \) is changed slightly then the answer, \( \cos(x) \) changes only slightly; quantitatively, the absolute condition number is \( C(x) |_{x=\pi/4} = | - \sin(x) |_{x=\pi/4} = \sqrt{2}/2 \), and the relative condition number is \( \kappa(x) |_{x=\pi/4} = | - x \sin(x) / \cos(x) |_{x=\pi/4} = \pi/4 \). When the problem of evaluating the derivative is replaced by one of evaluating the finite difference quotient, then the conditioning becomes bad. Later we will see how to use higher order accurate approximations to the derivative to obtain somewhat better results, but we will not achieve the full machine precision using finite difference quotients.
Ch. 6 Exercises

1. What are the absolute and relative condition numbers of the following functions? Where are they large?

   (a) \((x - 1)^\alpha\)  
   (b) \(1/(1 + x^{-1})\)  
   (c) \(\ln x\)  
   (d) \(\log_{10} x\)  
   (e) \(x^{-1}e^x\)  
   (f) \(\arcsin x\)

2. Let \(f(x) = \sqrt{x}\).

   (a) Find the absolute and relative condition numbers of \(f\).
   (b) Where is \(f\) well-conditioned in an absolute sense? In a relative sense?
   (c) Suppose \(x = 10^{-17}\) is replaced by \(\hat{x} = 10^{-16}\) (a small absolute change but a large relative change). Using the absolute condition number of \(f\), how much of a change is expected in \(f\) due to this change in the argument?

3. In evaluating the finite difference quotient \((f(x + h) - f(x))/h\), which is supposed to approximate \(f'(x)\), suppose that \(x = 1\) and \(h = 2^{-24}\). What would be the computed result using IEEE single precision? Explain your answer.

4. Use MATLAB or your calculator to compute \(\tan(x_j)\), for \(x_j = \frac{\pi}{4} + 2\pi \times 10^j, \ j = 0, 1, 2, \ldots, 20\). In MATLAB, use “format long e” to print out all of the digits in the answers. What is the relative condition number of the problem of evaluating \(\tan(x)\) for \(x = x_j\)? Suppose the only error that you make in computing the argument \(x_j = \frac{\pi}{4} + (2\pi) \times 10^j\) occurs because of rounding \(\pi\) to 16 decimal places; i.e., assume that the addition and multiplications and division are done exactly. By what absolute amount will your computed argument \(\hat{x}_j\) differ from the exact one? Use this to explain your results.

5. **Compound interest.** Suppose \(a_0\) dollars are deposited in a bank that pays 5% interest per year, compounded quarterly. After one quarter the value of the account is

   \[a_0 \times (1 + (.05)/4)\]

   dollars. At the end of the second quarter, the bank pays interest not only on the original amount \(a_0\), but also on the interest earned in the first quarter; thus the value of the investment at the end of the second quarter is

   \[a_0 \times (1 + (.05)/4) \times (1 + (.05)/4)\]

   dollars. At the end of the third quarter the bank pays interest on this amount, so that the account is now worth \(a_0 \times (1 + (.05)/4)^3\) dollars, and at the end of the whole year the investment is finally worth

   \[a_0 \times (1 + (.05)/4)^4\]
dollars. In general, if $a_0$ dollars are deposited at annual interest rate $x$, compounded $n$ times per year, then the account value after one year is

$$a_0 \times I_n(x), \quad \text{where} \quad I_n(x) = \left(1 + \frac{x}{n}\right)^n.$$ 

This is the compound interest formula. It is well-known that for fixed $x$, $\lim_{n \to \infty} I_n(x) = \exp(x)$.

(a) Determine the relative condition number $\kappa_{I_n}(x)$ for the problem of evaluating $I_n(x)$. For $x = .05$, would you say that this problem is well-conditioned or ill-conditioned?

(b) Use MATLAB to compute $I_n(x)$ for $x = .05$ and for $n = 1, 10, 10^2, ..., 10^{15}$. Use “format long e” so that you can see if your results are converging to $\exp(x)$, as one might expect. Turn in a table with your results and a listing of the MATLAB command(s) you used to compute these results.

(c) Try to explain the results of part (b). In particular, for $n = 10^{15}$, you probably computed 1 as your answer. Explain why. To see what is happening for other values of $n$, consider the problem of computing $z^n$ when $n$ is large. What is the relative condition number of this problem? If you make an error of about $10^{-16}$ in computing $z = (1 + x/n)$, about what size error would you expect in $z^n$?

(d) Can you think of a better way than the method you used in part (b) to accurately compute $I_n(x)$ for $x = .05$ and for large values of $n$? Demonstrate your new method in MATLAB or explain why it should give more accurate results.