1 Computer representation of numbers and roundoff error

In this chapter, we will introduce the notion and consequences of a finite number system. Each number in a computer must be physically stored. Therefore, a computer can only hold a finite number of digits for any number. Decades of research and experimentation has led us to a (usually) reasonable approximation of numbers by representing them with (about) sixteen digits in base 10. While this approximation may seem at first to be reasonable, we will explore its consequences in this chapter. In particular, we will discuss how to not make mistakes that can arise from using a finite number system. The representation of numbers is not specific to a program or programming language like MATLAB but they are part of the hardware of the processors of virtually every computer.

1.1 Examples of the effects of roundoff error

To motivate the need to study computer representation of numbers, let's consider first some examples taken from MATLAB - but we note the same thing happens in C, Java, etc.:

1. The order in which you add numbers on a computer makes a difference!

```
>> 1 + 1e-16 + 1e-16 + 1e-16 + 1e-16 + 1e-16 + 1e-16 + 1e-16

ans =

     1

>> 1e-16 + 1e-16 + 1e-16 + 1e-16 + 1e-16 + 1e-16 + 1e-16 + 1

ans =

   1.000000000000001
```

   Note: `AAAeBBB` is a common notation for a floating point number with the value $AAA \times 10^{BBB}$. So `1e-16` $= 10^{-16}$.
   As we will see later in this chapter, the computer stores about 16 base 10 digits for each number; this means we get 15 digits after the first nonzero digit of a number. Hence, if you try to add 1e-16 to 1, there is nowhere for the computer to store the 1e-16 since it is the 17th digit of a number starting with 1. It does not matter how many times you add 1e-16, it just gets lost in each intermediate step, since operations are always done from

left to right. So even if we add 1e-16 to 1, 10 times in a row, we get back exactly 1. However, if we first add the 1e-16's together, then add the 1, these small numbers get a chance to combine to become big enough not to be lost when added to 1.

2. Consider

$$f(x) = \frac{e^x - e^{-x}}{x}.$$

Suppose we wish to calculate

$$\lim_{x \to 0} f(x).$$

By L'Hopital's theorem, we can easily determine the answer to be 2. However, how might one do this on a computer? A limit is an infinite process, and moreover it requires some analysis to get an answer. Hence on a computer one is seemingly left with the option of choosing small x's and plugging them into f. The table below shows what we get back from MATLAB by doing so.

x	$\frac{e^x - e^{-x}}{x}$
10^{-6}	1.999999999946489
10^{-7}	1.999999998947288
10^{-8}	1.999999987845058
10^{-9}	2.000000054458440
10^{-10}	2.000000165480742
10^{-11}	2.000000165480742
10^{-12}	2.000066778862220
10^{-13}	1.999511667349907
10^{-14}	1.998401444325282
10^{-15}	2.109423746787797
10^{-16}	1.110223024625157
10^{-17}	0

Table 1.1: Unstable limit computation.

Moreover, if we choose x any smaller than 1e-17, we still get 0. The main numerical issue here is, as we will learn, subtracting two nearly equal numbers on a computer is bad and can lead to large errors.
Interestingly, if we create the limit table using a mathematically equivalent expression for $f(x)$, we can get a much better answer. Recall (that is, this is a fact which we hope you have seen before): the exponential function is defined by

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + ...$$

Using this definition, we calculate

$$\frac{e^x - e^{-x}}{x} = \frac{2x + 2\frac{x^3}{3!} + 2\frac{x^5}{5!} + 2\frac{x^7}{7!} + ...}{x} = 2 + 2\frac{x^2}{3!} + 2\frac{x^4}{5!} + 2\frac{x^6}{7!} + ...$$

Notice that if $|x| < 10^{-4}$, then $2\frac{x^4}{5!} < 10^{-16}$, and therefore, this term (and all the ones after it in the sum) will not affect the calculation of the sum in any of the first 16 digits. Hence, we have that, in MATLAB,

$$\frac{e^x - e^{-x}}{x} = 2 + \frac{x^2}{3}$$

provided $|x| < 10^{-4}$. We can expect this approximation of $f(x)$ to be accurate to 16 digits in base 10. Recalculating a limit table based on this mathematically equivalent expression provides much better results, as can be seen in table 1.2, which shows the numerical limit is clearly 2.

3. We learn in Calculus II that the integral

$$\int_1^{\infty} \frac{1}{x} = \infty.$$

Note that since this is a decreasing, concave up function, approximating it with the left rectangle rule gives an over-approximation of the integral. That is,

$$\sum_{n=1}^{\infty} \frac{1}{n} > \int_1^{\infty} \frac{1}{x},$$

and so we have that

$$\sum_{n=1}^{\infty} \frac{1}{n} = \infty.$$

But if we calculate this sum in MATLAB, we converge to a finite number instead of infinity. As mentioned above, the computer can only store (about)

x	$2+\frac{x^2}{3}$
10^{-6}	2.000000000000334
10^{-7}	2.000000000000004
10^{-8}	2.000000000000000
10^{-9}	2.000000000000000
10^{-10}	2.000000000000000
10^{-11}	2.000000000000000
10^{-12}	2.000000000000000
10^{-13}	2.000000000000000
10^{-14}	2.000000000000000
10^{-15}	2.000000000000000
10^{-16}	2.000000000000000
10^{-17}	2.000000000000000

Table 1.2: Stable limit computation.

16 digits. Since the running sum will be greater than 1, any number smaller than 1e-16 will be lost due to roundoff error. Thus, if we add in order from left to right, we get that

$$\sum_{n=1}^{\infty} \frac{1}{n} = \text{ (in MATLAB)} \sum_{n=1}^{10^{16}} \frac{1}{n} < \infty.$$

Hence numerical error has caused a sum which should be infinite to be finite!

1.2 Binary numbers

Computers and software allow us to work in base 10, but behind the scenes everything is done in base 2. This is because numbers are stored in computer memory (essentially) as 'voltage on' (1) or 'voltage off' (0). Hence, it is natural to represent numbers in their base 2, or binary, representation. To explain this, let us start with base 10, or decimal, number system. In base 10, the number

12.625 can be expanded into powers of 10, each multiplied by a coefficient:

$$12.625 = 1 \times 10^1 + 2 \times 10^0 + 6 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}.$$

It should be intuitive that the coefficients of the powers of 10 must be digits between 0 and 9. Also, the decimal point goes between the coefficients of 10^0 and 10^{-1}.

Base 2 numbers work in an analogous fashion. First, note that it only makes sense to have digits of 0 and 1, for the same reason that digits in base 10 must be 0 through 9. Also, the decimal point goes between the coefficients of 2^0 and 2^{-1}. Hence in base 2 we have, for example, that

$$(11.001)_{base2} = 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 2 + 1 + \frac{1}{8} = 3.125.$$

Converting a base 2 number to a base 10 number is nothing more than expanding it into powers of 2. To get an intuition for this, consider Table 1.3 that converts the base 10 numbers 1 through 10.

Base 10 representation	Base 2 representation
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Table 1.3: Binary representation of the numbers from 1 to 10.

The following algorithm will convert a base 10 number to a base 2 number. Note this is not the most efficient computational algorithm, but perhaps it is the easiest to understand for beginners.

Given a base 10 decimal d,

1. Find the biggest power p of 2 such that $2^p \leq d$ but $2^{p+1} > d$, and save the index number p.
2. Set $d = d - 2^p$.
3. If d is 0 or very small (compared to the d you started with), then stop. Otherwise go to step 1.

Finally, write down your number as 1's and 0's, putting 1's in the entries where you saved the indices (the p above) and 0 everywhere else.

Example 1. *Convert the base 10 number d=11.5625 to base 2.*

1. *Find* $p = 3$ *because* $2^3 = 8 \leq 11.5625 < 16 = 2^4$. *Set* $d = 11.5625 - 8 = 3.5625$
2. *Find* $p = 1$ *because* $2^1 = 2 \leq 3.5625 < 4 = 2^2$. *Set* $d = 3.5625 - 2 = 1.5625$
3. *Find* $p = 0$ *because* $2^0 = 1 \leq 1.5625 < 2 = 2^1$. *Set* $d = 1.5625 - 1 = 0.5625$
4. *Find* $p = -1$ *because* $2^{-1} = \frac{1}{2} \leq 0.5625 < 1 = 2^0$. *Set* $d = 0.5625 - 0.5 = 0.0625$
5. *Find* $p = -4$ *because* $2^{-4} = \frac{1}{16} = 0.0625 \leq 0.0625 < \frac{1}{8} = 2^{-3}$. *Set* $d = 0.0625 - 0.0625 = 0$

Thus the process has terminated, since d=0. Our base 2 number thus has 1's in the 3,1,0,-1, and -4 places, so we get

$$(11.5625)_{base10} = (1011.1001)_{base2}.$$

Of course, not every base 10 number will terminate to a finite number of base 2 digits. For most computer number systems, we have a total of 53 base 2 digits to represent a base 10 number.

We now introduce the notion of **standard binary form**, which can be considered analogous to exponential formatting in base 10. The idea here is that every binary number, except 0, can be represented as

$$x = \pm 1.b_1 b_2 b_3 \cdots \times 2^{exponent},$$

where each b_i is a 0 or a 1, and the exponent (a positive or negative whole number) is adjusted so that the only digit to the left of the decimal point is a single 1. Some examples are given in the table below.

Base 2	Standard binary form
1.101	1.101×2^0
-1011.1101	-1.0111101×2^3
0.000011101	1.1101×2^{-5}

Table 1.4: Examples of standard binary form. Note that the exponent is a number in decimal format.

1.3 64 bit floating point numbers

By far the most common computer number representation system is the 64-bit 'double' floating point number system. This is the default used by all major mathematical and computational software. In some cases it makes sense to use 32 or 128 bit number systems, but that is a discussion for later (later, as in "not in this book"), as first we must learn the basics. Each 'bit' on a computer is a 0 or a 1, and each number on a computer is represented by 64 0's and 1's. If we assume each number is in standard binary form, then the important information for each number is i) sign of the number, ii) exponent, iii) the digits after the decimal point. Note that the number 0 is an exception and is treated as a special case for the number system.

The IEEE standard divides up the 64 bits as follows:

- 1 bit sign: 0 for positive, 1 for negative;
- 11 bit exponent: the base 2 representation of (standard binary form exponent + 1023);
- 52 bit mantissa: the first 52 digits after decimal point from standard binary form.

The reason for the "shift" (sometimes also called bias) of 1023 in the exponent is so that the computer does not have to store a sign for the exponent (more numbers can be stored this way). The computer knows internally that the number is shifted, and knows how to handle it.

With the bits from above denoted as sign s, exponent E, and mantissa $b_1, \ldots, b_{52}$ the corresponding number is standard binary form is $(-1)^s \cdot 1.b_1 \ldots b_{52} \times 2^{E-1023}$.

Example 2. *Convert the base 10 number d=11.5625 to 64 bit double floating point representation.*

From a previous example, we know that $11.5625 = (1011.1001)_{base2}$*, and so has standard binary representation of* 1.0111001×2^3*. Hence, we immediately know that*

sign bit = 0

mantissa = 0111001000000000000000000000000000000000000000000000

For the exponent, we need the binary representation of (3+1023)=1026 = 1024 + 2, and thus,

exponent = 10000000010

There are some obvious consequences to this number system:

1. There is a biggest and smallest positive representable number!
   - The exponent can hold 11 bits total, which means the base 10 numbers 0 to 2047. Due to the shift, the biggest positive unshifted number it can hold is 1024, and the biggest negative unshifted number it can hold is -1023. However, the unshifted numbers 1024 and -1023 have special meanings (e.g. representing 0 and infinity). So the smallest and largest workable exponents are -1022 and 1023. This means the largest number that a computer can hold is

     $$n_{max} = 1.1111111...1 \times 2^{1023} \approx 10^{308}$$

     and similarly the smallest positive representable number is

     $$n_{min} = 1.00000.....0 \times 2^{-1022} \approx 10^{-308}$$

     Compare these with what MATLAB returns for `realmin` and `realmax`. That said, MATLAB can represent slightly smaller numbers up to approximately 10^{-324}. These numbers are called denormalized numbers.
   - Having these numbers as upper and lower bounds on positive representable numbers is generally not a problem. Usually, if one needs to deal with numbers larger or smaller than this, the entire problem can be rescaled into units that are representable.
2. The relative spacing between 2 floating point numbers is $2^{-52} \approx 2.22 \times 10^{-16}$.

- This relative spacing between numbers is generally referred to as **machine epsilon** or eps in MATLAB, and we will denote it by $\epsilon_{mach} = 2^{-52}$.
- Given a number $d = 1.b_1b_2...b_{52} \times 2^{exponent}$, the smallest increment we can add to it is in the 52^{nd} digit of the mantissa, which is $2^{-52} \times 2^{exponent}$. Any number smaller would be after the 52nd digit in the mantissa.
- As there is spacing between two floating point numbers, any real number between two floating point numbers must be rounded (to the nearest one). This means the maximum relative error in representing a number on the computer is about 1.11×10^{-16}. Thus, we can expect 16 digits of accuracy if we enter a number into the computer.
- Although it is usually enough to have 16 digits of accuracy, in some situations this is not sufficient. Since we often rely on computer calculations to let us know a plane is going to fly, a boat is going to float, or a reactor will not melt down, it is critical to know when computer arithmetic error can cause a problem and how to avoid it.

There are two main types of catastrophic computer arithmetic error: adding large and small numbers together, and subtracting two nearly equal numbers. We will describe each of these issues now.

1.3.1 Adding large and small numbers is bad

As we saw in Example 1 in this chapter, if we add 1 to 10^{-16}, it does not change the 1 at all. Additionally, the next computer representable number after 1 is $1 + 2^{-52} = 1 + 2.22 \times 10^{-16}$. Since $1 + 10^{-16}$ is closer to 1 than it is to $1 + 2.22 \times 10^{-16}$, it gets rounded to 1, leaving the 10^{-16} to be lost forever.

We have seen this effect in the example at the beginning of this chapter when repeatedly adding 10^{-16} to 1. Theoretically speaking, addition in floating point computation is not associative, meaning $(A + B) + C = A + (B + C)$ *may not hold*, due to rounding.

One way to minimize this type of error when adding several numbers is to add from smallest to largest (if they all have the same sign) and to use factorizations that lessen the problem. There are other more complicated ways to deal with this kind of error that is out of the scope of this book, for example the "Kahan Summation Formula".

1.3.2 Subtracting two nearly equal numbers is bad

The issue here is that insignificant digits can become significant digits, and the problem is illustrated in Example 2, earlier in this chapter. Consider the following MATLAB command and output:

```
>> 1 + 1e-15 - 1
ans =
     1.110223024625157e-15
```

Clearly, the answer should be 10^{-15}, but we do not get that, as we observe error in the second significant digit. It is true that the digits of accuracy in the subtraction operation is 16, but there is a potential problem with the "garbage" digits 110223024625157 (these digits arise from rounding error). If we are calculating a limit, for example, they could play a role.

Consider using the computer to find the derivative of $f(x) = x$ at $x = 1$. Everyone in the world, including babies and the elderly, knows the answer is $f'(1) = 1$. But suppose for a moment we do not know the answer and wish to calculate an approximation. The definition of the derivative tells us

$$f'(1) = \lim_{h \to 0} \frac{f(1+h) - f(1)}{h}.$$

It might seem reasonable to just pick a very small h to get a good answer, but this is a bad idea! Consider the following MATLAB commands, which plug in values of h from 10^{-1} to 10^{-20}. When h=10^{-15}, we see the "garbage" digits have become significant and alter the second significant digit! For even smaller values of h, $1 + h$ gives back 1, and so the derivative approximation is 0.

```
>> h = 10.^-[1:20]';
>> fp = ((1+h) - 1) ./ h;
>> disp([h, fp]);
   0.100000000000000   1.000000000000001
   0.010000000000000   1.000000000000001
   0.001000000000000   0.999999999999890
   0.000100000000000   0.999999999999890
   0.000010000000000   1.000000000006551
   0.000001000000000   0.999999999917733
   0.000000100000000   1.000000000583867
   0.000000010000000   0.999999993922529
   0.000000001000000   1.000000082740371
   0.000000000100000   1.000000082740371
   0.000000000010000   1.000000082740371
   0.000000000001000   1.000088900582341
   0.000000000000100   0.999200722162641
   0.000000000000010   0.999200722162641
```

```
0.000000000000001   1.110223024625157
0.000000000000000                   0
0.000000000000000                   0
0.000000000000000                   0
0.000000000000000                   0
0.000000000000000                   0
```

We will discuss in detail, in a later chapter, more accurate ways to approximate derivatives. Although roundoff error will always be a problem, more accurate methods can give good approximations for h only moderately small, and thus, minimize potential issues from roundoff.

There is another fact about floating point numbers to be aware of. What do you expect the following program to do?

```
i=1.0
while i~=0.0
  i=i-0.1
end
```

The operator "~=" means "not equal" in MATLAB. We would expect the loop to be counting down from 1.0 in steps of 0.1 until we reach 0, right? No, it turns out that we are running an endless loop counting downwards because i is never exactly equal to 0.

Rule: Never compare floating point numbers for equality ("==" or "~="). Instead, for example, use a comparison such as

```
abs(i)< 1e-12
```

to replace the comparison `i~=0.0` from the above code.

1.4 Exercises

1. Convert the binary number 1101101.1011 to decimal format.
2. Convert the decimal number 63.125 to binary format. What is its 64-bit floating point representation?
3. Show that the decimal number 0.1 cannot be represented exactly as a finite binary number (i.e. the base 2 expansion is not terminal). Use this fact to explain that "0.1*3 == 0.3" returns 0 (meaning false) in MATLAB.
4. Write your own MATLAB function `tobinary(n)` that, given a whole number, outputs its base 2 representation. The output should be a vector with 0s and 1s. Example:

```
>> tobinary(34)
```

```
ans =
     1     0     0     0     1     0
```

Hint: The MATLAB command `pmax=floor(log2(n))` finds the biggest power of 2 contained in n.

5. What is the minimum and maximum distance between two adjacent floating point numbers?
6. What is the best way to numerically evaluate $\|x\|_2 = \sqrt{\sum_{n=1}^{N} x_n^2}$? (i.e., is there a best order for the addition?)
7. Consider the function
$$g(x) = \frac{e^x - 1}{x}.$$
a) Find $\lim_{x\to 0} g(x)$ analytically.
b) Write a MATLAB program to calculate $g(x)$ for $x = 10^{-1},\ 10^{-2}, ..., 10^{-15}$. Explain why the analytical limit and the numerical 'limit' do not agree.
c) Extend your calculation to $x = 10^{-16},\ 10^{-17}, ..., 10^{-20}$, and explain this behavior.
8. The polynomial $(x-2)^6$ is 0 when $x = 2$ but is positive everywhere else. Plot both this function, and its decomposition $x^6 - 12x^5 + 60x^4 - 160x^3 + 240x^2 - 192x + 64$ near $x = 2$, on [1.99,2.01] using 10,000 points. Explain the differences in the plots.
9. What is the next biggest computer representable number after 1 (assuming 64 bit double floating point numbers)? What about after 4096? What about after $\frac{1}{8}$?
10. Read about the Patriot missile failure during the Gulf War that, due to roundoff error, accidentally hit an American Army barracks, killing 28 soldiers and injuring over 100 others:
http://www-users.math.umn.edu/~arnold/disasters/patriot.html.
More examples of serious problems are discussed at http://mathworld.wolfram.com/RoundoffError.html. Write a paragraph about the importance of roundoff error in software calculations.
11. The mathematical definition of the exponential function is
$$\exp(x) = e^x = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$
Since factorial dominates power functions, this infinite sum will converge, although one may need n to be very large. For $x = 2.2$, calculate e^x using MATLAB's 'exp' function, and also using the definition. How large must n be in order for these two methods to give the same answer to 16 digits?

12. The mathematical definition of the sine function is

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}.$$

Since factorial dominates power functions, this infinite sum will converge, although one may need n to be very large. For $x = \frac{\pi}{4}$, calculate $\sin(x)$ using MATLAB's 'sin' function and also using the definition. How large must n be in order for these two methods to give the same answer to 16 digits?