Chapter 5

Computer Arithmetic and Round-Off Errors

In the two previous chapters we have seen how numbers can be represented in the binary numeral system and how this is the basis for representing numbers in computers. Since any given integer only has a finite number of digits, we can represent all integers below a certain limit *exactly*. Non-integer numbers are considerably more cumbersome since infinitely many digits are needed to represent most of them, regardless of which numeral system we employ. This means that most non-integer numbers cannot be represented in a computer without committing an error which is usually referred to as *round-off error* or *rounding error*.

As we saw in Chapter 4, the standard representation of non-integer numbers in computers is as floating-point numbers with a fixed number of bits. Not only is an error usually committed when a number is represented as a floating-point number; most calculations with floating-point numbers will induce further round-off errors. In most situations these errors will be small, but in a long chain of calculations there is a risk that the errors may accumulate and seriously pollute the final result. It is therefore important to be able to recognise when a given computation is going to be troublesome or not, so that we may know whether the result can be trusted.

In this chapter we will start our study of round-off errors. The key observation is that subtraction of two almost equal numbers may lead to large round-off error. There is nothing mysterious about this — it is a simple consequence of how arithmetic is performed with floating-point numbers. We will therefore need a quick introduction to floating-point arithmetic. We will also discuss the two most common ways to measure error, namely *absolute error* and *relative error*.

5.1 Integer arithmetic and errors

Integers and integer arithmetic on a computer is simple both to understand and analyse. All integers up to a certain size are represented exactly and arithmetic with integers is exact. The only thing that can go wrong is that the result becomes larger than what is representable by the type of integer used. The computer hardware, which usually represents integers with two's complement (see section 4.1.3), will not protest and just wrap around from the largest positive integer to the smallest negative integer or vice versa.

As was mentioned in chapter 4, different programming languages handle overflow in different ways. Most languages leave everything to the hardware. This means that overflow is not reported, even though it leads to completely wrong results. This kind of behaviour is not really problematic since it is usually easy to detect (the results are completely wrong), but it may be difficult to understand what went wrong unless you are familiar with the basics of two's complement. Other languages, like Python, gracefully switch to higher precision. In this case, integer overflow is not serious, but may reduce the computing speed. Other error situations, like division by zero, or attempts to extract the square root of negative numbers, lead to error messages and are therefore not serious.

The conclusion is that errors in integer arithmetic are not serious, at least not if you know a little bit about how integer arithmetic is performed.

5.2 Floating-point arithmetic and round-off error

Errors in floating-point arithmetic are more subtle than errors in integer arithmetic since, in contrast to integers, floating-point numbers can be just a little bit wrong. A result that appears to be reasonable may therefore contain errors, and it may be difficult to judge how large the error is. A simple example will illustrate.

Example 5.1. On a typical calculator we compute $x = \sqrt{2}$, then $y = x^2$, and finally $z = y - 2$, i.e., the result should be $z = \left(\sqrt{2}\right)^2 - 2$, which of course is 0. The result reported by the calculator is

$$z = -1.38032020120975 \times 10^{-16}.$$

This is a simple example of round-off error. ∎

The aim of this section is to explain why computations like the one in example 5.1 give this obviously wrong result. To do this we will give a very high-level introduction to computer arithmetic and discuss the potential for errors in the four elementary arithmetic operations addition, subtraction, multiplication, and division with floating-point numbers. Perhaps a bit surprisingly, the conclusion will be that the most critical operation is addition/subtraction, which in certain situations may lead to dramatic errors.

A word of warning is necessary before we continue. In this chapter, and in particular in this section, where we focus on the shortcomings of computer arithmetic, some may conclude that round-off errors are so dramatic that we had better not use a computer for serious calculations. This is a misunderstanding. The computer is an extremely powerful tool for numerical calculations that you should use whenever you think it may help you, and most of the time it will give you answers that are much more accurate than you need. However, you should be alert and aware of the fact that in certain cases errors may cause considerable problems.

5.2.1 Truncation and rounding

Floating point numbers on most computers use binary representation, and we know that in the binary numeral system all real numbers that are fractions on the form a/b, with a an integer and $b = 2^k$ for some positive integer k can be represented exactly (provided a and b are not too large), see lemma 3.22. This means that numbers like 1/2, 1/4 and 5/16 are represented exactly.

On the other hand, it is easy to forget that numbers like 0.1 and 3.43 are *not* represented exactly. And of course all numbers that cannot be represented exactly in the decimal system cannot be represented exactly in the binary system either. These numbers include fractions like 1/3 and 5/12 as well as all irrational numbers. Even before we start doing arithmetic we therefore have the challenge of finding good approximations to these numbers that cannot be represented exactly within the floating-point model being used. We will distinguish between two different ways to do this, *truncation* and *rounding*.

Definition 5.2 (Truncation)**.** *A number is said to be truncated to m digits when each digit except the m leftmost ones is replaced by* 0*.*

Example 5.3 (Examples of truncation)**.** The number 0.33333333 truncated to 4 digits is 0.3333, while 128.4 truncated to 2 digits is 120, and 0.67899 truncated to 4 digits is 0.6789. ∎

Note that truncating a positive number a to an integer is equivalent to applying the floor function to a, i.e., if the result is b then

$$b = \lfloor a \rfloor.$$

Truncation is a very simple way to convert any real number to a floating-point number: We just start computing the digits of the number, and stop as soon as we have all the required digits. However, the error will often be much larger than necessary, as in the last example above. *Rounding* is an alternative to truncation that in general will make the error smaller, but is more complicated.

Definition 5.4 (Rounding)**.** *A number is said to be rounded to m digits when it is replaced by the nearest number with the property that all digits beyond position m is* 0*.*

Example 5.5 (Examples of rounding)**.** The number 0.33333333 rounded to 4 digits is 0.3333. The result of rounding 128.4 to 2 digits is 130, while 0.67899 rounded to 4 digits is 0.6790. ∎

Rounding is something most of us do regularly when we go shopping, as well as in many other contexts. However, there is one slight ambiguity in the definition of rounding: What to do when the given number is halfway between two m digit numbers. The standard rule taught in school is to *round up* in such situations. For example, we would usually round 1.15 to 2 digits as 1.2, but 1.1 would give the same error. For our purposes this is ok, but from a statistical point of view it is biased since we always round up when in doubt.

5.2.2 A simplified model for computer arithmetic

The details of computer arithmetic are technical, but luckily the essential features of both the arithmetic and the round-off errors are present if we use the same model of floating-point numbers as in section 4.2. Recall that any positive real number a may be written as a normalised decimal number

$$\alpha \times 10^n,$$

where the number α is in the range $[0.1, 1)$ and is called the *significand*, while the integer n is called the *exponent*.

Fact 5.6 (Simplified model of floating-point numbers)**.** *Most of the shortcomings of floating-point arithmetic become visible in a computer model that uses 4 digits for the significand, 1 digit for the exponent plus an optional sign for both the significand and the exponent.*

Two typical normalised (decimal) numbers in this model are 0.4521×10^1 and -0.9×10^{-5}. The examples in this section will use this model, but the results can be generalised to a floating-point model in base β, see exercise 5.

In the normalised, positive real number $a = \alpha \times 10^n$, the integer n provides information about the size of a, while α provides the decimal digits of a, as a fractional number in the interval $[0.1, 1)$. In our simplified arithmetic model, the restriction that n should only have 1 decimal digit restricts the *size* of a, while the requirement that α should have at most 4 decimal digits restricts the *precision* of a.

It is easy to realise that even simple operations may lead to numbers that exceed the maximum size imposed by the floating-point model — just consider a multiplication like $10^8 \times 10^7$. This kind of problem is easy to detect and is therefore not serious. The main challenge with floating-point numbers lies in keeping track of the number of correct digits in the significand. If you are presented with a result like 0.4521×10^1 it is reasonable to expect the 4 digits 4521 to be correct. However, it may well happen that only some of the first digits are correct. It may even happen that none of the digits reported in the result are correct. If the computer told us how many of the digits were correct, this would not be so serious, but in most situations you will get no indication that some of the digits are incorrect. Later in this section, and in later chapters, we will identify some simple situations where digits are lost.

Observation 5.7. *The critical part of floating-point operations is the potential loss of correct digits in the significand.*

5.2.3 An algorithm for floating-point addition

In order to understand how round-off errors occur in addition and subtraction, we need to understand the basic algorithm that is used. Since $a - b = a + (-b)$, it is sufficient to consider addition of numbers. The basic procedure for adding floating-point numbers is simple (in reality it is more involved than what is stated here).

Algorithm 5.8. *To add two floating-point numbers a and b on a computer, the following steps are performed:*

1. *The number with largest absolute value, say a, is written in normalised form*
$$a = \alpha \times 10^n,$$
and the other number b is written as
$$b = \beta \times 10^n$$

with the same exponent as a and the same number of digits for the significand β.

2. *The significands α and β are added,*

$$\gamma = \alpha + \beta$$

3. *The result $c = \gamma \times 10^n$ is converted to normalised form.*

This apparently simple algorithm contains a serious pitfall which is most easily seen from some examples. Let us first consider a situation where everything works out nicely.

Example 5.9 (Standard case)**.** Suppose that $a = 5.645$ and $b = 7.821$. We convert the numbers to normal form and obtain

$$a = 0.5645 \times 10^1, \quad b = 0.7821 \times 10^1.$$

We add the two significands $0.5645 + 0.7821 = 1.3466$ so the correct answer is 1.3466×10^1. The last step is to convert this to normal form. In exact arithmetic this would yield the result 0.13466×10^2. However, this is not in normal form since the significand has five digits. We therefore perform rounding, $0.13466 \approx 0.1347$, and get the final result

$$0.1347 \times 10^2. \quad \blacksquare$$

Example 5.9 shows that we easily get an error when normalised numbers are added and the result converted to normalised form with a fixed number of digits for the significand. In this first example all the digits of the result are correct, so the error is far from serious.

Example 5.10 (One large and one small number)**.** If $a = 42.34$ and $b = 0.0033$ we convert the largest number to normal form

$$42.34 = 0.4234 \times 10^2.$$

The smaller number b is then written in the same form (same exponent)

$$0.0033 = 0.000033 \times 10^2.$$

The significand in this second number must be rounded to four digits, and the result of this is 0.0000. The addition therefore becomes

$$0.4234 \times 10^2 + 0.0000 \times 10^2 = 0.4234 \times 10^2. \quad \blacksquare$$

The error in example 5.10 may seem serious, but once again the result is correct to four decimal digits, which is the best we can hope for when we only have this number of digits available.

Example 5.11 (Subtraction of two similar numbers I)**.** Consider next a case where $a = 10.34$ and $b = -10.27$ have opposite signs. We first rewrite the numbers in normal form

$$a = 0.1034 \times 10^2, \quad b = -0.1027 \times 10^2.$$

We then add the significands, which really amounts to a subtraction,

$$0.1034 - 0.1027 = 0.0007.$$

Finally, we write the number in normal form and obtain

$$a + b = 0.0007 \times 10^2 = 0.7000 \times 10^{-1}. \quad \blacksquare \tag{5.1}$$

Example 5.11 may seem innocent since the result is exact, but in fact it contains the seed for serious problems. A similar example will reveal what may easily happen.

Example 5.12 (Subtraction of two similar numbers II)**.** Suppose that $a = 10/7$ and $b = -1.42$. Conversion to normal form yields

$$\frac{10}{7} \approx a = 0.1429 \times 10^1, \quad b = -0.142 \times 10^1.$$

Adding the significands yield

$$0.1429 - 0.142 = 0.0009.$$

When this is converted to normal form, the result is

$$0.9000 \times 10^{-3}$$

while the true result rounded to four correct digits is

$$0.8571 \times 10^{-3}. \quad \blacksquare \tag{5.2}$$

5.2.4 Observations on round-off errors in addition/subtraction

In example 5.12 there is a serious problem: We started with two numbers with full four digit accuracy, but the computed result had only one correct digit. In other words, we lost almost all accuracy when the subtraction was performed. The potential for this loss of accuracy was present in example 5.11 where we also had to add digits in the last step (5.1), but there we were lucky in that the added digits were correct. In example 5.12 however, there would be no way for a computer to know that the additional digits to be added in (5.2) should be taken from the decimal expansion of 10/7. Note how the bad loss of accuracy in example 5.12 is reflected in the relative error.

One could hope that what happened in example 5.12 is exceptional; after all example 5.11 worked out very well. This is not the case. It is example 5.11 that is exceptional since we happened to add the correct digits to the result in (5.1). This was because the numbers involved could be represented exactly in our decimal model. In example 5.12 one of the numbers was 10/7 which cannot be represented exactly, and this leads to problems.

Our observations can be summed up as follows.

Observation 5.13. *Suppose the k most significant digits in the two numbers a and b are the same. Then k digits may be lost when the subtraction $a-b$ is performed with algorithm 5.8.*

This observation is very simple: If we start out with m correct digits in both a and b, and the k most significant of those are equal, they will cancel in the subtraction. When the result is normalised, the missing k digits will be filled in from the right. These new digits are almost certainly wrong and hence the computed result will only have $m-k$ correct digits. This effect is called *cancellation* and is a very common source of major round-off problems. The moral is: *Be on guard when two almost equal numbers are subtracted.*

In practice, a computer works in the binary numeral system. However, the cancellation described in observation 5.13 is just as problematic when the numbers are represented as normalised binary numbers.

Example 5.10 illustrates another effect that may cause problems in certain situations. If we add a very small number ϵ to a nonzero number a the result may become exactly a. This means that a test like

$$\textbf{if } a = a + \epsilon$$

may in fact become true.

Observation 5.14. *Suppose that the floating-point model in base β uses m digits for the significand and that a is a nonzero floating-point number. The addition $a+\epsilon$ will be computed as a if*

$$|\epsilon| < 0.5\beta^{-m}|a|. \tag{5.3}$$

The exact factor to be used on the right in (5.3) depends on the details of the arithmetic. The factor $0.5\beta^{-m}$ used here corresponds to rounding to the nearest m digits.

A general observation to made from the examples is simply that there are almost always small errors in floating-point computations. This means that if you have computed two different floating-point numbers a and $\tilde{a}$ that from a strict mathematical point of view should be equal, it is very risky to use a test like

$$\textbf{if } a = \tilde{a}$$

since it is rather unlikely that this will be true.

One situation where this problem may occur is in a loop like

```
$x = 0.0$;
**while** $x \le 1.0$
    **print** $x$
    $x = x + 0.1$;
```

What happens here is that 0.1 is added to x each time we pass through the loop, and the loop stops when x becomes larger than 1.0. The last time the result may become $1+\epsilon$ rather than 1, where ϵ is some small positive number, and hence the last time through the loop with $x = 1$ may never happen.

In fact, for many floating-point numbers a and b, even the two computations $a + b$ and $b + a$ give different results!

5.2.5 Multiplication and division of floating-point numbers

Multiplication and division of floating-point numbers is straightforward. Perhaps surprisingly, these operations are not susceptible to round-off errors. As for addition, both the procedures for performing the operations, and the effect of round-off error, is most easily illustrated by some examples.

Example 5.15. Consider the two numbers $a = 23.57$ and $b = -6.759$ which in normalised form are

$$a = 0.2357 \times 10^2, \quad b = -0.6759 \times 10^1.$$

To multiply the numbers, we multiply the significands and add the exponents, before we normalise the result at the end, if necessary. In our example we obtain

$$a \times b = -0.15930963 \times 10^3.$$

The significand in the result must be rounded to four digits and yields the floating-point number

$$-0.1593 \times 10^3,$$

i.e., the number −159.3. ∎

Let us also consider an example of division.

Example 5.16. We use the same numbers as in example 5.15, but now we perform the division a/b. We have

$$\frac{a}{b} = \frac{0.2357 \times 10^2}{-0.6759 \times 10^1} = \frac{0.2357}{-0.6759} \times 10^1,$$

i.e., we divide the significands and subtract the exponents. The division yields $0.2357/-0.6759 \approx -0.3487202$. We round this to four digits and obtain the result

$$\frac{a}{b} \approx -0.3487 \times 10^1 = -3.487. \quad \blacksquare$$

The most serious problem with floating-point arithmetic is loss of correct digits. In multiplication and division this cannot happen, so these operations are quite safe.

Observation 5.17. *Floating point multiplication and division do not lead to loss of correct digits as long as the the numbers are within the range of the floating-point model. In the worst case, the last digit in the result may be one digit wrong.*

The essence of observation 5.17 is that the above examples are representative of what happens. However, there are some other potential problems to be aware of.

First of all observation 5.17 only says something about *one* multiplication or division. When many such operations are stringed together with the output of one being fed to the next, the errors may accumulate and become large even when they are very small at each step. We will see examples of this in later chapters.

In observation 5.17 there is one assumption, namely that the numbers are within the range of the floating-point model. This applies to each of the operands (the two numbers to be multiplied or divided) and the result. When the numbers approach the limits of our floating-point model, things will inevitably go wrong.

Underflow. *Underflow* occurs when a positive result becomes smaller than the smallest representable, positive number; the result is then set to 0. This also happens with negative numbers with small magnitude. In most environments you will get a warning, but otherwise the calculations will continue. This will usually not be a problem, but you should be aware of what happens.

Overflow. When the absolute value of a result becomes too large for the floating-point standard, *overflow* occurs. This is indicated by the result receiving the value `infinity` or possibly `positive infinity` or `negative infinity`. There is a special combination of bits in the IEEE standard for these infinity values. When you see `infinity` appearing in your calculations, you should be aware; the chances are high that the reason is some programming error, or even worse, an error in your algorithm. An operation like $a/0.0$ will yield `infinity` when a is a nonzero number.

Undefined operations. The division $0.0/0.0$ is undefined in mathematics, and in the IEEE standard this will give the result `NaN` (not a number). Other operations that may produce `NaN` are square roots of negative numbers, inverse sine or cosine of a number larger than 1, the logarithm of a negative number, etc. (unless you are working with complex numbers). `NaN` is infectious in the sense that any arithmetic operation that combines `NaN` with a normal number always yields `NaN`. For example, the result of the operation 1 + `NaN` will be `NaN`.

5.2.6 The IEEE standard for floating-point arithmetic

On most computers, floating-point numbers are represented, and arithmetic performed according to, the IEEE[1] standard. This is a carefully designed suggestion for how floating-point numbers should behave, and is aimed at providing good control of round-off errors and prescribing the behaviour when numbers reach the limits of the floating-point model. Regardless of the particular details of how the floating-point arithmetic is performed, however, the use of floating-point numbers inevitably leads to problems, and in this section we will consider some of those.

One should be aware of the fact the IEEE standard is extensive and complicated, and far from all computers and programming languages support the full standard. So if you are going to do serious floating-point calculations you should check how much of the IEEE standard that is supported in your environment. In particular, you should realise that if you move your calculations from one environment to another, the results may change slightly because of different implementations of the IEEE standard.

[1] IEEE is an abbreviation for *Institute of Electrical and Electronic Engineers* which is a large professional society for engineers and scientists in the USA. The floating-point standard is described in a document called *IEEE standard reference 754*.

Exercises

1 **a)** Round 1.2386 to 1 digit.

b) Round 85.001 to 1 digit.

c) Round 100 to 1 digit.

d) Round 10.473 to 3 digits.

e) Truncate 10.473 to 3 digits.

f) Round 4525 to 3 digits.

2 Try and describe a rounding rule that is symmetric, i.e., it has no statistical bias.

3 The earliest programming languages would provide only the method of truncation for rounding non-integer numbers. This can lead sometimes lead to large errors as 2.000 - 0.001 = 1.999 would be rounded of to 1 if truncated to an integer. Express rounding a number to the nearest integer in terms of truncation.

4 Use the floating-point model defined in this chapter with 4 digits for the significand and 1 digit for the exponent, and use algorithm 5.8 to do the calculations below in this model.

a) $12.24+4.23$.

b) $9.834+2.45$.

c) $2.314-2.273$.

d) $23.45-23.39$.

e) $1+x-e^x$ for $x=0.1$.

5 **a)** Formulate a model for floating-point numbers in base β.

b) Generalise the rule for rounding of fractional numbers in the decimal system to the octal system and the hexadecimal system.

c) Formulate the rounding rule for fractional numbers in the base-β numeral system.

6 From the text it is clear that on a computer, the addition $1.0+\epsilon$ will give the result 1.0 if ϵ is sufficiently small. Write a computer program which can help you to determine the smallest integer n such that $1.0+2^{-n}$ gives the result 1.0

7 Consider the simple algorithm

```
$x = 0.0$;
while $x \le 2.0$
    print $x$
    $x = x + 0.1$;
```

What values of x will be printed?

Implement the algorithm in a program and check that the correct values are printed. If this is not the case, explain what happened.

8 **a)** A fundamental property of real numbers is given by the distributive law

$$(x+y)z = xz + yz. \tag{5.4}$$

In this problem you are going to check whether floating-point numbers obey this law. To do this you are going to write a program that runs through a loop 10 000 times and each time draws three random numbers in the interval $(0,1)$ and then checks whether the law holds (whether the two sides of (5.4) are equal) for these numbers. Count how many times the law fails, and at the end, print the percentage of times that it failed. Print also a set of three numbers for which the law failed.

b) Repeat (a), but test the associative law $(x+y)+z = x+(y+z)$ instead.

c) Repeat (a), but test the commutative law $x+y = y+x$ instead.

d) Repeat (a) and (b), but test the associative and commutative laws for multiplication instead.

5.3 Measuring the error

In the previous section we saw that errors usually occur during floating point computations, and we therefore need a way to measure this error. Suppose we have a number a and an approximation $\tilde{a}$. We are going to consider two ways to measure the error in such an approximation, the *absolute error* and the *relative error*.

5.3.1 Absolute error

The first error measure is the most obvious, it is essentially the difference between a and $\tilde{a}$.

Definition 5.18 (Absolute error)**.** *Suppose $\tilde{a}$ is an approximation to the number a. The absolute error in this approximation is given by $|a-\tilde{a}|$.*

If $a = 100$ and $\tilde{a} = 100.1$ the absolute error is 0.1, whereas if $a = 1$ and $\tilde{a} = 1.1$ the absolute error is still 0.1. Note that if a is an approximation to $\tilde{a}$, then $\tilde{a}$ is an equally good approximation to a with respect to the absolute error.

5.3.2 Relative error

For some purposes the absolute error may be what we want, but in many cases it is reasonable to say that the error is smaller in the first example above than in the second, since the numbers involved are bigger. The *relative error* takes this into account.

Definition 5.19 (Relative error)**.** *Suppose that $\tilde{a}$ is an approximation to the nonzero number a. The relative error in this approximation is given by*

$$\frac{|a-\tilde{a}|}{|a|}.$$

We note that the relative error is obtained by scaling the absolute error with the size of the number that is approximated. If we compute the relative errors in the approximations above we find that it is 0.001 when $a = 100$ and $\tilde{a} = 100.1$, while when $a = 1$ and $\tilde{a} = 1.1$ the relative error is 0.1. In contrast to the absolute error, we see that the relative error tells us that the approximation in the first case is much better than in the latter case. In fact we will see in a moment that the relative error gives a good measure of the number of digits that a and $\tilde{a}$ have in common.

We use concepts which are closely related to absolute and relative errors in many everyday situations. One example is profit from investments, like bank accounts. Suppose you have invested some money and after one year, your profit is 100 (in your local currency). If your initial investment was 100, this is a very good profit in one year, but if your initial investment was 10 000 it is not so impressive. If we let a denote the initial investment and $\tilde{a}$ the investment after one year, we see that, apart from the sign, the profit of 100 corresponds to the 'absolute error'in the two cases. On the other hand, the relative error corresponds to profit measured in %, again apart from the sign. This says much more about how good the investment is since the profit is compared to the initial investment. In the two cases here, we find that the profit in % is $1 = 100\,\%$ in the first case and $0.01 = 1\%$ in the second.

Another situation where we use relative measures is when we give the concentration of an ingredient within a substance. If for example the fat content in milk is 3.5 % we know that a grams of milk will contain $0.035a$ grams of fat. In this case $\tilde{a}$ denotes how many grams that are not fat. Then the difference $a - \tilde{a}$ is the amount of fat and the equation $a - \tilde{a} = 0.035a$ can be written

$$\frac{a - \tilde{a}}{a} = 0.035$$

which shows the similarity with relative error.

5.3.3 Properties of the relative error

The examples above show that we may think of the relative error as the 'concentration'of the error $|a - \tilde{a}|$ in the total 'volume'$|a|$. However, this interpretation can be carried further and linked to the number of common digits in a and $\tilde{a}$: If the size of the relative error is approximately 10^{-m}, then a and $\tilde{a}$ have roughly m digits in common. If the relative error is r, this corresponds to $-\log_{10} r$ being approximately m.

a	$\tilde{a}$	r
12.3	12.1	1.6×10^{-2}
12.8	13.1	2.3×10^{-2}
0.53241	0.53234	1.3×10^{-4}
8.96723	8.96704	2.1×10^{-5}

Table 5.1. Some numbers a with approximations $\tilde{a}$ and corresponding relative errors r.

Observation 5.20 (Relative error and significant digits)**.** *Let a be a nonzero number and suppose that $\tilde{a}$ is an approximation to a with relative error*

$$r = \frac{|a - \tilde{a}|}{|a|} \approx 10^{-m} \tag{5.5}$$

where m is an integer. Then roughly the m most significant decimal digits of a and $\tilde{a}$ agree.

Sketch of 'proof'. Since a is nonzero, it can be written as $a = \alpha \times 10^n$ where α is a number in the range $1 \leq \alpha < 10$ that has the same decimal digits as a, and n is an integer. The approximation $\tilde{a}$ be be written similarly as $\tilde{a} = \tilde{\alpha} \times 10^n$ and the digits of $\tilde{\alpha}$ are exactly those of $\tilde{a}$. Our job is to prove that roughly the first m digits of a and $\tilde{a}$ agree.

Let us insert the alternative expressions for a and $\tilde{a}$ in (5.5). If we cancel the common factor 10^n and multiply by α, we obtain

$$|\alpha - \tilde{\alpha}| \approx \alpha \times 10^{-m}. \tag{5.6}$$

Since α is a number in the interval $[1, 10)$, the right-hand side is roughly a number in the interval $[10^{-m}, 10^{-m+1})$. This means that by subtracting $\tilde{\alpha}$ from α, we cancel out the digit to the left of the decimal point, and $m - 1$ digits to the right of the decimal point. But this means that the m most significant digits of α and $\tilde{\alpha}$ agree. ∎

Some examples of numbers a and $\tilde{a}$ with corresponding relative errors are shown in Table 5.1. In the first case everything works as expected. In the second case the rule only works after a and $\tilde{a}$ have been rounded to two digits. In the third example there are only three common digits even though the relative error is roughly 10^{-4} (but note that the fourth digit is only off by one unit). Similarly, in the last example, the relative error is approximately 10^{-5}, but a and $\tilde{a}$ only have four digits in common, with the fifth digits differing by two units.

The last two examples illustrate that the link between relative error and significant digits is just a rule of thumb. If we go back to the 'proof'of observation 5.20, we note that as the left-most digit in a (and α) becomes larger, the right-hand side of (5.6) also becomes larger. This means that the number of common digits may become smaller, especially when the relative error approaches 10^{-m+1} as well. In spite of this, observation 5.20 is a convenient rule of thumb.

Observation 5.20 is rather vague when it just assumes that $r \approx 10^{-m}$. The basis for making this more precise is the fact that m is equal to $-\log_{10} r$, rounded to the nearest integer. This means that m is characterised by the inequalities

$$m - 0.5 < -\log_{10} r \leq m + 0.5.$$

and this is in turn equivalent to

$$r = \rho \times 10^{-m}, \quad \text{where } \frac{1}{\sqrt{10}} < \rho < \sqrt{10}.$$

The absolute error has the nice property that if $\tilde{a}$ is a good approximation to a, then a is an equally good approximation to $\tilde{a}$. The relative error has a similar property. It can be shown that if $\tilde{a}$ is an approximation to a with small relative error, then a is also an approximation to $\tilde{a}$ with small relative error.

5.3.4 Errors in floating-point representation

Recall from chapter 3 that most real numbers cannot be represented exactly with a finite number of decimals. In fact, there are only a finite number of floating-point numbers in total. This means that very few real numbers can be represented exactly by floating-point numbers, and it is useful to have a bound on how much the floating-point representation of a real number deviates from the number itself. From observation 5.20 it is not suprising that the relative error is a good tool for measuring this error.

Lemma 5.21. *Suppose that a is a nonzero real number within the range of base-β normalised floating-point numbers with an m digit significand, and suppose that a is represented by the nearest floating-point number $\tilde{a}$. Then the relative error in this approximation is at most*

$$\frac{|a - \tilde{a}|}{|a|} \leq 5\beta^{-m}. \tag{5.7}$$

Proof. For simplicity we first do the proof in the decimal numeral system. We write a as a normalised decimal number,

$$a = \alpha \times 10^n,$$

where α is a number in the range $[0.1, 1)$. The floating-point approximation $\tilde{a}$ can also be written as

$$\tilde{a} = \tilde{\alpha} \times 10^n,$$

where $\tilde{\alpha}$ is α rounded to m digits. Suppose for example that $m = 4$ and $\tilde{\alpha} = 0.3218$. Then the absolute value of the significand $|\alpha|$ of the original number must lie in the range $[0.32175, 0.32185)$, so in this case the largest possible distance between α and $\tilde{\alpha}$ is 5 units in the fifth decimal, i.e., the error is bounded by $5 \times 10^{-5} = 0.5 \times 10^{-4}$. A similar argument shows that in the general decimal case we have

$$|\alpha - \tilde{\alpha}| \le 0.5 \times 10^{-m}.$$

The relative error is therefore bounded by

$$\frac{|\alpha - \tilde{\alpha}|}{|a|} \le \frac{0.5 \times 10^{-m} \times 10^n}{|\alpha| \times 10^n} = \frac{0.5 \times 10^{-m}}{|\alpha|} \le 5 \times 10^{-m}$$

where the last inequality follows since $|\alpha| \ge 0.1$.

Although we only considered normalised numbers in bases 2 and 10 in chapter 4, normalised numbers may be generalised to any base. In base β the distance between α and $\tilde{\alpha}$ is at most $0.5\beta\beta^{-m-1} = 0.5\beta^{-m}$, see exercise 5. Since we also have $|\alpha| \ge \beta^{-1}$, an argument completely analogous to the one in the decimal case proves (5.7). ∎

Note that lemma 5.21 states what we already know, namely that a and $\tilde{a}$ have roughly $m - 1$ or m digits in common.

Exercises

1 Suppose that $\tilde{a}$ is an approximation to a in the problems below, calculate the absolute and relative errors in each case, and check that the relative error estimates the number of correct digits as predicted by observation 5.20.

a) $a = 1$, $\tilde{a} = 0.9994$.

b) $a = 24$, $\tilde{a} = 23.56$.

c) $a = -1267$, $\tilde{a} = -1267.345$.

d) $a = 124$, $\tilde{a} = 7$.

2 Compute the relative error of a with regard to ã for the examples in exercise 1, and check whether the two errors are comparable as suggested in the last sentence in section 5.3.3.

3 Compute the relative errors in examples 5.9–5.12, and check whether observation 5.20 is valid in these cases.

4 The Vancouver stock exchange devised a short-lived index (weighted average of the value of a group of stocks). At its inception in 1982, the index was given a value of 1000.000. After 22 months of recomputing the index and truncating to three decimal places at each change in market value, the index stood at 524.881, despite the fact that its 'true' value should have been 1009.811. Find the relative error in the stock exchange value.

5 In the 1991 Gulf War, the Patriot missile defence system failed due to round-off error. The troubles stemmed from a computer that performed the tracking calculations. The computer's internal clock generated integer values that were converted to real time values by multiplying the integer values by 0.1, with the arithmetic carried out in binary. The approximation which was used for 0.1 was

$$0.1_{10} \approx 0.00011001100110011001100_2 = \frac{209715}{2097152}.$$

a) Find the absolute and relative error in this approximation.

b) The computed time values corresponded to time given in tenths of a second and was therefore calculated 10 times every second by the equivalent of the naive code

```
$c = 209715/2097152$
**while** not ready
    $t = t + i_t * c$
```

where i_t is the integer increment of the internal clock.

Find the accumulated error after 1 hour.

c) Find an alternative algorithm that avoids the accumulation of round-off error.

After the system had been running for 100 hours, an error of 0.3433 seconds had accumulated. This discrepancy caused the Patriot system to continuously recycle itself instead of targeting properly. As a result, an Iraqi Scud missile could not be targeted and was allowed to detonate on a barracks, killing 28 people.

5.4 Rewriting formulas to avoid rounding errors

Certain formulas lead to large rounding errors when they are evaluated with floating-point numbers. In some cases though, the result can be evaluated with an alternative formula that is not problematic. In this section we will consider some examples of this.

Example 5.22. Suppose we are going to evaluate the expression

$$\frac{1}{\sqrt{x^2+1}-x} \tag{5.8}$$

for a large number like $x = 10^8$. The problem here is the fact that x and $\sqrt{x^2+1}$ are almost equal when x is large,

$$\begin{aligned} x &= 10^8 = 100000000, \\ \sqrt{x^2+1} &\approx 100000000.000000005. \end{aligned}$$

Even with 64-bit floating-point numbers the square root will therefore be computed as 10^8, so the denominator in (5.8) will be computed as 0, and we get division by 0. This is a consequence of floating-point arithmetic since the two terms in the denominator are not really equal. A simple trick helps us out of this problem. Note that

$$\frac{1}{\sqrt{x^2+1}-x} = \frac{(\sqrt{x^2+1}+x)}{(\sqrt{x^2+1}-x)(\sqrt{x^2+1}+x)} = \frac{\sqrt{x^2+1}+x}{x^2+1-x^2} = \sqrt{x^2+1}+x.$$

This alternative expression can be evaluated for large values of x without any problems with cancellation. The result for $x = 10^8$ is

$$\sqrt{x^2+1}+x \approx 200000000$$

where all the digits are correct. ∎

The fact that we were able to find an alternative formula in example 5.22 may seem very special, but it it is not unusual. Here is another example.

Example 5.23. For most values of x, the expression

$$\frac{1}{\cos^2 x - \sin^2 x} \tag{5.9}$$

can be computed without problems. However, for $x = \pi/4$ the denominator is 0, so we get division by 0 since $\cos x$ and $\sin x$ are equal for this value of x. This means that when x is close to $\pi/4$ we will get cancellation. This can be avoided by noting that $\cos^2 x - \sin^2 x = \cos 2x$, so the expression (5.9) is equivalent to

$$\frac{1}{\cos 2x}.$$

This can be evaluated without problems for all values of x for which the denominator is nonzero, as long as we do not get overflow. ∎

Exercises

1 Identify values of x for which the formulas below may lead to large round-off errors, and suggest alternative formulas which do not have these problems.

a) $\sqrt{x+1}-\sqrt{x}$.

b) $\ln x^2 - \ln(x^2+x)$.

c) $\cos^2 x - \sin^2 x$.

2 Suppose you are going to write a program for computing the real roots of the quadratic equation $ax^2+bx+c=0$, where the coefficients a, b and c are real numbers. The traditional formulas for the two solutions are

$$x_1 = \frac{-b-\sqrt{b^2-4ac}}{2a}, \quad x_2 = \frac{-b+\sqrt{b^2-4ac}}{2a}.$$

Identify situations (values of the coefficients) where the formulas do not make sense or lead to large round-off errors, and suggest alternative formulas for these cases which avoid the problems.

3 The binomial coefficient $\binom{n}{i}$ is defined as

$$\binom{n}{i} = \frac{n!}{i!\,(n-i)!} \tag{5.10}$$

where $n \geq 0$ is an integer and i is an integer in the interval $0 \leq i \leq n$. The binomial coefficients turn up in a number of different contexts and must often be calculated on a computer. Since all binomial coefficients are integers (this means that the division in (5.10) can never give a remainder), it is reasonable to use integer variables in such computations. For small values of n and i this works well, but for larger values we may run into problems because the numerator and denominator in (5.10) may become larger than the largest permissible integer, even if the binomial coefficient itself may be relatively small. In many languages this will cause overflow, but even in a language like Python, which on overflow converts to a format in which the size is only limited by the resources available, the performance is reduced considerably. By using floating-point numbers we may be able to handle larger numbers, but again we may encounter too big numbers during the computations even if the final result is not so big.

An unfortunate feature of the formula (5.10) is that even if the binomial coefficient is small, the numerator and denominator may both be large. In general, this is bad for numerical computations and should be avoided, if possible. If we consider the formula (5.10) in some more detail, we notice that many of the numbers cancel out,

$$\binom{n}{i} = \frac{1\cdot 2\cdots i\cdot(i+1)\cdots n}{1\cdot 2\cdots i\cdot 1\cdot 2\cdots(n-i)} = \frac{i+1}{1}\cdot\frac{i+2}{2}\cdots\frac{n}{n-i}.$$

Employing the product notation we can therefore write $\binom{n}{i}$ as

$$\binom{n}{i} = \prod_{j=1}^{n-i}\frac{i+j}{j}.$$

a) Write a program for computing binomial coefficients based on this formula, and test your method on the coefficients

$$\binom{9998}{4} = 416083629102505,$$

$$\binom{100000}{70} = 8.14900007813826 \cdot 10^{249},$$

$$\binom{1000}{500} = 2.702882409454366 \cdot 10^{299}.$$

Why do you have to use floating-point numbers and what results do you get?

b) Is it possible to encounter too large numbers during those computations if the binomial coefficient to be computed is smaller than the largest floating-point number that can be represented by your computer?

c) In our derivation we cancelled $i!$ against $n!$ in (5.10), and thereby obtained the alternative expression for $\binom{n}{i}$. Another method can be derived by cancelling $(n-i)!$ against $n!$ instead. Derive this alternative method in the same way as above, and discuss when the two methods should be used (you do not need to program the other method; argue mathematically).