Instructors: Prof. Markus Hegland

Mathematical Sciences Institute

Australian National University

First Semester 2017

Contents

1 Floating point numbers

- number representation for all the types and examples in python, binary and decimal
- real numbers: fixpoint and floating point, IEEE standard, representation, single and double precision, illustration: fixpoint numbers and floating point numbers – magnified, self-similarity, normalisation and non-uniqueness, non-normalised numbers and NaNs, infinities and zeros
- going outside of the range: over and underflow
- reading and writing numbers
- rounding and truncation – the rounding function
- floating point operations
  - multiplication
  - addition and subtraction
  - division
- an optimal floating point model
- errors: absolute and relative errors, cancellation, machine epsilon, also the floating point model

1.1 Real numbers and arithmetic

Real numbers are a very convenient construction used to model time, distance, volume and mass, probability, force, value, risk, temperature, parameters and many other things. They are computationally versatile as the laws governing their arithmetic allow us to rearrange computations and thus reduce computational costs. Most importantly, with real numbers we can solve a large variety of equations. They also are the foundation of calculus. However, only a very small subset of all real numbers can be implemented on a computer. Consequently, one is faced with approximation and errors in almost every computation performed on a digital computer. The analysis of these so-called rounding errors is the topic of this section.

There are several other types of numbers. In computing one frequently uses integers which model counts of items. In computing they are used for indexing arrays and for the enumeration of loops. Only a limited number of equations can be solved using integers, an example is

$$x + 4 = 5.$$

A system with two equations is

$$\begin{aligned} x + y + 3 &= 10 \\ y + 2 &= 5. \end{aligned}$$

Basically, the solution of these systems involves only subtractions and additions.

Project: Characterise classes of linear systems of equations with integer matrix and right-hand side which do not require any divisions for their solution. Comment on pivoting, the elementary matrices and the matrix factorisations involved. *Hint*: start with the examples above.

When one considers equations like

$$5x = 13$$

or general linear systems of equations the solutions are typically not integer even if the coefficients are all integers. For these problems one requires rational numbers. They are represented as fractions p/q which are pairs (p, q) of integers. Using only the arithmetic operations for integers one can define all the operations on fractions. Note that the representation is not unique, for example $2/4$ and $1/2$ denote the same fraction. Furthermore, the numerators and denominators tend to increase in size with the number of arithmetic operations performed. An efficient implementation will have to reduce the fractions regularly by dividing numerator and denominator by their greatest common divisor (gcd) regularly. Computing the gcd can be done with Euclid's algorithm. However, rational numbers are most suited for linear systems of equations with relatively few unknowns.

Project: Discuss an implementation of rational numbers in Python. *Hint*: An abstract class is available in Python 3.4. There is also a module for fractions.

Note that both integers and rational numbers are examples of real numbers. Also recall that not all systems of linear equations do have a solution. However, if a solution exists and if the coefficients and right-hand side is rational then so is the solution.

The next class of equations are the algebraic (or polynomial) equations like

$$4x^2 + 5x - 7 = 0.$$

For their solution one requires complex numbers and it is known that a polynomial of degree n has at least one solution in the set of complex numbers. Complex numbers are implemented as pairs of real numbers (x, y) where x and y are the real and imaginary part, respectively. In computations one either uses a predefined complex type or implements complex numbers as pairs of reals. In addition to the arithmetic operations one also has a conjugation operation which maps any complex number onto its conjugate complex. Note that there is a unique correspondence between complex numbers and the pairs.

Project: Implement a complex data type with all the algebraic operations and conjugation. *Hint*: complex numbers are available in Python.

In the following we will now discuss some properties of real numbers which are important for computational science. All the numerical algorithms can be boiled down to a sequence of arithmetic operations with real numbers and decisions based on comparisons between real numbers. We usually denote the set of all real numbers by $\mathbb{R}$ and real numbers are denoted by letters $x, y, z \in \mathbb{R}$ for example.

A fundamental property of real numbers is that one can add, subtract and multiply any two real numbers and the result is again a real number. Furthermore, the division of any real number by any nonzero real number is also a real number. Division by zero does never provide a real number. Furthermore, there are two special real numbers, zero which when added to any number returns the same number again and one, which when multiplied with any number does not change the factor. We note these properties here especially as the real number systems of computers do not satisfy them.

One now has $x - x = 0$ and $x/x = 1$ (for $x \neq 0$) so that one can define subtraction by taking adding the negative as

$$x - y = x + (-y)$$

and similarly for division

$$x/y = x * (1/y).$$

Thus one can reduce subtraction to addition if one has an effective way of computing $-y$ for any y.

Both addition and multiplication satisfy the associative law, for addition

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

and similarly for multiplication

$$(x * y) * z = x * (y * z).$$

They also satisfy the commutative law, for addition

$$x + y = y + x$$

and similarly for multiplication

$$x * y = y * x.$$

Finally, one has the distributive law which connects both addition and multiplication by

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

These fundamental laws of arithmetic are very important in developing algorithms especially for simplifying and making the algorithms more efficient. This can be seen directly for the distributive law where evaluation of the left hand side requires one addition and one multiplication while the right-hand side requires one addition and two multiplications which will take more time. While applying these laws can have a substantial effect on the time computations take we will see that they often do not hold for numerical computations and the order in which computations are done (relating to the associative law) in particular can lead to different results!

The real numbers are ordered and for any two numbers are either equal or one of them is larger than the other. This is particularly important when making decisions. Furthermore, if one takes two real numbers $x < y$ then there is always a third real number z which is in-between such that $x < z < y$. Arguably the most complex but equally important property of real numbers is that any Cauchy sequence of real numbers converges to a real number. Recall that a sequence x_n is a Cauchy sequence if for any $\epsilon > 0$ there exists an m such that for all $i, j \geq m$ one has

$$|x_i - x_j| \leq \epsilon.$$

Note that the absolute value $|x|$ of a real number is zero if and only if $x = 0$ and that the triangle inequality holds

$$|x + y| \leq |x| + |y|$$

and that

$$|x * y| = |x| * |y|.$$

These properties are important when we would like to assess the accuracy of algorithms.

1.2 Number representation

Integers

For hand calculations we use the decimal system. In the decimal system every integer is written as a polynomial with coefficients between zero and nine evaluated at ten. For example, one has

$$3067 = 3 * 10^3 + 0 * 10^2 + 6 * 10 + 7.$$

This representation is unique and the polynomial coefficients, called digits, are determined by repeatedly taking the remainder of a division by 10 and dividing by 10. In the *binary system* used

on most computers nowadays one uses a polynomial representation with coefficients zero and one and evaluating the polynomial at two. Again, be repeated division by two and computing the remainder one gets the binary digits or bits of the integer. For our example we have

$$\begin{aligned} 3067 &= 2 * 1533 + 1_2 = 4 * 766 + 11_2 = 8 * 383 + 11_2 = 16 * 191 + 1011_2 \\ &= 32 * 95 + 11011_2 = 64 * 47 + 111011_2 = 128 * 23 + 1111011_2 = 256 * 11 + 11111011_2 \\ &= 512 * 5 + 111111011_2 = 1024 * 2 + 1111111011_2 = 101111111011_2 \end{aligned}$$

Here we write the base B as subscript (omitting it if $B = 10$) so that $11_2 = 3_{10} = 3$ etc. If we rewrite the above sum as polynomial (omitting the terms with zero coefficients) we have

$$3067 = 2^{11} + 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2 + 1.$$

So far we have considered the representation of positive integers. Negative integers can be obtained by adding one bit for the sign. Note that in this way all integers have a unique representation except for zero which can be represented as either plus zero or minus zero. With this we get a representation for any integer of the form

$$x = \sum_{k=0}^{n} c_k B^k$$

to the base B (which could be any integer) and coefficients (digits) $c_k \in \{0, \ldots, B-1\}$.

Project: Write Python code which takes an integer (string) in decimal format and outputs a string of zeros and ones providing the binary format of the same number. Then, rewrite your code such that it can take arbitrary bases B and test this for $B = 16$ and $B = 256$. *Hint:* The values $B = 16$ and $B = 256$ are used in digital communication addressing for ethernet numbers and ip-numbers, respectively. For $B = 16$ one uses the symbols $\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}$ for the hexadecimal digits for $10, 11, 12, 13, 14, 15$ and writes the integers e.g. as $29_{10} = 1\mathrm{e}_{16}$. In the case of $B = 256$ this is not possible any more. Here the digits are written as decimal numbers and separated by dots. For example, one has here

$$290_{10} = 1.34_{256} \quad \text{and} \quad 65543_{256} = 1.0.7_{256}.$$

The application of the base 256 is used for so-called IP numbers and the (internal) IP number of "localhost", which is your local workstation or server is typically 127.0.0.1 which corresponds to $2,130,706,433_{10}$ (over 2 billion). This number is right in the middle of the domain of numbers with 4 digits to base 256.

One can now do additions, subtractions and multiplications with the numbers using their positional or polynomial representation just as if they were polynomials. However, in doing these operations the condition that the coefficients of the polynomials (the digits) are between zero and $B-1$ may no longer hold. For example, one has

$$34 + 78 = (3 * 10 + 4) + (7 * 10 + 8) = 10 * 10 + 12$$

so the "digits" of this product are now 10 and 12. One can easily check that the result is correct but the condition that the digits are less than 10 is violated. This is where one uses "carry-over" and in this case starts normalising the result from the lowest digit to get

$$34 + 78 = 10 * 10 + 12 = 10 * 10 + 10 + 2 = 11 * 10 + 2 = 1 * 100 + 1 * 10 + 2 = 112.$$

When we do these operations by hand we interleave the polynomial operations with the carry-over step but in principle both these steps have to be done. In the case of multiplication of

very large numbers with relatively large bases algorithms may actually exploit the fact that a polynomial multiplication is done by using fast polynomial multiplication with the help of fast Fourier transforms.

Representation of real numbers

For real numbers we often use a decimal representation like 15.743. Here 15 is the integer part of the real number and 0.743 is the fractional part. We have seen how to represent integers as polynomials in the base so it remains to represent the fractional part. Any real number is then the sum of the integer part and the fractional part and can be written as

$$x = \sum_{k=0}^{n} a_k B^k + \sum_{k=1}^{\infty} c_k B^{-k}.$$

Using this for define real arithmetic turns out to be somewhat difficult: One first requires the arithmetic for both the integer and fractional parts and in addition will require some sort of carry between the two parts.

A simpler representation is obtained by using the scientific notation of the form

$$x = \pm B^m \sum_{k=1}^{\infty} c_k B^{-k}.$$

We will use this representation in the following. Here the integer m is called the exponent and the infinite sum is the mantissa. As the digits c_k range from 0 to $B-1$ and the $B > 1$ the infinite series converges like a geometric series. Thus the main mathematical concept used here is an infinite series with bounded coefficients which is evaluated at the base B.

While it is possible to represent every real number in this way, the representation is not unique in two ways.

- First one has for any positive integer n

  $$x = B^{m+n} \sum_{k=n+1}^{\infty} c_{k-n} B^{-k}.$$

  Thus x also has the exponent $m+n$ and the digits c_{k-n} which are the same as before shifted by n positions to the right and filling in (or padding) the missing digits with 0.

- Second we have

  $$1 = \sum_{k=1}^{\infty} (B-1) B^{-k}$$

  so that the number one is represented both

  - as having exponent zero and a mantissa with all (infinite) digits being $B-1$ or
  - as $B * 0.1_B$ with exponent 1 and mantissa 0.1_B (subscript again denotes the underlying system, i.e., for decimal numbers $B = 10$).

Uniqueness is now achieved by selecting one particular representation and calling it *normalised*. We will always represent zero as having mantissa zero and all digits zero and having a positive sign. Then for any nonzero real number we choose the representation which has a first digit $c_1 \neq 0$. This uniquely defines the exponent except in the case where all digits $c_k = B-1$ for $k > n$ for some positive integer n. For this number we will call the representation which terminates after finitely many digits the normalised representation.

When defining the arithmetic one uses both normalised and un-normalised representations. For multiplication (or division) one adds (or subtracts) the exponents and uses multiplications (or

divisions) of the power series in B^{-1} combined with carry operations. At the end one might have an un-normalised representation of the result which then can be normalised.

For addition one first represents the term with the smaller exponent with in an un-normalised way such that both terms have the same exponent. Then one adds the two power series by adding all the coefficients and performs all necessary carry operations. Finally one normalises the result. Subtraction is easily defined by first changing the sign of the term to be subtracted and then adding the result to the first term.

In summary, we see that computer numbers and arithmetic as we will use it is based on two fundamental concepts

- operations with polynomials and power series
- a carry function.

These concepts are mostly algebraic but for the power series we require some calculus.

While this might all sound a bit complicated, this is actually close to what we usually do when we perform arithmetic by hand with $B = 10$. Computers do something very similar but instead of the decimal system they use the binary system with $B = 2$. However, like us, computers can only deal with finitely many digits in general. The consequences and corresponding system of numbers used by computers will be discussed in the remaining sections.

Floating point numbers

Floating point numbers are the real numbers implemented on a computer. They are represented using the binary system. They are given by their representation which has to be finite.

Lets first consider a system of integers with n digits so that they have the form

$$i = \pm \sum_{k=0}^{n-1} c_k B^k.$$

and $c_k \in \{0, \ldots, B-1\}$. The largest integer representable with n digits is thus

$$B^n - 1 = \sum_{k=0}^{n-1} (B-1) B^k.$$

Adding the negative ones we can represent a total of $2B^n - 1$ integers (0 is represented twice).

So consider a system with 64 bits words. Then one word represents the sign plus 63 binary digits so that $n = 63$ and the maximal number would be $2^{63} - 1 = 9.22\ldots * 10^{18}$. These integer representations are used for the exponent of floating point numbers.

Project: Determine in Python the largest and smallest integer represented. Lookup the documentation. What do you observe? Comment. Explore how Python's representation differs from ours.

Project: Determine the range of the exponent for Python floating point numbers.

If we now have a first (fictional) floating point number system with infinitely many digits for the mantissa but with finite exponents ranging from $m_1 = -B^n + 1$ to $m_2 = B^n - 1$ the largest positive number (un-normalised) we can represent is

$$x_{\max} = B^{m_1} \sum_{k=1}^{\infty} (B-1) B^{-k} = B^{m_1}.$$

For this largest number we do not have a normalised representation in our system. If we allow un-normalised numbers we can then represent all numbers in the interval $[-x_{\max}, x_{\max}]$ in this way.

If we would allow only normalised numbers then there is actually a smallest number larger than zero which is representable which is

$$x_{\min} = B^{m_2} B^{-1} = B^{m_2 - 1}.$$

So the set of all fictional floating point numbers which are normalised is the set

$$(-x_{\max}, -x_{\min}] \cup \{0\} \cup [x_{\min}, x_{\max}).$$

Note how the constraint of only allowing normalised numbers degrades the accuracy around zero.

The actual floating point numbers implemented on the computer have only a finite number of digits in the mantissa. Thus the computer numbers have the representation

$$x = \pm B^m \sum_{k=1}^{n} c_k B^{-k}$$

where n is now the length (number of digits) of the mantissa and m is the exponent which is an integer between m_1 and m_2. We can now again determine the largest and smallest floating point number.

Consider first the "un-normalised" case. Here the mantissa x_m is a multiple of B^{-n}, i.e.,

$$x_m = jB^{-n}, \quad j = 0, 1, \ldots, B^n - 1.$$

The largest number is then $x_{\max} = B^{m_1}(1 - B^{-n})$ and the smallest number larger than zero is B^{-n}. With this representation we get the following numbers

$$\begin{aligned}
&x = 0 \\
&x = \pm jB^{m_2 - n}, && j = 1, \ldots, B^n - 1 \\
&x = \pm jB^{m_2 + 1 - n}, && j = 1, \ldots, B^n - 1 \\
&\vdots \\
&x = \pm jB^{m_1 - n}, && j = 1, \ldots, B^n - 1.
\end{aligned}$$

You can see that the ranges lower in this table overlap with the upper ones. While in fact each range defined on a line extends the total range, only $(B-1)/B$ of all the numbers included are new. A unique representation is then obtained by choosing

$$\begin{aligned}
&x = 0 \\
&x = \pm jB^{m_2 - n}, && j = 1, \ldots, B^n - 1 \\
&x = \pm jB^{m_2 + 1 - n}, && j = B^{n-1}, \ldots, B^n - 1 \\
&\vdots \\
&x = \pm jB^{m_1 - n}, && j = B^{n-1}, \ldots, B^n - 1.
\end{aligned}$$

We see that the floating point numbers are equidistant in each of the intervals $[B^{m-1}, B^m]$ and also on $[0, B^{m_2}]$. We can rewrite the numbers in terms of their mantissa as

$$
\begin{aligned}
&x = 0 \\
&x = \pm x_m B^{m_2}, && x_m = B^{-n}, 2B^{-n}, \ldots, 1 - B^{-n} \\
&x = \pm x_m B^{m_2+1}, && x_m = B^{-1}, B^{-1} + B^{-n}, \ldots, 1 - B^{-n} \\
&\vdots \\
&x = \pm x_m B^{m_1}, && x_m = B^{-1}, B^{-1} + B^{-n}, \ldots, 1 - B^{-n}.
\end{aligned}
$$

While the actual storage of the floating point numbers may differ, the numbers themselves are usually the ones given above. In this course we are interested in understanding the effect of using a finite set of floating point numbers to do computations has on the accuracy of the result. Different choices of the size of the exponents and mantissa are used but most computers implement an IEEE standard. Even in this standard one has two different types of the numbers, the single and double precision numbers which use 32 and 64 bits, respectively.

Project: Look up the IEEE standard. What are the sizes of mantissa and exponents?

1.3 Rounding, floating point operations and an ideal model

As discussed previously, the numbers represented on a computer are binary fractions. An example is 1/8 which is 0.125 in the decimal system and 0.001_2 in the binary system. Like in the decimal system, all real numbers can be represented by an infinite binary fraction. Computers can of course only work with finite binary fractions where only a finite number of binary digits are not zero. While any binary fraction can be represented by a (finite) decimal fraction like in the case of the example 1/8 not all decimal fractions are also binary fractions. This is because 1/5 is not a binary fraction. Indeed, if it was, one could convert it to binary by evaluating the quotient and 1.0/5.0 and displaying the result as a decimal fraction (which can be done exactly). If one does this using the decimal module in Python one gets

$$1.0/5.0 = 0.2000000000000000011102230246251565404236316680908203125.$$

Clearly, the computer was not able to compute the quotient exactly. The infinite binary representation is

$$\frac{1}{5} = 0.00\overline{1100}_2$$

where the overline means that the digits 1100 are repeated infinitely. One can see that this is true by multiplying this value $\frac{1}{5}$ and from the result subtracting $\frac{1}{5}$ which gives 1100_2 which is just 3. While we often assume that the computer represents the decimal numbers exactly it does not and occasionally one can get problems because of that.

rounding function

Here we consider optimal rounding functions $\phi : \mathbb{R} \to \mathbb{F}$ only which satisfy

$$|\phi(x) - x| \leq |y - x| \quad \text{for all } y \in \mathbb{F}.$$

An obvious but important consequence of this property is that

$$\phi(x) = x, \quad \text{for all } x \in \mathbb{F}.$$

There are multiple optimal rounding functions and we will choose the for which the least significant digit (the last digit to the right) is even. Here $\mathbb{F}$ is a closed set (in particular $\mathrm{Fix}(B, t)$ or $\mathrm{Fl}(B, t)$) of real numbers.

examples

1. fixed point decimal numbers in $\mathrm{Fix}(10,2)$: $\phi(3.452) = 3.45$ and $\phi(0.675) = 0.68$ and $\phi(1/9) = 0.11$
2. fixed point binary number in $\mathrm{Fix}(2,3)$: $x = 3.1875 = 11.0011_2$ and so $\phi(11.0011_2) = 11.010_2 = 3.25$
3. floating point decimal numbers in $\mathrm{Fl}(10,2)$: $\phi(3.452) = 3.5$ and $\phi(0.67) = 0.68$ and $\phi(1/9) = 0.11$
4. floating point binary number in $\mathrm{Fl}(2,3)$: $x = 3.1875 = 11.0011_2$ and so $\phi(11.0111_2) = 11.1_2 = 3.5$

In Python one has the following rounding function for $\mathrm{Fix}(10,t)$ where digits $= t$.

```
Type:        builtin_function_or_method
String form: <built-in function round>
Namespace:   Python builtin
Docstring:
round(number[, ndigits]) -> floating point number

Round a number to a given precision in decimal digits
(default 0 digits). This always returns a floating point
number. Precision may be negative.

Example:

In [1]: round(3.455, ndigits=2)
Out[1]: 3.46
```

We can now get the rounding function ϕ in $\mathrm{Fix}(B,t)$ from a rounding function ϕ_0 for integers $\mathbb{Z} = \mathrm{Fix}(B,0)$ from the following result:

Lemma 1. *if $\phi_0 : \mathbb{R} \to \mathbb{Z}$ and $\phi : \mathbb{R} \to \mathrm{Fix}(B,t)$ rounding functions then*

$$\phi(x) = \phi_0(B^t x)/B^t$$

$$\text{for proof use:} \quad \mathbb{Z} = \{B^t y \mid y \in \mathrm{Fix}(B,t)\}$$

Using this result we can now code a rounding function for any base B which uses rouding ϕ_0 for integers (called round in Python):

```
In [1]: def roundB(number, ndigits=0, base=10):
   ...:     return round(base**ndigits*number)/base**ndigits
   ...:
In [2]: roundB(3.1875, ndigits=3, base=2)
Out[2]: 3.25
```

Using this previous lemma, one can now also get a bound for the rounding error for fixed point arithmetic which uses abound for rounding to integers:

$$|B^t\phi(x) - B^t x| = |\phi_0(B^t x) - B^t x| \leq 0.5$$

Proposition 1. *For all $x \in \mathrm{Fix}(B,t)$ on has*

$$|\phi(x) - x| \leq 0.5B^{-t}$$

Examples

1. fixed point decimal numbers in $\mathrm{Fix}(10, 2)$:

$$|\phi(0.675) - x| = |0.68 - 0.675| = 0.5 * 10^{-2}$$

2. fixed point binary number in $\mathrm{Fix}(2, 3)$:

$$|\phi(3.1875) - 3.1875| = |3.25 - 3.1875| = 0.0675 = 0.5 * 2^{-3}$$

For floating point arithmetic we now get a similar result for the rounding errors:
Lemma 2. *if* $\phi_0 : \mathbb{R} \to \mathbb{Z}$ *and* $\phi : \mathbb{R} \to \mathrm{Fl}(B, t)$ *rounding and* e *exponent of* x *then*

$$\phi(x) = \phi_0(B^{t-e}x)/B^{t-e}$$

for proof use: $\{B^{t-1}, \ldots, B^t - 1\} = \{B^{t-e}y \mid y \in \mathrm{Fl}(B, t)\}$

Rounding in $\mathrm{Fl}(2, t)$

In Python, one can get the binary representation of floating point numbers and use this to implement a rounding function on $\mathrm{Fl}(2, t)$ for any positive integer t as: (Of course t needs to be smaller than the number of bits used by the computer.)

```
In [1]: def roundfl2(number, ndigits=1):
   ...:         import math
   ...:         (xm, xe) = math.frexp(number)
   ...:         xr = round(xm*2.0**ndigits)/2.0**ndigits
   ...:         return math.ldexp(xr, xe)

In [2]: roundfl2(3.1875, ndigits=4)
Out[2]: 3.25
```

In the following figures, we have the rounding errors for fixed point arithmetic.

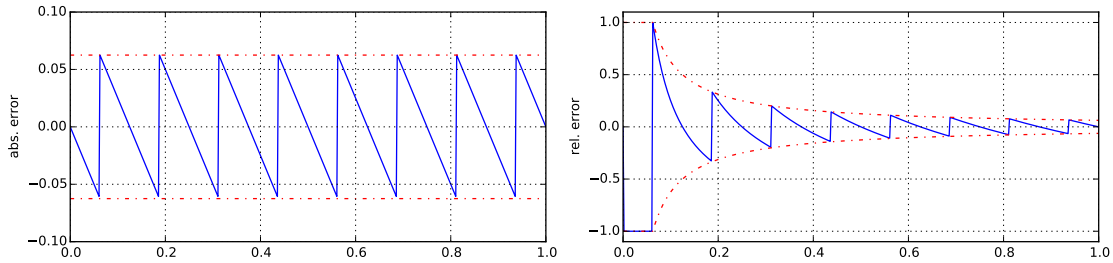

Figure 1: rounding error for $\mathrm{Fix}(2, 3)$

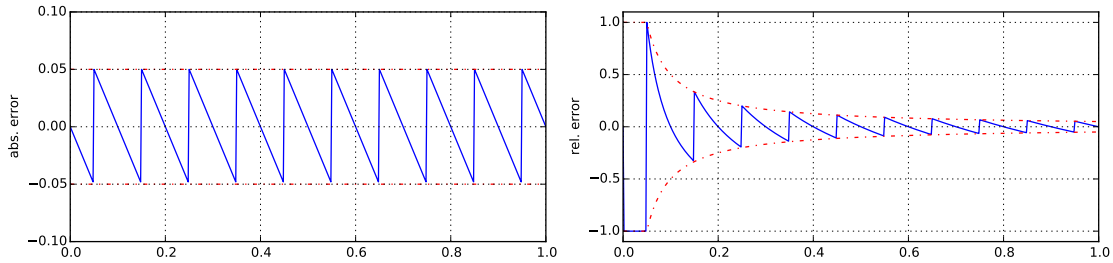

Figure 2: rounding error for $\mathrm{Fix}(10, 1)$

For base 10 we can use the Python Decimal module for rounding. This module implements decimal floating point numbers but we just use it for rounding. Note that if we output the results of any decimal numbers we get an extra error through rounding to the binary system. The following illustrates rounding in $\mathrm{Fl}(10, t)$ using the Decimal Python module.

```
In [1]: def roundfl10(x, t=1):
   ...:     from decimal import Context
   ...:     return float(Context(prec=t).create_decimal(x))
   ...:

In [2]: roundfl10(3.1875, ndigits=3)
Out[2]: 3.19
```

We have now a bound for the rounding error in floating point arithmetic:
Proposition 2. *For all $x \in \mathrm{Fl}(B, t)$ with exponent e on has an absolute error of*

$$|\phi(x) - x| \leq 0.5B^{-t+e}$$

and a relative error of

$$\frac{|\phi(x) - x|}{|x|} \leq 0.5B^{-t+1}$$

This can be shown by observing:

- use $|B^{t-e}\phi(x) - B^{t-e}x| = |\phi_0(B^{t-e}x) - B^{t-e}x| \leq 0.5$
- x normalised, thus $|x| \geq B^{-1}$ for rel. error

The result is confirmed for the following example:

$$|\phi(15.342) - 15.342|/15.342| = |15.3 - 15.342|/15.342 = 0.0027 \leq 0.5 * 10^{-2}$$

We have now computed and plotted the exact rounding errors for floating point arithmetic. As before, we have chosen t such that the case of basis 2 and basis 10 are comparable. See the relevant figures.

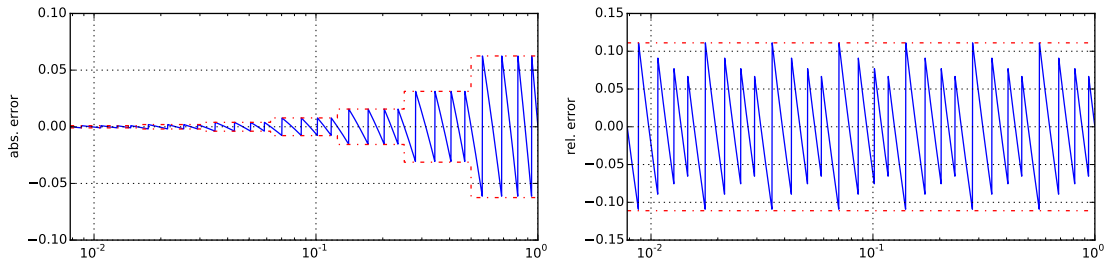

Figure 3: rounding error for $\mathrm{Fl}(2, 3)$

implementation of arithmetic operations in $\mathbb{F}$

In the real numbers one can do any arithmetic operations (except for division by zero) and always gets a real number back. This is usually not the case for $\mathbb{F}$. While the operations produce a real number this number is frequently not in $\mathbb{F}$ any more. The IEEE 754 standard suggests that floating point arithmetic should be implemented such that it returns the best possible element in $\mathbb{F}$ as the result of any arithmetic operation. So if $x \circ_{\mathbb{F}} y$ is the result in $\mathbb{F}$ returned by the arithematical operation $\circ_{\mathbb{F}}$ in $\mathbb{F}$ one has to have

$$x \circ_{\mathbb{F}} y = \phi(x \circ y).$$

So one gets for all $x, y \in \mathbb{F}$:

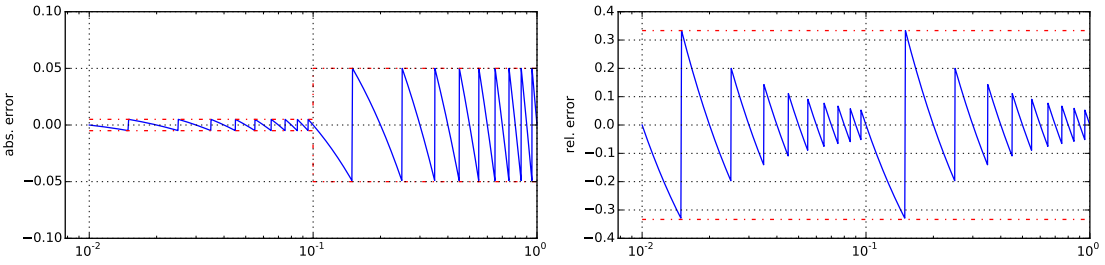

Figure 4: rounding error for $\mathrm{Fl}(10, 1)$

$$x +_{\mathbb{F}} y = \phi(x + y)$$
$$x -_{\mathbb{F}} y = \phi(x - y)$$
$$x *_{\mathbb{F}} y = \phi(x * y)$$
$$x /_{\mathbb{F}} y = \phi(x/y)$$

We will thus in the following not use the notation $x +_{\mathbb{F}} y$ etc anymore and use just $\phi(x + y)$ instead. A consequence of this is that the rounding error occurring for any operation is equal to the rounding error of the exact result.

We now have the following properties (where ϕ is always supposed the be the one used for the corresponding set $\mathbb{F}$):

- For $\mathrm{Fix}(B, t)$ all sums and differences are exact, i.e.,

$$\phi(x \pm y) = x \pm y, \quad x, y \in \mathrm{Fix}(B, t).$$

- For $\mathrm{Fix}(B, t)$ the differences are exact if x and y are sufficiently close
- The products of two numbers $x * y$ from $\mathrm{Fix}(B, t)$ or $\mathrm{Fl}(B, t)$ are in $\mathrm{Fix}(B, 2t)$ or $\mathrm{Fl}(B, 2t)$, respectively
- Quotients are typically fixed point numbers or floating point numbers with infinitely many nonzero digits.

If one compares the properties of the exact real arithmetic and the arithmetic defined by the rounded results, one finds that many common arithmetic lows to not hold exactly for the rounded operations. There is one big exception which is easy to show: the commutative law which is

$$\phi(x * y) = \phi(y * x)$$

holds because it holds for real numbers. The same is true for addition. Otherwise one does not have much luck: The associative law does usually not hold. Typically, one has

$$\phi(x + \phi(y + z)) \neq \phi(\phi(x + y) + z).$$

For example one has in $\mathrm{Fl}(10, 3)$

$$\phi(\phi(1.32 + 0.254) + 0.392) = 1.96 \neq 1.97 = \phi(1.32 + \phi(0.254 + 0.392)).$$

The same is true of the distributive law which is also violated by rounding.

Rounding, however, does have some nice properties:

- It is monotone, i.e., it maintains order:

$$x \leq y \to \phi(x) \leq \phi(y).$$

- The function ϕ is not continuous but piecewise constant.

Simple functions

IEEE 754 also requires that simple functions like exp, sin etc are implemented such that they are the rounded version of the exact function. Thus our computer version of the sine function which could be called $\sin_{\mathbb{F}}$ is defined as

$$\sin_F(x) = \phi(\sin(x))$$

etc. Again, the errors in function evaluation are thus the same as the rounding error.

Effect of the finite range

In the above we have neglected the effect of the finite range of the numbers. This causes errors which are called overflow (when the absolute values of the numbers are too large) or underflow (when the absolute values of the numbers are too close to zero (which can only happen for floating point numbers). The effects are simpler to understand than the effects of rounding and will not be discussed here.

1.4 Error analysis

In the following we will go through the steps to get an error bound for a numerical assignment. While this is a bit involved, the approach is relatively straight forward. We will analyse the errors in both fixed point and in floating point arithmetic. The discussion will be guided by a simple example of evaluating the expression

$$\sin(a_1 a_2) + a_3$$

where the parameters are $a_1 = 3.57$, $a_2 = 0.0723$ and $a_3 = 1.0$. In Python one gets the result

```
>>> sin(3.57*0.0723) + 1.0
1.2552545836362972
```

We will now study how to estimate the accuracy of this result. We will consider both fixed point and floating point numbers and use the generic term machine number for both.

First we make some assumptions (which are supported by the IEEE floating point standard):

- Any data a_k which is input will be rounded to the nearest machine number $\bar{a}_k$ before further processing:
$$\bar{a}_k = \phi(a_k)$$
- All arithmetic operations and simple functions acting on machine numbers are implemented optimally. For example $x + y$ is implemented as $\phi(x + y)$ and $\sin(x)$ as $\phi(\sin(x))$.
- All the simple functions considered are continuously differentiable.

The error analysis now has two major steps:

1. First the expression considered is recast as a sequence of simple assignments containing each at most one arithmetic operation or one function evaluation.
2. Each simple step is analysed with respect to the rounding error generated in the step and also the propagation of any earlier errors in this step.

Rewriting of expression as sequence of simple expressions

Let a denote the vector of input values and x the vector of computed terms. Then rewriting the expression takes the form

- $x_1 = f_1(x, a)$
- $x_2 = f_2(x, a)$
- $\vdots$

- $x_n = f_n(x, a)$

where the functions $f_k(x, a)$ only depend on the components x_j computed previously, i.e., where $j < k$. The computer first determines the rounded values

$$\bar{a} = \phi(a)$$

where the function ϕ is applied component wise to the vector and the sequence of simple functions evaluates as

- $\bar{x}_1 = \phi(f_1(\bar{x}, \bar{a}))$
- $\bar{x}_2 = \phi(f_2(\bar{x}, \bar{a}))$
- $\vdots$
- $\bar{x}_n = \phi(f_n(\bar{x}, \bar{a}))$

This simple form is the main reason for breaking expressions into sequences of simple functions.

Computing the errors for $\mathrm{Fix}(10, 2)$ **and** $\mathrm{Fl}(10, 2)$**:**

In the following we will now compute the results in both floating and fixed point arithmetic and compare agains the more exact results provided by Python. Note that for our *example* of evaluation of $\sin(a_1 a_2) + a_3$ one has three simple steps:

1. $x_1 = a_1 * a_2 = 0.258111$
2. $x_2 = \sin(x_1) = 0.25525458363629716$
3. $x_3 = x_2 + a_3 = 1.2555245836362972$

For simplicity with have included the value of the result. Note that x_3 is the value of the evaluated expression.

Consider now the case of **fixed point numbers in** $\mathrm{Fix}(10, 2)$ with 2 decimal digits after the decimal point. Before any of the computations are done the inputs a_k are rounded:

$$\bar{a}_k = \phi(a_k)$$

which results in a rounding error in the data given by

$$\beta_k = \bar{a}_k - a_k.$$

In our case we get the rounded inputs and their errors as

$$\bar{a} = (3.57, 0.07, 1.00), \quad \beta = (0, -0.0023, 0).$$

After the rounding step of the inputs, the sequence of simple function evaluations takes place. Computing the (unrounded) values of the simple functions using as arguments the rounded input data and the rounded results defined by

$$\bar{x}_k = \phi(f_k(\bar{x}, \bar{a})$$

gives

1. $f_1(\bar{x}, \bar{a}) = \bar{a}_1 \bar{a}_2 = 0.2499$ and $\bar{x}_1 = \phi(f_1(\bar{x}, \bar{a})) = 0.25$
2. $f_2(\bar{x}, \bar{a}) = \sin(\bar{x}_1) = 0.247307\ldots$ and $\bar{x}_2 = \phi(f_2(\bar{x}, \bar{a})) = 0.25$
3. $f_3(\bar{x}, \bar{a}) = \bar{x}_2 + \bar{a}_3 = 1.25$ and $\bar{x}_3 = \phi(f_3(\bar{x}, \bar{a})) = 1.25$

and thus

$$\bar{x} = (0.25, 0.25, 1.25).$$

The new rounding errors originating in the simple steps are

$$\delta_k = \bar{x}_k - f_k(\bar{x}, \bar{a})$$

which for this example are

$$\delta = (0.0001, 0.002693\ldots, 0).$$

Comparing against the exact values x obtained from Python we then get the error vector $e = \bar{x} - x$ as

$$e = (-0.008111, -0.005254\ldots, -0.005254\ldots).$$

Note that these errors are very close to the worst case errors for rounding which is ± 0.005.

We now do the same for **floating point arithmetic in** $\mathrm{Fl}(10, 2)$ which is also base 10 and has 2 significant decimal digits.

Here the rounding of the input is of the same form, i.e. $\bar{a}_k = \phi(a_k)$ and the rounding error is $\beta_k = \bar{a}_k - a_k$. For the example we get

$$\bar{a} = (3.6, 0.072, 1.00), \quad \beta = (0.03, -0.0003, 0).$$

Evaluating the exact values of the rounded data gives

- for the function values and rounded function values

  1. $f_1(\bar{x}, \bar{a}) = \bar{a}_1\bar{a}_2 = 0.2592$ and $\bar{x}_1 = \phi(f_1(\bar{x}, \bar{a})) = 0.26$
  2. $f_2(\bar{x}, \bar{a}) = \sin(\bar{x}_1) = 0.257080\ldots$ and $\bar{x}_2 = \phi(f_2(\bar{x}, \bar{a})) = 0.26$
  3. $f_3(\bar{x}, \bar{a}) = \bar{x}_2 + \bar{a}_3 = 1.26$ and $\bar{x}_3 = \phi(f_3(\bar{x}, \bar{a})) = 1.3$

We thus the after each elementary numerical function evaluation the following results and corresponding rounding errors $\beta_k = \phi(f_k(\bar{x}, \bar{a})) - f_k(\bar{x}, \bar{a})$:

$$\bar{x} = (0.26, 0.26, 1.3), \quad \delta = (0.0008, 0.002910\ldots, 0.04)$$

The error $e = \bar{x} - x$ is then:

$$e = (0.001889, -0.004745\ldots, 0.04474\ldots)$$

Observe: floating point error much larger! (not so bad for binary arithmetic)

So far we have determined the exact values of the errors.

Use bounds on β_k and δ_k to get bounds on e_k

We will now attempt to obtain some information about the errors in our results which is based solely on bounds on the local and initial rounding errors β_k and δ_k, respectively. We will use the following result

Proposition 3. *If $f_k \in C^1$ then*

$$e_k = \langle \nabla_x f_k(\xi, \alpha), e\rangle + \langle \nabla_a f_k(\xi, \alpha), \beta\rangle + \delta_k$$

where $\xi_k = \theta_k x_k + (1 - \theta_k)\bar{x}_k$ and $\alpha_k = \sigma_k a_k + (1 - \sigma_k)\bar{a}_k$ for some $\theta_k, \sigma_k \in [0, 1]$

Proof.

$$\begin{aligned} e_k &= \bar{x}_k - x_k = \phi(f_k(\bar{x}, \bar{a})) - f_k(x, a) \\ &= \phi(f_k(\bar{x}, \bar{a})) - f_k(\bar{x}, \bar{a}) + f_k(\bar{x}, \bar{a}) - f_k(x, a) \\ &= \delta_k + f_k(\bar{x}, \bar{a}) - f_k(x, a) \end{aligned}$$

the result follows from the intermediate value theorem □

We now apply the proposition to our example where we have

$$(f_k(x, a))_k = (a_1a_2, \sin(x_1), x_2 + a_3).$$

From this one gets the gradients as

- $\nabla_x f_1 = 0$ and $\nabla_a f_1(x, \alpha) = (\alpha_2, \alpha_1, 0)$
- $\nabla_x f_2(\xi, \alpha) = (\cos(\xi_1), 0, 0)$ and $\nabla_a f_2 = 0$
- $\nabla_x f_3 = (0, 1, 0)$ and $\nabla_a f_3 = (0, 0, 1)$

which are then used to get formulas for the errors e_k:

- $e_1 = \alpha_2\beta_1 + \alpha_1\beta_2 + \delta_1$
- $e_2 = \cos(\xi_1)e_1 + \delta_2$
- $e_3 = e_2 + \beta_3 + \delta_3$

This is a triangular linear system for the e_k, which can be solved by forward substitution to get

- $e_1 = \alpha_2\beta_1 + \alpha_1\beta_2 + \delta_1$
- $e_2 = \cos(\xi_1)(\alpha_2\beta_1 + \alpha_1\beta_2) + \cos(\xi_1)\delta_1 + \delta_2$
- $e_3 = \cos(\xi_1)(\alpha_2\beta_1 + \alpha_1\beta_2) + \beta_3 + \cos(\xi_1)\delta_1 + \delta_2 + \delta_3$

The rounding errors δ_k and β_k are different for the fixed point and floating point case. We consider first floating point arithmetic:

In this case one has $\beta_1 = 0$ and thus

$$e_1 = \alpha_1\beta_2 + \delta_1 = -0.008111.$$

For the error in the second stage one uses $\bar{x}_1 = 0.25$ and thus $\xi_1 \in [0, 0.26]$. Furthermore, one can show that $\cos(\xi_1) \in [0.96, 1.0]$. Consequently

$$-0.008111 + 0.002693 \leq e_2 \leq 0.96 * (-0.008111) + 0.002693$$

and one sees that

$$-0.0055 \leq e_2 \leq -0.005.$$

For the third and final error one has the same bound as $e_3 = e_2$.

We now consider the slightly more difficult case of floating point arithemetic. The idea we use here is to use the values $\bar{a}_k$ and $\bar{x}_k$ to get upper bounds for β_k and δ_k:

- For the input parameters we have $\beta_3 = 0$ and
  - $|\beta_1| \leq 3.6 * 0.05 \approx 0.18$
  - $|\beta_2| \leq 0.072 * 0.05 \approx 0.0036$
- For the rounding errors of the x_k one has (we use the relative errors here):
  - $|\delta_1|, |\delta_2| \leq 0.26 * 0.05 = 0.013$
  - $|\delta_3| \leq 1.3 * 0.05 = 0.065$

Using all of this we get the following bounds for the errors

- $|e_1| \leq 0.07 * 0.18 + 3.57 * 0.0036 + 0.013 = 0.038$
- $|e_2| \leq 0.038 + 0.013 = 0.051$
- $|e_3| \leq 0.051 + 0.065 = 0.116$

These error bounds are much more pessimistic – note that the bounds based on relative errors may be a factor of 10 off for base 10 arithmetic. Nonetheless, such bounds do often provide an estimate of accuracy using relatively little information. The bounds are substantially better for binary arithmetic.