Practical sheet 7: Floating-Point Computations

The first part of this sheet looks at errors in floating point calculations and the unfortunate possibility that they can effectively destroy a calculation if they accumulate in the wrong way.

By contrast, the rest of the sheet looks at some useful numerical methods where the errors do not generally cause problems.

Section 7a: Errors in floating point calculations

Task 7.1. The fact that a Python *float* uses a binary representation of numbers and has limited *magnitude* and *precision* has consequences which can be surprising.

Please watch the section 7 video about "Floating point numbers" if you have not already watched it.

Review `truncation.py` on Ultra used and discussed in detail in the video. It is probably best to use the "stepping through a program" feature of Thonny.

Here is another example for you to try. Note that the first line is a short-hand which assigns `0.1` to `x`, `0.2` to `y` and `0.3` to `z`.

```
x, y, z = 0.1, 0.2, 0.3
print(x)
print(y)
print(z)
print(x + y - z)
print((x + y) - z, x + (y - z))
print((x + y) * z == x*z + y*z)
```

What do we learn from these examples?

- that *float*s only approximate real numbers.
- that the error of approximation can depend on the order in which we do calculations.
- that it is not reliable to check if two *float*s are equal.

For the last of these reasons, a *float* should *never* be compared directly against `0.0`. Instead, one chooses a small $\varepsilon > 0$ and checks if a given expression is less than ε in absolute value:

```
eps = 1e-15
closeto0 = abs(x + y - z) < eps
print(closeto0)
```

Equivalently, if `x` and `y` are *floats*, one should check if `abs(x-y) < eps` instead of checking if `x == y`. Now what ε should one take? Well, it depends on the problem! Even worse, sometimes it only becomes clear after the fact ... This does not mean that we should despair, rather that we should be alert to the possibility that errors may arise if we are not careful. Too large a value of ε may mean an unnecessarily inaccurate answer to a calculation; too small a value may lead to an infinite loop or other undesirable behaviour.

Exercise 7.2. Here's a more complex and interesting example, relating to the **iteration**

$$x_{n+1} = \tfrac{14}{5}\, x_n + \tfrac{3}{5}\, x_{n-1}$$

Mathematical theory: **on paper, show that,** if we start with $x_0 = 1$ and $x_1 = -\frac{1}{5}$ and then apply the iteration, the result is that $x_n = (-\frac{1}{5})^n$ for all natural numbers n. *Hint: use the method of induction,* i.e. verify that the values given for x_0 and x_1 agree with this general formula and then show that, if we plug in $x_{n-1} = (-\frac{1}{5})^{n-1}$ and $x_n = (-\frac{1}{5})^n$ to the iteration, we obtain $x_{n+1} = (-\frac{1}{5})^{n+1}$.

Python: **Write a program** (no functions needed) to:

- Use the iteration to compute a list containing values $x_0, x_1, \ldots, x_{50}$.
- Compute a second list containing the numbers you would get by directly using the theoretical solution $x_n = (-\frac{1}{5})^n$.
- Compute a third list containing the difference between the corresponding values of x_n obtained using the two methods: this is the "error" of the iteration.
- Compute a fourth list containing the relative difference between the corresponding values of x_n obtained using the two methods, i.e. the difference divided by the "true value" in the second list. This is the "relative error" of the iteration.
- Print out, one row for each value of n, the four numbers obtained for each value of n.

You should see that your first Python list follows the theoretical pattern approximately and that the error gets larger while the true answer gets smaller with severe consequences for large n; the result is that the relative error explodes.

What causes the problem here is that each iteration computes the difference between two numbers which have the same sign and are quite close to each other. This is essentially the same issue as we saw in `truncation.py`.

Although these examples show that floating point calculations can go wrong, it is often the case that they do not and the rest of this sheet is about methods that do generally work. Designing numerical calculations to avoid accumulation of errors is an important topic in its own right but we do not have time to look at it further in this module.

Section 7b: Finding roots numerically using the bisection method

Task 7.3. For a continuous function $f(x)$, a reliable way to find a root, a solution of $f(x) = 0$, is the *bisection method*.

We need to start by finding (or guessing) two numbers a and b such that $f(a)$ and $f(b)$ have opposite sign, i.e. either $f(a) > 0$ and $f(b) < 0$ or $f(a) < 0$ and $f(b) > 0$. This means that there must be some x between a and b for which $f(x) = 0$, i.e. there is a root between a and b.

Suppose for now that $f(a) > 0$ and $f(b) < 0$. You can think about how to change what follows if it's the other way round (and you will need to do so for some examples which follow).

We can now repeat the following process until a is very close to b:

- set $m = (a + b)/2$, i.e. m is half-way between a and b.
- if $f(m) > 0$, change a to be equal to m; otherwise change b to be equal to m.

Why does this work? After each iteration it is still true that $f(a) > 0$ and $f(b) \leq 0$ and so the root lies between a and b. Moreover each iteration halves the distance between a and b.

Why not repeat until $a = b$? In fact there is no guarantee that this will definitely happen. Because *float*s have finite precision, we may end up with a and b different from each other but with no available *float* values between a and b. Then when we compute m it will either equal a or equal b. This may lead to an infinite loop. To avoid this problem, we stop when $|a - b|$ is less than some specified threshold which may need to depend on the context.

For example, suppose that we want to find the solution of $\mathrm{e}^{-x} = \frac{1}{2}$. Letting $f(x) = \mathrm{e}^{-x} - \frac{1}{2}$, we note that $f(0) > 0$ and $f(1) < 0$, so we take these as our starting points: $a = 0$ and $b = 1$. Write a Python program to find a root of $f(x)$ using the method described above and which stops when $|a - b| < 10^{-15}$. Remember that the way to write 10^{-15} in Python is `1e-15`.

In your program, print out a, b, m and $f(m)$. To print them nicely, you might want to look back at the later parts of section 5e.

Exercise 7.4. Pretend that Python cannot compute arbitrary powers of numbers but can only do the basic arithmetic operations: `+`, `-`, `*` and `/`.

Compute $\sqrt{2}$ with error less than 10^{-15} by solving $x^2 = 2$ using the bisection method and note the number of iterations needed (starting with $a = 1$ and $b = 2$).

Apply the same approach to compute $2^{1/3}$ with error less than 10^{-15}. You need to choose a function $f(x)$ which can be calculated without using `**` and which has $2^{1/3}$ as a root.

Exercise 7.5. Our examples so far all have solutions which are in fact easily computable directly using standard Python operators and functions: $2^{1/2}$, $2^{1/3}$ and $\log 2$. Now $f(x) = \mathrm{e}^{-x} - x = 0$ has no solution in closed form, yet it is easy to find it using bisection. Do so, and again note the number of iterations needed.

Exercise 7.6. Write a function `bisect(f,a,b,eps)` taking as arguments the name of a Python function that calculates a mathematical function whose root is sought, two suitable starting points a and b, and an ε. You may assume that the starting points are OK, i.e. that $f(a)f(b) < 0$. A good test for your function would be $f(x) = \cos(x)$ with $a = 1$ and $b = 2$, i.e. in Python you would do:

```
from math import cos, pi
x = bisect(cos, 1, 2, 1e-15)
print(x-pi/2)
```

Then try changing the range in which the root is sought: take $a = 4$ and $b = 5$. Does your `bisect` function still work? If not, why not and how can you fix it?

Section 7c: Finding roots numerically using the Newton-Raphson method

Exercise 7.7. The Babylonians computed $\sqrt{a}$ as follows. Starting with some $x_0 > 0$, let

$$x_{n+1} = \frac{x_n}{2} + \frac{a}{2x_n} \tag{7.1}$$

and stop when $|x_{n+1} - x_n|$ is small enough.

Using this method, the Babylonians found that $\sqrt{2} \simeq 30547/21600$, which is correct to six decimal places.

Code this in Python (using *float*s not fractions) to approximate $\sqrt{2}$. You can start with $x_0 = 1$ or $x_0 = 2$ and stop when $|x_{n+1} - x_n| < 10^{-15}$.

How many iterations does your code need? Compare this to the number of iterations needed in exercise 7.4. You should find that the bisection method needed many more iterations.

Task 7.8. The iteration in the previous exercise is a particular form of the *Newton–Raphson method*: to find a solution of differentiable $f(x) = 0$, we start at some suitable x_0 and iterate

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7.2}$$

until a desired accuracy is reached. For this to work, one must have $f'(x) \neq 0$ in some interval near the solution.

Verify on paper that, for function $f(x) = x^2 - a$, equation (7.2) gives equation (7.1).

Now apply the Newton-Raphson method to the function you used with the bisection method to approximate $2^{1/3}$:

- obtain $f'(x)$ and deduce the iteration implied by equation (7.2)
- implement the iteration in Python starting from $x = 1$.

Compare the number of iterations required[1] to the number required using the bisection method in exercise 7.4.

Remark 7.9. Newton–Raphson can be *very* efficient. However, unlike bisection, it can be temperamental: sometimes it can "jump away" even when starting quite close to the desired solution, e.g., try to find the smallest positive solution of $\cos x - 0.99 = 0$ starting with $x_0 = 0.001$. *Always* do a sanity check when using Newton–Raphson.

Exercise 7.10. Revisit exercise 7.5 and find the solution of $\mathrm{e}^{-x} = x$ using Newton-Raphson. Again compare the number of iterations needed for the bisection and Newton-Raphson methods.

---

[1] Impressed? Dr Wirosoetisno says that with the `clnum` library, it takes 7.5s to compute $2^{1/3}$ to a million digits on his desktop. A billion digits take a couple of days.

Section 7d: Series expansions

Task 7.11. Recall that the *Taylor series* for $\sin x$ is

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \tag{7.3}$$

which converges absolutely (cf. later in Analysis) for all x. The formula for the coefficient of x^{2i-1} is $(-1)^{i-1}/(2i-1)!$ for positive integer i.

Write a function `sintn(x,n)` which calculates the sum of the first n terms in the expansion for specified x.

Plot `sintn(x,n)` for $x \in (-\pi, \pi)$ and several values of n (on the same plot with different colour lines). You should be able to see the convergence of the Taylor series to $\sin x$ as n increases.

Section 7e: Numerical approximation of integrals

Task 7.12. Please watch the section 7 video about "The trapezoidal rule" if you have not already watched it.

In the video, the "trapezoidal rule" for approximating an integral was presented and the Python code used is presented in `trapezoidal.py` on Ultra.

Another method for approximating an integral is the "mid-point rule".

If you know what the mid-point and trapezoidal rules are, then skip to the next paragraph. Otherwise, Google "midpoint rule" and read the page on www.dummies.com. Then Google "trapezoidal rule" and read the Wikipedia page.

Now review `trapezoidal.py`.

Modify a copy of `trapezoidal.py` so that it implements the mid-point rule instead of the trapezoidal rule.

Exercise 7.13. Compare the performance of the mid-point and trapezoidal rules for approximating $\int_{-1}^{1} e^{2x}\,dx$. In other words, how does the accuracy depend on the number of "slices" for both rules. Which is better?

What next?

You have now reached the end of the core material for the module, i.e. what is needed in order to be able to complete all the quizzes and to be able to get most, perhaps all, of the marks for the second assignment.

If you wish to take the Mathematical Modelling II module next year, you should also make sure to complete section 8. Otherwise, it might be time to work through the "Beyond the basics" sections of practical sheets 2 to 6 if you have not already done so.