PYU33C01, S. Hutzler, 2023
Assignment 1: Iterations and numerical accuracy

A: Heron's root finding method

Given a real number a, with $a > 0$, how does one find the value of x so that $x^2 = a$? One can use the following *iterative* process, possibly dating back to the Babylonians: $x_{n+1} = \frac{1}{2}(x_n + a/x_n)$, where x_0 is a chosen starting value and n is the iteration number.

Write Python code to compute the square root of $a = 2$.

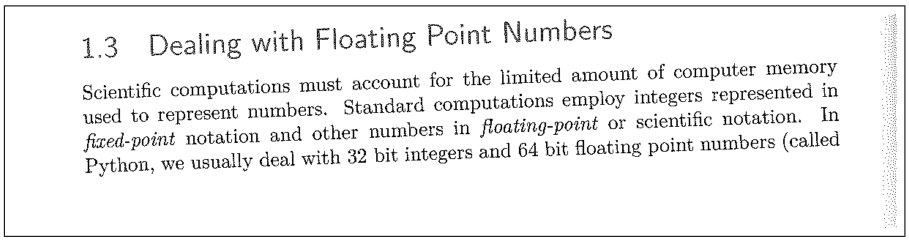
Plot x_n as a function of the iteration number n (try two or three different choices of x_0). Plot the relative error for the quantity $x_n \times x_n$ as a function of n.

B: Numerical accuracy

Pasted below is a short extract from the book "Computational Problems for Physics" by Landau and Páez, CRC press 2018.

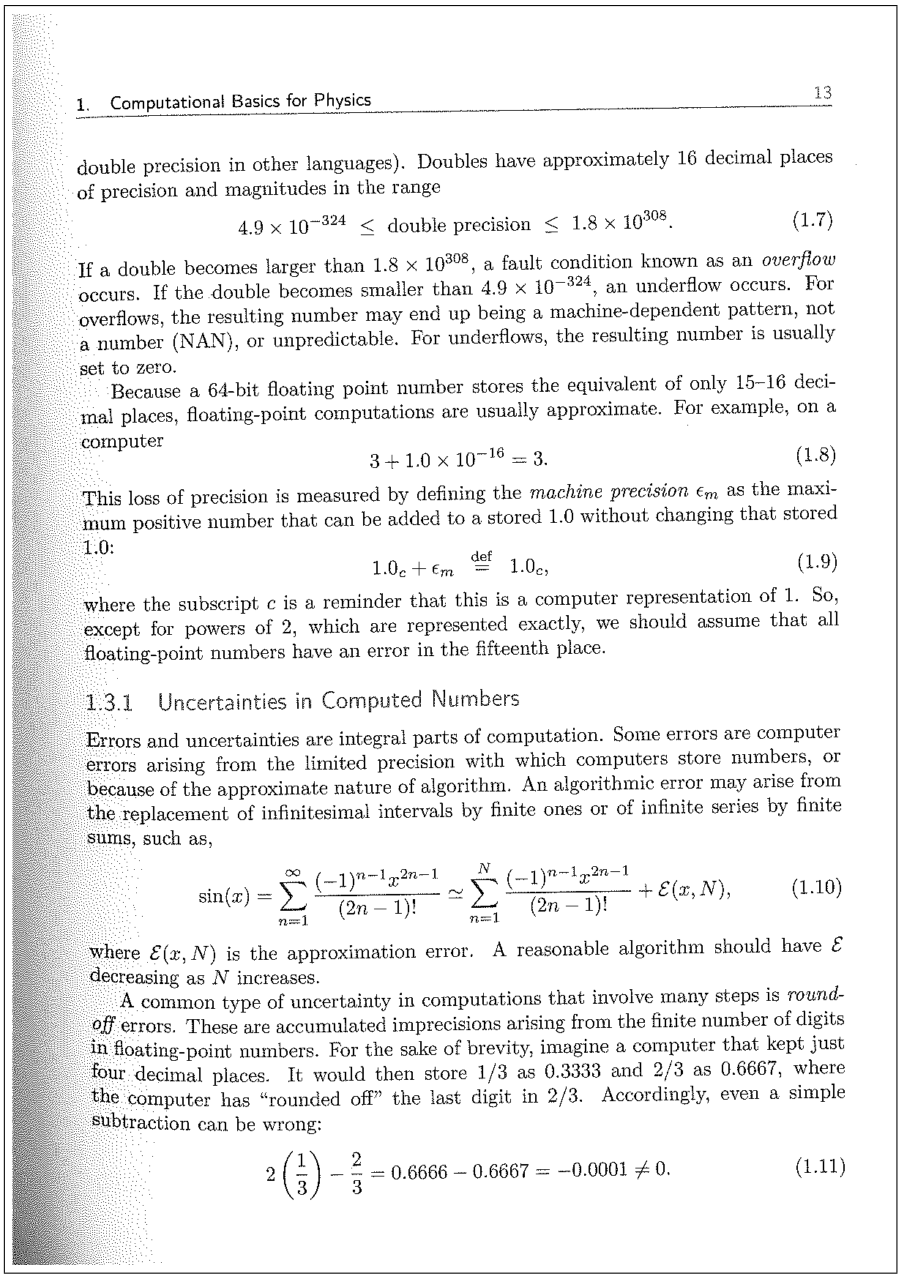
- Please read this extract carefully.
- Write Python code to address question 1 and 2 of Section 1.3 (Dealing with Floating Point Numbers) for the particular computer that you are using.
- Ignore question 1c. In the current version of Python there is no maximum value of integers anymore. (*int* in Python can handle as large a value as memory is available.)
- Write Python code for Questions 1a,b of Section 1.4 (Numerical Derivatives). For question 1b: Can one really achieve an accuracy for the derivative that is anywhere near the machine precision?

Write a brief report on Sections A and B, not more than 5 pages (excluding your code, which you should submit separately), detailing your findings. This should include relevant graphical output of your code. **Please stick to the 5 page limit (and a reasonable font size!). I will not read/mark material beyond this.**
Submit your report (as pdf) **via Blackboard**. Also submit your Python code as a *.py* file (i.e. ***not*** as a Word document). Upon execution, the code should also display your name and student number on the screen.
Submission deadline: Monday, October 2, 2023



1.3 Dealing with Floating Point Numbers

Scientific computations must account for the limited amount of computer memory used to represent numbers. Standard computations employ integers represented in *fixed-point* notation and other numbers in *floating-point* or scientific notation. In Python, we usually deal with 32 bit integers and 64 bit floating point numbers (called



1. Computational Basics for Physics 13

double precision in other languages). Doubles have approximately 16 decimal places of precision and magnitudes in the range

$$4.9 \times 10^{-324} \;\leq\; \text{double precision} \;\leq\; 1.8 \times 10^{308}. \qquad (1.7)$$

If a double becomes larger than 1.8×10^{308}, a fault condition known as an *overflow* occurs. If the double becomes smaller than 4.9×10^{-324}, an underflow occurs. For overflows, the resulting number may end up being a machine-dependent pattern, not a number (NAN), or unpredictable. For underflows, the resulting number is usually set to zero.

Because a 64-bit floating point number stores the equivalent of only 15–16 decimal places, floating-point computations are usually approximate. For example, on a computer

$$3 + 1.0 \times 10^{-16} = 3. \qquad (1.8)$$

This loss of precision is measured by defining the *machine precision* ϵ_m as the maximum positive number that can be added to a stored 1.0 without changing that stored 1.0:

$$1.0_c + \epsilon_m \overset{\text{def}}{=} 1.0_c, \qquad (1.9)$$

where the subscript c is a reminder that this is a computer representation of 1. So, except for powers of 2, which are represented exactly, we should assume that all floating-point numbers have an error in the fifteenth place.

1.3.1 Uncertainties in Computed Numbers

Errors and uncertainties are integral parts of computation. Some errors are computer errors arising from the limited precision with which computers store numbers, or because of the approximate nature of algorithm. An algorithmic error may arise from the replacement of infinitesimal intervals by finite ones or of infinite series by finite sums, such as,

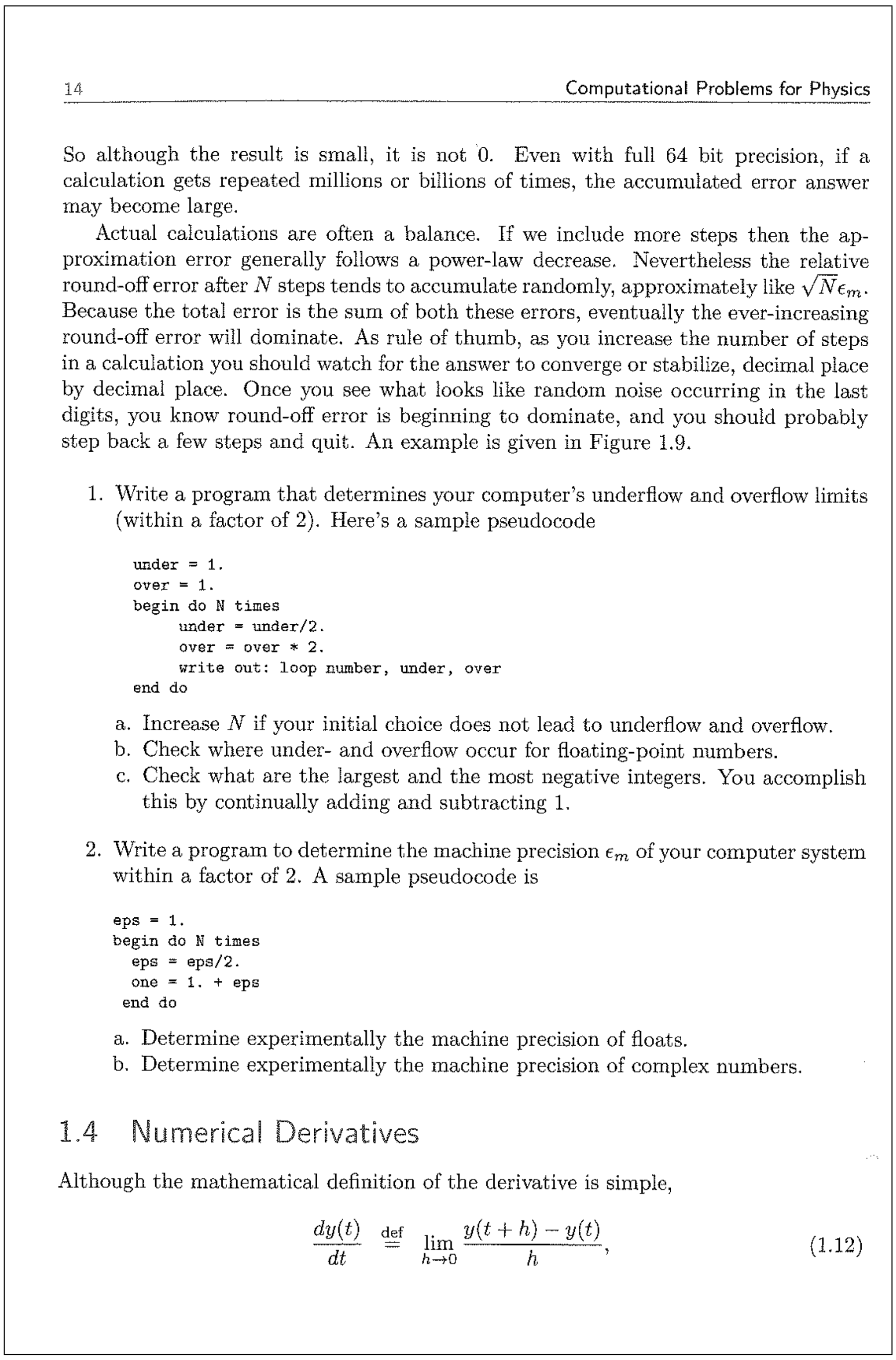
$$\sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \simeq \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} + \mathcal{E}(x, N), \qquad (1.10)$$

where $\mathcal{E}(x,N)$ is the approximation error. A reasonable algorithm should have $\mathcal{E}$ decreasing as N increases.

A common type of uncertainty in computations that involve many steps is *round-off* errors. These are accumulated imprecisions arising from the finite number of digits in floating-point numbers. For the sake of brevity, imagine a computer that kept just four decimal places. It would then store 1/3 as 0.3333 and 2/3 as 0.6667, where the computer has "rounded off" the last digit in 2/3. Accordingly, even a simple subtraction can be wrong:

$$2\left(\frac{1}{3}\right) - \frac{2}{3} = 0.6666 - 0.6667 = -0.0001 \neq 0. \qquad (1.11)$$

Figure 1: From "Computational Problems for Physics" by Landau and Páez, CRC Press 2018.



14 Computational Problems for Physics

So although the result is small, it is not 0. Even with full 64 bit precision, if a calculation gets repeated millions or billions of times, the accumulated error answer may become large.

Actual calculations are often a balance. If we include more steps then the approximation error generally follows a power-law decrease. Nevertheless the relative round-off error after N steps tends to accumulate randomly, approximately like $\sqrt{N}\epsilon_m$. Because the total error is the sum of both these errors, eventually the ever-increasing round-off error will dominate. As rule of thumb, as you increase the number of steps in a calculation you should watch for the answer to converge or stabilize, decimal place by decimal place. Once you see what looks like random noise occurring in the last digits, you know round-off error is beginning to dominate, and you should probably step back a few steps and quit. An example is given in Figure 1.9.

1. Write a program that determines your computer's underflow and overflow limits (within a factor of 2). Here's a sample pseudocode

   ```
   under = 1.
   over = 1.
   begin do N times
        under = under/2.
        over = over * 2.
        write out: loop number, under, over
   end do
   ```

   a. Increase N if your initial choice does not lead to underflow and overflow.
   b. Check where under- and overflow occur for floating-point numbers.
   c. Check what are the largest and the most negative integers. You accomplish this by continually adding and subtracting 1.

2. Write a program to determine the machine precision ϵ_m of your computer system within a factor of 2. A sample pseudocode is

   ```
   eps = 1.
   begin do N times
     eps = eps/2.
     one = 1. + eps
    end do
   ```

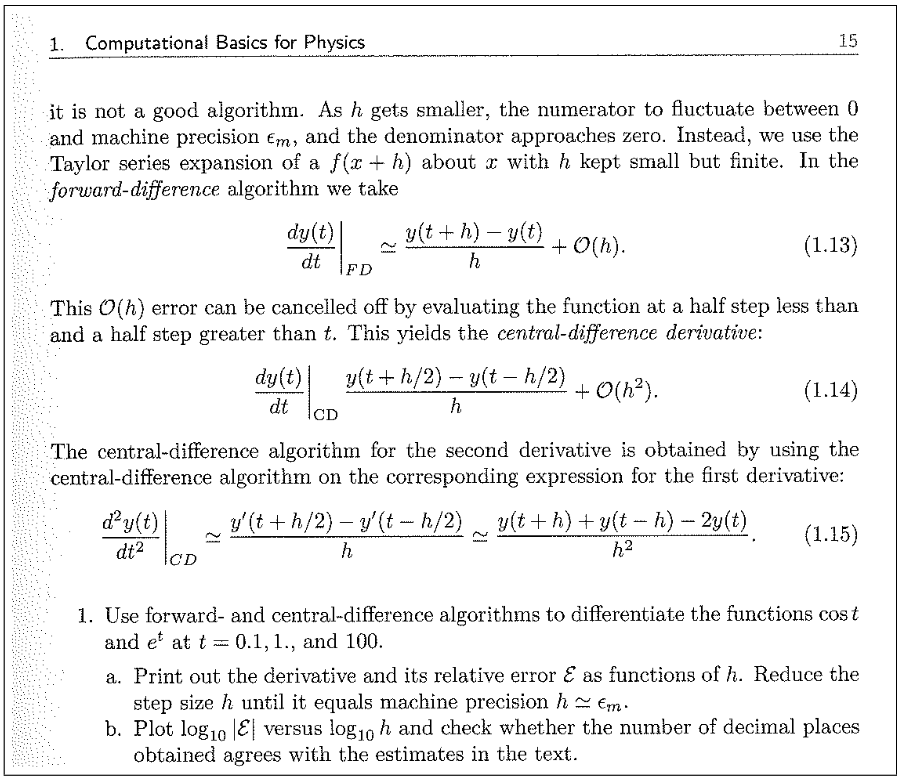
   a. Determine experimentally the machine precision of floats.
   b. Determine experimentally the machine precision of complex numbers.

1.4 Numerical Derivatives

Although the mathematical definition of the derivative is simple,

$$\frac{dy(t)}{dt} \overset{\text{def}}{=} \lim_{h\to 0} \frac{y(t+h)-y(t)}{h}, \tag{1.12}$$

Figure 2: From "Computational Problems for Physics" by Landau and Páez, CRC press 2018.



1. Computational Basics for Physics 15

it is not a good algorithm. As h gets smaller, the numerator to fluctuate between 0 and machine precision ϵ_m, and the denominator approaches zero. Instead, we use the Taylor series expansion of a $f(x+h)$ about x with h kept small but finite. In the *forward-difference* algorithm we take

$$\left.\frac{dy(t)}{dt}\right|_{FD} \simeq \frac{y(t+h)-y(t)}{h} + \mathcal{O}(h). \tag{1.13}$$

This $\mathcal{O}(h)$ error can be cancelled off by evaluating the function at a half step less than and a half step greater than t. This yields the *central-difference derivative*:

$$\left.\frac{dy(t)}{dt}\right|_{\mathrm{CD}} \quad \frac{y(t+h/2)-y(t-h/2)}{h} + \mathcal{O}(h^2). \tag{1.14}$$

The central-difference algorithm for the second derivative is obtained by using the central-difference algorithm on the corresponding expression for the first derivative:

$$\left.\frac{d^2y(t)}{dt^2}\right|_{CD} \simeq \frac{y'(t+h/2)-y'(t-h/2)}{h} \simeq \frac{y(t+h)+y(t-h)-2y(t)}{h^2}. \tag{1.15}$$

1. Use forward- and central-difference algorithms to differentiate the functions $\cos t$ and e^t at $t = 0.1, 1.$, and 100.
   a. Print out the derivative and its relative error $\mathcal{E}$ as functions of h. Reduce the step size h until it equals machine precision $h \simeq \epsilon_m$.
   b. Plot $\log_{10}|\mathcal{E}|$ versus $\log_{10} h$ and check whether the number of decimal places obtained agrees with the estimates in the text.

Figure 3: From "Computational Problems for Physics" by Landau and Páez, CRC press 2018.