Week 4 of CS319: Scientific Computing (with C++)

Lab 2: `floats` and numerical differentiation

Goal: To study the computer representation of numbers – particularly `float`s and `double`s, and the see the impact of their precision on numerical differentiation.

Assignment: Submit your work as a C++ program, and also a short report, as described on Slide 9. Upload it to Canvas... `2425-CS319` ... "Assignments... Lab 2".

Deadline: 12pm (noon), Wednesday 12 Feb 2025.

It is **important** that your files include comments with **your name**, email address, and **ID number**.

The assignment will be graded taking into account if the program compiles, if achieves the task described in Slide 8, and if it includes the requested information.

Collaboration policy. Collaboration is encouraged. It is acceptable for two people to work together and submit exactly the same work. However.

- Both need to submit the code independently. (No submission, no score, no exceptions).
- Your submission must include comments with YOUR name and ID number, and also give the name of your collaborator.
- The use of generative AI is **strictly prohibited**. Any suspected cases will be subject to standard University procedures.

Q1

Question 1

Compute the "machine epsilon" for `float`. That is: find the smallest `float`, x, such that we can distinguish between 1 and $1 + x$. Write a C++ program to do this. Some notes:

- The Wikipedia entry for machine epsilon is rather good.
- Compilers, and CPUs, often try to be clever, and may preform interim calculations at higher precision than asked. So:

  ```
  if ( 1.0 + x/2.0 > 1.0 ) ...
  ```

  can behave differently from

  ```
   z=1.0+x/2.0;
  if ( z > 1.0 ) ...
  ```

Q1

- In C++, one can actually check the "epsilon" value of any datatype, by including the `limits` header, and outputting the value of `std::numeric_limits<T>::epsilon()` where `T` is replaced with the type of interest. Compare your value with the value from `limits`.
- When you are done, compare the results with those given in the solution: https://www.niallmadden.ie/2425-CS319/lab1/Lab2-Q1.cpp

Q2

Question 2

Write a C++ programme that computes the *machine epsilon* for the `double` data type. That is, repeat Q1 but using the `double` data type instead of float.
Do the results agree with the theory covered in Week 3 lectures?

Q3

Question 3

Numerical Differentiation is the process of estimating the derivative of a function, $f = f(x)$, at some value of x. It is a hugely important topic, including in machine learning (for back-propagation steps in deep neural networks). Most modern approaches are based on **algorithmic** differentiation. However, we'll use a classic approach, called **finite differences**.

It is not hard to show (using truncated Taylor Series), that

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \tag{1}$$

This is called a "**two-point** method".

It is implemented in the code at

niallmadden.ie/2425-CS319/lab2/Lab1-Q2.cpp

That programme attempts to estimate $f'(1.0)$ with $f(x) = e^{-x}$ using the formula in (1) with $h = 2^{-2} = 0.25$ The correct answer, to 8 digits, is `-0.36787944` The estimate it computes is `-0.32549858`, giving an error of `0.0424`.

Q3

Suppose we decide we need a more accurate estimate. Then we have two choices:

(a) Use a smaller value of h, since (1) suggests the error is proportional to h.

(b) Use a better method such as the so-called *three-point* scheme:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \tag{2}$$

(Aside: *why might this be called a 3-point scheme, given that it involves only two points: $x-h$ and $x+h$? Thoughts?*)

Your task will be to try both these approaches, and establish what is the most accurate estimate that can be achieved, using both `float`s and `double`s.

If there were no round-off errors, you could take h as small as needed to make the error as small as you'd like. However, that is not possible in practice.

More detail below...

Assignment

Assignment

Task 1 Adapt the code in Lab1-Q2.cpp so that the formula in (1) is used to estimate $f'(x_0)$ for smaller and smaller values of h. Initially, you should find that smaller h gives smaller error. However, eventually you should find round-off error starts to dominate.

What is the smallest error you can achieve, and what is the corresponding value of h?

Tips:

- Try the algorithm for $h = 1$, $h = 1/2$, …, $h = 2^{-n}$, …. That is, at each iteration, reduce the size of h by a factor of 2.
- You can use a `for`-loop or a `while`-loop in your code, as you see fit. If using a `for`-loop, experiment to find the optimal number of iterations.

Task 2 Repeat **Task 1**, but using the formula in (2). What is the smallest h you can use, and what is the corresponding error?

Task 3 Repeat **Task 1**, but this time using a `double`.

What you upload

What to upload

1. Upload your code that implements any **one** of **Tasks 1, 2 or 3**. Your code does not have to do all three, but should be easily tweaked to do any of them. (E.g., it might run for `float`, but using a search/replace function changed to `double`. Or, if you are feeling very determined, you could use `#define`...).
   Make sure your code identifies you as the author, and names any collaborators.
   You file must be in plain text, and should have a *.cpp* suffix. If you are not sure about this, talk to Niall.
2. Write a very short report, outlining your findings for each of the three cases. Maximum half a page. Again: include your name and ID number, and any collaborators.

Further reading

The following is not part of the assignment.

- If one implements Algorithm (1) in a datatype that has precision ε, one can show the optimal h, and corresponding errors are, in theory:

$$h = 2\sqrt{\frac{\varepsilon}{M}}, \qquad Error \sim 2\sqrt{\varepsilon M}.$$

- To read more about that, have a look at Lecture 24 of "Afternotes on Numerical Analysis". You can find it online through the Library Catalogue: `https://search.library.nuigalway.ie/permalink/f/1pmb9lf/353GAL_ALMA_DS5164291970003626`
- Amazingly, there is a very similar scheme that does not suffer any of the same problems, called the "Complex Step Method". You can read about it at `https://nhigham.com/2020/10/06/what-is-the-complex-step-approximation`