Lecture notes FYS3150 – Computational Physics, fall 2023

Introduction

- **Welcome to this course!**
- About me:
  - Anders Kvellestad
  - Researcher in the Section for Theoretical Physics
  - Background: Bergen → Oslo → Stockholm → Oslo → London → Oslo
  - Work on exploring new theories in particle physics
  - Keywords: LHC, supersymmetry, dark matter, Higgs, statistics, coding (Python, C++, …), supercomputers, causing and fixing bugs, responsible for coffee supplies in the theory group
- The teaching team this semester:
  - Even Marius Nordhagen
  - David Richard Shope
  - Ingvild Bergsbak
  - Carl Andreas Lindstrøm
  - Nils Enric Canut Taugbøl
  - Felix Forseth
- In addition, we will have a guest lecture from Anna Kathinka Dalland Evans on how to write scientific reports
- **Question:** Who are you?
  - Study programmes?
  - Level of coding experience?
  - Main motivation for this course?
    * Solve those pesky equations
    * Learn C++ and other tools
    * Get filthy rich
    * I just like working with computers
- **Question:** What operating system are you using?
  - Linux?
  - macOS?
  - Windows?

About the course

- Course resources
  - Official UiO course page
  - Our own page, with course material
  - Our Git repo
  - Canvas, for handing in reports
- Teaching language: English
- Programming languages:
  - Main focus on C++
  - Python for data analysis, making plots, etc.
  - Bash for terminal examples, short scripts, etc.
- You *can* use Python for the projects, but we strongly recommend C++
- All lectures and group sessions will assume that you use C++
- This course has been taught by CompPhys guru Morten Hjorth-Jensen for many years
- I took over this course in 2021
- I follow Morten's old course fairly closely, but with a number of personal tweaks from my side
- Course background material: Morten's lecture notes / book draft, available via the UiO course page
  - I will often point you to relevant parts of Morten's notes
  - But our main curriculum is what we discuss in the lectures and as part of the projects
- Course philosophy:
  - Pragmatic, learning by doing
  - Will try to focus on concrete examples
  - Computational physics is a *huge* and highly active field — this course is just a first introduction
- Lectures:
  - Thursdays and Fridays, 10.15 – 12.00
  - New this year: Will try to lecture in ~30 minute sessions, with two short breaks
- Group sessions:

  - Also Thursdays and Fridays
  - Probably the most important arena for learning and mastering this course!
  - Four two-hour sessions per week
  - You can come to any group session you want
    * Try to avoid all going to the same session
    * Be patient with our group teachers — some have taught this course for several years, others are doing it for the first time

- Formal requirements
  - Two **problem sets**: Must be **passed**
  - Three **projects**: **Scored** from 0–100
  - Final grade based on weighted average of the project scores. Weighting: 20%, 40%, 40%

- For simplicity, we'll just refer to everything as "projects", i.e. we'll talk about projects 1–5, and the grade is based on projects 3–5.

- *Tentative* deadlines:
  - Project 1: September 12
  - Project 2: September 26
  - Project 3: October 24
  - Project 4: November 21
  - Project 5: December 12

- Policy on deadlines: **friendly, but strict**
  - Need to be strict on deadlines, both to keep things fair and to keep up with our time schedule
  - There are no second attempts
  - Substantial deadline extensions due to illness require a doctor's note

- **Collaboration is encouraged!**
  - We *strongly* encourage you to collaborate in small groups of 2–4 people.
  - 3 people per group is ideal
  - A group hands in a *joint* project report and code
  - By working together *you will learn more*, and we get more time for grading per project report → *more detailed feedback from us*

- Asking questions:
  - **Please ask questions!**
  - Any time during lectures — just cut in and ask

  - For help with your specific project/code:
    * Primary forum: group sessions
    * Secondary forum: our online discussion forum
  - *Think* and *try yourself* before you ask for help
  - When writing questions:
    * Keep it short and concise
    * Have respect for other people's time (your fellow students, the teachers, …)
  - Any personal or procedural issues: Send me an email. (We can also set up a meeting.)

- The broad topics of this course:
  - Learn basic C++, with focus on numerics
  - Matrix operations, eigenvalue problems
  - Solve ordinary and partial differential equations
  - Numerical integration
  - Monte Carlo methods, simulation of stochastic systems
  - Proper presentation of results
  - Debugging :)

- This course is good for your CV (beyond just the grade you get)
  - We're at a university, so hopefully our main motivation for following/teaching a course should be that learning new stuff is interesting and valuable in itself — that's at least my main motivation when teaching this!
  - Having said that, after completing this course you can probably also add some new points to your CV:
    * Experience with C++
    * Experience with the Unix terminal
    * Experience with git and GitHub
    * Experience with writing technical reports in LaTeX
    * …

The most useful advice you'll get all year

- Something you don't understand?
  - *Read* and *think*
  - *Discuss* with your fellow students
  - *Ask* us

- Code isn't working?
  - Don't just try stuff at random!
    * This rarely works, and when it does you typically still can't trust the results…
  - *Read the documentation* for the command/tool you are using
  - *Search online for the error message*, after removing things that are specific to your code (variable names, file names, etc.)
    * *Read* the explanations you find, don't just copy code
  - Try to isolate and reproduce the problem in a small, separate example code. (A *minimal working example*.)
  - Read the course pages on debugging
  - We'll also discuss debugging in the lectures
- How you present your results **really matters**
  - Quality of language
  - Quality of figures
  - Layout
  - Report structure
  - Referencing
  - Code comments and documentation

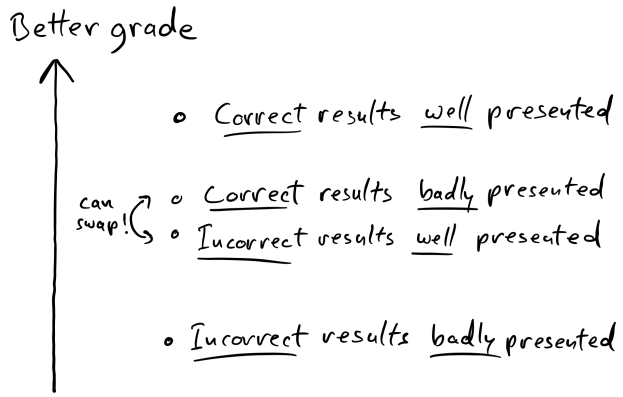

Figure 1: Presentation quality matters

- Spend time with pen and paper before you start coding
  - Make a rough sketch of program parts and flow
  - Sketch your program with code comments first, then start filling in the code
  - Make a sketch of discretisations, to avoid mistakes with indices

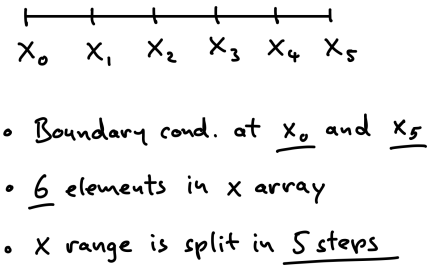

Figure 2: Sketch discretisations

- Make sure you understand the quantities you present in plots and tables
  - Makes it much easier to spot mistakes
  - Pay attention to units!
  - *Tip:* Always set axis ranges manually
- Read the report template we provide, plus the example student reports
- And read the *Checklist for reports* page on our webpage, to avoid many common mistakes

Plagiarism

- Plagiarism is **very serious**
- Have seen a few cases in the past
- Can have very serious consequences, e.g. loosing the right to study
- You must:
  - Write your own text — never copy text from others (unless it is marked a direct quote)
  - Write your own code, unless it's code we have provided to help
  - Always acknowledge contributions from others
  - Properly cite articles, books, webpages, …
    * We'll discuss this more in detail when you start writing project reports

Use of ChatGPT and related tools

- ChatGPT and other AI-based *large language models* (LLMs) can, like any new tool, be used in wise ways and not-so-wise ways
- The policy on LLM use in our course is as follows:

  - *If* you use an LLM in your work, you need to add to your report a description of what you used the LLM for. This would be part of the *Methods* section of your report. (We'll discuss report writing in detail later in the course.)
  - In the report template we will probably add a dedicated subsection called e.g. *Tools*, where you mention the key tools you have used, and what you have used them for, e.g. sentences like "All figures in this report have been made using the Python package `matplotlib`."
  - If we see that you have used an LLM in ways that you haven't described in the report, this will lead to a lower score, analogous to what happens if you don't provide proper references, or just have a very incomplete description of the methods you've used.
  - **Important:** When you hand in a report or code, you take full responsibility for all the content. That is, you can never put the blame for anything on an LLM model.

- Some advice:
  - Don't use LLMs like ChatGPT as search engines. An LLM is *not* a new, cool way of searching the web. There is no database, no in-built checks for correctness of content, etc.
    * So you should *not* use LLM output as a reference for a statement in your report.
  - For you to be able to judge the quality, correctness and appropriateness of some LLM output, you first need to actually build up your own expertise, That is, you need to
    * study the given scientific topic
    * know/learn how to write good texts
    * know/learn how a given coding language works
    * …
  - The best way to learn these things is to sit down and do them yourself, mostly from scratch
  - *Once* you have built up the necessary expertise, LLMs can become a useful tool for some tasks
  - Examples of tasks where an LLM may be useful in this course:
    * Help with debugging code problems
    * Help with suggesting language improvements (to text that you have already drafted)
  - My main advice: Don't use LLMs too much!
    * Learning how to use LLMs is itself a useful skill
    * But overuse will probably reduce your learning outcome in this course!
    * The most "painful" moments in your work – when you work through the math yourself, when you try to formulate a correct and good sentence for your report, when you

systematically go through your code to find that one strange bug, or when you think carefully about whether a given result makes sense – these are the moments when you actually learn the most!

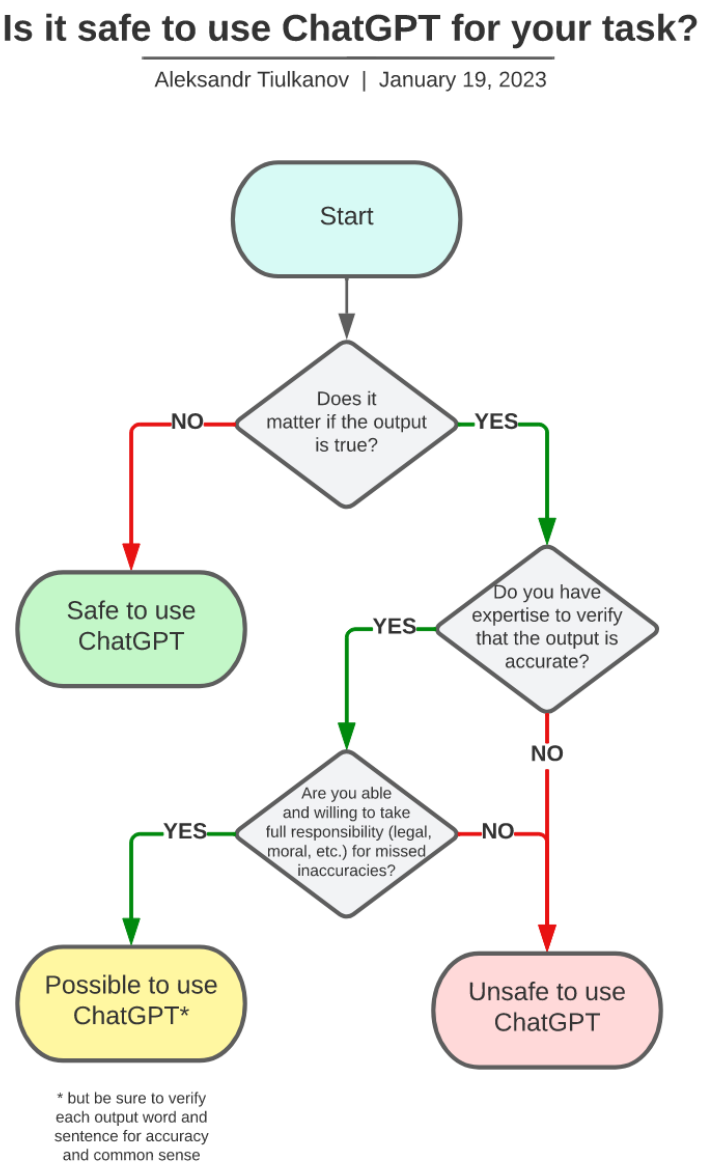

Figure 3: Example considerations to make before using ChatGPT or similar tools. Flowchart by A. Tiulkanov, included in the UNESCO report *ChatGPT and Artificial Intelligence in higher education*.

In-lecture code discussion #1

- We have two pages with coding resources
  - anderkve.github.io/FYS3150
  - github.com/anderkve/FYS3150/tree/master/code_examples
- All code examples I discuss in the lectures can be found in one of these places
- Long code examples, e.g. example programs involving multiple files, are typically found in the `code_examples` directory of our Git repo.
- Make sure to explore these pages on your own! There's lots of help and hints to be found there!
- Now let's introduce C++
  - anderkve.github.io/FYS3150/book/introduction_to_cpp/intro
  - anderkve.github.io/FYS3150/book/introduction_to_cpp/hello_world
  - anderkve.github.io/FYS3150/book/introduction_to_cpp/compiling_and_linking_take_1

The last lecture ended here.

Discretisation of continuous functions

- Computers can't represent all possible numbers (finite range and "resolution")
  → Need to discretise!
- Take some function $u(x)$, with $x \in [x_{\min}, x_{\max}]$. ($u(x)$ might e.g. be the solution of our diff. eq. in project 1.)
- u and x are *continuous* quantities

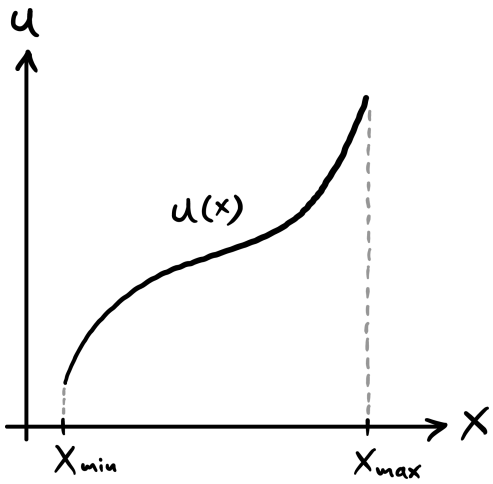

Figure 4: Continuous function

- Discretised representation

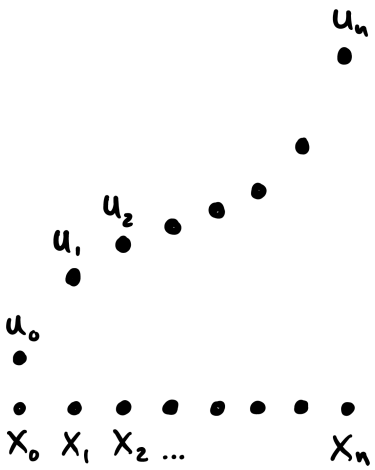

Figure 5: Discretised representation

Tip: When testing and debugging your code or trying to understand your results, it's often useful to work with a low number of points (coarse discretisation) and make plots that display your raw data points, i.e. not just directly draw lines between the points.

My notation

$$x \to x_i$$
$$u(x) \to u(x_i) \equiv u_i$$
$$u(x \pm h) \to u(x_i \pm h) \equiv u_{i\pm 1}$$

- So far u_i is the exact $u(x)$ at point $x = x_i$
- Our numerical methods will find an *approximation to the exact* u_i
- We will sometimes call this approximation v_i, to highlight that this approximation is not the same as the exact u_i

Basic relations

- $x_i = x_0 + ih$, with $i = 0, 1, 2, \ldots, n$
- step size: $h = x_1 - x_0 = \frac{x_2 - x_0}{2} = \ldots = \frac{x_n - x_0}{n}$.
  ($x_0 = x_{\min}$, $x_n = x_{\max}$)
- Will sometimes use notation Δx for h
- *Remember:* n **steps** corresponds to $n + 1$ **points**
- Always make a sketch if you are unsure about the discretisation

Numerical differentiation

See Chapter 3.1 in Morten's notes.

Main results

First derivative:

$$\left.\frac{du}{dx}\right|_{x_i} = u_i' = \frac{u_{i+1} - u_i}{h} + \mathcal{O}(h), \quad \text{(two-point, forward difference)}$$

$$\left.\frac{du}{dx}\right|_{x_i} = u_i' = \frac{u_i - u_{i-1}}{h} + \mathcal{O}(h), \quad \text{(two-point, backward difference)}$$

$$\left.\frac{du}{dx}\right|_{x_i} = u_i' = \frac{u_{i+1} - u_{i-1}}{2h} + \mathcal{O}(h^2) \quad \text{(three-point)}$$

Second derivative:

$$\left.\frac{d^2u}{dx^2}\right|_{x_i} = u_i'' = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \mathcal{O}(h^2)$$

Derivation

- Starting point: Taylor expansion of u around a point x

$$\begin{aligned} u(x+h) &= \sum_{n=0}^{\infty} \frac{1}{n!} u^{(n)}(x) h^n \\ &= u(x) + u'(x)h + \tfrac{1}{2}u''(x)h^2 + \tfrac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \end{aligned}$$

> An aside on notation:
>
> - $u(x+h) = u(x) + u'(x)h + \mathcal{O}(h^2)$, (exact)
> - $u(x+h) \approx u(x) + u'(x)h$, (approximation, with truncation error $\mathcal{O}(h^2)$)

- Can get expression for $u'(x)$:

$$u(x+h) = u(x) + u'(x) + \mathcal{O}(h^2)$$
$$\Rightarrow u'(x) = \frac{u(x+h) - u(x) - \mathcal{O}(h^2)}{h}$$

$$u'(x) = \frac{u(x+h) - u(x)}{h} + \mathcal{O}(h), \quad \text{(note power of } h\text{)}$$

Discretise:

$$u(x) \to u_i$$
$$\Rightarrow u'_i = \frac{u_{i+1} - u_i}{h} + \mathcal{O}(h)$$

(Two-point, forward difference)

- Compare to definition of the first derivative:

$$u'(x) \equiv \lim_{h \to 0} \frac{u(x+h) - u(x)}{h}$$

- We could have used the points x and $x - h$, which would have given us

$$u'(x) = \frac{u(x) - u(x-h)}{h} + \mathcal{O}(h)$$

Discretise:

$$u(x) \to u_i$$
$$\Rightarrow u'_i = \frac{u_i - u_{i-1}}{h} + \mathcal{O}(h)$$

(Two-point, backward difference)

- Quick illustration of forward difference method:
  - Example: $u(x) = a_0 + a_1 x + a_2 x^2$

- Exact: $u'(x) = a_1 + 2a_2x$
- Approximation:

$$\begin{aligned} u'(x) &\approx \frac{u(x+h) - u(x)}{h} \\ &= \frac{[a_0 + a_1(x+h) + a_2(x+h)^2] - [a_0 + a_1x + a_2x^2]}{h} \\ &= \frac{a_1h + a_2x^2 + 2a_2xh + a_2h^2 - a_2x^2}{h} \\ &= a_1 + 2a_2x + a_2h \end{aligned}$$

- Compare to the exact expression: our approximation is wrong by an $\mathcal{O}(h)$ term, as expected
- This **truncation error** gets smaller when we take $h \to 0$
- But doing this can lead to **roundoff errors** in the subtraction $u(x+h) - u(x)$, causing a **loss of precision**
- We will return to this topic later

- We can use more than two points to compute $u'(x)$:
  - Starting point: Taylor expansions for $u(x+x)$ and $u(x-h)$:

$$\begin{aligned} u(x+h) &= u(x) + u'(x)h + \tfrac{1}{2}u''(x)h^2 + \tfrac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \\ u(x-h) &= u(x) - u'(x)h + \tfrac{1}{2}u''(x)h^2 - \tfrac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \end{aligned}$$

  - Subtract:

$$u(x+h) - u(x-h) = 2u'h + \tfrac{2}{6}u'''h^3 + \mathcal{O}(h^5) \quad \text{(note power } h^5\text{)}$$

  - Rearrange:

$$u' = \frac{u(x+h) - u(x-h)}{2h} - \tfrac{1}{6}u'''h^2 - \mathcal{O}(h^4)$$

$$\boxed{u'(x) = \frac{u(x+h) - u(x-h)}{2h} + \mathcal{O}(h^2)}$$

Discretise:

$$u_i' = \frac{u_{i+1} - u_{i-1}}{2h} + \mathcal{O}(h^2)$$

(Three-point expression)

- The second derivative
  - Add Taylor expansions for $u(x+h)$ and $u(x-h)$

$$u(x+h) + u(x-h) = 2u(x) + u''(x)h^2 + \mathcal{O}(h^4)$$

  - Rearrange to isolate $u''(x)$

$$u''(x) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} + \mathcal{O}(h^2)$$

Discretise:

$$u_i'' = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \mathcal{O}(h^2)$$

Topics in project 1

Some things are covered in the lectures, other things via examples on the webpage

- Discretisation of a continuous problem (boundary value problem)

$$
\begin{aligned}
-\frac{d^2u}{dx^2} &= f(x) \\
u &\in [0,1] \\
u(0) &= 0 \\
u(1) &= 0
\end{aligned}
$$

- Mathematical approx. to second derivative (suitable for discretisation)
- Connection to standard matrix equation ($\mathbf{A}\vec{x} = \vec{b}$) and approaches to solve this
  - Gaussian elimination
  - LU decomposition
- Errors!
  - Truncation error (purely math)
  - Numerical roundoff error (can't represent numbers with infinite precision on computers)
    * → *loss of numerical precision*
- Counting floating-point operations (FLOPs)
- Coding:
  - Working with arrays/vectors and matrices
  - Input/output (nicely formatted output)
  - Timing the code
  - Compilation and linking, basic code design

Boundary value problems (BVPs)

- Our case in project 1:

$$-\frac{d^2u}{dx^2} = f(x)$$

  - $u(x)$ is an *unknown* function → what we want to find
  - $f(x)$ is some *known* function
  - $x \in [0, 1]$
  - Boundary values: $u(0 = 0$ and $u(1) = 0$ (Dirichlet)

- Special case of:

$$\alpha\frac{d^2u}{dx^2} + \beta\frac{du}{dx} + \gamma u(x) = f(x)$$

  - *Ordinary* diff. eq., since there is only one independent variable (x)
  - *Linear* diff. eq., since each term has maximum one power of u, u', u'', …
  - *Second order* diff. eq., since the highest-order derivative is u''
  - *Inhomogenous* diff. eq., when $f(x) \neq 0$
  - Most diff. eqs. in physics are linear
    * Then the sum of two solutions is a new, valid solution! (*superposition*)
    * Famous example: The Schrödinger eq. in quantum mechanics is linear → superposition of quantum states!

- Many approaches to finding a solution
  - **Shooting methods** (described quickly below)
  - **Finite difference methods** (project 1, described below)
  - **Finite elements methods** (not covered)

- Intution behind shooting methods
  - Turn a BVP into an *initial value problem* (know $u(x_0)$ and $u'(x_0)$)
  - Know $u(x_0)$
  - Guess $u'(x_0)$ and solve forward ("shoot")
    → first attempt for $u(x)$, let's call it $u_{(1)}(x)$

- Guess another $u'(x_0)$ and solve forward
  → second attempt for $u(x)$, let's call it $u_{(2)}(x)$
- Sum of solutions is a new solution (linearity):
  $u_c(x) = cu_{(1)}(x) + (1-c)u_{(2)}(x)$
- Require that $u_c(x_n) = u(x_n)$ (known from the second boundary condition)
- Use this to determine a value for c
  → final solution $u_c(x)$

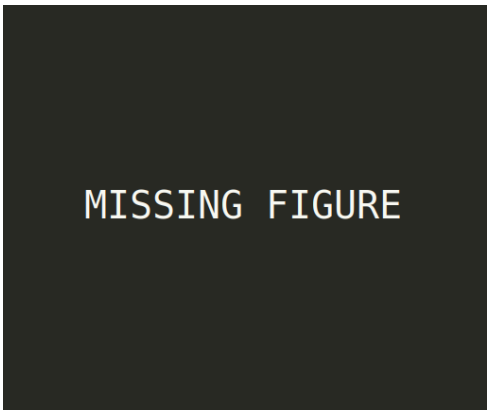

Figure 6: Shooting method

Finite difference method

TODO

Solving matrix equations with Gaussian elimination

TODO

Binary representation

TODO

Counting floating-point operations (FLOPs)

- Floating-point numbers, *floats*: (inexact) machine representation of the real numbers ($\mathbb{R}$)
- Floating-point operations: $\{+, -, \times, \div\}$ with floats
- Much slower than integer operations. (One FLOP consists of several integer operations.)
- Counting FLOPs is a way of estimating the efficiency of an algorithm
- *Note:* **FLOPs** (FLoating-point OPerations) vs **FLOPS** (FLoating-point OPerations per Second). FLOPS is a measure of *computer performance*, which we will not discuss in this course.

Examples

- Example 1:

$$y = ab + c, \quad \text{1 mult., 1 add.} \rightarrow \text{2 FLOPs}$$

- Example 2:

$$\begin{aligned} &\text{for } i = 1, \ldots, n: && n \text{ repetitions} \\ &\quad y_i = a y_{i-1} + i && \text{2 FLOPs} \\ & && \rightarrow 2n \text{ FLOPs} \end{aligned}$$

- Example 3:

$$\begin{aligned} &\text{for } i = 1, \ldots, n: && n \text{ repetitions} \\ &\quad y_i = \frac{a}{b} y_{i-1} + i && \text{3 FLOPs} \\ & && \rightarrow 3n \text{ FLOPs (silly!)} \end{aligned}$$

- A more efficient version of example 3:

$$\begin{aligned} &c = \frac{a}{b} && \text{1 FLOP} \\ &\text{for } i = 1, \ldots, n: && n \text{ repetitions} \\ &\quad y_i = c y_{i-1} + i && \text{2 FLOPs} \\ & && \rightarrow (2n+1) \text{ FLOPs} \approx 2n \text{ FLOPs} \end{aligned}$$

> *Tip:* When code speed is important, avoid recomputing constants within a loop.