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
    * I just like working with computers
- **Question:** What operating system are you using?
  - Linux?
  - macOS?

  - Windows?

- I will need at least two student representatives for the course evaluation
  - You'll join a meeting (~1 hour) with us teachers at the end of the semester, where we discuss what worked and what we can improve in the course
  - If you are willing to do this, just send me an email

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 (and learning by failing)
  - 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
  - Lectures are *not* recorded – if you miss a lecture, read the detailed lecture notes
- 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
  - We start the group sessions already the first week
- 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, ...)
  - *Almost* all questions are welcome: The only type of questions I don't like are the "questions" that aren't really questions at all, but just someone trying to show off how much they know.
  - 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

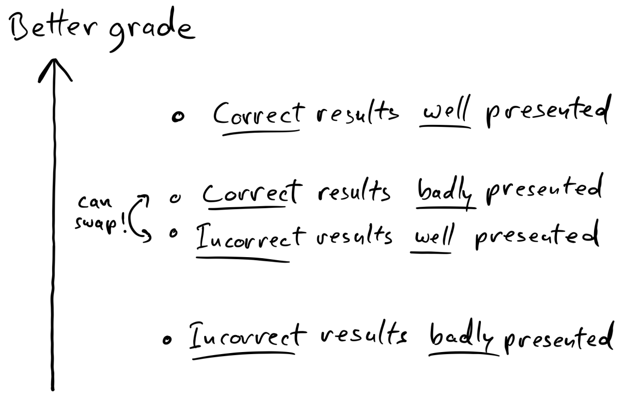

Figure 1: Presentation quality matters

- 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

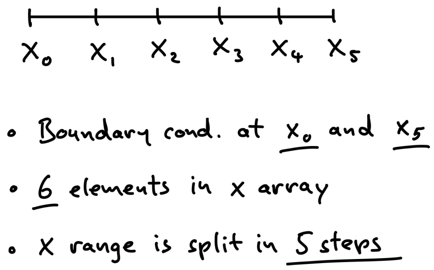

Figure 2: Sketch discretisations

- 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!

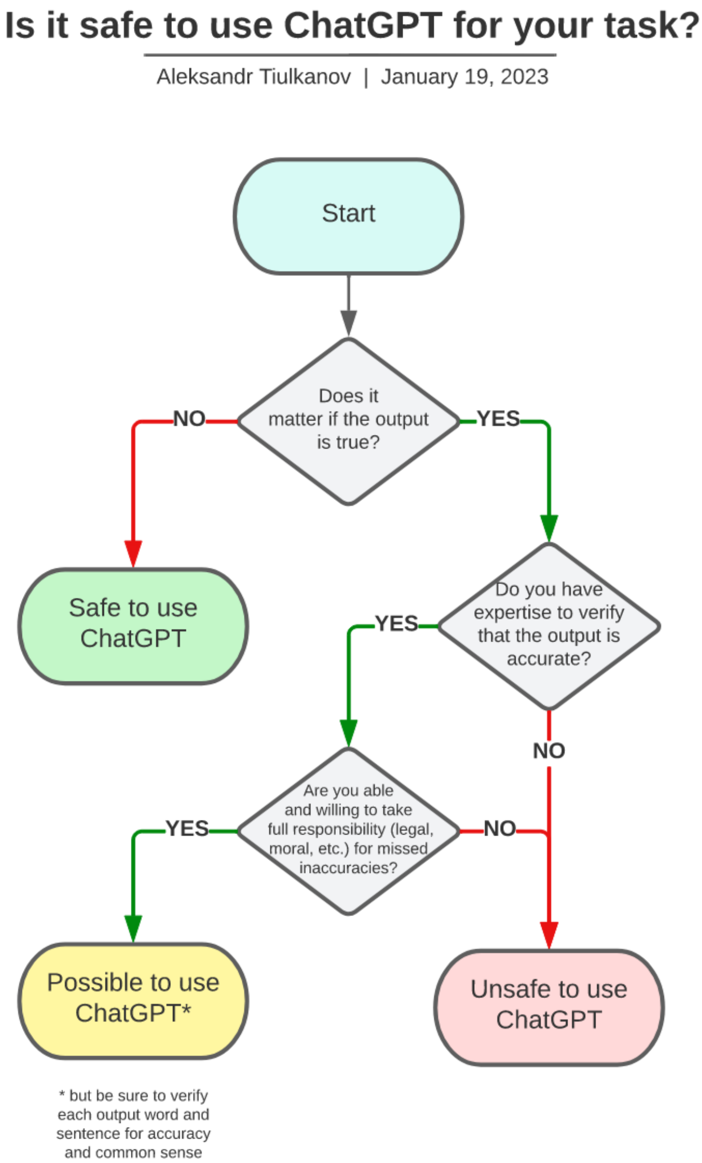

Figure 3: Example considerations to make before using ChatGPT or similar tools. Flowchart by A. Tiulkanov, included in the UNESCO report *ChatGPT and Artificial Intelligence in higher education*.

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!
  - In the first group sessions, spend some time going through the different introductory material on anderkve.github.io/FYS3150 before you start on project 1.
- Now let's introduce C++!
  - (Note that we won't have time in the lectures to talk about all C++ details you need for the projects.)
  - Intro: anderkve.github.io/FYS3150/book/introduction_to_cpp/intro
  - Hello World: anderkve.github.io/FYS3150/book/introduction_to_cpp/hello_world
  - Compiling and linking: anderkve.github.io/FYS3150/book/introduction_to_cpp/compiling_and_linking_take_1
  - Source files and header files: anderkve.github.io/FYS3150/book/introduction_to_cpp/source_files_and_header_files
  - Code structure: anderkve.github.io/FYS3150/book/introduction_to_cpp/code_structure
    * See also this example: github.com/anderkve/FYS3150/tree/master/code_examples/code_structure/example_1
  - Compilation and linking example with multiple files: github.com/anderkve/FYS3150/tree/master/code_examples/compilation_linking/example_1
  - *Strongly typed* languages (e.g. C++) vs *weakly typed* languages (e.g. Python).
    * anderkve.github.io/FYS3150/book/introduction_to_cpp/variables

- Write to file:
  anderkve.github.io/FYS3150/book/introduction_to_cpp/write_to_file
  * Also, remember that in cases with small output, simply *redirecting* terminal output into a file can be an easy and quick way to store output to a file – see the *Write terminal output to file* section of anderkve.github.io/FYS3150/book/using_the_terminal/basics

Topics in project 1

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

- Discretisation of a continuous problem, in this case the following boundary value problem (BVP):

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

- Mathematical approx. to second derivative (suitable for discretisation)
- Connection between a BVP and a 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

Discretisation of continuous functions

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

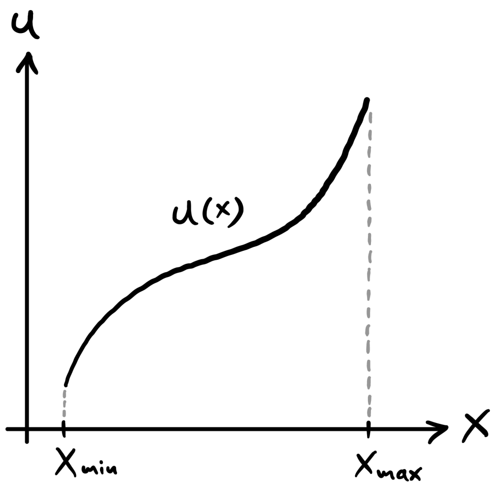

Figure 4: Continuous function

- Discretised representation

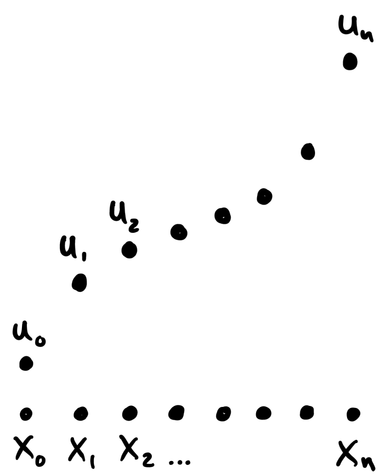

Figure 5: Discretised representation

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

$$x \to x_i$$
$$u(x) \to u(x_i) \equiv u_i$$
$$u(x \pm h) \to u(x_i \pm h) \equiv u_{i\pm1}$$

- 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: $h = x_1 - x_0 = \frac{x_2 - x_0}{2} = \ldots = \frac{x_n - x_0}{n}$.
  $(x_0 = x_{\text{min}}, x_n = x_{\text{max}})$
- 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:

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

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

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

Second derivative:

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

Derivation

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

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

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

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

$$
\begin{aligned}
u(x+h) &= u(x) + u'(x)h + \mathcal{O}\left(h^2\right) \\
\Rightarrow u'(x) &= \frac{u(x+h) - u(x) - \mathcal{O}\left(h^2\right)}{h}
\end{aligned}
$$

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

Discretise:

$$
\begin{aligned}
&u(x) \to u_i \\
&\Rightarrow u'_i = \frac{u_{i+1} - u_i}{h} + \mathcal{O}\left(h\right)
\end{aligned}
$$

(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

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

Discretise:

$$
\begin{aligned}
&u(x) \to u_i \\
&\Rightarrow u'_i = \frac{u_i - u_{i-1}}{h} + \mathcal{O}\left(h\right)
\end{aligned}
$$

(Two-point, backward difference)

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

  - Exact: $u'(x) = a_1 + 2a_2 x$
  - Approximation:

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

  - 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 $u(x-h)$:

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

  - Subtract:

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

  - Rearrange:

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

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

Discretise:

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

(Three-point expression)

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

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

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

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

Discretise:

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

In-lecture code discussion #2

- Hidden files on Unix systems
  - Hidden files have file names starting with a dot
  - Some relevant examples:
    * `.bashrc` and/or `.profile` in your home directory
    * `.gitignore` in your git repositories
  - Use the `-a` option to see hidden files in your file listings: `ls -a`
- Terminal-based text editors
  - Useful when you want to make quick file edits
  - Useful when you are logged into another system via a Unix terminal, e.g. if you are working on a supercomputer
  - Some people use the terminal-based editors as their main editors – can become very powerful and efficient tools
  - Two popular examples: `vim` and `nano`
- Short discussion of the `std::vector` class:
  anderkve.github.io/FYS3150/book/introduction_to_cpp/containers
  - Note the use of `my_vector.at(10)` as a safe alterative to `my_vector[10]` for accessing vector elements.
- How (not) to use `using namespace` in C++ programs:
  anderkve.github.io/FYS3150/book/introduction_to_cpp/source_files_and_header_files
- Integer vs floating-point division:
  - In Python, the statement `x = 7/10` will by default evaluate to `x = 0.7`
  - However, in C++ the statement `x = 7/10` will evaluate to `x = 0`
  - Since 7 and `10` are written as integers, C++ will do **integer division**
  - In integer division, it is correct that $7/10 = 0$
  - If we instead write `7.` and `10.`, C++ will treat these as floating-point numbers and perform floating-point division
  - So `x = 7./10.` will give the result `x = 0.7`
  - (The combinations `x = 7./10` and `x = 7/10.` will also give `x = 0.7`)
  - *Question:* Given the variable assignment `double x = 7/10;`, what value will `x` get?
  - *Answer:* `x` will be set to `x = 0.0`, since the assignment is evaluated as `double x = 0;`

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'', $\ldots$
  - *Second order* diff. eq., since the highest-order derivative is u''
  - *Inhomogenous* diff. eq., when $f(x) \neq 0$
  - Many 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:
  - We want to solve a boundary value problem (BVP), where we start with known $u(x_{\min})$ and $u(x_{\max})$
  - We'll do this by instead repeatedly solving an *initial value problem* (IVP), where we start with known $u(x_{\min})$ and $u'(x_{\min})$:
    * Start from the known $u(x_{\min})$

    - Guess a value for $u'(x_{\min})$
    - Solve the corresponding IVP forward ("shoot"). (We will discuss IVPs later in the course.)
    - Repeat the previous two steps until we find a solution $u(x)$ that hits the known boundary condition at $u(x_{\max})$
    - This solution $u(x)$ is then a solution to our original BVP
  - Things are easier when our diff. eq. is *linear*:
    - Guess a value for $u'(x_{\min})$, solve the IVP
      → let's call this solution $u_{(1)}(x)$
    - Guess another $u'(x_{\min})$, solve the IVP
      → let's call this solution $u_{(2)}(x)$
    - Since we have a linear diff. eq., a sum of solutions is a new solution:
      $u_c(x) = cu_{(1)}(x) + (1-c)u_{(2)}(x)$
    - Require that $u_c(x_{\max})$ should equal the known $u(x_{\max})$ (known from the second boundary condition)
    - Use this condition to determine a value for c
      → this $u_c(x)$ is then the solution $u(x)$ to our BVP
  - A drawback: Need to solve multiple IVPs to find the single solution to our BVP

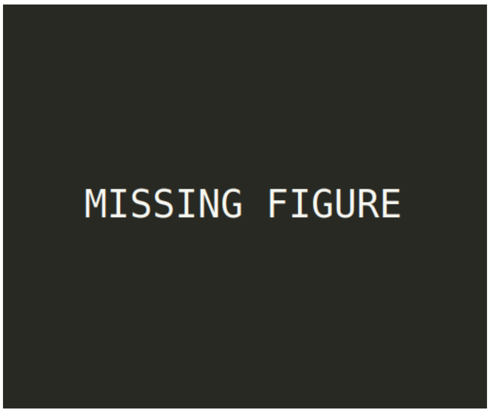

Figure 6: Shooting method

Finite difference method

- Our problem: Find the function $u(x)$ that solves this diff. eq.:

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

- We know $u(0)$, $u(1)$, $f(x)$ and that $x \in [0, 1]$

- Strategy:
  - **Step 1:** Express problem as a matrix eq.
  - **Step 2:** Solve the matrix eq.

Step 1: Express as matrix eq.

- Discretise equation:

$$-\left[\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \mathcal{O}\left(h^2\right)\right] = f_i, \quad f_i \equiv f(x_i)$$

- Approximate (leave out the $\mathcal{O}\left(h^2\right)$ terms) and change notation: $v_i \approx u_i$
- Arrange terms:

$$\boxed{-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i}$$

- Note: this is a collection of multiple equations, one for each value we can insert for i
- New goal: Determine $v_1, v_2, \ldots, v_{n-1}$
- We know: v_0, v_n and all the f_i
- Below we'll consider the special case with $n_{\text{steps}} = 5$
  - $v_0, v_1, v_2, v_3, v_4, v_5$: 6 points
  - v_0 and v_5 are known boundary points
  - 4 unknowns: v_1, v_2, v_3, v_4
  - $h = \frac{v_5 - v_0}{n_{\text{steps}}} = 0.2$ (very large, just for illustration)
- The boxed expression represents a set of four equations. Let's write them out in a suggestive manner…

$$\begin{array}{lcccccccc}
(i=1) & -v_0 & +2v_1 & -v_2 & & & & = & h^2 f_1 \\
(i=2) & & -v_1 & +2v_2 & -v_3 & & & = & h^2 f_2 \\
(i=3) & & & -v_2 & +2v_3 & -v_4 & & = & h^2 f_3 \\
(i=4) & & & & -v_3 & +2v_4 & -v_5 & = & h^2 f_4
\end{array}$$

- v_0 and v_5 are known – let's move them over to the right-hand side and define some simpler notation g_1, g_2, g_3, g_4:

$$
\begin{array}{ccccccccc}
+2v_1 & -v_2 & & & = & h^2 f_1 + v_0 & \equiv & g_1 \\
-v_1 & +2v_2 & -v_3 & & = & h^2 f_2 & \equiv & g_2 \\
& -v_2 & +2v_3 & -v_4 & = & h^2 f_3 & \equiv & g_3 \\
& & -v_3 & +2v_4 & = & h^2 f_4 + v_5 & \equiv & g_4
\end{array}
$$

- This can be written as

$$
\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}
\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}
=
\begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix}
$$

$$\boxed{\mathbf{A}\vec{v} = \vec{g}}$$

- $\mathbf{A}$ and $\vec{g}$ are known, we want to solve for $\vec{v}$
- Note that $\mathbf{A}$ is a **tridiagonal matrix**.
- The *diagonal* has only 2's, while the *superdiagonal* and *subdiagonal* contain only -1's.
- Note that the vector $\vec{v} = [v_1, v_2, v_3, v_4]$ in this equation only contains the *unknown* v_i. The known values at the boundaries, v_0 and v_5, are *not* included in $\vec{v}$.

Step 2: Solve the matrix eq.

- Overview of things we'll discuss:
  1. *Now:* Method for solving $\mathbf{A}\vec{v} = \vec{g}$ when $\mathbf{A}$ is a *general*, *tridiagonal* matrix
     - Gaussian elimination turns into the Thomas algorithm
  2. *A task for you in Project 1:* Method for solving $\mathbf{A}\vec{v} = \vec{g}$ when $\mathbf{A}$ is the *special, tridiagonal* matrix above (with only -1's and 2's along the diagonals)
  3. *Later in the course:* Methods for solving a general matrix equation $\mathbf{A}\vec{x} = \vec{b}$
     - Gaussian elimination, LU decomposition, iterative methods

Matrix equations: Gaussian elimination and the Thomas algorithm

Introduction

- A matrix equation $\mathbf{A}\vec{x} = \vec{b}$ ($\mathbf{A}$ and $\vec{b}$ known, $\vec{x}$ unknown) represents a set of linear equations

$$\begin{array}{rccccccccc}
(\text{eq. } 1) & a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\
(\text{eq. } 2) & a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\
(\dots) & & & & & & & & & \\
(\text{eq. } m) & a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m
\end{array}$$

- m equations, each with n terms — one for each unknown variable $(x_1, \dots, x_n)$

$$\underset{(m\times n)}{\mathbf{A}}\underset{(n\times 1)}{\vec{x}} = \underset{(m\times 1)}{\vec{b}}$$

- We will focus on the case of a *square* matrix, i.e. when $m = n$
- This means we have n equations and n unknowns
- If all our equations are *linearly independent*, i.e. when each equation represents information not contained in the other equations, we should be able to solve for all our n unknowns $(x_1, \dots, x_n)$
- Some equivalent statements:
  - All the equations are linearly independent
  - $\mathbf{A}$ is *not* singular (all eigenvalues of $\mathbf{A}$ are non-zero)
  - $\det \mathbf{A} \neq 0$

Side note: When we have *more* equations (constraints) than unknowns, there is generally *no exact solution*. But we can *fit* our unknowns such that all our equations are as close to solved as possible. This is the typical case in science: you have a model with a few free parameters and the model needs to match many observations (constraints) as closely as possible.

Gaussian elimination, overview

- Start from general matrix equation

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} = \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

- **Step 1:** Forward substitution/elimination
  - Turn matrix into upper-triangular form

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet \\ & & \bullet & \bullet \\ & & & \bullet \end{bmatrix} \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} = \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

  - Can then read off solution for x_m (last row)
- **Step 2:** Back substitution/elimination
  - Use the now known x_m to find x_{m-1}, then use these to find x_{m-2}, and so on
  - End up with this

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix} \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} = \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

Thomas algorithm: Gaussian elimination on a tridiagonal matrix

- Let's go back to the notation of project 1: $\mathbf{A}\vec{v} = \vec{g}$
- Let $\mathbf{A}$ be a *general* tridiagonal matrix
  - For concreteness we look at the case with a 4×4 matrix:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix}$$

- Note that we have used indices that correspond to the row numbers:

- Subdiagonal: $\vec{a} = [a_2, a_3, a_4]$
- Diagonal: $\vec{b} = [b_1, b_2, b_3, b_4]$
- Superdiagonal: $\vec{c} = [c_1, c_2, c_3]$

- Now let's do **step 1**, the *forward substitution*
  - We start from the *augmented matrix*:

$$\begin{array}{lcccc|c} R_1: & b_1 & c_1 & 0 & 0 & g_1 \\ R_2: & a_2 & b_2 & c_2 & 0 & g_2 \\ R_3: & 0 & a_3 & b_3 & c_3 & g_3 \\ R_4: & 0 & 0 & a_4 & b_4 & g_4 \end{array}$$

  - To move towards an upper-triagonal form, we want to set the a_2 entry in R_2 to 0
  - Use a row operation with R_1 to achieve this: $R_2 \to R_2 - \frac{a_2}{b_1} R_1$
  - This turns the a_2 entry into $a_2 - \frac{a_2}{b_1} b_1 = 0$

$$\begin{array}{lcccc|c} R_1: & b_1 & c_1 & 0 & 0 & g_1 \\ R_2: & 0 & (b_2 - \frac{a_2}{b_1} c_1) & c_2 & 0 & (g_2 - \frac{a_2}{b_1} g_1) \\ R_3: & 0 & a_3 & b_3 & c_3 & g_3 \\ R_4: & 0 & 0 & a_4 & b_4 & g_4 \end{array}$$

  - Introduce shorthand notation:
    * $\tilde{b}_1 = b_1$
    * $\tilde{b}_2 = b_2 - \frac{a_2}{\tilde{b}_1} c_1$
    * $\tilde{g}_1 = g_1$
    * $\tilde{g}_2 = g_2 - \frac{a_2}{\tilde{b}_1} \tilde{g}_1$
  - We then have

$$\begin{array}{lcccc|c} R_1: & \tilde{b}_1 & c_1 & 0 & 0 & \tilde{g}_1 \\ R_2: & 0 & \tilde{b}_2 & c_2 & 0 & \tilde{g}_2 \\ R_3: & 0 & a_3 & b_3 & c_3 & g_3 \\ R_4: & 0 & 0 & a_4 & b_4 & g_4 \end{array}$$

  - Now continue in the same way to turn the a_3 entry to zero
  - Row operation: $R_3 \to R_3 - \frac{a_3}{\tilde{b}_2} R_2$
  - Define notation:

    * $\tilde{b}_3 = b_3 - \frac{a_3}{\tilde{b}_2} c_2$
    * $\tilde{g}_3 = g_3 - \frac{a_3}{\tilde{b}_2} \tilde{g}_2$

  - We then get

$$
\begin{array}{lcccc|c}
R_1: & \tilde{b}_1 & c_1 & 0 & 0 & \tilde{g}_1 \\
R_2: & 0 & \tilde{b}_2 & c_2 & 0 & \tilde{g}_2 \\
R_3: & 0 & 0 & \tilde{b}_3 & c_3 & \tilde{g}_3 \\
R_4: & 0 & 0 & a_4 & b_4 & g_4
\end{array}
$$

  - And once more, with feeling...
  - Row operation: $R_4 \to R_4 - \frac{a_4}{\tilde{b}_3} R_3$
  - Define notation:
    * $\tilde{b}_4 = b_4 - \frac{a_4}{\tilde{b}_3} c_3$
    * $\tilde{g}_4 = g_4 - \frac{a_4}{\tilde{b}_3} \tilde{g}_3$

$$
\begin{array}{lcccc|c}
R_1: & \tilde{b}_1 & c_1 & 0 & 0 & \tilde{g}_1 \\
R_2: & 0 & \tilde{b}_2 & c_2 & 0 & \tilde{g}_2 \\
R_3: & 0 & 0 & \tilde{b}_3 & c_3 & \tilde{g}_3 \\
R_4: & 0 & 0 & 0 & \tilde{b}_4 & \tilde{g}_4
\end{array}
$$

- The forward substitution is now done! Here's the summary:

Forward substitution:

$$
\begin{aligned}
\tilde{b}_1 &= b_1 \\
\tilde{b}_i &= b_i - \frac{a_i}{\tilde{b}_{i-1}} c_{i-1} \qquad i = 2, 3, 4 \\
\tilde{g}_1 &= g_1 \\
\tilde{g}_i &= g_i - \frac{a_i}{\tilde{b}_{i-1}} \tilde{g}_{i-1} \qquad i = 2, 3, 4
\end{aligned}
$$

The lecture on August 31 ended here.

- Now let's do **step 2**, the *back substitution*

- Starting point

$$
\begin{array}{lcccc|c}
R_1: & \tilde{b}_1 & c_1 & 0 & 0 & \tilde{g}_1 \\
R_2: & 0 & \tilde{b}_2 & c_2 & 0 & \tilde{g}_2 \\
R_3: & 0 & 0 & \tilde{b}_3 & c_3 & \tilde{g}_3 \\
R_4: & 0 & 0 & 0 & \tilde{b}_4 & \tilde{g}_4
\end{array}
$$

- We now want to get to an identity matrix form, starting from the bottom row
- Row operation: $R_4 \rightarrow \frac{R_4}{\tilde{b}_4}$

$$
\begin{array}{lcccc|cc}
R_1: & \tilde{b}_1 & c_1 & 0 & 0 & \tilde{g}_1 & \\
R_2: & 0 & \tilde{b}_2 & c_2 & 0 & \tilde{g}_2 & \\
R_3: & 0 & 0 & \tilde{b}_3 & c_3 & \tilde{g}_3 & \\
R_4: & 0 & 0 & 0 & 1 & \frac{\tilde{g}_4}{\tilde{b}_4} & \rightarrow v_4
\end{array}
$$

- We now have the solution for v_4:

$$
\boxed{v_4 = \frac{\tilde{g}_4}{\tilde{b}_4}}
$$

- Now we want to get R_3 on the form (0,0,1,0)
- We can subtract $c_3 R_4$ to get rid of the c_3 entry in R_3, and then divide by $\tilde{b}_3$ to set the third element to 1
- Row operation: $R_4 \rightarrow \frac{R_3 - c_3 R_4}{\tilde{b}_3}$

$$
\begin{array}{lcccc|cc}
R_1: & \tilde{b}_1 & c_1 & 0 & 0 & \tilde{g}_1 & \\
R_2: & 0 & \tilde{b}_2 & c_2 & 0 & \tilde{g}_2 & \\
R_3: & 0 & 0 & 1 & 0 & \frac{\tilde{g}_3 - c_3 v_4}{\tilde{b}_3} & \rightarrow v_3 \\
R_4: & 0 & 0 & 0 & 1 & v_4 &
\end{array}
$$

- This gives us the solution for v_3:

$$
\boxed{v_3 = \frac{\tilde{g}_3 - c_3 v_4}{\tilde{b}_3}}
$$

- We can continue upwards like this to find all the remaining v_i. In summary:

Back substitution:

$$v_4 = \frac{\tilde{g}_4}{\tilde{b}_4}$$

$$v_i = \frac{\tilde{g}_i - c_i v_{i+1}}{\tilde{b}_i} \qquad i = 3, 2, 1$$

- Let's summarise what we've done:
  - Given a general tridiagonal matrix $\mathbf{A}$ and a vector $\vec{g}$, we have found the vector $\vec{v}$ that solves the equation $\mathbf{A}\vec{v} = \vec{g}$.
  - We used Gaussian elimination, which has two steps:
    * *forward substitution*
    * *back substitution*
  - Because $\mathbf{A}$ was tridiagonal, the Gaussian elimination procedure resulted in a fairly simple algorithm, which is known as the **Thomas algorithm** (Llewellyn Thomas, 1903–1992)

Coding tip: Note that we don't need to work with an entire matrix in memory here. To implement the Thomas algorithm above, we just need some arrays/vectors $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{g}$, $\vec{\tilde{b}}$, $\vec{\tilde{g}}$ and $\vec{v}$.

Back to our boundary value problem

- We now have the tools we need to use a **finite difference method** to solve a boundary value problem like

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

where $f(x)$ is some known function, and we know $u(x_{\min})$ and $u(x_{\max})$

  - Discretise the problem, using a discretised approximation for the second derivative
    * At this step we changed notation $u_i \to v_i$
  - Formulate the resulting set of equations as a matrix equation $\mathbf{A}\vec{v} = \vec{g}$
    * The second derivative in the diff. eq. → the matrix $\mathbf{A}$ will be tridiagonal, with a simple (-1,2,-1) form
  - Use the Thomas algorithm to solve the matrix equation
    * **However**, the Thomas algorithm is a method that can solve *any* tridiagonal matrix equation, but in the case of our BVP we are only interested in the case of a particularly

simple, tridiagonal matrix. This means that we can simplify the Thomas algorithm for our usecase – something you will do in project 1.

- *A reasonable question:* Why are we doing all this? Why not rather find $\mathbf{A}^{-1}$ and solve the equation as $\vec{v} = \mathbf{A}^{-1}\vec{g}$?
  - Finding $\mathbf{A}^{-1}$ numerically takes $\mathcal{O}\left(n^3\right)$ operations for an $n \times n$ matrix. This approach becomes useful if we need to solve *many* different equations ($\mathbf{A}\vec{v}_1 = \vec{g}_1, \mathbf{A}\vec{v}_2 = \vec{g}_2, \ldots$) that all involve the same matrix $\mathbf{A}$. But for solving a single equation $\mathbf{A}\vec{v} = \vec{g}$, other methods are quicker.

Counting floating-point operations (FLOPs)

- Floating-point numbers, *floats*: (inexact) machine representation of the real numbers ($\mathbb{R}$)
- Floats are numbers where the decimal point can be placed anywhere (it can "float") in a given string of digits, depending on which number we need to represent
  - Example: The digits `112358` can represent `11.2358` or `1123.58`, depending on the placement of the decimal point
- 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.
  - So how long a given task will take on a given computer will depend both on the number of **FLOPs** required for the task *and* the number of **FLOPS** for the computer – and a bunch of other things…

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:

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

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

Binary representation

- *In short:* How to represent numbers using only two different symbols
- Basic element: a **bit**
  - 1/0, on/off, true/false, yes/no, hole/not-hole (punched cards), red/blue, ...
  - The term *bit* is originally a contraction of *binary information digit*
- A bit doesn't have to be related to computers – it's a basic concept from information theory
  - A bit is the expected amount of *information* or *surprise* contained in the outcome of a 50/50 random draw. (The more surprising a result/message/signal is, the more information it contains – look up litterature on *Shannon entropy* for more on this.)
- In principle, any physical system with two possible states can be used to represent the digits 0 and 1
- So we better use a numeral system that only needs two different digits to represent any number → the **binary system** or the **base 2 system**
- In base 10, we have ten different symbols (0–9) that can be used per position
- In base 2, we only have two different symbols per position
  → need to use more positions to express numbers
  → longer strings of symbols compared to the decimal system
- *Side note:* In computing and mathematics we also sometimes encounter the *hexadecimal* (base 16) system. In this case there are 16 different symbols (0–9 and A–F) per position.

Integers

- Example: 137 in base 10 and base 2: $(137)_{10} = (10001001)_2$
- Representation in the decimal (base 10) system:

$$
\begin{aligned}
(137)_{10} = \frac{10^2 \quad\; 10^1 \quad\; 10^0}{1 \quad\;\; 3 \quad\;\; 7} &= (1 \times 10^2) + (3 \times 10^1) + (7 \times 10^0) \\
&= 100 + 30 + 7 \\
&= 137
\end{aligned}
$$

- Representation in the binary (base 2) system:

$$
\begin{aligned}
(10001001)_2 &= \frac{2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0}{1 \quad\ 0 \quad\ 0 \quad\ 0 \quad\ 1 \quad\ 0 \quad\ 0 \quad\ 1} \\
&= (1 \times 2^7) + (0 \times 2^6) + \ldots + (1 \times 2^3) + \ldots + (1 \times 2^0) \\
&= 128 + 0 + 0 + 0 + 8 + 0 + 0 + 1 \\
&= 137
\end{aligned}
$$

- How can we find the correct binary string of 0's and 1's for a given number?
  - The same way we (without thinking about it) identify the correct string of digits in the decimal system: by doing *integer division and keeping track of remainders*

	Remainder	Position
$137 \setminus 10 = 13$	7	10^0
$13 \setminus 10 = 1$	3	10^1
$1 \setminus 10 = 0$	1	10^2

Table 1: Repeated integer division with 10 produces the base 10 representation of 137.

	Remainder	Position
$137 \setminus 2 = 68$	1	2^0
$68 \setminus 2 = 34$	0	2^1
$34 \setminus 2 = 17$	0	2^2
$17 \setminus 2 = 8$	1	2^3
$8 \setminus 2 = 4$	0	2^4
$4 \setminus 2 = 2$	0	2^5
$2 \setminus 2 = 1$	0	2^6
$1 \setminus 2 = 0$	1	2^7

Table 2: Repeated integer division with 2 produces the base 2 representation of 137.

- The more bits we have available, the longer the the integer we can store
- If we are working with *signed* integers, we need one additional bit to represent the sign: $(-1)^0$ or $(-1)^1$

Floating-point numbers

- How to represent the real numbers ($\mathbb{R}$) in binary?

- Strategy: use *normalised, scientific notation in base 2*
- Example in decimal:

$$-9.90625 \times 10^0, \text{ or}$$
$$-0.990625 \times 10^1$$

- The latter convention, where the first digit is always zero, is often used in computing
- General form:

$$\pm \left[\text{number in } \left(\tfrac{1}{10}, 1\right)\right] \times 10^{[\text{integer exponent}]}$$

- In binary (base 2):

$$\pm \left[\text{number in } \left(\tfrac{1}{2}, 1\right)\right] \times 2^{[\text{integer exponent}]}$$

- Terminology:

$$[\text{sign}][\text{mantissa}] \times 2^{[\text{exponent}]}$$

- Whether the mantissa should be a number within $(\frac{1}{2}, 1)$ or within $(1, 2)$ is a matter of convention
- Another common term for the mantissa is the **significand**
- We already know how to represent the integer exponent and the sign bit in binary
- Binary representation of the mantissa:
  - Example: $(0.5625)_{10}$

$$\begin{aligned}
(0.1001)_2 &= \frac{2^0 \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4}}{0 \quad\;\; 1 \quad\;\; 0 \quad\;\; 0 \quad\;\; 1} \\
&= (0 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4}) \\
&= 0 + 0.5 + 0 + 0 + 0.0625 \\
&= 0.5625
\end{aligned}$$

- **Single precision**: Using 32 bits (4 bytes) to represent a floating-point number
  - Sign: 1 bit
  - Exponent: 8 bits
  - Mantissa: 23 bits
- Example: The number -3.25
  - In normalised, scientific notation in base 2: -0.8125×2^2
    * Sign: -1 (so the *sign bit* will be 1 since $-1 = (-1)^1$)

    * Exponent: 2
    * Mantissa: 0.8125

- In memory, something like this:

$$\underset{1}{\underline{\text{sign bit}}} \quad \underset{00000010}{\underline{\text{8-bit exponent (2)}}} \quad \underset{1101000\ldots000}{\underline{\text{23-bit mantissa (0.8125)}}}$$

- On most computer systems, the type `float` in C++ will correspond to a 32-bit number
- **Double precision**: Using 64 bits (8 bytes) to represent a floating-point number
    - Sign: 1 bit
    - Exponent: 11 bits
    - Mantissa: 52 bits
- On most systems, the type `double` in C++ will correspond to a 64-bit number
- An 11-bit exponent gives an exponent range of $(-1024, 1024)$, since $2^{11} = 2048$
    - Since $2^{1024} \approx 10^{308}$, the range of numbers that can be represented in double-precision is roughly $(10^{-308}, 10^{308})$
    - You can test this quickly in your Python terminal, since a floating-point number in Python (the `float` type in Python) by default will be a 64-bit number on most systems:

```
>>> 2.**1023
8.98846567431158e+307
>>>
>>> 2.**1024
OverflowError: (34, 'Numerical result out of range')
```

- Finite number of bits → unavoidable problems with range and accuracy
- Limited number of bits for the **exponent**:
  → a limited **range** of $\mathbb{R}$ can be represented
    - With 11 bits for the exponent, we get a range of $\sim (10^{-308}, 10^{308})$
- Limited number of bits for the **mantissa**:
  → a limited **resolution/precision** in our representation of the continuous $\mathbb{R}$
    - With 52 bits for the mantissa, we get a precision of around **15 digits** in the decimal system $(\log_{10}(2^{52}) \approx 15.654)$

Hidden bit: When using normalised, scientific notation in base 2, we know that the most significant digit, i.e. the first digit of the mantissa, will always be 0 (if the $(\frac{1}{2}, 1)$-convention is used), or always be 1 (if the $(1, 2)$-convention is used). So we don't need to explicitly store this bit in memory. This trick is referred to as the *hidden bit*, and it effectively increases the mantissa precision by one bit, e.g. from 52 bits to 53 bits for a double-precision number.

- *A silly example to illustrate the effect of limited range and precision*:
  - Let's work in base 10
  - Assume we only had memory for *one digit in the exponent* and *one digit in the mantissa*
  - We could then only represent these numbers:

$$\ldots, 1 \times 10^{-1}, 2 \times 10^{-1}, \ldots, 1 \times 10^{0}, 2 \times 10^{0}, \ldots, 1 \times 10^{1}, 2 \times 10^{1}, \ldots$$

  - We would have a range of $\sim (10^{-5}, 10^{5})$
  - The only numbers we would be able to use would be

$$\ldots, 0.1, 0.2, 0.3, \ldots, 1, 2, 3, \ldots, 10, 20, 30, \ldots, 100, 200, 300, \ldots$$

  - So a number 17 would just end up as 10, and a computation like 100 + 80 would just give the result 100

Errors

Truncation errors

- Truncation errors are purely mathematical in origin
- Typical case: we cut of a series expansion at some point
- Example: leaving out the $\mathcal{O}\left(h^2\right)$ terms in

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

- *Note*: here a smaller step size h will give a smaller truncation error

Roundoff errors

- Numbers can only be stored with limited accuracy
- For *doubles*, our precision is ~15 digits
  - We will often refer to this as our *machine precision*
- So *almost* all numbers we store are **approximations** to the true number we intended to store
  - True number: a
  - Floating-point representation of a: $fl(a)$
  - Given a true a, your $fl(a)$ will be in the range

$$a(1 - \delta_m) < fl(a) < a(1 + \delta_m)$$

    where δ_m is the machine precision (e.g. $\delta_m \sim 10^{-15}$)
  - So given $fl(a)$, all you know is that the true number a is in the range

$$fl(a)(1 - \delta_m) < a < fl(a)(1 + \delta_m)$$

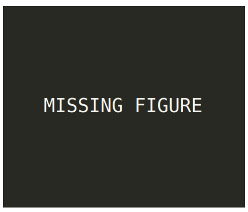

Figure 7: Illustration of the continuous real number line and the discrete floating-point representation

The lecture on September 1 ended here.

Loss of numerical precision

- Also known as **loss of significance**
- Typical case: subtraction with similar numbers
  → we loose the most significant digits, left with digits that are more affected by roundoff errors
- Example:
  - True values: $a = 1.0054321$, $b = 1.0040001$
  - Assume a machine precision of $\delta_m \sim 10^{-4}$ (just for illustration)
  - Approximate floating-point representations: $fl(a) = 1.005$, $fl(b) = 1.004$
  - 4 significant digits
  - Relative errors in the approximations:

$$\left|\frac{a - fl(a)}{a}\right| \approx 10^{-4}$$

$$\left|\frac{b - fl(b)}{b}\right| \approx 10^{-7}$$

  - So $fl(a)$ and $fl(b)$ are clearly very reasonable approximations to a and b, given our assumed machine precision
  - Now perform a subtraction:
  - True value: $a - b = 0.0014320$
  - Approximate: $fl(a) - fl(b) = 1.005 - 1.004 = 0.001$
  - Now we only have 1 significant digit!
  - Relative error:

$$\left|\frac{0.0014320 - 0.001}{0.0014320}\right| \approx 3 \times 10^{-1}$$

  - Suddenly we have a **30% error**, even though our input numbers where reasonble representations of the true values

  - Such **loss of precision** can easily happen in the middle of some long, complicate computation, and then all subsequent computations will end up with a large error.
  - Another common term for this is **catastrophic cancellation**

- Note that when discussing errors, we are usually most interested in the *relative* error:
- Example from project 1:
  - Absolute error: $\Delta = |v_i - u_i|$
  - Relative error: $\epsilon = |\frac{v_i - u_i}{u_i}|$
  - It may be useful to e.g. study plots or tables of $\log_{10}(\epsilon)$ vs $\log_{10}(h)$
- Typical case for us:
  - If the step size is large: truncation error dominates
  - If the step size is tiny: roundoff errors lead to loss of precision → garbage results
  - So we expect that there is some optimal, intermediate step size that gives the smallest overall error

An example error analysis

> This topic includes a small in-lecture code discussion, using the `error_analysis` code example.

- Consider the function $u(x) = e^{2x}$
  - (We choose this example function just because it's trivial to differentiate many times, and the 2 in the exponent ensures that all the derivatives are not exactly equal.)
- We will use our familiar expression

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}$$

  to implement a computer program that computes (an approximation to) the true second derivate u_i'' at a given point x_i

- *Our question*: How do we expect that the *relative error* of our code output will depend on our choice of step size h?
- First of all, we know the exact answer for u_i'':

$$u_i'' = 4e^{2x_i}$$

- In what follows we will first consider the **absolute error**, and then later the **relative error**

  – Absolute error:

$$\Delta(h) \equiv |\text{approx.} - \text{true}| = \left| \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} - u_i'' \right|$$

  – Relative error:

$$\epsilon(h) \equiv \left| \frac{\text{approx.} - \text{true}}{\text{true}} \right| = \left| \frac{\Delta(h)}{u_i''} \right|$$

- Now, as a first step towards answering our question, let's construct a simple *model* for the absolute error $\Delta(h)$
- We will assume that $\Delta(h)$ is the sum of two contributions, namely a truncation error $\Delta_{\text{tr}}(h)$ and a roundoff error $\Delta_{\text{ro}}(h)$:

$$\Delta(h) = \Delta_{\text{tr}}(h) + \Delta_{\text{ro}}(h)$$

- Let's first look at the truncation error:

$$\begin{aligned}
\Delta_{\text{tr}}(h) &= \left| \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} - u_i'' \right| \\
&= \left| \left(\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} \right) - \left(\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + \mathcal{O}\left(u_i^{(4)} h^2\right) \right) \right| \\
&= \left| \mathcal{O}\left(u_i^{(4)} h^2\right) \right|
\end{aligned}$$

  – Note that here we have included in our big-O notation that the leading term is not only proportional to h^2, but also to the fourth derivative, $u_i^{(4)}$. (To see this, go back to our derivation of the discretised expression for the second derivative.)
  – It is useful here to keep track of this dependence on the fourth derivative, since for other choices of the example function $u(x)$ the different-order derivatives at x_i could have vastly different values – or indeed be zero, if our $u(x)$ was a low-order polynomial.

- Now let's look at the roundoff error:

  – What we *want* to compute is

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}$$

or to rewrite it slightly,

$$\frac{(u_{i+1} - u_i) - (u_i - u_{i-1})}{h \times h}$$

- However, the computation we *actually* end up performing on the computer is something like this:

$$fl\left[\frac{fl\Big[fl\Big(fl(u_{i+1}) - fl(u_i)\Big) - fl\Big(fl(u_i) - fl(u_{i-1})\Big)\Big]}{fl\Big(fl(h) \times fl(h)\Big)}\right]$$

- Consider the limit of small h and focus on the subtractions of near identical numbers

$$fl(u_{i+1}) - fl(u_i)$$

and similar for $fl(u_i) - fl(u_{i-1})$

- Recall:

$$a(1 - \delta_m) < fl(a) < a(1 + \delta_m)$$

- We can then estimate an upper bound for the result of the subtraction

$$\begin{aligned} fl(u_{i+1}) - fl(u_i) &\leq u_{i+1}(1 + \delta_m) - u_i(1 - \delta_m) \\ &= (u_{i+1} - u_i) + (u_{i+1} + u_i)\delta_m \end{aligned}$$

- In the limit $h \to 0$, i.e. when $u_{i+1} \to u_i$, the first parenthesis vanishes, but the second parenthesis does not
- So we are left with

$$fl(u_{i+1}) - fl(u_i) \leq \mathcal{O}\left(u_i \delta_m\right)$$

- *What this means*: While we know that the *true* value of $u_{i+1} - u_i$ goes to 0 when $h \to 0$, we have no guarantee that our actual computation $fl(u_{i+1}) - fl(u_i)$ will go to exactly 0 in this limit.
- Assuming that this subtraction is the most "dangerous" part in our computation of

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}$$

we can estimate the roundoff error Δ_{ro} to be

$$\Delta_{\text{ro}}(h) = \mathcal{O}\left(\frac{u_i \delta_m}{h^2}\right)$$

- We can now put everything together in our simple model for the absolute error:

$$\begin{aligned}\Delta(h) &= |\Delta_{\text{tr}}(h) + \Delta_{\text{ro}}(h)| \\ &= \left|\mathcal{O}\left(u_i^{(4)} h^2\right) + \mathcal{O}\left(\frac{u_i \delta_m}{h^2}\right)\right|\end{aligned}$$

- Our model for the *relative* error ϵ in our computation of u_i'' then becomes

$$\begin{aligned}\epsilon(h) &= \left|\frac{\Delta(h)}{u_i''}\right| \\ &= \left|\mathcal{O}\left(\frac{u_i^{(4)}}{u_i''} h^2\right) + \mathcal{O}\left(\frac{u_i \delta_m}{u_i''} \frac{1}{h^2}\right)\right|\end{aligned}$$

  - The first term grows when h *increases*, while the second term grows when h *decreases*
  - For our choice of example function we also know that $\mathcal{O}(u_i) \approx \mathcal{O}(u_i'') \approx \mathcal{O}\left(u_i^{(4)}\right)$, so the factors $\frac{u_i^{(4)}}{u_i''}$ and $\frac{u_i}{u_i''}$ won't suppress or enlarge the error terms much
- Let's now look at $\log_{10} \epsilon(h)$
  - Collecting the stuff that doesn't depend on h in two constants C_1 and C_2, we can write

$$\log_{10} \epsilon(h) = \log_{10} \left|C_1 h^2 + C_2 h^{-2}\right|$$

  - Look at the behaviour in the limits $h \to \infty$ (first term dominates) and $h \to 0$ (second term dominates)

$$\log_{10} \epsilon(h) \approx \begin{cases} -2 \log_{10} h + \log_{10} C_2 & \text{for } h \to 0, \text{ i.e. } \log_{10} h \to -\infty \\ 2 \log_{10} h + \log_{10} C_1 & \text{for } h \to \infty, \text{ i.e. } \log_{10} h \to \infty \end{cases}$$

  - Note that these are the equations for two straight lines (slopes 2 and -2) in a plot of $\log_{10} \epsilon(h)$ vs $\log_{10} h$
  - So we see that our model for the error suggests the qualitative behaviour we expected, namely that there should be some *optimal, intermediate choice for the step size* that gives the smallest overall error

- We also see that we get a quantitative prediction for how quickly the error will grow when we move far away from the optimal step size choice

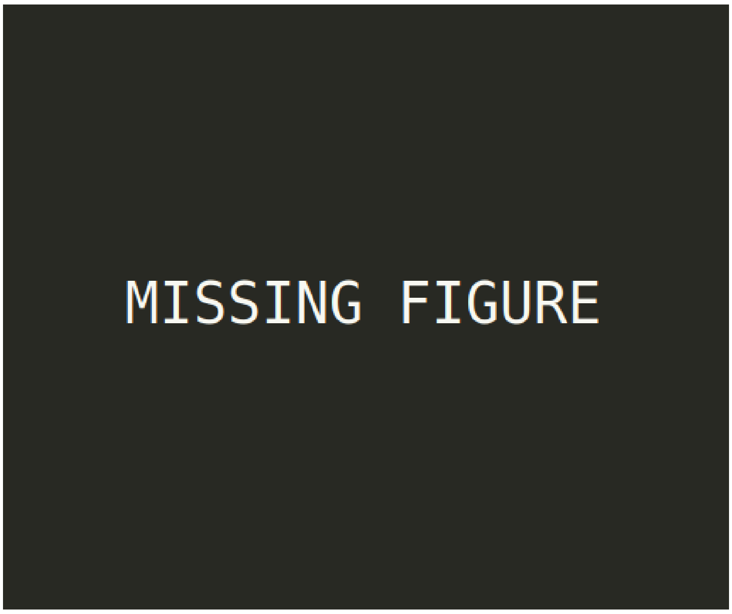

Figure 8: Sketch of a $\log_{10} \epsilon(h)$ vs $\log_{10} h$ plot, as suggested by our simple error model

Recap: solving matrix equations

- We have discussed how to solve matrix equations $\mathbf{A}\vec{x} = \vec{b}$
  - (We used the notation $\mathbf{A}\vec{v} = \vec{g}$ in project 1)
- **Gaussian elimination**:
  - Can be used to solve $\mathbf{A}\vec{x} = \vec{b}$ for a *general* (dense) $\mathbf{A}$
  - In that case it requires $\mathcal{O}\left(n^3\right)$ FLOPs, or more accurately $\mathcal{O}\left(\frac{2}{3}n^3\right)$
  - We have only looked at the special case for a *tridiagonal* $\mathbf{A}$ (more efficient)
- Next up: **LU decomposition**
- Later: **Iterative methods**

Classification of methods for solving matrix equations

Direct methods

- Examples:
  - Gaussian elimination
  - LU decomposition
- In theory, these methods give the *exact* answer in a *finite number of steps*
- In practice, these methods can suffer from numerical instabilities
- They typically work with the entire matrix at once
  → keeps the full matrix stored in memory

Indirect methods

- Examples:
  - Jacobi's iterative method
  - Gauss-Seidel
  - Relaxation methods
- *Iterate* closer and closer to the exact answer, but *will never get there exactly*
- Can often work without keeping the full matrix in memory
- Are often less susceptible to roundoff errors

Lower-upper (LU) decomposition

- Also commonly known as **lower-upper (LU) factorisation**
- We will introduce LU decomposition as an approach for solving $\mathbf{A}\vec{x} = \vec{b}$
- Actually a starting point for several different matrix tasks, as we will see
- Our plan:

  1. What is LU decomposition?
  2. What is it good for? (And what's the difficulty?)
  3. An algorithm for LU decomposition

1. What is LU decomposition?

- We will only consider *square* matrices
- A matrix $\mathbf{A}$ is said to admit an **LU decomposition** if it can be written as a product of a lower-triangular matrix ($\mathbf{L}$) and an upper-triangular matrix ($\mathbf{U}$)

$$\mathbf{A} = \mathbf{LU}$$

$$\begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix} = \begin{bmatrix} \bullet & & \\ \bullet & \bullet & \\ \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} \bullet & \bullet & \bullet \\ & \bullet & \bullet \\ & & \bullet \end{bmatrix}$$

- Consider 3×3 example:
  - $\mathbf{A}$ contains 9 elements a_{ij}
  - $\mathbf{L}$ contains 6 elements l_{ij} and $\mathbf{U}$ contains 6 elements u_{ij}
  - So the relation $\mathbf{A} = \mathbf{LU}$ implies 9 equations (one for each known element a_{ij}) involving 12 unknowns (the l_{ij}'s and u_{ij}'s)
  - This is an *underdetermined* (underconstrained) set of equations (infinitely many solutions)
  - We can choose 3 elements to get a unique solution
  - Common to set the diagonal elements of $\mathbf{L}$ to 1

$$\mathbf{L} = \begin{bmatrix} 1 & & \\ \bullet & 1 & \\ \bullet & \bullet & 1 \end{bmatrix}$$

- If $\mathbf{A} = \mathbf{LU}$ and all the diagonal elements of $\mathbf{L}$ are 1's, the matrix $\mathbf{A}$ can also be factorised in the form $\mathbf{A} = \mathbf{LDU}'$, where
  - $\mathbf{D}$ is a diagonal matrix with the diagonal elements u_{ii} of the original $\mathbf{U}$ matrix
  - $\mathbf{U}'$ is the matrix generated by taking $\mathbf{U}$ and multiplying each row i with $\frac{1}{u_{ii}}$, so that both $\mathbf{L}$ and $\mathbf{U}'$ have only 1's along their diagonals

$$\mathbf{A} = \mathbf{LDU}'$$

$$\begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix} = \begin{bmatrix} 1 & & \\ \bullet & 1 & \\ \bullet & \bullet & 1 \end{bmatrix} \begin{bmatrix} \bullet & & \\ & \bullet & \\ & & \bullet \end{bmatrix} \begin{bmatrix} 1 & \bullet & \bullet \\ & 1 & \bullet \\ & & 1 \end{bmatrix}$$

    * This is (unsurprisingly) called **LDU decomposition** or **LDU factorisation**

- Computational complexity:
  - It takes $\mathcal{O}\left(\frac{2}{3}n^3\right)$ operations to determine $\mathbf{L}$ and $\mathbf{U}$ for a given $\mathbf{A}$ $(n \times n)$
  - So the computational complexity of performing the decomposition $\mathbf{A} = \mathbf{LU}$ is the same as that of solving $\mathbf{A}\vec{x} = \vec{b}$ with Gaussian elimination
  - LU decomposition can be seen as the matrix representation of Gaussian elimination

- Existence:
  - If a square matrix $\mathbf{A}$
    * is *non-singular* (invertible), and
    * *all* its *leading principal minors* are non-zero (see note below)

    then it admits an LU (or LDU) decomposition
  - If a square matrix $\mathbf{A}$
    * is *singular* (not invertible),
    * has rank k, and
    * *the first* k of the leading principal minors are non-zero

    then it admits an LU (or LDU) decomposition
  - (For more details about this, see textbooks on linear algebra)

> *Side note:* The *leading principal minors* of $\mathbf{A}$ are the *determinants* of the square submatrices you get from $\mathbf{A}$ if you start in the upper left-hand corner and grow the submatrix by one row and column at a time

> - Example: The leading principal minors of the matrix
>
> $$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$
>
> are the determinants
>
> $$\det \begin{bmatrix} a \end{bmatrix}, \det \begin{bmatrix} a & b \\ d & e \end{bmatrix} \text{ and } \det \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

2. What is it good for?

- Assume we *have performed* the LU decomposition
- We can now
  - solve **matrix equations**, $\mathbf{A}\vec{x} = \vec{b}$, at $\mathcal{O}(n^2)$ cost
  - easily compute **the determinant**, $\det \mathbf{A}$, at $\mathcal{O}(n)$ cost
  - find **the inverse**, $\mathbf{A}^{-1}$, at $\mathcal{O}(n^3)$ cost
    * Finding $\mathbf{A}^{-1}$ would have cost $\mathcal{O}(n^4)$ if we did it by treating $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ as n equations of the form $\mathbf{A}\vec{x} = \vec{b}$ and solved each one with Gaussian elimination

Solving matrix equations after LU decomposition

- We have $\mathbf{A} = \mathbf{L}\mathbf{U}$
- Want to solve $\mathbf{A}\vec{x} = \vec{b}$ for $\vec{x}$
- We will solve $\mathbf{A}\vec{x} = \mathbf{L}\mathbf{U}\vec{x} = \vec{b}$ in two steps
  - First we define some notation: $\vec{w} \equiv \mathbf{U}\vec{x}$.
    * Since $\vec{x}$ is unknown, $\vec{w}$ is unknown.
    * We can now write $\mathbf{L}\mathbf{U}\vec{x} = \mathbf{L}\vec{w} = \vec{b}$
  1. Solve $\mathbf{L}\vec{w} = \vec{b}$ for $\vec{w}$
  2. Solve $\mathbf{U}\vec{x} = \vec{w}$ for $\vec{x}$
- **Step 1:** Solve $\mathbf{L}\vec{w} = \vec{b}$ for $\vec{w}$

- Consider example with 4×4 matrices

$$\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

- Solve by forward substitution:
  * From $l_{11}w_1 = b_1$ we immediately get the soloution for w_1:

$$w_1 = \frac{1}{l_{11}} b_1$$

  * From $l_{21}w_1 + l_{22}w_2 = b_2$, and using the now known w_1, we get

$$w_2 = \frac{1}{l_{22}} \Big[b_2 - l_{21}w_1 \Big]$$

  * Continuing the same way, we get

$$w_3 = \frac{1}{l_{33}} \Big[b_3 - l_{31}w_1 - l_{32}w_2 \Big]$$
$$w_4 = \frac{1}{l_{44}} \Big[b_4 - l_{41}w_1 - l_{42}w_2 - l_{43}w_3 \Big]$$

  * In general, when $\mathbf{A}$ is $n \times n$:

$$\boxed{w_i = \frac{1}{l_{ii}} \Big[b_i - \sum_{j=1}^{i-1} l_{ij} w_j \Big]}$$

  * Counting FLOPs (here we assume $l_{ii} = 1$):

$$\sum_{i=1}^{n} (2i - 1) = n^2$$

    (which is less than the $\mathcal{O}\left(n^3\right)$ cost of doing the LU decomposition in the first place)

- Now we know $\vec{w}$ and can so step 2

- **Step 2:** Solve $\mathbf{U}\vec{x} = \vec{w}$ for $\vec{x}$

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

- Solve for $\vec{x}$ by back substitution
  * From $u_{44}x_4 = w_4$ we immediately get the solution for x_4:

$$x_4 = \frac{1}{u_{44}} w_4$$

  * From $u_{33}x_3 + u_{34}x_4 = w_3$ we then get

$$x_3 = \frac{1}{u_{33}} \Big[w_3 - u_{34}w_4 \Big]$$

  * And so on…

$$\begin{aligned} x_2 &= \frac{1}{u_{22}} \Big[w_2 - u_{23}x_3 - u_{24}x_4 \Big] \\ x_1 &= \frac{1}{u_{11}} \Big[w_1 - u_{12}x_2 - u_{13}x_3 - u_{14}x_4 \Big] \end{aligned}$$

  * In general, when $\mathbf{A}$ is $n \times n$:

$$\boxed{\begin{aligned} x_n &= \frac{1}{u_{nn}} w_n \\ x_i &= \frac{1}{u_{ii}} \Big[w_i - \sum_{j=i+1}^{n} u_{ij}x_j \Big], \quad i = (n-1), (n-2), \ldots, 1 \end{aligned}}$$

  * This also takes $\mathcal{O}\left(n^2\right)$ FLOPs, so the combined task of forward + back substitution to find $\vec{x}$ has an $\mathcal{O}\left(n^2\right)$ cost.

• In summary:
  - If we already have $\mathbf{A} = \mathbf{LU}$, we can solve $\mathbf{A}\vec{x} = \vec{b}$ at a total $\mathcal{O}\left(n^2\right)$ cost as follows:
    1. From $\mathbf{L}\vec{w} = \vec{b}$, find $\vec{w}$ by forward substitution
    2. From $\mathbf{U}\vec{x} = \vec{w}$, find $\vec{x}$ by back substitution

A difficulty

- We need to store the full matrix $\mathbf{A}$ ($n \times n$) in memory for the LU decomposition
  - That's n^2 floating-point numbers
  - At double precision (64 bits = 8 bytes per number), this requires $n^2 \times 8$ bytes of memory
  - Example:
    * Assume $n = 10^4$
    * We then need 8×10^8 bytes $\approx 10^9$ bytes $= 1$ GB of memory
    * So we can quite quickly run out of memory
  - Also, since the decomposition is an $\mathcal{O}(n^3)$ operation, it will be slow when n is large

Finding the determinant after LU decomposition

- Once we have the decomposition $\mathbf{A} = \mathbf{LU}$, computing the determinant of $\mathbf{A}$ is trivial:

$$
\begin{aligned}
\det(\mathbf{A}) &= \det(\mathbf{LU}) \\
&= \det(\mathbf{L})\det(\mathbf{U}) \\
&= (1)(u_{11}u_{22}\dots u_{nn})
\end{aligned}
$$

  where we have assumed that $\mathbf{L}$ is on the standard form with 1's on the diagonal
- So in summary:

$$
\boxed{\det(\mathbf{A}) = \prod_{i=1}^{n} u_{ii}}
$$

  or equivalently

$$
\boxed{\log\Big(\det(\mathbf{A})\Big) = \sum_{i=1}^{n} \log u_{ii}}
$$

- For large matrices it is often useful to work numerically with the logarithm of the determinant, rather than the determinant itself

Finding the inverse after LU decomposition

3. An algorithm for LU decomposition

TODO

Topics in project 2

TODO

Scaling equations

TODO

Physics of project 2: the buckling beam

TODO

Eigenvalue problems

TODO

Jacobi rotation method

TODO

In-lecture code discussion #3

TODO

- Topic: Debugging

Iterative methods for solving matrix equations

TODO

Direct vs iterative methods

Checking convergence for an iterative method

Iterative methods

Jacobi method (not to be confused with the Jacobi rotation method)

Gauss-Seidel

Over-relaxation

In-lecture code discussion #4

TODO

- Topic: Classes in C++

How to write a scientific report

TODO

Grading system for reports

TODO

Topics in project 3

TODO

Physics of project 3: the Penning trap

TODO

Code design for simulations

TODO

Initial value problems

TODO

Forward Euler

Predictor-Corrector

Runge-Kutta

Leapfrog

Verlet

Algorithm classification

- Consistency, order of global error, one-step vs multi-step, stability

Stability

- Includes a small in-lecture code discussion: code example: `IVP_comparison`

In-lecture code discussion #4

TODO

- Topic: static variables

Intro to probability

TODO

Properties of probabilities and probability density functions

TODO

My notation

Properties

Some important one-dimensional probability distributions

Probability density functions of many variables

Expectation values

TODO

Moments

Summarising probability distributions with a single number

Introduction to Monte Carlo methods

TODO

Physics of Project 4: the Ising model

TODO

Markov chains

TODO

Markov chain Monte Carlo (MCMC)

TODO