Lecture notes FYS3150 – Computational Physics, Fall 2024

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:
  - Ingvild Bergsbak
  - Felix Forseth
  - Isabel Sagen
  - David Richard Shope
  - Alv Johan Skarpeid
  - Nils Enric Canut Taugbøl
- **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
  - Our discussion forum
  - 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.
- 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
- Main curriculum:
  - What we discuss in lectures (my lecture notes) and what we work on in the projects
- 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
- Course philosophy:
  - Pragmatic, learning by doing (and learning by failing)
  - Will try to focus on concrete examples
  - Computational physics is a *huge* and active field — this course is just a first introduction
- Lectures:

  - Thursdays and Fridays, 10.15 – 12.00
  - I try to lecture in ~30 minute sessions, with two short breaks

- Group sessions:
  - Schedule
    * Tuesday, 14.15–16.00 (room Ø397)
    * Thursday, 12.15–14.00 (room Ø434)
    * Friday, 08.15–10.00 (room Ø434)
    * Friday, 12.15–14.00 (room Ø434)
  - Probably the most important arena for learning in this course!
  - 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.

- The projects are closely connected with the learning outcomes listed on the course page

- *Tentative* deadlines:
  - Project 1: September 11
  - Project 2: September 25
  - Project 3: October 23
  - Project 4: November 20
  - Project 5: December 11

- 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 **2–3** 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
    * …

- In order to learn what you should in this course we expect you to:
  - attend the lectures
  - attend at least one group session per week
  - read/work through the content on the course web page
  - put significant effort into the projects, both coding and writing

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

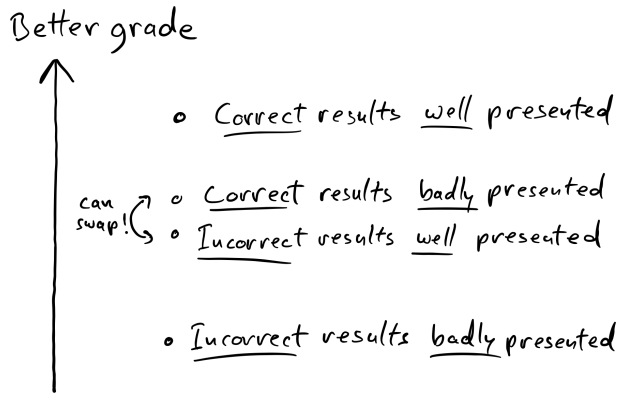

Figure 1: Presentation quality matters

- 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

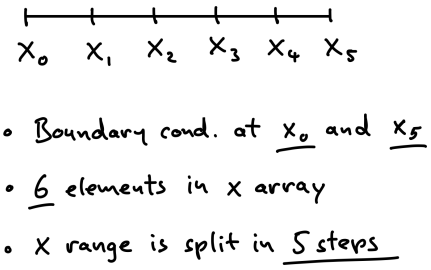

Figure 2: Sketch discretisations

- 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 at UiO
- 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 AI tools

- AI-based *large language models* (LLMs) like UiO-GPT and ChatGPT 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 project 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 have included a dedicated *Tools* subsection in the *Methods* section, 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 some AI tool.
- Some advice:
  - 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 many of 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
- 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!
- Have a look at the KURT pages on AI: www.mn.uio.no/kurt/universitet/kunstig-intelligens/

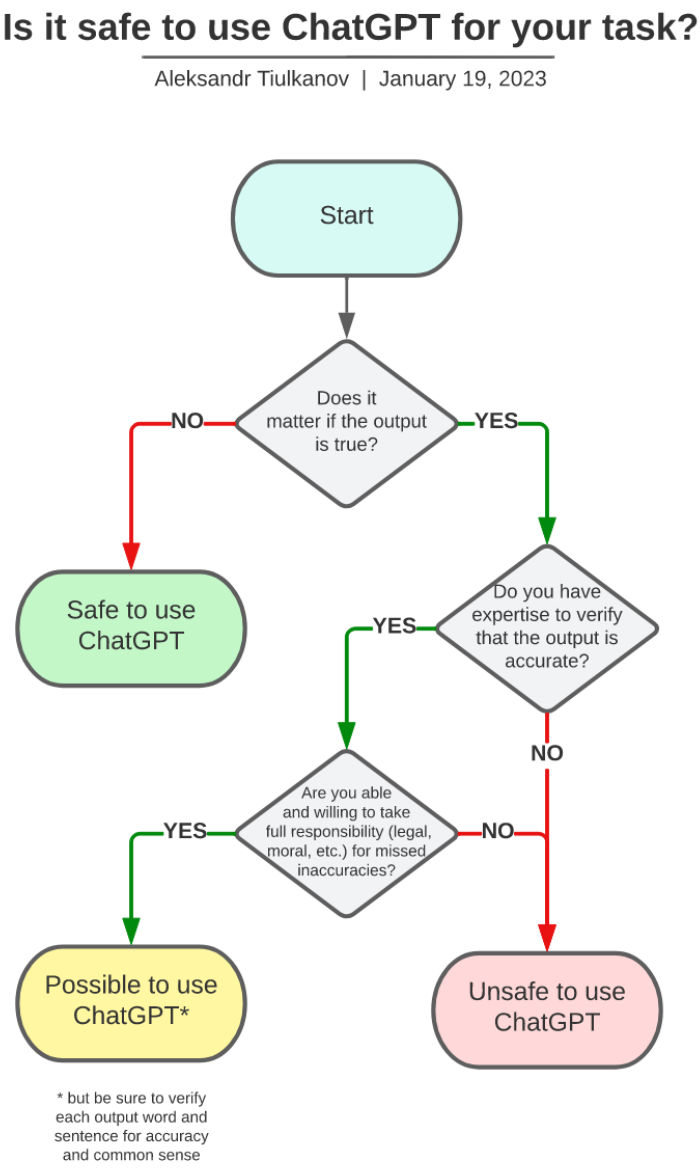

Figure 3: Example considerations to make before using ChatGPT or similar tools. Flowchart by A. Tiulkanov, included in the UNESCO report *ChatGPT and Artificial Intelligence in higher education*.

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 introductary 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):

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

- 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 $x \in [x_{\min}, x_{\max}]$. ($u(x)$ might e.g. be the solution of our diff. eq. in project 1.)
- u and x are *continuous* quantities

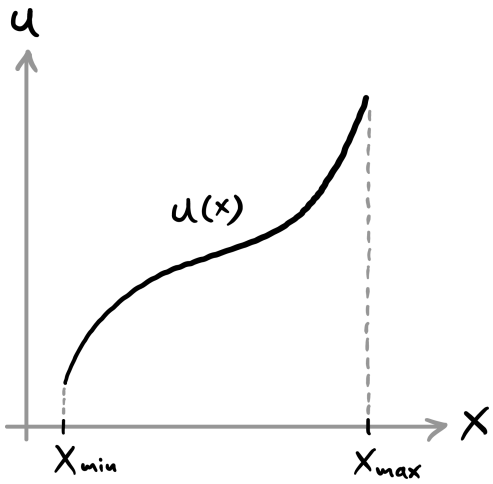

Figure 4: Continuous function

- Discretised representation

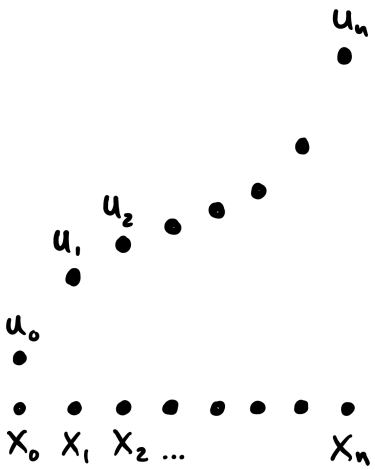

Figure 5: Discretised representation

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:

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

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

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

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 $u'(x)$:

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

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

Discretise:

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

(Two-point, forward difference)

- Compare to definition of the first derivative:

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

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

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

Discretise:

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

(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 $u(x-h)$:

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

  - 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)

- Accuracy-efficiency trade-off:
  - The three-point expression for u' is more accurate than the two-point expressions
  - Price to pay: Need to evaluate (and keep track of) the function at more points
  - Could go further: A *five-point* expression for u' would have $\mathcal{O}\left(h^4\right)$ error, etc.
  - Trade-off between accuracy and number of function evaluations

- 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 alternative 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;`
- Function arguments: a first example of **pass-by-reference** versus **pass-by-value** github.com/anderkve/FYS3150/tree/master/code_examples/function_arguments/example_1

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 $f(x) \neq 0$

- 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)

Quick description of 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 (in some clever way) 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 $u(x_{\max})$ (the second boundary condition)
  - Use this condition to determine a value for the free parameter 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

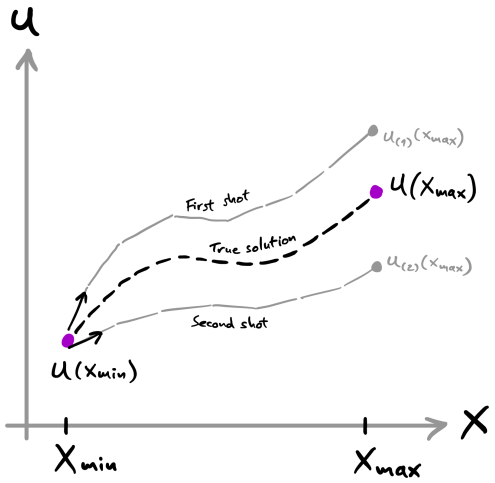

Figure 6: Sketch of the shooting method

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…

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

- 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:

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

- 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. *Later in the course*: Methods for solving a *general* matrix equation $\mathbf{A}\vec{x} = \vec{b}$
     - Gaussian elimination
     - LU decomposition
     - Iterative methods
  2. *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
  3. *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

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

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

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

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

- 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 $(x_1, \ldots, x_n)$
- 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 some 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: $R_3 \rightarrow R_3 - \frac{a_3}{\tilde{b}_2} R_2$
- 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: $R_4 \rightarrow R_4 - \frac{a_4}{\tilde{b}_3} R_3$
- 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:

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

- 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: $R_4 \rightarrow \frac{R_4}{\tilde{b}_4}$

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

  - 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: $R_3 \rightarrow \frac{R_3 - c_3 R_4}{\tilde{b}_3}$

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

- 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 $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{g}$, $\tilde{\vec{b}}$, $\tilde{\vec{g}}$ 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:

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

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

- **Example 4:** matrix-vector multiplication $\vec{y} = \mathbf{A}\vec{x}$, where A is $n \times n$:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \quad i = 1, \ldots, n$$

  - For each i we have n multiplications and $n-1$ additions, so $2n-1$ FLOPs
  - Repeat for all i: $\rightarrow n(2n-1) = 2n^2 - n$ FLOPs
  - So we say matrix-vector multiplication has $\mathcal{O}\left(n^2\right)$ computational complexity

- **Example 5:** standard matrix-matrix multiplication $\mathbf{A} = \mathbf{BC}$, with $n \times n$ matrices:

$$a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj} \quad i, j = 1, \ldots, n$$

  - For each (i, j) we have n multiplications and $n-1$ additions, so $2n-1$ FLOPs
  - Repeat for all i: $\rightarrow n(2n-1) = 2n^2 - n$ FLOPs
  - Repeat for all j: $\rightarrow n(2n^2 - n) = 2n^3 - n^2$ FLOPs
  - So we say matrix-matrix multiplication using the "schoolbook" algorithm has $\mathcal{O}\left(n^3\right)$ computational complexity
  - *But, there exist faster algorithms for matrix-matrix multiplication!*
    - Matrix multiplication is an active field of research
    - If you are interested, read about *Strassen's algorithm* and more recent advances

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 literature 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:

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

- Representation in the binary (base 2) system:

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

- 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 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:

sign bit	8-bit exponent (2)	23-bit mantissa (0.8125)
1	00000010	01101000 ... 000

- 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

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

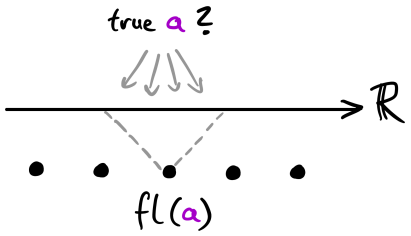

Figure 7: The continuous number line and the discretised floating-point representation

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 reasonable 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 derivative 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

$$\log_{10}\epsilon(h) = \log_{10}\left|C_1h^2 + C_2h^{-2}\right|$$

  - 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

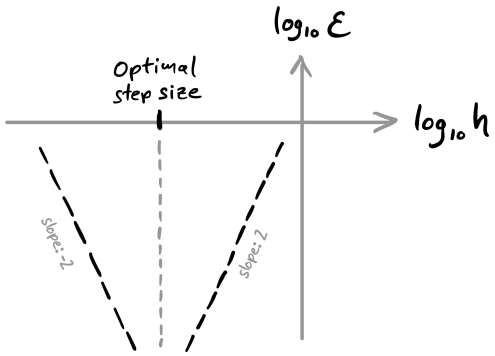

Figure 8: Sketch of a $\log_{10} \epsilon(h)$ vs $\log_{10} h$ plot, as suggested by our simple error model

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…

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

  - 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 $\mathcal{O}\left(n^3\right)$ 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

- Once we have $\mathbf{A} = \mathbf{LU}$, we can find $\mathbf{A}^{-1}$ at $\mathcal{O}\left(n^3\right)$
- We know

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix} = \mathbf{A}\mathbf{A}^{-1}$$

- Write $\mathbf{A}^{-1}$ as column vectors

$$\mathbf{A}^{-1} = \begin{bmatrix} \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} \dots \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \vec{\alpha}_1 & \vec{\alpha}_2 & \vec{\alpha}_3 & \vec{\alpha}_4 \end{bmatrix}$$

  where we have used a notation where e.g. $\vec{\alpha}_1$ contains the first column of $\mathbf{A}^{-1}$

$$\vec{\alpha}_1 \equiv \begin{bmatrix} (\mathbf{A}^{-1})_{11} \\ (\mathbf{A}^{-1})_{21} \\ (\mathbf{A}^{-1})_{31} \\ (\mathbf{A}^{-1})_{41} \end{bmatrix}$$

- Using $\mathbf{A} = \mathbf{LU}$ and our notation for $\mathbf{A}^{-1}$ we then have

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{LU}\begin{bmatrix} \vec{\alpha}_1 & \vec{\alpha}_2 & \vec{\alpha}_3 & \vec{\alpha}_4 \end{bmatrix} = \mathbf{I} = \begin{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{bmatrix}$$

- We can read this as four matrix equations, $(\mathbf{LU})\vec{\alpha}_i = \hat{e}_i$, where $\hat{e}_i$ are the unit vectors and $\vec{\alpha}_i$ are the unknown vectors we want to determine:

$$(\mathbf{LU})\vec{\alpha}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \dots \quad (\mathbf{LU})\vec{\alpha}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

- Note that each matrix equation is on the familiar form $(\mathbf{LU})\vec{x} = \vec{b}$, which we have already seen how to solve
- This was for a 4×4 example. In the general case, with $\mathbf{A}$ being $n \times n$:
  - If we have $\mathbf{A} = \mathbf{LU}$, then finding $\mathbf{A}^{-1}$ requires solving n matrix equations on the form $(\mathbf{LU})\vec{x} = \vec{b}$
  - Solving each equation has an $\mathcal{O}\left(n^2\right)$ cost
  - Thus, we can find $\mathbf{A}^{-1}$ in $\mathcal{O}\left(n^3\right)$ FLOPs
  - Finding the decomposition $\mathbf{A} = \mathbf{LU}$ in the first place also had an $\mathcal{O}\left(n^3\right)$ cost
  - So the **total cost of LU decomposition + finding $\mathbf{A}^{-1}$ is** $\mathcal{O}\left(n^3\right)$
  - (Recall that finding $\mathbf{A}^{-1}$ would have required $\mathcal{O}\left(n^4\right)$ FLOPs if we had naively solved each of the n equations $\mathbf{A}\vec{\alpha}_i = \hat{e}_i$ "from scratch" using Gaussian elimination)

> Finding the inverse of a large matrix can be the main bottleneck in many applications, e.g. in various methods in machine learning. In such cases matrix decomposition techniques like LU decomposition, or the related *Cholesky decomposition* $\mathbf{A} = \mathbf{LL}^*$, typically play a key role.

3. An algorithm for LU decomposition

- How to determine the elements of $\mathbf{L}$ and $\mathbf{U}$ such that $\mathbf{A} = \mathbf{LU}$?
- Consider a 4×4 case:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & \bullet & \bullet & \bullet \\ a_{31} & \bullet & \bullet & \bullet \\ a_{41} & \bullet & \bullet & \bullet \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

- Let's look at the first column of $\mathbf{A}$, i.e. the elements a_{i1}:
  - From a_{11} we get:

$$a_{11} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_{11} \\ 0 \\ 0 \\ 0 \end{bmatrix} = u_{11}$$

$$\boxed{u_{11} = a_{11}}$$

- From a_{21} we get:

$$a_{21} = \begin{bmatrix} l_{21} & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_{11} \\ 0 \\ 0 \\ 0 \end{bmatrix} = l_{21} u_{11}$$

$$\boxed{l_{21} = \frac{a_{21}}{u_{11}} \quad \text{(where } u_{11} \text{ is now known)}}$$

- Similarly, from the equations $a_{31} = l_{31} u_{11}$ and $a_{41} = l_{41} u_{11}$ we get

$$\boxed{l_{31} = \frac{a_{31}}{u_{11}}}$$

$$\boxed{l_{41} = \frac{a_{41}}{u_{11}}}$$

- Now for the second column $\mathbf{A}$ (elements a_{i2}):
  - From a_{12}:

$$a_{12} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_{12} \\ u_{22} \\ 0 \\ 0 \end{bmatrix} = u_{12}$$

$$\boxed{u_{12} = a_{12}}$$

  - From a_{22}:

$$a_{22} = \begin{bmatrix} l_{21} & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_{12} \\ u_{22} \\ 0 \\ 0 \end{bmatrix} = l_{21} u_{12} + u_{22} \quad \text{(where } u_{22} \text{ is the unknown)}$$

$$\boxed{u_{22} = a_{22} - l_{21}u_{12}}$$

- From a_{32}:

$$a_{32} = \begin{bmatrix} l_{31} & l_{32} & 1 & 0 \end{bmatrix} \begin{bmatrix} u_{12} \\ u_{22} \\ 0 \\ 0 \end{bmatrix} = l_{31}u_{12} + l_{32}u_{22} \quad \text{(where } l_{32} \text{ is the unknown)}$$

$$\boxed{l_{32} = \frac{a_{32} - l_{31}u_{12}}{u_{22}}}$$

- Similarly, from $a_{42} = l_{42}u_{22} + l_{41}u_{12}$:

$$\boxed{l_{42} = \frac{a_{42} - l_{41}u_{12}}{u_{22}}}$$

- We can continue like this for the third and fourth columns
- The general pattern that appears gives us the following recipe:

$$\boxed{\begin{aligned} l_{ij} &= \frac{1}{u_{jj}}\left[a_{ij} - \sum_{k=1}^{j-1} l_{ik}\, u_{kj}\right] \quad \text{with } i > j \\ u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik}\, u_{kj} \quad \text{with } i \leq j \end{aligned}}$$

Pivoting

- For numerical stability we need to avoid $u_{11} \approx 0$
- Use a permutation matrix to interchange rows
- Instead of $\mathbf{A} = \mathbf{LU}$ we will then have

$$\mathbf{A} = \mathbf{PLU}$$

or equivalently

$$\mathbf{P}^T\mathbf{A} = \mathbf{LU}$$

- Here $\mathbf{P}$ is called the *pivot matrix*
- It satisfies the relation $\mathbf{P}^T = \mathbf{P}^{-1}$, or equivalently, $\mathbf{P}\mathbf{P}^T = \mathbf{I}$

Topics in project 2

- Scaling equations
- Physics case: “the buckling beam”
  - Will lead to a two-point boundary value problem of the form

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

- Eigenvalue problems, connection to two-point boundary value problem
- Jacobi’s rotation method for eigenvalue problems
- Coding:
  - Unit testing
  - Using the Armadillo library

Scaling equations

- Also known **nondimensionalisation** or **using natural units** (but don't confuse this with the term *natural units* as used in e.g. particle physics)
- We can only represent a limited range of numbers on a computer
- We therefore want to avoid too large/small numbers in our codes.
  - First, simple approach: choose some sensible units
  - Better approach: scale away units, i.e. work with **dimensionless** variables
  - This is also useful for debugging, since it's then easier to notice if a result is surprisingly small/large

Example 1: A simulation of the solar system

- Silly choice of units: kg, m
  - In these units we would get e.g.
    $m_{\text{sun}} \approx 1.989 \times 10^{30}$ kg
    $r_{\text{earth-sun}} \approx 1.496 \times 10^{11}$ m
- Sensible choice of units: M_{sun}, au
  - Then we would have
    $m_{\text{sun}} = 1\ \text{M}_{\text{sun}}$
    $r_{\text{earth-sun}} = 1$ au

Example 2: Exponential decay of radioactive nuclei

- Consider the differential equation

$$\frac{dN(t)}{dt} = -\lambda N(t)$$

  or, written on a standard form

$$\frac{dN(t)}{dt} + \lambda N(t) = 0$$

- Initial value: $N(t=0) = N_0$

- Units:
  - N: number of nuclei, $[N] = 1$
  - t: time, $[t] = \text{s}$
  - λ: decay constant, $[\lambda] = \text{s}^{-1}$

> For this simple example we know that the analytical solution is $N(t) = N_0 e^{-\lambda t}$

- Define a scaled, dimensionless time variable (the **independent** variable):

$$\hat{t} \equiv \lambda t$$

- We can then rewrite the time derivative as

$$\frac{d}{dt} = \frac{d\hat{t}}{dt}\frac{d}{d\hat{t}} = \lambda \frac{d}{d\hat{t}}$$

- We insert this in our original differential equation (and remember that λ is just a constant)

$$\lambda \frac{dN}{d\hat{t}} + \lambda N = 0$$

$$\frac{dN}{d\hat{t}} + N = 0$$

> - Note that once we have changed variable from t to $\hat{t}$, we should think of N as new function, compared to the original $N(t)$.
>   - Technically we should have indicated this with some new notation, e.g. $N_{\hat{t}}(\hat{t})$
>   - This new function $N_{\hat{t}}$ is defined by the requirement $N_{\hat{t}}(\hat{t}(t)) = N(t)$
>   - So our differential equation above should more properly be written out as something like this:
>
> $$\frac{dN_{\hat{t}}(\hat{t})}{d\hat{t}} + N_{\hat{t}}(\hat{t}) = 0$$
>
>   - However, this is often not written out so explicitly
>   - It is common to just let the argument in $N(\hat{t})$ tell us that we should read this as $N_{\hat{t}}(\hat{t})$
>   - This is similar to how we often use the simple notation $p(x)$ and $p(y)$ for two different

> probability distributions, when we technically should have written something like $p_x(x)$ and $p_y(y)$, or $f(x)$ and $g(y)$

Since $\hat{t}$ is dimensionless and defined using the typical time scale $(1/\lambda)$ of our problem, we know

- that a step size $h \ll 1$ along the $\hat{t}$ axis will actually correspond to a *small* step size for this problem
- that solving our differential equation for a time span $\hat{t} \in [0, \text{a few}]$ will be a reasonable starting point for studying the problem, rather than starting with a time span of say $\hat{t} \in [0, 0.1]$ or $\hat{t} \in [0, 100]$

- In the differential equation above, both N and $\hat{t}$ are dimensionless numbers
- But the typical value for N may still impractical, e.g. some very large number
- So we may also consider scaling the **dependent** variable (N)
- Sensible choice here: use the initial value N_0 to set a scale for N

$$\hat{N} \equiv \frac{N}{N_0}$$

- The initial value in terms of our new variable $\hat{N}$ (let's call the initial value $\hat{N}_0$) is then simply

$$\hat{N}_0 = 1$$

> In the not-so-sloppy notation, the variable change would be
>
> $$\hat{N}_{\hat{t}} \equiv \frac{N_{\hat{t}}}{N_0}$$
>
> and the initial value relation would be
>
> $$\hat{N}_0 \equiv \frac{N_{\hat{t}}(\hat{t}(t=0))}{N_0} = \frac{N(t=0)}{N_0} = \frac{N_0}{N_0} = 1$$

- If we insert $N = N_0\hat{N}$ in our differential equation, we get

$$N_0 \frac{d\hat{N}}{d\hat{t}} + N_0\hat{N} = 0$$

or simply

$$\boxed{\frac{d\hat{N}}{d\hat{t}} + \hat{N} = 0}$$

In this final form of the differential equation, all quantities are dimensionless, and both the *independent* variable $\hat{t}$ and the *dependent* variable $\hat{N}$ have typical values of $\mathcal{O}(1)$.

- The analytical solution for our scaled differential equation is simply

$$\hat{N}(\hat{t}) = \hat{N}_0 e^{-\hat{t}} = e^{-\hat{t}} \quad \text{(recall that } \hat{N}_0 = 1\text{)}$$

- Using our definitions $\hat{N} = \frac{N}{N_0}$ and $\hat{t} = \lambda t$, we can of course get the solution in terms of our original variables N and t:

$$\frac{N}{N_0} = e^{-\lambda t}$$

$$N(t) = N_0 e^{-\lambda t}$$

> In the more careful notation, we get back to our original variables N and t by using
>
> $$\hat{t} = \lambda t$$
> $$\hat{N}_{\hat{t}}(\hat{t}(t)) = \frac{N_{\hat{t}}(\hat{t}(t))}{N_0} = \frac{N(t)}{N_0}$$
>
> which again leads to the solution
>
> $$N(t) = N_0 e^{-\lambda t}$$
>
> in terms of the original variables

Another reason why it is useful to do our numerical work in terms of scaled equations, is that it makes it easier to notice potential code bugs simply by looking out for surprisingly large (or small) outputs. If you know that all the quantities in your differential equation are $\mathcal{O}(1)$ numbers, but your code suddenly outputs some huge value, like $\mathcal{O}(10^{10})$, you immediately recognise that there may be a bug somewhere.

Presenting results

- When presenting our results, we should either *use the original, dimensionful variables*, or *specify the natural units used*

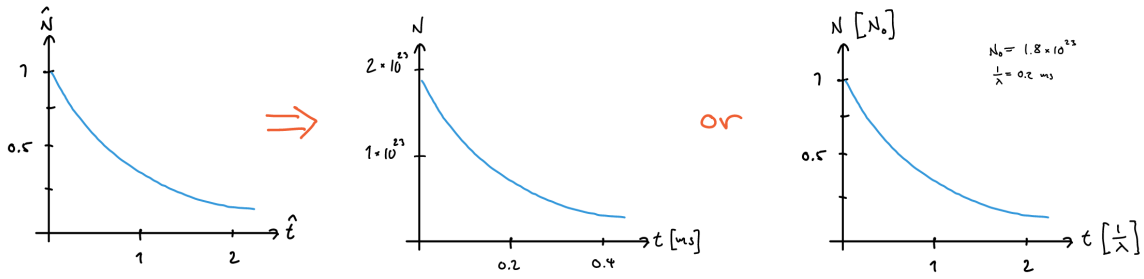

Figure 9: Our raw numerical solution will be like the left-hand plot, but the result we present should be like the middle or right-hand plot

Physics case for project 2: the buckling beam

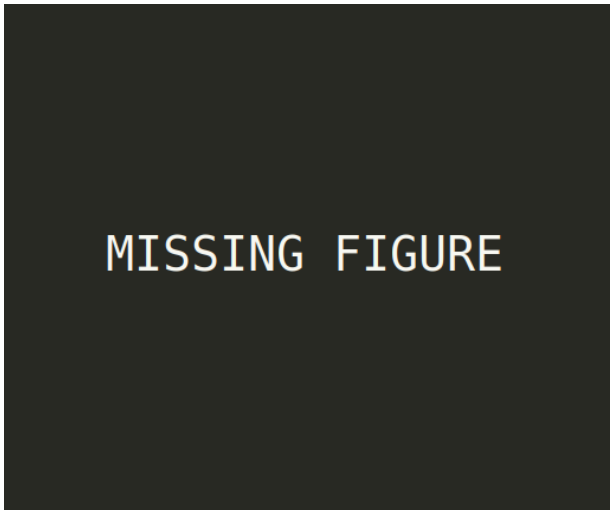

Figure 10: The buckling beam

- Description of the setup:
  - A beam of some material is extended between two points $x = 0$ and $x = L$
  - The function $u(x)$ denotes the shape of the beam (displacement from the x axis)
  - A force F is applied at the endpoint $x = L$, directed *into* the beam (i.e. along the x axis)
  - The material properties of the beam are collected in a constant γ
  - The beam can not be displaced from the x axis at the end points:
    - Boundary conditions: $u(0) = 0$ and $u(L) = 0$
  - We'll consider a so-called *pin endpoints* case:
    - The beam shape $u(x)$ can have non-zero derivative $u'(x)$ at $x = 0$ and $x = L$
- If the force F is large enough, the configuration is *unstable*
  - Any tiny perturbation will cause the beam to *buckle* (bend) into some shape
- **Our question**: What beam shapes (i.e. what function forms $u(x)$) can arise?
- We are considering this as a *static* problem – there is no time dependence here
  - We are looking for what static beam shapes are theoretically allowed under the conditions described above

- Differential equation:

$$\boxed{\gamma \frac{d^2u}{dx^2} = -Fu(x)}$$

- Do the following steps:
  - Scale the equation, to use dimensionless position $\hat{x} \equiv \frac{x}{L}$
  - Define new notation: $\lambda_c \equiv \frac{FL^2}{\gamma}$
  - Discretise with n *steps* (so we get $n+1$ points *including endpoints*, which means we have $n-1$ *interior* points)
- As you will see when you work on the project, after the above steps we end up with the following **eigenvalue problem**:

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

- $\mathbf{A}$ has size $N \times N = (n-1) \times (n-1)$
- $\lambda = \lambda(h)$, which should go to λ_c in the limit $h \to 0$
- The elements v_i of $\vec{v}$ are approximations to the exact $u_i = u(\hat{x}_i)$
- $\vec{v}$ contains the *interior* points:

$$\vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_N \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{n-1} \end{bmatrix}$$

- Our complete, approximate solution to the BVP: $\vec{v}^* = [v_0, v_1, v_2, \dots, v_{n-1}, v_n]$
- The solutions we find for $\mathbf{A}\vec{v} = \lambda\vec{v}$ are eigenvector-eigenvalue pairs, $(\vec{v}^{(i)}, \lambda^{(i)})$
- In the continuous limit ($h \to 0$, i.e. $n \to \infty$) these correspond to eigenvalues $\lambda_c^{(i)}$ and *eigenfunctions* $u^{(i)}(\hat{x})$
- Like in project 1 we have that $\mathbf{A}$ is a tridiagonal matrix (due to the second derivative), but this time we keep the factor $1/h^2$ from the second derivative as part of $\mathbf{A}$

- 4×4 example:

$$\mathbf{A} = \begin{bmatrix} \frac{2}{h^2} & -\frac{1}{h^2} & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2} & -\frac{1}{h^2} & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2} & -\frac{1}{h^2} \\ 0 & 0 & -\frac{1}{h^2} & \frac{2}{h^2} \end{bmatrix} = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

- Keeping $\frac{1}{h^2}$ on the left-hand side makes it easier to see how our approach would work also for a more general case, where we had a more complicated differential equation:

$$\Big[\text{some general operator}\Big] u(x) = \lambda u(x)$$

$$\left[\frac{d^2u}{dx^2} + \dots\right] u(x) = \lambda u(x)$$

Eigenvalue problems

- See Chapters 7.1 – 7.4 in Morten's notes.
- Actually **eigenvalue and eigenvector** problems, or **eigensystems**
- Find pairs of $(\lambda, \vec{x})$ that satisfy

$$\mathbf{A}\vec{x} = \lambda\vec{x}$$

- Sometimes only interested in one of them
  - Example from quantum mechanics: might be interested in only the allowed energies (eigenvalues), not the wavefunctions (eigenvectors)
- First consider the standard analytical approach:

$$\begin{aligned} \mathbf{A}\vec{x} &= \lambda\vec{x} \\ (\mathbf{A} - \lambda\mathbf{I})\vec{x} &= 0 \end{aligned}$$

- A matrix eq. $\mathbf{M}\vec{x} = 0$ has a non-trivial solution iff $\det(\mathbf{M}) = 0$
- So look at $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$
- $\det(\mathbf{A} - \lambda\mathbf{I})$ is a polynomial of degree N in λ

  - The *characteristic polynomial*:

$$P(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I}) = (\lambda_1 - \lambda)(\lambda_2 - \lambda)\dots$$

  - The eigenvalues are the N roots of $P(\lambda)$
  - The set of roots (eigenvalues) of $\mathbf{A}$ is called *the spectrum*, denoted as

$$\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \dots, \lambda_N\}$$

    * The determinant of $\mathbf{A}$ is the product of the eigenvalues:

$$\det(\mathbf{A}) = \lambda_1\lambda_2\dots\lambda_N$$

- This is all nice analytically, but inefficient to compute $\det(\mathbf{A} - \lambda\mathbf{I})$ when N is large
- We will instead focus on **Jacobi's rotation method**
  - Classic algorithm
  - Reasonably intuitive
  - Suitable for symmetric, real matrices
  - Gives us an example of an iterative method
  - Not very efficient, especially for large matrices
  - But well-suited for parallelisation
  - Gives us a first example of an iterative algorithm
- Many other methods exist, with different strengths and weaknesses. A few examples:
  - *QR algorithm* (general purpose)
  - *Power iteration* (largest eigenvalue)
  - *Ivnerse iteration* (smallest eigenvalue)
  - *Shift method* (eigenvalue closest to a given value)
- A basis for Jacobi's rotation method (and other methods) is **similarity transformations**

Similarity transformations

- Transformations with an orthogonal matrix $\mathbf{S}$:

$$\mathbf{S}^T = \mathbf{S}^{-1}$$
$$\mathbf{S}^T\mathbf{S} = \mathbf{S}\mathbf{S}^T = \mathbf{I}$$

- We will assume $\mathbf{A}$ is **real** and **symmetric**
- A is $N \times N$, with N eigenvalues: $\lambda^{(1)}, \lambda^{(2)}, \ldots, \lambda^{(N)}$
  - The eigenvalues are not necessarily distinct
- From **the spectral theorem**: Then there exists an orthogonal matrix $\mathbf{S}$ such that

$$\mathbf{S}^T \mathbf{A} \mathbf{S} = \mathbf{D} \quad \text{(diagonal matrix)}$$

- So $\mathbf{S}$ diagonalises $\mathbf{A}$, but how is this connected to our eigenvalue-eigenvector task?
  - Apply $\mathbf{S}$ from the left

$$\mathbf{AS} = \mathbf{SD}$$

  - Express $\mathbf{S}$ as column vectors:

$$\mathbf{S} = \begin{bmatrix} \vec{s}_1 & \vec{s}_2 & \ldots & \vec{s}_N \end{bmatrix}$$

  - Then $\mathbf{AS} = \mathbf{SD}$ becomes

$$\begin{aligned} \mathbf{A} \begin{bmatrix} \vec{s}_1 & \vec{s}_2 & \ldots & \vec{s}_N \end{bmatrix} &= \begin{bmatrix} \vec{s}_1 & \vec{s}_2 & \ldots & \vec{s}_N \end{bmatrix} \mathbf{D} \\ &= \begin{bmatrix} d_{11}\vec{s}_1 & d_{22}\vec{s}_2 & \ldots & d_{NN}\vec{s}_N \end{bmatrix} \end{aligned}$$

  - This is a set of N equations of a familiar form:

$$\begin{aligned} \mathbf{A}\vec{s}_1 &= d_{11}\vec{s}_1 \\ \mathbf{A}\vec{s}_2 &= d_{22}\vec{s}_2 \\ &\ldots \\ \mathbf{A}\vec{s}_N &= d_{NN}\vec{s}_N \end{aligned}$$

  - We see that:
    * The elements of $\mathbf{D}$ are the **eigenvalues** of $\mathbf{A}$
    * The column vectors in $\mathbf{S}$ are the **eigenvectors** of $\mathbf{A}$
- Idea behind Jacobi's rotation method:

- Find $\mathbf{S}$ by iteratively applying similarity transformations (rotations) $\mathbf{S}_1, \mathbf{S}_2, \ldots$, to $\mathbf{A}$

$$\mathbf{A} \to \mathbf{S}_1^T \mathbf{A} \mathbf{S}_1 \to \mathbf{S}_2^T \mathbf{S}_1^T \mathbf{A} \mathbf{S}_1 \mathbf{S}_2 \to \ldots$$

  such that each new transformation makes the resulting matrix a little more diagonal-like

- Once our transformed matrix is sufficiently close to being diagonal, the product of all our applied transformations is our approximation to $\mathbf{S}$

$$\mathbf{S} \approx \mathbf{S}_1 \mathbf{S}_2 \ldots$$

> *Side note*: Similarity transformations preserve eigenvalues. If we apply some similarity tranformation $\mathbf{Q}$ to the equation $\mathbf{A}\vec{x} = \lambda\vec{x}$, we transform the equation into an equation with
>
> - a transformed matrix, $\mathbf{A} \to \mathbf{Q}^T \mathbf{A} \mathbf{Q}$
> - transformed eigenvectors, $\vec{x} \to \mathbf{Q}^T \vec{x}$
> - but the *same* eigenvalues, λ
>
> To see this, just left-multiply both sides with $\mathbf{Q}^T$ and remember that $\mathbf{Q}\mathbf{Q}^T = \mathbf{I}$:
>
> $$\begin{aligned} \mathbf{A}\vec{x} &= \lambda\vec{x} \\ \mathbf{Q}^T\mathbf{A}(\mathbf{Q}\mathbf{Q}^T)\vec{x} &= \mathbf{Q}^T\lambda\vec{x} \\ (\mathbf{Q}^T\mathbf{A}\mathbf{Q})(\mathbf{Q}^T\vec{x}) &= \lambda(\mathbf{Q}^T\vec{x}) \end{aligned}$$

Jacobi's rotation method

- We want to diagonalise $\mathbf{A}$ using a similarity transformation
- That is, we want to find $\mathbf{S}$ such that $\mathbf{S}^T\mathbf{A}\mathbf{S} = \mathbf{D}$
- Will do this by applying a series of similarity transformations (rotation matrices) $\mathbf{S}_1, \mathbf{S}_2, \ldots$

$$\begin{aligned} \mathbf{A} &\equiv \mathbf{A}^{(1)} \\ \mathbf{S}_1^T\mathbf{A}\mathbf{S}_1 &\equiv \mathbf{A}^{(2)} \\ \mathbf{S}_2^T\mathbf{S}_1^T\mathbf{A}\mathbf{S}_1\mathbf{S}_2 &\equiv \mathbf{A}^{(3)} \\ \ldots & \end{aligned}$$

until we get a matrix $\mathbf{A}^{(M)}$ that is *close enough* to being diagonal:

$$\mathbf{S}_{M-1}^T \dots \mathbf{S}_1^T \mathbf{A} \mathbf{S}_1 \dots \mathbf{S}_{M-1} \equiv \mathbf{A}^{(M)} \approx \mathbf{D}$$

- Then we will have $\mathbf{S} \approx \mathbf{S}_1 \mathbf{S}_2 \dots \mathbf{S}_{M-1}$ and we are done:
  - $\mathbf{S}$ contains the eigenvectors of $\mathbf{A}$
  - $\mathbf{D}$ contains the eigenvalues of $\mathbf{A}$
- Notation-wise, each new transformation in this procedure can be written as

$$\boxed{\mathbf{A}^{(m+1)} = \mathbf{S}_m^T \mathbf{A}^{(m)} \mathbf{S}_m}$$

- For bookkeeping, it is also useful to have a matrix $\mathbf{R}$ that encodes the combined transformation so far, i.e.

$$\boxed{\mathbf{R}^{(m+1)} = \mathbf{R}^{(m)} \mathbf{S}_m}$$

- Starting from $\mathbf{R}^{(1)} = \mathbf{I}$ this means

$$\begin{aligned} \mathbf{R}^{(2)} &= \mathbf{R}^{(1)} \mathbf{S}_1 = \mathbf{S}_1 \\ \mathbf{R}^{(3)} &= \mathbf{R}^{(2)} \mathbf{S}_2 = \mathbf{S}_1 \mathbf{S}_2 \\ &\dots \\ \mathbf{R}^{(M)} &= \mathbf{R}^{(M-1)} \mathbf{S}_{M-1} = \mathbf{S}_1 \mathbf{S}_2 \dots \mathbf{S}_{M-1} \end{aligned}$$

  where the final $\mathbf{R}^{(M)}$ is our approximation to $\mathbf{S}$
- Now it's time to see how the individual matrices $\mathbf{S}_i$ are constructed
- Jacobi's rotation method is (unsurprisingly) based on *rotation matrices*

- Let's use a 10×10 matrix to illustrate the general structure

$$\mathbf{S}_m = \begin{bmatrix} 1 & & & & & & & & & \\ & 1 & & & & & & & & \\ & & 1 & & & & & & & \\ & & & c_\theta & & & s_\theta & & & \\ & & & & 1 & & & & & \\ & & & & & 1 & & & & \\ & & & -s_\theta & & & c_\theta & & & \\ & & & & & & & 1 & & \\ & & & & & & & & 1 & \\ & & & & & & & & & 1 \end{bmatrix}$$

- Here we use the shorthand notation $c_\theta \equiv \cos\theta$ and $s_\theta \equiv \sin\theta$
- All empty elements are 0's
- This example matrix represents a clockwise rotation in the (x_4, x_7) plane, by an angle θ
  - If we flip the signs of the two s_θ entries we would have a counterclockwise rotation. (Here we use a clockwise rotation to be consistent with the notation in Morten's notes.)
- Notation:
  - k: row index of the upper s_θ entry (our example: $k = 4$)
  - l: column index of the upper s_θ entry (our example: $l = 7$)
- If we know (k, l) we also know the positions of the three other c_θ and s_θ entries
- **Note**: The matrix $\mathbf{S}_m$ is fully specified by the three numbers (k, l, θ)
- In Jacobi's rotation method the next rotation matrix $\mathbf{S}_m$ is constructed as follows:
  1. In the current matrix $\mathbf{A}^{(m)}$, find the indices (k, l) of the largest (absolute value) off-diagonal element in the upper half of the matrix
  2. Determine the θ needed to "rotate away" this off-diagonal element, i.e. find the θ such that the element $a_{kl}^{(m+1)}$ in $\mathbf{A}^{(m+1)} = \mathbf{S}_m^T \mathbf{A}^{(m)} \mathbf{S}_m$ will be zero

An example with a two-by-two matrix

- Let's see how one such rotation matrix is constructed using a 2×2 example

- Start from

$$\mathbf{A} = \mathbf{A}^{(1)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} \end{bmatrix}$$

- We also assume $\mathbf{A}$ is symmetric: $a_{21}^{(1)} = a_{12}^{(1)}$
- What we *want* our similarity transformation to achieve is

$$\mathbf{A}^{(2)} = \begin{bmatrix} a_{11}^{(2)} & a_{12}^{(2)} \\ a_{21}^{(2)} & a_{22}^{(2)} \end{bmatrix} = \begin{bmatrix} a_{11}^{(2)} & 0 \\ 0 & a_{22}^{(2)} \end{bmatrix}$$

- So we need a transformation $\mathbf{S}_1$ that gives $a_{12}^{(2)} = 0$
- This is a 2×2 example, so we have $k = 1$ and $l = 2$
- Our rotation matrix is

$$\mathbf{S}_1 = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \equiv \begin{bmatrix} c_\theta & s_\theta \\ -s_\theta & c_\theta \end{bmatrix}$$

- Recall that $\mathbf{A}^{(1)}$ is symmetric. If we write out the transformation $\mathbf{A}^{(2)} = \mathbf{S}_1^T \mathbf{A}^{(1)} \mathbf{S}_1$

$$\begin{bmatrix} a_{11}^{(2)} & a_{12}^{(2)} \\ a_{21}^{(2)} & a_{22}^{(2)} \end{bmatrix} = \begin{bmatrix} c_\theta & -s_\theta \\ s_\theta & c_\theta \end{bmatrix} \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} \\ a_{12}^{(1)} & a_{22}^{(1)} \end{bmatrix} \begin{bmatrix} c_\theta & s_\theta \\ -s_\theta & c_\theta \end{bmatrix}$$

  we get the following equations for the four elements of $\mathbf{A}^{(2)}$:

$$\begin{aligned} a_{11}^{(2)} &= a_{11}^{(1)} c_\theta^2 - 2a_{12}^{(1)} c_\theta s_\theta + a_{22}^{(1)} s_\theta^2 \\ a_{22}^{(2)} &= a_{22}^{(1)} c_\theta^2 + 2a_{12}^{(1)} c_\theta s_\theta + a_{11}^{(1)} s_\theta^2 \\ a_{12}^{(2)} &= (a_{11}^{(1)} - a_{22}^{(1)}) c_\theta s_\theta + a_{12}^{(1)} (c_\theta^2 - s_\theta^2) \\ a_{21}^{(2)} &= (a_{11}^{(1)} - a_{22}^{(1)}) c_\theta s_\theta + a_{12}^{(1)} (c_\theta^2 - s_\theta^2) \end{aligned}$$

- We see explicitly that the transformation preserves the symmetry: $a_{21}^{(2)} = a_{12}^{(2)}$
- To determine θ we now apply our requirement: $a_{12}^{(2)} = 0$. This gives us the equation

$$\left(\frac{a_{11}^{(1)} - a_{22}^{(1)}}{a_{12}^{(1)}} \right) c_\theta s_\theta + c_\theta^2 - s_\theta^2 = 0 \qquad (\star)$$

- Define some shorthand notation:

$$\tan\theta \equiv t_\theta = \frac{s_\theta}{c_\theta}$$
$$\tau \equiv \frac{a_{22}^{(1)} - a_{11}^{(1)}}{2a_{12}^{(1)}}$$

- Divide Eq. $(\star)$ by c_θ^2 and use the new notation to get

$$\boxed{t_\theta^2 + 2\tau t_\theta - 1 = 0}$$

- This second order equation for t_θ has solutions

$$t_\theta = -\tau \pm \sqrt{1+\tau^2}$$

- So the two choices for θ that correspond to the two t_θ values above will ensure $a_{12}^{(2)} = 0$
- In the Jacobi rotation method, choosing the smaller of the two angles (the smaller t_θ value) typically gives faster convergence
- With t_θ known we can compute c_θ and s_θ

$$c_\theta = \frac{1}{\sqrt{1+t_\theta^2}}$$
$$s_\theta = c_\theta t_\theta$$

- Now we can compute the new matrix elements $a_{11}^{(2)}$ and $a_{22}^{(2)}$ using the equations above
  - No need to compute $a_{12}^{(2)}$ or $a_{21}^{(2)}$ since these are 0 by our choice of θ
- We now have our new $\mathbf{A}^{(2)}$:

$$\mathbf{A}^{(2)} = \begin{bmatrix} a_{11}^{(2)} & 0 \\ 0 & a_{22}^{(2)} \end{bmatrix}$$

- So, using a similarity transformation of the original $\mathbf{A}$ we have reached a diagonal matrix
- We then know that $a_{11}^{(2)}$ and $a_{22}^{(2)}$ are the **eigenvalues** of $\mathbf{A}$
- The **eigenvectors** of $\mathbf{A}$ will be the column vectors contained in the combined transformation

matrix $\mathbf{R}$ (which in this case consists of only one rotation)

$$\mathbf{R}^{(2)} = \mathbf{R}^{(1)}\mathbf{S}_1 = \mathbf{I}\mathbf{S}_1 = \begin{bmatrix} \begin{bmatrix} c_\theta \\ -s_\theta \end{bmatrix} & \begin{bmatrix} s_\theta \\ c_\theta \end{bmatrix} \end{bmatrix}$$

- Note:
  - Because this was a 2×2 example, we got an *exactly* diagonal matrix
  - In the general $N \times N$ case we will need many transformations $\mathbf{S}_m$, and we will in general never reach an exactly diagonal matrix
  - The reason is that each transformation $\mathbf{S}_m$ also affects the other elements along rows/-columns k and l
  - So, an off-diagonal element a_{ij} that had previously been rotated to $a_{ij} = 0$ can be transformed back to $a_{ij} \neq 0$ by a later rotation
  - Still, the method is guaranteed to converge towards a diagonal matrix

A general algorithm for Jacobi's rotation method

(Until these notes are ready, see the notes from 2022 in the GitHub repo.)

TODO

In-lecture code discussion #3

TODO

- Topic: Debugging

Iterative methods for solving matrix equations

- We now return to the topic of how to solve matrix equations of the general form $\mathbf{A}\vec{x} = \vec{b}$

Direct vs iterative methods

- We have previously looked at **direct methods** for solving $\mathbf{A}\vec{x} = \vec{b}$:
  - Gaussian elimination
    * In the case of a tridiagonal matrix $\mathbf{A}$ this became the Thomas algorithm
  - LU decomposition
    * A starting point for many matrix tasks, including finding the inverse $\mathbf{A}^{-1}$
    * When we have $\mathbf{A}^{-1}$, we get $\vec{x}$ from matrix-vector multiplication: $\vec{x} = \mathbf{A}^{-1}\vec{b}$
- The direct methods in principle give **the exact answer** in a **finite number of steps**
- But due e.g. to their reliance on exact equalities/cancellations in the mathematics, these methods can be susceptible to numerical problems
- Alternative class of methods: **iterative methods**
- Iterative methods iterate closer and closer to the true solution $\vec{x}$, but will generally never get there exactly
- Need some convergence criteria for deciding when to stop the iterations
- Compared to direct methods, iterative methods are often faster and with a smaller memory footprint
  - Particularly important for very large matrices
- Added bonus: the simplest iterative methods are often very easy to implement in code
  - As we will see, when an iterative method is expressed in matrix form it can look very complicated
  - But often it boils down to some very simple set of equations for the individual components x_i of the solution vector, and it is usually these component-level equations we would implement in our code

> At this point you may be thinking: *Why didn't we use an iterative method to solve the matrix equation in project 1?* The answer is that we definitely could have done that – coding-wise it would have been very easy! But since Gaussian elimination is such an important stepping stone for many more advanced algorithms, it's an important algorithm to get some hands-on experience with, and that's why we focused on that (in the form of the Thomas algorithm) in project 1.

- In the following we will look at three examples of iterative methods:

  - **The Jacobi method** (not to be confused with with Jacobi's rotation method for eigenvalue problems $\mathbf{A}\vec{x} = \lambda\vec{x}$)
  - **Gauss-Seidel**
  - **Successive over-relaxation**

- But first we need to discuss how to check for convergence when using an iterative approach

Checking convergence for an iterative method

- An iterative method has the conceptual form of a `while` loop in a program: we keep doing some steps over and over again until we have reached some stopping criterion
- Below are two approaches to how that stopping criterion can be formulated

Alternative 1: monitor the relative change in our estimate for $\vec{x}$

- We let $\vec{x}^{(m)}$ denote our estimate for the true $\vec{x}$ after m steps of our iterative method
- Let ϵ denote the *relative change in the vector norm* from one iteration step (m) to the next ($m+1$):

$$\epsilon = \left| \frac{|\vec{x}^{(m+1)}|_l - |\vec{x}^{(m)}|_l}{|\vec{x}^{(m)}|_l} \right|$$

- Here the subscript l indicates that we are using some particular l*-norm* to measure the length of a vector:

$$|\vec{x}|_l \equiv \left[\sum_{i=1}^{N} |x_i|^l \right]^{\frac{1}{l}}$$

  where N is the number of elements x_i in the vector $\vec{x}$

- Some common choices for l:

$$\begin{aligned}
l = 1: &\quad |\vec{x}|_1 = |x_1| + |x_2| + \ldots + |x_N| \quad \text{(sum of absolute values of the vector elements)} \\
l = 2: &\quad |\vec{x}|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_N^2} \quad \text{(the familiar Euclidian vector length)} \\
l = \infty: &\quad |\vec{x}|_\infty = \max_i |x_i| \quad \text{(the largest vector element in absolute value)}
\end{aligned}$$

> To see why $|\vec{x}|_\infty$ ends up picking out the largest vector element, just check what happens in the general expression for $|\vec{x}|_l$ when you set l to some very large value, say $l = 1000$. In that case, the

> sum $|x_1|^{1000} + |x_2|^{1000} + \ldots + |x_N|^{1000}$ will be highly dominated by the term containing the largest vector element. For example, consider the vector $\vec{x} = [0.9, 1.2, 1.19, 0.7]$. Then we would have that $0.9^{1000} + 1.2^{1000} + 1.19^{1000} + 0.7^{1000} \approx 1.2^{1000}$. So the 1000-norm of this vector $\vec{x}$ would be $|\vec{x}|_{1000} = [0.9^{1000} + 1.2^{1000} + 1.19^{1000} + 0.7^{1000}]^{\frac{1}{l}} \approx [1.2^{1000}]^{\frac{1}{l}} = 1.2 = \max_i |x_i|$. As we increase l towards $l = \infty$ the approximation in the previous expression becomes more and more precise.

- When we have chosen a particular l-norm, we can decide on some small threshold value such that when ϵ falls below this value, we stop the iterations
- In other words, we stop when $\vec{x}^{(m+1)}$ is sufficiently similar to our previous $\vec{x}^{(m)}$, so that we don't think we will gain much from running more iterations
- *Pro*: Computing the l-norm of the new vector $\vec{x}^{(m+1)}$ is a quick operation
- *Con*: We are here just checking whether or not our estimate for $\vec{x}$ has (almost) stopped changing – we are *not* measuring how close our estimate $\vec{x}^{(m)}$ is to actually solving the equation $\mathbf{A}\vec{x} = \vec{b}$

Alternative 2: monitor the residual

- Recall that a true solution $\vec{x}$ should satisfy $\mathbf{A}\vec{x} = \vec{b}$
- Let **the residual** $\vec{r}$ be the vector difference between $\vec{b}$ and $\mathbf{A}\vec{x}^{(m)}$ (our current estimate for $\mathbf{A}\vec{x}$)

$$\vec{r} = \mathbf{A}\vec{x}^{(m)} - \vec{b}$$

- At every iteration we can then check the ratio of the lengths of $\vec{r}$ and $\vec{b}$:

$$\frac{|\vec{r}|_l}{|\vec{b}|_l}$$

- We stop the iterations when this ratio falls below a threshold value that we have decided
- *Pro*: Compared to our previous approach to checking for convergence, this approach more directly monitors how far our estimate $\vec{x}^{(m)}$ is from satisfying the equation $\mathbf{A}\vec{x}^{(m)} = \vec{b}$
- *Con*: This approach is computationally more expensive than the previous approach, since it requires a matrix-vector multiplication.

The Jacobi method

> Despite the similar name, do not confuse this iterative method for solving $\mathbf{A}\vec{x} = \vec{b}$ with *Jacobi's rotation method*, which is a method for solving an eigenvalue-eigenvector problem $\mathbf{A}\vec{x} = \lambda\vec{x}$.

- The starting point is to rewrite our matrix $\mathbf{A}$ as the sum $\mathbf{L} + \mathbf{D} + \mathbf{U}$, where $\mathbf{L}$ is a strictly lower-triangular matrix, $\mathbf{D}$ is a diagonal matrix and $\mathbf{U}$ is a strictly upper-triangular matrix
- Here is a 3×3 example:

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} & & \\ a_{21} & & \\ a_{31} & a_{32} & \end{bmatrix} \begin{bmatrix} a_{11} & & \\ & a_{22} & \\ & & a_{33} \end{bmatrix} \begin{bmatrix} & a_{12} & a_{13} \\ & & a_{23} \\ & & \end{bmatrix}$$

  - Note that this simple decomposition is *not* the same as *LU decomposition*, which was the much more complicated task of writing $\mathbf{A}$ as a *product* $\mathbf{A} = \mathbf{LU}$ (or alternatively as $\mathbf{A} = \mathbf{LDU}$)

- We can now rewrite our matrix problem as follows

$$\begin{aligned} \mathbf{A}\vec{x} &= \vec{b} \\ (\mathbf{L} + \mathbf{D} + \mathbf{U})\vec{x} &= \vec{b} \\ \mathbf{D}\vec{x} &= -(\mathbf{L} + \mathbf{U})\vec{x} + \vec{b} \\ \vec{x} &= -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\vec{x} + \mathbf{D}^{-1}\vec{b} \end{aligned}$$

- This last expression has the form $\vec{x} = [\text{some matrix}]\vec{x} + [\text{stuff independent of } \vec{x}]$
- We will take this as inspiration and suggest the following iterative recipe for how to change some current estimate $\vec{x}^{(m)}$ to a new (and hopefully improved) estimate $\vec{x}^{(m+1)}$:

$$\boxed{\vec{x}^{(m+1)} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\vec{x}^{(m)} + \mathbf{D}^{-1}\vec{b}}$$

- Note that since $\mathbf{D}$ is diagonal, we already know the that the inverse $\mathbf{D}^{-1}$ is simply the diagonal matrix with the reciprocal matrix elements

$$\mathbf{D}^{-1} = \text{diag}\left(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \ldots, \frac{1}{a_{NN}}\right)$$

- The matrix $\mathbf{T} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$, which operates on $\vec{x}^{(m)}$ on the right-hand side of our iteration rule, is called the **iteration matrix** or the **update matrix**
- Our iterative matrix recipe above might look complicated, but it turns into some very simple update rules when expressed in terms of the individual vector components $x_i^{(m+1)}$

- To see this, let's write out the matrix multiplication above for a 4×4 example:

$$x_1^{(m+1)} = \frac{1}{a_{11}}\left[b_1 - a_{12}x_2^{(m)} - a_{13}x_3^{(m)} - a_{14}x_4^{(m)}\right]$$

$$x_2^{(m+1)} = \frac{1}{a_{22}}\left[b_2 - a_{21}x_1^{(m)} - a_{23}x_3^{(m)} - a_{24}x_4^{(m)}\right]$$

$$x_3^{(m+1)} = \frac{1}{a_{33}}\left[b_3 - a_{31}x_1^{(m)} - a_{32}x_2^{(m)} - a_{34}x_4^{(m)}\right]$$

$$x_4^{(m+1)} = \frac{1}{a_{44}}\left[b_4 - a_{41}x_1^{(m)} - a_{42}x_2^{(m)} - a_{43}x_3^{(m)}\right]$$

- So for a general case with a length-N vector, the iterative recipe on component form is simply

$$\boxed{x_i^{(m+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{\substack{j=1 \\ j\neq i}}^{N} a_{ij}x_j^{(m)}\right]}$$

- To start our iterative method, we need to make an **initial guess**, $\vec{x}^{(0)}$
- In general, the closer our initial guess $\vec{x}^{(0)}$ is to the true $\vec{x}$, the fewer iterations we need to perform
- Starting from *any* initial guess $\vec{x}^{(0)}$, the method will converge towards the true solution if the largest eigenvalue (in absolute value) of the update matrix $\mathbf{T}$ is less than 1
  - This is an example of a general result regarding convergence of iterative methods:
    * The **spectral radius** $\rho(M)$ of some matrix $\mathbf{M}$ is defined as $\rho(\mathbf{M}) = \max_i |\lambda_i|$
    * If the spectral radius of the update matrix $\mathbf{T}$ satisfies $\rho(\mathbf{T}) < 1$, the iterative method will converge starting from any initial guess
- A useful special case to remember is that the requirement $\rho(\mathbf{T}) < 1$ is always satisfied if the original matrix $\mathbf{A}$ is **diagonally dominant**
  - The matrix $\mathbf{A}$ is diagonally dominant if
    * $|a_{ii}| \geq \sum_{j\neq i} |a_{ij}|$ for *every* row i in $\mathbf{A}$, *and*
    * $|a_{ii}| > \sum_{j\neq i} |a_{ij}|$ for *at least one* of the rows in $\mathbf{A}$

Gauss-Seidel

- The Gauss-Seidel method builds on the Jacobi method, but it immediately starts using each computed estimate $x_i^{(m+1)}$ in subsequent computations during the same iteration
- Let's see this in action for an example where $\mathbf{A}$ is 4×4:

$$x_1^{(m+1)} = \frac{1}{a_{11}}\left[b_1 - a_{12}x_2^{(m)} - a_{13}x_3^{(m)} - a_{14}x_4^{(m)}\right]$$

$$x_2^{(m+1)} = \frac{1}{a_{22}}\left[b_2 - a_{21}x_1^{(m+1)} - a_{23}x_3^{(m)} - a_{24}x_4^{(m)}\right]$$

$$x_3^{(m+1)} = \frac{1}{a_{33}}\left[b_3 - a_{31}x_1^{(m+1)} - a_{32}x_2^{(m+1)} - a_{34}x_4^{(m)}\right]$$

$$x_4^{(m+1)} = \frac{1}{a_{44}}\left[b_4 - a_{41}x_1^{(m+1)} - a_{42}x_2^{(m+1)} - a_{43}x_3^{(m+1)}\right]$$

- So we are effectively doing a form of forward substitution
- For the general case where $\mathbf{A}$ is $N \times N$, the update rule on component form is

$$\boxed{x_i^{(m+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(m+1)} - \sum_{j=i+1}^{N} a_{ij}x_j^{(m)}\right]}$$

  - Note that the elements $x_i^{(m+1)}$ must be computed *sequentially*, in the order $i = 1, 2, \ldots, N$
  - In contrast, for the Jacobi method we could have computed all the $x_i^{(m+1)}$ in parallel, since these computations were independent of each other

- On matrix form, the Gauss-Seidel method corresponds to starting from the matrix equation

$$\begin{aligned}
\mathbf{A}\vec{x} &= \vec{b} \\
(\mathbf{L} + \mathbf{D} + \mathbf{U})\vec{x} &= \vec{b} \\
(\mathbf{L} + \mathbf{D})\vec{x} &= -\mathbf{U}\vec{x} + \vec{b} \\
\vec{x} &= -(\mathbf{L} + \mathbf{D})^{-1}\mathbf{U}\vec{x} + (\mathbf{L} + \mathbf{D})^{-1}\vec{b}
\end{aligned}$$

  and from this suggest the update rule

$$\boxed{\vec{x}^{(m+1)} = -(\mathbf{L} + \mathbf{D})^{-1}\mathbf{U}\vec{x}^{(m)} + (\mathbf{L} + \mathbf{D})^{-1}\vec{b}}$$

- The Gauss-Seidel method typically converges faster than the Jacobi method

Successive over-relaxation

- Successive over-relaxation (SOR) is a modified version of the Gauss-Seidel method
- SOR can achieve faster convergence than Gauss-Seidel, but it comes with a free *weight parameter* ω, and the optimal choice for ω is unknown in most cases
- SOR can be seen as the following schematic generalisation of the Gauss-Seidel method:

$$[\text{new}] = [\text{old}] + [\text{weight}] * [\text{the change [new]-[old] from Gauss-Seidel}]$$

- If we choose the weight $\omega = 1$ we simply get back the Gauss-Seidel method
- For $\omega > 2$ the method will *not* converge
- For $1 < \omega \leq 2$ the method *will converge*, but the optimal choice is problem-specific (and generally unknown)
  - Note that SOR will also converge for $0 < \omega < 1$, but such *under-relaxation* is typically not useful for improving convergence compared to e.g. Gauss-Seidel
- On component form, the update rule for SOR is

$$\boxed{x_i^{(m+1)} = x_i^{(m)} + \frac{\omega}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(m+1)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(m)} - a_{ii} x_i^{(m)} \right]}$$

  - As with the Gauss-Seidel method, the vector components $x_i^{(m+1)}$ must be computed *sequentially*, in the order $i = 1, 2, \ldots, N$
- The update rule on matrix form is given by the following (rather confusing-looking) expression:

$$\boxed{\vec{x}^{(m+1)} = [\omega \mathbf{L} + \mathbf{D}]^{-1} \left[- [\omega \mathbf{U} + (\omega - 1)\mathbf{D}] \, \vec{x}^{(m)} + \omega \vec{b} \right]}$$

  - The best choice for ω would be the choice that gives the smallest spectral radius for the update matrix $\mathbf{T} = [\omega \mathbf{L} + \mathbf{D}]^{-1} [- [\omega \mathbf{U} + (\omega - 1)\mathbf{D}]]$

In-lecture code discussion #4

TODO

- Topic: Classes in C++

How to write a scientific report

TODO

Grading system for reports

- We judge the project reports on the following aspects, which each has a given importance weight

Aspect	Weight
Title and abstract	1
Introduction	2
Theory/methods	4
Code	4
Results and discussion	5
Conclusions	1
References	1
Overall impression	2

Table 3: Aspects and weights considered when grading reports

- When grading a report we first score each of these aspects using a 0–5 points scale, and then combine these scores in a weighted sum using the weights above
- So the maximum score is 100 points
- Each report is independently scored by two teachers
  - If the two scores are similar we take the average as the final score
  - If the two scores are very different we take a second look together to discuss the report and agree on a final score

- While each report is scored by two teachers, only one of the teachers will write the written feedback

Topics in project 3

- Methods for solving *initial value problems* (ODEs)
  - (In previous projects we have focused on *boundary value problems*)
- Main algorithm: Runge-Kutta, 4th order
- Coding:
  - Object-oriented programming, classes in C++
  - More use of the Armadillo library
- Writing a proper scientific report
- Physics case: simulating a *Penning trap*

Physics of project 3: the Penning trap

TODO

Code design for simulations

TODO

Initial value problems

- Previously we have worked on solving *boundary value problems*, i.e. ODEs where we know the solution at the boundaries of our domain
  - We did this in both projects 1 and 2
  - After discretising our ODE using a finite difference scheme, we ended up with some matrix problem
- Now we will turn to *initial value problems*, i.e. ODEs where we know how the solution starts, and we need to evolve it forwards

  - This will be our focus in project 3
  - For initial value problems the independent variable is often time – but it doesn't have to be!
- Later in the course (project 5) we will look at *partial differential equations* (PDEs)
- We will discuss the following methods for solving initial value problems:
  - Forward Euler
  - Predictor-Corrector
  - Runge-Kutta
  - Leapfrog
  - Verlet, including *velocity Verlet*
- But before we look at the different methods, let's recap some basics about ODEs

Classification

- Since different methods work well for different types of differential equations, it's useful to remind ourselves of some basic ODE classification
- First order ODE:

$$\frac{dy}{dt} = f(t, y)$$

  - This equation is *first order*, since the highest-order derivative of y is $\frac{dy}{dt}$
  - In an initial value problem we would typically know $y(t_0)$
- Second order ODE:

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right)$$

  - This equation is *second order*, since the highest-order derivative of y is $\frac{d^2y}{dt^2}$
  - In an initial value problem we would typically know both $y(t_0)$ and $y'(t_0)$
  - A typical example from physics would be Newton's second law:

$$\frac{d^2x}{dt^2} = \frac{1}{m}F\left(t, x, \frac{dx}{dt}\right)$$

- Linear versus non-linear ODEs:

- Example of a *linear*, first-order ODE:

$$\frac{dy}{dt} = g^3(t)y(t)$$

  * Linear, since the highest power of y in any term is 1

- Example of a *non-linear*, first-order ODE:

$$\frac{dy}{dt} = g^3(t)y(t) - h(t)y^2(t)$$

  * Non-linear, since the dependence on y is more complicated than a simple linear dependence (note the y^2 term)

- In project 3, the Coulomb interaction between the charged particles in the Penning trap give rise to a set of non-linear differential equations
- Non-linear problems typically require a numerical approach. (Often a closed-form analytical solution does not exist.)

From a second-order equation to coupled first-order equations

- If we have to solve an m^{th}-order ODE, we can always rewrite it as a set of m first-order ODEs
- The resulting first-order equations will typically be *coupled*
- Let's take Newton's second law in one dimension as example:

$$\frac{d^2x}{dt^2} = \frac{1}{m}F\left(t, x, \frac{dx}{dt}\right)$$

- This is a second-order differential equation for $x(t)$
- We can turn this into two first-order differential equations as follows:
  - First we *define* a new variable with a rather suggestive name: $v \equiv \frac{dx}{dt}$
  - We can now write $\frac{d^2x}{dt^2}$ as $\frac{dv}{dt}$
  - We are then left with the following two first-order differential equations:

$$\frac{dx}{dt} = v(t) \quad \text{(from the definition of } v\text{)}$$

$$\frac{dv}{dt} = \frac{1}{m}F(t, x, v) \quad \text{(from the original diff. eq.)}$$

  - These equations are coupled, since the unknown $v(t)$ appears in the diff. eq. for $x(t)$, and the unknown $x(t)$ appears in the diff. eq. for $v(t)$

- In project 3 you will use this approach to obtain a set of coupled, first-order equations that your numerical programs will solve

Local versus global errors

- Before we work our way through a series of methods for solving initial value problems, we should quickly discuss the difference between the **local error** and the **global error** of a method
- Our standard expressions for numerically computing a derivative introduce some **truncation error**
- The reason is typically that we truncate a Taylor expansion at some point
- Let's say our method for solving an initial value problem involves truncating a Taylor expansion starting from the $\mathcal{O}\left(h^k\right)$ terms in the exapansion (h is step size)
- Thus, at *every time step* of our method we expect to introduce an $\mathcal{O}\left(h^k\right)$ error
  - This is referred to as the **local error**
- Now, say that we evolve our solution for a total of n time steps
- Since h is the step size, we have that $n \propto 1/h$
- Let's assume that the local errors at each step simply accumulate
- We then expect the **global error** (the final error) to be of order $\mathcal{O}\left(nh^k\right) = \mathcal{O}\left(\frac{1}{h}h^k\right) = \mathcal{O}\left(h^{k-1}\right)$
- Remember that h is a small number ($h < 1$), so the $\mathcal{O}\left(h^{k-1}\right)$ global error is larger than the $\mathcal{O}\left(h^k\right)$ local error — as expected
- We categorise different methods by referring to how their global error depends on the step size:
  - A **first-order** method has a $\mathcal{O}\left(h^1\right)$ global error
  - A **second-order** method has a $\mathcal{O}\left(h^2\right)$ global error
  - etc.
- Note that in some cases, the relationship between the local and global error will be more complicated than simply subtracting 1 from the power of the local error

Forward Euler

- Consider a first-order, ordinary differential equation:

$$\frac{dy}{dt} = f(t, y)$$

  - We seek the unknown function $y(t)$
  - We know the starting value $y(t_0)$
  - The right-hand side $f(t, y)$ is some known function of t and $y(t)$

- The **Forward Euler** algorithm for solving this is simply

$$\boxed{y_{i+1} = y_i + hf_i}$$

  - Notation: $f_i \equiv f(t_i, y_i)$
  - Local error: $\mathcal{O}\left(h^2\right)$
  - Global error: $\mathcal{O}\left(h\right)$
    - So Forward Euler is a **first-order** method
  - Forward Euler is a **single-step** method, since we only need the current point y_i to compute our next point y_{i+1}

- Forward Euler is a very simple method, and a basic building block in many other algorithms
- Alternative formulation, to more easily see the connection with later methods:

$$\boxed{\begin{aligned} k &= hf_i = hf(t_i, y_i) \\ y_{i+1} &= y_i + k \end{aligned}}$$

Derivation of Forward Euler

- First, replace $\frac{dy}{dt}$ in the differential equation with our familiar forward-difference expression:

$$\frac{y(t+h) - y(t)}{h} + \mathcal{O}\left(h\right) = f(t, y)$$

- Rearrange for $y(t+h)$:

$$y(t+h) = y(t) + hf(t,y) + \mathcal{O}\left(h^2\right)$$

- Discretise: $y(t) \to y_i$, $f(t,y) \to f_i$
- Approximate: leave out the $\mathcal{O}\left(h^2\right)$ terms
- Result:

$$y_{i+1} = y_i + hf_i$$

Forward Euler and Euler-Cromer for coupled equations

- Consider a second-order initial value problem for the unknown function $x(t)$:

$$\frac{d^2x}{dt^2} = f\left(t, x, \frac{dx}{dt}\right)$$

- As usual, we can turn this into two, coupled first-order equations:
  - Define $v \equiv \frac{dx}{dt}$
  - Then we get the two coupled equations:

$$\frac{dv}{dt} = f(t,x,v)$$

$$\frac{dx}{dt} = v(t)$$

- The Forward Euler method is then simply

$$\boxed{\begin{aligned} v_{i+1} &= v_i + hf_i \\ x_{i+1} &= x_i + hv_i \end{aligned}}$$

- A very simple improvement of this method is called **Euler-Cromer**
- It consists of simply using the new v_{i+1} in the computation of x_i

$$\boxed{\begin{aligned} v_{i+1} &= v_i + hf_i \\ x_{i+1} &= x_i + hv_{i+1} \end{aligned}}$$

- Like Forward Euler, the Euler-Cromer method is a first-order method
- But the Euler-Cromer method is also a **symplectic** method
  - For physics applications, this in practice means that solutions found with Euler-Cromer will nearly satisfy energy conservation
  - In contrast, a solution found with Forward Euler will typically exhibit **energy drift**, that is, the numerical errors will add up in such a way that the energy of the solution keeps increasing or decreasing
  - Symplectic methods are therefore particularly useful when simulating physics systems over long time spans

Predictor-Corrector

- We return to our example of a first-order initial value problem:

$$\frac{dy}{dt} = f(t, y)$$

  - We seek the unknown function $y(t)$
  - We know the starting value $y(t_0)$
  - The right-hand side $f(t, y)$ is some known function of t and $y(t)$
- The **Predictor-Corrector** method can be seen as a slightly more advanced variant of Forward Euler
- Recall Forward Euler:
  - When we "shoot" our way across the time interval (t_i, t_{i+1}), we only use the gradient (i.e. the right-hand side of our differential equation) evaluated at the start point of the interval:

$$y_{i+1} = y_i + hf_i$$

- Predictor-Corrector:
  - We can get a better estimate for the true, average gradient across the time interval (t_i, t_{i+1}) if we combine the gradients at the start point (f_i) and end point (f_{i+1})
  - That is, what we *want* is a method like this:

$$y_{i+1} = y_i + h\left(\frac{f_i + f_{i+1}}{2}\right)$$

- Complication:
  - The right-hand side $f(t, y)$ depends on $y(t)$, so to evaluate $f_{i+1} = f(t_{i+1}, y_{i+1})$ we need to know y_{i+1}
  - But y_{i+1} is exactly what we are trying to estimate in the first place…
- Solution:
  - First, use a simple Forward-Euler step to estimate (or *predict*) the unknown y_{i+1}. Notation: y^*_{i+1}
  - Then, use y^*_{i+1} to estimate f_{i+1}. Notation: f^*_{i+1}
  - Finally, use f^*_{i+1} to compute an improved estimate for y_{i+1} (or *correct* the estimate for y_{i+1})

1) Predict:

$$y^*_{i+1} = y_i + hf_i$$

$$f^*_{i+1} = f(t_{i+1}, y^*_{i+1})$$

2) Correct:

$$y_{i+1} = y_i + h\left(\frac{f_i + f^*_{i+1}}{2}\right)$$

- Local error: $\mathcal{O}\left(h^3\right)$
- Global error: $\mathcal{O}\left(h^2\right)$
  - So Predictor-Corrector is a **second-order** method
- Predictor-Corrector requires *two* evaluation of the right-hand side $f(t, y)$ for each time step
- Like Forward Euler, Predictor-Corrector is a single-step method, since it only requires keeping track of the current point y_i to get the next point y_{i+1}

- Predictor-Corrector in alternative notation, where we use two different k's to denote two different shifts in the y direction:

$$\boxed{\begin{aligned} k_1 &= hf_i = hf(t_i, y_i) \\ k_2 &= hf^*_{i+1} = hf(t_i + h, y_i + k_1) \\ y_{i+1} &= y_i + \tfrac{1}{2}(k_1 + k_2) \end{aligned}}$$

Derivation of local error for Predictor-Corrector

- Start from a Taylor expansion for y_{i+1}:

$$y_{i+1} = y_i + hy'_i + \tfrac{1}{2}h^2 y''_i + \mathcal{O}\left(h^3\right)$$

- Insert $y'_i = f_i$ and $y''_i = f'_i$:

$$y_{i+1} = y_i + hf_i + \tfrac{1}{2}h^2 f'_i + \mathcal{O}\left(h^3\right)$$

- Insert a forward-difference expression for f'_i:

$$\begin{aligned} y_{i+1} &= y_i + hf_i + \tfrac{1}{2}h^2\left[\frac{f_{i+1} - f_i}{h} + \mathcal{O}(h)\right] + \mathcal{O}\left(h^3\right) \\ &= y_i + hf_i + \tfrac{1}{2}hf_{i+1} - \tfrac{1}{2}hf_i + \mathcal{O}\left(h^3\right) \\ &= y_i + h\left(\frac{f_i + f_{i+1}}{2}\right) + \mathcal{O}\left(h^3\right) \end{aligned}$$

- We recognise the first terms in this expression as the Predictor-Corrector expression, and therefore that the local truncation error in Predictor-Corrector will be $\mathcal{O}\left(h^3\right)$
  - Note that in the Predictor-Corrector method we use an approximation f^*_{i+1} rather than f_{i+1}. But the error in this approximation is $\mathcal{O}\left(h^2\right)$, which combines with the factor h in the expression above, so the local error in Predictor-Corrector remains $\mathcal{O}\left(h^3\right)$

Runge-Kutta

- We stick with our example of a first-order initial value problem:

$$\frac{dy}{dt} = f(t, y)$$

- The **Runge-Kutta** methods can be seen as further generalisations of Predictor-Corrector
- Recall from above that Predictor-Corrector:
  - can be seen as a second-order Taylor expansion of y_{i+1}
  - achieved an $\mathcal{O}\left(h^2\right)$ global error
  - used multiple (two) estimates of the gradient, computed at different points in the (t_i, t_{i+1})
  - combined the gradient estimates in a weighted sum (a simple average) when computing the final y_{i+1}
- An *m-th order Runge-Kutta method*:
  - corresponds to an m-th order Taylor expansion of y_{i+1}
  - will have an $\mathcal{O}\left(h^m\right)$ global error
  - will use multiple estimates of the gradient, computed at different points in the (t_i, t_{i+1})
  - will combine these gradient estimates in a weighted sum when computing the final y_{i+1}
- For each choice of order m, there are different possible choices for:
  - where in the interval (t_i, t_{i+1}) the gradient should be evaluated
  - how to do the weighted combination of these gradient evaluations
- So Predictor-Corrector corresponds to a specific *second-order Runge Kutta* method, where the gradient is evaluated at the points t_i and t_{i+1} and these gradients are combined with equal weights
- How to choose the order m? Must strike a balance between a low number of evaluations of $f(t, y)$ (computation time) and a low global error (accuracy)
- A very common choice: 4-th order Runge-Kutta method (**RK4**):
  - With RK4, reducing the step size h by a factor 10 will reduce the global truncation error by a factor 10,000
  - RK4 requires four f evaluations per step, but the high accuracy means we can use much larger step sizes h than with e.g. Predictor-Corrector or Forward Euler
  - So with the right step size, RK4 is typically more efficient than Predictor-Corrector or Forward Euler
- The RK4 algorithm:

$$
\begin{aligned}
k_1 &= hf(t_i, y_i) \\
k_2 &= hf(t_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}k_1) \\
k_3 &= hf(t_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}k_2) \\
k_4 &= hf(t_i + h, y_i + k_3) \\
y_{i+1} &= y_i + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
$$

  - Local error: $\mathcal{O}\left(h^5\right)$
  - Global error: $\mathcal{O}\left(h^4\right)$
  - Note the use of four different evaluations of $f(t, y)$:
    * One at the start point t_i
    * Two different estimates at the middle point $t_i + \frac{1}{2}h$
    * One estimate at the end point $t_{i+1} = t_i + h$
  - Like Forward Euler and Predictor-Corrector, Runge-Kutta is a single-step method

RK4 and Simpson's rule for integration

- The form of the weighted combination $\frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ in RK4 can be seen as arising from Simpson's rule for numerical integration
- Simpson's rule:

$$
\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]
$$

- To see this, start from the Fundamental Theorem of Calculus expressed with our unknown function $y(t)$:

$$
\int_{t_a}^{t_b} \frac{dy}{dt}\,dt = y(t_b) - y(t_a)
$$

- Choose the intergration end points t_a and t_b to be the end points t_i and t_{i+1} for one step of the RK4 method

- Then $y(t_a) = y_i$ and $y(t_b) = y_{i+1}$
- Also, insert our differential equation $\frac{dy}{dt} = f(t, y(t))$ for the integrand
- We then get

$$\int_{t_i}^{t_{i+1}} f(t, y(t))\, dt = y_{i+1} - y_i$$

- Rearrange for y_{i+1}

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y(t))\, dt$$

- We can solve the time step integral (approximately) using Simpson's rule

$$\begin{aligned} y_{i+1} &\approx y_i + \frac{t_{i+1} - t_i}{6}\left[f_i + 4f_{i+\frac{1}{2}} + f_{i+1}\right] \\ &= y_i + \frac{1}{6}\left[hf_i + 4hf_{i+\frac{1}{2}} + hf_{i+1}\right] \\ &= y_i + \frac{1}{6}\left[hf_i + 2hf_{i+\frac{1}{2}} + 2hf_{i+\frac{1}{2}} + hf_{i+1}\right] \end{aligned}$$

  where the shorthand notation $f_{i+\frac{1}{2}}$ means $f_{i+\frac{1}{2}} = f(t + \frac{1}{2}h, y(t + \frac{1}{2}h))$
- The last line above has exactly the form of the RK4 algorithm, except that in RK4 the two occurances of $f_{i+\frac{1}{2}}$ and the f_{i+1} are replaced by estimates

RK4 for sets of coupled equations

- In the Runge-Kutta algorithm we have to compute a series of k's
- Each new such k computation relies on estimating the right-hand side of the differential equation at a new half/whole time step away from the current time t_i
- Now assume we have a system of *coupled* differential equations, e.g. two first-order equations describing a position x and a velocity v:

$$\frac{dx}{dt} = v(t)$$

$$\frac{dv}{dt} = f(t, x, v)$$

- When solving these equations with RK4, we will for each time step compute two sets of k's:
  - One set for estimating x_{i+1}: $k_{x,1}, k_{x,2}, k_{x,3}, k_{x,4}$
  - One set for estimating v_{i+1}: $k_{v,1}, k_{v,2}, k_{v,3}, k_{v,4}$
- To get all these k evaluations correct, we need to perform them "in sync":

$$
\begin{aligned}
k_{x,1} &= hv_i \\
k_{v,1} &= hf(t_i, x_i, v_i) \\
\\
k_{x,2} &= h(v_i + \tfrac{1}{2}k_{v,1}) \\
k_{v,2} &= hf(t_i + \tfrac{1}{2}h, x_i + \tfrac{1}{2}k_{x,1}, v_i + \tfrac{1}{2}k_{v,1}) \\
\\
k_{x,3} &= h(v_i + \tfrac{1}{2}k_{v,2}) \\
k_{v,3} &= hf(t_i + \tfrac{1}{2}h, x_i + \tfrac{1}{2}k_{x,2}, v_i + \tfrac{1}{2}k_{v,2}) \\
\\
k_{x,4} &= h(v_i + k_{v,3}) \\
k_{v,4} &= hf(t_i + h, x_i + k_{x,3}, v_i + k_{v,3}) \\
\\
x_{i+1} &= x_i + \tfrac{1}{6}(k_{x,1} + 2k_{x,2} + 2k_{x,3} + k_{x,4}) \\
v_{i+1} &= v_i + \tfrac{1}{6}(k_{v,1} + 2k_{v,2} + 2k_{v,3} + k_{v,4})
\end{aligned}
$$

Example: many-particle simulation

- To see how this plays out in practice, consider a case similar to that in project 3:
- We are writing code to simulate a collection of interacting particles (in the classical physics sense) in three space dimensions
- For each particle we will have a position vector $\vec{r}$ and a velocity vector $\vec{v}$
- So, for each particle, we need to solve a set of differential equations on the form

$$\frac{d\vec{r}}{dt} = \vec{v}$$

$$\frac{d\vec{v}}{dt} = \frac{\vec{F}}{m}$$

  where $\vec{F}$ is the total force acting on the particle and m is the particle mass
- Let's say the particles interact via Coulomb interactions

- Therefore, the total force $\vec{F}$ acting on a particle depends on the current positions of all the other particles in the simulation
  - This means the equations above will not only be coupled, but also non-linear, due to the Coulomb interaction terms in $\vec{F}$
- Since we are working in three space dimensions, the equations above represent six differential equations for each particle (three position components and three velocity components)
- So to evolve one particle one time step will involve computing $6 \times 4 = 24$ different k's
- To keep track of all these k's it's useful to simply work at the level of 3-vectors, i.e. use vectors also for the k's:
  - For position: $\vec{k}_{\vec{r},1}, \vec{k}_{\vec{r},2}, \vec{k}_{\vec{r},3}, \vec{k}_{\vec{r},4}$
  - For velocity: $\vec{k}_{\vec{v},1}, \vec{k}_{\vec{v},2}, \vec{k}_{\vec{v},3}, \vec{k}_{\vec{v},4}$
- To evolve this simulation one time step using the RK4 algorithm, we would do something like the following:
  - Make a temporary copy of the current state of the simulation (all the particle positions and velocities), since we'll need the original positions and velocities to perform the final RK4 update step
  - For each particle: compute $\vec{k}_{\vec{r},1}$ and $\vec{k}_{\vec{v},1}$
  - For each particle: update position and velocity using the corresponding $\vec{k}_{\vec{r},1}$ and $\vec{k}_{\vec{v},1}$
  - For each particle: compute $\vec{k}_{\vec{r},2}$ and $\vec{k}_{\vec{v},2}$
  - For each particle: update position and velocity using the corresponding $\vec{k}_{\vec{r},2}$ and $\vec{k}_{\vec{v},2}$
  - For each particle: compute $\vec{k}_{\vec{r},3}$ and $\vec{k}_{\vec{v},3}$
  - For each particle: update position and velocity using the corresponding $\vec{k}_{\vec{r},3}$ and $\vec{k}_{\vec{v},3}$
  - For each particle: compute $\vec{k}_{\vec{r},4}$ and $\vec{k}_{\vec{v},4}$
  - Final step: For each particle, perform the proper RK4 update of position and velocity using the original particle position and velocity, together with all the $\vec{k}_{\vec{r},i}$ and $\vec{k}_{\vec{v},i}$ computed above

Leapfrog

TODO

Verlet

TODO

Algorithm classification

TODO

- 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

Topics in project 4

- Physics case: the Ising model (in 2D)
- Basics of probabilities and probability densities
- Expectation values
- Monte Carlo methods
  - Sampling from low-dimensional pdfs
    * Rejection sampling
    * Inverse transform sampling
  - Sampling from high-dimensional pdfs
    * Markov chain Monte Carlo
  - Random number generation
- Coding:
  - Parallelisation (with OpenMP)
- Numerical integration
  - High-dimensional integrals
  - Low-dimensional integrals

Intro to probability: what does probability mean?

- [In-class discussion of a die throw]
  - Q: If I throw this die, what's the probability that I'll get a six?
  - [I throw the die, look at the result, but hide the result from everyone else]
  - Q: Now, what's the probability that the result is a six?
  - [Discuss if/when it makes sense for two different people to claim different probabilities]
- The big question: What does a probability statement like $P(X) = 10\%$ really *mean*?
- We don't know! Or at least, we don't agree…
- The *mathematics* of probabilities is well understood, but we don't know what a probability *is*

- The **interpretation of probability** is an open question in philosophy!
- Recommended reading: *Interpretations of Probability* in the Stanford Encyclopedia of Philosophy: plato.stanford.edu/entries/probability-interpret

> *Probability is the most important concept in modern science, especially as nobody has the slightest notion what it means.* — Bertrand Russel, 1929

- There are several different philosophical views on the interpretation of probability
- But in practical use, there are two classes of interpretations that dominate: **Frequentist probability** and **Bayesian probability**
- **Frequentist probability**

$$\boxed{P(X) \equiv \lim_{n\to\infty} \frac{n_X}{n}}$$

  - Here probability is *defined* to be related to long-run relative frequencies of events/outcomes
  - So frequentist probability is inherently connected to something **repeatable**
  - There must also be some source of variation (randomness?) between trails, otherwise all repeated trails would give the same result (in which case all frequentist probabilities would be 0 or 1)

- **Bayesian probability**

$$\boxed{P(X) \equiv \text{degree of belief/knowledge that } X \text{ is true}}$$

  - A broader definition of probability, compared to the frequentist definition
  - Here probability theory can be seen as a mathematical framework for *reasoning under uncertainty* or for *making consistent bets*
  - In the Bayesian philosophy, we can assign probabilities to any type of statement X, not only statements about something repeatable
    - (That of course means that we can *also* assign Bayesian probability to statements that concern long-run relative frequencies)
  - Two main schools of thought within the Bayesian camp:
    - *Subjective Bayesianism*: Typically refers to $P(X)$ as a degree of *belief*

* *Objective Bayesianism*: Typically refers to $P(X)$ as a degree of *knowledge/information*

> *Disclaimer*: I personally find the Bayesian approach to probability very convincing. My presentation of this topic is therefore probably (in the Bayesian sense!) coloured by my own view. So make sure to also discuss this philosophical question with your friendly neighbourhood frequentist!

- Frequentist and Bayesian probability ends up producing the same *mathematical rules* for probability theory, i.e.

$$\begin{aligned} P(X) + P(\bar{X}) &= 1 \\ P(X,Y) &= P(X|Y)P(Y) \\ P(X|Y)P(Y) &= P(Y|X)P(X) \end{aligned}$$

- But philosophically, frequentists and (different schools of) Bayesians often point to different sets of axioms (e.g. Kolmogorov, Cox, Dutch-book coherence) as the logical starting point for probability theory

Consequences

- Below we will mention three topics where the interpretation of probability has important consequences, both practically and philosophically

1. Different approaches to statistics

- Different interpretations of probability give rise to different approaches to statistics
- That's the reason why you will often at universities find separate courses for frequentist (or *classical*) statistics and Bayesian statistics
- Within Bayesian statistics we can ask questions on the form $P(\text{hypothesis} \,|\, \text{data})$, since this just means *What is my/our degree of belief/knowledge that the hypothesis is true, given the data we have?*
  - *Example*: What is the probability that a die is not loaded (the hypothesis), given that we have gotten a six in nine out of ten die rolls (the data)
- From the frequentist point of view, $P(\text{hypothesis} \,|\, \text{data})$ is not a valid probability, since the *hypothesis* is not a repeatable experiment.

- But both frequentists and Bayesians are perfectly happy with probabilities on the form $P(\text{data} \mid \text{hypothesis})$
  - *Example*: What is the probability that we will get at least nine sixes in ten die rolls (data), assuming a probability of 1/6 to get a six in each roll (hypothesis)

2. Relation between probability and randomness

- Our interpretation of probability can also affect how we view the relation between probability and **randomness**
- If probability is just a degree of belief/knowledge (the Baysian view), there is *no necessary* link to randomness
  - In this view, probability can just as well be used to discuss outcomes of deterministic processes, or outcomes that have already happened — the use of probability can simply express a lack of information/certainty
- In the frequentist view, a probability of e.g. $P(X) = 50\%$ means that if we repeat some experiment indefinetly, the fraction of trails that give outcome X will tend towards 0.5
  - But what is the source for the variation? Why don't we always get the same result?
- This forces us to think carefully about what we exactly mean by statements like "a **random** process" or "repeating the **same** experiment"
  - If we get different outcomes in different trails, have we really repeated **the same** experiment?
  - To what extent does this depend on whether there exists some **true randomness** in Nature?
  - [Discuss: in what sense is e.g. a coin flip a *random* process?]

3. The interpretation of quantum mechanics

- We know how to *use* probability theory, and we know that it *is useful*, even if we are not sure what probabilities most fundamentally *are*
- In an analogous way, we know how to *use* quantum mechanics, and we know that it *is immensily useful* as a physical theory, but physicists and philosophers are still wondering what quantum mechanics truly *means* (and in particular, what *quantum states* really represent)
  - Recommended reading: *Philosophical Issues in Quantum Theory* in the Stanford Encyclopedia of Philosophy: plato.stanford.edu/entries/qt-issues/

- In quantum mechanics, the experimental predictions we produce from our theory come in the form of probability distributions
- So it's not surprising that several established philosophical interpretations of quantum mechanics are closely tied to particular philosophical views on probabilities

Properties of probabilities and probability density functions

My notation

- In my notation $p(x)$ can mean two different things, depending on context
- If x is a **discrete variable**:
  - $p(x)$ is the **probability** (or **probability mass**) for x
  - Example: $p(a) = \mathrm{Prob}(x = a)$
  - Since $p(x)$ is a probability, it is unitless: $[p(x)] = 1$
- If x is a **continuous variable**:
  - $p(x)$ is the **probability density** at x
  - Example: $p(a)dx = \mathrm{Prob}(x \in [a, a + dx])$
  - Since $p(x)$ is a probability density, it has units $[p(x)] = \frac{1}{[x]}$

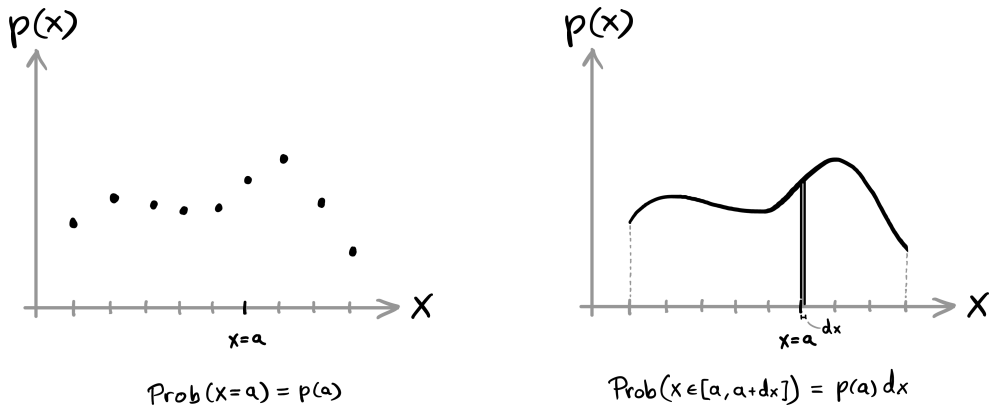

Figure 11: Examples of a probability function $p(x)$ for a discrete x (left), and a probability density $p(x)$ for a continuous x (right)

- When working with multiple variables, we *should* technically use a notation like this:

$$p_x(x), \quad p_y(y), \quad p_{x,y}(x,y), \quad p_{x|y}(x|y)$$

- Or alternatively, a notation where we give each probability (density) a completely new name:

$$p_x(x) = f(x), \quad p_y(y) = g(y), \quad p_{x,y}(x,y) = h(x,y), \quad p_{x|y}(x|y) = q(x)$$

- But personally I often find this verbose notation more confusing than helpful
- So I will be "sloppy" and use a simpler notation:

$$p(x), \quad p(y), \quad p(x,y), \quad p(x|y)$$

- In this notation, it is the argument in $p(\cdot)$ that clarifies which probability (density) function we are talking about
- In cases where this can be misunderstood, I will use a more verbose notation
- I will use the abbreviation **pdf** for *probability distribution function*, both for the continuous variable case (the pdf is a *probability density* function) and the discrete variable case (the pdf is a *probability mass* function)
  - In the statistics and probability literature, the abbreviation **pmf** is often used for *probability mass function*
- A pdf will, like any other function, have a given **domain** $\mathcal{D}$
  - When we think of x as an outcome of some experiment/sampling, $\mathcal{D}$ is often called the **sample space** or **outcome space**
    * *Example*: A die throw: $\mathcal{D} = \{1, 2, 3, 4, 5, 6\}$
    * *Example*: Survival time of a given unstable nucleus: $\mathcal{D} = [0, \infty)$
- Some terminology:
  - We will often say:
    * X *is a stochastic variable*
    * X *is random variable*
  - But for a Bayesian it may be more natural to say:
    * X *is an unknown/uncertain variable*

  - We will also often use phrases like these:
    * X *has a pdf* $p(x)$
    * X *follows a pdf* $p(x)$
    * X *is distributed as* $p(x)$
  - A shorthand notation for this, much used in the statistics and probability literature:

$$X \sim p(x)$$

  - This notation can be confusing to physicists, because we sometimes use $\sim$ to mean *roughly equal to*, or *proportional to* — which it certainly doesn't mean in the above context

Some basics

- Probabilities are numbers in $[0, 1]$
  - Discrete case: $0 \leq p(x) \leq 1$
  - Continuous case: $0 \leq p(x)dx \leq 1$
  - *Note*: A probability *density* $p(x)$ can itself have *arbitrarily large, positive numerical value*
    * It's just the product $p(x)dx$ that must be in $[0, 1]$
    * The numerical value for $p(x)$ will for instance depend on what units we use for the variable x
- Pdfs are *normalised to unity*:
  - Discrete case:

$$\sum_{x \in \mathcal{D}} p(x) = 1$$

  - Continuous case:

$$\int_{x \in \mathcal{D}} p(x)dx = 1$$

- A *function of a random variable* is itself a random variable
  - Say y is some function $y(x)$, where x is a random variable with pdf $p(x)$
  - Then y is itself a random variable, with some other pdf $p(y)$
  - Example:

* Let y be the sum in a dice throw with two dice
* If you assign a flat probabilty function over the outcomes {1,2,3,4,5,6} for each die, the probability function for the *sum* takes on a triangle shape

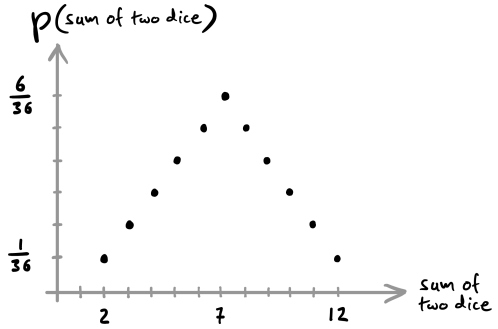

Figure 12: Probability function for the sum in a throw with two dice, if we assume a flat probability function for the outcome of each die

- For a given pdf we also have a **cumulative distribution function** (**cdf**)
  - Common notation for the cdf: $F(x)$
  - Defintion: $F(x) \equiv \text{Prob}(X \leq x)$
  - So $F(x)$ is the total probability for the random variable X to take on a value *lower than or equal to some value* x
  - For a continuous variable x, assuming the domain of x is $\mathbb{R}$:

$$F(x) \equiv \text{Prob}(X \leq x) = \int_{-\infty}^{x} p(x')dx'$$

  - Note that this means that $F(-\infty) = 0$ and $F(\infty) = 1$
  - The relationship between the cdf $F(x)$ and the pdf $p(x)$ follows from the Fundamental Theorem of Calculus: Since $F(x)$ is given by

$$F(x) = \int_{-\infty}^{x} p(x')dx'$$

    we have that $p(x)$ must correspond to the derivative of $F(x)$:

$$p(x) = \frac{d}{dx}F(x)$$

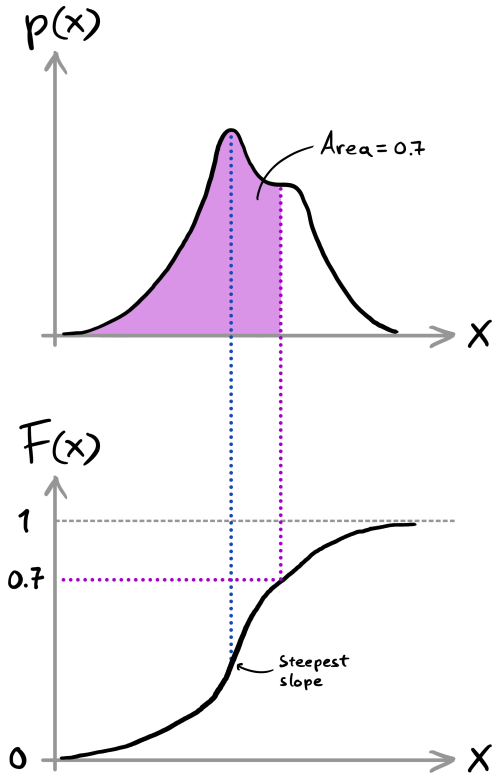

Figure 13: Illustration of the relation between a pdf $p(x)$ and the corresponding cdf $F(x)$

Some important one-dimensional probability distributions

TODO

Probability distributions of many variables

- My notation:
  - General case: $p(x_1, x_2, x_3, \ldots)$ or $p(\vec{x})$
  - When only two variables: $p(x, y)$
- In the following we will consider the two-variable case for illustration
- Need to distinguish between three types of probability distributions:
  - The **joint distribution**: $p(x, y)$ (2D distribution)
  - The **conditional distributions**: $p(x|y), p(y|x)$ (1D distributions)

  - The **marginal distributions**: $p(x), p(y)$ (1D distributions)

- Below we will consider each type of distribution in turn

The joint distribution

- Two-variable example: $p(x, y)$
- Probability distribution over the (x, y) plane
- Meaning:

$$p(x, y)dxdy = \text{Prob}(X \in [x, x + dx] \text{ and } Y \in [y, y + dy])$$

- Normalisation:

$$\int_{x\in\mathcal{D}_x} \int_{y\in\mathcal{D}_y} p(x, y)\, dx\, dy = 1$$

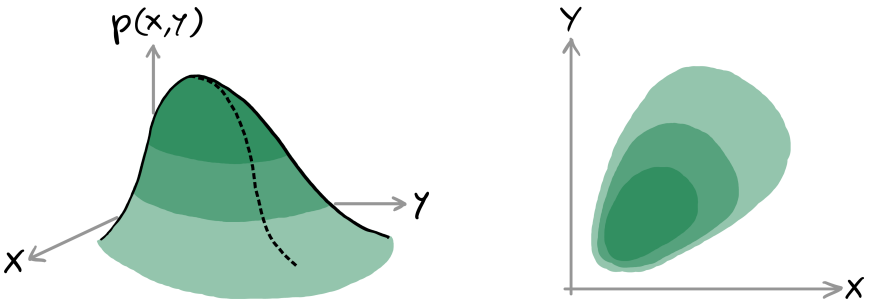

Figure 14: A two-variable joint probability distribution $p(x, y)$ visualised as a 3D figure (left) and as a 2D colour map (right)

Conditional distributions

- Two-variable examples: $p(x|y)$ and $p(y|x)$
- $p(x|y)$ is a 1D distribution over x, and similarly $p(y|x)$ is a 1D distribution over y
- Meaning:

$$p(x|y)dx = \text{Prob}(X \in [x, x + dx] \text{ given that } Y = y)$$
$$p(y|x)dy = \text{Prob}(Y \in [y, y + dy] \text{ given that } X = x)$$

- Each choice of the condition y gives a different $p(x|y)$

  - So in general, a single joint distribution $p(x, y)$ can give rise to infinitely many different $p(x|y)$ and $p(y|x)$ distributions.

- Being 1D pdfs, each $p(x|y)$ and $p(y|x)$ is normalised to integrate to unity

$$\int_{x\in\mathcal{D}_x} p(x|y)\,dx = 1, \qquad \int_{y\in\mathcal{D}_y} p(y|x)\,dy = 1$$

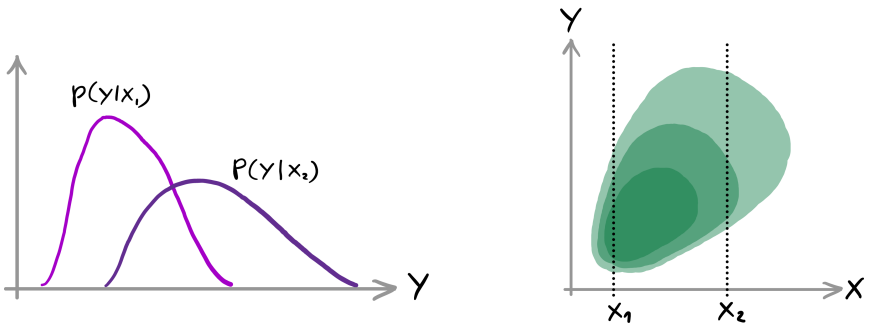

Figure 15: Two conditional probability distributions $p(y|x_1)$ and $p(y|x_2)$ (left), both derived from the joint distribution $p(x, y)$ (right)

The marginal distributions

- Two-variable example: $p(x)$ and $p(y)$, derived from the single, joint pdf $p(x, y)$
- Meaning:

$$p(x)dx = \text{Prob}(X \in [x, x + dx] \text{ regardless of } Y)$$
$$p(y)dy = \text{Prob}(Y \in [y, y + dy] \text{ regardless of } X)$$

- The marginal distributions are the distributions obtained by starting from the joint distribution and integrating out ("marginaling over") some variables:

$$p(x) = \int_{y\in\mathcal{D}_y} p(x, y)\,dy$$
$$p(y) = \int_{x\in\mathcal{D}_x} p(x, y)\,dx$$

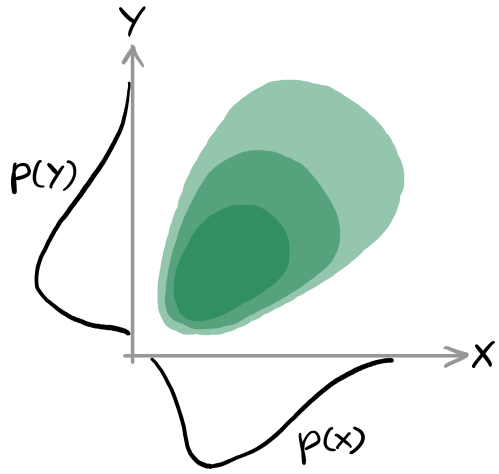

Figure 16: The two marginal probability distributions $p(x)$ and $p(y)$, derived from the joint probability distribution $p(x, y)$

Expectation values

- Let x be a random variable with pdf $p(x)$, and let y be some function of x
- The the **expectation value** of $y(x)$ is:

$$\text{Continuous case:} \quad \mathrm{E}[y] \equiv \int_{x \in \mathcal{D}} y(x)p(x)dx$$

$$\text{Discrete case:} \quad \mathrm{E}[y] \equiv \sum_{x_i \in \mathcal{D}} y(x_i)p(x_i)$$

  - Remember that here $p(x)$ is the only pdf — $y(x)$ is just some arbitrary function of x
  - Two ways of thinking about $\mathrm{E}[y]$:
    1. *Compute the $y(x)$ value corresponding to every possible x value, and combine these y values in a weighted sum, where the weight for each $y(x)$ value is the probability $p(x)$ for the corresponding x value*
    2. *If we sample an infininte number of x values from $p(x)$ and compute $y(x)$ for each sample, then $E[y]$ will be the average of all these computed y values*
- Alternative notation, much used in physics:

$$\boxed{\langle y \rangle \equiv \mathrm{E}[y]}$$

  - We will use this notation in Project 4

- Example:
  - Let x be the outcome of a coin flip: $x \in \mathcal{D} = \{\text{heads}, \text{tails}\}$
  - Let's assign a flat probability function $p(x)$:

$$\begin{aligned} p(\text{heads}) &= 0.5 \\ p(\text{tails}) &= 0.5 \end{aligned}$$

  - Now assume you are making a bet with a friend on such a coin flip
  - If the outcome is heads you win 10 NOK, and if the outcome is tails you lose 10 NOK
  - Let $y(x)$ denote the amount of money you earn in the bet
  - Given your probability assignment $p(x)$ for the coin flip, how much money do you *expect* to earn in this bet?
  - Answer:

$$\begin{aligned} \mathrm{E}[y] &= \sum_{x \in \mathcal{D}} y(x)p(x) \\ &= y(\text{heads})p(\text{heads}) + y(\text{tails})p(\text{tails}) \\ &= (10\,\text{NOK}) \times 0.5 \; + \; (-10\,\text{NOK}) \times 0.5 \\ &= 0\,\text{NOK} \end{aligned}$$

  - So, unsurprisingly, you can't expect to earn anything on this bet
  - This example illustrates an important point: *The expectation value does not necessarily correspond to a high-probability value, or even to a possible value!*
  - In our example, the only possible outcomes for y are y = 10 NOK and y = -10 NOK
    * The outcome y = 0 NOK is impossible in a single coin flip
    * But the *expected* y is still 0 NOK

Moments: particularly useful expectation values

Example: moments in physics

- Analogous concept from physics:

$$n\text{-th moment of quantity } Y = \int r^n \, [\text{density of } Y \text{ at } r] \, dr$$

- Example: *moments of mass*
  - Let $\rho(\vec{r})$ be a mass density at position $\vec{r}$ for an object of total mass M
  - The zeroth moment of mass ($n = 0$) is just the *normalisation condition*:

$$M = \int \rho(\vec{r}) \, d\vec{r}$$

  - The first moment of mass ($n = 1$), normalised by the total mass M is what we call the *centre of mass*, $\vec{R}$:

$$\vec{R} = \frac{1}{M} \int \vec{r} \rho(\vec{r}) \, d\vec{r}$$

  - The second moment of mass ($n = 2$) is the *moment of inertia*, I, describing how spread-out the mass distribution is:

$$I = \int r^2 \rho(\vec{r}) \, d\vec{r}$$

Moments of probability distributions

- The n-th moment of a pdf $p(x)$ is the expectation value $\mathrm{E}[x^n]$:

$$\mathrm{E}[x^n] = \int x^n p(x) \, dx$$

- The zeroth moment is just the normalisation condition:

$$\mathrm{E}\left[x^0\right] = \mathrm{E}[1] = 1 = \int x^0 p(x) \, dx = \int p(x) \, dx$$

- The first moment, $\mathrm{E}[x]$ or $\langle x \rangle$, is **the mean** of $p(x)$, often denoted μ:

$$\boxed{\mu = \mathrm{E}[x] = \int x p(x)\, dx}$$

  - This is just the average x value, given the pdf $p(x)$
  - It's a sum (integral) of all possible x values, each weighted by its corresponding probability (density) $p(x)$

- In general, moments are relative to some reference point c

$$\mathrm{E}[(x-c)^n] = \int (x-c)^n p(x)\, dx$$

- If make the particular choice $c = \mu$ we get the **central moments**

$$\mathrm{E}[(x-\mu)^n] = \int (x-\mu)^n p(x)\, dx$$

- The first two central moments are rather uninteresting:

$$n = 0: \quad \mathrm{E}\Big[(x-\mu)^0\Big] = \mathrm{E}[1] = 1$$

$$n = 1: \quad \mathrm{E}\Big[(x-\mu)^1\Big] = \mathrm{E}[x] - \mathrm{E}[\mu] = \mu - \mu = 0$$

- The second central moment is **the variance** of $p(x)$, denoted $\mathrm{Var}(x)$ or σ^2

$$\boxed{\mathrm{Var}(x) = \sigma^2 = \mathrm{E}\Big[(x-\mu)^2\Big] = \int (x-\mu)^2 p(x)\, dx}$$

- The variance is a measure for how spread out the pdf $p(x)$ is, i.e. to what extent much of the probability is associated with x values far away from the expected x value $\mathrm{E}[x] = \mu$

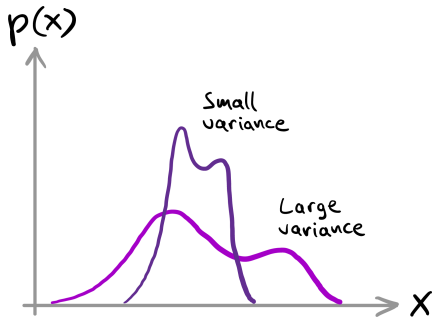

Figure 17: Variance is a measure for how spread out a probability distribution is

- We can express the variance $\mathrm{Var}(x)$ in terms of $\mathrm{E}[x]$ and $\mathrm{E}\left[x^2\right]$

$$\begin{aligned}
\mathrm{Var}(x) &= \mathrm{E}\Big[(x-\mu)^2\Big] \\
&= \int (x-\mu)^2 p(x)\, dx \\
&= \int (x^2 - 2x\mu + \mu^2) p(x)\, dx \\
&= \int x^2 p(x)\, dx - \int 2x\mu p(x)\, dx + \int \mu^2 p(x)\, dx \\
&= \mathrm{E}\Big[x^2\Big] - 2\mu \mathrm{E}[x] + \mu^2 \\
&= \mathrm{E}\Big[x^2\Big] - 2\mu^2 + \mu^2 \\
&= \mathrm{E}\Big[x^2\Big] - \mu^2 \\
&= \mathrm{E}\Big[x^2\Big] - (\mathrm{E}[x])^2
\end{aligned}$$

- We note that variance has units of $[x^2]$
- The related quantity in units of $[x]$ is the **standard deviation**, denoted σ

$$\boxed{\sigma = \sqrt{\mathrm{Var}(x)}}$$

- Computation of different means and variances will be important in project 4

Summarising probability distributions with a single number

TODO

Introduction to Monte Carlo methods

- The name refers to the famous *Casino de Monte-Carlo* in Monaco
- The term **Monte Carlo methods** refers to computational methods that in one way or another make use of sampling of random numbers
- Some typical situations in which Monte Carlo methods are used:
  - Working with uncertain/random variables

  - Simulating random/uncertain systems
  - Working with high-dimensional problems (many variables)
  - Numerical integration

- A random variable has a corresponding probability distribution function (pdf)
- For realistic cases (which often involve many variables) it is usually too complicated to work directly with the pdfs — in most cases we don't even have analytical expressions for the pdfs
- Monte Carlo approach: work with **samples from the pdfs** instead of working with the pdfs directly
- For instance we will often
  - make **histograms of samples** to approximate pdfs
  - compute **sample means** to estimate the true expectation values of pdfs
  - compute **sample variances** to estimate the true variances of pdfs
- Basic challenges
  1. Given that we are working with deterministic computers, how can we generate (seemingly) random numbers, i.e. numbers drawn from a uniform probability distribution?
  2. Assuming we *can* draw random numbers from a uniform distribution, how can we use this to draw random numbers from other probability distributions?
  3. How can we maintain sufficient computational efficiency when working in high-dimensional spaces?
  - All these topics will be discussed in subsections belo

The Monte Carlo approach, broad overview

- Assume $\vec{x}$ is an uncertain/random variable with pdf $p(\vec{x})$
- Assume we have a way of generating samples from $p(\vec{x})$
- The typical main steps of a Monte Carlo approach is then:
  1. Draw a sample $\vec{x}$ from $p(\vec{x})$
  2. Compute other quantities of interest that depend on $\vec{x}$, e.g. $f(\vec{x})$, $g(\vec{x})$ and $h(f(\vec{x})), g(\vec{x}))$
  3. Store the results we are interested in

4. Repeat from step 1

- The precision in our final estimates increases with the number of Monte Carlo samples we use
- In the end we are left with a set of $\vec{x}$ samples $\{\vec{x}_1, \vec{x}_2, \ldots\}$, as well as the corresponding samples of the computed quantities $\{f_1, f_2, \ldots\}$, $\{g_1, g_2, \ldots\}$ and $\{h_1, h_2, \ldots\}$
- Practical note:
  - Sometimes we only store a subset of the information per sample, to keep file sizes small
  - For instance, if all we are interested in is to make histograms of the f, g and h samples, we don't need to save the $\vec{x}$ samples to file

Figure 18: Three-part figure with a set of samples, the corresponding histogram and the true pdf

- Given a set of Monte Carlo samples, we can use them for many different purposes
- Here are some common examples, where N is the number of samples and we use f as our example variable:
  - Create a histogram of all our f samples to approximate the pdf $p_f(f)$
  - Create a histogram using only a specific subset of the f samples, to approximate some condiational pdf
  - Estimate some probability or integral, e.g.

$$\mathrm{Prob}(f < 10) = \int_{-\infty}^{10} p_f(f)\, df \approx \frac{N_{f<10}}{N}$$

    where $N_{f<10}$ is the number of our f samples for which $f < 10$
  - Approximate the expectation value $\mathrm{E}[f]$ using the **sample mean** $\bar{f}$

$$\mathrm{E}[f] \approx \bar{f} = \frac{1}{N}\sum_{i=1}^{N} f_i$$

- Approximate the variance $\mathrm{Var}(f)$ using the **sample variance** s_f^2

$$\mathrm{Var}(f) \approx s_f^2 = \frac{1}{(N-1)} \sum_{i=1}^{N} (f_i - \bar{f})^2$$

  * Alternatively, approximate the standard deviation $\sigma_f = \sqrt{\mathrm{Var}(f)}$ using the **sample standard deviation** s_f

$$\sigma_f \approx s_f = \sqrt{\frac{1}{(N-1)} \sum_{i=1}^{N} (f_i - \bar{f})^2}$$

- A side note on the uncertainty in using the sample mean $\bar{f}$ to approximate $\mathrm{E}[f]$:
  - The sample mean $\bar{f}$ is a function of the N random samples f_i, so $\bar{f}$ is itself a random variable with an expectation value $\mathrm{E}\left[\bar{f}\right]$ and a variance $\mathrm{Var}\left(\bar{f}\right)$
  - The sample mean $\bar{f}$ is an *unbiased estimator* for the true expectation value $\mathrm{E}[f]$, since the expectation value for $\bar{f}$ satisfies $\mathrm{E}\left[\bar{f}\right] = \mathrm{E}[f]$
  - The *variance of the sample mean*, $\mathrm{Var}\left(\bar{f}\right)$, gets smaller the more f samples that $\bar{f}$ is based on
  - $\mathrm{Var}\left(\bar{f}\right)$ is related to the variance of the original variable ($\mathrm{Var}(f)$) as follows:

$$\mathrm{Var}\left(\bar{f}\right) = \frac{\mathrm{Var}(f)}{N}$$

  - Or, in terms of standard deviations:

$$\sigma_{\bar{f}} = \frac{\sigma_f}{\sqrt{N}}$$

  - This result agrees well with our intuition:
    * If we use many f samples in our computation of $\bar{f}$, we expect that we would get a very similar $\bar{f}$ value if we were to perform the computation again with a second set of f samples
    * If we use few f samples in our $\bar{f}$ computation we expect that we could get quite a different result if we tried again with a second set of f samples

> – So, when we use a set of samples to approximate $\mathrm{E}[f]$, we can write our estimate as
>
> $$\mathrm{E}[f] \approx \bar{f} \pm \frac{s_f}{\sqrt{N}}$$

Sampling from low-dimensional pdfs

- Rejection sampling
- Inverse transform sampling

TODO

Sampling from high-dimensional pdfs: Markov chain Monte Carlo

- The previous methods for sampling from pdfs run into problems in high dimensions:
  - *Rejection sampling*: The probability of rejecting a proposed new sample will in general increase exponentially with the number of dimensions, so the method quickly becomes very inefficient
  - *Inverse transform sampling*: When working with high-dimensional pdfs, we in general don't know the cumulative distribution function, let alone its inverse
- So we need another approach
- The approach we will discuss is the famous **Markov chain Monte Carlo** (MCMC) method
- Roughly speaking, Markov chain Monte Carlo alleviates the inefficiency problem of rejection sampling by using the position of the previous accepted sample when proposing the next sample — typically from somewhere in the "neighbourhood" of the previous sample
- We will first discuss what a Markov chain is, and then look at the Markov chain Monte Carlo method

Markov chains

- A **Markov chain** is a random sequence in which *the probability for the next state only depends on the current state*, not the history of states before that.
- That the probability of the next state only depends on the current state is known as the *Markov property*

- Another way of expressing this is to say that a Markov chain is *memoryless*
- More mathematically: If $x_1, x_2, \ldots, x_{i-1}, x_i$ is our current history of states, the Markov property says that

$$\mathrm{Prob}(x_i \to x' \,|\, x_i, x_{i-1}, \ldots, x_2, x_1) = \mathrm{Prob}(x_i \to x' \,|\, x_i)$$

- This just expresses that when we know the current state x_i, the additional information about all the previous states does not alter the probability distribution for what state x' the chain will transition to next
- A Markov chain is fully defined by two pieces of information:
  1. The state space
  2. The transition probabilities between states
- An important concept for Markov chains is the **stationary distribution** or **equilibrium distribution**
  - Assume we start the chain in some initial state
  - We then keep iterating/evolving the chain and count how many times the chain visits each state
  - In the long run, the distribution of these counts will tend towards a given distribution across the state space
  - This limiting distribution is called the *stationary distribution* or *equilibrium distribution*
  - If normalised to the to the total number of iterations, the distribution is a probability distribution
  - If we think of each iteration as a time step, the stationary distribution expresses how much time, on average, the chain will spend in each state
  - A Markov chain is said to be **time reverisble** (or just **reversible**) if it satisfies the following criterion, known as **detailed balance**:

$$p(\text{state } i)\,\mathrm{Prob}(\text{state } i \to j) = p(\text{state } j)\,\mathrm{Prob}(\text{state } j \to i)$$

  - Here the probabilities $\mathrm{Prob}(\text{state } i \to j)$ and $\mathrm{Prob}(\text{state } j \to i)$ represent transition probabilities, while $p(\text{state } i)$ and $p(\text{state } j)$ are probabilities from the stationary distribution of the chain
- Next we will look at two simple Markov chain examples

Example 1: A drukard's walk on the integers

- The state space is the space of all integers
- Given a current integer i, there is a 50% probability of moving to $i + 1$ and a 50% probability for moving to $i - 1$:

$$\text{Prob}(i \to i + 1) = 0.5$$
$$\text{Prob}(i \to i - 1) = 0.5$$

Figure 19: Two-part figure showing an example walk on the space of integers, and an illustration of the state space with transition probabilities

- Assume we limit the state space to just the integers from 0 to 9, i.e. the integers modulo 10:

$$\text{Prob}(i \to i + 1 \,(\text{mod}\, 10)) = 0.5$$
$$\text{Prob}(i \to i - 1 \,(\text{mod}\, 10)) = 0.5$$

- In this case the (normalised) stationary distribution will simply be a uniform probability distribution on the integers 0 to 9

Example 2: Socks in the duvet cover

- A familiar experience: you wash your socks together with a duvet cover, and when the washing machine stops you find that many of your socks are now inside the duvet cover
- Let's try to describe this as a Markov chain
- Let each cycle of the washing machine count as one step in the Markov chain
- The state space consists of two possible states:
  1. sock inside duvet cover
  2. sock outside duvet cover

- Next we assign some transition probabilities:

$$\begin{aligned} &\text{Prob}(\text{outside} \to \text{outside}) = 0.99, \quad &&\text{Prob}(\text{outside} \to \text{inside}) = 0.01 \\ &\text{Prob}(\text{inside} \to \text{inside}) = 0.995, \quad &&\text{Prob}(\text{inside} \to \text{outside}) = 0.005 \end{aligned}$$

- Whether or not these are reasonable probability assignments is not so important for our example

Figure 20: A figure showing the state space and example probabilities for the "socks in the duvet cover" example

- This Markov chain is reversible and only has two states, so we can very easily find the stationary distribution analytically:
  - From the detailed balance equation we have

$$p(\text{inside})\,\text{Prob}(\text{inside} \to \text{outside}) = p(\text{outside})\,\text{Prob}(\text{outside} \to \text{inside})$$

  - Inserting the transition probabilities we assigned above, we get

$$\frac{p(\text{inside})}{p(\text{outside})} = \frac{\text{Prob}(\text{outside} \to \text{inside})}{\text{Prob}(\text{inside} \to \text{outside})} = \frac{0.01}{0.005} = 2$$

  - So in assigning the transition probabilities we did, we said that in the long run we think it is twice as probable to find a given sock inside as outside the duvet cover
  - If we also use the normalisation requirement for our two-state probability distribution

$$p(\text{outside}) + p(\text{inside}) = 1$$

    we get our fully defined stationary distribution:

$$p(\text{outside}) = \frac{1}{3}, \quad p(\text{inside}) = \frac{2}{3}$$

  - So, if we put 10 socks in our washing machine together with a duvet cover, our model tells us that on average we'll find 6.666…socks inside the duvet cover

    * Note that this is an example where the *expectation value* is itself an *impossible* value (assuming your washing machine does not destroy socks)

> - *A silly exercise*: Write a short program that simulates the Markov chain above
>   - Start with 10 socks in the *outside* state and 0 socks in the *inside* state
>   - For each iteration (time step) of the chain: Draw a random number from $U(0,1)$ for each sock to determine if the sock should transition to the other state
>   - After each iteration, save the current number of socks in the *outside* and *inside* states
>   - Do your numerical results agree with the stationary distribution we found above?

Ergodicity

- A Markov chain is **ergodic** if *any state can be reached from any other state in a finite number of steps*
- So ergodicity is a statement about the *connectedness* of a chain
- Our examples above, the drukard's walk on the integers 0 to 9 and the "socks in the duvet cover" example, are both ergodic chains
- Here's an example of a Markov chain that is *not* ergodic:
  - State space: A, B, C
  - Transition probabilities:

$$\begin{aligned} &\text{Prob}(A \to A) = 0.5, \quad \text{Prob}(A \to B) = 0.5 \\ &\text{Prob}(B \to C) = 1.0 \\ &\text{Prob}(C \to C) = 0.3, \quad \text{Prob}(C \to B) = 0.7 \end{aligned}$$

  - We see that if the chain is currently in state B or C, it can never reach state A at any point in the future
    * Note that the chain can of course *start* in state A, and maybe remain there for many iterations, but as soon as it makes the transition $A \to B$ it can never return to A

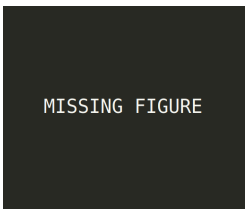

Figure 21: A figure showing the state space and transition probabilities for an example nonergodic Markov chain

Markov chain Monte Carlo

- **Markov chain Monte Carlo** (MCMC) is a method that produces a Markov chain of samples/events/steps *whose stationary distribution is a given probability distribution*
- In other words: MCMC is a method to generate a set of samples from some chosen pdf $p(x)$ by recording the steps of a Markov chain on x space
- When we say that an MCMC will generate a chain of samples $\{x_1, x_2, \ldots, x_n\}$ distributed according to a pdf $p(x)$, what we mean is that in the limit of an inifinitely long chain ($n \to \infty$), we will have

$$\frac{n_{[x,x+dx]}}{n} = p(x)dx$$

  where $n_{[x,x+dx]}$ is the number of steps in the chain where the state was in the range $[x, x+dx]$
- A word on notation:
  - For simplicity I will in the following denote a state simply as x
  - But keep in mind that MCMC is typically used to generate samples in high-dimensional spaces, meaning that we probably should have used the notation $\vec{x}$ instead
- Here are the general steps of an MCMC procudure:
  1. Generate a new candidate state x' using a **proposal pdf** that only depends on the current state x_i
  2. Apply some **acceptance rule** (see below)
     - If accepted: $x_{i+1} = x'$
     - If rejected: $x_{i+1} = x_i$
  3. Repeat from 1.

- In this procedure, it is the acceptance rule that will ensure that the stationary distribution of the Markov chain is our chosen pdf $p(x)$
- Note that at every iteration we do get a next state x_{i+1} — the acceptance rule only determines if the next state should be set equal to the current state x_i or to the candidate state x'
- For the recorded set of samples $\{x_1, x_2, x_3, \ldots\}$ to end up as a set of samples distributed according to our chosen pdf $p(x)$, we must have that
  1. the Markov chain is ergodic
  2. the acceptance rule ensures detailed balance:

$$p(x_i)\,\mathrm{Prob}(x_i \to x') = p(x')\,\mathrm{Prob}(x' \to x_i)$$

- We will now discuss the **Metropolis-Hastings algorithm**, which is a much-used acceptance rule that ensures detailed balance

The Metropolis-Hastings algorithm

- The starting point for Metropolis-Hastings is to write the transition probability $\mathrm{Prob}(x_i \to x')$ as a product of a **proposal probability** $T(x_i \to x')$ and an **acceptance probability** $A(x_i \to x')$:

$$\mathrm{Prob}(x_i \to x') = T(x_i \to x')A(x_i \to x')$$

  - If the proposal pdf is *symmetric*, that is if $T(x_i \to x') = T(x' \to x_i)$, the method is simply called the **Metropolis algorithm**

- The **MCMC procedure with a Metropolis-Hastings acceptance rule** then goes as follows:

1. Sample a candidate x' according to the proposal pdf $T(x_i \to x')$
2. Calculate the acceptance probability $A(x_i \to x')$ as

$$A(x_i \to x') = \min\left(1,\, \frac{p(x')\,T(x' \to x_i)}{p(x_i)\,T(x_i \to x')}\right)$$

   or, if the proposal pdf is symmetric (Metropolis), simply as

$$A(x_i \to x') = \min\left(1,\, \frac{p(x')}{p(x_i)}\right)$$

3. Generate a random number r from $U(0,1)$
   - If $r \leq A(x_i \to x')$, then accept x', i.e. set $x_{i+1} = x'$
   - If $r > A(x_i \to x')$, then reject x', i.e. set $x_{i+1} = x_i$
4. Repeat from 1.

- We note that this algorithm will
  - *always* accept a move to a state with higher probability (when $p(x') > p(x_i)$)
  - *sometimes* accept a move to a state with lower probability (when $p(x') < p(x_i)$)
- The probability for accepting a move to a lower-probability state x' is set such that detailed balance is satisfied
  - This is what ensures that the long-run chain of samples $\{x_1, x_2, x_3, \ldots\}$ will be distributed according to our chosen $p(x)$
- We also note that the acceptance rule only depends on $p(x_i)$ and $p(x')$ through the ratio $p(x')/p(x_i)$
  - Any normalisation constant in the pdf $p(x)$ will cancel in this ratio
  - This is important in cases where the normalisation constant in $p(x)$ is unknown or difficult to compute, but the ratio $p(x')/p(x_i)$ is easy to evaluate
  - We will encounter such a case in project 4
- We have freedom to choose the proposal distribution T
  - If we run the chain for long enough, it should theoretically visit all possible states and spend the correct "amount of time" (number of iterations) in each state, as given by $p(x)$

  - However, how long is "long enough" can depend strongly on our choice of proposal distribution T

- Below is an explanation for why we set $A(x_i \to x')$ to the value

$$A(x_i \to x') = \min\left(1, \frac{p(x')\,T(x' \to x_i)}{p(x_i)\,T(x_i \to x')}\right)$$

  - We start from the requirement of detailed balance, which when using the factorisation $\mathrm{Prob}(x_i \to x') = T(x_i \to x')A(x_i \to x')$ takes the form

$$p(x_i)\,T(x_i \to x')\,A(x_i \to x') = p(x')\,T(x' \to x_i)\,A(x' \to x_i)$$

  - The target pdf $p(x)$ is given by the particular problem we are trying to solve with MCMC, and we assume that we have already chosen our proposal pdf (the T's)
  - The only thing left to figure out are what to use as our acceptance probabilities $A(x_i \to x')$ and $A(x' \to x_i)$
  - We can rewrite the detailed balance requirement as

$$\frac{A(x_i \to x')}{A(x' \to x_i)} = \frac{p(x')\,T(x' \to x_i)}{p(x_i)\,T(x_i \to x')}$$

  - This highlights that detailed balance only sets a requirement for the *ratio* of the two acceptance probabilities
  - So we are free to choose one of the A's as we want, as long as we then set the other one according to the ratio above
  - We can use this freedom to maximise the efficiency with which we explore the pdf $p(x)$, by choosing our A's such that we *accept the proposed move to x' as often as possible*
  - That is, we want to set $A(x_i \to x')$ as large as possible, while still satisfying the ratio above
  - Both A's are probabilities, i.e. numbers between 0 and 1
  - So the way to ensure the largest possible $A(x_i \to x')$ consistent with the ratio, is to say that we will set the larger of the two A's equal to 1, and then the other A is set by the ratio above
  - The expression $A(x_i \to x') = \min\left(1, \frac{p(x')\,T(x' \to x_i)}{p(x_i)\,T(x_i \to x')}\right)$ used in the Metropolis-Hastings algorithm corresponds to making this choice
  - To see this, let's first simplify the situation by assuming we have chosen a symmetric proposal pdf, so $T(x_i \to x') = T(x' \to x_i)$

- The detailed balance requirement is then simply

$$\frac{A(x_i \to x')}{A(x' \to x_i)} = \frac{p(x')}{p(x_i)}$$

- First, let's consider the case where the proposed move is to a higher-probability x value, i.e. that $p(x') \geq p(x_i)$
  * Then we have the requirement

$$\frac{A(x_i \to x')}{A(x' \to x_i)} = \frac{p(x')}{p(x_i)} \geq 1$$

  * Here $A(x_i \to x')$ is clearly the larger of the two A's, so we set it to

$$A(x_i \to x') = 1$$

  * This choice ensures that we will *always* accept a move from a lower-probability point x_i to a higher-probability point x'
- Second, consider the case where the proposed move is to a lower-probability x value, i.e. when $p(x') < p(x_i)$
  * Then we have the requirement

$$\frac{A(x_i \to x')}{A(x' \to x_i)} = \frac{p(x')}{p(x_i)} < 1$$

  * Here the larger of the two A's is $A(x' \to x_i)$, so we set that to 1

$$A(x' \to x_i) = 1$$

    which means that $A(x_i \to x')$ becomes

$$A(x_i \to x') = \frac{p(x')}{p(x_i)}$$

- Finally, we see that we can summarise these two cases with a single probability assignment on the form

$$A(x_i \to x') = \min\left(1, \frac{p(x')}{p(x_i)}\right)$$

Burn-in

- Recap: In the theoretical limit of an infinitely long MCMC run, the resulting chain of samples $\{x_1, x_2, \ldots\}$ will be distributed according to our target pdf $p(x)$:

$$\frac{n_{[x,x+dx]}}{n} = p(x)dx$$

- In this limit, computing e.g. some expectation value $\mathrm{E}[f(x)]$ as the average of all our $f(x)$ samples would give exactly the correct result
- But in practice we can never collect an inifinte number of samples
- This means our resulting set of samples will always be somewhat influenced by where we started the chain
- If we happen to start the chain in an x region that according to $p(x)$ should be highly improbable to sample *given the limited number of samples we collect*, then the resulting distribution of samples may be significantly biased towards this region
  - In other words, the resulting set of samples will *not* be a reasonable approximation for $p(x)$
  - Consequently, any expectation values we compute from this set of samples can also be strongly biased by the starting point

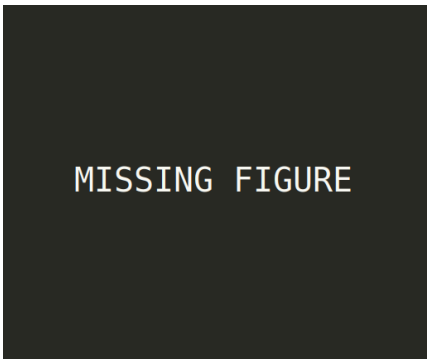

Figure 22: Illustration of burn-in

- A pragmatic solution is what is called **burn-in**: *to throw away the first set of samples that the chain collects*
- Sometimes you may also see the burn-in referred to as **equilibration** or **mixing**
- The reasoning behind this is as follows:
  - *If* we have started the chain from a "too improbable" position x, the chain will (probably) begin by mainly evolving towards the higher-probabiliy regions of x space

  - Once the chain has reached this higher-probability region, the samples collected from that time onwards will likely constitute a more reasonable representation of the target pdf $p(x)$

- So, how many of the first samples should be discarded?
- This is difficult to say!
- One practical approach is to plot some time series of your samples and see if some early part of the run has a very different behaviour (typically moving systematically in one direction) compared to the rest of the run
  - Such a time series plot will have the number of iterations along the horisontal axis, but what should the vertical axis be?
  - If x represents points in a one-dimensional space, we'd obviously just plot the x value on the vertical axis
  - But in the one-dimensional case we often wouldn't be using MCMC in the first place…
  - If we are working in a many-dimensional space, you may have to study time series plots for many x coordinates
  - Or, if you are mainly interesed in some derived quantity $f(x)$, the time series plot for that quantity is probably the most useful
  - It can also be useful to study the time series of some expectation value $\mathrm{E}[f(x)]$, which at each step of the time series is computed with all the samples collected up to that iteration
- Finally: keep in mind that the burn-in issue becomes less important the longer we can run our MCMC for
- In the limit of infinite samples there is no burn-in issue at all — then all parts of $p(x)$ will be explored in the right proportion
- For an interesting read/rant about burn-in, see this web page: http://users.stat.umn.edu/~geyer/mcmc/burn.html

Correlated samples

- When we are generating samples from some pdf $p(x)$, we usually want **independent samples**
- That is, in the ideal case each sample is generated completely independently from all other samples
- Going back to the methods for low-dimensional sampling, both *rejection sampling* and *inverse transform sampling* are methods that produce independent samples from the given $p(x)$

- But for a Markov chain Monte Carlo, we have seen that a key part is to use the current step x_i to propose the next step x_{i+1}
- So MCMCs typically generate **correlated samples**
  - If I tell you the value of sample x_{1433} you will probably make a more accurate guess for the value of sample x_{1436} than you will if I don't tell you the value of x_{1433}
  - In other words: $\mathrm{Prob}(x_{1436} \mid x_{1433}) \neq \mathrm{Prob}(x_{1436})$
  - Using a wide proposal pdf, i.e. a proposal pdf that often suggests large steps, will give a lower correlation between samples
    * But it can also lead to slower exploration of $p(x)$ and longer burn-in time, since more steps are likely to be rejected
- n correlated samples contain less information about $p(x)$ than n independent samples do
  - This makes intutitive sense: if two samples, say x_i and x_{i+1} are correlated, then part of the information about $p(x)$ provided by sample x_{i+1} was already provided by sample x_i
- This is often quantified by computing an **effective sample size**, n_{eff}, based on the n samples the MCMC produced
- The effective sample size expresses the actual information content in the n correlated samples by estimating how many *independent* samples the correlated samples correspond to
- In most cases we will have $n_{\text{eff}} < n$
- So, if we treat our n MCMC samples as if they are independent samples $p(x)$, we will usually be *underestimating our uncertainties*
  - *Example*: Say we naively estimate some expecation value $\mathrm{E}[f(x)]$ as

    $$\mathrm{E}[f(x)] \approx \bar{f} \pm \frac{s}{\sqrt{n}}$$

    where $\bar{f}$ is the sample average, s_f is the sample standard deviation and n is the number of collected MCMC samples
  - In dividing by $\sqrt{n}$ we are assuming that our samples are independent
  - To more accurately estimate our uncertainties, we should divide by $\sqrt{n_{\text{eff}}}$ instead
- We will not discuss effective sample size in more detail here, but it is important to be aware of the concept and keep in mind that MCMC samples are typically *not* independent
- For more detailed explanation and examples, see e.g. these pages:

  - https://andrewcharlesjones.github.io/journal/21-effective-sample-size.html
  - https://mc-stan.org/docs/2_21/reference-manual/effective-sample-size-section.html

Thinning

- **Thinning** refers to a common MCMC practice of *only storing every m-th step in the chain*
- The reason for doing this is typically to
  - save disk space
  - save memory, if samples are not immediately written to file
  - save time, if you have to evaluate some computationally costly $f(x)$ at every saved x sample
- In particular, if the proposal distribution is such that the chain usually makes very small steps, it may be impractical to save every step
- When thinning is used, the degree of correlation between the saved samples is lower compared to if every step in the chain was saved
  - This means that naive error estimates based on $\sqrt{n}$ (see example above) will be less wrong, compared to if thinning was not used
  - However, if it's practically doable, the best approach is usually to save every MCMC step, compute an estimate for the effective sample size n_{eff}, and use this in uncertainty estimates
- In project 4 we will in practice use a form of thinning: Our MCMC will make many tiny steps, but only save every N-th step, where N is the number of spins in a 2D Ising model, and we will ignore the fact the saved samples are correlated

Physics of Project 4: the Ising model

- We should probably call it *the Ising-Lenz model* (Ernest Ising, Wilhelm Lenz, 1920s)
- Originally a model for ferromagnetism in statistical mechancics, but the same basic idea have since been used to model *many* different phenomena in science
- We will work with the 2D Ising model
- Basic setup of the model:
  - A lattice/grid of spins (variables) that can only take the values -1 (↓) or +1 (↑)

$$\begin{array}{ccccccc} & \vdots & \vdots & \vdots & \vdots & & \\ \ldots & \uparrow & \downarrow & \uparrow & \downarrow & \ldots & \\ \ldots & \uparrow & \uparrow & \downarrow & \downarrow & \ldots & \\ \ldots & \downarrow & \downarrow & \uparrow & \downarrow & \ldots & \\ \ldots & \uparrow & \uparrow & \uparrow & \downarrow & \ldots & \\ & \vdots & \vdots & \vdots & \vdots & & \end{array} \quad \Leftrightarrow \quad \begin{array}{ccccccc} & \vdots & \vdots & \vdots & \vdots & & \\ \ldots & +1 & -1 & +1 & -1 & \ldots & \\ \ldots & +1 & +1 & -1 & -1 & \ldots & \\ \ldots & -1 & -1 & +1 & -1 & \ldots & \\ \ldots & +1 & +1 & +1 & -1 & \ldots & \\ & \vdots & \vdots & \vdots & \vdots & & \end{array}$$

  - Each spin interacts with its nearest neighbours

$$\begin{array}{ccccc} & & \bullet & & \\ & & \updownarrow & & \\ \bullet & \leftrightarrow & \bullet & \leftrightarrow & \bullet \\ & & \updownarrow & & \\ & & \bullet & & \end{array}$$

  - ↑↑ and ↓↓ pairs have lower energy than ↑↓ and ↓↑ pairs
  - The system can exchange heat with the environment (a thermal bath at temperature T)
  - Change in system energy due to heat ↔ spins flip ("thermal fluctuations")
- The main topics for Project 4:
  - We will study the properties of the system at equilibrium, as function of the chosen temperature T and lattice size
  - Properties of interest:
    * Mean energy
    * Mean magnetisation

    * Heat capacity
    * Magnetic susceptibility
  - We will also determine *the critical temperature* (T_c)
    * If we gradually lower the temperature, T_c is the temperature at which the system undergoes *a phase transition*, from a *disordered, non-magnetised state* to an *ordered, magnetised state*
    * In 1944 Lars Onsager found an analytical expression for T_C for the 2D Ising model with an infinitely large lattice
    * We will use our numerical results for T_c for finite-sized lattices to extrapolate towards the infinite limit and compare to Onsager's result

Overview of important variables and functions

- There are many different quantities to keep track of in this project
- Let's introduce them one by one
- Lattice length: L
- Number of spins in lattice: $N = L^2$
- Spin value for a single spin: $s_i \in \{-1, +1\}$
- A *spin configuration* or *microstate* (or simply *state*)
  - This is the state of the entire lattice of spins
  - To keep the notation simple, we will here use vector notation to denote a microstate:

$$\vec{s} = [s_1, s_2, \dots, s_N]$$

  - In your code it may be more natural to represent the microstate (spin lattice) as a *matrix*, since we are working with a 2D model
  - The N-dimensional state space (the space of all possible $\vec{s}$) contains 2^N possible states, since each individual spin can take two values
- The system energy for a given state $\vec{s}$:

$$E(\vec{s}) = -J \sum_{\langle kl \rangle} s_k s_l$$

  - Here $\sum_{\langle kl \rangle}$ denotes the sum over all neighbouring *spin pairs*, with no double-counting
  - J denotes the coupling constant that sets the interaction strength
  - We will choose $J > 0$
  - The choice $J > 0$, combined with the overall negative sign in the $E(\vec{s})$ expression, means that:
    * An equal-spin interaction (↑↑ or ↓↓) will *lower* the system energy by $-1J$, since $s_k s_l = (\pm 1)(\pm 1) = 1$
    * An unequal-spin interaction (↑↓ or ↓↑) will raise the system energy by $+1J$, since $s_k s_l = (\pm 1)(\mp 1) = -1$
  - That aligned spins give lower energy is the key modelling choice that makes this a model of ferromagnetism
  - We do not include an external magnetic field
    * If we had done so, we would have additional terms in our $E(\vec{s})$ expression to represent the interaction between the individual spins and the external magnetic field

- The system magnetisation for a given state $\vec{s}$:

$$M(\vec{s}) = \sum_{i=1}^{N} s_i$$

- State *degeneracy*:
  - The *number of different states* $\vec{s}$ that have the same value for some quantity, e.g. $E(\vec{s})$ or $M(\vec{s})$
  - Consider an example with a simple one-dimensional model with only three spins

* All possible states along with their magnetisations:

$$\begin{aligned}
\vec{s}_1 &= [+1,+1,+1], \quad M(\vec{s}_1) = 3 \\
\vec{s}_2 &= [+1,+1,-1], \quad M(\vec{s}_2) = 1 \\
\vec{s}_3 &= [+1,-1,+1], \quad M(\vec{s}_3) = 1 \\
\vec{s}_4 &= [+1,-1,-1], \quad M(\vec{s}_4) = -1 \\
\vec{s}_5 &= [-1,+1,+1], \quad M(\vec{s}_5) = 1 \\
\vec{s}_6 &= [-1,+1,-1], \quad M(\vec{s}_6) = -1 \\
\vec{s}_7 &= [-1,-1,+1], \quad M(\vec{s}_7) = -1 \\
\vec{s}_8 &= [-1,-1,-1], \quad M(\vec{s}_8) = -3
\end{aligned}$$

* We see that:
  - $M = 3$ has a degeneracy of 1
  - $M = 1$ has a degeneracy of 3
  - $M = -1$ has a degeneracy of 3
  - $M = -3$ has a degeneracy of 1

- Probability distribution for $\vec{s}$: *the Boltzmann distribution*

$$p(\vec{s};T) = \frac{1}{Z} e^{-\frac{E(\vec{s})}{k_B T}} = \frac{1}{Z} e^{-\beta E(\vec{s})}$$

  - $\beta = \frac{1}{k_B T}$, where k_B is the Boltzmann constant
  - The normalisation factor Z is the so-called *partition function* (see below)
  - Choosing the Boltzmann distribution as our probability distribution $p(\vec{s};T)$ means that we are studying the system at thermal equilibrium with the environment
  - Keep in mind that $p(\vec{s};T)$ is the pdf for the *states* $\vec{s}$, it is *not* the pdf for energy $E(\vec{s})$
    * Easy to forget, since $\vec{s}$ only appears in $p(\vec{s};T)$ via the energy $E(\vec{s})$
    * If we insert our expression for $E(\vec{s})$ we more clearly see how $p(\vec{s};T)$ is a joint pdf for

the individual components of $\vec{s}$:

$$\begin{aligned} p(\vec{s};T) &= p(s_1, s_2, \ldots, s_N; T) \\ &= \frac{1}{Z} e^{-\beta E(\vec{s})} \\ &= \frac{1}{Z} e^{\beta J \sum_{\langle kl \rangle} s_k s_l} \end{aligned}$$

- The partition function:

$$Z = \sum_{\substack{\text{all possible} \\ \text{states } \vec{s}}} e^{-\beta E(\vec{s})}$$

  - It is called the partition *function* since it is a function of the temperature T (via $\beta = \frac{1}{k_B T}$)
  - For a given temperature, Z serves as the normalisation constant in the pdf $p(\vec{s};T)$
  - It essentially describes how the distribution of probability across the state space changes when T changes
  - It can be used to derive how various thermodynamic quantities depend on temperature

- Probability distributions for other quantities
  - In this project we will explore numerically what the pdf for energy, $p_E(E;T)$, looks like
    * We will use MCMC to sample microstates $\vec{s}$ from $p(\vec{s};T)$
    * For each sampled microstate $\vec{s}$ we can compute the corresponding $E(\vec{s})$
    * This $E(\vec{s})$ can then be seen as one sample from the unknown pdf $p_E(E;T)$
  - Similarly, we can compute the magnetisation $M(\vec{s})$ for each sampled microstate, and view these M values as samples from the unknown pdf $p_M(M;T)$

- We will use our E and M samples to compute various expectation values:
  - $\langle E \rangle$, $\langle E^2 \rangle$, $\langle |M| \rangle$, $\langle M^2 \rangle$
  - The reason why we work with the absolute value of the magnetisation, $|M|$:
    * Since we do not have an external magnetic field to pick out a preferred direction, there is an overall and exact "up-down symmetry" in our problem:
      · For every state $\vec{s}$ there is an opposite state $-\vec{s}$ (the state where *all* the individual spins are flipped) that has exactly the same energy, $E(-\vec{s}) = E(\vec{s})$, and hence has the same probability, $p(-\vec{s};T) = p(\vec{s};T)$

  - But this opposite state $-\vec{s}$ would have the *opposite* magnetisation: $M(-\vec{s}) = -M(\vec{s})$
  - So in our simple model, the expectation value of the signed magnetisation, $\langle M \rangle$, would technically always be 0, independent of the temperature T
  - But in reality, one direction is spontaneously/randomly chosen and we see a strong dependence on the degree of magnetisation with temperature
  - To capture this effect without worrying about whether the magnetisation is in the up or down direction, we will work with the absolute value $|M|$

- To easily compare results obtained for different lattice sizes, we will often work with energy and magnetisation *per spin*:
  - Our notation:
$$\frac{E}{N} \equiv \epsilon, \qquad \frac{M}{N} \equiv m$$
  - This means we will also be interested in the expectation values
$$\langle \epsilon \rangle = \left\langle \frac{E}{N} \right\rangle = \frac{1}{N} \langle E \rangle$$
$$\langle |m| \rangle = \left\langle \frac{|M|}{N} \right\rangle = \frac{1}{N} \langle |M| \rangle$$

- Heat capacity (C_V) and magnetic susceptibility (χ):
  - We will compute these using the expressions
$$\frac{C_V(T)}{N} = \frac{1}{N} \frac{1}{k_B T^2} \mathrm{Var}(E) = \frac{1}{k_B T^2} \left[\langle E^2 \rangle - \langle E \rangle^2 \right]$$
$$\frac{\chi(T)}{N} = \frac{1}{N} \frac{1}{k_B T} \mathrm{Var}(M) = \frac{1}{k_B T} \left[\langle M^2 \rangle - \langle |M| \rangle^2 \right]$$
    since these expressions are given directly in terms of some simple expectation values
  - Note that here we have also included a factor $1/N$, to get heat capacity and magnetic susceptibility per spin

> Some probably more familiar expressions for these quantities are
>
> $$C_V(T) = \frac{\partial \langle E \rangle}{\partial T}$$
>
> $$\chi(T) = \frac{\partial \langle M \rangle}{\partial H}$$
>
> where H is magnetic field intensity

The basic idea in Project 4

- Choose a lattice size (L) and a temperature (T)
- Use MCMC to sample microstates $\vec{s}$ according to the Boltzmann distribution $p(\vec{s}; T)$ and compute derived quantities of interest, such as the corresponding energy samples $E(\vec{s})$ and various expectation values ($\langle E \rangle, \ldots$)
- Repeat for different choices of lattice size and temperature
- Study how the results depend on the lattice size and the temperature, and in particular how the system behaves for temperatures close to the critical temperature T_c
- A word of warning: *Know your types of sums*
  - This project involves computing many different types of sums
  - Make sure you know what you are supposed to be summing over
  - Examples:

$$\sum_{\substack{\text{all spins} \\ \text{in lattice}}} \qquad \sum_{\substack{\text{all neighbouring} \\ \text{spin pairs}}} \qquad \sum_{\substack{\text{all possible} \\ \text{states } \vec{s}}} \qquad \sum_{\text{all } \vec{s} \text{ samples}}$$

$$\sum_{\substack{\text{all possible} \\ E(\vec{s}) \text{ values}}} \qquad \sum_{\text{all } E(\vec{s}) \text{ samples}} \qquad \ldots$$

Suggested algorithm for Project 4

- As mentioned earlier in the section on *thinning*, for Project 4 we suggest you implement an MCMC algorithm that performs many tiny steps in the space of microstates, but only saves/uses the state at every N-th step for further computations
- One reason for this is purely practical:

  - We *could* have saved/used the state after every single step as a new MCMC sample
  - But we would end up collecting a huge number of almost-identical samples, which would cost more disk space and/or memory

- Another reason why we only use every N-th step is that the collected samples are then less correlated with each other
  - This means we can more reasonably ignore complications like *autocorrelation* and *effective sample size* when we compute variances and error estimates
- We'll use the following terminology:
  - *A step*: Flip a single spin in our 2D lattice and accept/reject the move to this new microstate
  - *A cycle*: Attempt N single-spin flips
- So, *a single MCMC cycle* consists of N *steps*
- The total number of MCMC cycles = the number of pdf samples we collect/use
- Here's the suggested algorithm:

1. Choose a temperature (T) and a lattice size ($L \times L = N$ = number of spins)
2. Choose the number of MCMC cycles (n_{cycles}) to perform
3. Choose an initial state ($\vec{s}$) for the system
4. For each MCMC cycle:
   a) For each step within the cycle:
      i. From the current state $\vec{s}$, generate a new candidate state $\vec{s}\,'$ by picking a random spin in the lattice (use a uniform pdf) and flip it
      ii. Compute the ratio $\frac{p(\vec{s}\,')}{p(\vec{s})}$
      iii. Generate a random number $r \sim U(0,1)$ and accept the move $\vec{s} \to \vec{s}\,'$ if $r < \frac{p(\vec{s}\,')}{p(\vec{s})}$
   b) After N steps, use the resulting state $\vec{s}$ to compute any derived quantities of interest, e.g. the current energy and magnetisation
   c) Store all the numbers you need

- A few comments to the suggested algorithm above:

  - Our procedure for generating $\vec{s}'$ (select a single spin at random and flip it) means that our proposal distribution $T(\vec{s} \rightarrow \vec{s}')$ is a uniform distribution on all the "neighbour states" of the current state $\vec{s}$
  - For every tiny step $\vec{s} \rightarrow \vec{s}'$ we attempt we will need the ratio $\frac{p(\vec{s}')}{p(\vec{s})}$, so you should think about how you can evaluate this ratio as efficiently as possible
  - In particular, note that we just need the *ratio* $\frac{p(\vec{s}')}{p(\vec{s})}$, not the individual $p(\vec{s})$ and $p(\vec{s}')$

Parallel computing

- Examples of parallelisation in C++ using OpenMP: github.com/anderkve/FYS3150/tree/master/code_examples/omp_parallelization

TODO

Random number generation

- We have seen that being able to generate samples x from some probability distribution $p(x)$ can be incredibly useful
- But our starting assumption has always been that we are able to generate samples (numbers) from a uniform (flat) distribution
- How can we do this, given that our computers are deterministic?
- This is the purpose of a **random number generator** (RNG)
- Old-school ways of generating random numbers: dice, coin flips, …
- Hardware RNGs can be used to generate "actually random" numbers, i.e. series of numbers that we can't predict
  - Generates numbers based on some unpredictable feature of the physical environment, e.g. thermal noise
- But we will focus on the concept of a **pseudorandom number generators** (PRNG): a *deterministic algorithm* that produce numbers that *are predetermined, but that still appear random (unpredictable)*
- PRNG algorithms are initialised by a starting number, called the **seed**

  - Two PRNG runs that start from the same seed will produce the same sequence of numbers (great for reproducibility of simulations, MCMC runs, etc.)

- Desired properties of pseudorandom number generators:

  1. Produce numbers that are distributed according to the standard uniform distribution, $U(0, 1)$
  2. Negligble correlations between numbers

  - Knowing a previous nymber shouldn't help you in guessing the next number — unless you actually know the algorithm, of course

  3. The **period** before the sequence of numbers repeates should be as long as possible (more on that below)
  4. The algorithm should be computationally cheap

- A classic PRNG algorithm is the **Linear Congruential Generator** (LCG), which dates back to the 1950s
- The algorithm is as follows

$$\boxed{N_{i+1} = (aN_i + c) \bmod (m)}$$

  where we have the following parameters:

  - a (the *multiplier*): $0 < a < m$
  - c (the *increment*): $0 \leq c < m$
  - m (the *modulus*): $0 < m$
  - N_0 (the *seed*): $0 \leq N_0 < m$

> A side note on the modulo operator:
>
> - The modulo operator returns the remainder after division

> – *Examples*:
>
> $$13 \bmod (2) = 1$$
> $$17 \bmod (5) = 2$$
> $$8 \bmod (8) = 0$$
> $$16 \bmod (17) = 16$$

- To get a number x_i on $[0, 1)$ we simply do $x_i = N_i/m$
- How good the generator is dependes crucially on the choice of parameters: a, c, m, and potentially N_0
  - There is a lot of research on the parameter choices for LCGs and other PRNG algorithms
- Some silly examples, for illustration:
  - *Example 1*:
    * Parameters: $m = 9$, $a = 2$, $c = 0$, $N_0 = 1$
    * Sequence: $1, 2, 4, 8, 7, 5, 1$ (repeats)
    * Period: 6
  - *Example 2*:
    * Parameters: $m = 9$, $a = 2$, $c = 0$, $N_0 = 3$
    * Sequence: $3, 6, 3$ (repeats)
    * Period: 2
  - *Example 3*:
    * Parameters: $m = 9$, $a = 4$, $c = 1$, $N_0 = 0$
    * Sequence: $0, 1, 5, 3, 4, 8, 6, 7, 2, 0$ (repeats)
    * Period: 9
- A more realistic example:
  - Parameters: $m = 2^{32} = 4\,294\,967\,296$, $a = 1\,664\,525$, $c = 1\,013\,904\,223$
  - This is the *Rand1* algorithm from the book *Numerical Recipes*
- It's recommended to look up the period of a PRNG that you plan to use for serious work
  - Some famous PRNGs have had surprisingly short periods, meaning the results are no longer random-looking when a sufficiently large number of samples are needed

  - Famous worst-case example: the *RANDU* algorithm (IBM, 1960s)
    * While the samples looked uniformly distributed in 1D and 2D
    * But when used to generate points in a 3D space, these points would all lie on particular 2D planes in the 3D space
    * This algorithm was used in many applications and research papers before this problem was discovered…

- The period is not the only concern:
  - Question: What happens for the case $m = \text{some large number}$, $a = 1$, $c = 1$?
  - Answer: We simply get a "modulo counter"
    * Sequence: $N_0, N_0 + 1, N_0 + 2, \ldots$
  - The period may be long, but does this sequence look random/unpredictable? No!
- There are collections of statistical randomness tests that are used to test PRNG algorithms
  - All PRNGs fail some such tests…
  - Famous sets of tests: *Diehard tests* and *TestU01*
  - Donald Knuth (the inventor of TeX) was the first to propose a set of such tests
- Other examples of PRNG algorithms:
  - The **shift-register** algorithm:

$$N_{i+1} = (aN_{i-j} + cN_{i-k}) \bmod (m)$$

    * This uses more than just the preceeding number N_i
  - The **Mersenne Twister** algorithm:
    * A very commonly used algorithm
    * Developed in 1997 by Makoto Matsumoto and Takuji Nishimura
    * Available from the <`random`> library in `C++11`
    * The standard version is known as `MT19937` and has a period of $2^{19937} - 1$, which is more than 10^{6001}
- A common pitfall when using PRNGs in parallelised code
  - If the PRNGs on different threads/processes use the same seed, they will generate identical sets of samples!

- In that case the result would be equivalent to many identical copies of a single run, instead of many independent runs (which is probably what you want)
- A trick to give each thread a different seed while still starting from a single seed:
  * Start by choosing a base seed
  * Then let each thread generate its own unique seed by combining (in some way) the base seed with its own *thread number*
  * Then each thread will have its own seed, but the results are still perfectly reproducible from just the base seed, as long as you run with the same number of threads

Numerical integration: high-dimensional integrals

TODO

Numerical integration: low-dimensional integrals

TODO