Contents

Preface **vii**
Course information . . . vii
A note on writing our own code . . . viii
A note on our choice of programming language . . . ix
Sources . . . x
Visual elements in these notes . . . xi

0 Module 0: Hello, π! Julia as a second (programming) language **1**

1 Setting up Julia **4**
1.1 Installing Julia . . . 4
1.2 The REPL . . . 4
1.2.1 Adding packages . . . 5
1.2.2 Configuring the REPL . . . 6
1.3 Hello, π! (Method 1: Asking a friend) . . . 6
1.4 Notebooks and IDEs . . . 7

2 Variables, primitive types, and functions **10**
2.1 Variables and types . . . 10
2.2 Functions and control flow . . . 12
2.2.1 Operators and special functions . . . 12
2.2.2 Writing functions . . . 13
2.2.3 Function arguments . . . 13
2.3 Hello, π! (Method 2: Computing functions) . . . 15
2.3.1 Our first loop! . . . 15
2.4 Expressiveness in code . . . 16

3 Composite types and data structures **19**
3.1 Collections . . . 19
3.1.1 Tuples . . . 19
3.1.2 Arrays, Vectors, and Matrices . . . 20
3.1.3 Functions on Arrays . . . 22
3.1.4 Dictionaries, Sets, and the rest . . . 23
3.2 Iteration and Loops . . . 24

3.2.1 Ranges . . . 25
3.2.2 Collecting and Comprehending . . . 25
3.3 Structs and constructors . . . 26
3.4 Hello, π! (Method 3: Using geometry) . . . 27
3.4.1 A geometric solution . . . 29
3.5 Naming conventions and commenting code . . . 30

4 Data, plots, and visualization **33**
4.1 Reading and writing data . . . 33
4.1.1 File input and output . . . 34
4.1.2 Reading and writing simple delimited data . . . 35
4.2 Visualizing data . . . 36
4.2.1 Julia's plotting ecosystem . . . 36
4.2.2 A simple scatter plot . . . 37
4.3 Hello, π! (Method 4: Using noise) . . . 39
4.4 Performance and profiling . . . 41
4.4.1 Profiling . . . 42

5 Projects, parametric types, and multiple dispatch **45**
5.1 Environments . . . 45
5.2 Modules . . . 47
5.2.1 Scopes in Julia . . . 50
5.3 Testing our code . . . 52
5.4 Building type hierarchies: parametric and abstract types . . . 54
5.4.1 Parametric composite types . . . 54
5.4.2 Abstract types and subtyping . . . 55
5.5 Multiple dispatch . . . 57
5.6 Hello, π! (Method 5: Counting collisions) . . . 59
5.7 Additional resources . . . 62
5.8 Confession . . . 63

I Module 1: Introduction and foundations **65**

6 Version control with Git **67**
6.1 Git in practice . . . 67
6.1.1 The core commands . . . 67
6.1.2 Configuring git . . . 72
6.1.3 Common workflows . . . 73
6.2 How git stores a repository . . . 75
6.2.1 Blobs and trees . . . 75
6.2.2 Commits . . . 77
6.2.3 References . . . 80

7 Blueprints for a computational research project 82
7.1 Organizing your project . . . 82
7.1.1 File organization . . . 83
7.2 Planning and executing computational experiments . . . 84
7.2.1 Fermi estimation for your code . . . 85
7.2.2 Designing scripts for reproducibility . . . 86
7.3 Algorithms and the logical architecture of a project . . . 92
7.3.1 Algorithmic complexity . . . 92
7.3.2 Performance vs. Simplicity . . . 94
7.4 Coda . . . 96

II Module 2: ODEs and molecular dynamics 97

8 Ordinary differential equations 99
8.1 The naive solution: Euler's Method . . . 99
8.2 Case study: N-body simulations and planetary dynamics . . . 100
8.2.1 First attempt: A monolithic script . . . 100
8.2.2 A top-down design . . . 102
8.2.3 Particle data structures . . . 104
8.2.4 Integrators . . . 105
8.2.5 Force calculators . . . 107
8.2.6 Results . . . 107

9 Integration schemes for ODEs 110
9.1 Higher-order integrators . . . 110
9.1.1 Deriving the RK2 Family . . . 111
9.1.2 The Butcher tableau . . . 112
9.1.3 The workhorse ODE solver: RK4 . . . 113
9.1.4 Planetary dynamics with RK4 . . . 114
9.2 Time-reversible integration . . . 115
9.2.1 Symplectic Euler integration . . . 116
9.2.2 The Velocity Verlet algorithm . . . 117
9.3 Symplectic integrators and energy conservation . . . 118
9.3.1 The Liouvillian . . . 118
9.3.2 Backward error analysis and the shadow Hamiltonian . . . 119

10 Molecular Dynamics 121
10.1 An N-body problem in a box . . . 121
10.1.1 Periodic boundary conditions . . . 122
10.1.2 Interparticle potentials . . . 124
10.1.3 Initial conditions . . . 125
10.2 Force calculations for short-ranged interactions . . . 125
10.2.1 Neighbor list structures . . . 126
10.3 Thermodynamic ensembles and equilibration . . . 127

10.3.1 Computational thermostats . . . 127
10.4 Observables and measurements . . . 130
10.4.1 Equilibration . . . 130
10.4.2 From microscopic trajectories to macroscopic properties . . . 131
10.5 Coda . . . 133

III Module 3: PDEs and hydrodynamics 135

11 The landscape of partial differential equations 137
11.1 From particles to fields . . . 137
11.1.1 Coarse graining . . . 137
11.1.2 Behavior of the continuum . . . 138
11.2 The PDE zoo . . . 138
11.2.1 Elliptic equations . . . 139
11.2.2 Parabolic equations . . . 139
11.2.3 Hyperbolic equations . . . 139
11.3 Finite differencing . . . 140
11.4 The method of lines . . . 140

12 Systems evolving in one dimension: a chapter that needs a real title 142
12.1 Parabolic equations / stability crisis . . . 142
12.1.1 Stability analysis . . . 142
12.1.2 Implicit methods . . . 143
12.2 Hyperbolic equations / stability crisis . . . 144
12.2.1 Upwind schemes . . . 145
12.3 The onset of chaos: nonlinearity and shocks . . . 145
12.4 The zoo of methods . . . 145
12.4.1 Finite Volume Method (FVM) . . . 145
12.4.2 Finite Element Method (FEM) . . . 146
12.4.3 Spectral Methods . . . 146
12.4.4 Lattice Boltzmann (LBM) . . . 146

13 Steady states and flows in two dimensions 147
13.1 Elliptic equations: solving for equilibrium . . . 147
13.1.1 Iterative solution methods . . . 147
13.2 A glimpse of computational fluid dynamics . . . 147
13.2.1 The Navier-Stokes equations . . . 147
13.2.2 Anatomy of a CFD Solver . . . 147
13.3 Coda: Frontiers in PDE research . . . 147
13.3.1 High-order and spectral element methods . . . 147
13.3.2 Multiphysics and multiscale methods . . . 148
13.3.3 Machine learning and the curse of dimensionality . . . 148

IV Module 4: Random numbers and Monte Carlo methods 149

14 Pseudorandomness and Monte Carlo integration 151
14.1 Pseudorandom number generators . . . 151
14.1.1 A simple PRNG: linear congruential generators . . . 151
14.1.2 Modern approaches . . . 153
14.1.3 Reproducibility and Seeding . . . 154
14.1.4 Generating non-uniform random numbers . . . 154
14.2 Application: Monte Carlo Integration . . . 156
14.2.1 The Basic Idea: Throwing Darts . . . 156
14.2.2 Convergence and the curse of dimensionality . . . 156
14.3 Importance sampling . . . 157
14.3.1 Biasing and re-weighting . . . 157
14.3.2 The bridge to statistical physics . . . 158

15 The Metropolis algorithm 160
15.1 Markov chain Monte Carlo . . . 160
15.1.1 Engineering the stationary distribution . . . 162
15.1.2 The Metropolis-Hastings algorithm . . . 162
15.2 Case Study I: The Ising model . . . 164
15.2.1 A top-down design . . . 165
15.2.2 System data structures . . . 166
15.2.3 Monte Carlo moves . . . 168
15.2.4 Energy calculations . . . 169
15.2.5 Analyzing MCMC data . . . 170
15.3 Case Study II: Particle-based systems . . . 172
15.3.1 Implementation details . . . 172
15.3.2 Comparing Methods: MC vs. MD . . . 175

16 Hamiltonian Monte Carlo and Bayesian inference 176
16.1 Hybrid Monte Carlo . . . 176
16.2 Bayesian parameter inference . . . 178
16.3 Hamiltonian Monte Carlo . . . 179
16.3.1 Case study: inferring exponential decay . . . 180
16.4 Flies in the ointment . . . 183
16.4.1 The trouble with tuning . . . 183
16.4.2 The problem with partials . . . 184

V Module 5: Machine learning in physics 185

17 Derivatives 187
17.1 Symbolic differentiation . . . 187
17.2 Numerically approximate differentiation . . . 188
17.3 Forward mode automatic differentiation . . . 192

17.3.1 An Algebra for Derivatives . . . 192
17.3.2 Implementing a `Dual` Type in Julia . . . 193
17.3.3 Differentiating through algorithms . . . 194
17.3.4 The direction of differentiation . . . 196
17.4 Reverse mode AD . . . 196
17.4.1 Reversing the direction of differentiation . . . 197
17.4.2 Implementing a tape-based reverse mode in Julia . . . 198

18 Neural networks 203
18.1 From a "neuron" to a network . . . 203
18.1.1 The perceptron . . . 204
18.1.2 Multi-layer perceptrons . . . 204
18.1.3 Training and gradient descent . . . 204
18.2 Case study: learning a function . . . 205
18.3 Case study: classifying data . . . 205
18.3.1 MNIST data . . . 205
18.3.2 Output and loss functions . . . 206
18.3.3 Overfitting, validation, and testing . . . 207
18.4 Engineering . . . 208

19 Applications: force inference and phase classification 209
19.1 Classification . . . 209
19.1.1 Identifying phase transitions . . . 209
19.2 Inference . . . 209
19.2.1 Inferring force laws . . . 209

Bibliography 210

Preface

This is a set of lecture notes prepared for PHYS 436: Advanced Computational Physics (Emory University, Fall 2025). It is more verbose than what I will actually cover in class, but also not a comprehensive textbook. I am sure there are both typos and errors in this document, as well as more general areas for improvement. Please email any comments or corrections to:

daniel.m.sussman@emory.edu

Course information

In the syllabus I said something faintly ridiculous:

> Computational physics is both deeply rooted in the historical foundations of modern physics, and is simultaneously at the cutting edge of current research. It's a powerful lens through which to view the universe, and it delivers a set of tools that can tackle problems once deemed intractable.

That's all well and good, but what will this class actually be about? On the one hand, it is is structured around a handful of modules, each of which loosely corresponds to a different category of numerical methods and the typical physical questions that we can ask with those methods. This will be a somewhat high-level overview; entire textbooks have been written not only about each of these modules, but often on each *lecture* within each module.

So what are we *really* trying to do? A first course in computational physics is often about building a toolbox: you learn fundamental algorithms for finding roots of polynomials, or solving ODEs, or working with data, and so on. While we will certainly cover more advanced algorithms, our focus will shift from the specific tools to the more subjective art of how to use them well. You already know about `if` statements and `for` loops and functions, so in an important sense you already know everything you need to write arbitrarily complex code. We'll use the modules as arenas to ask harder questions: How can we build a computational project that doesn't collapse under its own weight? How do we write code that is not just correct for one problem, but robust, reproducible, and reusable for a whole class of problems? What patterns of thinking allow us to solve complex physical challenges elegantly?

In a sense, this class will be as much about developing an attitude towards computational physics as the specific methods. That's why this course will begin with a pair of "foundations" modules, covering both a programming language and language-agnostic topics like version control, software design, and algorithmic complexity. The modules that follow will share some common themes and focus on similar systems viewed in different ways. But they will also be

an opportunity for us to practice these more fundamental skills. We will also be thinking hard about code as a form of technical writing – while it is tempting to think about code as a series of instructions to be handed to a computer, it is really just another tool we are using to try to solve a problem. And if we are to convince others that we have solved a problem, we should apply the same ideas of clear communication to this tool[1].

A note on writing our own code

Throughout these notes, we'll be adopting a largely "first-principles" approach. While we might use a few packages that define convenient data structures, we'll be writing our own implementations for nearly every main algorithm we encounter – our own ODE solvers, our own Monte Carlo samplers, etc. We will try to write clean, efficient, modular code; our primary goal, though, will always be pedagogical. We are writing code to learn.

This raises an important question: is this first-principles approach how you should tackle problems in your own research? *No!* In a real research project, you should almost never write (e.g.) your own ODE solver from scratch – you should look for a high-quality existing implementation. The scientific community has collectively invested thousands of hours into building robust, performant, and reliable tools; reinventing the wheel is not merely inefficient, but it is also a strategy for introducing subtle bugs and errors into your work.

How do we identify "high-quality" implementations? And why are we spending our time building tools we won't (and/or shouldn't) use? The answers come down to an attitude of not treating existing libraries as black boxes. To be able to both assess a tool and use it effectively we need to be able to understand how it works, what its assumptions are, and where it might break or fail itself.

Thus, one of the most vital skills this course aims to cultivate is our ability to know *how to trust code*, and the intuitions that go along with that ability. How can we be confident that a library, a snippet found online, or a function generated by an LLM is actually correct? To be able to verify such code, we need to develop certain skills and habits. We need to be able to **test against known cases** – does the code reproduce analytical solutions or have the correct asymptotic properties when such things are known? We need to be able to **reason about invariants** – does the code correctly preserve physical symmetries and other properties we know are generic features of the problem beyond just testing specific examples? We also need to be able to **reason about behavior** – does the output make physical sense, and what qualitative signatures of the code "working correctly" can we robustly rely on?

In the age of large language modes, these skills are more critical than ever: an LLM can generate complex functions in seconds – these functions can work well, or look plausible and yet be catastrophically wrong.

So, as we build the various tools up from scratch in this course, remember that the goal is not really the tools themselves but rather the skills and intuitions we're forging by building them. We are learning various algorithms and approaches, but we are more importantly learning to be discerning and critical scientists. The world is full of code we didn't write – trust, perhaps, but definitely verify.

[1] "...programs must be written for people to read, and only incidentally for machines to execute." - Abelson and Sussman [1] (A different, much smarter Sussman)

A note on our choice of programming language

You might be wondering why we're choosing Julia as our programming language for the journey ahead.

On the surface – and this could be a reasonable justification on its own – Julia stands out as an excellent language for the kinds of problems we'll be tackling this semester. It's a modern, high-performance language designed with scientific and numerical computation in mind. It is a dynamically typed "scripting language" in which simple, expressive code can be written very quickly and with minimal boilerplate – perhaps even more so than Python, Julia often lets you almost directly transcribe mathematical expressions from a textbook into your code. At the same time, Julia's type system and just-in-time compilation model enable it to produce extremely fast code – often competitive with the kinds of bare-metal speed typically associated with languages like C. In combination: its expressive syntax, features like multiple dispatch, and strong ecosystem of shared numerical packages make it a compelling choice for scientific researchers. I expect that this largely captures the flavor of the answer to "Why Julia?" you anticipated. On the other hand: there are many languages that I could have written a similarly plausible paragraph about while highlighting different strengths. Julia might be more friendly to beginning scientists than many languages, but it would just be one of several excellent choices we could have made.

Thus, there's a second, more pedagogical motivation. One of the core goals of this course is not just to teach you how to *code*, but to think more fundamentally about writing *programs* that translate ideas into computational reality. Coding – where you type-type-type away as arcane symbols materialize on your screen – is an important skill (albeit one whose role is evolving rapidly as LLMs grow increasingly powerful). Programming, though, is the art and craft of weaving together algorithms and data structures to solve problems. When working solely within one's first programming language, there's a common tendency to conflate the general challenge of creating a program with the specific challenge of creating a program *within that language's particular syntax and constraints.*

We often grasp the underlying rules of a system more profoundly not when we directly study it but when we encounter – and can contrast with – a different but related one. For example, I gained a much deeper understanding of the grammar of English only after I started learning a second language. I learned to identify and abstract out concepts – "noun," "past perfect tense," "imperfect aspect," "finite complement," and so on – that I had been using for years but that didn't have a label or category for. By choosing a language that I anticipate most of you haven't encountered extensively before, I aim to provide that "second language experience," but in the realm of programming. Hopefully seeing familiar concepts in the context of Julia will help crystallize your understanding of programming's universal building blocks, independent of any particular language's syntax.

Beyond these considerations of language features and the theory of learning, there is a final reason for my choice of Julia. After taking this journey, I'm convinced that Julia's core features align extremely well with many of the core lessons I want you to take away from this course. When it comes to scientific practice, Julia's built-in package manager and testing framework make it natural to take dependency management and reproducibility seriously. On the architectural side, Julia's multiple dispatch paradigm naturally guides us towards the design principles that this course aims to teach: the language itself nudges you, gently, towards writing

modular, extensible, and composable code.

My hope is that by learning to "think in Julia," you will gain more than just mastery of the course material. I hope you'll gain a new perspective on what it means to craft software – a skill that will serve you well in any programming language you use in your future career. I wish I had learned many of these lessons and patterns of thought far earlier in my own career, and I'm excited to share this perspective with you!

Sources

Much of the intellectual content of these notes is obviously not original to me. Throughout I will cite and link to relevant literature and textbooks; I would like to highlight the following as particularly strong general sources I have drawn from or been inspired by:

1. Moore and Mertens, *The nature of computation* ([2]); a fantastic book on the theory of computation. An excellent bridge to the subject even for folks that don't have a formal CS background.

2. Krauth, *Statistical mechanics: algorithms and computations* ([3]); an strong introduction to computational approaches in (no surprise here) statistical physics.

3. Frenkel, *Simulations: the dark side* ([4]); an article that serves as a reminder of just how much implicit / tribal knowledge there is in computational science. This article was an important motivator for me to write down a lot of the mundane, practical tips that you'll find throughout these notes. Of course, Frenkel and Smit's textbook is also excellent [5].

4. Gezerlis, *Numerical methods in physics with Python* [6]; a book that covers many core numerical methods very nicely, and with motivating examples from a broad range of physical systems.

5. Novak, *Numerical methods for scientific computing: The definitive manual for geeks* [7]; a nice, Julia-centric introduction to numerical analysis, linear algebra, and differential equations. Practical and readable.

6. Gilpin's *Computational Physics* class; a "very broad survey of computational methods that are particularly relevant to modern physics research." This course (under active development) has a lot of nice materials.

7. Clark and Wagner's iteration of Illinois' "Introduction to modern computational physics"; I saw the title of the first section ("N ways to measure π") and immediately closed my browser. The title of that section seemed like a great idea, and obviously inspired the intro to Julia that I wrote. To this day I wonder both how large N is and what methods they picked!

Visual elements in these notes

Throughout these notes you'll see blocks of text with different styles. Text that is meant to represent typing at the command prompt (along with the results of entering those commands) will look like this:

```
$ ls -la
total 8
drwxr-xr-x 2 daniel daniel 4096 May 21 09:42 ./
drwxr-xr-x 5 daniel daniel 4096 May 21 09:42 ../
```

Interactions with the Julia REPL will look like this:

```
julia> x=1
1
```

When I want to indicate blocks of code (either actual code or pseudo-code), it will have syntax highlighting and look like this:

```
# sampleCodeblock.jl
function f(x)
    println("You have got to be kidding me -- ",x,"!?")
    return acos(x) + 17
end
```

I will make occasional comments, sometimes out of the flow of the text; e.g.:

Comment!

I find fiddling with aesthetic choices soothing, but I should probably spend more time writing. Also, I'm not completely sold on the current choices[a]. Good thing LaTeX makes separating form from content (relatively) easy!

[a] I personally code in dark themes chosen based on my whims, but a light theme is much better for readability in a PDF. Lacking a background in graphic design or some other domain that would help me choose, I've gone with a best-guess at what will be clean, pleasant, and readable for most people.

Occasional questions to stop and ponder will appear like this:

Question!

Do you like these aesthetic choices? Which ones would you have made?

If I feel like I particularly need to call your attention to something, I will try to do so like this:

Attention!

"De la forme naît l'idée" – attributed to Flaubert in the Goncourt Journal. I will try to reserve these boxes for things that are... more relevant.

Finally: as you've already seen, these notes will make liberal use of footnotes. I like them[2].

Fonts and colors

In case you're curious: These notes were typeset using STIX Two for the main text and mathematics. Code and other monospace elements use JetBrains Mono, with all of the ligatures disabled and scaled in size to match the main text. I have, thus, spent a great deal of fiddling and fussing only to end up with two very standard, sensible fonts.

Visual elements in these notes are based on several open source color schemes:

- The primary color palette (questions, comments, notes, and code blocks) are derived from the "Kanagawa" theme by Tommaso Laurenzi (MIT License).
- Julia REPL colors use the "gruvbox-material" theme by Sainnhe Park (MIT License).
- The representation of the command prompt uses the "modus-operandi-tinted" theme by Protesilaos Stavrou (GPL-3.0).

[2]Many authors will instead invoke the famous Noël Coward quote, "Having to read footnotes resembles having to go downstairs to answer the door while in the midst of making love." They and Sir Coward presumably... read books more intensely than I.

Module 0

Module 0: Hello, π! Julia as a second (programming) language

Read the manual!

While I aim to cover many essentials for getting you up and running with Julia – and I hope you find this module engaging – this guide is not intended to be a substitute for reading the language documentation (which is excellent). My hope is that if you've already worked with, e.g., Python or C++ this guide will help you get familiar with Julia faster, but there are *many* topics and corners of the language I won't touch on here.

This course assumes that you have already taken an introductory course in computational modeling and have some experience with programming concepts. In this course we'll be working with the Julia programming language; I suspect many of you have not used it before[3], and so this module aims to walk you through the basics – the syntax, its common patterns, and so on.

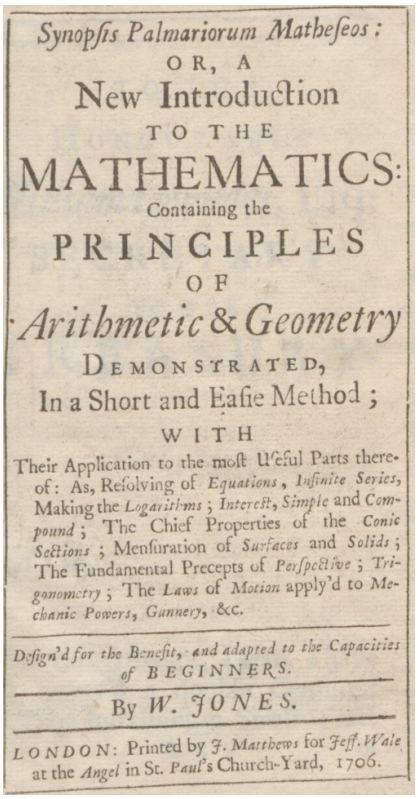
Synopſis Palmariorum Matheſeos:
OR, A
New Introduction
TO THE
MATHEMATICS:
Containing the
PRINCIPLES
OF
Arithmetic & Geometry
DEMONSTRATED,
In a Short and Eaſie Method;
WITH
Their Application to the moſt Uſeful Parts thereof: As, Reſolving of Equations, Infinite Series, Making the Logarithms; Intereſt, Simple and Compound; The Chief Properties of the Conic Sections; Menſuration of Surfaces and Solids; The Fundamental Precepts of Perſpective; Trigonometry; The Laws of Motion apply'd to Mechanic Powers, Gunnery, &c.
Deſign'd for the Benefit, and adapted to the Capacities of BEGINNERS.
By W. JONES.
LONDON: Printed by J. Matthews for Jeff. Wale at the Angel in St. Paul's Church-Yard, 1706.

Figure 0.1: A page from the first book to use the symbol π with its modern meaning [8]. "Deſign'd for the Benefit, and adapted to the Capacities of BEGINNERS"!

Writing a program that displays "Hello, World!" is a traditional starting point when learning to program (or when learning the differences between a language you already know and a new one). Given the context of this class (and Julia's increasing popularity in the scientific computing community) I thought it would be more fun to do something a little bit more mathematical. Thus, in this module we'll be cooking up increasingly elaborate ways to output the digits of π as we learn the language we'll use this semester. Fun fact: π was first used to represent the ratio of the circumference to diameter of a circle in 1706 by William Jones[4]. Earlier the symbol was used by William Oughtred to refer to the circumference of whatever circle was being considered at the time [9]. Presumably π was chosen because it is the first letter in the Greek word for "perimeter" (or "periphery"). Its modern use as a constant was introduced by Jones and popularized by Euler. Euler, amusingly, seems to have used the symbol to refer to both the constant 3.14... and the constant 6.28... over the course of his life [10] – a wrinkle in the Pi vs. Tau debate!

The structure of each of the following chapters will largely follow the same pattern: initial sections introduce important concepts, the penultimate section will apply what we've just learned to calculate or approximate π in some way, and then the final section will offer broader reflections on a topic in computational research or programming.

For a deeper dive into the Julia programming language, I

[3] See the preface for the whole spiel.

[4] William Jones' son – also named William Jones – was one of the first to suggest the existence of a common ancestor language for Sanskrit, Latin, Greek, and other languages. We now call this the Proto-Indo-European language (PIE)!

suggest you look at the resources suggested on the julialang webpage; some recent textbooks are also nice resources: [11, 12, 7]

Chapter 1

Setting up Julia

Arguably the most basic method of finding the digits of π is to... ask someone who already knows the answer[5].

1.1 Installing Julia

The recommended way to install Julia on your computer is by installing the "`juliaup`" binary, which in turn installs the latest stable version of Julia and can be used to keep it up to date. Follow the linked instructions for your specific operating system; on Linux, for example, it's as simple as running this command at the prompt:

```
$ curl -fsSL https://install.julialang.org | sh
```

Installation on Windows and macOS is similarly straightforward. One can opt instead to download specific versions of Julia, but the "`juliaup`" method should work seamlessly. As a comment: Julia is an evolving language; these notes reflect version 1.12, but minor differences may arise in newer versions.

1.2 The REPL

One of the main ways of interacting with Julia is in an interactive session. The REPL ("read-eval-print loop") is an environment in which the computer waits for input, executes commands once input is received, potentially displays some output, and then waits for more input. The REPL is what starts when you run Julia from the command line, and it is a great way to experiment with the language. After you start Julia you'll be greeted with a "`julia>`" prompt. There you can define variables, manipulate functions, and generally work with arbitrary code. Here's what I did when I opened Julia for the first time:

[5]This, in fact, is reflected in William Jones' book: "[the] Diameter is to the Periphery, as 1.000, &c. to 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803 48253421170679, True to above a 100 Places; as Computed by the Accurate and Ready Pen of the Truly Ingenious Mr. John Machin" (Ref. [8], page 243).

```
julia> 1+1
2

julia> exit()
```

Powerful stuff.

The REPL also has a useful "help" mode, which you can access by pressing the "`?`" key at the beginning of the "`julia>`" prompt (which will change to "`help?>`"). Once there, you can get help on functions, variables, types, or other Julia objects by typing their name and hitting enter. If you know the name of the thing you want the documentation for, you would type (e.g.) `?exp`. You can also find all instances in which a given string appears in the *documentation* of a functions (or types, or...). For instance:

```
help?> "square root"
Base.abs2
Base.isapprox
Base.issqrt
Base.sqrt
```

You could then use the help feature for each of these functions to figure out which one you actually want to use.

1.2.1 Adding packages

Julia comes with a built-in package manager which can be used to install various modular components that you might want to use – we'll learn more about this in Chapter 5. You enter the package-management mode of the REPL by pressing the "`]`" key at the beginning of an empty "`julia>`" prompt. The prompt will change to "`(v1.x) pkg>`" (where `x` reflects the version of Julia installed).

To start off, let's install a few common packages that will be nice to always have available as we write code. Enter package mode and type

```
(@v1.x) pkg> add Revise BenchmarkTools OhMyREPL
```

"`Revise`" is a package that will make working with stand-alone files from the REPL easier, "`BenchmarkTools`" will help with analyzing code performance, and "`OhMyREPL`" adds convenient syntax highlighting to the REPL (and lets you tinker with color schemes, if you enjoy that sort of thing). Just adding these via the package manager does not automatically bring their capabilities into your current session. To do so, you need to tell Julia you want their functionality, for instance like so:

```
julia> using Revise
```

The packages we just added were installed into your *default global environment*. Julia, however, makes it easy to specify different local (or even temporary) environments. This allows for fine-grained control over which versions of which packages are used for different projects. This ability is especially important for ensuring the reproducibility of how your code executes — you should be able to hand someone else your code, and its exact set of dependencies, and expect that they will get numerically the same result that you did! The principle of reproducibility is a cornerstone of reliable computational science, and Julia's tooling is designed to support it robustly. We'll learn more about this in Section 5.1.

1.2.2 Configuring the REPL

We've already seen that packages are not automatically used in your session, but what if there are packages that you really do want to use all of the time? Every time you start Julia it checks for a file named "`startup.jl`" in a root configuration directory[6]. This file is executed every time you start the REPL, which means you can use it to customize your default environment (always loading certain packages that you've already installed, or setting a preferred colorscheme, or...). For instance, if you always wanted to have some of the packages we installed just above active every time you start the REPL, you could have code block 1.1 as your startup file.

```
# startup.jl
if isinteractive()
    using Revise
    using BenchmarkTools
    using OhMyREPL
end
```

Code block 1.1: A simple startup file for Julia.

The first line is just a comment labeling the file – not important for Julia, but I'll often use this kind of convention when I want to indicate that a code snippet is part of a particular file. The `if isinteractive() ... end` block is a conditional statement: the code inside this block (here, just using the different packages) only executes if Julia is running in an interactive mode, such as when you launch the REPL directly.

1.3 Hello, π! (Method 1: Asking a friend)

With all of that... let's finally go ahead and ask Julia for the value of π – it turns out that it's a built-in constant of the language! In the REPL, just type "`pi`", hit enter, and there you go: if you didn't know it before, $\pi = 3.1415926535897\ldots$! Interestingly, Julia can not only work natively

[6] By default, this will be in `C:\Users\USERNAME\.julia\config\startup.jl` on Windows or `/Users/USERNAME/.julia/config/startup.jl` on Linux or Mac

with unicode input (so that you can write lines in your files that really look exactly like the mathematical equations you want to implement!), but the REPL will tab-complete many LaTeX commands into their corresponding glyph. Thus: you can also type "`\pi`", hit tab (and watch a "π" show up on your screen), and then hit enter. In this case, Julia knows that `pi` and π refer to the same numerical constant.

Finally, Julia has output formatting options for when you want to print combinations of strings and numbers to the screen – if you've used "`print`" in Python or "`printf`" in C you'll be familiar with the syntax:

```
julia> using Printf

julia> @printf("Hello, pi!\npi=%.40f",pi)
pi=3.1415926535897931159979634685441851615906
```

The "`Printf`" module is part of Julia's standard library, so no separate installation is needed. The "`@printf`" function[7] works like the C function of the same name; the result is that we see a bunch of digits of pi.

Question: floating point π?

In the example above, we used `%.40f`, which converts the corresponding argument to a *floating point number* (the "`f`") and prints at a *precision* specifying the number of digits to appear after the decimal place (the "`40`"). But standard floating point numbers do not have arbitrary precision – they use a fixed number of bits to represent numbers, so they can only be so precise! Assuming that the function is converting Julia's representation of π to a standard double-precision representation (i.e., a 64-bit base-2 format with 1 bit for the sign, 11 bits for the exponent, and 52 bits for the significand), how many of the displayed digits do you expect to be correct[a] before the rest are just numerical noise?

[a] How can you get more precision if you need it? Julia has special types like `BigFloat` that implement multiple-precision arithmetic. The flexibility to represent numbers at arbitrary levels of precision comes at the cost of the speed and memory efficiency of fixed-size floating point numbers; learning when to make such trade-offs is a fundamental skill in scientific computing!

1.4 Notebooks and IDEs

Using the REPL can be an extremely powerful way to quickly iterate on ideas. I'm particularly accustomed to two workflows when it comes to coding up something more permanent: coding in an interactive notebook environment, or working in an IDE. For the former, it turns out that Julia has dedicated "Pluto" notebooks, which are particularly good for writing *reproducible* notebooks. I won't be using Pluto notebooks in this course, but it turns that the "Ju" in Jupyter

[7] Actually a macro, a special Julia feature that lets you transform code before it is run

is a nod to the Julia language (along with Python and R) – if you're already familiar with working in Jupyter notebooks you might find this a convenient onramp. Conveniently, Google Colab includes a Julia kernel: just go to the "`Runtime`" menu at the top and select "`Change runtime type`." As of this writing there is a small difference in what version of Julia the Colab runs compared to the most up-to-date version from installing Julia locally, but for the purposes of this class that shouldn't matter.

Personally I prefer developing and writing code in an editor rather than a notebook (this is probably just a matter of taste). If you do want to use an editor-based workflow, the VS Code IDE is widely recommended in the Julia community. Regardless of whether you're working with a full IDE or a simpler text editor, one thing you can do is have Julia process a text file *as if* you were entering each command, from top to bottom, into the REPL. To see this: create a new file, perhaps "`helloPi.jl`" in some directory, and use your favorite text editor to make the contents of that file match code block 1.2.

```
# helloPi.jl
using Printf
@printf("Hello, pi!\npi=%.40f", pi)
```

Code block 1.2: Perhaps the simplest way to find π I could think of.

If you then go to the command line and run this command:

```
$ julia ./helloPi.jl
```

you should see a familiar result. While running scripts from the command line like this is sometimes convenient, when developing and exploring code *this pattern is discouraged*, because every time you launch Julia (including when you start it just to run a script like this) there is a relatively long startup time. That makes working with this workflow – editing a text file and periodically launching it from the command line – feel slow.

Much better is to keep the REPL open and run the same script by including it:

```
julia> include("HelloPi.jl")
```

This executes the contents of the file, and if you make changes to the file you can simply "`include("HelloPi.jl")`" again to execute commands again or update the definitions of functions you've defined in that file. An even more convenient "keep the REPL up-to-date with the contents of my file" is provided by the "Revise.jl" package that we installed earlier. It provides an "include with tracking" command, so that if you change any function definitions in the file, then the function in the REPL can access immediately gets updated. It's not much use for the code we've written so far, but the pattern is just:

```
julia> includet("HelloPi.jl")
```

That is, "`include()`" gets replaced with "`includet()`" – we'll get radically more use out of a similar workflow throughout the course. Embracing this interactive workflow not only speeds up development cycles but becomes a superpower enabling a remarkably fluid and exploratory approach to problem-solving in Julia.

Chapter 2

Variables, primitive types, and functions

"Calculating" π by retrieving a predefined value from memory is, arguably, not that satisfying. Let's push a little bit farther as we start to learn about Julia's type system and how to build functions.

2.1 Variables and types

Most programming languages are either statically or dynamically typed – *type systems* are the rules that assign properties to the different allowed constructs in the language ("is variable x an integer? a string?), and a program can be checked for consistency when it is compiled (static) or when it is run (dynamic). Julia is a dynamically typed language, and dynamic typing is fantastic (among other things) for quickly prototyping software: since everything only needs to be correct at the moment the code is running you can change your mind about what you want variables to *be* and how you want to connect them. This means that in Julia you can easily write code a la Python. In a statically typed language, you would have to declare that, for instance, "x" is an integer. Later on you had a change of heart and want it to be a floating point number? Too bad: re-write your code and recompile everything.

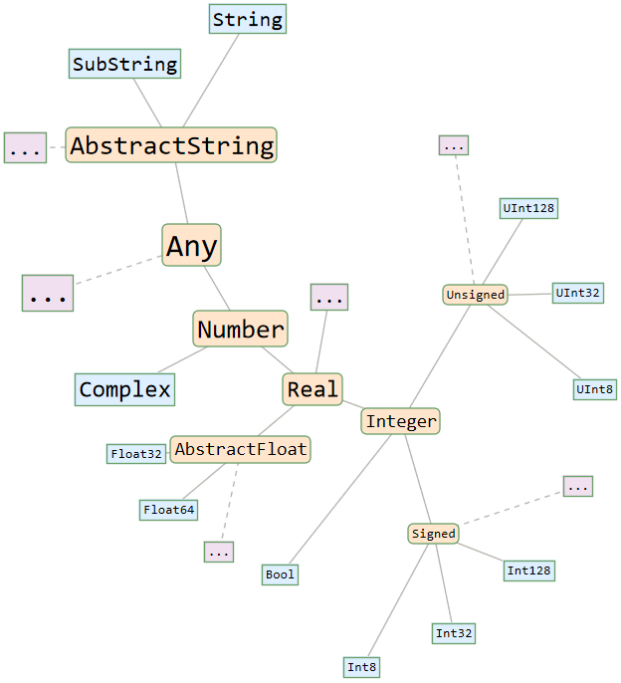

Figure 2.1: A small subset of Julia's type tree. Abstract types are in orange, and concrete types are in blue. A potentially large number of nodes are implied by the ellipses in purple.

Julia's type system is a vast tree (a tiny portion of which is in Fig. 2.1) with a special type called "`Any`" as the root. This means that `Any` is a supertype of every other type in the language, including those predefined by Julia and any additional ones you might define yourself. By default, variables in Julia don't have a fixed type

– they are simply names that can be bound to any value. That flexibility is what allows this nonsense:

```
julia> x=1
1

julia> x="one"
"one"
```

Julia organizes its type system by distinguishing between *abstract* and *concrete* types. Abstract types (like `Number`) exist as nodes on the type tree, but you cannot create a value of an abstract type. Concrete types (like `Int64` or `String`) are leaf nodes of the type tree, and are what the compiler can actually create values for. This means that while a variable name can be rebound, at any given moment it will always refer to a value with a specific, concrete type:

```
julia> x = 1; typeof(x)
Int64

julia> x = "one"; typeof(x)
String
```

To emphasize a core principle of the language: in Julia *variables* do not have types, only *values* do. Variables are just names that get associated with values. (As a few side notes: the `typeof()` function lets us inspect a value's type, and you can explore the type tree further with the `subtypes()` and `supertypes()` functions. Also, notice that you can use the semi-colon `;` in the REPL to run multiple commands on one line and to suppress output.)

One of the interesting features of Julia, though, is that even though it has a dynamic type system you can use *type annotations* in a few different ways. Type annotations make use of the `::` operator – this operator plays a few different roles that we'll meet over the next few chapters. One of those roles is its use in *variable type declarations*, which are a promise that we will only ever associate values of a certain type to a variable (and also converting the RHS of an assignment to the right type when we do so). For instance, we could write

```
julia> a::Float64 = 1 + 3;
```

Here, although "`1 + 3`" would be an "`Int64`" in Julia, it is (implicitly) converting that value to a floating point number (from 4 to 4.0). This `Float64` value is then bound to the variable `a`, with the annotation promising that `a` will consistently hold this specific floating point type. Making promises like this is a powerful tool for ensuring correctness and clarity in your code. If we try to make a promise that cannot be fulfilled, Julia will throw an error:

```
julia> b::Int64 = 1.3;
ERROR: InexactError: Int64(1.3)
```

We'll learn progressively more about types in Julia in Chapter 3 and Section 5.4, but for now we'll focus on using Julia's built-in *primitive types*, which are types whose content has a direct representation with a fixed number of bits. Julia defines a very standard set of primitive types (signed and unsigned integers, floating point numbers, boolean values, characters, etc).

2.2 Functions and control flow

2.2.1 Operators and special functions

Julia of course comes with a standard set of arithmetic operators. They work nicely and as you would hope, with automatic conversion of values when combining values of different types. As a non-numerical example: when working with strings Julia defines the "times" operator (`*`) as the thing that does string concatenation[8], which means we can do

```
julia> x="Hello"; x*x
HelloHello

julia> x^4
HelloHelloHelloHello

julia> x^2.2
ERROR: MethodError: no method matching ^(::String, ::Float64) The
function `^` exists, but no method is defined for this combination of
argument types.
```

It also comes with a great set of standard mathematical functions (powers, logs, trig functions, and so on). This includes inverse trig functions predefined, so we can already compute π: all we need to do is

```
julia> 2*acos(0)
3.141592653589793
```

While great when actually writing code, somehow I doubt you'll be satisfied with this method of "calculating" π.

[8] Perhaps you expected this to be the role played by "plus" operator, but the Julia documentation notes that when both addition and multiplication are defined, and if one of them is not commutative, then multiplication is usually the noncommutative one. The kind of amusing tidbit most likely to appeal to some mathematicians, I suppose, but regardless: it's just a convention.

Active reading and... AbstractMatrix?

This is good time to mention that I expect, like with most lecture notes, you are reading this *actively*! Have you used the REPL help mode to confirm that, e.g., a UInt8 is exactly what you expect it to be? Did you find yourself surprised at the type annotation on the argument to the built-in `acos` function?

2.2.2 Writing functions

Writing functions is a core part of writing Julia code, and Julia has a few different ways we can write them. For extremely simple one-liners you can use an abbreviated notation that is exactly like writing a mathematical formation:

```
julia> f(x) = 2*acos(x)
```

One can annotate this function with type information, but again: that's something we'll have more to say about in Section 5.5 – for now, we'll just pinkie-promise that we won't try to pass a string to this function.

To do something more interesting than directly evaluating a special function, Let's consider a famous[9] formula used to compute the digits of π, which is due to John Machin[10]:

$$\pi = 16 \arctan \frac{1}{5} - 4 \arctan \frac{1}{239}$$

In the rest of this section we'll build up to an approximation of this expression (which, again, we could directly evaluate since Julia has inverse trig functions!). I'll be using a "Revise"-based workflow: in the REPL I've used the "`includet()`" function on the following file:

```
# MachinFunction.jl
function f(x)
    return 2*acos(x)
end
```

This behaves exactly like the one-liner above, but now I can edit the file as I go and have the REPL keep up-to-date with the current version of the function.

2.2.3 Function arguments

Let's first just type out Machin's formula, but in a way which helps illustrate something about how Julia's function arguments work:

[9] "Famous"

[10] The same Mr. John Machin we met in Footnote 5

```
# MachinFunction.jl
function f(x)
    # How does x behave in this context?
    x = 16*atan(1/5) - 4*atan(1/239)
    return x
end
```

In this particular function[11] the input argument is not even used in the calculation; instead, whatever `x` is inside the function is immediately assigned to the result of some trigonometric calculation. Clearly the return value of this function will be π, but what happens to the `x` that was passed to the function?

Julia's function arguments are passed by "sharing." When you pass a value to a function, the argument names inside the function (for instance, `x` in `f(x)`) become new local names. These new local names *initially* refer to the exact same values or objects that were passed in from the calling scope, and no copy of the underlying data is automatically made just by passing it to a function. What happens once inside the function depends on what you do with these local names. First, you can always *rebind the local name*. That is, you can assign a new value or object to a local argument name. For instance, if we pass `x` to a function, inside the function we could write `x = 100` or `x = "hello"`. This rebinds the local name `x` within the function's scope to point at this new data. Such a reassignment of the local name itself *never* affects any variable in the scope that called the function: the original variable outside the function will still refer to its original value.

Second, Julia divides the world into *immutable* types (for instance: Ints and Floats) and *mutable* types (for instance: Arrays, which we will meet more properly in Chapter 3). If a local name refers to a mutable object, you can change the internal state of that object. Thus, e.g., if the Array `v` is passed as an argument to a function, `v[1]=100` modifies one of the elements of the array. In this case, the local name in the function and a variable name outside the function refer to the same underlying mutable object, mutations *inside* the function affect the variable in the scope that called the function.

Let's see this, using `f(x)` from the most recent version of `MachinFunction.jl` above:

```
julia> x=0; f(x)
3.1415926535897936

julia> x
0

julia> x=f(x); x
3.1415926535897936
```

Clearly, rebinding the local argument name in the function doesn't affect the caller's variable.

[11] Which also illustrates something about the return type of some of Julia's basic operators. Note that in Julia something like "`1/5`" – the division of two integers – returns a floating point number. Other type conversions are also happening in this expression to make sure we end up with the correct value of π.

Mutating elements and binding names

Hold onto this thought about mutability and variable binding. After we've learned about Arrays revisit this example, making sure you explore what happens when you mutate an element of an array that you pass to a function *and* what happens when you rebind the whole array to a local variable and mutate *that new* array!

All of this behavior is directly linked to how Julia manages the scope of variables. Functions in Julia always introduce a new local scope, and writing functions that don't use information that isn't passed to them is a great way to save yourself from several headaches down the road. We'll dive deeper into Julia's scoping rules in Section 5.2.

2.3 Hello, π! (Method 2: Computing functions)

Let's do a little bit more work on our own to calculate π – rather than use the special function, let's make use of the Madhava[12] series expansion for the arc tangent,

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

2.3.1 Our first loop!

One way we could do this is to introduce a basic for loop with some control flow:

```
# MachinFunction.jl
function approximate_atan(x, number_of_terms)
    if number_of_terms <= 0
        error("number_of_terms must be positive")
    end
    result = 0.
    for i in 0:number_of_terms-1
        result += (-1)^i*x^(2*i+1) / (2*i+1)
    end
    return result
end
function machin(x)
    result = 16*approximate_atan(1/5,x) - 4*approximate_atan(1/239,x)
    return result
end
```

We've got an if statement ensuring that we're summing a positive number of terms. We're also using "`0:numberOfTerms-1`" to create an iterable collection – which, as you probably suspect

[12] Madhava founded the Kerala school of mathematics, which is credited (among other things!) with discovering infinite series expansions for trigonometric functions. The Yuktibhasa, written around 1530, describes these results more than a century before the work of Gregory, Leibniz, Newton, and Taylor.

from the name, is a collection (like a set, or a dictionary, or many other things) that Julia knows how to iterate over. We'll learn more about collections in Section 3.1, but for now we'll just use the above as a way of writing a `for` loop.

We can check both that the series expansion is working, and that the Machin formula converges faster:

```
julia> pi - 4*atanSeries(1,10)
0.09975303466038987

julia> pi - machin(10)
8.881784197001252e-16
```

2.4 Expressiveness in code

I imagine that many of you are quite comfortable with this style of writing a loop – perhaps the syntax is different from other languages you've used, but the basic idea of explicitly stepping through each trip through the loop, sprinkling in some "`if-then`" control flow, and steadily building up an answer is probably pretty familiar. On the other hand: there's a sense in which the above loop is *explicit* in detailing the mechanics of the computation but not *expressive* of the overall intent[13]. Our goal for the loop was to perform a computation for each integer in a specified range and then sum the results. We can infer that goal by tracing the logic in our code (especially code as simple as the above) by stepping through the loop, but Julia makes it easy to be more expressive by treating functions as "first-class citizens" of the language.

That means that you can assign functions to variables, you can store them in a data structure, you can pass a function as an argument to another function, you can have a function be returned from a different function. Thus, creating a vector of functions and iterates over them as in code block 2.1... works.

This enables a powerful functional programming paradigm within Julia, and many of the most common higher-order functions – like map (which applies a function to every element of a collection, filter (which selects elements based on a condition), fold and reduce (which reduce the elements of an array to a single result by repeatedly applying an operation to combine the elements), and others – are built into the Base module of the language.

Do I care if you write in a functional style[14] or if you write raw[15] loops? Nope. I think of programming as composition. We articulate solutions by weaving together data structures and algorithms to accomplish some goal, and we have many choices to make as we translate a mathematical concept or a physical system into code. We might opt to write your code in an imperative style, with a sequence of explicit operations that change the program's state. We

[13] This distinction is also commonly made by contrasting an *imperative* style that specifies *how* something should be done and a *declarative* style that focuses on *what* should be done.

[14] Especially if you are not familiar with functional programming, you might enjoy this nice article from Mary Rose Cook emphasizing how one might translate the same code between imperative and functional styles of the same code.

[15] If you've written a lot of verbose C++ code you might enjoy Sean Parent's "C++ Seasoning."

```
# basicFunctional.jl
function square(x)
    return x^2
end
function cube(x)
    return x^3
end

operations = [square, cube, x -> x + 1] # A vector of functions
# x -> x + 1 is the notation for defining an "anonymous function"
for op in operations
    println(op(1.5))
end
# This would output:
# 2.25
# 3.375
# 2.5
```

Code block 2.1: Functions as first-class citizens in Julia.

might lean into an object-oriented approach to dividing up the state of your program and what pieces of it are responsible for acting on different components of that state. We might instead prefer a more declarative style. We could, for instance, write the loop above as what it is: the summation resulting from applying the same function to a set of numbers. That might look more like the following, in which we define a small function that calculates each term and then use Julia's `sum` function to sum those terms over the relevant range:

```
julia> atan_series_term(x,i) = (-1)^i*x^(2*i+1) / (2*i+1);

julia> approximate_atan(x,n) = sum(i->atan_series_term(x,i),0:n-1);
```

I really don't believe any of these approaches are inherently "better" than the others. Imperative loops are great for complex control flow, and declarative approaches are often superior for common, established patterns. Thus, part of our task in programming is not to adhere to a dogmatic preference, but to understand the different ways we can encode the logic of what we want to accomplish. Different styles might resonate more clearly or feel more natural depending on the specific problem, on your own background, or even on the conventions of the team working together to try to create something. The key is to be aware of these varied patterns, to choose intentionally, and to strive for code that is both robust in its function and clear in its purpose.

Correctness of our code?

Above we've written a few functions – and in this case they are simple enough that we can convince ourselves that they are probably correct just by looking at them. But how do we *know* everything is correct? We could run it in the REPL and check that the output looks reasonable, but "looks good to me" is hardly a rigorous standard for scientific code! A better approach is to write automated *unit tests* that check the behavior of functions we write against known results. Julia has an excellent, built-in testing framework as part of its standard library. We'll be using it all semester as part of the assignments for this course, and learn more about it in Section 5.3.

Chapter 3

Composite types and data structures

In this chapter we'll continue to explore Julia's data structures and its powerful type system. We'll cover how Julia manages data structures designed to hold multiple values, and how Julia performs iterations over such structures. Along the way, we'll target a much older, geometric route to approximating π.

3.1 Collections

In Julia a *collection* is a general term for a data structure which groups multiple values together. A collection might be *homogeneous* – holding values of all the same type – or not; it might be *mutable* – capable of having its values altered after the collection is created – or not; it might be *indexable* – in which values can be identified by pointing to their position in the collection – or not; and it might be *associative* – in which values are associated not with a position in the collection but by some more general kind of key – or not. More specialized collections can support even more properties, for instance enforcing a specific ordering of elements or enforcing the uniqueness of values the collection contains. Below we'll look at some of the most common collections, thinking about how they combine these defining characteristics.

3.1.1 Tuples

Tuples are *immutable* and *indexable* collections that can contain heterogeneous elements. You would typically use a tuple when you have a small group of related items that won't change (such as an `(x,y,z)` coordinate of a fixed object), or when you want to return a group of multiple values from a function. They are declared using a syntax that looks like the argument list to a function – indeed, the idea of a tuple is an abstraction of an argument list – with parenthesis enclosing the tuple and values separated by commas:

```
julia> a = (1,1.0,"wow");
```

Tuples can have any number of values of different types, and are accessed by indexing. For

the sake of avoiding confusion / potential bugs, I don't think I can emphasize the following [16] enough:

Don't forget!

Julia is a 1-indexed language!

That is, you access the element of the tuple above (or a vector, or, indeed, any indexable collection) by starting with element 1 rather than element 0:

```
julia> println(a[1]," ",a[2]," ", a[3])
1 1.0 wow
```

I think there's no need to get into arguments about what indexing style is better[17]: they correspond to different mental models of the underlying data. Indexing starting at 1 maps nicely onto counting and indexing in mathematical expressions; indexing starting at 0 maps nicely onto offsets in memory in which the data is stored. Different languages have different conventions, and unlike Python and C++, Julia indexes starting at 1; it might take some getting used to if you've spent a lot of time with the alternative.

Functions in Julia can take both normal arguments and also *keyword* arguments; the example from the manual is a function whose definition begins

```
function plot(x, y; style="solid", width=1, color="black")
```

The idea that a tuple is an abstraction of the arguments of a function means that we should expect that Julia also has the notion of a *named* tuple. These can have their values accessed either by index or by name, as in the following:

```
julia> b = (first=1,second="two");

julia> println(b[1]," ",b.second)
1 two
```

3.1.2 Arrays, Vectors, and Matrices

Arrays are *mutable* and *indexable* collections that contain homogeneous values. Arrays are a workhorse of computational physics – they can store lists of positions of particles evolving in time, or the values of a grid representing evolving density and velocity fields, to say nothing of the many applications of Arrays in the context of applying linear algebra to physical problems. They can be constructed with square brackets and commas, like so:

[16] Don't worry: I appreciate the irony of having this module appear as "Module 0" in the table of contents.
[17] For a contrasting view, note that people much smarter than me have much stronger opinions.

```
julia> a=[10,100,1000];

julia> typeof(a)
Vector{Int64} (alias for Array{Int64, 1})
```

We already learn a few things: Arrays can be not only single-dimensional but also used to represent collections that can be indexed on a multidimensional grid (including a grid of dimension zero), and in Julia a "Vector" is just an alias for an existing type: a one-dimensional array. While thinking about Arrays as homogeneous, it's important to remember that that *does not* mean that all of its values must be of the same primitive type!

A homogeneous collection... but of what?

What is the type of the `[1,pi]` Array? What about the `[1,pi,"pi"]` Array?

Julia is flexible, and if you initialize an array with mixed types, it will determine a suitable shared supertype (which might be `Any`) to hold them. This has important performance implications, but can also sometimes be very convenient.

Just as `Vector` is an alias for a one-dimensional `Array`, in Julia a `Matrix` is just a two-dimensional array. Matrices can be constructed in a variety of ways, and an array can be of any dimensionality and size can be constructed with all zeros (or all ones) by using built-in functions. For instance, a two-dimensional Array with two rows and four columns could be made by writing

```
julia> zeros(Float64, (2,4))
2×4 Matrix{Float64}:
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
```

Since Arrays are mutable, one could now populate the elements of this Array however you wanted. Julia also has a nice syntax by which arguments separated by semicolons (or newlines) imply "vertical concatenation" and spaces (or double semicolons) imply "horizontal concatenation" – we won't focus on this now, but as always: the language documentation will be your friend if you want to quickly construct matrices or higher-dimensional arrays quickly and easily using this syntax.

Efficiently working with matrices

Different programming languages lay out matrices (and higher-dimensional arrays) in memory differently, choosing either row-major or column-major formats. Julia is column-major. In practice, that means that if you are iterating through the elements of a multi-dimensional array, your inner-most loop (the index which changes "most rapidly") should correspond to the left-most index.

3.1.3 Functions on Arrays

Julia comes with many built-in functions for working with Arrays and other collections. For instance, if you have a Vector but want a sorted version of it you could simply:

```
julia> a=[4,1,3,2]; b=sort(a);
```

Our original array is unsorted[18], and we've created a new array which holds the sorted version. We could also call a completely different function that, rather than returning a copy of the sorted array *mutates* the array we pass in:

```
julia> a=[4,1,3,2]; sort!(a);
```

A rich set of functions is available for array manipulation. These include operations for querying their size, appending new values to them, sorting them, getting the index of particular values or filtering by arbitrary conditions, or getting "views" (efficient subsections of arrays that don't involve copying data). When working with Arrays, ask whether some of these standard algorithms might be ready and available to do the task for you!

Idiomatic naming convention

The above functions demonstrate a convention found throughout Julia and that you should adhere to: if you write a function that alters the values of mutable arguments passed to it, put an exclamation mark at the end of the function's name!

Julia also has a convenient options for operating on Arrays. Standard arithmetic operators like `+` or `*` often have specific mathematical meanings when applied to arrays as a whole, and these will work as expected in Julia (i.e., standard matrix multiplication can be written as `A*B`). One can also easily specify that an operation should be performed on *all* elements of an array using a "dot" syntax (i.e., putting a dot before the operator):

```
julia> a = [1 2 3]; a .+ 1
1×3 MatrixInt64:
2 3 4
```

In fact, this "vectorized" syntax can be used not just for the standard arithmetic and comparison operators, but with *any* function in Julia! Thus:

[18] Well, I suppose every integer sequence is sorted according to *some* function, but it's certainly unsorted with respect to "hey, I just want this sorted normally!"

```
julia> g(x) = cos(x)+14; g.(a)
1×3 MatrixFloat64:
14.5403 13.5839 13.01
```

3.1.4 Dictionaries, Sets, and the rest

A bit more briefly, a Dict is a *mutable* and *associative* collection that maps keys of a consistent type `K` to values of a consistent type `V`. It is the data structure to use when you need to store and look up values based on a unique identifier (a *key*) – like a word and its definition, or a user ID and their profile information – rather than by an index. They are best when you need fast access to the data associated with specific labels, and do not necessarily care about the order in which the key/value pairs are stored. They can be created and accessed like this:

```
julia> indexing_style = Dict("C++"=>0,"Python"=>0,"Scheme"=>0);

julia> indexing_style["Scheme"]
0
```

You can get a collection of keys or values in a Dict by calling the appropriate function. Since Dicts are mutable, we can (for instance), add new key/value pairs to them by direct assignment (the idiomatic approach) or by using a function we've already encountered:

```
julia> indexing_style["Julia"]=1;

julia> push!(indexing_style,"Smalltalk"=>1);
```

A Set acts like a set, serving as a collection of unique values. They are the collection of choice if you just need to store a collection of unique items, and if all you want is to know if items are present in the set (or, of course, if you want to perform standard union/intersection kinds of operations you expect to be able to do).

```
julia> s = Set("Daniel Sussman")
SetChar with 11 elements:
```

Notice, by the way, what this example teaches us about Strings: in Julia, strings are just a kind of collection of characters. Thus, while they primarily represent text, a String behaves like an ordered, immutable collection of characters – you can check its length, access parts of it, and iterate over it just like a collection. Speaking of...

3.2 Iteration and Loops

Another key feature of collections – so important that it warrants its own section, however brief! – is that they are *iterable*. While we gave an example of a simple for-loop earlier, here we'll explore some of the primary ways of building loops in Julia. Perhaps the most basic is a standard "while" loop:

```
function test_while_loop()
    i = 1
    while i < 10
        if i % 2 == 1
            i += 1
            continue
        end
        println(i)
        if i >= 7
            break
        end
        i += 1
    end
    return i
end
```

This has a lot of features that should be familiar: a loop that continues until some expression evaluates to false, the `continue` statement to advance to the next iteration, and the `break` statement to exit the loop early.

I rarely directly use while loops[19], but iterating through the same basic set of operations on well laid out data happens all the time. A fundamental version of this is a for loop that iterates once per value in a collection. It can be convenient to either have direct access to the *n*th value in the collection, or to the index associated with that value, and these are two idiomatic ways to iterate through an indexable collection:

```
function test_for_loops()
    A = [1,2,3,4,5,6,7,8,9,10]
    for a in A
        # do something!
    end
    for i in eachindex(A)
        # do something with i
        # (including, of course, accessing A[i])
    end
end
```

If you want to iterate through an associative collection, say `A`, you can also use the `keys(A)` and `values(A)` functions, which return iterators over the keys and values (respectively – I bet you can guess which is which) of the collection.

[19] I Don't trust 'em! Probably because I've used them incorrectly too many times...

3.2.1 Ranges

What about when you want to iterate through some sort of sequence of numbers, but you don't feel like you really need to create an object which holds all of those values? For instance, doesn't it seem silly to create an array of the integers from one to ten just to have a loop that executes ten times? *Ranges* are an iterable way of representing such a sequence, and they are very memory efficient: they don't need to store the values in the sequence, just the rules needed to generate them. There are a number of ways of constructing Ranges – including for representing sequences that are either linearly or logarithmically spaced – but the most explicit is to call `range` with three keyword arguments (any three out of "start," "stop," "length," and "step"). There are various assumed defaults depending on which three keywords you use. You can also use a colon to denote a (start):(stop) range (in steps of one), or a (start):(step):(stop) range. Among other things[20] ranges can be used to write simple for loops, such as this one which iterates from one to five:

```
julia> for i in 1:5
           println(i+1)
       end
```

3.2.2 Collecting and Comprehending

Given a collection or an iterator, Julia's `collect` function will return an Array containing all of its items. This is, for instance, one helpful way of quickly constructing Arrays. For instance:

```
julia> a = collect(1:0.25:1.5)
3-element Vector{Float64}
1.0
1.25
1.5
```

An even more powerful and general way to construct Arrays is to use the *comprehensions*:

```
julia> a = [ f(x) for x in xIterable];
```

In this example, an array will be generated whose elements correspond to the application of some function `f` to each value in the `xIterable` – this can be anything that can be iterated over, and in practice will most typically be a collection like range. As implied by the variable name, there is a similar syntax for using comprehensions to build multidimensional arrays.

[20] Notably, they are also fundamental for getting slices of an Array

3.3 Structs and constructors

More general than the collections discussed above is a *Composite type*. These can be any group of named fields (each of which may or may not be annotated as being a particular type, with the default being `Any`), and which taken as a whole can be treated as a single value. User-defined composite types are defined by using the `struct` keyword, like so:

```
julia> struct Particle
           mass::Float64
           charge::Float64
           funny_name::String
       end
```

Best practices

Annotate the fields of your structs with concrete types whenever possible! When Julia knows the concrete type of every field, it can lay out the data in memory efficiently and predictably.

By default structs are immutable, and the default way of constructing them is by calling its type name as a function, providing arguments for each field in the order they are defined. Fields are then accessed by name:

```
julia> a = Particle(1836.152673426,1.0,"proton")

julia> a.charge
1.0
```

Structs can be made mutable just by using the `mutable` keyword in their definition:

```
julia> mutable struct ParticlePosition
           x::Float64
           y::Float64
       end
```

Naturally, you can write functions that take instances of your custom structs as arguments. We'll touch on this more when we discuss multiple dispatch in Section 5.5, but you can also *extend* existing methods to let them operate on your custom composite types. Julia also makes it easy to define alternate ways of creating instances of your structs: while Julia provides a default constructor that accepts arguments for each field in order, you can also add convenient *outer*[21] constructors. These are simply functions (often using the same name as the struct) that return a new instance of the struct in question.

[21] Yes, there are also inner constructors

As an example of some of these ideas, and in preparation for the geometric approximation to π we're about to do, let's define a "PolygonVertex" as being a location in a two-dimensional space. For later convenience, I'll define a constructor that takes an angle and returns a PolygonVertex at that angle relative to the x-axis and on the unit circle, and a function that defines the norm of a PolygonVertex to be its distance from the origin. Finally, with some hesitation[22], we'll extend the binary subtraction operator.

```
# polygonPerimeter.jl
struct PolygonVertex
    x::Float64
    y::Float64
end

PolygonVertex(theta) = PolygonVertex(cos(theta), sin(theta))

function norm(a::PolygonVertex)
    return sqrt(a.x*a.x + a.y*a.y)
end
import Base: - # explicitly import to add a method
function -(a::PolygonVertex,b::PolygonVertex)
    return PolygonVertex(a.x-b.x, a.y-b.y)
end
```

Annotating function arguements

The rule of thumb is to *write generic functions* and avoid annotating arguments with types. Annotations are **not** for performance – Julia's compiler will automatically create fast, specialized code based on the actual input types you use.
Instead, type annotations for function arguments can serve two primary goals. First, they can be *for correctness* – they can enforce a contract and ensure that an argument has a required property that the functions logic depends on or structure (e.g., specific fields, as in the `norm` function above). Second, *for multiple dispatch* – we'll see in Chapter 5 that we can define different methods of the same function that provide unique behavior based on the input types.

3.4 Hello, π! (Method 3: Using geometry)

Instead of relying on modern[23] calculus or pre-computed forward and inverse trigonometric functions, let's leap backwards in time to consider Archimedes' elegantly geometric approach to calculating π. Among his many remarkable achievements was his use of the method of exhaustion. By carefully calculating the perimeters of regular polygons either inscribed within

[22] Is the difference of two vertices, which presumably I'm about to interpret as a vector, really another PolygonVertex? It certainly shares exactly the same fields of the same types, but...

[23] Well, to the extent that the 17th and 18th century counts as modern!

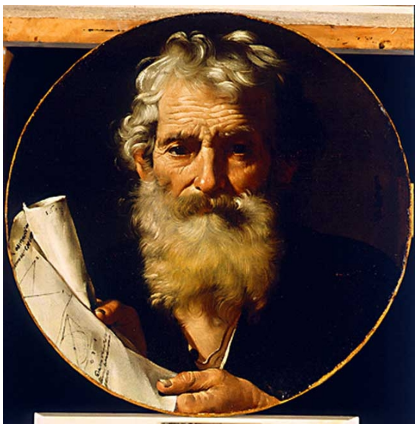

Figure 3.1: A photograph of Archimedes, or perhaps a self-portrait by Jusepe de Ribera; funny how it's hard to tell sometimes. " Ritratto di matematico (Archimede?)" Image in the public domain.

or circumscribed around a circle, he was able to bound the value of π by considering sequences of polygons with increasing numbers of sides. By considering polygons of up to 96 sides he came up with his famous bound: $\frac{223}{71} < \pi < 22/7$. An accuracy of three digits – not too bad for around 250 BC! A fourth digit wouldn't be recorded for another 400 years (although that fourth digit may have actually been obtained earlier)!

Rather than apply the full method of exhaustion to find bounds for the value of π, let's just take the simpler approach of looking at the perimeter of inscribed regular polygons as the number of sides gets large. Making use of our PolygonVertex as defined above, we can easily write a pair of relevant functions. The first will be a straightforward loop over the vertices in a polygon – our function will assume that a "polygon" is some iterable collection of PolygonVertex values – and calculates the total perimeter by summing up the distance between consecutive vertices. We will call this with a second function which does the work of constructing a regular polygon inscribed in the unit circle (making use of the range and comprehension techniques we just learned) calls the `perimeter` function, and returns half of the result:

```
function calculate_perimeter(polygon)
    result = zero(polygon[1].x);
    last_point = last(polygon)
    for current_point in polygon
        distance = current_point - last_point
        result += norm(distance)
        last_point = current_point
    end
    return result
end
```

We can now call this function from the REPL and confirm that as we increase the argument of the `archimedes` function we get an increasingly accurate estimate for π! Here, again, one might pause to recall our discussion in Section 2.4: the `calculatePerimeter` function employs a common looping pattern that involves managing state from the previous iteration (the `lastPoint`). Could we, perhaps, be more expressive in our code by recognizing that our loop in the `perimeter` function can be written using a standard algorithm? Perhaps:

```
function perimeter_by_circshift(polygon)
    #That's a rotate!
    rotated_polygon = circshift(polygon, 1)
    displacements = polygon .- rotated_polygon
    return mapreduce(norm, +, displacements)
end
```

One could compress this even further, writing the `mapreduce` with the help of an anonymous function that handles the vertex-loop-logic for us:

```
function perimeter_by_mapreduce(p)
    len = length(p)
    return mapreduce(i-> norm(p[i] - p[mod1(i-1, len)]), +, 1:len)
end
```

The goal, I need to emphasize, is *not* to get really good at writing elaborate, code-golf-style one-liners[24]. As before, we should think hard about what version of a function simultaneously optimizes our goal of writing robust, clear, and expressive code.

3.4.1 A geometric solution

The persnickety reader will complain that our calculation above still relied on having built-in trigonometric functions available to us. And that, after all, is still basically cheating. Fortunately, Archimedes was quite clever, and his construction did not actually involve working out the locations of the vertices of inscribed polygons. Instead, he came up with an elegant geometric argument: by drawing the correct triangles, he showed that if you start with an inscribed polygon with N sides and side length d_N, the inscribed polygon of $2N$ sides will have side length

$$d_{2N} = \left(2 - 2\sqrt{1 - \frac{d_N^2}{4}}\right)^{1/2}.$$

So, starting out with a square inscribed in the unit circle (each of whose sides is clearly $\sqrt{2}$), if you are good at calculating roots you can get the perimeter of polygons with 4, 8, 16, ... sides. Here's an implementation that starts with the hexagon (as Archimedes did):

```
# iterative Archimedes method
function iterative_archimedes(doublings)
    n_sides::BigInt = 6
    side_length_squared::BigFloat = 1.0

    for i in 1:doublings
        n_sides *= 2
        side_length_squared = 2-2sqrt(1-side_length_squared/4)
    end
    return n_sides*sqrt(side_length_squared)/2
end
```

The Chinese mathematician Zu Chongzhi obtained a bound for π that was accurate to seven digits in the 5th century – this stood as the record level of precision for hundreds of years. His original calculations are lost, but later authors suggested that he may have used an independently discovered version of Archimedes' method (computing areas rather than perimeters).

[24] A two-liner here, but only for reasons of line length.

If this was indeed his approach[25], obtaining the first six digits of π after the decimal would have required starting with a hexagon and using 12 doubling steps (ending with a 24576-gon!). Much later, Ludolph van Ceulen spent a large amount of his life basically using Archimedes' method, culminating in a 35-digit estimate of π in 1593. This "Ludolphian number" would have involved calculating the properties of polygons with 2^{62} sides!

3.5 Naming conventions and commenting code

For truly short collections of functions it barely matters what style you write in, or how you choose your variable and functions names, etc – anyone familiar with the language would be able to glance at the code to see what it does. As you write more complex programs, clear communication of the program's intent becomes increasingly important. This is not just being clear with the compiler about what you want to do, but also being clear with your collaborators[26]. Clear communication of this sort occurs by different means.

For instance, you should try to write "self-documenting" code by choosing descriptive names for your variables, functions, and types. Good names – descriptive nouns for types and variables, active verbs for functions – significantly reduce the need for extra explanatory comments by making the code's purpose intuitive and easily readable. Consider the following two functions:

```
julia> flabbergast(moose1,moose2) = moose2/moose1;

julia> compute_acceleration(force,mass) = force/mass;
```

If you were working on a physics simulation and you were to encounter the first function (or, more realistically, a similar but more complicated example like it), you would likely be a bit flabbergasted, yourself. You would then have to work through the logic of what the function does, where it is called, and how the results are used. If, on the other hand, you encountered the second function in the same context, you would *immediately* have the correct mental model for what the function does, the context it is used in, and what kinds of arguments you should pass in. Importantly: the computer does not care which of these two functions you write – to the compiler they are *the same*! Make sure you are writing programs that can be read by humans, and trust the compiler to do the translation to the computer for you.

Julia has a style guide that includes established naming conventions that also help. We already saw an important one: functions that modify their arguments typically end with an exclamation mark (e.g., `sort!`). In Julia type names are usually written in upper camel case (like `PolygonVertex`), whereas function and variable names are usually written in all lowercase or with snake_case (like `calculateperimeter` or `element_type`). Adhering to these conventions makes your Julia code more accessible and idiomatic, and as I said above – the conventions of your team should be an important factor in determining the style of your code!

[25]The doubt basically being related to the existing book-keeping technology for recording and managing the intermediate calculations while tediously evaluating square roots.

[26]This set should be understood to include, most notably, *your future self*.

Please forgive the mild hypocrisy

In these notes sometimes I will occasionally use variables like `x` and `n` when something more descriptive would have been better. This is usually because I want to avoid typesetting individual lines of code across multiple lines of text[a].

[a] hopefully in not too many cases was it just laziness!

While good naming conventions certainly reduces the burden, comments still play a crucial role. Avoid cluttering your code with comments that merely restate what the code clearly does. Effective comments typically focus on the *why* of your code – this is especially true for detailing particularly intricate code logic, important assumptions, and non-obvious design decisions. In Julia, single-line comments (of which we've seen a few in the examples above) start with an octothorpe, `#`. Longer comments can be made by enclosing (multiple) lines of interest in the hash-equal combo: `#= ... =#`. Finally, as you define functions and types that are part of larger codebases, consider writing documentation. Documentation for functions and types can be made by enclosing text – which can be written in a flavor of markdown – in triple quotes. An example is shown in code block 3.1.

This documentation can be automatically processed into documentation and is invaluable for larger projects. You'll also note that if you write this kind of documentation for your functions, then in the REPL you can use help mode to get back this information – that's handy for your future self, and absolutely crucial for someone else who might be interested in using your work [27]! In that context, it is useful to typically include standard sections for your docstrings, like the `# Arguments` (to clarify inputs), `# Returns` (to specify outputs), and `# Examples` (to demonstrate usage and allow for automated testing) in the above.

[27] Sadly, you just *know* that whoever wrote that flabbergasting moose function did not write any documentation to go along with it.

```
"""
    perimeter_by_mapreduce(p::AbstractVector{PolygonVertex}) -> Float64

Calculates the perimeter of a `polygon` defined by an ordered vector
of `PolygonVertex` objects.

The perimeter is computed by summing the Euclidean distances between
consecutive pairs of vertices. This version uses `mapreduce` with an
anonymous function for a compact representation.

# Arguments
- `p`: An `AbstractVector` where each element is a `PolygonVertex`.
  The vertices are assumed to be ordered.

# Returns
- `Float64`: The total perimeter of the polygon.

# Examples
```julia-repl
julia> square = [PolygonVertex(0.0,0.0), PolygonVertex(1.0,0.0),
PolygonVertex(1.0,1.0), PolygonVertex(0.0,1.0)];

julia> perimeter_by_mapreduce(square)
4.0
```
"""
function perimeter_by_mapreduce(p)
    len = length(p)
    return mapreduce(i-> norm(p[i] - p[mod1(i-1, len)]), +, 1:len)
end
```

Code block 3.1: Documenting an arguably hard-to-parse function by specifying its behavior and giving examples of its use.

Chapter 4

Data, plots, and visualization

Perhaps you are still a bit worried about our calculation of π? We may have gotten away from using trig functions, but taking roots is still... well, it's doable, but we're still relying on a built-in mathematical function, and is that not also cheating? Maybe less so than using `acos`, but at least a little bit?

We've already met many of the most important aspects of the Julia language when it comes to writing scripts and simple functions using a variety of built-in and custom data structures, and in this section we're going to focus on how to handle data, and how to turn that data into plots and other visualizations. This probably feels a bit different in character from the focus of the previous sections, but make no mistake: plotting and visualizations are some of the most important aspects of computational research! They let us compress and efficiently communicate enormous amounts of information quickly, and are invaluable in testing hypothesis about the systems we're studying.

As we go, we'll estimate π by a simple *Monte Carlo* approach. This is the name for a broad class of algorithms that use repeated sampling of random numbers to obtain numerical estimates of different quantities[28], and we'll learn a lot more about these approaches later in this class (see Module IV)!

4.1 Reading and writing data

Data comes in many flavors, and it is non-trivial to write an overview of how it should be handled without knowing the specifics – is the data we're interested in reading and writing to a file a set of 2D or 3D points that will be used to make a plot? Is it several gigabytes of data in a recurring pattern (for instance, snapshots of a large number of simulated particle positions at different times)? Is it a massive atlas mapping voxels to cell types in a mouse brain? Is it a heterogeneous data set where for any particular entry some of the expected attributes are missing?

Eventually, depending on your projects, you will probably want to dig into packages designed to help with tabular data in easy-to-read formats (like CSV.jl), or provide more general

[28]The name for these methods was coined by physicist Nicholas Metropolis during World War II – apparently fellow physicist Stanislaw Ulam (who together with von Neumann pioneered the modern version of Monte Carlo methods) had an uncle often gambled at the Monte Carlo Casino in Monaco.

data science tools (like DataFrames.jl), or to deal with data in a variety of other common data formats. I'm not going to try to cover all of these cases here. Instead, I am going to emphasize the absolute basics built into the Base and Standard Library: opening files, reading and writing simple delimited files, and so on.

In preparation for what we're going to do later, let's write a few helper functions that for the time being we'll use to generate some random "data:"

```
generate_point(L) = rand(Float64,2) .*L .-L/2
scatter_points(n,box_side) = [generate_point(box_side) for i in 1:n]
```

The first function uses the `rand` function to generate a 2-element Vector of positions, using the broadcasting "dot" syntax to scale and shift the output so that each point lies in a square of side length L centered at the origin. The second just uses a comprehension to create an Array of such points of whatever size we want.

4.1.1 File input and output

Perhaps the most fundamental file operation is just writing to a file and then reading it back. As in many other languages, Julia treats file operations as interactions with an "I/O stream." The standard way to ensure that a file is correctly closed after interacting with it is to use an "open() block" syntax. It looks like this, opening the file from the current directory in write (`"w"`) mode:

```
julia> open("data.txt", "w") do io
           println(io,"Text")
           println(io,"Text and data:", 3.14)
           write(io, "writes binary representation")
       end
```

Calling the stream "`io`" above is just by convention. For simple text we can use the `println` function. The `write` function writes the raw byte representation of its second argument. For strings, this just means writing the text bytes without adding a newline, but it can be used to write other types as well. It's good to know that this exists, but you might instead explore the Serialization functions or use some of the packages mentioned above for handling arbitrary data if you need to.

Reading this data back is similar. We could open the file in read (`"r"`) mode and process it line by line:

```
julia> open("data.txt", "r") do io
           for line in eachline(io)
               print(line)
           end
       end
```

Or we could read the entire file into a single string:

```
julia> content = open("data.txt", "r") do io
           read(io,String)
       end
```

4.1.2 Reading and writing simple delimited data

Just with the read/write capabilities from the above, we *could* write functions that (a) take a multidimensional array of data and save it in something like a comma-separated format and then (b) load such files and carefully parse what we know the format to be to turn it back into data of the sort we saved. But we're not here to re-invent the wheel. For structured data – for instance, the output of our `scatterPoints` function, which returns a vector of 2-element vectors – the `DelimitedFiles` module in the standard library is very convenient. We first need to tell Julia that we want to use the module, but then saving our data is simple:

```
julia> using DelimitedFiles

julia> writedlm("scatterPoints.txt", scatterPoints(1000,2), ",")
```

If you're following along, you'll now find a file with 1000 rows, each of which contains two numbers separated by a comma.

Reading delimited data from a file is just as easy:

```
julia> dataRead = readdlm("scatterPoints.txt", ',');
```

This reads in the data as a *matrix* – Julia can't know here what exact data structure was used when you were saving the file, so if we wanted to wrangle it back into exactly the structure we saved it as we would have to do a bit more work. Notice that there is a slight asymmetry between these functions: `writedlm` allows a string as a delimiter (e.g., `","` or `"\t"` or `" banana "`), while `readdlm` expects a single character (e.g., `','` or `'\t'`).

Save your data before you plot!

A tip for your computational workflow: if generating data for a plot involves significant calculation, *save that data to a file first*. This decouples the data generation from your visualization of it. You can then quickly load the data and iterate on plot aesthetics without the frustration of re-running lengthy computations every time you or your collaborator makes a request like[a] "Perhaps that should be a dot-dashed line that is 20% thinner?" or "Can we just tweak the color scheme?"

[a] Surely not something *I've* ever said before, of course.

4.2 Visualizing data

Visualizing data is an important skill, and one could easily write books about the visual presentation of information. This section is not going to try to teach you how to make beautiful figures, or try to dictate best practices. Nor is it going to be a comprehensive guide to the many ways to make plots in Julia. It will focus on the basics: visualizing data and making simple but informative plots using one of the many options Julia presents to us.

4.2.1 Julia's plotting ecosystem

At first, Julia's ecosystem of plotting packages can be quite daunting – there isn't just a `plot` command you can pull off the shelf. Rather than having a built-in plotting library, there are numerous packages we can add. Many of these operate on a "frontend/backend" model. The frontend defines the syntax you use to make figures– what functions you call, what options you can specify, etc – and then passes that information to the backend. The backend is responsible for taking the information from the frontend and doing something with it – saving a plot to a file, or drawing it on screen, or creating an interactive window, etc. Some backends excel at creating high quality vector graphics; others might be specialized for creating embeddable components for a website; yet another might be best for rendering complex 3D scenes on the fly. When it comes to crafting extremely detailed figures this model is fantastic – it lets you pick exactly the right tool for the job.

When you're just starting out, though, you might feel beset by the paradox of choice. Should you use Plots.jl or Makie or Gadfly.jl or PGFPlotsX.jl or Gaston.jl or... It's a lot to choose from, especially before you have a lot of context and experience with which to help judge the pros and cons. In the spirit of this section I'm just going to put a lot of options in a list and use a random number generator to pick a plotting package to focus on[29]: Oh! It turns out we'll be using Makie! See this "beautiful Makie" site for a sample of cool things other people have made with this plotting package.

Makie offers a unified ecosystem – the same frontend API is used by all of the backends. Below we'll focus on two backends: `CairoMakie`, which is excellent for static 2D graphics, and `GLMakie`, which is excellent for interactive graphics and 3D plots and figures. The documentation and available tutorials are quite helpful, but I'll also try to highlight some of the basics just below. First to add these packages we can go to the package manager[30] in the REPL

```
(@v1.x) pkg> add CairoMakie GLMakie
```

These will take a bit of time to install, but we only have to do that once.

[29] Just kidding! Or rather, partially kidding – the RNG I used was not an unbiased one.

[30] I'm adding this to the default environment here – in general we want to keep the default environment as light as possible, so you might consider already setting up other environments. We'll learn more about this in Section 5.1

4.2.2 A simple scatter plot

As a first step, we're going to make a simple scatter plot of the points that we "threw down" just above. To see some of the options available to us, and to emphasize that it is straightforward to plot multiple datasets in the same figure, let's first define a function[31] which will let us determine which points are inside of and outside of the unit circle:

```
in_unit_circle(point) = sum(point .* point) < 1.
```

Next, we'll take our own medicine and load some of the data we saved above. There's probably a better way to do this, but let's be very explicit in wrangling our data into a form that Makie can easily work with. We'll use the `filter` function to make two sets of points – inside and outside the circle. Makie often works best with its own geometry types (like "`Point2f`" for 2D points), so we'll use a simple comprehension to make arrays of them.

```
using CairoMakie
interior = filter(in_unit_circle, eachrow(dataRead))
exterior = filter(!in_unit_circle, eachrow(dataRead))
#Convert to Makie coordinates
inpoints = [Point2f(p[1],p[2]) for p in interior]
outpoints = [Point2f(p[1],p[2]) for p in exterior]
```

Here we've used one of the many helpful functions that Julia has built-in ("`eachrow`"). While we could have used, e.g., array slices to achieve this, `eachrow` serves as a good reminder of the many functions of convenience available in Jullia. How to learn about them? As always: reading the (friendly) manual.

All of that was just to get an array of a type that Makie easily plots – if we didn't care about what the data was we could have just as well been making a data set like

```
julia> pointsToPlot = [Point2f(rand(),rand()) for i in 1:100];
```

How are we going to use this data structure to make a plot? Makie uses a hierarchical object system to create plots – this makes it extremely composable (i.e., it allows you to build extremely complex figures by composing together many simpler elements), but it might take some getting used to. The core of this hierarchy involves `Figure`, `Axis`, and `Plot`. A `Figure` is the top-level container for everything, handles overall layout, and holds some global attributes (for instance, the size or resolution of the overall figure). An `Axis`[32] defines a coordinate system that can map data values to positions within the Axis' boundaries. The Axis is also responsible for drawing decorations (axis and plot labels, tick marks, etc), and it acts as a container for `Plot` objects. A `Plot` is the visual representation of the data – the heatmap or the points or the lines – and as such it holds data-specific attributes like the plotmarkers to use or the color and thickness of lines to draw.

[31] This could also been written, e.g., as `sum(point.^2) < 1.0` or `point[1]^2 + point[2]^2 < 1.0` or...

[32] And some other types, like `Slider` or `Legend`, but Axis is the one we'll focus on.

Let's set up a simple version of this. First, we'll make a Figure with only the default attributes, and then create an Axis that will live inside the figure. We'll tell it that it lives in the first row and first column of the figure (default Figures are laid out in a grid), and give it a few attributes (including a "`DataAspect()`", which just means we want the figure to have the same aspect ratio as the ranges that the data covers).

```
fig = Figure()
ax = Axis(fig[1,1],title = "random points",
       xlabel = "X-axis", ylabel= "Y-axis",
       aspect = DataAspect())
```

We will then use the `scatter` function, one of the basic plotting functions, to make a Plot. Notice that we're using a function defined with the usual "exclamation marks indicate functions that mutate arguments" – in this case we're modifying the axis that lives inside the figure. We will do this twice, giving our two sets of points different colors:

```
#add interior and exterior points in different colors
scatter!(ax, inpoints, color = :darkorange)
scatter!(ax, outpoints, color = :steelblue)
```

We could stop here and ask Makie to display the figure (or save it with the `save("filename", fig)` command). Let's mutate the axis one more time and use the `poly!` function to also draw a thin circle, and *then* display the figure:

```
center = Point2f(0.,0.)
circle = Circle(center,1.0)
poly!(ax, circle, color = (:red, .1),
     strokecolor = :black, strokewidth = .5)
display(fig)
```

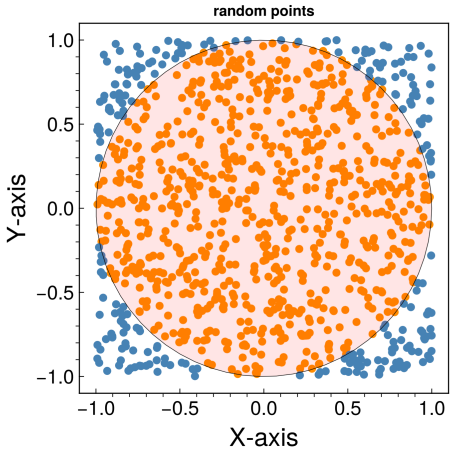

Figure 4.1: Points scattered in a square close to the origin.

The result is in Fig. 4.1; not the world's most amazing plot, but not the worst, either! Visually, the ratio of the number of orange points to the total number of points gives an intuitive suggestion of how we could estimate π using this kind of random deposition of points.

4.3 Hello, π! (Method 4: Using noise)

Let's take that visual suggestion and actually generate some estimates of π! Let's first write a quick function[33] that will take a set of points and determine the fraction of them that are inside the unit circle:

```
function unit_circle_proportion(points)
    # return count(in_unit_circle,points)/length(points)
    result = 0.0
    for p in points
        if in_unit_circle(p)
            result += 1.0
        end
    end
    return result/length(points)
end
```

Geometrically, it's clear that whatever fraction is returned should be one quarter of our estimate of π. Let's write a function that accepts two parameters – a number of points to throw down, and a number of trials to run – and uses some of the basic features of the `Statistics` library to estimate π.

```
using Statistics # part of the standard library
function estimate_pi(n,trials)
    data = [ 4 * unit_circle_proportion(scatter_points(n,2))
             for t in 1:trials ]
    return (mean(data),var(data))
end
```

Given this basic function, I called it a bunch of times for various values of n and the number of trials to average over (actually, I wrote a function that would do this for me, and I was extremely lazy and called it "`est`" – not very good naming on my part! – which returns a Vector of Vector of (Int64, Int64, Float64) tuples). Really I just wanted to demonstrate that we can easily make a 3D version of a scatter plot. To do so, though, we have to switch from `using CairoMakie` to `using GLMakie`. The syntax, though, is otherwise the same. Here is the plotting code, where I'm introducing just a few of the options we have to style our plots:

[33] You can see that this block includes both an explicit "loop over the elements and use a counter to determine the number" and a commented-out version that uses the `count` function. I'll stop belaboring the point that there are many ways to write all of these functions, and that we should take some time to consider why we're writing whatever version we choose.

```
using GLMakie
fig = Figure(fontsize = 24)
ax = Axis3(fig[1,1],xlabel="N",ylabel="trials",zlabel="estimate of
pi")
for set in est()
    scatter!(ax, set, markersize = 25)
end
```

The result is the most naive plot indicating our estimate of π, shown in Fig. 4.2. It looks terrible, but at least we can tell that π is probably some value between 3.0 and 3.4.

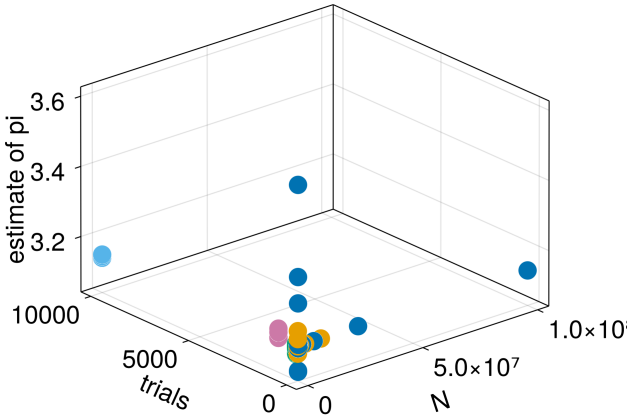

Figure 4.2: A 3D plot estimating π as a function of number of points thrown down in a square and the number of trials. This plot is not meant to look good (and it doesn't).

Plots as hypotheses

Plotting data is a key skill. When we are writing papers we can use plots to communicate our findings, compress huge amounts of information, and tell entire stories. During the research process itself, though, plots also serve a vital role in both the *exploration* and *understanding* of the system under study. I encourage you to think of making plots in this stage not just as a picture, but as *experiments that test hypotheses* about your system. Sometimes those hypotheses might be as simple as "Is this signal changing, or is it just noise?" or "I think *this* is the range over which some function varies in an interesting way." Often, however, we should aim for more. We should let our hypotheses guide our plotting choices: What *functional form* do I expect the data to take? Given that expectation, should we make our axes linearly scaled, or make them logarithmic? Would plotting a transformed version of the data be more revealing?

The real power of using these visualizations to explore data comes when you articulate your hypothesis *before* you generate the plot. Based on your understanding and how you plan to display a plot, what *should* it look like? If the plot matches your expectations, great – some part of your understanding gains credence. But if

it *surprises* you, that's often even better: it might point to an opportunity to learn something interesting!

Taking that comment to heart, we should be honest: Fig. 4.2 is a poor hypothesis. It has no thoughts about the range of the variables, or how the answer depends on them. It is purely exploratory. Sometimes this is okay, but we can usually do better. Figure 4.3 is a step in the right direction – far from perfect, but better.

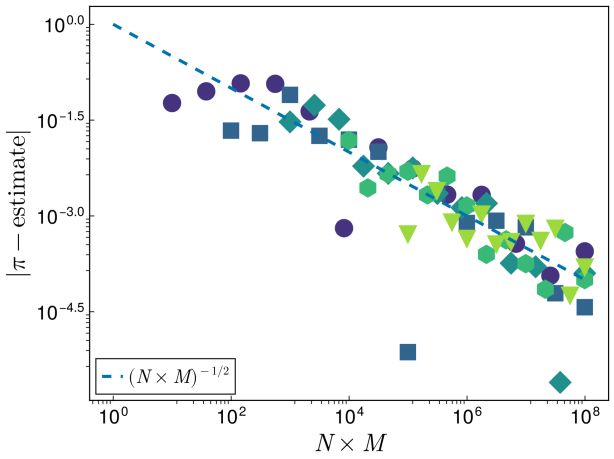

Figure 4.3: Difference between π and its simple Monte Carlo estimate vs the product of the number of points (N) and the number of trials averaged over (M). Notice, also, that Makie let's us easily customize the aesthetics of our plots. Please don't pick tick mark label fonts and axis label fonts that clash as much as they do here.

4.4 Performance and profiling

Alongside writing code that is robust (correct) and expressive (clear), we sometimes need our code to be *performant* (fast). This adds yet another dimension to what we might mean when we say we are trying to write "good" code. Context here is absolutely crucial: performance doesn't always matter, and even when it does only a small amount of code might actually need to be optimized. For one-off scripts, initialization routines for other numerically intensive tasks, or parts of your code that already run fast enough, prioritizing clarity over performance is often the better approach. When performance does matter – perhaps you expect your simulation to take weeks to run, or you know you have a core loop that will execute a billion times – though, follow the golden rule:

The golden rule of optimizing code

When optimizing code, don't guess. **Measure!**
Our intuition about where code spends its time is often wrong, and what is and isn't performant might even change from one year's version of the compiler to the next.

> Thus, if you are concerned with how fast your code is running, measure it. Only after you have identified hot-spots in your code should you dive in and think about spending time optimizing it.

Before we learn how to make those measurements, let's understand the core principle behind Julia's speed: *when all of the types of values used in computations are stable and predictable*, Julia can generate extremely performant code[34]. This principle is called "type stability," but how can we achieve it?

There are a few general principles we can adopt from the official documentation regarding how we write Julia code. Perhaps the most important is to *write code as a composition of functions*. Julia's compiler is able to perform optimally when it is able to determine the type of all values it needs to work in, and the compiler specializes and optimizes code *at function boundaries*. For instance, if we pass a global variable as an argument to some function, then at the moment the function is called the compiler can determine its type (for that specific call) and generate a specialized version of the function tailored to that type – this kind of specialization is a core part of how Julia achieves high performance. On the other hand, if we *directly use* a global variable from within a function without passing it as an argument, Julia's compiler must be extremely conservative – the type of that variable might change at any moment from an Int64 to a String to a Vector, and so the compiler needs to generate a version of the function that can handle Any value. This typically leads to slow code.

If you *do* need to use global variables from within a function, it usually makes sense (both in the logical structure of your code and for performance reasons) to declare them explicitly as constants. For instance, something like the following will let you define a global value and also let the compiler optimize functions that use it (the "screaming snake case" is just a convention to indicate a const):

```
julia> const FINE_STRUCTURE_CONSTANT = 0.0072923525643;
```

Functions should consistently return values of the same type[35], and within functions you should try not to change the type of variables.

Another important general practice we've already seen is to use type annotations to make sure that all fields in the definition of `struct` are concrete types. As mentioned earlier, this ensures that the composite type can be efficiently laid out in memory, and it also means that the compiler will definitively know the type of all fields within the structure.

4.4.1 Profiling

When you want to move beyond those general principles (and the other more specialized performance tips from the Julia documentation), Julia has excellent tools for actually measuring performance. The standard for easy and reliable benchmarking is the `BenchmarkTools.jl`

[34] Leading some to say that Julia "walks like Python and runs like C."

[35] More specifically, *methods* should return a consistent type for specific input types. We'll see what the distinction I'm making here means in Chapter 5.

package, which we set up when we first installed Julia. It provides convenient macros for measuring code: for instance you can prepend a "`@btime`" to a function call[36] in the REPL to get two important pieces of information: the mean time to execute that function and the number and amount of memory allocations on the heap (which take time to both allocate and deallocate, and can interrupt the flow of computation) that that function needed to make. A typical result might look something like:

```
julia> @btime estimatePi(100000,10);
88.812 ms (6000032 allocations: 236.51MiB)
```

These results come from running the same code multiple times to get stable results, and you can get the full distribution of the timing results by using instead the `@benchmark` macro.

To be honest, I don't really care about optimizing a function that takes a handful of milliseconds to execute and that I don't plan to call all that many times in my life. But if I did care, or if I was more serious about using this method to estimate π, I might think to myself that that seems like a large number of memory allocations. And indeed, the sequence of functions that we used was repeatedly allocating memory for the array of points in every trial, and also allocating memory for random pairs of points to fill those arrays. Without too much work we can improve things a little:

```
# Convention: the mutated argument goes first
function generate_point!(point, L)
    point[1] = rand(Float64) * L - L/2
    point[2] = rand(Float64) * L - L/2
end
# perform the same logic, but pre-allocating the arrays
function estimate_pi_in_place(n, trials)
    #pre-allocate arrays
    current_trial = [Vector{Float64}(undef, 2) for i in 1:n]
    data = Vector{Float64}(undef, trials)
    for i in 1:trials
        for j in 1:n
            generate_point!(current_trial[j], 2.0)
        end
        data[i] = 4.0 * unit_circle_proportion(current_trial)
    end
    return (mean(data), var(data))
end
```

This makes the function allocate less memory and run in about half of the time (on my laptop) compared to the `estimatePi` function we had earlier.

[36] We saw our first macro, `printf`, in Chapter 1, and here we see another one. Here the code transformation is probably even more clear: the macro is generating a bunch of code that wraps around the function we are calling, and that new code is both running the original function many times and also keeping track of timing information

In addition to these direct timing and allocation benchmarks, Julia has additional tools (like `Profile`, or the `JET.jl` package) that let you analyze your code's performance, look for hotspots, debug, look for type instabilities, etc. I'm sure, if we wanted, we could do even better than what we did above! But at that point, we should probably start asking ourselves some higher-level questions about what we are trying to achieve. Could a different Monte Carlo method converge to the answer faster, rather than throwing more computational power or code-optimization time at *this* method? Could a non-Monte Carlo method be *even* better[37]?

[37] You bet!

Chapter 5

Projects, parametric types, and multiple dispatch

Maybe you accepted "Oh, sure – let's just use a random number generator" or maybe you thought "Wait – 'Random' numbers on a computer?! Surely that's even blacker magic than just calling a trig function!" In this chapter we'll implement a version of calculating π that just involves counting the number of collisions in a simulation of a physical system.

Along the way we'll tackle a final set of important topics in Julia. It's remarkable that Julia gives us the efficiency it does while feeling like an easy-to-code scripting language in our examples above, but what really makes the language sing? Below we'll first learn about Julia's system for organizing projects, which uses environments to manage dependencies and modules to organize code. Given its importance in scientific computing, we will also discuss Julia's built-in set of tools for testing our code. We'll then discuss how Julia's type system facilitates powerful generic programming patterns, and how its implementation of multiple dispatch gives us tremendous power in writing well-organized yet flexible programs. Each of these topics could easily deserve their own chapter, but this will at least get us started.

5.1 Environments

Julia has a fantastic system of package management, easily allowing you to pull in powerful collections of code that people in the community have written. We used the REPL's package mode to add a small number of these packages to our default environment; perhaps (especially if you have not yet worked on multiple projects that invoked different dependencies) you found yourself wondering "Why not just always add packages to the default environment? Is it really worth bothering about multiple environments?"

Story time!

Let's imagine that at some point you decide that your Monte Carlo estimation of π could benefit from a higher-quality source of randomness than Julia's built-in `rand()` function. You find a promising package another researcher wrote: `SweetRNGSuite.jl`. You `]add SweetRNGSuite` to your default environment, which

installs v2.3 of the package, and everything works beautifully. You write up your findings and send a paper detailing your exploration of π to a journal – fame and fortune await!
While waiting for the referee reports to come back, you work on a different project that, at some point, uses some dynamic Hamiltonian Monte Carlo to perform non-linear fits to data and extract model parameters. Some time into your work you find a robust package, `FancyHMC.jl`, that will do this for you, and you add it to your default environment. Unbeknownst to you, this package also uses the `SweetRNGSuite` package – it requires the newly released v2.6 of that RNG suite, and the package manager updates `SweetRNGSuite` to this latest version.
Your HMC project is going well, although some odd things crop up every time you try to run your older π-estimate code: you still get results that are reasonable, but the actual numbers are no longer the same! (It turns out that the authors of the `SweetRNGSuite` package changed the behavior of their functions so that they default to using a random seed rather than a fixed seed[a]. Like many packages, it was using Semantic versioning, but sadly not everyone has the same definition of what constitutes a breaking change.) You're a little bit concerned – what if the referees are not able to reproduce your results with the code you made available? – but you try not to worry too much.
You then realize that you need to compute some numerical integrals – you definitely do not want to implement some of the sophisticated techniques to accurately compute integrals of complicated function, handle integrable singularites, etc – and find a robust, community-approved `VersatileIntegration.jl` package that implements *many* different approaches to evaluating definite integrals. You `]add VersatileIntegration` to your default environment, but – disaster! It turns out that the `VersatileIntegration` implements a Monte Carlo method for computing integrals – very useful for high-dimensional integration! – but the package *requires* a `SweetRNGSuite` version in the v1.x series. The `FancyHMC` package, on the other hand, relies on the behavior in the v2.x series of releases. The package manager *cannot* satisfy all of the constraints, and simply refuses to add the integration package. This is, arguably, better than installing it and having other things break, but it doesn't help the fact that you're stuck.
Welcome to dependency hell, newest resident: you!

[a] We'll learn much more about all of the subtleties of generating "random" numbers on a computer in Module IV!

Fortunately, Julia makes working with different environments extremely easy, and its package manager is really one of its strengths. The guiding principle is to keep your project dependencies isolated. To achieve this, the first thing we should do is keep the default environment as light as possible (i.e., adding only the bare minimum to it). General development tools that you use across all projects (like `Revise` and `BenchmarkTools`), especially those not directly called within your project code, are often conveniently placed in the default environment. You *might* also consider adding domain-specific packages, if they are really of the type that you plan to include in *everything* you do. Something like a plotting package is arguably another reasonable

choice, although these bring in so many indirect dependencies that you might start to worry a little bit (an alternative: make a dedicated "plotting" local environment! You can manage packages (add, remove, update) within any active environment, so even if you've already added such packages to the default environment you can go back and put them instead in a different local environment!).

Basically everything else, though, should be added to local environments as you go. The simplest way to do this is to launch Julia from the directory where you have some project you want to start working on[38] and type "`]activate .`". This will tell the package manager to set the current primary environment to the current directory – either creating a new environment there if it doesn't exist or loading information about one that does. (you can, of course, specify a different target by replacing the dot with a different path). You will see the package manager prompt change accordingly. Now if you add a package, say, `]add Symbolics` two things will happen. First, the package manager will get to work, installing a bunch of dependencies and pre-compiling various functions. Second, you will find two new files in the directory you started from: "Manifest.toml" and "Project.toml". The "Project" contains general information about the package and its direct (but not indirect) dependencies. The "Manifest" contains the exact versions of all packages you added and the packages indirectly installed as dependencies of those packages – this is a crucial tool in being able to reproduce *exact* behavior of your code.

As you might expect me to say by now, the documentation for the package manager is excellent, and you should look there for more details – this explanatory blog post is also excellent. Two quick things I want to point out, though: First, you can layer ("stack") environments on top of each other, and anything in a base layer will be available in an environment that sits on top of it. *This includes*, by default, the default environment, which is a base layer for any other environment you define. This is why you should keep the default environment clean, mostly just populating it with tools you use for development. Second, in addition to being trivial to create and activate, local environments are *cheap*. If you have multiple local environments that use the same versions of the same packages, the package manager won't install and maintain multiple identical versions. On the other hand, if different environments *need* different versions of the same package, everything gets taken care of for you.

5.2 Modules

Up to now we've been using a "Revise"-based workflow as we modified individual files and invoked the functions they defined from the REPL. As we write larger and larger projects, it makes sense to organize our code in a way that is more structured, more maintainable, and more easily shared with others. The primary patterns that Julia gives us, here, are to organize our work into modules and packages. A *module* acts as a namespace (and defines its own "global" scope – see below!), and can be used to organize code and prevent naming conflicts. Inside of a module you can define custom structs and functions and constants and not worry about naming conflicts with other people's code (an important consideration given what I know will be the temptation to call one of your functions "`f(x)`"). A *package* is a distributable collection of Julia code (which will play nicely with the package manager), and it usually

[38] Alternately, from the commandline you can start Julia with a specific environment by pointing to its path: `julia --project=pathToProject`

consists of one (or a few) module bundled together with some metadata about the package's version information, its dependencies, etc.

Declaring a module is as easy as wrapping whatever you want to in a `module MyModule ... end` block of code. When a module exists because you installed its associated package we've already seen that we can load it by doing something like

```
julia> using ModuleName
```

If you've written `MyModule` in a local file you can execute `include("MyModule.jl")` to make the module known in your current session and then use things it defines by accessing through the namespace. For instance:

```
julia> include("MyModule.jl")

julia> MyModule.amazing_function_defined_in_MyModule()
```

Note that `include()` evaluates the file contents in the current scope (here, the scope of the REPL), whereas `import`/`using` typically interact with Julia's package loading path[39]

A nice workflow to switch to once `includet("myfile.jl")` is insufficient involves defining modules within packages. The official recommendation is to use the PkgTemplates package to do this for you – it can handle relatively complicated scenarios (for instance, in which you want to set up a package with test coverage and documentation and GitHub hosting out of the box). For our purposes, though, let's learn about using the built-in package manager to set up a minimal local package for us to work with.

Event-driven molecular dynamics

In this section we're ultimately going to use – bizarrely enough – "event-driven molecular dynamics" (EDMD) [13] to estimate the value of π. EDMD is an alternative to standard molecular dynamics (something we'll spend much more time on in Module Module II) – it simulates a physical system by jumping forward in time from the instant of one collision to the instant of the next. It is a great representation of a kind of billiard-ball model of particles interacting with each other.
The basic idea is that in between collisions the particles experience no interactions and, hence, move at constant velocity. Given that, at any moment in the simulation you can calculate when the next collision will occur, advance the entire system forward in time to that moment, calculate what happens in the collision process, and then calculate when the next collision will occur.

In preparation for writing an EDMD simulation, let's set up a new package. Starting from an empty base directory, we launch Julia and execute the following two commands:

[39] The `]activate .` command, for instance, modifies this path for the active project, allowing Julia to locate its modules.

```
julia> import Pkg

julia> Pkg.generate("EventDrivenMolecularDynamics")
Generating project EventDrivenMolecularDynamics...
```

You will get the following[40] directory and file structure generated:

```
$ tree
./
└── EventDrivenMolecularDynamics/
    ├── Project.toml
    └── src/
        └── EventDrivenMolecularDynamics.jl
```

In this skeletal template we have a Project file comes pre-populated with some basic information, and a `.jl` file whose name matches the package name and which just defines a placeholder function. Here's what that file looks like (with some additional comments indicating what it will look like eventually).

```
# EventDrivenMolecularDynamics.jl
module EventDrivenMolecularDynamics

# we'll add export, using, and import statements here. E.g:
# using StaticArrays

greet() = print("Hello World!") # this line comes from the template
greet2() = print("Hello World2!") # this line comes from the template
# For small packages we can fit everything in this file.
# For larger packages, we'll add more files to the package and
# include them here. E.g.:
# include("elasticCollisions.jl")
# include("eventQueueHandler.jl")

end # module EventDrivenMolecularDynamics
```

Our workflow for building this package up from its humble beginnings to our eventual goal will be the following. We'll navigate to the root of this directory[41], start Julia, activate a local environment with "`]activate .`" and import[42] our module. The functions in the package are now available to us using the module's namespace, for instance:

[40] Mimicking the output of the `tree` Linux utility.

[41] i.e., `cd /path/to/EventDrivenMolecularDynamics`

[42] We could also use `using` instead of `import`. If we export-ed a list of names in our module then bring the package into our session with `using` would let us access names from the module without the namespace-dot syntax.

```
julia> import EventDrivenMolecularDynamics
```

After which we could do:

```
julia> EventDrivenMolecularDynamics.greet()
Hello world!
```

Since we configured `Revise` to be used automatically, we can start directly working in the REPL and separately on the files in our package simultaneously: changes in the files included by the module will be tracked and updated automatically, just like when we earlier used the `includet()` function.

The core workflow

In case you didn't quite catch that, here is the core workflow for a package:

1. Navigate to the directory of the project you're working on.
2. Start the Julia REPL, and start using `Revise`[a].
3. `] activate .`
4. `import NameOfYourPackage`
5. Run some functions in the REPL, figure out what you need to change or do.
6. Write or modify source code in your package.
7. Go back to step 5 and iterate until done.

[a] Or configure Julia to do that on startup

As a first step, let's add a dependency to our project. Perhaps for an underlying data type we'll use a "static array" – which we can use as a data structure containing a fixed, known number of elements. When we run `]add StaticArrays` we can see that the package management prompt correctly identifies the new local environment; after the package has been added we can see that the Project file has been updated and a new Manifest file has been created.

5.2.1 Scopes in Julia

We've been using "scope" a lot recently, so let's briefly talk about how scope works in Julia. In programming, the scope of a name (like the name of a variable, or of a function) is just the region of code where it is "visible" and can be used. Julia uses *lexical* scoping, which means that scope is completely determined by the organization of the source code[43]. The need for different scopes is natural in the context of writing ever larger programs. We often want "inner," more specialized parts of our code (a function, or the body of a loop) to be able to see the broader context of an "outer" scope. At the same time, we want *encapsulation*: we don't want an outer

[43] This is in contrast with *dynamic* scoping, in which scope is determined by the current state of the program while running. Dynamic scoping is uncommon, but is used in things like bash or LaTeX.

scope to be *accidentally* changed by what happens inside a function, and we don't want separate functions to interfere with each other if they happen to both use "x" as a variable name.

Julia organizes scope fairly naturally, and it defines two different flavors of scope: *global* and *local*. A global scope is just the outermost scope within any self-contained piece of code. Crucially, "global" in Julia does *not* mean "universal to the whole program!" Instead, each module defines an independent global scope, and there is a separate global scope (called `Main`) that Julia sets as the currently active module when it starts. Let's define the following:

```
julia> x = "A name in Main's global scope";

julia> module MyModule
           x = "A variable in MyModule's global scope";
       end
```

We can then explore how these global scopes work:

```
julia> println(x)
"A name in Main's global scope"

julia> println(Main.x)
"A name in Main's global scope"

julia> println(MyModule.x)
"A name in MyModule's global scope"
```

In the above, we see that we can *define* a global scope (there, `MyModule`) within another global scope, but those scopes act as independent outermost scopes. In contrast to this are *local* scopes, which act as nested "workspaces" for names. In Julia things the primary constructs[44] that create local scopes are functions, `for` and `while` loops, array comprehensions, and `let` blocks.

Now, what are the rules for how scopes interact? The fundamental rule is that inner scopes can see names from their outer scopes: a function can see the global variables of the modules it's defined in, a loop inside a function can see that function's local variables, and so on. Given this rule, what happens if you write something like "`x = 13`"? In a local scope, if `x` is already a local variable, that existing variable gets the assignment. If `x` is *not* already a local variable, then a new local variable of that name is created and it gets the assignment. Similarly, if you are in a global scope, this will either create a new global variable of that name (if it doesn't already exist) and assign it a value or just assign the value to the existing global.

So far, easy enough. If you are in a local scope and there *is* a global variable of that name there can be some subtleties. If you *want* to modify a global variable from within a local scope you can be unambiguous about this by using the `global` keyword. Sometimes, though, especially when working in the REPL, you often want to modify global variables without decorating

[44] Along with `structs`, but see the manual for some subtleties

your REPL code with this extra keyword[45] As an interactive convenience, Julia has a notion of "hard" and "soft" local scopes, and when working interactively, soft local scopes *will* change global variables rather than making a new local variable if a global variable of the name you're trying to use exists.

Finally, there are some rules about where in your code you are allowed to define different scopes. For instance: modules and structs can each only be defined within a global scope themselves, whereas functions, loops, and comprehensions can be defined in either global or local scopes. As a simple pair of examples, that means you *are* allowed to define a function inside another function, but *are not* to define a convenient struct inside a function. The details are a bit involved – and you should *definitely* read the manual's scoping section to get all of them – but the general principle should be fairly intuitive. If you want to sidestep nearly all of the complexities (and also follow good code practices), consider this:

Best practices for scope (and Julia code in general)

Organize your projects into modules, keep the logic of your code inside of functions, and have functions communicate only through their arguments and return values (rather than by reading and modifying global variables).

5.3 Testing our code

How do we know whether the code that we're writing does what we think it does? Or how can we be confident that when we make some change to an existing code base that we didn't accidentally break something that used to work? The answer is to write automated tests. While there are many different testing philosophies – both for the tests themselves, and how much the writing of those tests should be integrated into the development phase of the project – it is helpful to have in mind three main categories of tests. *Unit tests* are the most common – a unit tests focuses on testing a small piece of code in isolation. This is typically targeted at individual functions, verifying that they work correctly over a given set of inputs designed to probe typical inputs, edge cases, and so on. *Integration tests* focus on how closely related parts of your code are sewn together, checking the "seams" between the functions. This often involves making sure that functions are passing data between each other correctly. *End-to-end tests* check a complete workflow from start to finish, mimicking how you will actually be using the entire codebase.

To have some concrete examples of these, in our EDMD we are going to need to be able to calculate the next time – if ever – two objects will collide, what happens after a collision, and we need to be able to update the state of our system as time moves forward. We might write unit tests that cover each function individually: does our "time to collision" function give the correct time interval for different initial conditions? have we correctly implemented the physics of elastic collisions? We might write integration tests that confirm that we have connected the functions needed to advance the system to the instant after the next collision correctly: does our "evolve the system" function correctly identify which pair of objects will collide next, move

[45] Perhaps more importantly, not needing to write things like `global x = ...` makes it easier to debug code you might be working on by pasting it from a file into the REPL.

all particles to their updated positions, and update the velocities of the correct set of colliding objects? Finally, we might write end-to-end tests for the whole set up: given a particular set of initial conditions, if we evolve the system for a known number of collisions do we arrive at a state that we know is the correct one?

The `Test` package is part of Julia's standard library, and it establishes a strong, standard convention for how testing in Julia projects is organized. The tests for a package live in a dedicated `test/` directory, which contains a script named `runtests.jl` that orchestrates all of the tests for the project. Thus, a basic `MyResearchProject` has these elements:

```
$ MyResearchProject/
├── src/
│   └── MyResearchProject.jl
├── test/
│   └── runtests.jl
├── Manifest.toml
└── Project.toml
```

A typical `runtests.jl` file is often organized like this:

```julia
# test/runtests.jl
import MyResearchProject #I'm `import`-ing for clarity below
# having `using MyResearchProject` instead would be idiomatic
using Test

@testset "Tests for MyResearchProject" begin
    # Tests for a function that adds one
    @testset "add_one function" begin
        @test MyResearchProject.add_one(3) == 4
        @test MyResearchProject.add_one(-1) == 0
    end

    # Julia has built in functions of testing approximate equality
    # The ≈ symbol does this; and you can specify the tolerance
    @testset "magical pi function" begin
        @test MyResearchProject.magical_pi(1.0) ≈ 3.14159 atol=1e-3
    end
end
```

The `@test` macro checks if the expression following it is true; if it is the test will pass, and if not an error will be thrown. The `@testset` macro can be used to group related tests together – this is particularly helpful as you test larger and larger projects. Because the `runtests.jl` is itself just a normal Julia script, it is straightforward to write logic to conditionally run specific tests, working on the correctness of one part of your codebase at a time.

To run the tests for our project, we can use the package manager's built-in `test` command from the REPL – by convention, this command looks for a `test/runtests.jl` file and executes it in a clean environment, populated by only those packages specified in the main `Project.toml` and `Manifest.toml` files[46]. Launching Julia from our project's root directory, this process looks like:

```
(@v1.x) pkg> activate .
(MyResearchProject) pkg> test
```

A slightly elaborated version of this is used for all of our assignments; look in the `test/` folders of each to see how this testing functionality can be used!

5.4 Building type hierarchies: parametric and abstract types

5.4.1 Parametric composite types

Back in Section 3.3 we defined a "`ParticlePosition`" mutable structure which held some Float64 values. What if we wanted to use a different representation of a floating point number (perhaps less precise, so that our code would run faster), or to have the positions be integers (perhaps because we only wanted to allow positions on a cubic lattice)? Do we have to define a different "`ParticlePositionInt64`" or the like for each new primitive type we want to use? No! Julia lets us define *parametric* types – types that take parameters – so that we can define an entire family of types all at once. The syntax for doing so looks like this:

```
mutable struct ParticlePosition{T}
    x::T
    y::T
    z::T
end
```

Here "`T`" represents any type we want. Having defined this type we could go to the REPL and do something like the following:

```
julia> a = ParticlePosition{Int32}(3,2,1);

julia> b = ParticlePosition{Float64}(1.5,2.5,3.5);

julia> a.x + b.y
5.5
```

[46] One can also introduce additional testing-specific dependencies to your project. As always, details in the documentation.

In this example we've explicitly written the type of ParticlePosition we want, but note that Julia can often infer type parameters from the arguments we pass to the constructor. In this case, we could have written `a = ParticlePosition(3,2,1)` and used `typeof` to determine that Julia had created a `ParticlePosition{Int64}` for us. The power here, as we're about to see, isn't just the flexibility for this one random struct, but that we can now write functions that operate on this struct *without knowing* what `T` is. Julia will create fast, specialized versions of these functions automatically – this is a cornerstone of writing reusable and efficient generic code in Julia.

Parametric types are actually already familiar to us – it's precisely what a type like `Vector{Int64}` is, for instance – and Julia lets us build up our own mutable or immutable parametric types as we desire. Not only can a parameter be a type, as above, but it can also be a *value* of a type. For instance: we are going to want our EDMD simulation to be able to handle collisions between objects not just in two dimensions but also in three dimensions. Do we really have to define a `ParticlePosition2D` and a `ParticlePosition3D`? Again, no! Let's define a "particle" as something that has a position and a mass; rather than hard-coding the dimension of space, we'll let `D` parameterize the dimension of space. Using the StaticArrays package, our structure might look like:

```
struct Particle{D,T}
    position::SVector{D,T}
    velocity::SVector{D,T}
    mass::Float64
end
```

We could construct a stationary particle at some location with unit mass as follows[47]:

```
julia> a = Particle{3,Float64}((1.,2.1,3.),(0.,0.,0.),1.);
```

5.4.2 Abstract types and subtyping

Earlier we referred to abstract types as nodes in the type hierarchy; Julia gives us the power to extend its type hierarchy arbitrarily, and that includes creating new abstract types. This can be extremely useful in organizing related concrete types that we might want to create. We could, for instance, implement a taxonomic hierarchy of life by representing kingdoms, orders, clades, and so on by building up an abstract type tree, and then representing specific species as the concrete types that could actually be a value. This involves combining the new `abstract` keyword and the subtype operator[48], `<:`, as in code block 5.1. Unlike in some other languages where classes can inherit member functions and variables from other classes, Julia's

[47] There *are* some important performance-related implications to using values rather than types as parameters. The StaticArrays package handles this issue, and is already well optimized for working with vectors and arrays with small fixed size. As indicated in the link you can in general still write highly performant code while using values as parameters, but you have to do some extra work to make sure the compiler can infer the types being operated on at all times.

[48] Which you can read in your head in these examples as "`X` is a subgroup of `Y`."

```
abstract type AbstractAnimal end
""" A clade of "lizard-faced" amniotes"""
abstract type Sauropsida <: AbstractAnimal end
""" A crown group of "ruling reptiles" """
abstract type Archosauria <: Sauropsida end
""" A concrete Archosaur (skipping a division)"""
struct SaltwaterCrocodile <: Archosauria
    name::String
    number_of_teeth::BigInt
end
""" Another concrete Archosaur"""
struct BeeHummingbird <: Archosauria
    name::String
    length_in_millimeters::Float64
end
```

Code block 5.1: An artificial construction of a type hierarchy with abstract and concrete types. But see the "Creating and using type hierarchies" comment below!

`structs` (concrete types) cannot be subtypes of other `structs`. Instead, a `struct` can only be a direct subtype of an abstract type. Thus, for better and for worse, there are no concrete `Circle` structures that are subtypes of `Ellipse` structures.

How might we use these ideas in the context of our EDMD simulation? Let's imagine that we'll be working with "`Particles`" as we've already defined them – things that can move around and bounce off of things – but also "`Obstacles`" of different sorts. These are meant to represent stationary objects that particles will be able to bounce off of, but which do not themselves move around or interact with other obstacles. We want to be able to define in our simulation a Vector of Particles and a Vector of Obstacles, but we'll probably need different *properties and rules* for different obstacles. For instance, a flat wall can be defined by a surface normal and a point on the plane (i.e., two `SVector{D,T}` values), whereas a spherical obstacle can be defined by the position of its center and its radius. Rather than having all obstacles carry around irrelevant or redundant values just so that we can describe every possible flavor of obstacle we might come up with now or in the future, we'll harness the power of Julia's extensible type hierarchy to declare parametric versions of a *new abstract type* and concrete types that are subtypes of that abstract type.

```
abstract type AbstractObstacle{D,T} end
struct Hyperplane{D,T} <: AbstractObstacle{D,T}
    normal_vector::SVector{D,T}
    point_on_plane::SVector{D,T}
end

struct SphericalObstacle{D,T} <: AbstractObstacle{D,T}
    center::SVector{D,T}
    radius::T
end
```

Having done this, we can now easily create a `Vector{AbstractObstacle{D,T}}` (for specific parameters, like `Vector{AbstractObstacle{2,Float64}}`) whose elements can be any kind of concrete obstacle subtype that matches those parameters. This is crucial for writing generic functions that can operate on any obstacle type.

Creating and using type hierarchies

Abstract type hierarchies like the animal one above might be cute (?), but just because we *can* create an extensive "`A` is a `B` is a `C` is a..." classification doesn't mean that we *should*! The primary power of abstract types in Julia is to define a common set of behaviors (an *interface*) that enables multiple dispatch – this allows us to write generic code that behaves differently for different concrete types, as we'll see next.

5.5 Multiple dispatch

We now can declare vectors of abstract obstacles whose elements can be one of a number of concrete obstacle types. The way a particle interacts with an obstacle depends on what type of obstacle it is, so how much work is it going to be for us to call the correct method given any specific interacting pair (which might involve two particles, or a particle and a obstacle)? Thanks to one of the defining features of Julia, *multiple dispatch*, the answer is "none!"

"Dispatch" is the term for how at run time a program selects what specific method (i.e., implementation of a function) to execute based on the types, numbers, and/or values of arguments passed to the function call. In a language like C, you write a function with a unique name and some number of typed arguments, and that's what gets called – end of story (in very non-standard terminology, we might call this "zero dispatch"). "Static dispatch" (or compile-time method resolution) is a characteristic of a language like C++ that has *function overloading*, allowing you to give the same name to different functions as long as they have different numbers or types of their arguments. "Single dispatch" was a breakthrough in traditional object-oriented languages, allowing you to dispatch to different functions depending on the type of *one* of the arguments. This is done in C++-like[49] languages by defining classes with methods, and using a syntax that elevates one argument over the others. This single-dispatch style,

[49] For which there is, interestingly, apparently a long history of *wishing* the language had been written with multiple dispatch baked in.

common in traditional OO languages, often leads to syntax like `abacus.multiply(2,12)` and `fingers.multiply(2,12)` – here method selection depends on the type of the object preceding the dot.

In a language with multiple dispatch like Julia, the specific method chosen to execute is determined by the combination of the runtime types of *all* of the arguments. This allows you to write methods that look like `multiply(x::Number, y::Number, a::algorithm)`. Multiple dispatch is one of the solutions to the so-called *expression problem*, and it has been argued it is one of the driving features that created a vibrant ecosystem of shared, reusable code in the Julia community. Given the kind of type system Julia possesses, there need to be rules for choosing which method not only matches but *best* matches a particular call – a nice visualization of this can be found in this blog post. Julia generally does the thing that you expect: it selects the *most specialized*[50] method that matches the argument list.

Specializing on methods?

Double check your understanding – what does the above paragraph imply about which of the methods `add(x::Number,y::Number)` and `add(x::Signed, y::Signed)` and `add(x::Int64,y::Int64)` that you've defined would be called as you try to `add` different values? How does this relate to what we did when we extended the `Base -` binary subtraction operator to work with PolygonVertex types in Section 3.3?

In the context of our EDMD simulation, we can write an `collision` function that both takes and returns two values, where the return values correspond to the state of the argument values after a purely elastic collision. By writing multiple *methods* for this function that specialize on the types of the arguments, we can use Julia's multiple dispatch to handle the business of calling the right method regardless of what combination of particles and obstacles we pass it. This might look something like code block 5.2 Notice that we're not modifying the Particle or AbstractObstacle types when we add these collision methods; we're defining new methods that are *external* to those structs. This means that different parts of a system, or even different packages, can independently extend how types interact without needing to change the original type definitions – this is key aspect of what makes Julia so composable, and what we mean when we say that multiple dispatch is a solution to the expression problem.

In the above context, by the way, the "`where`" keyword part of the method declaration makes the parameters of the argument types (like `D` and `T`) available as parameters *for the method itself*. This allows Julia to compile a specialized version of the method for that particular combination of concrete types. The "`where`" keyword can be used not just for parametric types, and can also be used to specify constraints you want to hold, for instance in a method declaration like `function foo(x::T, y::T) where {T <:Number}`.

Notice, also, that we don't need to specialize the methods more than necessary. In this case, if we wanted particles to perfectly reflect off of all obstacles (not, indeed, how a collision with a

[50] Defining "most specialized" heuristically as "farthest from the root of the type tree". Julia has complex rules to handle exactly what methods will be called when specializing on multiple types, how to handle tie-breakers between specializations, and when it will complain that something is so ambiguous that you, the coder, must be more explicit in which method to call.

```
function collision(p1::Particle{D,T}, p2::Particle{D,T}) where {D,T}
    #logic to define new Particles post-collision...
    #new_particle1=...
    #new_particle2=...
    return new_particle1, new_particle2
end

"""particles colliding with abstract obstacles reverse velocity"""
function collision(p::Particle{D,T}, o::AbstractObstacle{D,T}) where
{D,T}
    new_vel = -p.velocity
    new_particle = Particle{D,T}(p.position, new_vel, p.mass, p.radius)
    return new_particle, o
end
```

Code block 5.2: Multiple dispatch for defining collisions between different types.

sphere would work!) we could write the method signature as above. If we wanted to handle the SphericalObstacle case separately and correctly, we could keep the above method and add an additional specialization on `o::SphericalObstacle`. It's worth remembering that the order of the arguments matters, so here for completeness we'll make sure that we correctly handle the case of passing an obstacle and then a particle:

```
function collision(o::AbstractObstacle{D,T}, p::Particle{D,T}) where
{D,T}
    new_particle, newObstacle = collision(p,o)
    return newObstacle,new_particle
end
```

Actually implementing our EDMD simulation now involves a few further primary steps. The first, of course, is to actually handle the logic of collisions between different types. Next, we should figure out an efficient way of calculating when the next collision between any pair of types might be (including, for instance, the option of returning `Inf` if the two types would *never* collide given their current positions and velocities). For efficiency we could expand this into the population of a data structure – perhaps a `PriorityQueue` from the `DataStructures.jl` package – that keeps track of upcoming collision events. This would allow us to efficiently advance the system from one collision to the next. Finally we would add a way of saving data, or of visualizing the results of these simulations. In the Problems you'll do some of this work yourself, but see the course git repo for a motivating movie I made using `GLMakie` that lets you interactively add colliding particles moving on an ergodic-billiards-like table!

5.6 Hello, π! (Method 5: Counting collisions)

If you haven't seen this calculation before, you might be thinking to yourself, "Daniel, that's all well and good, but what does *any* of this EDMD business or 'counting collisions' have to

do with how we'll calculate π?!" Indeed, this first time I saw Ref. [14] I laughed out loud – not something that happens very often to me while reading the physics literature. Before you go and read that paper I'll describe the set up (also depicted in Fig. 5.1), and you should spend some time thinking about why this might even be tangentially related to π.

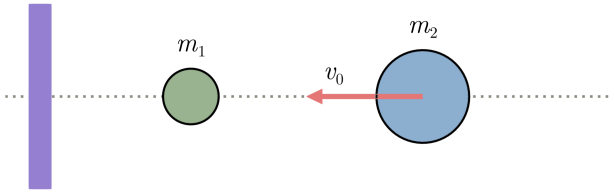

Figure 5.1: Our physical setup for calculating π: an immovable wall (e.g., of infinite mass) is on the left. In the middle is an initially stationary particle of mass m_1. On the right is a particle of mass m_2 moving with initial velocity v_0 along the x axis towards the middle mass.

The plan is the following. We'll set up a system of three objects in one dimension. At the origin is an immovable wall – imagine that it has effectively infinite mass. Somewhere far to the right of the wall is a particle of mass m_2 moving to the left with velocity v_0. In between this particle and the wall is an initially stationary particle of mass m_1. The particles move in a frictionless environment (or perhaps they're flying around in the vacuum of deep space), and all collisions are *perfectly elastic*: collisions between the two particles conserve both energy and momentum, and collisions with the infinitely massive wall just flip the sign of the colliding particle's velocity vector.

We'll write a function `pi_pool_initialization(mass_ratio)` that will accept the ratio m_2/m_1 and initialize a `System` in the configuration just described, working in units where, say, $|v_0| = 1$. Assuming we've written an `evolveStep` function that takes a system and advances it to the next time of collision (assuming that there is still a collision that will happen) and returns how much time had to elapse to do so, we'll define the following:

```
mutable struct System{D,T}
    particles::Vector{Particle{D,T}}
    obstacles::Vector{AbstractObstacle{D,T}}
    total_collisions::Int64
    current_time::Float64
end

function pi_from_pool(mass_ratio::Float64)
    s::System = pool_pi_initialization(mass_ratio)
    time_to_next_collision = 0
    while(time_to_next_collision != Inf)
        time_to_next_collision = evolve_step!(s)
    end
    return s.total_collisions
end
```

What do we expect will happen? If $m_1 = m_2$ the analysis is straight out of physics 101: particles 1 and 2 will collide (after which particle 2 will be stationary and particle 1 will have velocity v_0), then particle 1 and the wall will collide (after which particle 1 will reflect and move at speed v_0 to the right), and then particles 1 and 2 will collide again (after which particle 1 will be stationary and particle 2 will move with speed v_0 to the right). Nothing interesting happens after this: particle 2 flies off to infinity, leaving us with three total collisions. Well... let's see what happens as we play with the mass ratio:

```
julia> EventDrivenMolecularDynamics.pi_from_pool(1.)
3
julia> EventDrivenMolecularDynamics.pi_from_pool(100.)
31
julia> EventDrivenMolecularDynamics.pi_from_pool(10000.)
314
julia> EventDrivenMolecularDynamics.pi_from_pool(1000000.)
3141
julia> EventDrivenMolecularDynamics.pi_from_pool(100000000.)
31415
julia> EventDrivenMolecularDynamics.pi_from_pool(10000000000.)
314159
julia> EventDrivenMolecularDynamics.pi_from_pool(1e16)
314159265
```

All we had to do in order to get the first 9 digits of pi was calculate collisions between particles whose masses differed by a factor of ten quadrillion? You've got to be kidding me.

5.7 Additional resources

The concepts and tools we've covered will already let us do a tremendous amount – you're already equipped to tackle many of the computational challenges we'll encounter this semester (and beyond)! However: Julia is an extremely deep and versatile language, and our exploration has in many ways only scratched the surface (for instance it's extensive metaprogramming capabilities and it's built-in paradigms for parallel and concurrent computing[51]). For those looking for a place to start diving deeper, the following are resources that I found useful while I was learning myself, along with some other highly regarded guides to help continue your journey.

Core resources and community

- If you haven't picked up on this yet: the official documentation should be on your must-read list. It's comprehensive, authoritative, and the ultimate reference.

- Speaking of community ... the Julia discourse is a central hub for community support, questions, and discussions. It's an active and welcoming environment, where questions often receive strong, detailed, and positive feedback.

- JuliaPackages and the JuliaHub package search page. As you continue your journey, if you want to find packages that deal with specific problems, or want to see how other people write Julia code, these are both great places to browse and search for Julia packages.

Helpful books and in-depth guides

- "Think Julia: How to think like a computer scientist" is a longer book (roughly 300 pages) which is also available online. I found it only after writing most of these notes, but a quick look suggests that it is a very pedagogical and more thorough look at many corners of Julia than I presented here. It is aimed at students who may not have any programming experience, and thus also walks more carefully through the fundamentals of coding and of the Julia language.

- "Practical Julia" is an even longer book, and it seems to be of very high quality. It goes through the basics quite thoroughly in Part 1, and then dives into applications from different fields in Part 2.

- A Deep Introduction to Julia for Data Science and Scientific Computing. This workshop material is targeted at people who already know languages like Python or MATLAB, and uses an active, problem-based approach to start from the beginning and then go deep into the Julia language and ecosystem.

[51] So much so that Julia is one of only a handful of languages – the others that I know about being FORTRAN, C, and C++ – that have achieved petaflop-scale performance.

Other guides and references

- Modern Julia workflows has a particularly nice explanation of the environment / module / package system in Julia, and (as you might expect from the name) strong recommendations for productive workflows. It also provides guidance on using IDEs like VSCode or notebook environments such as Jupyter or Pluto, as alternatives or complements to the text-file-plus-REPL workflow emphasized in these notes.
- Learn Julia the Hard Way has some excellent pedagogical content. It is targeted, in its words, at "... people who need to get a job done, not computer scientists."
- julianotes.jl is a collection of explanations and practical tips, frequently distilled from conversations on the Julia discourse.
- Learn X in Y minutes, where X=Julia, offers a concise "cheat-sheet" – perfect for a quick reminder of core language syntax.
- Scientific Programming in Julia is a course that focuses more heavily on design patterns in Julia and various performance-centric topics. It's "Lecture 2" ("The power of Type System and multiple dispatch") is an excellent introduction to that part of the language; especially if my explanations above didn't resonate, I encourage you to give it a read.

As we progress through this course I encourage you to learn more, and tell me about particularly helpful resources you find along the way!

5.8 Confession

I will very occasionally bend the truth in these notes if I feel like there is a strong enough pedagogical reason, but I'll always come clean. Back in Chapter 1 I said that when I first opened Julia I added one and one together, and then closed the REPL. While true in spirit, the very first time I opened Julia it wasn't *exactly* that smooth; it actually looked more like:

```
julia> 1+1
2

julia> exit
exit (generic function with 2 methods)

julia> quit
ERROR: UnDefVarError: `quit` not defined in `Main`
Suggestion: To exit Julia, use Ctrl-D, or type exit() and press enter.
Suggestion: check for spelling errors or missing imports.

julia> exit()
```

It wasn't exactly my finest moment. But it *was* an encouraging early indication of how Julia was going to help make the process of learning it easier!

Module I

Module 1: Introduction and foundations

We're reading this Module and Module 0 in tandem. In "Hello, π..." we are learning the vocabulary and grammar of a programming language, giving us the ability to express scientific ideas as code. This is sufficient for many projects, but it leaves many aspects of the craft of scientific computing — aspects which are independent of whether we're using Julia, or C++, or Python, or yet another language — to the side. How do we organize a project so that it doesn't collapse under its own weight as it grows?

Figure I.1: Robert Boyle (left) and Christiaan Huygens (right), the two natural philosophers involved in the earliest dispute (that I could find) in which the reproducibility of an experiment was questioned and resolved in a basically modern scientific manner [15]. Images in the public domain.

How can we actually keep a meticulous record of our work, ensuring its reproducibility, when we are simultaneously running simulations and writing more code to both fix bugs and explore new corners of our models? How do we design our computational experiments so they are both scientifically sound but also computationally feasible? And what tools are available to make *collaborating* with other scientists on our work easy?

This module covers the essential tools and practices in modern computational research. We'll learn about version control of our programs, scripting languages for automating analyses and simulations, the principles of reproducible research, and fundamental concepts in algorithmic complexity. This module will ensure you can manage coding projects effectively, collaborate efficiently, and think critically about the computational resources that would be needed for different tasks.

For a deeper dive into topics in this module – version control, best practices in computational research, and algorithmic complexity, consider the following references [16, 17, 18, 2].

Chapter 6

Version control with Git

Version control is a systematic way of managing multiple versions of programs, of documents, of databases, and so on. Using version control makes working collaboratively with others much easier. It is also *crucial* for modern scientific reproducibility. Git is an extremely powerful system for distributed version control, and it has been overwhelmingly adopted. There are numerous guides and tutorials that will help get you up and running; my goal in this chapter is twofold. First, I want to introduce the most common, practical subset of git commands that you'll want to use from day one. Second, I want to talk about how git actually works – it does not need to be a black box, and by having a solid mental model of what happens when you use different git commands you'll avoid some of the common pitfalls that sometimes trip beginners up.

6.1 Git in practice

From a practical point of view, there are two important sets of things to learn about git. The first set is just the essential commands: How do you actually create a new "version" of your whatever your project is? How do you go back and forth between versions or compare differences between them? How do you synchronize your changes with collaborators? In the first subsection below I'll quickly cover the basics[52].

The second set is a byproduct of (a) git's "branching" model of projects and (b) the fact that git is a *distributed* version control system. "Distributed" here means that no copy of the project is more or less important than any other, except by convention. This has several nice features, but it also means there are many different patterns – often called "workflows" – for interacting with git and using it for version control. Especially when collaborating – but even when just working on your own code base – it is helpful to choose a consistent pattern of using git. While there are many different possible workflows, I think there are two that are most useful for solo projects or those involving only a small number of collaborators at a given time. I'll describe those two after covering the core commands.

6.1.1 The core commands

Nobody likes to be told to read the documentation, but the first superpower at your disposal is

[52]For alternate perspectives, consider some of the many other resources out there.

```
$ git help [command]
```

Here "`[command]`" is, not surprisingly, any git command. The output will remind you what the command does, what its options are, and other helpful pieces of information.

Getting started: init and clone

A repository ("repo") is git's fundamental unit, and each project (a paper, a codebase, a website, etc) will live in its own repo. That repo will contain all of the files associated with that project, each file's complete version history, and can even contain various parallel or alternate versions of files. You can create a new repository by moving to some directory and typing

```
$ git init
```

This initializes a repo by creating and populating a hidden `.git/` subdirectory. You can also create a new copy of an existing repo like so:

```
$ git clone [repository]
```

Here `[repository]` is either a URL pointing at a remote repo, or a path to a local repo already on your computer.

Creating versions of your project: status, diff, add, commit

Git has a "stage-and-commit" model for updating repositories. You can think of commits as the actual versions of the project you'll be able to go back and forth between, and the "staging area" as a rough draft space for the version you are about to commit. This staging area is different from the actual current state of your project, which might have many changes (new or modified files and deleted files) that are not currently staged. One reason for having this distinction is that some workflows favor having small commits that each address some specific update to the project, and using git's stage-and-commit model let's you take the many changes you may have made during a coding session and record them as a specific sequence of project versions.

In practice, you can see the current status of your project by running

```
$ git status
```

This will give an overview of your project relative to the last version that was committed: what files are in the staging area ready to be committed, and what files have been changed (or what new files have been added) but are not in the staging area. If you want more granular detail here – for instance, what lines of various files have actually been changed – you can use the `git diff` command.

To actually move a file to the staging area you run

```
$ git add [path/to/filename]
```

Turning changes in the staging area into a new version of the project is done like so:

```
$ git commit
```

By default, running `git commit` will open up a text editor asking you to specify a "commit message" – this should be a short description of the changes associated with the new version of your project.

There are *many* options for all of these git commands – and other commands you can use – that make them easier to work with. For instance, if you want to add all new and changed files to the staging area you can use `git add .` On the other hand, what if you add a file to the staging area by mistake[53]? You can "unstage" it using the `git restore` command:

```
$ git restore --staged [filename]
```

This moves the file from the staging area back into the "changed but not staged" category, i.e., without discarding your actual changes.

If you want to skip the "let's open up a text editor for the commit message" business you can use a `-m` flag on `git commit`. And if you want to skip `git add` step *for files that git is already tracking*, you can use a `-a` flag on `git commit`. This flag automatically stages all modified or deleted tracked files before committing, but it will not stage new, untracked files. So, for instance, after I finish writing this section I'll probably go to the shell and do something like:

```
$ git commit -am "added core git commands to git.tex"
```

Navigating history: log, switch, and restore

After you have committed several versions of your project to the repo, you can use the `git log` command to view the history of your project. By default it will list commits in reverse chronological order, with four pieces of information for each commit: (1) a SHA-1 checksum that serves as the commit's identifier, (2) the author of the commit, (3) the date of the commit, and (4) the commit message. While there are many options to make the log easier to parse, in practice it is typically easier to view the log through a web interface – thus, the basic command line version is sufficient most of the time.

Also, unless you're very good at memorizing hashes, write useful messages.

Having the complete history of your project would be of only limited use if you couldn't actually access past versions of your project. Here we'll use the `git restore` and `git switch` commands. To discard the changes to a specific file in your working directory (i.e., restoring it to its last committed state), you can do this:

[53] Something extremely easy to do if you go around typing `git add .` all the time!

Figure 6.1: XKCD with an astute observation about commit messages

```
$ git restore [path/to/file]
```

This is a safe and quite common way to undo unwanted edits. If you want to more specifically restore a file to the state it was in at an older commit, all you have to do is specify the source[54]:

```
$ git restore --source=[commitID] [path/to/file]
```

If you want to restore not a single file but your *entire project* to the state it was in at a past commit, you use the `git switch` command with a special flag:

```
$ git switch --detach [commitID]
```

The `--detach` flag indicates that you are entering the special "detached HEAD" state.

The detached HEAD state

When you use `git switch --detach` with a specific commit hash, you are asking Git to show you your past project exactly as it was at that time; this is very useful, but it places your repository in a special state called a *detached HEAD* (`HEAD` being special reference for git that usually points at your current position at the end of a branch). The crucial thing about this special state is that it *does not* belong to any branch – it's a floating reference that can and will be deleted by git's cleanup processes when you switch back to a real branch.
In practical terms, you need to know the following:

1. **If you just want to look around,** or run code from this point in time: you are save to do so: look at files, compile code, run tests, go wild. When you're done, go back to any branch (e.g., `git switch main`).

[54] Replacing `[commitID]` with the SHA hash of the commit you are interested in.

2. **If you want to build upon this old code:** you *must* create a new branch to save your work – creating a new branch from your current "detached HEAD" state gives this point in your project's history a name and a future. You can do this, e.g., by `git switch -c newExperimentalFeature`.

A note on the checkout command

If you look at git tutorials, you'll find a `git checkout` command that gets used for many different contexts: creating branches, restoring files, moving between commits, and so on. This one commands many different functions were a common source of confusion, and in 2019 git introduced `git switch` and `git restore` to provide more tailored commands for those actions. While `git checkout` still works (and is sometimes necessary for advanced workflows), we'll use `switch` and `restore`: they make the user's intent clearer and are safer to use.

Synchronizing clones: remote, push, pull

Using git entirely locally is itself very useful, but you will almost invariably want to synchronize your work with a remote – these are just versions of your (or someone else's) repository that live somewhere on the internet (e.g., on GitHub, or GitLab, or...). If you started a repo with `git init` you can connect it to a remote repository using the `git remote [various options]` command; if instead you started by cloning a remote repo things will be configured to synchronize easily with that remote by default. If you want you can associate arbitrarily many different remotes with your repository – this just involves many uses of the `git remote` command, and you can use `git remote -v` to see what your remotes are and how they are configured (read only, read-write).

Most of the time, you'll probably keep things simple and only have a single remote associated with your project. In this case, once you've set the remote up synchronizing with it is quite straightforward. To move your local commits to the remote you `git push`, and to take any updates that the remote has and combine it with what you have locally you `git pull`. If you want to grab all of the data the remote has but you do not want to immediately try to combine it with what you have (perhaps you're worried about incompatible changes you and a collaborator may have made to a file, and want to check everything out first), you can instead do a `git fetch`.

Creating parallel versions: branch and merge

Git's branching model makes it easy to have multiple parallel versions of your project that can develop independently from each other. It is possible because, at the end of the day, branches are just lightweight, movable pointers to a commit – cheap to make but powerful in practice. You can create a new branch – perhaps to experiment with a new feature, or develop such a feature over a long time – without interfering with whatever is going on with the main part of your project. Creating a new branch is as simple as `git branch [nameOfYourNewBranch]`, and switching between branches is as simple as `git switch [nameOfBranchYouWant]`. You can also create a new branch and immediately switch to it in a single command:

```
$ git switch -c [nameOfYourNewBranch]
```

Each branch can independently maintain its own version history (see more on that below), and when synchronizing with a remote you have complete control – you can push and pull all of your branches, a subset of them, or just have.

Eventually, you will decide that the feature you developed on some branch should be combined with your main branch (for the purposes of this guide, I'll assume that your main branch is named `main` – the actual name makes no difference to git). The basic way of doing this is to use a "merge" command: assuming that you are currently on `main` (i.e., you've recently done a `git switch main`) and you want to combine the work you did on a `featureBranch1`, all you need to do is this:

```
$ git merge featureBranch1
```

This will attempt to replay all of the changes made on `featureBranch1` on top of `main`. If there is a conflict that cannot be automatically resolved – perhaps incompatible changes were made to the same file on these two branches – the merge will stop, and you will have options for how to resolve these conflicts. I strongly recommend not beginning a merge if you have any uncommitted changes in your project.

Pull is a combination of other commands

Now that we've met both `fetch` and `merge`, I can tell you that `git pull` is essentially just a combination of these two other commands: `fetch` to get data from the remote and `merge` to combine the remote branch with your local one[a]. This is why conflicts sometimes happen during a `pull`.

[a] The `pull` command can also be configured to use a different pattern: a `fetch` followed by a `rebase`. Rebase and merge are alternate strategies for integrating changes from one branch into another. Using `rebase` – and especially its interactive version – is incredibly powerful, but it also re-writes the history of your git repo and can be dangerous. We'll leave this more advanced topic aside for now.

6.1.2 Configuring git

In addition to the basic set of git commands, there are important configurational options when it comes to setting up and using git that we should know about right out of the gate.

The first is your global `.gitconfig` file, which lives in your home directory (not your project directory), and can be created by hand or using the `git config` command. This needs to be used to set the name and email that will be associated with your commits, but can also be used to set lots of options – alias for commands you want git to use, the default editor to use when writing commit messages when not using the `-m` flag, and so on.

The second is the `.gitignore` file, which is a per-project file located in the root of your repo which instructs git that, by default, certain files should not be added to your repo. For

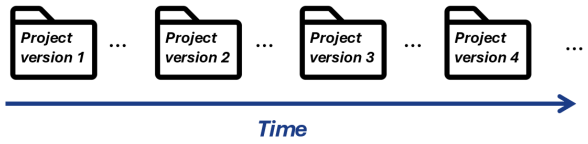

Figure 6.2: Schematic view of a project with a centralized workflow.

instance, in a repo containing a LaTeX document you might not want to version control all of the ancillary files that TeX generates as it compiles your document, so you might write a `.gitignore` file like this:

```
## ignore core auxilliary files
*.aux
*.log
*.out
# ... other files
```

If you compile your TeX document (so files like this exist) and type `git add .` you will find that files matching either the specific names or the patterns you have written in the gitignore file are *not* moved to the staging area. You can force git to add an ignored file (using `git add -f`); the `.gitignore` file just controls the default behavior. This is extremely useful for keeping a clean repo, only version controlling the files for which version control is appropriate.

6.1.3 Common workflows

Patterns of interacting with git can get quite involved, especially as the size of a project and the number of contributors to it grow. For the kind of smaller-scale projects we will be working with, the following two simplest patterns will serve us well.

Centralized workflow

The first is usually called the *centralized workflow*, in which we ignore git's branching model altogether and have all of our project's history linearly recorded on the repo's main (i.e., only) branch. Schematically, our project's history will look like that of Fig. 6.2.

In this schematic time progresses from left to right, and I've deliberately used dots to connect the project versions themselves rather than the arrows you might have expected. The reason for this choice will be clear when we talk about how git stores a repository in Section 6.2, and we realize that arrows connecting the different versions should be drawn opposite to the flow of time.

This workflow works well for small computational projects – perhaps small analysis scripts, or collections of related Mathematica / python notebooks – and also especially for working on papers (which should definitely be version-controlled with a method better than saving files with names like `manuscriptDraft_v10_final_revision_v2.tex`).

Centralized workflow

For concise reference: the basic pattern of operation with this workflow is

1. Initialize the repo: `git init` or `git clone`
2. Make and commit changes: `git status`, `git add`, and `git commit`
3. Synchronize with the remote
    - Get any changes (if you are collaborating): `git pull`
    - Handle any conflicts, if necessary
    - Push changes: `git push`
4. Repeat steps 2 and 3 until done.

This simple pattern uses only a handful of git commands, but is already very useful. It can also be meaningfully thought of as a building block for more complicated ways of working with a repo. The next example demonstrates this, where I'll use the abbreviation CWF to refer to steps 2–4 above.

Feature branch workflow

As projects get larger, the main branch of your project using CWF can start to get cluttered and chaotic. Perhaps you're building a particle-based simulation framework, and you are simultaneously adding new forces and equations of motion and boundary conditions, all while occasionally finding and fixing a bug or two. The commits for these additions are all interwoven in your project's history – does a particular new feature rely on a bug fix earlier in the commit history, or not? Are the changes you needed to make to a file containing a common `simulationFramework` class when working on two different features compatible, incompatible, or completely independent from each other?

A nice pattern for encapsulating the work on separate parts of your project is called a "feature branch workflow. It is based around having a primary `main` branch for your project, and instead of committing directly to it you are encouraged to create a new branch when you want to start working on a new feature. These feature branches get merged with the main branch when they are ready, and then they don't need to be touched again (again, since branches are just lightweight pointers, there's no real cost to having a bunch of unused ("stale") branches around). It looks schematically like Fig. 6.3.

I use this pattern for my open-source scientific code packages, where I want the main branch to always (hopefully!) have working code that others can use. This is one of the main benefits of a feature branch workflow: you can be developing and testing multiple items independently, and the main branch can stay clean and functional. Yes, I still sometimes commit to main when implementing a quick bugfix, but the basic pattern of "branch for a new feature, merge when it's done" is very convenient.

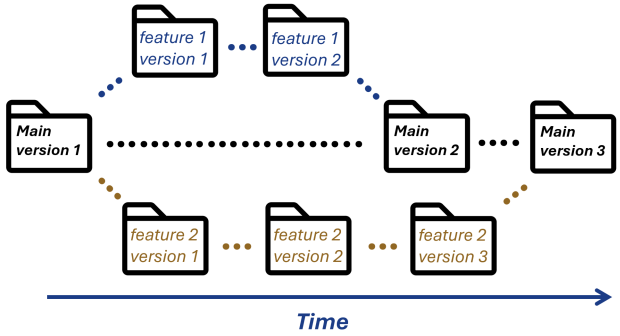

Figure 6.3: Schematic view of a project with a feature branch workflow

Feature branch workflow

The corresponding basic pattern of operation is something like:

1. Start from the main branch (perhaps after some period CWF development): `git switch main`.
2. Create a new branch for a new feature: `git switch -c nameOfNewBranch`
3. Work on that feature branch using a CWF
4. Bring your changes back to the main branch: `git merge`

One thing to note is that this pattern tends to favor relatively small and shorter-lived branches, since merge conflicts get more common and can be harder to resolve the longer a branch has diverged from main. Finally, I'll note that a feature branch workflow can itself be a building block for more complex workflows. The complexity of your workflow should probably scale[55] with the size of your project and the number of collaborators.

6.2 How git stores a repository

It helps to know about how git actually storing your project and its version history. This lets you know what all of the commands in the previous section are actually doing, and it also helps you correctly reason about what will happen when you make a commit, or when you want to merge two branches. So: git is a directed acyclic graph – the nodes of this graph will be a small number of different types of "git objects", almost all of which can then point to other git objects (forming the directed edges of the graph). We'll see how this turns into a system for version control by meeting the important git objects.

6.2.1 Blobs and trees

Blobs

[55] Logarithmically?

The most basic git object is the *blob*, which represents the *contents* of a file.We'll represent them like in Fig. 6.4.

Figure 6.4: A blob object. Mr. Blobby photo credit: Kerryn Parkinson/NORFANZ

What's going on here, and why have I written `d23a1` on the blob? When we `git add` a file (or when a merge changes an existing file's content), git reads that file into a memory buffer and uses a lossless data compression algorithm to figure out what the compressed version of that file is. It then prepares a file which contains a short header followed by that compressed representation of the file. Git next calculates the SHA-1 hash of the blob, and uses the 40-hexadecimal-digit representation of the hash value as the *name* of the blob file, which then gets stored in the `.git/objects/` directory (technically, git uses the first two hex digits of the hash as a subdirectory name and the remaining 38 digits as the file name). The header schematically looks something like this:

```
blob (size of compressed blob in bytes)
(binary representation of compressed blob)
```

That is, it has information that this is a "blob"-type object that will be of a certain size, followed by the compressed version of the file. I don't want to type 40 digits for the names of blobs, so I'll just use 5 letter/number combinations to represent these hash values, as in `d23a1` above[56].

Trees

You may have noticed that nowhere in the blob is there information about what the file is named, or where to find it relative to your project's root directory. Perhaps you are extremely good at memorizing SHA-1 hashes, but the rest of us would probably like to go on using file names and paths as usual. The next kind of object that git uses is a *tree* object – these are objects who's purpose is to point to other blobs and trees, and associate the usual information that we think of when we work with files in a file system. In that sense they are like directories and subdirectories. A visual representation of such an arrangement is in Fig. 6.5.

There is nothing mystical about all of these arrows pointing from trees to other objects; schematically a tree object is represented in a file as something like:

```
tree (size of tree in bytes)
(mode) (blob file name) (blob objectID)
(mode) (blob file name) (blob objectID)
(mode) (tree path name) (tree objectID)
...
(mode) (blob file name) (blob objectID)
```

[56]SHA-1 is a useful hashing algorithm, but it is not cryptographically secure against "collision attacks," where an attacker could craft two files that result in the same hash. For the purposes of version control and protecting against accidental corruption, it remains perfectly suitable.

That is: each tree contains a header (this is a "tree"-type object that will be of a certain size) followed by a list of things the tree points to. Each item in that list has a "mode" – is it a normal file, an executable file, a symlink, a type of directory, a git submodule – a file/path name, and finally an object ID. It's in the sense that the header of the object contains the type and ID of other objects that a node in git's graph "points" to another node.

Naturally, the SHA-1 hash of the tree object gets used as the tree's object ID.

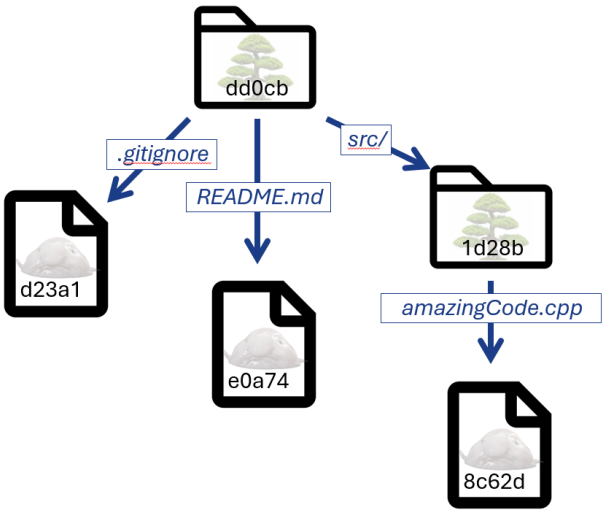

Figure 6.5: A tree object pointing at blobs and trees

6.2.2 Commits

Thinking about the above, we see that we could create different versions of a project by being able to point at different tree objects – in the above image, if I could remember the `dd0cb...` hash I would be able to find the file corresponding to that tree, and from there get all of the sub-trees and blobs that contain information about the state of the project at that time.

Again, unless you are extremely good at memorizing SHA-1 hashes, you probably want a new type of object for this purpose; that is what a *commit* object is for. The format of a commit object is schematically

```
commit (size of commit in bytes)
tree (tree's object ID)
parent (parent commit's ID)
...
(commit information)
```

Yet again we have a header saying that this is a "commit"-type object of a certain size. Here that header is followed by the relevant information about the commit. This includes the tree that itself points to the blobs and trees that make up the state of the project, along with information about any "parent" commits. Typically a commit will have one parent commit, but (a) the first commit of a repository will have zero parents and (b) when merging branches a commit can have multiple parents. Finally, there is a bunch of additional information about the commit: the author, commit date, the commit message, and so on. Because all of this information is used to generate the commit's hash changing anything – even a single character in the commit message! – will result in a completely new commit with a different hash. By now, you will not be surprised to learn that a commit objects' ID is just the SHA-1 hash of the commit object.

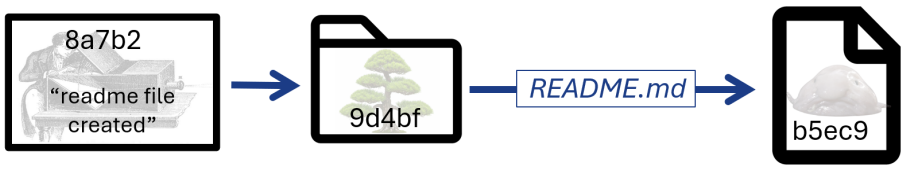

Figure 6.6: The state of the repo after our first commit.

A sample repo over time

Bringing these three basic git object types together, let's see what our repo looks like over the course of a few simple commits. Below I'll go back and forth between simple commands at the shell and a visual representation of the repo. It is implied that whenever the shell command is `nvim [some file]` I am creating or editing that file and saving it. We'll start out simply: in a completely empty directory let's initialize a git repo, edit a single file, then add and commit it.

```
$ git init
$ nvim README.md
$ git add .
$ git commit -m "readme file created"
```

After this, Fig. 6.6 shows what our repository looks like.

Pretty simple: a single commit represented as a snapshot[57] which points at a tree, which points at a blob. Let's add a little bit of complexity by adding a new file in a new subdirectory of our project:

```
$ mkdir src
$ nvim src/amazingCode.cpp
$ git add .
$ git commit -m "code added"
```

Now our repository looks like in Fig. 6.7.

There are a few things to notice. First, as promised, the new commit points both to the parent commit and to a tree. Second, git is happy to re-use any existing data it can: here, the `README.md` file didn't change, so the same blob object is pointed to. On the other hand, the *tree* at the root of our project did change: it contains the file it already had *and* a new sub-tree. Thus, the new commit cannot reuse the original root tree.

To advance one step further, what if we make a new commit that (a) adds a file and (b) edits an existing file? Something like:

[57] Depicted with a drawing of a camera obscura... It's no "blobfish for a blob", but it gets the job done.

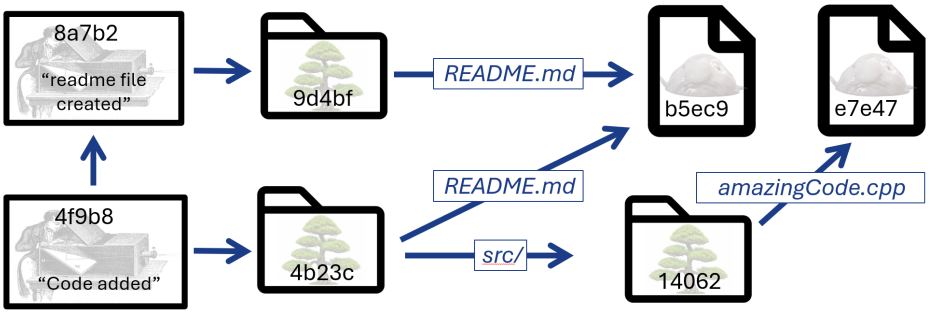

Figure 6.7: The state of the repo after a subfolder and new file are added.

```
$ nvim .gitigore
$ nvim README.md
$ git add .
$ git commit -m "gitignore added and readme edited"
```

Based on what we know, we expect the following. The `src/` directory and its contents haven't changed, so the new commit should point to a tree that points to the *same* `src/` tree as in the last illustration. The root tree that the commit points to should be different, because it needs to be a tree that points at two blobs and one tree (unlike the "one blob and one tree" root tree of the previous commit). Finally, we should see an entirely new blob appear, corresponding to the contents of the edited `README.md` file. Indeed, this is what we have (see Fig. 6.8).

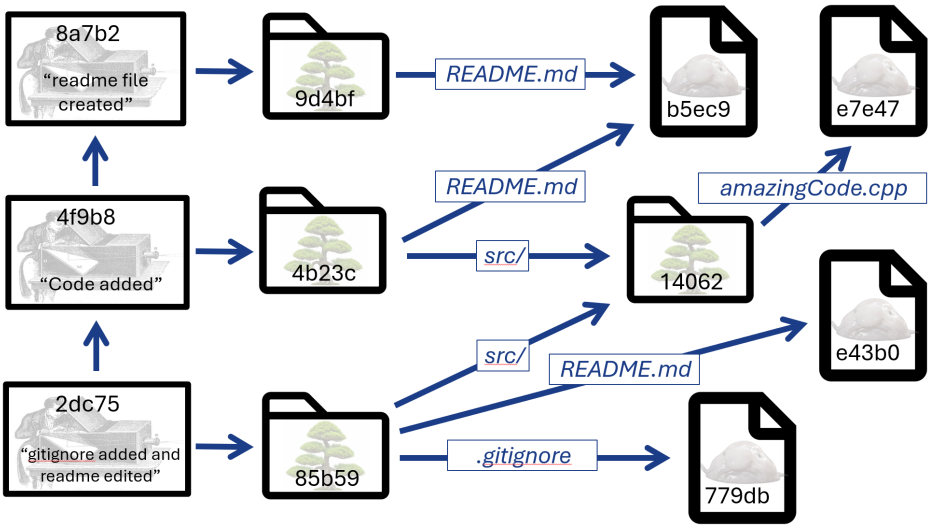

Figure 6.8: The state of the repo after a new file is added and an old file is edited.

It is worth emphasizing again that there are now two different blobs corresponding to the two different versions of the `README.md` file in the repository. And, since both are reachable in the graph from the `2dc75` commit, you have access to both of them. Exactly as you would hope for a version control system.

6.2.3 References

There is one more class of git object to meet as we finish things up, and these are git references – HEADS, tags, and remotes. References are pointers, acting like sticky notes that can tell you where you currently are in your project's history, noting an interesting commit (or, in fact, any interesting node in the graph), or noting an objectID on different clones of your project.

First: where are you currently in your project's history, and what commit do you want to base your next commit off of? You probably don't want to memorize the SHA-1 hash of the answer to this (something of a recurring theme in this section), so git maintains a list of HEAD references (these are files in the `.git/refs/` directory, one for each branch). Each of these files just contains the SHA-1 hash of a commit object corresponding to the current snapshot of the branch in question.

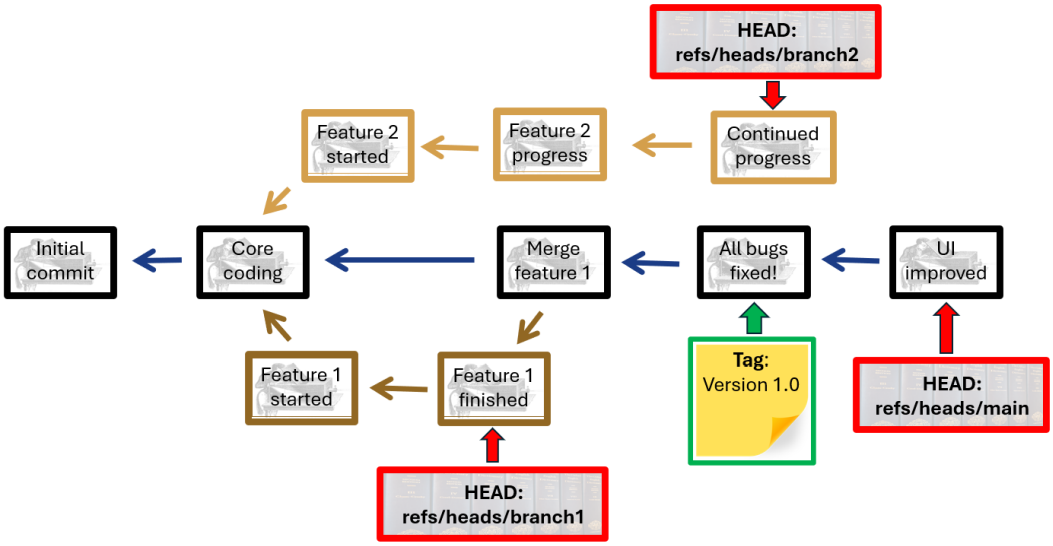

Figure 6.9: Schematic view of a project with HEADS and tags.

Second: you might want to have some extra mechanism for pointing at specific object in your project's history – perhaps the commit you want to correspond to version 1.0 of a code release, or the first submission of a paper and then the finalized revision after you get the referee reports sorted out – and git provides "tags" for this purpose. Basically, a tag is just a time-stamped message that points to a specific commit. The storage format is similar to that of a commit object; technically tags can point at anything (important blobs, or important trees), but those use-cases are probably not something you need to worry about right now. I certainly don't.

A simplified view of a remote with a few branches and a tag might look something like in Fig. 6.9.

Finally, there are remote references. These contain the object ID of the HEAD of the various branches on your remote(s) *the last time you communicated with the remote server*. You will probably not really ever need to look at these remote references (which are created in the `.git/refs/remotes/` directory when you set up a remote), but if you do and you want up-to-date information, you'll need to `git fetch` first.

Git and compression

Git stores the contents of files in a compressed representation. For for files that have changed, git will periodically do its best to represent these different versions not as totally independent blobs which are mostly the same as each other. Instead it will (schematically) try to store them as a base file and a sequence of minimal changes needed to move between different versions of it[58]. The upshot is that if your repository is mostly just text files and perhaps a few images (as it might be for some code, or when writing a paper), you absolutely do not need to worry about how much space and overhead git uses to implement the model of version control described here. On the other hand, occasional changes to large files – for instance, to videos – can cause a repo to quickly grow in size. At a minimum, every file in your project that you track with git requires both the space for the file itself and for git's compressed blob representation of it that sits in the `.git/` directory. For text files this is a trivial addition, but for already-compressed video formats this might roughly double the amount of storage space you are using.

[58] For full details, you can read about git's packfiles.

Chapter 7

Blueprints for a computational research project

Organizing successful scientific projects often requires planning on three different levels: there are practical nuts-and-bolts decisions to be made about the organization of its files, there are choices that must be made about the core tools you will use and the algorithms you will employ, and there is planning to be done in structuring the way you will use those tools to analyze data or conduct numerical experiments. Oftentimes all of these aspects organically grow over time with the project, but it is extremely helpful to think through each of them even before you write a single line of code.

The first two sections below will be extremely practical: what are the nuts and bolts of how to organize the files in your project, and how can you set up scripts that will actually run your code in a reproducible way? This material is not complicated, but it is often simply not mentioned at all. The last section will be somewhat more general: if the recurring theme of this course involves the composition of algorithms and data structures to solve problems, what are the kinds of criteria should we actually use to choose our algorithms?

7.1 Organizing your project

For many of the projects in this course – and in your own research! – you will be developing a core set of Julia code, running it with various scripts, generating data, and then producing plots or other results from that data. A scientific paper may or may not ensue. Given this common pattern, it makes sense to organize each project in a broadly similar way. By doing so, a collaborator (or a future version of yourself, after you have forgotten all of the nitty gritty details) should be able to quickly figure out what you did in your project, and why you made those choices.

7.1.1 File organization

Some of the details will, of course, be particular to your project and the tools that you use in it, but most of it will not be. A structure[59] that handles most reasonable scientific workflows often looks something like code block 7.1.

```
MyResearchProject/
├── data/
│   ├── README.md
│   └── awesomeData.h5
├── plots/
│   └── coolPlot.png
├── research/
│   ├── README.md
│   └── interestingDerivation.tex
├── scripts/
│   ├── README.md
│   └── runSimulation.jl
├── src/
│   ├── MyResearchProject.jl
│   └── (other source files, like greatFunctions.jl, etc.)
├── .gitignore
├── Manifest.toml
├── Project.toml
└── README.md
```

Code block 7.1: A sample structure for organizing a scientific project

Let's break down what's going on here. First, the directory itself has a sensible name, allowing me to identify the purpose of the directory from the command line / a file explorer. The `src/` directory is where the core logic of the project lives. For a Julia project this will contain the source files (with extension `.jl`) that define your primary modules, types, and functions. We'll see in Chapter 5 that Julia's package manager can generate this directory (and some of the files in the root directory that we'll talk about in a moment) for you.

The `data/` directory is... well, a dedicated directory for the data associated with the project. You should *always save the raw data*, but sometimes extremely large datasets associated with a project might not fit nicely into this tidy directory structure (and, for instance, if you're hosting the repository on GitHub the files might just be too big). Processed data, and data that can be directly used to generate plots, should probably live here.

[59] Just one of many, of course. For reasonable alternatives that span a range of disciplines you might see, for instance, this guide to organizing projects in computational biology, or this generic git template, or the default project structure suggested by the DrWatson Julia package.

Do not version control large data files!

Large data files should not be tracked by Git. Version control is for code and text, and not binary blobs of data. Adding such files will bloat your repository, making it slow and difficult to work with. Thus, your `.gitignore` file should include entries that ignore large data files; a `README.md` file inside the `data/` directory the correct place to document the origin and current location of your data.

While `src/` contains the reusable, library-like code, the `scripts/` directory holds the "executable" scripts that *use* that code to perform specific tasks. A script might run a simulation with a specific set of parameters, or take processed data and generate a plot for a paper. As a rule of thumb: if it's executable code that *generates* another file on your computer, you should probably think of it as a script.

The `plots/` directory is pretty self-explanatory: it's a dedicated place for figures and other visualizations that you generate. Like data, it is often reasonable to think of these as *products of your code* and hence can be safely ignored by Git. The scripts that access the data in the directories described above should be trivially able to regenerate plots whenever you want.

The `research/` directory is where I like to keep notes, relevant derivations, to-do lists, related papers, and so on. Think of it as a digital lab notebook for the project, containing the messy, exploratory work that might one day inform a more polished manuscript – I want a record of all of this work, but I don't necessarily think all of it will end up being core to the project. If the scope of the project is *extremely* clear – that is, there is no way that the project corresponds to anything other than exactly one paper – I sometimes add a 'paper/' directory with all of the LaTeX, bibliographic information, and final figures. It is rarely so clear, and I almost always just have a separate repository for the paper.

At the root of the project's directory there are some configuration files and some documentation. We'll learn more about the `Project.toml` and `Manifest.toml` in Section 5.2, and these are specific to Julia[60]. There is of course a `.gitignore` file, which tells Git which files and directories to ignore.

Finally, notice how this and, in fact, almost all of the directories have a `README.md` file. In the root of the directory this is the "front page" of your project, and should explain what the project is, how to install any dependencies, how to run the code, and so one. GitHub is set up so that a readme file will be presented with nice markdown formatting if one is in any subdirectory, though. Since having a human-readable summary is nice when navigating online, it's worth learning a little bit of GitHub flavored markdown for this task.

The key principle in this project structure is a separation of concerns. Source code is separate from the scripts that run it; scripts are separate from the data they produce and the plots that visualize it.

7.2 Planning and executing computational experiments

The `scripts/` directory is where the logic of your code meets reality. What set of parameters should I sweep over in order to solve my problem, and how large is the set of parameters that

[60] If this had been a C++ project, there might instead be a `CMakeLists.txt` file here, for instance.

I actually have the resources to sweep over? Is one of those sets larger than the other? How should you call the code you wrote in order to perform numerical experiments that are not only robust but also *reproducible*?

7.2.1 Fermi estimation for your code

At the risk of being overly explicit by doing arithmetic in front of you – what follows is going to be both extremely simple *and* extremely powerful at a practical level – let's actually connect the idea of algorithmic complexity to the practice of designing numerical experiments.

Suppose we're running a simulation of the stars in a galaxy, and we realize that to evolve the motion of $N = 10^3$ stars forward in time for the equivalent of 7 year takes about 3.3 seconds. We've also run some small tests on our code and have found that its runtime scales *quadratically* with the number of stars – that is, the time taken is proportional to N^2 – a scaling that is common for direct implementations of algorithms that involve checking all pairs of interactions. As we will formalize in Section 7.3, this scaling behavior is usually denoted using "Big-O" notation and written $O(N^2)$; for now, all we need is the observed quadratic relationship to make some sensible estimates.

So, with what we have so far – and assuming the runtime scales *linearly* with the forward simulation time – how long will it take to simulate the motion of $N = 10^6$ stars[61] for the equivalent of 100 years? Our first estimate should be approximately $(3.3\text{s}) \times (100/7) \times (10^6/10^3)^2 \approx 4.714 \times 10^7$ seconds (i.e., about a year and a half). Do you have that much computer time available (and do you need to graduate before then?)? Is a simulation covering 100 years even close to what it takes to answer your question, and do you need to find a different way of tackling your problem?

Those are the kinds of questions whose answer depends on the science, on your resources, and on what you can actually code up. But making these back-of-the-envelope estimates of how long it will take to do something given the tools you already have is both a crucial skill and is independent of those answers. In addition to using these estimates to figure out whether you can, e.g., get something done by a certain deadline or given a certain amount of resources, another place these estimates always come up is in determining parameter sweeps. Perhaps you want to study how some material behaves as you control density and temperature. Even if you know what range of density and temperature you want your simulations to span, how many simulations should you actually launch? Can you estimate how long each simulation will take to finish based on the asymptotic scaling of the algorithms and what you know about how long you need each simulation to run for? Combining those estimates, and thinking about whether you will have each simulation run independently of the others (as, for, instance, if you were just planning on simulating the parameters on a regularly spaced grid between the endpoints you already decided on) or not (as, for instance, if you want to use information from one simulation to help decide what point in parameter space would give you the most information), is a bread-and-butter part of computational research.

[61] Which could, perhaps, be a reasonable estimate for some dwarf galaxies.

7.2.2 Designing scripts for reproducibility

We often want to write functions that we will call many times as we vary key parameters of some physical model. There is a subtle danger in the fact that Julia is a language in which it is easy to write both robust, performant code and disposable interactive scripts. Namely: it is so easy to iterate and quickly generate results, that it arguably requires *more* care to generate reproducible results.

This can be seen by contrasting with a compiled language like C++. In such languages there is a clear separation between being in the phase of working on code and being in a phase of *using* that code: the clear separation is generated by needing to compile the source code into an executable. Thus, a natural workflow involves working on the code itself until it compiles and functions correctly, making sure the program can accept command-line arguments for the parameters of interest, and then writing a script in some other language – bash or python, perhaps – that calls that code repeatedly across the array of parameters. This leaves behind multiple artifacts that can be version controlled and referred to later. Do you want to reproduce exactly the results of some paper? Go into the git repo, checkout the commit where you checked in the calling script (which will ideally also restore the codebase to its state when that script was run!), and then run that script.

The fact that Julia's workflow is so fluid means that we – the programmers! – must provide the discipline that a compile-step enforces in other languages. Let's walk through a progression of different patterns we might use to work with Julia in the context of scientific computations. Each pattern might be reasonable in different contexts, but we'll see what kinds of problems and pitfalls each might entail. All of this will use, as an example, the Julia file in code block 7.2, which defines a simple `estimatePi` function that takes two parameters.

```
# pi_estimation.jl
# (Our Monte Carlo estimate_pi function from before)
using Random
using Statistics
generate_points(n,L) = [(rand(Float64,2) .*L .-L/2) for i in 1:n]
in_unit_circle(point) = sum(point .* point) < 1.
function unit_circle_proportion(points)
    return count(in_unit_circle,points)/length(points)
end
function estimate_pi(n,trials)
    data = [ 4 * unit_circle_proportion(generate_points(n,2))
             for _ in 1:trials]
    return (mean(data),var(data))
end
```

Code block 7.2: The beginning of a script containing functions for a Monte Carlo estimation of π (see Section 4.3).

Workflow 1: The interactive session

The REPL is an essential, powerful tool for exploration, prototyping, and debugging. The challenge is in capturing the final, successful version of the process we followed in permanent form.

Consider the following sequence of events. The very first thing we might find ourselves doing is in the spirit of the `Revise`-based workflow[62] we discussed in Chapter 1. Earlier in our Julia session we had done[63]

```
julia> includet("pi_estimation.jl")
```

As soon as we were happy with the state of the functions we could start generating data, for instance

```
julia> first_estimates=estimate_pi(10000,15);
```

Do that a few times, and we're already able to start generating a plot with data to use!

All of which is to say: in a language like Julia, the transition from writing code to testing it to generating data is much blurrier. The instant you think you have some working code you might start calling the relevant functions from the REPL, saving data, and so on. It feels amazing, but how are you going to instruct someone else to reproduce that data? Do you remember exactly what you typed into the REPL? Are you sure that you didn't revise a function, run something in the REPL, revise a function again, and run the same thing in the REPL, and then undo that revision before committing to the git repo?

To be explicit, a REPL-driven workflow, for all of its power, presents two major challenges to scientific reproducibility: *provenance* (what is the exact state of the code that produced a result) and *history* (what were the exact sequence of commands and parameters that were run).

Workflow 2: The self-contained script

A crucial first step that solves these problems is to create a dedicated script whose sole purpose is to run our experiment. After we create it, we add it (and the rest of the state of our codebase) as a new commit to our repository. This gives us a permanent, version-controlled artifact that records exactly the computation that was performed. It might look something like this:

```
#run_single_pi_estimation.jl
include("pi_estimation.jl")
n_points = 10000
n_trials = 10

mean_pi, variance_pi = estimate_pi(n_points,n_trials)
println("Parameters: N=$n_points, Trials=$n_trials")
println("Pi estimate: $mean_pi (variance: $variance_pi)")
```

[62] Even before this, I suppose, we might define *all of our functions* directly in the REPL. Let's not do that.

[63] Or, even better, we had written a module for our whole π estimation project, and we were `using` that module.

Here we've written a script that prints the output to the screen – in a real script you would, of course, save the data to a file. To use this script we just call it from the command line:

```
$ julia run_single_pi_estimation.jl
```

This is already an improvement over the REPL-based approach, as it leaves us with a permanent record of what we did.

On the other hand, it is true that every time we want to run our experiment with different parameters we have to go in and edit this file. We can avoid this – at the cost of potentially losing out on some of the provenance of our results – passing parameters from the command line. Julia makes this easy: when you call Julia with command line arguments those arguments populate a global `ARGS` constant, and one can then parse this constant to extract information[64].

```
#run_and_save_pi_estimation.jl
include("pi_estimation.jl")
n_points = 10000
n_trials = 10
seed = 1234 # a poor default

#fragile parsing of command line arguments
if length(ARGS) >=1
    seed = parse(Int,ARGS[1])
end
Random.seed!(seed)

mean_pi, variance_pi = estimate_pi(n_points,n_trials)
filename = "./pi_N$(n_points)_T$(n_trials)_seed$(seed).txt"
open(filename, "w") do f
    write(f, "$mean_pi, $variance_pi")
end
```

Code block 7.3: A script with very simple command-line-argument parsing.

Let's use this to solve *another* problem with the above script, which is that it is not actually reproducible at all! It makes use of Julia's RNG, and if you have your friend clone your git repo and run the script from the precise commit in question, they will *not* get the same results as you[65]. We'll learn more about this in Module IV, but for now let's tentative content ourselves with thinking that by setting a "seed" – an initial value of some sort – we'll ensure a reproducible, deterministic sequence of random numbers. Our revised script may now look like code block 7.3.

[64] If you find yourself parsing command line arguments frequently, consider using dedicated packages (e.g., `ArgParse.jl`) to handle many of the details for you.

[65] Barring, of course, a coincidence of astronomical unlikelihood.

This embodies a few best ("reasonable") practices. It explicitly seeds the random number generator, ensuring that anyone that runs it with the same seed will get the same numerical result. It can be run with default parameters:

```
$ julia run_and_save_pi_estimation.jl
```

Those parameters can be overridden from the command line:

```
$ julia run_and_save_pi_estimation.jl 9823475
```

It then saves the output to a systematically named file, so that from the filename itself we can reconstruct what parameters were used to generate it. That is, even if we extended the command line argument parsing to include passing in the other parameters of our experiment – `nPoints` and `nTrials` – and then ran everything from the command line, we would still have a record of what experiments were actually run.

Workflow 3: The parallel-execution strategy

What we have above is already quite powerful. It would also be quite tedious – typing in variations of command line arguments over and over is both tiresome and error-prone. We could of course just hard-code the complete set of parameters we want to sweep over, but we have still limited ourselves to a fundamentally *serial* execution of our experiments. For our example of estimating *pi*, each experiment is independent of the others, but we still have to wait for each one to finish before the next begins. On modern, multi-core computers this leaves a huge amount of computational resources we could be exploiting on the table.

Fortunately, Julia has *fantastic*, built-in support for multi-threading. A common pattern would be like in code block 7.4, where we use this built-in support to perform our parameter sweep for us. The only thing we need to tell it from the command line is how many threads to use, for instance like this:

```
$ julia --threads=4 run_pi_parameter_sweep.jl
```

This is incredibly powerful.

By adding the `@threads` macro to our `for` loop, Julia automatically divides the work among the available threads[66]. This is a form of *data parallelism*, which is safe in this example because each iteration of our loop is completely independent – each run works on its own parameters and saves results to a unique file.

[66] If, on the other hand, we set the `threads` option to 1 (or if we just don't specify it at all) Julia understands that the number of threads available is 1 and then happily returns to running the loop serially for us.

Parallel programming

The simplicity of `@threads` is deceptive, and it should *only* be used in loops where the iterations are truly independent. If they are not, you will encounter a *race condition*: the output of your program will depend on the unpredictable order in which multiple threads or processes access and modify shared data. This can lead to silently corrupted data, incorrect results, and non-reproducible errors that are extremely difficult to track down. We're using `@threads` in a carefully controlled, safe context here. True parallel programming requires much more advanced patterns.

Another caveat to using `@threads` is its scheduling behavior. The *scheduler* determines how the loop's iterations are assigned to available threads, and the default[67] breaks the work into small chunks and uses a shared queue. This is a good, all-around choice that keeps threads busy even if the work per iteration is unbalanced. Note, though, that the order in which these small chunks are processed is not guaranteed. Two main alternatives exist. If you want slightly stronger guarantees about which thread processes which iteration you can use the `:static` scheduler (`@threads :static for...`), which evenly divides the iteration space and gives each thread a large continuous chunk of work to do. For workloads with high variability from one iteration to the next, the `:greedy` scheduler can be good: as soon as a thread is free it just grabs whatever the next available iteration happens to be.

Relative paths for files

Another best practice embodied in code block 7.4 is the use of *relative paths* when saving files[a] – you can see this in the dot slash at the start of `./data/...`. Avoid hard-coding directory paths if you can, so that your script will work regardless of what computer it is run on. Note, by the way, that you can use the `mkpath()` function to safely create directories so that you don't have to worry about whether they already exist

[a] While building path strings works well, it doesn't always play nicely across operating systems. Julia has a built-in `joinpath()` function we can use, like so:

```
filepath=joinpath(".","data","pi_N$(nPoints)_T$(nTrials)_seed$(seed).txt")
```

This automatically uses the correct path separator ("/" or "\") for the operating system.

Workflow 4: The managed project

Honestly, the pattern demonstrated in code block 7.4 is what I've used for the majority of my own computational projects over the years – not necessarily in Julia, but one can easily build up similar patterns with, e.g., bash scripts for running an executable program that might or might not take command line arguments. Often, as you write scripts like this over an over, you can find your self building progressively more intricate functions of convenience: perhaps a function that will nicely generate filenames for saving data for any number of parameters you

[67] Which currently corresponds to typing out `@threads :dynamic for...`

```julia
#run_pi_parameter_sweep.jl
using Base.Threads
include("pi_estimation.jl")

# The grid of parameters we want to explore
parameter_grid = [(1000, 100), (5000, 100), (10000, 50),
             (20000, 50), (50000, 20), (100000, 20)]

@threads for (n_points, n_trials) in parameter_grid
    # A simple way to get a unique seed per run
    seed = n_points + n_trials
    Random.seed!(seed)
    mean_pi, variance_pi = estimate_pi(n_points, n_trials)

    #Save results
    filename = "./data/pi_N$(n_points)_T$(n_trials)_seed$(seed).txt"
    open(filename, "w") do f
        write(f, "$mean_pi, $variance_pi")
    end
end
```

Code block 7.4: A script that loops over a grid of parameters, using the number of threads available to potentially run iterations of the loop in parallel.

want to vary, or routines that perform a simulation only if the output of that simulation doesn't already exist[68], and so on.

There is also a kind of fragility to these kinds of scripts. The way we parse the command line arguments is extremely fragile, requiring us to invoke all commands in exactly the right order. We have to manually create the `./data/` directory *before* we run our script, otherwise it will fail (perhaps in one of the worst ways: running all of the expensive computations but then silently not saving any of the data because the path to the target file doesn't exist). All of these specific problems are, in fact, easily solvable in Julia. But the broader point is that, in fact, this entire broader class of problem is not only solvable but *already solved* – we could solve them, but this is just creeping up to the edge of re-inventing the wheel.

The specific solution you want to use here is often language (and sometimes domain) specific, so I don't want to dwell overly long on the details of using what are known as *scientific workflow systems*. It's more important simply to know that they exist; when your project workflow starts becoming long enough, or you find yourself starting to write a lot of boilerplate code in the scripts that are running your main code for you, that's a sign you should investigate more. In the Julia ecosystem, a commonly recommended tool is the "scientific project assistant," `DrWatson.jl` package. It is designed to solve exactly these problems of fragile, boilerplate code: it provides standardized functions for accessing data paths, running files from parameters, and

[68] Very handy, e.g., for dealing with the occasional dropped job when running thousands of simulations on a cluster!

safely running experiments, all while integrating with the reproducible project structure we discussed earlier.

7.3 Algorithms and the logical architecture of a project

"The `src/` directory is where the core logic of the project lives." But even before we write a single line of code in `MyResearchProject.jl`, we need to think about one of the most consequential decisions in our project's lifecycle: what set of algorithms we will use to accomplish our goals. It is often the case that there are multiple ways to *numerically* implement the solution to those goals – perhaps a "brute force" method along with an increasingly slick set of "clever" methods. Where should we start, and how can we be quantitative about making this decision? How should we weigh the pros and cons of using "better" algorithms with things like the amount of time it will take us to code them, or the number of bugs per line of code we expect to have lurking in our project?

And, honestly, even before we think about the algorithms, we should think about the overall *architecture* of our code. By this I mean: how are the data needed in different parts of the program grouped together? How do different functions and weave together to operate on those groupings? Every program has *some* organization, even if it is just "everything is a global variable and there is a single monolithic function that does everything."

I prefer not to talk about architecture in the abstract – I think it is clearer as we get to designing programs that are meant to solve specific problems – so I'll leave most of this discussion to future chapters. For now, let's agree to think about good architecture as being predicated on change[69]. If nobody, including yourself, is ever going to change the undoubtedly perfect code being written, architecture and design is largely irrelevant. But if a program is ever intended to evolve – perhaps your physical simulation of particles should be adapted to a more stable integration scheme, or perhaps you want to be able to solve fluid flow given different types of boundary conditions – good architecture makes that easy and bad architecture usually means you convince yourself to just start over from scratch.

7.3.1 Algorithmic complexity

Time (how long it will take to run an algorithm given some data) and *space* (how much memory the computer will need) are two of the most fundamental computational resources we need to think about. Occasionally it is useful to think of these quantities in absolute terms – how many days will it take once I start this code for it to finish? More often, though, we don't really care about those questions, as they are too dependent on all of the details[70]. It is often better to ask how a particular algorithm *scales* with the size of the problem. For instance, if you are simulating a galaxy with N stars and then decide you want to simulate twice as many, how much harder does the problem get? Will the solution to a system of $2N$ stars take twice as long?

[69] "For me, good design means that when I make a change, it's as if the entire program was crafted in anticipation of it. I can solve a task with just a few choice function calls that slot in perfectly, leaving not the slightest ripple on the placid surface of the code" – Robert Nystrom, from Ref. [19].

[70] What CPU does your computer have? What other processes might be running at the same time, sharing both CPU cores and RAM?

Four times as long? Exponentially longer? Similarly, we can ask about *space complexity*: will the simulation require twice as much memory, or will the memory requirements grow more steeply? In the following I'll focus on the time complexity, but the memory footprint of a chosen approach can be equally important.

This is just an introduction!

The field of algorithmic analysis is a deep, fascinating discipline in its own right! It would be impossible to provide a comprehensive treatment of the subject in just the few pages I've written here, and that is not my goal. Instead, I will introduce the essential concepts and the vocabulary that will let us reason about our choices and make informed decisions as practicing scientists.

In order to explain the answer to such questions, it is common to use asymptotic notation. If we have some function which captures, say, the running time of an algorithm as a function of the "problem size" (for instance, the number of stars we are simulating, or the number of elements of a list we want to sort, or...), $f(N)$, we might write:

$$f(N) = O(g(N)).$$

This means that *in the limit* where $N \to \infty$, the ratio $f(N)/g(N)$ is finite[71]. Most commonly the function g will things like a constant ($O(1)$), a log ($O(\log N)$), a polynomial ($O(N^p)$), or an exponential ($O(2^N)$).

This "big-O" notation is all about the asymptotic behavior of an algorithm, *and* there is an important hidden prefactor (the precise ratio between f and g): it could be the case that for a particular problem size an $O(1)$ algorithm is actually slower than an $O(N^2)$ algorithm. It is often also important to distinguish between the asymptotic behavior of an algorithm given "typical" or "average-case" data to work on, vs the asymptotic behavior of an algorithm in the worst possible case. But even with all of these caveats, understanding the asymptotic behavior of algorithms is a crucial first step. If you are working with an $O(N)$ solution to your problem, you can be confident that if you later realize you need to work with a problem which is an order of magnitude larger you'll be okay. On the other hand, if your solution to the problem is $O(2^N)$ and you have the same realization...well, I hope you're prepared to be a student for an extremely long time!

As a final comment: time and space are just two of many resources that may be important to think about. Modern computers – not only supercomputers but also consumer laptops – have multiple processing cores and specialized GPUs that often offer thousands of simple cores. Thus, in modern scientific computing, the potential for an algorithm to be *parallelized* is increasingly important. The potential speedup from parallelization is not infinite, and depending on the structure of the problem can sometimes be much less than expected. Amdahl's Law – the seemingly obvious point that the overall performance gained by optimizing a part of a program is limited by the amount that part gets used – can be written as

$$S = \left((1-p) + p/a\right)^{-1},$$

[71] To be more precise, Big-O provides an *asymptotic upper bound*. Computer scientists also have different notation for, e.g., *tight bounds* ($\Theta(g(N))$) – where $f(N)$ is bounded both above and below by $g(N)$ – and lower bounds ($\Omega(g(N))$).

where S is the amount the program execution is sped up, p is the proportion of time the program spends in the section of code that is being optimized, and a is the factor by which the optimized section is accelerated. The quantity S might be less than you expect at first glance. For instance, if in your program you take a section of code that currently uses 75% of the computational time and (working some real magic!) make it 10 times faster, your program will only run 3 times faster than before. Even more extreme: if you take a section of code that currently uses 98% of the computational time and implement a strategy that makes it run 10^10 faster – blazing fast! – your program will still only run 50 times faster than before.

These arguments apply to speeding up serial sections of code, and they equally apply to accelerating a section of of a program by employing parallel computing resources. Because of these constraints, the architectural choice of an algorithm need not just be a trade-off between $O(N^2)$ and $O(N \log N)$; it can also be a trade-off between clever-but-sequential algorithm and a simpler one that can more effectively harness the power of parallel hardware. We'll explore these concepts – and how to think about what parts of your code are actually worth optimizing – in more detail later in the course, but it is an important part of the modern computational landscape to have in your head from the beginning.

7.3.2 Performance vs. Simplicity

When deciding which algorithm to use, there is often an important trade-off between performance and simplicity. By simplicity I often mean something about how complex the algorithm is to implement[72], and also how easy it is to establish the correctness of that implementation. The performance of an algorithm is often reasoned about in terms of its asymptotic scaling, although the constant factor hidden by the Big-O notation can be quite important. That constant factor is, itself, often directly correlated with the simplicity of the algorithm; depending on the problem size you are interested in, it might be absolutely crucial.

As an example of a problem where these considerations come up, consider the multiplication of two $N \times N$ matrices. I'm choosing this example because matrix multiplication is not just a textbook exercise; it is a fundamental operation at the heart of countless computational methods. A simpler example – perhaps different algorithms to sort a list – could be used to make the same point, but matrix multiplication is more interesting, and there are *open questions* at the cutting edge of algorithmic research about how fast matrix multiplication can actually be!

The simplest matrix multiplication algorithm is one that directly implements what you probably think of as the *definition* of matrix multiplication. As I'm sure I don't need to tell you, given matrices A and B, the i, j element of $\mathbf{C} = \mathbf{AB}$ is

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}.$$

Said another way, the i, j element of the product is the dot product of the ith row of $\mathbf{A}$ with the jth column of $\mathbf{B}$.

Assuming that the matrices in question don't have any special structure (they aren't the identity matrix, they aren't extremely sparse, etc), it is straightforward to analyze the asymptotic

[72] With the caveat that some programming languages or libraries make it quite simple to *use* algorithms that would be horrible to write from scratch.

scaling of this approach to matrix multiplication: For every element in **C** we compute the dot product, which involves N multiplications and $N-1$ additions. Since there are N^2 elements to compute this for, we confidently say that naive matrix multiplication is $O(N^3)$. Implementing the algorithm corresponding to this is straightforward, and the correctness of that algorithm is easy to verify (and, indeed, extend to matrices whose shape is not square).

If you haven't heard of Strassen's algorithm [20] you might have concluded that matrix multiplcation *itself* is $O(N^3)$. Consider this product of 2×2 matrices:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.$$

This is standard matrix multiplication, and we can see that it involves 8 total multiplications and 4 total additions – exactly as we computed above. Of course, I would have written down exactly the same formula if each of the elements above was itself a block matrix – in which case this is an equation with 8 total *matrix* multiplications and 4 total *matrix* additions.

Let's go ahead and define the following seven objects (they might be scalars, or they might themselves be block matrices):

$$\begin{aligned} I &= (A_{11} + A_{22})(B_{11} + B_{22}) & V &= (A_{11} + A_{12})B_{22} \\ II &= (A_{21} + A_{22})B_{11} & VI &= (A_{21} - A_{11})(B_{11} + B_{22}) \\ III &= A_{11}(B_{12} - B_{22}) & VII &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ IV &= A_{22}(B_{21} - B_{11}) \end{aligned}$$

In terms of these, we can write the original matrix product as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} I + IV - V + VII & III + V \\ II + IV & I - II + III + VI \end{pmatrix}.$$

We have turned 8 total multiplications and four total additions into 7 total multiplications and 18 total additions. If these were scalars that would be a net loss, but for matrices it could be a massive win – naive matrix multiplication is $O(N^3)$ and matrix addition is $O(N^2)$, so reducing the number of matrix multiplications involved is great! The real trick comes from the idea of *recursively* applying this block decomposition over and over again; in the asymptotic limit this approach to multiplying matrices has complexity $O(N^{\log_2 7})$. This might not seem like much, but as $N \to \infty$ it is a huge win[73].

Clearly, though, actually implementing Strassen's approach is much more complex than implementing naive matrix multiplication, it requires extra memory to store the intermediate matrices, and for small matrices it actually involves *more* arithmetic operations! Thus, in this case there is a crossover in matrix scale beyond which Strassen's approach is faster and below which it is actually slower; the precise value of the crossover (usually estimated to be for dense matrices with thousands of rows and columns) is also a function of the *hardware* the algorithms are run on. The general lesson is that the choice of algorithm you employ should depend on the scale of the problem you intend to solve (perhaps with some consideration given to the *range of scales* you might ever consider).

[73]Strassen's approach is not even the best known asymptotic algorithm! At the moment the bound on how matrix multiplication intrinsically scales is bounded by something like $O(N^{2.371552})$ [21].

7.4 Coda

The three components of project architecture discussed above are linked together. The *static* organization of our files – hopefully straightforward – provides a clean, version controlled home for our entire project. The *execution* architecture of our scripts determines how we carry out the reproducible process of our scientific explorations. Together, these two components govern the projects "health" – they determine how maintainable, extensible, and reproducible our project will be (and whether we will be able to understand any of it a week or a month after we work on it).

It is the *logical* architecture – the algorithms we choose and the way we weave them together – that casts the longest shadow. An inefficient algorithm cannot be saved by a tidy file structure, and a fundamentally slow or unstable numerical method cannot be fixed by a clever execution script. The logical design sets the hard limits on what is computationally feasible; this leads to some core principles for project planning.

Project planning for computational projects

1. **Identify the computational core:** In most projects, one or just a few parts of the code consumer the majority of the computational time – this might be the pairwise calculation of all forces, or the need for matrix inversion, or some geometric calculation on your data structures. Identify the "hot loops" first; your project's scope will be determined by how efficient the computational bottlenecks are.

2. **Design for change:** Your first idea is rarely your best, so build your code with a modular, high-level API that allows you to swap out the core components without having to demolish the entirety of your code.

3. **Optimize what matters:** Use Amdahl's Law as your guide. Before you spend a week optimizing a function, *measure its impact*. A 10x speedup on a part of the code that only accounts for 1% of the runtime is probably a waste. Not of your computer's resources, but of your own time.

Module II

Module 2: ODEs and molecular dynamics

From the "clockwork universe" evolution of the planets to the chaotic motion of molecules in a fluid, our understanding of how systems change over time is encoded in the language of ordinary differential equations (ODEs). The foundational example is Newton's second law, $\mathbf{F} = m\mathbf{a}$, a deceptively simple equation that predicts the behavior of extraordinarily complex systems. However – for all but the most idealized setups there are no analytic solutions to the kinds of ODEs we typically encounter. We turn, instead, to the computer

Figure II.1: A 2007 recreation of the Antikythera mechanism – the oldest known analogue computer – which could be used to predict eclipses and the motion of various celestial bodies [22]. Mogi Vicentini, CC Attribution 2.5 generic.

In this module we'll use the classical "N-body" problem – in which we try to predict the motion of N classical point particles interacting via a conservative potential – as our paradigmatic example. We'll explore different numerical methods, from robust "black-box" solvers to specialized algorithms that leverage physical symmetries to achieve remarkable long-term stability. This contrast will reveal an important lesson: the best numerical method is *not* always the most mathematically accurate, but rather the one that respects the underlying structure of the governing physical laws.

With the N-body problem as our target, this module will be where some of the abstract ideas in Modules 0 and I become concrete[74]. "Weaving together algorithms and data structures[75]" will no longer a theoretical comment in the context of toy problems, but a practical necessity for designing our code. *Testing* will not just be about verifying simple functions, but about verifying physical conservation laws in complex systems. How can we put things together so that we can apply the right tools to the right problems? We'll think about all of this as we build simulations that are not only physically correct, but also robust, flexible, and clear.

For a deeper dive into the methods and physics discussed in this module, consider the following references [23, 24, 5]

[74] Does that count as a pun in the context of Julia?

[75] Has anyone been keeping track of how often I've said that during lectures?

Chapter 8

Ordinary differential equations

We are often interested in the time-evolution of some physical system – perhaps the trajectory that the planets will trace out over the course of a year, or how defects in a crystal will move, or how populations of competing species evolve. The central, paradigmatic challenge is to solve the *initial value problem*: We are given a set of first-order ODEs that describe how our system evolves,

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

along with the initial state of the system at a reference time, $\mathbf{y}(t_0)$. From this, we want to find the trajectory, $\mathbf{y}(t)$ for $t > t_0$.

The state vector $\mathbf{y}$ is, conceptually, a list of numbers that completely specifies the configuration of the system at any given moment. For a single point particle in one dimension, the state vector is simply the position and velocity, $\mathbf{y} = \{x, v\}$. For a molecular dynamics simulation of N particles in 3D, the state vector grows to a list of all positions and velocities – a list with $6N$ components.

The standard technique for handling higher-order equations – like Newton's law – is to recast them as a larger system of coupled first-order equations. For instance, as you already know, we can recast Newton's second law for a particle evolving in one dimension like so:

$$\ddot{x} = F/m \quad \longrightarrow \quad \dot{\mathbf{y}} \equiv \begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v \\ F/m \end{pmatrix} = \mathbf{f}(\mathbf{y}, t)$$

To keep the derivations in this chapter clean, we will often discuss and analyze our numerical methods using a single scalar equation, $\dot{y} = f(y)$. This is just for convenience – the same methods apply directly to the large state vectors that describe the complex physical systems we often care about.

8.1 The naive solution: Euler's Method

You will have already seen this approach in your previous computational methods class, but let's ramp up to this chapter by reminding ourselves what the simplest approach to time-discretized solutions to ODEs could be. The essential idea is to convert from a continuous-time representation of the problem to one in which the system evolves forward via small *discrete* timesteps

of size Δt. We can derive this method directly from a first-order Taylor series expansion of the state at time $t + \Delta t$:

$$\mathbf{y}(t + \Delta t) - \mathbf{y}(t) = \Delta t \cdot \dot{\mathbf{y}}(t) + \mathcal{O}(\Delta t^2) = \Delta t \cdot \mathbf{f}(\mathbf{y}(t), t) + \mathcal{O}(\Delta t^2).$$

Euler's method involves just stopping here, truncating the series and ignoring the terms of order Δt^2 and higher. This generates a *local truncation error* – the amount of error we make during each small step in our approach. We simply accept this, and use the above equation as the core iterative update rule that lets us propagate our system arbitrarily far forward in time. Adopting the common notation where $t_n \equiv t_0 + n\Delta t$ and $\mathbf{y}_n \equiv \mathbf{y}(t_n)$, the *Forward Euler Method* [25] is

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n). \tag{8.1}$$

This is the simplest possible numerical integrator. To see its characteristic flaws, let's apply it to a classical problem in celestial mechanics: simulating the time evolution of our solar system.

8.2 Case study: N-body simulations and planetary dynamics

Let's build a simulation of our very own Solar System. We will treat the planets and the Sun as a classical "N-body" problem, with point particles interacting according to some pairwise potential, in this case Newton's law of gravitation. Presumably we will find that the planets will trace out periodic orbits as they continue the celestial waltz that they've done for ages untold[76].

8.2.1 First attempt: A monolithic script

Our first instinct might be to write the most direct script possible – some highly specific "Solar System simulator." What might that look like? Code block 8.1 is one version. It defines the data as a simple list of lists, and has a few nested loops: an outer loop for each time step, during which we loop over objects in the solar system to compute the relevant accelerations (the function $\mathbf{f}(\mathbf{y}_n, t_n)$ in the notation above), and then perform another loop to perform a forward Euler update. This results in a short, self-contained piece of code.

This code certainly has it's merits: it is short, linear, and fairly easy to read and reason about from top to bottom. And, not forgetting that our own time is important, it can be written very quickly. On the other hand, the simplicity of this code might be deceptive. If the script actually contains everything we will ever want to do with the solar system we might be okay, but what happens when we want to change or improve it? For instance, what if[77] it turns out that Euler's method is not that good, and we want to be able to swap in a better numerical integration scheme? Or what if we want to reuse our code to simulate a similar system but with a different force law?

Unfortunately, the simple script above has a design that is much too brittle. The data and logic of it are inextricably mixed together, with the core loop accessing magic numbers and

[76] Ages untold? Just kidding – they've danced for about 4.6 billion years.

[77] Perish the thought!

```
# Data format (units?):
# SOL = [mass, x, y, z, v_x, v_y, v_z]
# MERCURY = ...
objects=[SOL,MERCURY,VENUS,EARTH,MOON,MARS,JUPITER,SATURN,URANUS,NEPTUNE]
const G = 6.67430 * 1e-11 # m^3 /(kg *s^2)

Δt = 0.0001
for t in 1:1e6
    # find the accelerations
    a = [[0.0,0.0,0.0] for _ in eachindex(objects)]
    for i in eachindex(objects)
        ai = [0.0,0.0,0.0]
        for j in eachindex(objects)
            if i != j
                r = objects[i][2:4] - objects[j][2:4]
                rn = sqrt(r[1]^2 + r[2]^2 + r[3]^2)
                forceij = (G*objects[i][1]*objects[j][1]/(rn^3)) * r
                ai += forceij / objects[i][1]
            end
        end
        a[i] = ai
    end

    # update the system
    for i in eachindex(objects)
        objects[i][2:4] += objects[i][5:7]*Δt
        objects[i][5:7] += a[i]*Δt
    end
end

# IS THIS EVEN CORRECT???????????
```

Code block 8.1: A self-contained Solar System simulator.

mutating global variables. It also clearly lacks any sense of modularity – the physics of gravity and the Eulerian numerical intergration method are mixed together in a single function, even though neither actually needs to know about the other. That lack of modularity also makes it hard to *test*. With such a script, how could we make sure that the calculation of the gravitational force is correct[78]? We cannot – we can basically only test the entire script, making it hard to isolate bugs.

There are any number of other problems with this code – some related to performance, others to brittleness, others to coding conventions – and certainly a better version of this monolithic script could be written without changing its fundamental design. Perhaps a better practice,

[78] By inspection? There's definitely a sign error in the code, right?

though, is to step back and ask ourselves how we could design a more robust, flexible, and verifiable architecture for our simulation in the first place.

8.2.2 A top-down design

One more structured philosophy we could turn to is the practice of *top-down design*. We'll *start* not with small pieces that we hope we can glue together later, but by designing the high-level function that will be our program's API[79]. We'll do that here, by first writing a high-level `run_simulation!` function. The design contract of that API will, in turn, help guide the implementation of all of the lower-level pieces. Code is about communication, and we'll try to see how a well-designed API can communicate the intent and structure of the entire underlying program.

Performance considerations and top-down design

Top-down design can be a powerful approach, but it can have serious drawbacks. Particularly when trying to design high-performance code – a common goal in computational physics – the top-down approach has a potential trap. One might design a beautiful / elegant API that ends up being fundamentally incompatible with the most efficient (or the most parallelizable) implementation of part of the program that might happen to consume the overwhelming majority of the runtime.
One solution is to practice top-down design and then rewrite everything whenever you discover this kind of implementation / performance incompatibility. A solution that involves pulling out less of your hair is sometimes called "yo-yo" or "meet-in-the-middle" design. In that pattern you sketch out the highest-level API, and then immediately jump to the lowest level – at this lowest level you build a prototype of the most computationally expensive part of the program. The performance characteristics at that level suggests some good design, allowing you to jump back to the high-level API and refine it to make sure it supports that efficient implementation. By designing the top and bottom levels in concert – working through all details and eventually meeting in the middle with a completed program – one can find a solution that is elegant *and* performant.

With that design philosophy in mind, let's define the high-level API for our simulation. The goal is to create a function signature that is clean, powerful, and flexible enough to accommodate different physical systems and numerical algorithms. There are multiple good design decisions we could make; code block 8.2 is one concrete proposal both for the function signature and its complete implementation.

Just by looking at the function signature we can read the entire architecture of our program. This most important feature is the clear *separation of concerns*: the API demands that the physical state (`system`), the force calculation logic (the `force_calculator`), and the time-stepping algorithm (the `integrator`) all be provided as distinct, independent objects. This

[79] "Application programming interface" – the contract that a piece of code presents, specifying the functions it has and how they must be used

```
function run_simulation!(
    system, force_calculator, integrator,
    time_step_size, number_of_steps;
    callback = (sys, step) -> nothing,
    callback_interval::Int=1
)
    for i in 1:number_of_steps
        integrate!(system,force_calculator,integrator,time_step_size)
        if i % callback_interval == 0
            callback(system, i)
        end
    end
end
```

Code block 8.2: A high-level API for a classical N-body simulation.

modularity in not an accident – it is a choice that mirrors the structure of the underlying problem. An integrator like Euler's method is a generic mathematical tool, and shouldn't need to know about the physics of gravity. Similarly, force laws are physical principles, independent of what numerical operations they are used for. The API mirrors, but also *enforces* this clean separation.

Top-down design and designing tests

Do you think this strategy of top-down design for APIs and writing modular code makes *writing good tests* for our code easier or harder? Why?

We also see the practical details. The function expects a scale for our discretization of time (`time_step_size`) and a number of iterations to run for (`number_of_steps`)[80]. It also provides two keyword arguments: a `callback` function[81] and a specification of how frequently to run that callback. This gives the user a "hook" into the `run_simulation!` function they can use to perform analysis or save data without having to modify the core simulation loop.

Reading the function body itself confirms the payoff of this design: the resulting code is extremely simple. "Running the simulation" becomes the trivial task of orchestrating the components, because all of the complexity has been abstracted away and encapsulated with the objects that the function calls. This is part of the essence of good design: complexity is not – cannot! – be erased, but it can be isolated to the components that are actually responsible for it.

[80] Perhaps a better design would be to specify Δt and a total duration to integrate forward in time for. However, choosing Δt and `number_of_steps` is pretty typical, and it avoids any danger of accumulating floating-point errors in the loop counter.

[81] A "callback" is a common term for function passed as an argument to another function, with the expectation that it will be "called back" (executed) at a specific time or when a certain event occurs.

8.2.3 Particle data structures

The design contract of our API dictates that we need a stateful `system`. Let's go ahead and build data structures suitable for *any* particle-based simulation.

A very intuitive approach is an "Array of Structs" (AoS) arrangement of our data, where all of the data for a single particle is bundled together:

```
struct Particle{D,T}
    position::SVector{D,T}
    velocity::SVector{D,T}
    mass::T
end
struct System{D,T}
    particles::Vector{Particle{D,T}}
end
```

This pattern is often the most convenient to work with as a programmer – all of the data associated with a given particle is immediately at hand. However, in high-performance computing you will frequently see the alternative "Struct of Arrays" (SoA) pattern:

```
struct System{D,T}
    positions::Vector{SVector{D,T}}
    velocities::Vector{SVector{D,T}}
    masses::Vector{T}
end
```

The SoA pattern often leads to faster code – this is not because of any algorithmic advantage, but rather because it organizes data in a way that is friendly to modern computer hardware. By storing the positions (e.g.) contiguously in memory, it allows for better cache efficiency and often allows the processor to perform operations on multiple data points simultaneously ("vectorization" of operations or "SIMD"). In these notes, for clarity we'll use the AoS pattern.

Given the above definitions, we can implement a custom *constructor* that builds the specific system we're interested in. In this case, we can grab actual data from NASA's remarkable JPL Horizons System[82] I've gone ahead and scraped the data for the mass, position (relative to the solar system's barycenter), and velocity of various celestial objects, and put them in constants. From that, we can build our `System` like so:

[82] An interesting combination of solar system data that includes both historical data on the location of celestial bodies, and also forward computation services.

```
const SOL::Vector{Float64} = [1988410., ...] # truncated for clarity
const MERCURY::Vector{Float64} = [0.3302, ...]
function SolarSystem()
    JPL_DATA = [SOL,MERCURY,...] # truncated for clarity
    solar_bodies::Vector{Particle{3,Float64}} = []
    MASS_UNIT = SOL[1] # 1 solar mass
    POS_UNIT = 149597870.7 # km (1 AU)
    TIME_UNIT = 31556736.0 # s (1 year)
    VEL_UNIT = POS_UNIT / TIME_UNIT # AU/year
    for orb in JPL_DATA
        mass = orb[1] / MASS_UNIT
        position = SVector{3,Float64}(orb[2:4]) / POS_UNIT
        velocity = SVector{3,Float64}(orb[5:7]) / VEL_UNIT

        particle = Particle(position, velocity, mass)
        push!(solar_bodies, particle)
    end
    return System{3,Float64}(solar_bodies)
end
```

Notice the very deliberate choice of units. On a computer all numbers are zeros and ones – they are *definitionally dimensionless*. The choice of a system of units is up to us, and due to floating point arithmetic that choice has practical consequences. If the numbers in a simulation vary by many orders of magnitude, floating point precision gets lost. A good rule of thumb is to use units so that the core quantities you need to work with are of order $\mathcal{O}(1)$. For the Solar System, using astronomical units, solar masses, and years is a natural choice.

Know your units

Never perform a simulation without knowing what your units are. Saying that the velocity is 1.0 is meaningless – is that a meter per second, or a light year per year? Failing to work with units correctly is a surprisingly common source of error in computational science. So: choose a consistent system for your simulation, convert all initial inputs to that system, and convert your outputs back to a more familiar or convenient set of units for analysis if you need to.

8.2.4 Integrators

The high-level API from code block 8.2 leads us to our next design challenge. The contract requires that we pass in an `integrator` object, and the main loop will itself call a function like `integrate!(system, force_calculator, integrator, dt)`. How will we design this next, internal contract? Just as we did with the `run_simulation!` function, we want to include only the complexity that is native to the integrator itself. How will we do this in a way that allows us to flexibly use not just the forward Euler method but any other integration scheme we later want to adopt?

Simple: we will leverage the power of Julia's multiple dispatch paradigm. We will define an abstract type that represents the *concept* of an integrator, and then build specific concrete `structs` for each different algorithm.

```
abstract type AbstractIntegrator end

struct ForwardEuler{D,T} <: AbstractIntegrator
    accelerations::Vector{SVector{D,T}}
end
function ForwardEuler(system::System{D,T}) where {D,T}
    return ForwardEuler(zeros(SVector{D,T},length(system.particles)))
end
```

Here we have made a crucial design choice: our integrator `structs` will be stateful. The `ForwardEuler` struct, for example, contains a pre-allocated vector to store accelerations. Since our integrator is stateful, we have provided a constructor – here we just need to allocate a large-enough array, but other integrators might require more complex initialization. A "purer" functional design might have a `compute_acceleration` that returns a new acceleration vector at each step, but allocating potentially large vectors at every timestep would be terrible for performance. By pre-allocating a kind of "scratch space" and mutating it in place with a `compute_acceleration!` function instead, we are making a deliberate trade-off in favor of performance. This is a classic example of a kind of meet-in-the-middle design philosophy, letting the lower-level performance considerations inform our higher-level API.

With our basic type hierarchy in place, we can now write a specific *method* for `integrate!` that dispatches on our `ForwardEuler` type. As shown in code block 8.3, this allows us to define the forward Euler algorithm as a nicely self-contained function.

```
function euler_step(p::Particle, acceleration, dt)
    new_position = p.position + p.velocity * dt
    new_velocity = p.velocity + acceleration * dt
    return Particle(new_position, new_velocity, p.mass)
end

function integrate!(system::System{D,T}, force_calc,
                integrator::ForwardEuler{D,T}, dt::Float64) where {D,T}

    compute_acceleration!(integrator.accelerations,system,force_calc)
    system.particles .= euler_step.(system.particles,
                                integrator.accelerations, dt)
end
```

Code block 8.3: An `integrate!` method that specializes on the `ForwardEuler` type. By adding new methods for other integrator types, we can extend our simulation's capabilities without changing other code.

To help with that, we've written pure `euler_step` function whose sole responsibility is to encapsulate the logic of the forward Euler algorithm, leaving the `integrate!` function as a

high level orchestrator whose code reads like a simple description of what it does: "calculate the accelerations and update all of the particles with an Euler step." Using the broadcasting assignment (`.=`) performs the operation efficiently and in place, giving us both the readability of a declarative style and the performance of a hand-written loop.

8.2.5 Force calculators

The contract that we just wrote for the integrator tells us what to do next: we need to flexibly design a way of calculating pairwise forces. Just as with the forward Euler method, we want to start with the simplest implementation, but give ourselves the flexibility to easily change to better ways of calculating the forces. Following precisely the same pattern as above, we first build a type hierarchy for our `ForceCalculators`, and while we're at it we'll define a simple helper function[83] that adds the contribution from a pairwise force to the vector of particle accelerations.

```
abstract type AbstractForceCalculator end

struct BruteForceCalculator{F} <: AbstractForceCalculator
    pairwise_force::F
end

function _apply_force_pair!(accelerations, i, j, particles, force)
    force_on_p1 = force(particles[i], particles[j])
    accelerations[i] += force_on_p1 / particles[i].mass
    accelerations[j] -= force_on_p1 / particles[j].mass
end
```

Our type declaration has the force calculators as, again, stateful entities – in this case, even our simplest force calculator will contain a reference to the specific force law that it will apply when computing interactions. But, with this work done, our `compute_acceleration!` is nicely declarative: it resets the acceleration vector and then, for every unique pair of particles, it call our helper function to do the work.

Once again we have a certain payoff for our modular design process. The `compute_acceleration` function is easy to read and reason about, with all of the details of applying Newton's third law encapsulated in a helper and all of the specifics of the physical force law encapsulated within the `calculator` object itself. In addition to making our code easier to extend with more advanced methods later, this clean separation also makes our code easier to test and maintain.

8.2.6 Results

We're now in a position to present our first test of all of this machinery by running a simulation of our Solar System! There are many ways we could present our results – traces of the trajectories of the planets relative to the Solar System's barycenter over time, phase-space plots of positions

[83] Another common Julia convention: functions in modules that start with an underscore are typically understood to be private "helper" functions that the user probably should not be calling.

```
function compute_acceleration!(accelerations::Vector{SVector{D,T}},
        sys::System{D,T},calculator::BruteForceCalculator) where {D,T}

    fill!(accelerations, zero(SVector{D,T}))

    n = length(sys.particles)
    for i in 1:(n-1)
        for j in (i+1):n
            _apply_force_pair!(accelerations, i, j, sys.particles,
                               calculator.pairwise_force)
        end
    end
    return nothing
end
```

Code block 8.4: A `compute_acceleration!` method that specializes on the `BruteForceCalculator` type. By adding new methods for other ways of computing forces, we can extend our simulation's capabilities without changing other code.

vs velocities in each planet's orbital plane. But the most fundamental test for any physical simulation is not about aesthetics; it's a test at the heart of physics: are quantities we know to be physically conserved *actually conserved numerically*.

Figure 8.1 puts our `ForwardEuler` integrator to the test. We simulate the solar system forward in time for one millennium and track the relative error in the total energy of the system. Since our method comes from a truncation of a Taylor series, we of course expect that our choice of Δt might matter. We thus compare several values of Δt – the smallest corresponds to jumping forward by less than an hour at a time, and the largest corresponds to a time step of roughly half a week at a time.

The plot is a damning indictment. There is a *systematic* error, with the total energy of our simulated solar system monotonically increasing. The rate of the energy drift depends on the step size, but the *fact* that it happens can't be changed by using ever more finely resolved increments of time.

As an aside, while the magnitude of the relative errors plotted above seem relatively modest on the scale of this plot, we have to remember that this is the relative error in the *total potential and kinetic energy of the entire solar system*. To drive the point home: in the simulation with the smallest time-step size, $\Delta t = 10^{-4}$ years, the unphysical energy gain is such that the Earth's orbit has widened all the way to ≈ 2.97 AU from the Sun after just one thousand years and the moon has drifted so that it sits a distance 0.45 AU away from us[84]. This tragic failure is not some bug in our code, but a fundamental flaw in the algorithm we chose. The obvious culprit is the crude, first-order accuracy of the Euler method – perhaps the logical next step, then, is to work with more mathematically sophisticated *higher-order* integrators? We'll explore that idea in Chapter 9.

[84] I suppose we wouldn't have to worry much about global warming – or the tides – in that scenario. Hardly comforting.

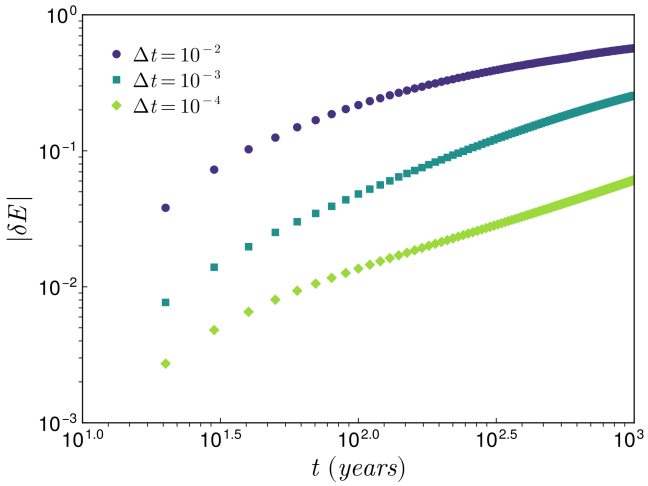

Figure 8.1: The relative error, $\delta E = (E(t) - E(t_0))\,/|E(t_0)|$, in numerically computing the total energy of our solar system simulation over the course of a millennium relative to its state when the simulation was started. Data correspond to using the forward Euler method with three different values of Δt (in units of Earth-years).

Chapter 9

Integration schemes for ODEs

In the last chapter we first introduced the simple forward Euler method for solving the initial value problem, and then set up a framework of code to simulate classical N-body systems. There are some settings where the humble Euler method works well, but when we tried to apply it to the problem of celestial mechanics with just a handful of objects in our solar system things quickly fell apart.

9.1 Higher-order integrators

One obvious suspect in the poor results above is the crude, first-order accuracy of the forward Euler method. The local truncation error of $\mathcal{O}(\Delta t^2)$ means that the *global* error, accumulated over many steps to reach a fixed time $T = N\Delta t$, will scale as $\mathcal{O}(\Delta t)$. For many applications, this is simply not good enough. A logical next step would be to find a method that accounts for higher-order terms in the Taylor series expansion of $\mathbf{y}(t + \Delta t)$.

Expanding $\mathbf{y}(t + \Delta t)$ to higher order, this time going back to a scalar example to keep the notation clean:

$$y(t + \Delta t) = y(t) + \Delta t \dot{y}(t) + \frac{\Delta t^2}{2}\ddot{y}(t) + \mathcal{O}(\Delta t^3).$$

We know that $\dot{y} = f(y, t)$. Adopting the notation in which f_y and f_t mean the derivative of f with respect to the subscripted variable, we can find the second derivative, $\ddot{y}$, by applying the chain rule to f:

$$\ddot{y} = \frac{d}{dt}f(y(t), t) = \frac{\partial f}{\partial y}\frac{dy}{dt} + \frac{\partial f}{\partial t} = f_y f + f_t.$$

Substituting these into the Taylor expansion gives us a second-order accurate update rule:

$$y(t + \Delta t) = y(t) + \Delta t f + \frac{\Delta t^2}{2}(f_y f + f_t) + \mathcal{O}(\Delta t^3).$$

While this "Taylor series method" is indeed more accurate, it's often impractical. It requires us to analytically calculate the partial derivatives of $\mathbf{f}$, which can be extremely complicated for some of the complex functions that arise in physical simulations. So, while tempting in its simplicity, such Taylor series methods are very rarely used in practice.

The insight of mathematician Carl Runge [26] – developed substantially farther by Wilhelm Kutta [27] – was to recognize that we can approximate this higher-order Taylor expansion with explicitly calculating higher-order derivatives of $\mathbf{f}$. The key idea is to use multiple evaluations of $\mathbf{f}$ *within each timestep* to probe the function's higher-order derivatives.

9.1.1 Deriving the RK2 Family

To see this, let's derive the "RK2" family of integrators. We will first take a small step which is a fraction of the discretized Δt, and then use the information from that step to compute a more accurate final update to get to the end of the timestep. The general version of this would look like:

$$\begin{aligned}\mathbf{k}_1 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n)\\ \mathbf{k}_2 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \alpha\mathbf{k}_1, t_n + \beta\Delta t)\\ \mathbf{y}_{n+1} &= \mathbf{y}_n + a\mathbf{k}_1 + b\mathbf{k}_2\end{aligned}$$

Here the first stage, $\mathbf{k}_1$, is the familiar forward Euler step. The second stage, $\mathbf{k}_2$, evaluates the slope at some intermediate point in both time ($t_n + \beta\Delta t$) and state space ($\mathbf{y}_n + \alpha\mathbf{k}_1$). The actual integration to get to $\mathbf{y}_{n+1}$ uses a weighted average of the two slope estimates, with weights a and b. In order to actually be useful, we now choose the free paramaters a, b, α, β to make the update rule match the second-order Taylor expansion.

Performing a Taylor expansion of $\mathbf{k}_2$ around $(\mathbf{y}_n, t_n)$ (again, switching to scalar notation for clarity), we have

$$\begin{aligned}k_2 &= \Delta t \cdot f(y_n + \alpha k_1, t_n + \beta\Delta t)\\ &= \Delta t\left[f(y_n, t_n) + \alpha k_1 f_y + \beta\Delta t f_t + \mathcal{O}(\Delta t^2)\right]\\ &= \Delta t f + \Delta t^2(\alpha f f_y + \beta f_t) + \mathcal{O}(\Delta t^3)\end{aligned}$$

Substituting both this and the $k_1 = \Delta t f$ back into the update rule for y_{n+1} gives

$$y_{n+1} = y_n + (a+b)\Delta t f + b\Delta t^2(\alpha f f_y + \beta f_t) + \mathcal{O}(\Delta t^3). \tag{9.1}$$

The actual Taylor expansion is

$$y_{n+1} \approx y_n + \Delta t f + \frac{\Delta t^2}{2} f f_y + \frac{\Delta t^2}{2} f_t. \tag{9.2}$$

Matching the coefficients of Eqs. (9.1) and (9.2), we arrive at a system of three equations and four unknowns:

$$\begin{aligned}(\text{coeff of } \Delta t f): &\quad a + b = 1\\ (\text{coeff of } \Delta t^2 f f_y): &\quad b\alpha = 1/2\\ (\text{coeff of } \Delta t^2 f_t): &\quad b\beta = 1/2\end{aligned}$$

Thus, we see there is a *family* of second-order Runge-Kutta (RK2) methods: we are free to choose one parameter arbitrarily, and then determine the others from the system of equations above. Two popular choices are

- **Heun's Method:** $\{\alpha = 1,\ \beta = 1,\ a = 1/2,\ b = 1/2$. This update rule takes the average of the slope at the beginning and an estimated end of the interval.
- **The Midpoint Method:** $\{\alpha = 1/2,\ \beta = 1/2,\ a = 0,\ b = 1$. This uses an Euler step to estimate the state at the middle of the time step, evaluates the slope there, and uses *that* slope to make the step from t_n to t_{n+1}.

Both methods have a local truncation error of $\mathcal{O}(\Delta t^3)$, leading to a much more favorable global error of $\mathcal{O}(\Delta t^2)$. Different choices of the parameters do lead to different characteristics of the solvers, though. For instance, the midpoint method tends to do well when the ODE's behavior is dominated by oscillations, whereas Heun's method is a more robust general-purpose solver that tends to be slightly more stable than the midpoint method. Thus, there tends to be a certain art to determining which higher-order method to use for any particular problem.

9.1.2 The Butcher tableau

Extending this process to higher orders is a straightforward but algebraically tedious process. To simplify the notation and classification of different schemes, a compact representation known as *Butcher tableau* was developed [28]. This notation gives a simple, unambiguous "recipe" that completely specifies all of the parameters of general Runge-Kutta methods.

An s-stage explicit Runge-Kutta method is defined by the general form:

$$\begin{aligned}
\mathbf{k}_1 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n) \\
\mathbf{k}_2 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + a_{21}\mathbf{k}_1, t_n + c_2\Delta t) \\
\mathbf{k}_3 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + a_{31}\mathbf{k}_1 + a_{32}\mathbf{k}_2, t_n + c_3\Delta t) \\
&\vdots \\
\mathbf{k}_s &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \sum_{j=1}^{s-1} a_{sj}\mathbf{k}_j, t_n + c_s\Delta t) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \sum_{i=1}^{s} b_i\mathbf{k}_i.
\end{aligned}$$

In order to be consistent (i.e., achieve *at least* first-order accuracy), we require $\sum_i b_i = 1$. A common convention is to additionally impose a row-sum requirement, $c_i = \sum_j a_{ij}$. Other constraints come from the order of accuracy that is desired, and methods are often thought of as a pair of integers labeling the number of stages they require and the order $\mathcal{O}(\Delta t^n)$ they achieve.

In any case, the coefficients that uniquely define a method – the a_{ij}, b_i, and c_i values – can be neatly arranged in the "tableau":

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \end{array} \quad = \quad \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \tag{9.3}$$

For the *explicit* methods we are considering, the matrix A is strictly lower-triangular ($a_{ij} = 0$ for $j \geq i$), and $c_1 = 0$. The zeros in the upper-right of A are often omitted for clarity. Using this notation, we can represent our previous examples very concisely.

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \end{array} \qquad \begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array}$$

Forward Euler (RK1) Midpoint method (RK2) Heun's method (RK2)

9.1.3 The workhorse ODE solver: RK4

Perhaps the most famous and widely used integrator is the "classic" fourth-order Runge-Kutta method, which provides an excellent balance of accuracy and computational cost. Its update rule is given by the Simpson's rule weighted average:

$$\begin{aligned} \mathbf{k}_1 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n) \\ \mathbf{k}_2 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \frac{1}{2}\mathbf{k}_1, t_n + \frac{1}{2}\Delta t) \\ \mathbf{k}_3 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \frac{1}{2}\mathbf{k}_2, t_n + \frac{1}{2}\Delta t) \\ \mathbf{k}_4 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \mathbf{k}_3, t_n + \Delta t) \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \end{aligned}$$

And its corresponding Butcher Tableau is:

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

This method has a local error of $\mathcal{O}(\Delta t^5)$ and a global error of $\mathcal{O}(\Delta t^4)$, making it a robust and popular choice for a wide variety of problems.

But just as with the RK2 methods, integrators with different tableau may perform better or worse on specific ODEs. This is an area of current research; there is great freedom in choosing the tableau of an integrator in the RK family, and despite the age and history of the field contributions continue to be made [29]. It is also important to consider the number of stages in these integrators – more stages imply more evaluations of the function $\mathbf{f}$. In a physical simulation each computation of the forces might be *the* most computationally expensive step; if modest accuracy is acceptable, it might be that a lower-order method with a smaller Δt might takes less total time to run for a fixed duration than a higher-order method with a larger Δt. On the other hand, if the evaluation of the function with derivative information is cheap – perhaps you are simulating a large system of linear ODEs – or you really need highly accurate answers, using higher-order methods can be tremendously advantageous.

9.1.4 Planetary dynamics with RK4

The brute-force mathematical solution to the low accuracy of Euler's method is to use a higher-order integrator. But is the extra implementation complexity worth it? Thanks to the flexible, extensible framework we designed in Chapter 8, we don't have to refactor any of our code to find out – we just need to define a new integrator struct and write a new `integrate!` method that can dispatch on it. For RK4, we need to calculate four intermediate "slope" vectors (the k_1 - k_r above) at each time step. To avoid allocating new memory for these vectors at each time step, we'll pre-allocate a scratch space inside of our integrator object:

```
struct RungeKutta4{D,T} <: AbstractIntegrator
    initial_particles::Vector{Particle{D,T}}
    k1_accel::Vector{SVector{D,T}}
    k2_accel::Vector{SVector{D,T}}
    k3_accel::Vector{SVector{D,T}}
    k4_accel::Vector{SVector{D,T}}
end
```

The constructor will make sure these scratch spaces are the correct size, and we'll add an `integrate!` method that specializes on this new type, implementing the pattern of updates we wrote down in Section 9.1.3.

Was it worth the effort for our investigation of the future of our solar system? Figure 9.1 compares the relative error in the energy, again over the course of a millennium, for forward Euler approaches with $\Delta t = 10^{-4}$ and $\Delta t = 10^{-6}$ with the classic RK4 integrator discretized at $\Delta t = 10^{-3}$. From our analysis above we expect that the RK4 approach will take less time than either of the forward Euler integrations – it needs 10 and 1000 fewer time steps, respectively, to integrate a fixed duration in time compared, so the fact that each time step is broken into 4 relatively expensive "compute the acceleration and do some calculations" mini-steps doesn't bother us. We also expect that it will be *much* more accurate than either forward Euler approach: global errors of $\mathcal{O}(\Delta t^4)$ are just much smaller than global errors of $\mathcal{O}(\Delta t)$.

Well, Fig. 9.1 confirms both of those expectations. But it *also* confirms that we haven't solved the fundamental problem – just as with the forward Euler methods, the RK4 method also fails to conserve energy. The magnitude is much smaller, but the qualitative error – a slow, steady, upward drift – persists. Perhaps we've delayed the catastrophic consequences, delaying them from a milliennium's time to millions of years from today, but what's a few million years to to Solar System? The blink of an eye.

What then, are we to do? The Runge-Kutta family of integrators is extremely powerful, and is an excellent tool for general-purpose ODE solving, but it is blind to the special structure of physical laws. Do we just have to resign ourselves to using time steps so tiny that the inevitable drift of our integrator stays under some acceptable threshold? No. We have forgotten to think like physicists: the solution to our problem is not to be found in ever-higher-order Taylor expansions, but in a different class of integrators. Integrators that are designed from the ground up to respect the fundamental symmetries of physical systems.

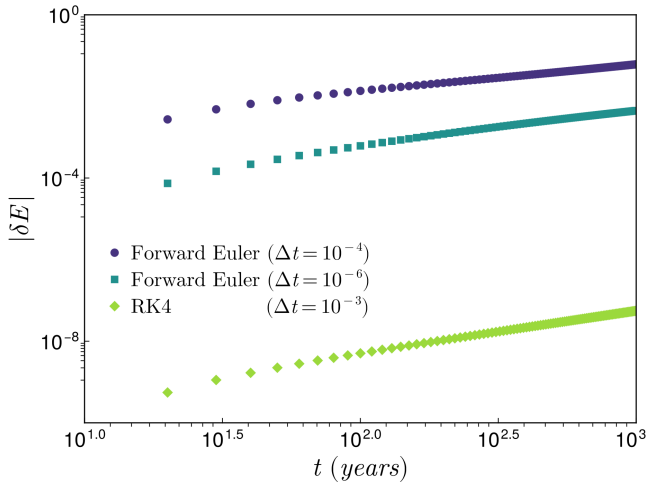

Figure 9.1: The relative error, $\delta E = (E(t) - E(t_0))/|E(t_0)|$, in numerically computing the total energy of our solar system simulation over the course of a millennium relative to its state when the simulation was started. Data correspond to using the forward Euler method with two different values of Δt (in units of Earth-years), compared to the classic RK4 method using a larger time step than either forward Euler simulation.

9.2 Time-reversible integration

One of the most important features of Hamiltonian dynamics is that it is time-reversible – do the integrators we are using so far have this property? Let's explicitly show that the answer is "clearly not." To demonstrate, let's return to the forward Euler method in the context of a simple harmonic oscillator in one dimension. The state vector will just be $\vec{y} = \{q, p\}$, and (choosing a convenient system of units) the time derivative is $\vec{f} = \{p, -q\}$. We'll probe the time-(ir)reversibility of the method by (1) applying the forward Euler rule forward by one step from some initial state, and then (2) applying the rule again, but in the backwards $(-\Delta t)$ time direction; we'll see if the system ends up where it started.

Starting at $\vec{y}_0$, the forward step is just

$$\vec{y}_1 = \begin{pmatrix} q_0 + \Delta t\, p_0 \\ p_0 - \Delta t\, q_0 \end{pmatrix}. \tag{9.4}$$

The final position after a backwards step is then $\vec{y}_f = \vec{y}_1 + (-\Delta t) f(\vec{y}_1)$. Evaluating this expression, we find

$$\vec{y}_f = \begin{pmatrix} q_0 + \Delta t\, p_0 - \Delta t\, p_0 + \Delta t^2 q_0 \\ p_0 - \Delta t\, q_0 + \Delta t\, q_0 + \Delta t^2 p_0 \end{pmatrix} = (1 + \Delta t^2) \begin{pmatrix} q_0 \\ p_0 \end{pmatrix}. \tag{9.5}$$

That is: after running time forward and then backwards, we find our system at a state different from where it started. Surprisingly, just a tiny adjustment to our forward Euler method can correct this flaw.

9.2.1 Symplectic Euler integration

The symplectic[85] Euler method proposes an unequal way of propagating positions and momenta forward in time. In the context of a system governed by the Hamiltonian $\mathcal{H} = T + V$ – i.e., composed of a kinetic term that depends only on particle momenta and a potential term that depends only on positions – the symplectic Euler method is formulated as either

$$\begin{aligned} p_{n+1} &= p_n - \Delta t V'(q_n) \\ q_{n+1} &= q_n + \Delta t T'(p_{n+1}) \end{aligned} \tag{9.6}$$

or

$$\begin{aligned} q_{n+1} &= q_n + \Delta t T'(p_n) \\ p_{n+1} &= p_n - \Delta t V'(q_{n+1}) \end{aligned} \tag{9.7}$$

Notice that in the first formulation, advancing the position requires knowing the *future* momenta (and vice verse in the second formulation). Notice, furthermore, that these two formulations do the same operations but in the reverse order: a "kick and then drift" or a "drift and then kick" of the particles. Because of the nicely separable structure of these equations, though, implementing this is straightforward:

```
struct SymplecticEuler{D,T}<: AbstractIntegrator
    accelerations::Vector{SVector{D,T}}
end
function symplectic_euler_step(p::Particle, acceleration, dt)
    new_velocity = p.velocity + acceleration * dt
    new_position = p.position + new_velocity * dt
    return Particle(new_position, new_velocity, p.mass)
end
function integrate!(system::System{D,T}, force_calc,
             integrator::SymplecticEuler{D,T}, dt::Float64) where {D,T}

    compute_acceleration!(integrator.accelerations,system,force_calc)
    system.particles .= symplectic_euler_step.(system.particles,
                                integrator.accelerations, dt)
end
```

We can now ask: if we use (e.g.) Eq. (9.6) to go forward by one time step, what operation would return the system to *exactly* where it started? We can answer this by simply rearranging those equations: the inverse operation is

$$\begin{aligned} q_n &= q_{n+1} - \Delta t T'(p_{n+1}) \\ p_n &= p_{n+1} + \Delta t V'(q_n) \end{aligned} \tag{9.8}$$

That is, the inverse operation of Eq. (9.6) is just Eq. (9.7), but with $\Delta t \to -\Delta t$. Adopting the notation where $\phi_{(1)}$ refers to the operator that carries out one step of the first formulation,

[85] "Symplectic" because it preserves phase space volumes under Hamiltonian evolution. The word was proposed by Weyl as alternative name to what he had previously called "complex groups" [30], a structure with close analogy to the orthogonal group.

Eq. (9.6), we have found that the *adjoint* of that operator – the operator $\phi^*_{(1)}$ which reverses the order of all operations in the operator and reverses the direction of time – is the inverse of the operator:

$$\phi^{-1}_{(1)}(\Delta t) = \phi_{(2)}(-\Delta t) = \phi^*_{(1)}(\Delta t). \tag{9.9}$$

This is, in fact, the general requirement for a time-reversible operator.

This time-reversibility is a crucial first step – we have made sure that our integrator respects a fundamental symmetry of the underlying physics, and this gives us a method which is inherently more stable than the forward Euler method when simulating Hamiltonian systems. But is this property alone sufficient to *guarantee* the long-term energy conservation we desire?

9.2.2 The Velocity Verlet algorithm

Before answering that question, it's worth noting that while the symplectic Euler method is an improvement on forward Euler, there is an immediate, essentially "free" improvement we can make to it. Knowing that the most computationally expensive part of a particle-based system is usually the calculation of forces, we can do a very mild version of the RK trick and split the timestep in a way that looks very symmetric. The algorithm is now commonly called the "velocity Verlet" (after Loup Verlet's work on molecular dynamics in the 1960's [31]) or "Størmer-Verlet" algorithm (after Størmer's work in 1907 studying particles moving in a magnetic field [32]), but it was used by Delambre in 1791 [33] to calculate astronomical tables, and (!) by Newton in his proof of Kepler's second law [34].

Here's what that algorithm looks like. We perform the following three-step waltz for each particle in every timestep:

$$(1) \quad \vec{p}_i(t + \frac{\Delta t}{2}) = \vec{p}_i(t) + \frac{\Delta t}{2}\vec{F}_i(\mathbf{q}(t)) \tag{9.10}$$

$$(2) \quad \vec{q}_i(t + \Delta t) = \vec{q}_i(t) + \frac{\Delta t}{m}\vec{p}_i(t + \Delta t/2) \tag{9.11}$$

$$(3) \quad \vec{p}_i(t + \Delta t) = \vec{p}_i(t + \frac{\Delta t}{2}) + \frac{\Delta t}{2}\vec{F}_i(\mathbf{q}(t + \Delta t)) \tag{9.12}$$

Notice that the result of the force calculation at step (3) of this update is the same force needed in step (1) of the next iteration. Thus, as long as we set up a stateful integrator and initialize it properly, this pattern still only requires one force calculation per Δt.

```
struct VerletIntegrator{D,T} <: AbstractIntegrator
    accelerations::Vector{SVector{D,T}}
end
function VerletIntegrator(system::System{D,T},force_calculator) where
{D,T}
    initial_accelerations = zeros(SVector{D,T},
length(system.particles))
    compute_acceleration!(initial_accelerations, system,
force_calculator)
    return VerletIntegrator(initial_accelerations)
end
```

The combination of simplicity, computational efficiency, and (as we'll see) excellent stability has made this the go-to algorithm for simulating particles and planets evolving under Hamiltonian dynamics for literally centuries.

9.3 Symplectic integrators and energy conservation

To understand the origin of that excellent stability, we turn for a moment to a more formal operator formulation of Hamiltonian dynamics.

9.3.1 The Liouvillian

For any function $\rho(\mathbf{q}, \mathbf{p})$ of classical phase space variables, the evolution of that function governed by a Hamiltonian $\mathcal{H}$ is given by the Poisson bracket:

$$\frac{d\rho}{dt} = \{\rho, \mathcal{H}\} = \sum_i^N \frac{\partial \rho}{\partial \vec{q}_i}\frac{\partial \mathcal{H}}{\partial \vec{p}_i} - \frac{\partial \rho}{\partial \vec{p}_i}\frac{\partial \mathcal{H}}{\partial \vec{q}_i} \tag{9.13}$$

For the purposes of writing down a formal solution, we define the Liouvillian operator, $L_{\mathcal{H}}\rho = \{\rho, \mathcal{H}\}$. This lets us cast the time evolution as a simple equation with the formal solution:

$$\frac{d\rho}{dt} = L_{\mathcal{H}}\rho \quad \Rightarrow \rho(t) = e^{tL_{\mathcal{H}}}\rho(t=0). \tag{9.14}$$

Restricting our attention to Hamiltonians of the form $\mathcal{H} = T(\mathbf{p}) + V(\mathbf{q})$. The properties of the Poisson bracket mean that we can decompose the Liouvillian:

$$L_{\mathcal{H}}\rho = (L_T + L_V)\rho = \{\rho, T\} + \{\rho, V\}. \tag{9.15}$$

Each of these parts are exactly solvable, albeit only accounting for part of the system dynamics.

Sadly, the evolution operator isn't $L_{\mathcal{H}}$ but $e^{tL_{\mathcal{H}}}$, and the Baker-Campbell-Hausdorff (BCH) formula tells us that we are not allowed to just split the evolution into a part that talks only to the positions and a part that talks only to the momenta. In particular, [35]:

$$e^X e^Y = e^Z, \text{ where } Z = X + Y + \frac{1}{2}[X,Y] + \frac{1}{12}\left([X,[X,Y]] - [Y,[X,Y]]\right) + \cdots. \tag{9.16}$$

This formula tells us, for instance, that evolving the system by $Y = \Delta t L_T$ and then $X = \Delta t L_V$ (a la Eq. (9.6)) is different from evolving the system by the true $\Delta t L_{\mathcal{H}}$ by an amount proportional to Δt^2 and the commutator of L_T and L_V.

In this language, we can write our velocity Verlet algorithm as this propagator:

$$\phi(\Delta t) = e^{\frac{\Delta t}{2}L_V} e^{\Delta t L_T} e^{\frac{\Delta t}{2}L_V}. \tag{9.17}$$

An explicit calculation – applying BCH first with $X = A\Delta t/2$, $Y = B\Delta t$ and then with the result of that as the new X and $Y = A\Delta t/2$ – shows that this symmetric splitting of the operator cancels the coefficient of the Δt^2 term, approximating the true evolution operator to $\mathcal{O}(\Delta t^3)$ in the exponential. This fact, in combination with still only needing one force computation per timestep, is why velocity Verlet is essentially universally preferred to the symplectic Euler approach.

9.3.2 Backward error analysis and the shadow Hamiltonian

We've now seen that the velocity Verlet algorithm is both time-reversible and second-order accurate... but it still corresponds to an evolution operator which is only an approximation of the true operator, so you might complain that we still haven't explained why the algorithm should do a particularly good job at conserving energy. The final piece of the puzzle comes from an area of study known as backward error analysis [24].

The core idea is to ask: if our algorithm doesn't perfectly solve the original Hamiltonian, might it perhaps *perfectly solve a different but related Hamiltonian?* For a symplectic integrator like Verlet, the answer is, indeed, yes. It can be shown that the algorithm exactly conserves a nearby "shadow Hamiltonian[86]."

The math is mildly tedious, and each different integration scheme requires a separate analysis, but it is straightforward to mechanically carry out. In the case of the velocity Verlet algorithm, one finds [24, 36] that it rigorously conserves

$$\tilde{\mathcal{H}} = \mathcal{H} + \Delta t^2 \mathcal{H}_2 + \mathcal{O}(\Delta t^4). \tag{9.18}$$

That is: it conserves a Hamiltonian which is *very close* to the actual Hamiltonian of interest. The error term is

$$\mathcal{H}_2 = \frac{1}{12}\{T,\{T,V\}\} + \frac{1}{24}\{V,\{T,V\}\}. \tag{9.19}$$

We can evaluate these terms – the Poisson bracket $\{T,V\} = \sum_i (\vec{p}_i/m)\vec{F}$, and the Poisson brackets of T and V with that are, in turn

$$\{T,\{T,V\}\} = -\sum_i^N \frac{p_i^2}{m^2}\frac{d\vec{F}_i}{d\vec{q}_i} \tag{9.20}$$

$$\{V,\{T,V\}\} = -\sum_i^N \frac{F_i^2}{m} \tag{9.21}$$

These have a nicely interpretable physical meaning. Equation (9.20) involves gradients of the forces, capturing how they vary across space; Eq. (9.21) is proportional to the square of the forces, representing the effect of strong interactions. The shadow Hamiltonian is thus perturbed away from the true one by both the magnitude and curvature of the potential energy landscape.

The proof of the pudding is in the eating [37]. Figure 9.2 once again shows a simulation of the Solar System – this time over ten millenia – comparing the workhorse RK4 algorithm with the symplectic Euler and velocity Verlet algorithms. As before, the RK4 simulation has small relative errors in the total energy of the system, but those errors grow linearly, inexorably, with time. In contrast, both symplectic methods have some error in the total energy, but the energy of the system stays consistently *close* to that of the true system: we've devised schemes that rigorously conserve *a* Hamiltonian, and results like Eq. (9.18) tell us that the conserved Hamiltonian is closely related to the Hamiltonian we actually care about.

[86] Sounds like the original Hamiltonian's evil doppleganger.

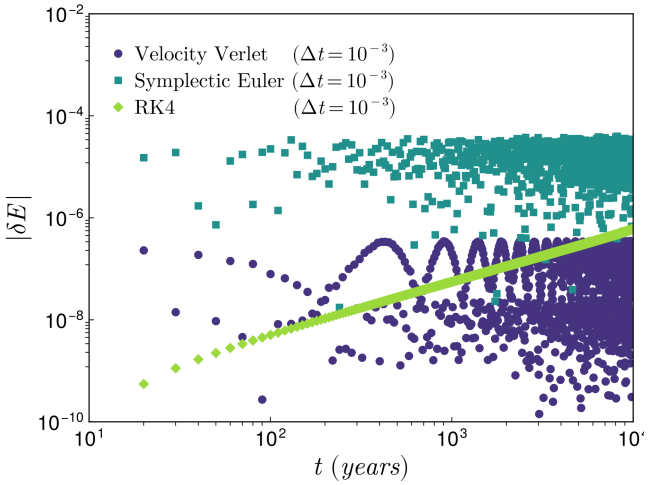

Figure 9.2: The relative error, δE, in numerically computing the total energy of our solar system simulation over the course of ten thousand years relative to its state when the simulation was started. Data correspond to using the symplectic Euler, velocity Verlet, and RK4 algorithms, all using the same Δt.

Chapter 10

Molecular Dynamics

Chapters 8 and 9 have given us a solid foundation for studying ordinary differential equations. We have our toolbox of different integration methods with their different pros and cons, and we have a modular computational structure that will let us apply that toolbox to a breathtaking range of problems in the physical sciences. Indeed, the language of ODEs is ubiquitous: we could study classical mechanical systems governed by Newton's laws, charges in an electrical circuit, solutions to the Schrodinger equation in time-independent potentials, the ecosystem dynamics of competing species, networks of chemical reactions, the expansion equations governing the entire universe... It's all at our fingertips!

In this coda to Module II, we'll focus on a particular example –classical simulations of the movements of atoms, molecules, and coarse-grained "particle" more generally – to see how the specific details of a problem can shape the specific algorithms and approaches we use to adapt our more general structure. We will, again, just scratch the surface of the full complexity and power of these "molecular dynamics" simulations; I recommend Ref. [5] as a good place to start diving into increasingly interesting details.

10.1 An N-body problem in a box

Our goal will be to simulate and study the properties of a macroscopic, "bulk" material – a liquid or gas containing something like Avogadro's number of particles. Doing this directly would be absurd: our computers run at only gigahertz speeds, so even just asking the computer to look at (let alone store in memory) the position of so many particles would take of order a million years. Instead we will simulate much smaller systems containing perhaps $10^3 - 10^6$ particles[87]. But how could such a tiny system replicate the behavior of one that is, in comparison, effectively infinite?

The validity of this approach relies on a key physical principle: in most bulk systems, interactions are *local*. That is, the behavior of a given particle is dominated by its interactions with

[87] An early but important paper in the literature on molecular dynamics was published in 1959 [13], which optimistically noted that "[c]omputers now being planned should be able to handle ten thousand molecules..." We can, obviously, simulation much larger systems, but it is important to remember that we can often extract all of the physical information we want from these smaller systems. Brute force and simply scaling up to ever larger sizes is not always the winning strategy.

its immediate neighbors, with far away particles typically having a negligible influence. This gives rise to a characteristic correlation length, ξ, which the distance over which particle positions and motions are meaningfully correlated. As long as we make our "simulation box" much larger than this correlation length, the particles in the center of the box will behave almost precisely as they would in the true bulk system, blissfully oblivious to the distant boundaries. This leaves of with a critical, immediate problem to solve: what to do about those particles at the boundaries of our (relatively) small simulation?

10.1.1 Periodic boundary conditions

We could let them interact with an artificial wall we build into the simulation to keep everything contained, or let them interact with an empty vacuum, effectively simulating a tiny isolated droplet rather than the behavior of a bulk system. If our interest is at small systems at the nanoscale this may well be the correct thing to do, but not if we are interested in understanding bulk properties: in such systems a huge fraction of the particles are reasonably close to the surface, where their physical behavior (their structural arrangements, their dynamics, their pressure, etc) is completely different from the particles in the interior.

The standard solution is to eliminate either free or confining surfaces entirely by imposing periodic boundary conditions (PBCs) on the simulation box [38], as illustrated in Fig. 10.1. The idea is to imagine our simulation box (our "primary unit cell") surrounded on all sides by an infinite lattice of identical copies of itself. Each particle in our simulation represents not a single entity, but an infinite set of periodic images. For example, when a particle leaves the primary unit cell by traveling across one face, an image of it is simultaneously entering the primary unit cell through the opposite face with the same velocity. This creates a system that is finite in size – and for which we only need to actually keep track of the finite set of particles in the primary unit cell – but which has no edges or surfaces.

Using PBCs has a crucial algorithmic consequence for how we compute the distance between particles, known as the *minimum image convention*. Since each particle represents an infinite set, when we compute the distance (or the forces) between " particles i and j" what do we actually mean? Rather than summing an infinite set of forces and calculating an infinite set of interparticle distances[88], we only need to find the distance between the *closest perdiodic image of the pair,*calculating the forces based on that interaction alone.

Implementing the minimum image convention requires "wrapping" the separation vector between two particles into the dimensions of the central box – along any coordinate it is impossible for particles to be separated by more than the linear size of the simulation box in that direction. Thus, for a cubic box of side length L, the raw separation vector $\Delta x = x_i - x_j$ between particles in the unit cell is adjusted, as we can see visually in Fig. 10.1. If $\Delta x > L/2$, the closest image of particle j is actually in the neighboring box in the negative direction, and the minimum image separation is actually $\Delta x - L$. Similarly, if $\Delta x < -L/2$ the minimum image separation is $\Delta x - L$. In code, this might look like

[88] A bit impractical, at least in this real-space formulation

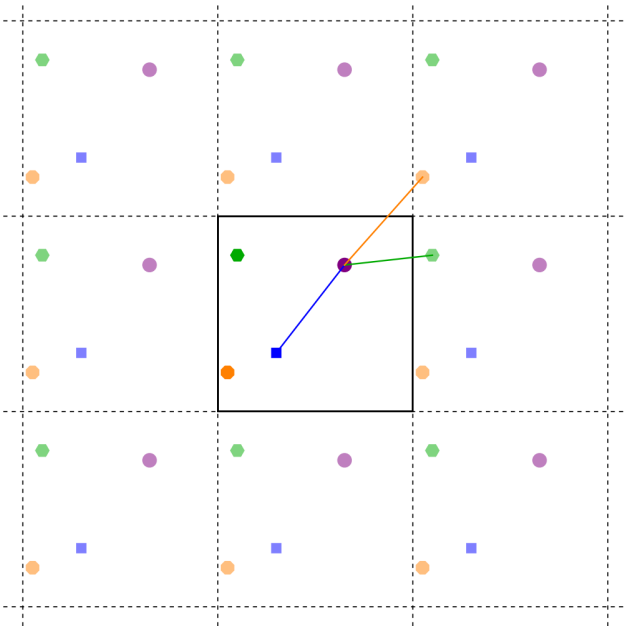

Figure 10.1: A representation of periodic boundary conditions and the minimum image convention in a two dimensional system. Different particles in the primary unit cell are indicated with shapes and full opacity colors, whereas their periodic images have reduced opacity. The minimum image distance between the purple circular particle and its nearest neighbors is shown with solid lines: the shortest separation vector may be contained entirely within the primary unit cell, or it may cross any number of faces.

```
# A simple, explicit implementation
function minimum_image_distance(dx, L)
    if dx > L / 2
        return dx - L
    elseif dx < -L / 2
        return dx + L
    else
        return dx
    end
end

# A vectorized version that works for a vectors `dr` and `L`
function minimum_image_vector(dr, L)
    return dr .- L .* round.(dr ./ L)
end
```

So: when calculating forces we first compute this minimum image separation vector, and then use that to determine distances and directions/magnitudes of forces. This ensures that every particle interacts with its “true” nearest neighbor in the infinite system we are modeling.

10.1.2 Interparticle potentials

With our simulation box defined, we now need to specify the physical interactions that will govern our particles. We will focus on central pairwise interactions, for which we can write an interparticle potential that depends only on the relative separation between two particles, $V(r)$, from which the force is $\vec{F} = -\nabla V(r)$. The choice of V is a modeling decision that depends on the physical system. For charged particles like ions, we might use the long-range Coulombic potential; for gravitationally interacting planets we might use Newton's universal law of gravitation. For many neutral atoms and molecules, and for sterically interacting mesoscale "particles" like colloids, the interactions are effectively short ranged. This is the case we'll focus on, and we'll explore the algorithmic optimizations that can be applied when one knows the interactions act only over a finite distance.

The canonical model for neutral atoms and molecules is the Lennard-Jones potential [39],

$$V_{LJ}(r) = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right). \tag{10.1}$$

This potential, illustrated in Fig. 10.2, is a simple but extremely effective model for noble gases, but it also more generically captures two characteristic features of atomic interactions: a harsh short-ranged repulsion (due, e.g., to Pauli exclusion) and a weaker but more long-ranged attraction (due, e.g., to van der Waals forces). The parameters ε and σ characterize the energy and length scales in the interaction, respectively. Because the LJ potential decays rapidly, the force between distance particles is negligible. This allows us to make a very pragmatic approximation: we introduce a cutoff radius, r_c, and write

$$\tilde{V}_{LJ}(r) = \begin{cases} V_{LJ}(r) & \text{if } r < r_c \\ 0 & \text{if } r > r_c \end{cases} \tag{10.2}$$

This harsh truncation introduces small discontinuities in both the energy and the force, and one can implement models that smooth these discontinuities over [5], but for our purposes it will be sufficient.

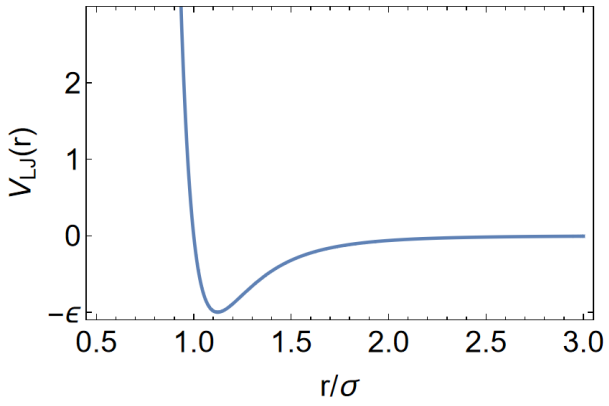

Figure 10.2: A (placeholder) plot of the Lennard-Jones potential.

An alternate short-ranged family of models often used in the study of sterically interacting

particles is based on the overlap between particles of some size:

$$\tilde{V}_s(r) = \begin{cases} \varepsilon\left(1 - \frac{r}{\sigma}\right)^\alpha & \text{if } r < \sigma \\ 0 & \text{if } r > \sigma \end{cases} \tag{10.3}$$

Here $\sigma = r_c$ and ε again set the length and energy scales, and α parameterizes the steepness of the interaction; $\alpha = 2$ corresponds to harmonic repulsions, $\alpha = 5/2$ corresponds to Hertzian repulsion, etc.

10.1.3 Initial conditions

Our simulation also requires an initial state: how will we assign the positions and velocities of our N particles? Unlike the Solar System in Chapter 8, we cannot simply look up the positions of particles in a fluid in a convenient NASA database; instead we must generate plausible starting configurations that represent the system at a desired density and temperature.

This can be surprisingly fussy. The velocities are relatively straightforward: we can generate random numbers so that each component of the velocity of each particle is drawn from a Maxwell-Boltzmann distribution corresponding to our target temperature. Assuming we have the facilities to generate Gaussian numbers with zero mean and unit variance, we simply take those results and scale them so that each velocity component has zero mean and variance $k_B T/m$. After assigning these random velocities, we should be careful to calculate the total momentum of the system and subtract the center-of-mass velocity from each particle – this ensures the system as a whole has zero net momentum and will not drift through the periodic box during our simulation.

The positions require a bit more care. One approach is to place particles at positions that are uniformly randomly distributed through throughout the box. This is easy to do, but will almost certainly result in pairs of particles that happen to be placed very close to each other[89]. If we are using steric repulsions this won't cause too many issues, but if we are using LJ interactions this would seed the system in an initial state of massively high potential energy and hence subject our system to enormous repulsive forces. Alternate strategies include Poisson disk sampling [40] or "soft push-off" initialization [41] – these allow the initial state to be disordered but not uniformly random – or seeding particles on the sites of a regular crystal lattice (cubic, FCC, etc) at the desired density. All of these other approaches have benefits and drawbacks, depending on the eventual target of the simulation.

10.2 Force calculations for short-ranged interactions

With our physical setup defined, we can turn to the computational heart of every molecular dynamics simulation: the calculation of the forces. At every time step we must calculate the net force on every particle; we implemented the most direct method in the `BruteForceCalculator` of Chapter 9. It simply loops through all unique pairs of particles, calculates their (minimum

[89] The expectation value of the minimum distance between all pairs of N points placed uniformly randomly in a d-dimensional hypercube of side length L is $\langle r_{\text{min}} \rangle \sim LN^{-2/d}$ – much smaller than the expectation value of the distances themselves.

image) separation, and sums the forces. This approach is both simple and guaranteed to be correct, but for a system of N particles it requires checking each of the $N(N-1)/2$ pairs. That is, it's complexity scales as $\mathcal{O}\left(N^2\right)$.

Is that scaling a real problem, or just a theoretical concern? The answer depends on precisely what we want to do, but let's perform a quick Fermi estimate. A single Lennard-Jones force calculation involves subtracting vectors, computing the magnitude of the minimum image separation, and then some divisions and multiplications. It can be hard to reason precisely about floating point operations per second on modern hardware[90], but we can run a quick benchmark on a modern CPU core and find that it takes, say, 50 nanoseconds to execute a single LJ force calculation. Our processors run at gigahertz speeds; if we want to calculate the interactions for all pairs of forces between, say, $N = 10^5$ particles, it will take of order $\sim (5 \times 10^{-8}$ s/pair$) \times (5 \times 10^9$pairs$) \approx 250$ s. For *one* timestep.

A simulation long enough to observe the diffusion of molecules might require a million timesteps, and at this rate our "modest" simulation would take almost eight years to complete. The brute-force approach is not just inefficient – for non-trivial systems it is simply not viable. Fortunately, the short-ranged nature of our potentials unlocks much faster algorithms for our use.

10.2.1 Neighbor list structures

While the brute-force approach may be a dead end, our physical intuition might provide an escape. For the short-ranged potentials described above, each particle only feels a force from its nearby neighbors, and so we spend the overwhelming majority of our computational effort checking the distance between the $\mathcal{O}\left(N^2\right)$ pairs that are too far apart to interact. This is a practical application of the complexity analysis from Section 7.3: by identifying an algorithms inefficiency, we can try to find a better one.

A solution in this case is to build a *neighbor list*, a data structure that will allow us to more quickly identify which particles are close enough to potentially interact with each other. It involves a classic trade-off: we accept a more complex implementation and an increase in memory usage in exchange for a dramatic improvement in time complexity. One of the most common (and intuitive) neighbor list structures is the cell list [42]. It an its corresponding algorithm work in two stages.

First is the *binning* stage. We divide up the d-dimensional simulation box into a grid of smaller hyperrectangular cells, where crucially every side length of these small cells is at least as large as the force cutoff radius r_c. This guarantees that the neighbors of a particle can only reside in the same cell as the particle itself or one of the immediately adjacent $3^d - 1$ cells. We then perform an $\mathcal{O}(N)$ sweep over the particles, placing each particle's index into the appropriate cell.

Second is the actual force calculation. For each of the N particles, we refrain from checking the distance to each of the other $N-1$ particles; we instead only check particles in the same or adjacent cells of the target particle. For a system of roughly uniform density, the number of

[90] Modern CPUs can execute multiple instructions at once, use micro-operations to effectively amortize the high cost of otherwise "slow" operations light division, can vectorize similar operations as we compute multiple particle pairs at the same time, and so on.

particles per cell cells will on average be a constant. Since the number of neighboring cells is also a constant, we suddenly find that we can reduce the complexity of the force calculation from $\mathcal{O}\left(N^2\right)$ to $\mathcal{O}\left(N\right)$. The prefactor is, of course, important – for sufficiently small systems it can be slower to use a cell list than the brute force system – but for large systems this is an inevitable improvement.

To implement this in the context of our general, d-dimensional system, we can first create a `CellList` object that stores the grid of cells, along with a function that does the necessary assignment of particles at each step.

With this data structure in hand, we can define a new `CellListCalculator` the plugs directly into our existing simulation framework. The `compute_acceleration!` method for this new type will first update the cell list, and then use that structure to perform the more algorithmically efficient force calculation. This is part of the continuing payoff of our modular design: by creating a new `AbstractForceCalculator` type, we can swap out the fundamental force calculation without changing a line of the other parts of our code. Moreover, we have a built in set of natural tests: we can compare the results of this newer method directly with our older `BruteForceCalculator` approach.

10.3 Thermodynamic ensembles and equilibration

We're making serious progress in our efforts to simulate the properties of bulk physical systems: we have symplectic integration methods that will conserve energy, we have relevant force laws for inter-particle interactions, and we have boundary conditions and algorithmic solutions that will let us simulate reasonably large numbers of particles in a reasonable amount of time. The is a set up that corresponds to conserving the number of particles, N, the volume of our simulation domain V, and the energy of our system, E – in the language of thermodynamics this is the "NVE ensemble".

Perhaps a small, niggling doubt enters your mind at this point, as you realize that you've never actually carried out an experiment in which you have specified *precisely*, down to the electron volt, exactly how much energy is contained in the system you are studying. Much more commonly, you have down your best to control the *temperature* of your system – you do this by connecting your system to a much large "reservoir" whose temperature you know. By doing so, you let the energy of the system itself fluctuate as it trades energy back and forth with your system, even as it maintains a constant temperature. This corresponds to the "NVT" ensemble – can we accommodate this kind of physics in our simulations?

10.3.1 Computational thermostats

Of course we can. We'll mimic the effects of a thermal reservoir by introducing "thermostatting algorithms." Thermostats are modifications to our simulation that allow energy to flow into or out of the system, steering towards a target temperature while trying to preserve the correct statistical properties of the NVT ensemble. There are two fundamentally different approaches we can take here, differing both in how they are implemented and what properties of the system under study they want to preserve.

Stochastic thermostats

Stochastic thermostats try to explicitly model the random interactions the system might have with the heat bath – a molecule in the surface of the system has a chance collision with the reservoir, suddenly experiencing a change in momentum. The most intuitive example of such a thermostat was proposed by Andersen [43]: it imagines that every particle occasionally undergoes such a chance collision with a fictitious particle from the reservoir, which instantly thermalizes the particle to the bath's temperature.

This is straightforward to implement: we perform standard integration steps (e.g., with the velocity Verlet algorithm) for some number of steps. At regular intervals, we select a small fraction of the total number of particles at random; for each selected particle we discard its current velocity and replace it with one drawn from the Maxwell-Boltzmann distribution corresponding to the target temperature. This can be as simple as code block 10.1.

```
using Random
function andersen_thermostat!(sys::System{D,T}, temperature::T,
                probability::Float64) where {D,T}
    for i in eachindex(system.particles)
        if rand() < probability
            p = system.particles[i]
            sigma = sqrt(temperature / p.mass)
            new_v = randn(SVector{D,T}) * sigma
            system.particles[i] = Particle(p.position, new_v, p.mass)
        end
    end
    return nothing
end
```

Code block 10.1: A simple implementation of an Andersen thermostat. Note that `randn` generates random numbers drawn from a zero mean unit variance Gaussian, and that we are working in a system of units where $k_B = 1$.

With a reasonable choice of the probability of selecting particles and the frequency of performing this operation, the Andersen thermostat is a robust way of driving the system to equilibrium and maintaining a constant temperature. Its fundamental drawback is that it is *non-deterministic* and it breaks the true dynamics of the system. By occasionally assigning random velocities, the trajectories we observe are no longer continuous solutions to Newton's equations. This means that while this approach (and those using other stochastic thermostats) are excellent for sampling the static equilibrium properties of our system – what are its bulk elastic properties? how are particles typically arranged with respect to each other at the microscopic scale? – it is unsuitable for measuring dynamical properties.

Deterministic thermostats

An alternative to the stochastic approach is a deterministic, "extended Hamiltonian" thermostat, and perhaps the most famous and widely used version is the Nosé-Hoover (NH) thermostat

[44, 45]. The core idea is a clever abstraction of a physical heat bath Rather than simulating 10^{23} particles in an actual reservoir, the basic NH thermostat models its entire effect with a single extra degree of freedom. The "thermal piston" has a fictitious mass Q that represents the reservoir's thermal inertia and a velocity ζ that acts as a time-dependent "friction coefficient" that acts on the particles in the system.

The equations of motion for the real system are modified to include this friction term:

$$\frac{d\mathbf{r}_i}{dt} = \frac{\mathbf{p}_i}{m_i}$$
$$\frac{d\mathbf{p}_i}{dt} = \mathbf{F}_i - \zeta \mathbf{p}_i$$

The equations of motion for the thermostat degrees of freedom are:

$$\frac{d\zeta}{dt} = \frac{1}{Q}\left(\sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{m_i} - gk_BT\right).$$

Here g is the number of degrees of freedom in the system, and the mass Q determines the timescale of the coupling between the system and the reservoir. A large value of Q corresponds to a slow, weakly-coupled thermostat that lets the system's temperature fluctuate over long times. A small Q corresponds to a fast thermostat that very tightly controls the system's temperature, but which can introduce artificial high-frequency oscillations into the system. Choosing an appropriate value of this parameter is a key part of setting up a stable NVT simulation.

If the system gets too hot (i.e., its kinetic energy grows too large), ζ increases, and this increased "friction" cools the system; if the system is too cold, ζ becomes negative and accelerates the particles in the system, heating everything back up. The result is a set of *deterministic*, time-reversible equations for the motion of this extended system-plus-thermostat. Implementing the NH thermostat is certainly more complex than implementing the Andersen thermostat[91], but it is a proper tool for studying dynamical properties in the NVT ensemble. This is because the extended system not only samples the correct equilibrium structure of our physical system; it samples the correct dynamical trajectories that particles at constant temperature might take.

Other Ensembles

The idea of extending the system you are simulating with fictitious degrees is a quite general and powerful one. A very similar approach can be used to construct not only sophisticated thermostats but also *barastats*, which control the system's pressure, P, by allowing the entire shape and volume of the simulation box to fluctuate. Combining thermostats and barostats allows for simulations in the NPT ensemble, which most closely mimics standard lab experiments.

The Nosé-Hoover equations may seem cleverly constructed, but they are not ad hoc. They are a practical reformulation of the dynamics derived from a conserved Hamiltonian for an extended system. The original formulation by Nosé was

$$\mathcal{H}_{\text{Nosé}} = \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2s^2 m_i} + V(\{\mathbf{q}_\mathbf{i}\}) + \frac{p_s^2}{2Q} + gk_BT\log(s). \tag{10.4}$$

[91] Especially when wants to include it in an explicitly reversible, symplectic time evolution scheme [46].

In that expression, s is a dynamic time-scaling variable (the thermostat's "position") and p_s is its conjugate momentum. While working with this Hamiltonian and its "virtual time" is more complex, the existence of this conserved quantity is the fundamental reason that the NH thermostat is stable, generates the correct NVT ensemble, and possesses a Hamiltonian structure that allows it to be integrated with a symplectic algorithm.

10.4 Observables and measurements

We have finally assembled the components for a complete, efficient[92], and flexible simulation engine. We can model a physical system, choose an appropriate force law, an integrate its equation of motion using physically motivated ODE solvers in multiple thermodynamic ensembles. All that is left for us to do? In the immortal words of XKCD author Randall Munroe: "Stand back – I'm going to try SCIENCE."

How do we bridge the gap between the microscopic trajectories of individual particles that we can trace in our simulations and the macroscopic, measurable properties of the material we are simulating? Much of the answer to that is the focus of statistical physics, but below we will cover the fundamental methods for extracting physical meaning from our simulations.

10.4.1 Equilibration

Before we measure *anything*, we have to address a crucial artifact of our simulation's setup. Our initial conditions – whether those were on a perfect crystalline lattice or a completely random placement of particles – is an artificial state. It is not representative of the natural, equilibrium configuration of, e.g., a fluid that has been sitting at constant temperature for a long time. If we start measuring the properties of our system immediately, our results will be contaminated with the relaxation of our simulation from the initial artificial state to its more typical steady state.

The solution is the same as in a laboratory experiment: after having prepared our system in an unusual way, we must wait for it to settle down and equilibrate. This involves integrating the system forward in time during an "equilibration" or "burn-in" time – this allows the particles to interact, exchange energy, and eventually settle into a statistical steady state that is characteristic of the target ensemble. After this initial transient period is over we can begin a "production run," during which we continue simulating our system, collecting data to be included in our eventual analyses.

How long is long enough for this equilibration phase? The standard answer is to monitor the various quantities you eventually hope to measure – perhaps the total potential energy, or the diffusion constant of the particles. You will observe that as you start measuring from the very start of the simulation, these quantities will show a systematic drift as the system relaxes to equilibrium. As an example, if you start a simulation of a fluid with uniformly random particle positions (i.e., the kind of positions you would expect to see in an ideal gas), you will find that

[92] We have deliberately avoided one of the key topics in modern computational research: writing code that makes efficient use of parallel computing resources. There have been other places where we prioritized clarity and pedagogy over a maximally performant implementation, but for the most part we have written very solid albeit single-threaded code.

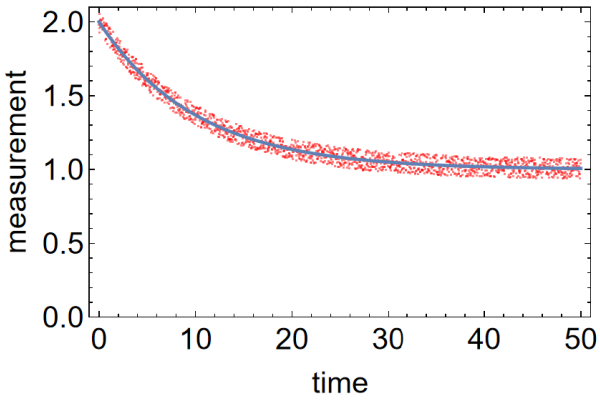

Figure 10.3: A (placeholder) plot showing the equilibration of a system. We have a noisy measurement of some property, and we are able to fit its decay to some plateau value with an exponential function. Our "equilibration" phase is many multiples of the time constant in that exponential fit. This plot is an idealization, and will be replaced with an actual example from a simulation.

the total potential energy starts at a very high value but then steadily decreases, eventually plateauing and then fluctuating around a steady-state value. Ideally, these quantities show an exponential decay from the initial values to their steady state values: this exponential decay gives you a time scale over which the system relaxes to equilibrium, and you want to make sure your equilibration phase is several multiples of this timescale. An example of this is shown in Fig. 10.3

10.4.2 From microscopic trajectories to macroscopic properties

With our system in a statistically steady state, we can begin our production runs. A core principle of statistical mechanics tells us that by averaging observables over a long enough trajectory, we can calculate the macroscopic thermodynamic properties of the system.

Thermodynamic properties

The most fundamental observables are the state variables (or "thermodynamic coordinates") that define the ensemble we are working in. Consider, for instance, our simulations of the NVT ensemble – how could we verify that our thermostat is doing its job? One definition [47] of the temperature of a system relates T with the instantaneous kinetic energy:

$$\langle KE \rangle = \left\langle \sum_{i=1}^{N} \frac{p_i^2}{2m} \right\rangle = \frac{g}{2} k_B T, \tag{10.5}$$

where g is the total number of momentum degrees of freedom (e.g., dN for N point particles in d dimensions). By calculating an "instantaneous kinetic temperature" at each time step, we could determine T for our system and verify that our thermostat is working as desired.

We can also measure state variables that are conjugate to the ones we control. For example, if we are working at fixed V, we could measure the pressure, P, of the system. This pressure

arises from the complex interplay between the motion of the particles and the forces between them; in equilibrium it can be calculated via the virial theorem:

$$P = \frac{Nk_BT}{V} + \frac{1}{V}\left\langle \frac{1}{d}\sum_{i<j} \mathbf{F}_{ij} \cdot \mathbf{r}_{ij} \right\rangle. \tag{10.6}$$

Here the first term is the familiar ideal gas pressure; the second "virial" term is the contribution due to interparticle interactions, and it can be calculated by averaging the dot product of the force and separation vectors for all interacting pairs of particles[93].

Structural properties

Beyond measuring simple thermodynamic variables, MD simulations give us a direct window into the microscopic structure of matter. One of the most important ways of quantifying that structure is the "radial distribution function," $g(\vec{r})$, which describes how the density of particles varies as a function of relative separation from a reference particle.

For an isotropic system, $g(r)$ is the ratio of the average local density at a distance r from a reference particle, $\rho(r)$, to the bulk density $\rho = N/V$. In three dimensions:

$$g(r) = \frac{\rho(r)}{\rho} = \frac{V}{N}\frac{\langle N(r, \Delta r)\rangle}{4\pi r^2 \Delta r}. \tag{10.7}$$

Here, $\langle N(r, \Delta r)\rangle$ is the average number of particles found in a thin spherical shell of radius r and thickness Δr around any given particle. This can be computed in a simulation by building a histogram of all pairwise distances[94]

The importance of $g(r)$ is not just in the simple structural picture it gives us. It's Fourier transform is directly related to the static structure factor $S(k)$, which is precisely what is measured in x-ray and neutron scattering experiments:

$$S(k) = 1 + \rho \int e^{-i\mathbf{k}\cdot\mathbf{r}}(g(r) - 1), d^d\mathbf{r} \tag{10.8}$$

Measure testable observables

The fact that the radial distribution function is related to the structure factor reinforces a general important rule: Simulations correspond closely to experiments – ones we happen to running on a computer rather than with fancy physical instrumentation, but experiments nonetheless. As a consequence, the simulator should in general report and measure things that correspond to experiments and to testable hypotheses.

The shape of $g(r)$ is a direct structural fingerprint of the material's phase. In an ideal gas there are no correlations between particle positions, and $g(r) = 1$ for all r. For a crystal, $g(r)$

[93] Some care must be taken, as always, to account for periodic boundaries.

[94] Or by numerically differentiating the cumulative probability of observing particles separated by some distance – a method which is sometimes more robust.

is characterized by a series of sharp, well-defined peaks that correspond to the crystal lattice spacings. As shown in Fig. 10.4, for a liquid, $g(r)$ shows a relatively sharp peak at the average nearest-neighbor distance, followed by decaying oscillations that represent subsequent "solvation shells" – an example of this is shown in Fig. 10.4

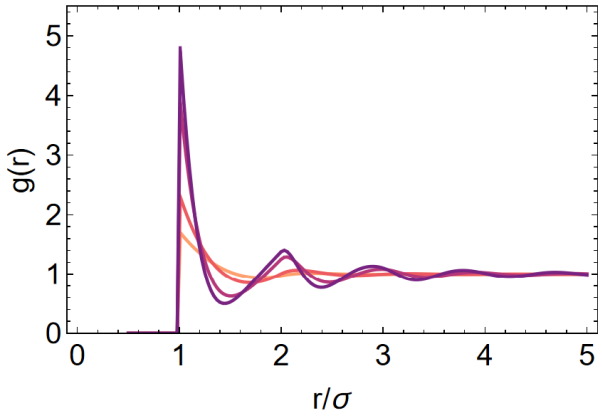

Figure 10.4: A (placeholder) plot of the radial distribution function of a fluid. Current data is for the Percus-Yevick solution of a hard sphere fluid at four different volume fractions (0.2, 0.3, 0.45, 0.5).

Dynamical properties

Finally, because MD generates particle trajectories, we can directly study how particles move over time. One of the most straightforward properties to measure is the *mean-squared displacement* (MSD). This measure how far, on average, particles have moved from a position at an earlier time:

$$\mathrm{MSD}(\Delta t) = \left\langle \frac{1}{N} \sum_{i=1}^{N} |\mathbf{r}_i(t_0 + \Delta t) - \mathbf{r}i(t_0)|^2 \right\rangle_{t_0}, \tag{10.9}$$

where the average, $\langle \ldots \rangle_{t_0}$, is taken over all starting times in the equilibrated trajectory.

Just as $g(r)$ is a fingerprint of the structure, the MSD is a fingerprint of the dynamics. In a solid, particles vibrate around fixed lattice sites, so the MSD plateaus at a small value after a short time. In a fluid, particles are free to move, and the MSD eventually grows linearly with time. This is described by the Einstein relation [48]:

$$\mathrm{MSD}(t) = 2dDt, \tag{10.10}$$

where d is the dimensionality of space and D is the diffusion coefficient. By calculating the MSD from our trajectories and finding its slope in the linear regime, we can directly measure this transport property of our simulated material.

10.5 Coda

Having built our simulation engine on the foundation of Chapters 8 and 9, and having filled it with the tools from this chapter, our basic toolbox is now complete. Our simulations are no

longer just generic solutions to N-body problems: we can use them as a virtual laboratory to probe the properties of a material. We can set the thermodynamic coordinates (N, V, T), we can prepare a sample, let it reach equilibrium, and perform measurements of its fundamental properties. By measuring structure like $g(r)$ – which is itself connected to *many* other material properties – and transport coefficients like D, we can bridge the gap between the microscopic laws of motion and the macroscopic world of material science. By turning these tools on specific problems, we could already find ourselves pushing the boundaries of science outward.

And the code we have is already better than what I wrote as a postdoc.

Module III

Module 3: PDEs and hydrodynamics

When considering either more coarse grained systems *or* quantum mechanical particles (two very different extremes!) one needs to think about not ordinary but *partial* differential equations. This module will explore powerful methods for solving PDEs — such as finite difference, finite volume, and lattice Boltzmann methods. By the end of this module you'll be able to simulate and analyze a wide range of hydrodynamic phenomena, and appreciate the different numerical techniques that are needed when moving into PDEs.

Have in module intro an explicit coarse-graining of particle simulation data from Module II.

For a deeper dive into the methods and physics discussed in this module, consider the following references [49, 50].

Figure III.1: A missing image! For your consideration, instead, is a low poly whale with faintly constructivist touches. It was generated in a mathematical framework based on the Fokker-Planck PDE.

Chapter 11

The landscape of partial differential equations

Under construction

This chapter is still under construction – see handwritten notes

The goal of this chapter is a map of the territory. Note that PDEs are a vast subject. The remarkable *Numerical recipies: the art of scientific computing* text, which runs well over a thousand pages, memorably noted that the intent of its chapter on PDEs was to give the "briefest possible useful introduction," and that ideally "there would be an entire second volume of *Numerical Recipes* dealing with partial differential equations alone" [51].

11.1 From particles to fields

In Module II we simulated N particles, which was fine for the solar system ($N \sim 10$) or some tens of thousands of particles, but we can't possible simulate ODEs for $N \approx 10^{23}$ molecules in a few grams of stuff (let alone, say, simulating the the $\sim 10^{46}$ atoms in the ocean. And even if we somehow could, what would we pay attention to? We can't even store the kind of information from a single "snapshot" of such a system, so we *need* to think about a different kind of physical description to focus on.

11.1.1 Coarse graining

Imagine placing a grid over one of our MD simulations and computing *fields*. Perhaps the two most important are the density field, $\rho(\vec{x}, t)$, which we could construct by counting particles in each grid cell and dividing by the volume of that cell, and the velocity field, $\vec{u}(\vec{x}, t)$, computed as the average velocity of particles in each cell. This is sometimes discussed as the difference between a Lagrangian perspective (in which we look at the fluid motion as if we are observers following individual "parcels" of fluid moving through space and time) and an Eulerian one (in which we focus on observing specific points in space and watching the flow of fluid past

that point). Naturally, despite the names, both of these perspectives were introduced by Euler [52, 53].

Simple binning like the above is quite crude: it might work well in the limit of large numbers of particles per bin, but a particle crossing a grid line still causes discontinuous jumps. A more robust approach is to treat particles not as a point, but as a smeared-out distribution (e.g., a Gaussian). The field value at any point is then the sum of these contributions:

$$\rho(\vec{x}) = \sum_i m_i W(|\vec{x} - \vec{q}_i|, h), \tag{11.1}$$

where W is a smooth kernel function with width h. This is the foundation of methods like Smoothed Particle Hydrodynamics (SPH).

11.1.2 Behavior of the continuum

By tracking how particles move across the boundaries of the cells we use to calculate the field, we find a local formulation for the classical conservation of mass. Consider a small volume V at some fixed location in space. The total mass inside is $M = \int_V \rho \, \mathrm{d}V$, and since mass is neither created nor destroyed, the only way M can change is if mass flows across the surface S that bounds V. The flux of mass is given by $\rho\vec{u}$, so we can write:

$$\frac{d}{dt}\int_V \rho \, \mathrm{d}V = -\oint_S (\rho\vec{u}) \cdot \mathrm{d}\vec{a}. \tag{11.2}$$

First, since V is fixed, we can move the time derivative inside the integral (where, since ρ depends on space and time, it becomes a partial derivative). Next, we can use the divergence theorem to write $\oint_S (\rho\vec{u}) \cdot \mathrm{d}\vec{a} = \int_V \nabla \cdot (\rho\vec{u}) \, \mathrm{d}V$. Combining the terms under a single integral, and noting that our derivation holds for any arbitrary volume, we note that the integrand itself must vanish:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho\vec{u}) = 0. \tag{11.3}$$

This is the *continuity equation*, our first PDE. It represents a fundamental shift in perspective: our state variables explicitly depend on space and time simultaneously.

11.2 The PDE zoo

Partial differential equations are the standard language of physical law, describing everything from quantum mechanical wavefunctions to crashing ocean waves to the general relativitistic description of the universe. Because they cover such a breadth of phenomena, there is a bewildering variety of different PDEs. To navigate this, we classify them based on their *mathematical character*.

Consider a general linear second-order PDE in two independent variables:

$$A\frac{\partial^2 u}{\partial x^2} + 2B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + \cdots = 0. \tag{11.4}$$

The character of the solution depends entirely on the highest-order terms. Just as the equation $Ax^2+2Bxy+Cy^2 = F$ defines different geometric shapes based on the value of the discriminant $\Delta = B^2 - AC$, PDEs are classified by the same criteria. *Elliptic* PDEs have $\Delta < 0$, *parabolic* PDEs have $\Delta = 0$, and *hyperbolic* PDEs have $\Delta > 0$. Intuitively, these distinctions map onto an important physical property: *how does information propagate through the system?*

11.2.1 Elliptic equations

The archetypal example is Poisson's equation

$$\nabla^2\phi = f(x, y). \tag{11.5}$$

Physics: These equations describe systems in equilibrium, such as electrostatics or steady-state heat distribution. They are statements about how spatial variations of a field are locked in balance with sources, sinks, and boundary conditions. (Note the absence of time derivatives!).

Behavior: Information propagates *instantly*. If you change the boundary condition on one side of the domain (e.g., wiggle a charge at infinity), the solution ϕ everywhere changes immediately to accommodate the new equilibrium. Solving these usually amounts to solving a massive system of simultaneous linear equations.

11.2.2 Parabolic equations

Definition:

The archetypal example is the diffusion (or heat) equation:

$$\frac{\partial u}{\partial t} = D\nabla^2 u \tag{11.6}$$

Physics: These describe dissipation, diffusion, and smoothing. Whether it is heat flowing in a metal rod or ink spreading in water, these systems evolve towards a uniform equilibrium.

Behavior: Information spreads infinitely fast, but effectively decays exponentially with distance. Crucially, these systems have an "arrow of time." You can predict the future (smoothing), but you generally cannot reconstruct the past (un-smoothing is mathematically ill-posed and numerically unstable), reflecting the entropic nature of the physics.

11.2.3 Hyperbolic equations

The archetypal example is the wave equation:

$$\frac{\partial^2 u}{\partial t^2} = c^2\nabla^2 u \tag{11.7}$$

or the advection equation:

$$\frac{\partial u}{\partial t} + v \cdot \nabla u = 0. \tag{11.8}$$

Physics: These describe sound, light, and other propagating waves. Unlike parabolic equations which smooth out features, hyperbolic equations preserve them (a sharp pulse remains a sharp pulse as it travels).

Behavior: Information travels at a finite speed, c. This preserves causality: an event at point x can only affect a point y if enough time has passed for the wave to travel the distance $|x-y|$. This "light cone" structure makes numerical stability particularly tricky; the "simulation information speed" set by our discretization ($\Delta x/\Delta t$) must be at least as fast as the physical information speed c.

11.3 Finite differencing

Perhaps the most basic way to solve any of these PDEs is to return to the idea of discretization. Here, though, we discretize space, keeping track of the value of our fields on a lattice of closely spaced points. Just as we had for time discretization in Module II, in which $t_n = n\Delta t$, we discretize each spatial dimension of, e.g., the density field.

Working in 1D for notational simplicity, we write $\rho_i = \rho(x_i)$, where $x_i = i\Delta x$. This lets use recover spatial derivatives from the values of the field on the grid. For instance, the first order derivatives can be written in a number of ways (forward, backwards, central differences), for instance

$$\left.\frac{\partial \rho}{\partial x}\right|_{x=x_i} = \frac{\rho_{i+1} - \rho_{i-1}}{2\Delta x}.$$

Higher-order derivatives can similarly be written in a variety of ways, for instance the famous[95] "1-2-1" pattern for the second derivative

$$\left.\frac{\partial^2 \rho}{\partial x^2}\right|_{x=x_i} = \frac{\rho_{i+1} - 2\rho_i + \rho_{i-1}}{\Delta x^2}.$$

This converts derivative operations into linear algebra operations.

11.4 The method of lines

With this spatial discretization in hand, we face a choice. We could discretize time as well – creating a fully discrete algebraic map like $u_i^{n+1} = f(u_j^n)$ – or we could realize that we have reduced the problem to one we have already solved.

For instance, suppose we're interested in the diffusion equation, $\partial_t u = D\partial_x^2 u$. If we imagine discretizing *only* space and leaving time as a continuous variable, we transform our PDE into a massive system of coupled ODEs:

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} = \frac{D}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right). \tag{11.9}$$

Looking closely at this equation, we could treat the value of the field at grid point i, u_i, as the "position" of a particle; thus, this is just an equation of motion. The term on the right looks exactly like a force law where every particle is connected to its nearest neighbors by springs.

This technique is called the *Method of Lines*. We have a state vector $\mathbf{y}$ composed of an array of all of the grid points in our system, $\{u_1, u_2, \ldots, u_N\}$. The spatial derivatives define a function

[95] "famous"

$\mathbf{F}(\mathbf{y})$ that returns the time derivative of this state vector. This means we can reuse our entire ODE toolbox! We can pass a suitably defined `DiffusionForceCalculator` and a `RungeKutta4` integrator directly to a `run_simulation!` function. This approach is quite common, and is (for instance) precisely the approach I once took in a study of PDEs used to model liquid crystals [54].

Chapter 12

Systems evolving in one dimension: a chapter that needs a real title

Under construction

This chapter is still under construction – see handwritten notes

In this chapter we'll look at a few PDEs in one spatial dimension, to explore some topics.

12.1 Parabolic equations / stability crisis

Consider the diffusion equation again: $\partial_t \phi = D\partial_x^2 \phi$.

If we do our spatial discretization (using centered finite differences for accuracy) for the method of lines and consider the simple forward Euler method to solve the resulting ODE, we have the following difference equation for each grid point:

$$\frac{\phi_j^{n+1} - \phi_j^n}{\Delta t} = \frac{D}{\Delta x^2}\left[\phi_{j+1}^n - 2\phi_j^n + \phi_{j-1}^n\right]. \tag{12.1}$$

Combining the constants into $\alpha = D\Delta t/(\Delta x^2)$, this is

$$\phi_j^{n+1} = \phi_j^n + \alpha\left[\phi_{j+1}^n - 2\phi_j^n + \phi_{j-1}^n\right]. \tag{12.2}$$

12.1.1 Stability analysis

We saw when studying planetary motion that the finite approximations to the continuous equations of motion could lead to inaccurate results, but is that all we have to worry about? Floating point arithmetic is not the same as math with real numbers, so how does the finite accuracy of our computations affect the quality of our solutions. We'll answer this with von Neumann stability analysis.

The Martians

Historical color here – what a remarkable bunch, and the source of a good joke.

Here's our perspective. Let Φ be the *mathematical* solution to the finite difference equation above, Eq. (12.2), and define the error at timestep n and gridpoint j as $\varepsilon_j^n = \phi_j^n - \Phi_j^n$. How does this error term itself behave? Because Eq. (12.2) is linear, the error is, in fact, controlled by the same update rule:

$$\varepsilon_j^{n+1} = \varepsilon_j^n + \alpha\left[\varepsilon_{j+1}^n - 2\varepsilon_j^n + \varepsilon_{j-1}^n\right]. \tag{12.3}$$

Does this error grow over time? Shrink?

As always, life is easier to analyze in Fourier space (i.e., consider decomposing our solutions into the fundamental Fourier modes). Suppose we consider an error term which is one of these Fourier modes at some time:

$$\varepsilon_j^n = \xi^n \exp\left(ik(j\Delta x)\right), \tag{12.4}$$

where ξ^n is the amplitude of the mode at some time – we want to know if this amplitude grows or shrinks as time evolves.

Defining $\xi \equiv \xi^{n+1}/\xi^n$, if we substitute Eq. (12.4) into Eq. (12.3) we find

$$\xi = 1 + \alpha\left(e^{ik\Delta x} + e^{-ik\Delta x} - 2\right) = 1 - 4\alpha\sin^2\left(\frac{k\Delta x}{2}\right). \tag{12.5}$$

That is, modes of different wavevector have their amplitude change in different ways, and stability requires that $|\xi| \leq 1$ for *all possible* k. Since $\sin^2$ and α are both positive, clearly $\xi < 1$; we really just have to worry about whether $\xi > -1$ or not.

The worst case – i.e., the mode most likely to cause problems – is the $k\Delta x = \pi$, shortest-possible-wavelength mode, for which $\sin^2 = 1$. This leaves us with the following: for all modes to be stable, we have

$$\alpha \leq \frac{1}{2} \quad \Rightarrow \quad \Delta t \leq \frac{\Delta x^2}{2D}. \tag{12.6}$$

This gives us a speed-limit on our explicit solvers, and for problems like this one it is a bit restrictive: in order to study space at a resolution twice as fine, we need to reduce the timestep by a factor of *four*. This is not great, forcing us to "care" about timescales that are much shorter than the physical ones we think might be interesting.

12.1.2 Implicit methods

"Stiff" numerical problems, like diffusion, demand smaller timesteps than the *physics* we care about. Here, for instance, we see that if we refine the spatial discretization by a factor of ten, we must reduce our timestep by a factor of 100 – we quickly find ourselves spending all of our computational effort satisfying stability rather than making forward progress on the physical problem we're interested in.

As one solution, we can turn to *implicit* rather than explicit methods. Instead of using the forward Euler for our time derivative, look at using backwards Euler, $\partial_t \phi^n \approx \frac{\phi^n - \phi^{n-1}}{\Delta t}$. The

finite difference equation becomes (BTCS):

$$\frac{\phi_j^{n+1} - \phi_j^n}{\Delta t} = \frac{D}{\Delta x^2}\left(\phi_{j+1}^{n+1} - 2\phi_j^{n+1} + \phi_{j-1}^{n+1}\right). \tag{12.7}$$

This is an implicit equation, in which the value of the field at some gridpoint at the next timestep depends on the value of the field at the next timestep in many locations.

We rearrange these terms (and again using $\alpha = D\Delta t/\Delta x^2$) to get:

$$-\alpha\phi_{j+1}^{n+1} + (1+2\alpha)\phi_j^{n+1} - \alpha\phi_{j-1}^{n+1} = \phi_j^n. \tag{12.8}$$

This is just a a matrix equation, $\mathbf{A}\vec{\phi}^{n+1} = \vec{\phi}^n$, where

$$A = \begin{pmatrix} \ddots & \ddots & \ddots & & \\ & -\alpha & \beta & -\alpha & \\ & & -\alpha & \beta & -\alpha \\ & & & \ddots & \ddots & \ddots \end{pmatrix}. \tag{12.9}$$

Here, just to get the columns to line up nicely, we've defined $\beta = 1 + 2\alpha$. This, indeed, reflects our comment about turning derivatives into linear algebra.

In general, if there are N grid points, it might cost $\mathcal{O}(N^3)$ to solve a matrix equation (i.e., performing Gauss-Jordan elimination / inverting the matrix directly). This matrix has a special "tridiagonal" structure, and for such matrices the equation is actually only $\mathcal{O}(N)$ – the same complexity as an explicit Euler-like pass!

What's the payoff? If we again do a stability analysis, we discover that this approach is unconditionally *stable*! That means we can take Δt to be as large as we want – 10^9 years! – and your simulation won't explode. Of course, it might not be accurate, but it will not be unstable.

What are the trade-offs? Well, for one, this is (arguably) harder to code up than the forward Euler version. But there is also a "numerical damping" feature of this and many other implicit methods: they tend to decrease the amplitude of the *solution* and not just the error term, smoothing out features too quickly.

Root of PDE method proliferation: tackling the trade-off between accuracy and stability, with different methods needed to situate on that continuum for different parabolic / elliptic / hyperbolic equations.

12.2 Hyperbolic equations / stability crisis

Let's try another one: advection! $\partial_t\phi + v\partial_x\phi = 0$ (pollution being carried by fluid. Let's say $v > 0$ (left-to-right flow))

Note: exact solution is easy: just translate initial shape to the right: $\phi(x,t) = f(x - vt)$

FTCS approach ($\sigma = \frac{v\Delta t}{\Delta x}$):

$$\phi_j^{n+1} = \phi_j^n - \frac{v\Delta t}{2\Delta x}\left[\phi_{j+1}^n - \phi_{j-1}^n\right]. \tag{12.10}$$

Same error analysis:

$$\begin{aligned}\xi &= 1 - \frac{\sigma}{2}\left(e^{ik\Delta x} - e^{-ik\Delta x}\right) \\ &= 1 - i\sigma \sin(k\Delta x)\end{aligned}$$

We need $|\xi|^2 \leq 1$, but $|\xi|^2 = 1 + \sigma^2 \sin^2(k\Delta x)$? That is *unconditionally unstable*!

Where did we go wrong? We traded mathematical accuracy (centered spatial diff) for physics (causality): our update rule looked at ϕ_{j+1}, but info was coming from ϕ_{j-1}.

12.2.1 Upwind schemes

For $v > 0$, use backwards diff for space: less accurate, but respects physics. Our update rule becomes

$$\phi_j^{n+1} = \phi_j^n - \sigma(\phi_j^n - \phi_{j-1}^n). \tag{12.11}$$

Applying the same error analysis, we get

$$\xi = 1 - \sigma(1 - e^{-ik\Delta x})$$

$$|\xi|^2 \leq 1 \quad \Rightarrow \quad 0 \leq \frac{v\Delta t}{\Delta x} \leq 1$$

This is the "Courant-Friedrichs-Lewy" condition, and is equivalent here to $v\Delta t \leq \Delta x$. That is: don't let stuff move too much in each time step.

Fundamental tradeoff: accuracy vs stability. Upwind will smear out features over time, etc.

12.3 The onset of chaos: nonlinearity and shocks

Inviscid Burger's eqn; wave speed depends on the solution itself. Nonlinearities *rightarrow* another can of worms.

12.4 The zoo of methods

Finite differences and basic implicit solvers are just the tip of the iceberg. Different PDEs demand different solutions to sit along their own comfortable continuum of tradeoffs between accuracy and stability. Thus ,while finite differences are a nice, direct translation of calculus to code, it is rarely the first choice for modern industrial or research codes. Real-world problems at the forefront of research rarely happen on perfect rectangular grides, and "smoothness" is often a luxury we don't actually possess. In addition to the zoo of PDEs types, there is also a zoo of methods we might encounter in the wild.

12.4.1 Finite Volume Method (FVM)

The Logic: Instead of approximating the derivative at a point, FVM approximates the *integral* over a cell. It relies on the divergence theorem: the rate of change in a cell equals the net flux through its faces.

The Why: It guarantees local conservation of mass and momentum by construction. If flux leaves cell A, it *must* enter cell B. This makes it the standard for Computational Fluid Dynamics (CFD) in engineering (e.g., simulating flow over an airplane wing).

12.4.2 Finite Element Method (FEM)

The Logic: We approximate the solution as a sum of basis functions (like small tents pitched on the grid points) and minimize the error in an integral sense.

The Why: It excels at complex geometries. Unlike FD grids which struggle with curved boundaries, FEM meshes can be unstructured triangles or tetrahedra that conform perfectly to a human bone or a car chassis. It dominates structural mechanics.

12.4.3 Spectral Methods

The Logic: Instead of local grid points, we represent the solution as a sum of global basis functions (like sines and cosines via the FFT).

The Why: *Spectral accuracy.* If the solution is smooth, the error decays exponentially ($O(e^{-N})$) rather than polynomially ($O(\Delta x^2)$). It is unbeatable for simple geometries (like a periodic box), making it the tool of choice for fundamental turbulence research and weather prediction on spheres.

12.4.4 Lattice Boltzmann (LBM)

The Logic: Instead of solving the macroscopic Navier-Stokes equations directly, LBM simulates fictive particles jumping between lattice sites and colliding, designing the collision rules so that the continuum limit recovers fluid dynamics.

The Why: It is incredibly easy to parallelize (local collisions) and handles complex boundaries (like porous media or blood flow in capillaries) naturally.

Chapter 13

Steady states and flows in two dimensions

Under construction

This chapter is still under construction – see handwritten notes

13.1 Elliptic equations: solving for equilibrium

13.1.1 Iterative solution methods

13.2 A glimpse of computational fluid dynamics

13.2.1 The Navier-Stokes equations

13.2.2 Anatomy of a CFD Solver

13.3 Coda: Frontiers in PDE research

The naive thought is that, after 300 years, differential equations would be "solved." Nope: method development is arguably more active now than ever. Here are three major frontiers where physicists and applied mathematicians are currently working:

13.3.1 High-order and spectral element methods

Researchers are trying to combine the geometric flexibility of FEM with the accuracy of Spectral methods.

Techniques like *Discontinuous Galerkin (DG)* allow for high-precision simulations of wave propagation (e.g., gravitational waves from black hole mergers) on complex, adaptive meshes.

13.3.2 Multiphysics and multiscale methods

Many modern problems couple problems in which the relevant physics happens on vastly different scales. For instance, simulating a fusion reactor requires coupling the fluid dynamics of the plasma (macroscopic), the electromagnetism of the confinement field, and the kinetic transport of neutrons (microscopic). Direct approaches are limited to have discretization scales set by the fastest and most microscopic processes, making these multi-scale problems prohibitively expensive.

Developing "interconnects" that allow, e.g., a Finite Volume fluid solver to talk to a Monte Carlo particle transport code without violating conservation laws is a massive engineering challenge.

13.3.3 Machine learning and the curse of dimensionality

Standard grid-based methods scale poorly with dimension (N^d). Solving the Schrödinger equation for two electrons is easy (6 dimensions); solving it for 100 electrons is impossible. *Physics-Informed Neural Networks (PINNs)* and *Neural Operators* are attempting to bypass the grid entirely, using neural networks as mesh-free function approximators to solve high-dimensional PDEs that were previously intractable.

Module IV

Module 4: Random numbers and Monte Carlo methods

When we talked about algorithmic complexity, we thought about time, space, and parallelization as fundamental resources. There is another, more subtle resource: *randomness*. If we allow our algorithms access to a source of random numbers, we can unlock solutions to problems that would otherwise be computationally intractable. We can think of the ability to make a random choice as a computational primitive that, like the introduction of a `for` loop or an `if` statement, fundamentally expands what we can easily achieve. We can trade the guarantee of a deterministic answer for a high-quality statistical one, designing algorithms that get around the "curse of dimensionality" by sampling the problem space instead of trying to exhaustively explore it.

Figure IV.1: Caravaggio's *The Cardsharps*, depicting the use of a not-so-random number generator. (Painting: c. 1594. Oil on canvas. Kimbell Art Museum, Fort Worth. Image via Google Arts & Culture)

You might worry: if our computers are fundamentally deterministic machines, where will we find a source of random numbers? We'll start this module by exploring pseudorandom number generators (PRNGs), which are deterministic algorithms that aim to produce long sequences of numbers that appear statistically indistinguishable from a truly random sequence. This apparent limitation is a benefit for scientific studies: by specifying a "seed" we can get our PRNGs to reproduce exactly the same sequence of "random" numbers every time. This means we can both use randomness as a resource *and* guarantee reproducible results.

With this tool in hand, we'll cover classic applications in physics, ranging from direct use in estimating numerical integrals to Markov Chain Monte Carlo (MCMC) methods for simulating particles in space or spins on a lattice. We'll close by combining the Hamiltonian dynamics from Module II with these MCMC methods to build a Hamiltonian Monte Carlo algorithm and apply it to one of the central tasks of modern science: Bayesian parameter inference.

For a deeper dive into the methods and physics discussed in this module, consider the following references [2, 3, 55].

Chapter 14

Pseudorandomness and Monte Carlo integration

What does randomness mean in a deterministic computer, and how can we harness this peculiar resource to solve concrete problems? We'll first explore the first question by looking at simple pseudorandom number generator – not because we should ever use it for research, but because seeing how it works (and how it fails) will help us understand the nature of the tools we're relying on. We'll also see, in a recurring theme, just how close we are to scraping the boundaries of current research. The second half of the chapter will focus on a classic application that addresses the second question. There we'll use randomness to estimate numerical integrals that might otherwise take the age of the universe to deterministically evaluate.

14.1 Pseudorandom number generators

A true random number generator would be a physical device that draws on a truly unpredictable process – perhaps one that watches for the radioactive decay of a sample. These "hardware random number generators" or "true random number generators" have a long history[96]. Such devices have value, but (1) they typically produce random samples at a rate that is far too low for our purposes and (2) by their very nature they are inherently non-reproducible. Instead, we'll rely on PRNGs – a combination of data and algorithm that produces a completely reproducible sequence of apparently random numbers given an initial state. The quality of PRNGs is judged by how well the output sequence passes a battery of statistical tests: to what extent and in which ways does that sequence have the properties you would expect of a true RNG?

14.1.1 A simple PRNG: linear congruential generators

At their core, all PRNGs are state machines: they hold an internal state, and each time you ask for a number they perform some mathematical operation to first update their state and then

[96] One early example used a disk spinning at some high rate (which could "if necessary, be made constant to a high degree of approximation by means of a tuning fork"!) in the dark. The disk was periodically lit up so that a human could try to transcribe the digit they happened to see every three or four seconds. This was a hilarious method, but it was explicitly and favorably compared with selecting "random" digits from a telephone book [56].

transform that state into some output. One of the oldest and simplest of these is the Linear Congruential Generator (LCG) [57]. It generates a sequence of integers with the following simple recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m. \tag{14.1}$$

Here X_n is the current internal state of the generator, and the constants a, c, and m are parameters the describe different LCGs. The choice of these "magic numbers" is critical: poor choices lead to a very poor quality PRNG. An example of such a generator is shown in code block 14.1 – it uses a modulus $m = 2^{32}$ so that the properties of computer integer arithmetic automatically handles the modulus operation, but this is just a convenience.

```
mutable struct LCG
    state::UInt32
    a::UInt32
    c::UInt32
    # We'll use a modulus of 2^32, and directly use `/` below
end
function LCG()
    seed = time_ns() & 0xFFFFFFFF
    # numbers from cc65's `rand()` function
    return lcg(UInt32(seed), UInt32(16843009), UInt32(3014898611))
end

function rand_uint!(rng::LCG)
    rng.state = rng.a * rng.state + rng.c
    return Int(rng.state)
end
function rand_real!(rng::LCG)
    rng.state = rng.a * rng.state + rng.c
    return Float64(rng.state) / Float64(typemax(UInt32))
end
```

Code block 14.1: A simple LCG. A mutable struct holds the state and parameters of our generator, and a simple constructor uses (a) 32 bits from the current system time and (b) magic numbers for a and c (constants taken from cc65's `rand` function, released under the zlib license). The `rand_uint!` and `rand_real` functions mutate the state of the LCG, and return pseudorandom positive integers in $[0, 2^{32})$ and floats in $[0, 1)$, respectively.

The memory footprint of this RNG is small – just a single 32 bits for the state and 64 bits to hold the parameters – and the operations needed to generate random numbers are extremely fast. For convenience I've used the computer's internal time as the initial state of the generator – in an actual application you would choose a specific seed (or at least have a mechanism for recording the seed you used). Figure 14.1 shows the result of generating a few thousand real numbers with this generator. The results in the left panel look reasonable: at a minimum, the generator uniformly samples the space.

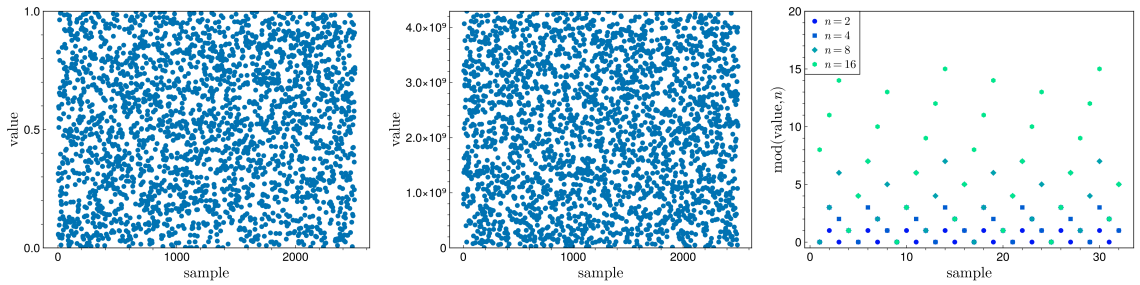

Figure 14.1: Samples from the LCG in code block 14.1. (Left) 2500 floating point samples, roughly uniformly distributed in the unit interval. (Center) 2500 integer samples, again roughly uniformly distributed in the space of `UInt32`s. (Right) Integer samples modulo small even numbers, where deterministic correlations in the integer sequence is seen.

Sadly, "uniformly samples the output domain" is not the only statistical property we usually want. The flaws of this particular LCG are on display when looking at the actual integer sequence of the internal state. Imagine using it to flip an imaginary coin; one method might be to assign "heads" to all of the even numbers and "tails" to all of the odd numbers. The result, shown in the right panel of Fig. 14.1, is highly suspicious (as, indeed, are all of the "modulo small even numbers" properties of the internal state). Problems like this were particularly problematic in early uses of LCGs for Monte Carlo problems [58].

14.1.2 Modern approaches

The precise failure mode of different LCGs depends sensitively on the magic numbers that define them – some have fixed points or short limit cycles, eventually repeating the same output sequence forever, some have the property of Fig. 14.1, in which not all of the bits of the state have the same statistical properties, and so on. Better approaches might start with an LCG but add nonlinearity in the function that maps the internal state to the output, or build nonlinearity into the state transformation itself. It will come as little surprise, then, that different programming languages have different default implementations for their `rand` function.

For isntance, python and many other languages use a "Mersenne Twister" generator [59], whose internal state and parameters take up a relatively whopping 20032 bits, but which has a cycle length (the number of output states that can be generated before the RNG repeats itself) of $2^{19937} - 1$. Numpy uses a "permuted congruential generator" [60], which uses a LCG with a power-of-two modulus but then adds a transformation between the state and the output to maintain the speed and light footprint of an LCG while avoiding their worst statistical properties. Julia uses the quite recent `Xoshiro256++` generator [61], which takes the 256 bits of internal state and performs particular sequences of xor, shift, and rotate operations on the bits. I bring all of this up to emphasize that generation of random numbers continues to be an active and evolving area of research: there are many competing demands on a PRNG – high statistical quality, fast, minimal internal state, possibly cryptographically secure – and different implementations balance those demands differently.

Do not roll your own RNG

Unless you are actively doing research on pseudorandom number generation, in your own code just use a well-tested library implementation for your RNG. Many a scientific "result" in the bad old days was caused by poor statistical properties or buggy implementations of a RNG.

14.1.3 Reproducibility and Seeding

As mentioned in the module introduction, the fact that PRNGs are deterministic should be viewed not as a flaw but an essential feature for scientific computing. By controlling the initial state, or "seed" of a PRNG, we can guarantee that we will get the exact same sequence of random numbers every time we run our code.

In practice, if we don't provide a seed most programming languages will initialize the PRNG with a source of entropy – a common choice is the one demonstrated in code block 14.1, in which the current nanosecond count on a system clock is used. This is why a program that uses this default will produce different outcomes each time it is run. You should think of this as non-negotiable in your scientific work. You can set the seed for Julia's default RNG like so:

```
julia> using Random

julia> Random.seed!(123321)
```

This will guarantee that the stream of numbers resulting from subsequent calls to `rand()` will be identical[97]. Explicitly setting the seed – or using a RNG to choose the seed but then recording the value used – is a basic practice for reproducible scientific work.

14.1.4 Generating non-uniform random numbers

If we were stuck with generating pseudorandom floats and ints uniformly in some range our toolbox would feel a bit limited. Many physical processes correspond to distributions that are not uniform, so is there a way to directly sample from some of these more interesting distributions?

Perhaps the most fundamental method is known as *inverse transform sampling*. The technique relies on the probability density function (PDF), $p(x)$, and its corresponding cumulative distribution function (CDF). The CDF, $P(x)$, gives the probability that a random variable X drawn from $p(x)$ will have a value less than or or equal to x:

$$P(x) = \int_{-\infty}^{x} p(x')\,\mathrm{d}x'. \tag{14.2}$$

[97] An important caveat is that the implementation details of the RNG may change between due to bug fixes, speed improvements, or algorithmic changes to Julia. Thus, the stream of numbers will be the same with a fixed seed on a fixed *version of Julia*.

By definition, the CDF is a monotonically increasing function whose range is between zero and one.

The key insight is that if we draw a random variable X from the distribution $p(x)$, then the transformed variable $y = P(X)$ will be uniformly distributed in the interval $[0, 1]$. Inverse transform sampling runs this logic in reverse: we generate a uniform random number $y \in [0, 1]$ – which we already know how to do – and then transform that into a random number x drawn from $p(x)$ by calculating $x = P^{-1}(y)$. As long as we can analytically calculate the inverse of the target CDF, we're done!

A canonical example of this is to generate samples from the exponential distribution. The probability distribution itself is parameterized by λ, and given by

$$p(x; \lambda) = \lambda e^{-\lambda x}, \quad \text{for } x \geq 0. \tag{14.3}$$

The CDF is found by directly integrating this up to x:

$$P(x) = \int_0^x \lambda e^{-\lambda x'} \, \mathrm{d}x' = 1 - e^{-\lambda x}. \tag{14.4}$$

Finally, we find the inverse function by setting $y = P(x)$ and solving for x:

$$y = 1 - e^{-\lambda x} \quad \Rightarrow x \quad = -\frac{1}{\lambda} \ln(1 - y).$$

We now have a simple algorithm: draw a uniform random number and apply this formula. In code (and using Julia's built in random number generator), this is as simple as:

```
using Random
function rand_exp(lambda)
    y = rand()
    return -log(1.0 - y) / lambda
end
```

This technique is powerful, but it's limited to distributions where the CDF can be easily inverted in closed form. An example of a common distribution without this property is the Gaussian distribution: its CDF involves the error function, which has no simple inverse. For this and other distributions, other techniques are required. Common ones involve variable transformation method (such as the Box-Muller transform [62], which generates pairs of uniformly distributed random numbers and maps them to pairs of Gaussian distributed random numbers) and rejection-sampling methods (which work by drawing samples from simple distributions and probabilistically accepting or rejecting them to match a target distribution). Eventually we'll see that the distributions we're interested in in the coming chapters are too complex for any of these methods to work well; for now our key takeaway is that our simple uniform PRNG can serve as the fundamental building block from which all other distributions can be constructed.

14.2 Application: Monte Carlo Integration

We now turn to the second question posed at the start of this chapter: how to use randomness to solve a deterministic problem? One of the most classic, straightforward, and powerful applications is Monte Carlo integration.

14.2.1 The Basic Idea: Throwing Darts

We, in fact, already did this in its basic form in Section 4.3, where we calculated π by "throwing darts" at a unit square and counting the fraction that landed inside an inscribed circle. The first generalization of this idea is to find the value of a one-dimensional definite integral, $\int_a^b f(x)\,\mathrm{d}x$, by computing the average value of the function over the interval and multiplying it by the length of the interval. The Monte Carlo approach estimates the average by sampling the function at N points chosen at random uniformly throughout the interval:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\cdot\frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{14.5}$$

14.2.2 Convergence and the curse of dimensionality

Why is this a good idea? Mathematically, the central limit theorem tells us that as long as we have a reasonable, well-behaved functions, the error in our estimate of the average value will behave like the standard error of the mean of a random sample. As a result, the error of our estimate of the integral with N samples will scale as

$$\mathrm{Error}_{\mathrm{MC}} \propto \frac{1}{\sqrt{N}}. \tag{14.6}$$

This is an error which converges slowly: to reduce the error by a factor of 10, we need to generate 100 times as many samples. This should be contrasted with a deterministic method like even the simple trapezoid rule. There, splitting the interval into N subintervals (i.e., in a setting in which we perform the same number of *function evaluations* as in the MC estimate above), the error scales as

$$\mathrm{Error}_{\mathrm{trap.}} \propto \frac{1}{N^2}. \tag{14.7}$$

Perhaps the question above should have said "*Is* this a good idea?" The power becomes apparent when we move to higher dimensions. Consider integrating a polite function over a d-dimensional unit hypercube, $[0,1]^d$. The error in the Monte Carlo approach is still controlled by the central limit theorem, and has an error which remains $\mathcal{O}(N^{-1/2})$. Similarly, the error in the trapezoid rule continues to be set by *the size of the discrete subregions*: we have N function evaluations, but them must be split into a grid composed of M points along each dimension: $N = M^d$. Since the error scales like $1/M^2$, we find that the error for our integral scales like $\mathcal{O}(N^{-2/d})$. Said another way: achieving a certain precision in $d = 1$ with the trapezoid rule might require $N = 10$ points with the trapezoid rule and $N = 10^4$ points with the MC approach. To get that same precision in 10 dimensions with the trapezoid rule would require $N = 10^{10}$

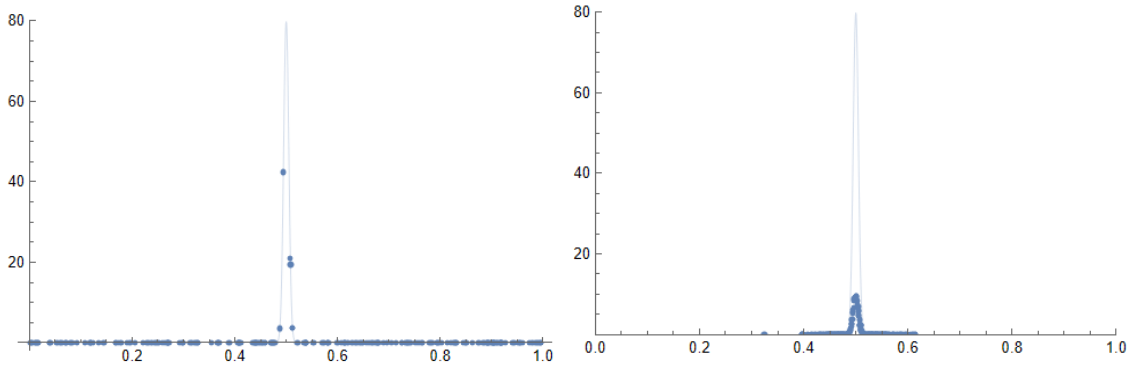

Figure 14.2: (Left) A narrow Gaussian function (thin line) together with 200 uniformly randomly sampled locations in the unit interval. (Right) The same function, but now the points are themselves sampled from a Gaussian distribution and reweighted accordingly.

function evaluations, whereas the MC method still needs only of order 10^4 (albeit with a larger prefactor, often one which scales *linearly* with d).

Thus, as d increases, the deterministic method becomes *exponentially* slower if we try to maintain the same accuracy. This catastrophically bad scaling in high dimensions is famously called the "curse of dimensionality." For typical, well-behaved functions, a dimensionality of $d = 4$ is the tipping point where the effectiveness of the deterministic vs random methods flips. For the extraordinarily high-dimensional integrals that appear in fields like statistical mechanics, MC methods are not just a better option, they are the *only* option.

14.3 Importance sampling

The Monte Carlo approach is incredibly powerful, but there is something a bit brute-force and, perhaps, inelegant about how it uniformly samples random numbers in the interval of interest. We often know at least some things about the functions we are trying to integrate, and for functions with sharp peaks or other localized features, the majority of the N samples we use might be "wasted" in regions where the integrand is essentially zero. Multiple evaluations of the function in those regions don't contribute very much to our understanding of the integral, but they still cost computational time and slow down the convergence.

To be concrete, consider the Monte Carlo integration of a narrow Gaussian function, like the one shown in Fig. 14.2. With the direct MC method, we randomly sample the value of the function throughout the unit interval, but only a small fraction of these samples actually contribute to the integrand. The asymptotic scaling of the error here hasn't changed, but the prefactor has gotten massively worse, as our estimate of the integral is dominated by the statistical noise from the handful of lucky "hits" when we sample in the correct region.

14.3.1 Biasing and re-weighting

We can do better by no longer sampling uniformly but instead focusing our computational effort where it matters – the is the core idea behind *importance sampling*. Instead of drawing samples

from a uniform distribution, we instead draw them from a different probability distribution, $p(x)$, which puts more of its weight in regions where the integrand $f(x)$ deviates from zero.

In order not to let this non-uniform distribution bias our estimate, we rewrite our original integral like so:

$$I = \int \frac{f(x)}{p(x)} p(x)\,\mathrm{d}x. \tag{14.8}$$

We can interpret this as the expectation value of the function $g(x) = f(x)/p(x)$ with respect to the probability distribution $p(x)$. This lets us write our MC estimator as

$$I \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}, \quad \text{where } x_i \text{ is drawn from } p(x). \tag{14.9}$$

Algorithmically, this is straightforward. We draw a sample x_i from the "importance" distribution $p(x)$, and then add the value $f(x_i)/p(x_i)$ to a running sum. The $1/p(x_i)$ is the "re-weighting" factor that corrects for the biased sampling by giving less weight to samples drawn from high-probability regions and more weight to to the rare sample drawn from low-probability regions. This is visualized in the right panel of Fig. 14.2 – we again draw 200 points but now from a narrow Gaussian distribution close to our target distribution, and re-weight those points accordingly. The resulting estimate is, in fact, on average much closer to the correct answer for a fixed number of samples.

Why is this better? The goal of importance sampling is to reduce the variance in our estimate. A low-variance estimator is one that converges quickly to the correct answer with few samples, and it can be shown that the variance of our estimator in Eq. (14.9) is minimized with a particular choice for our sampling distribution:

$$p_{\text{ideal}}(x) = \frac{|f(x)|}{\int |f(x)|\,\mathrm{d}x}. \tag{14.10}$$

This equation says that the perfect sample distribution is the one proportional to the absolute value of the function we are trying to integrate. In the case where $f(x)$ itself is non-negative, it even tells us that we could get the exact answer with a single sample!

Remarkable, but also perhaps a bit circular? Of course, that's only because constructing this perfect sampling distribution requires knowing how it is normalized, $\int |f(x)|\,\mathrm{d}x$, and this is basically the integral we are trying to estimate in the first place. Thus, the are of importance sampling is not finding the perfect distribution, but to choose a simpler distribution $p(x)$ that (a) we can easily sample from and (b) mimics the shape of our original function better than a uniform distribution does.

14.3.2 The bridge to statistical physics

This insight leads directly to the heart of computational statistical mechanics. Calculating a thermodynamic observable for an equilibrium system corresponds to a evaluating a high-dimensional integral over all possible states of the system. For a system at temperature T, the probability of being in a particular state μ with energy $E(\mu)$ is given by the famous Boltzmann distribution,

$$p(\mu) \propto e^{-\beta E(\mu)}, \tag{14.11}$$

where $\beta = (k_B T)^{-1}$. Let's heuristically think of this as the *ultimate importance distribution*, provided to us by nature itself. It tells us that the important states – the ones that contribute the most to any thermodynamic average – are the low-energy states. Our goal, then, is to perform a Monte Carlo calculation by drawing samples from this distribution.

And yet, for any non-trivial system, the space of possible states is unimaginably vast – considering a collection of N particles in a box of volume V, every different arrangement of particles in that volume is its own state. The volume to the power of avogadro's number is a terrifyingly large number, and that's before we say anything about the momentum degrees of freedom in the system. How could we possibly draw independent samples directly from such a high-dimensional monstrosity? In a related challenge, the normalization constant for the Boltzmann distribution is the partition function,

$$Z = \sum_{\mu} e^{-\beta E(\mu)}. \tag{14.12}$$

Sum over *all* of the immense number of these states? How?

This all seems... hard. How will we perform importance sampling with a distribution we don't know how to sample from, and whose normalization constant we can't hope to compute? This is the problem that the Metropolis algorithm [38], and the field of Markov Chain Monte Carlo (MCMC) was invented to solve – that's what we'll turn to in the next chapter.

Chapter 15

The Metropolis algorithm

In the previous chapter we ended with a powerful but puzzling idea: the key to Monte Carlo integration is importance sampling – a way of focusing our computational effort on the "important" regions of a problem – and the optimal distribution to use for the importance distribution is one which already contains information about the very integral you are trying to perform.

For a system in thermal equilibrium nature hands us the Boltzmann distribution, $p(\mu) \propto e^{-\beta E(\mu)}$. To calculate any thermodynamic averages for our system we need to draw samples from this distribution, but how? It is defined over a state space of terrifyingly, stare-into-the-abyss large dimensionality. We typically cannot even calculate its normalization constant (the partition function Z), let alone draw independent samples from it

This chapter introduces a solution which is a workhorse of computational statistical physics: the *Metropolis* algorithm. This is a brilliant method that lets us generate a sequence of states that, in the long term, correctly samples from the Boltzmann distribution even without knowing Z. We'll first develop this algorithm from first principles, and then apply it to canonical model systems.

15.1 Markov chain Monte Carlo

The conceptual leap is actually to abandon the idea of drawing *independent* samples altogether. Instead we will try to construct a random walk – which we'll call a *Markov chain* – that explores the state space of our system. A Markov chain is a sequence of states in which the next state depends only on the current state (and not any of the states that came before); it is a "memoryless" process. The challenge will be to design the rules of the random walk so that, in the long run, the fraction of the time the system spends in any particular state μ is directly proportional to the true probability of being in that state, $p(\mu)$.

The idea of constructing these memoryless sequence of states goes back to the work of Markov in the early 20th century [63]. In an early application, Markov analyzed 20000 characters from the writing of Pushkin's poem Eugene Onegin [64]. His method was to first define the "states" of his system – is the current character a vowel or a consonant – and then empirically measure the probability of moving between them. This creates a *transition matrix*, T, where T_{ij} is the probability of transitioning from state i to j.

For example, we can analyze the complete works of Shakespeare to construct the transition

From \ To	a	e	i	o	u	C
a	$\frac{957}{312\,083}$	$\frac{1176}{312\,083}$	$\frac{12\,809}{312\,083}$	$\frac{404}{312\,083}$	$\frac{4429}{312\,083}$	$\frac{292\,308}{312\,083}$
e	$\frac{15\,519}{162\,506}$	$\frac{7881}{162\,506}$	$\frac{19\,777}{487\,518}$	$\frac{5527}{243\,759}$	$\frac{621}{81\,253}$	$\frac{127\,587}{162\,506}$
i	$\frac{4179}{136\,025}$	$\frac{5619}{136\,025}$	$\frac{297}{54\,410}$	$\frac{5166}{136\,025}$	$\frac{277}{27\,205}$	$\frac{237\,867}{272\,050}$
o	$\frac{1178}{83\,535}$	$\frac{571}{66\,828}$	$\frac{1228}{83\,535}$	$\frac{7211}{167\,070}$	$\frac{60\,179}{334\,140}$	$\frac{12\,353}{16\,707}$
u	$\frac{6}{217}$	$\frac{5581}{137\,795}$	$\frac{594}{27\,559}$	$\frac{852}{137\,795}$	$\frac{159}{137\,795}$	$\frac{124\,423}{137\,795}$
C	$\frac{82\,563}{842\,369}$	$\frac{13\,425}{76\,579}$	$\frac{76\,699}{842\,369}$	$\frac{297\,076}{2\,527\,107}$	$\frac{66\,532}{2\,527\,107}$	$\frac{1\,242\,688}{2\,527\,107}$

Figure 15.1: The transition matrix between some vowels and any consonant, as determined by analyzing the complete works of Shakespeare [67]. The entries of the dominant eigenvector predict the empirical distribution of vowels and consonants in the source within a precision of 10^{-7}. However, a random walk using this matrix produces samples like: "gsfsaedptoawfjtqxiiz." Very poetic, but also demonstrative of the fact that the model captures "equilibrium statistics" but not the long-range correlations of real language.

matrix shown in Fig. 15.1, which shows the probability of transitioning in the text from the vowels $\{a, e, i, o, u\}$ to each other or to any consonant. If we represent the current state as the probability vector (e.g., the vector $\{1, 0, 0, 0, 0, 0\}$, corresponding to "the current state is definitely the letter a"), we can find the probability of the next state simply by multiplying by the transition matrix. The power of the Markov chain is that is we apply this matrix repeatedly, the state vector will inevitably converge to a unique *stationary distribution* – a vector that no longer changes upon further multiplication by T. This stationary distribution is, by definition, the dominant eigenvector of the transition matrix [65, 66] and, as Markov showed, it almost perfectly predicts the overall frequency of vowels and consonants in the text.

This result is both powerful and surprising. As the gibberish in the caption shows, our simple Markov model has failed to learn much about the actual structure of the English language. But it has perfectly succeeded at a more narrow task: finding the correct equilibrium distribution of the characters themselves. This is an important lesson: the Markov chain may not be a perfect replica of the system; it is a carefully crafted tool designed to reproduce some of the correct statistical properties in the long time limit.

The process illustrated above – using a Markov chain to generate samples from a Monte Carlo calculation of some property – is known as Markov Chain Monte Carlo (MCMC). To emphasis again: we will construct a simulation in which the state at step $n+1$ depends only on its state at step n, and not on the full history of the trajectory. The central task will be to define transition probabilities, $T(\mu \to \mu')$ for moving from state μ to state μ'. This set of probabilities must be carefully crafted so that the stationary distribution of the Markov chain (i.e., the distribution of states it settles into after running for a long time) is precisely the Boltzmann distribution.

15.1.1 Engineering the stationary distribution

We're making progress, but there are still major questions here. In the case of letter transition probabilities we just empirically measured the transition matrix – this was a descriptive model. But how do we turn a similar idea into a *prescriptive* method, in which we do not just measure but design transition rules that will guarantee the stationary distribution matches the target? A full treatment, along with the actual conditions under which this whole machinery will work, would require a deep dive into the theory of ergodic Markov chains. Instead, we'll think through some clear necessary conditions, and the physical shortcut that will make our algorithmic approach more tractable.

One condition that must clearly be satisfied is the condition of *global balance*. Suppose you already have a stationary distribution, in which the probability of observing your system in state i perfectly matches the true equilibrium probability, which we'll denote π_i. In order for the continued application of the transition matrix not to take you *away* from the stationary distribution, the total flow of probability into and out of each state must be balanced: mathematically, global balance corresponds to

$$\sum_i T_{ij}\pi_i = \pi_j = \sum_k T_{jk}\pi_k. \tag{15.1}$$

This is a very general kind of "no net flux of probability" condition, but it is in general hard to use to to actually build the transition matrix.

We can make our life easier by imposing a stricter requirement: rather than having this global balance of inflowing and outflowing probability, we demand that the total flow of probability between every pair of states is balance. This *detailed balance* condition – which clearly also satisfies the demands of global balance – is

$$T_{ij}\pi_i = T_{ji}\pi_j. \tag{15.2}$$

Markov chains satisfying detailed balance are called reversible Markov chains. And indeed, when modeling physical systems with microscopically reversible dynamics, there is a natural justification for adopting the additional constraints that detailed balance imposes.

15.1.2 The Metropolis-Hastings algorithm

With that as background, let's return to our target of studying thermodynamical systems that can reside in microstates μ. Although there is a relationship, the approach we're about to outline marks a fundamental shift from the importance-sampling strategy. As we'll see, the Metropolis [38] algorithm gives up on the idea of drawing independent samples from a simple distribution and then reweighting to correct for bias; instead it provides a way to generate a sequence of correlated samples drawn directly from the correct, complex distribution.

So: the Metropolis algorithm splits the task of constructing the transition rule $T(\mu \to \mu')$ into two steps: a *proposal* and an *acceptance/rejection* step. The proposal step selects a potential new state, μ', starting from the current state μ. This is done according to some probability distribution $g(\mu \to \mu')$. For many physical problems we have the luxury of choosing simple symmetric proposal distributions – uniformly randomly picking a spin to flip or a particle to displace, for instance – and for such distributions we have $g(\mu \to \mu') = g(\mu' \to \mu)$. The

next step decides whether to accept the proposed state update or not, which happens with probability $A(\mu \to \mu')$.

In combination we have a total transition probability of $T(\mu \to \mu') = g(\mu \to \mu') \cdot A(\mu \to \mu')$. Substituting this into the detailed balance condition for a symmetric proposal distribution gives

$$p(\mu) \cdot g(\mu \to \mu') \cdot A(\mu \to \mu') = p(\mu') \cdot g(\mu' \to \mu) \cdot A(\mu' \to \mu) \quad \Rightarrow \quad \frac{A(\mu \to \mu')}{A(\mu' \to \mu)} = \frac{p(\mu')}{p(\mu)}. \tag{15.3}$$

That is: for this subdivision of the problem, imposing detailed balance corresponds to a condition on the ratio of the acceptance probabilities.

A brilliantly simple choice was made by Metropolis et al.:

$$A(\mu \to \mu') = \min\left(1, \frac{p(\mu')}{p(\mu)}\right), \tag{15.4}$$

i.e., proposals to more probable states are always accepted, and proposals to less probable states are accepted some of the time[98] Intuitively, this is a strong starting point: such a Markov chain can easily get near local minima, but also with some probability escape them and explore the entire state space.

For our physical system and for nature's importance function, $p(\mu) = \exp(-\beta E(\mu))/Z$, the ratio of probabilities is

$$\frac{p(\mu')}{p(\mu)} = \frac{Z}{Z} \frac{e^{-\beta E(\mu')}}{e^{-\beta E(\mu)}} = e^{-\beta \Delta E}.$$

The partition function which we didn't know in the first place cancels out, leaving us with a powerful result: the acceptance probability depends only on the change in energy, which is often a local and easily computable quantity.

This simple acceptance rule represents an interesting shift in our strategy relative to the importance-sampling problem we ended Chapter 14 with. There, direct importance sampling required us to draw samples from a simple distribution, $q(\mu)$, and reweight them to evaluate an integral. In the context of computing ensemble averages with respect to a Boltzmann weight, $\langle A(\mu) \rangle = \int A(\mu) p(\mu) \, \mathrm{d}\mu$, a resulting estimator would be

$$\langle A(\mu) \rangle \approx \frac{1}{N} \sum_{i=1}^{N} \frac{A(\mu_i) p(\mu_i)}{q(\mu_i)}.$$

The problem is that, well, we have no good way to draw independent samples from $p(\mu)$ itself, and no idea of a good, simple $q(\mu)$ to sample from instead.

The Metropolis algorithm solves this problem with a very different strategy. Instead of asking, "How do we choose a reasonable q and then correct for sampling from the wrong distribution?", it provides a mechanism for generating *correlated* samples $\{\mu_i\}$ that are drawn

[98] The assumption of a symmetric proposal distribution is the distinction between the original Metropolis algorithm and the more general "Metropolis-Hastings" algorithm [68]. The Hastings generalization allows for asymmetric proposals by modifying the acceptance ratio we're about to derive to include the proposal probabilities, with an acceptance ratio given in Eq. (16.1).

directly from the true distribution. It achieves the "ideal" sampling scenario, and our estimator for any thermodynamic average becomes a simple, unweighted sum:

$$\langle A \rangle \approx \frac{1}{N} \sum_{i=1}^{N} A(\mu_i). \tag{15.5}$$

The price we pay is that our samples are no longer independent; this introduces subtleties we will address in the next section. But with that said, we can now state the complete algorithm for our Markov chain:

1. Start in state μ_i with energy $E(\mu_i)$.
2. Propose a trial move to a new state μ'.
3. Evaluate ΔE.

4a. If $\Delta E \leq 0$, accept the move: $\mu_{i+i} = \mu'$.

4b. If $\Delta E > 0$, generate a uniform random number $r \in [0, 1)$. If $r < e^{-\beta \Delta E}$ accept the move: $\mu_{i+i} = \mu'$. If not, reject the move and set $\mu_{i+1} = \mu_i$.

5 . Repeat.

15.2 Case Study I: The Ising model

For our first case study using the technology above, let's study the Ising model, which is typically introduced as a toy model for magnetic systems. Schematically depicted in Fig. 15.2, the Ising model consists of a set of N spins, $\mu = \{s_i\}$. There are 2^N possible states μ, and so clearly for a lattice of even quite modest size there are more states than we could even enumerate, let alone actually actual work with. This is a canonical use case for MCMC-based sampling: we'll use the Metropolis algorithm to generate a sequence containing a tiny fraction of the possible states, but we will generate them with the *correct* distribution of probabilities.

These spins interact with their neighbors, and may also be coupled to an external magnetic field. The Hamiltonian governing the spins is

$$\mathcal{H} = -J \sum_{\langle ij \rangle} s_i s_j - h \sum_i s_i, \tag{15.6}$$

where J determines the strength of the spin-spin interaction, h is the external field, and $\sum_{\langle ij \rangle}$ indicates a sum over all spins i and j that are neighbors of each other. This model has many variations[99], but for now we will focus on the simplest case of nearest-neighbor spins with constant J on a hyper-cubic lattice.

[99] Is J the same for all pairs of spins? What lattice do the spins live on? Should neighbors only be nearest-neighbor, or should we include other interactions?

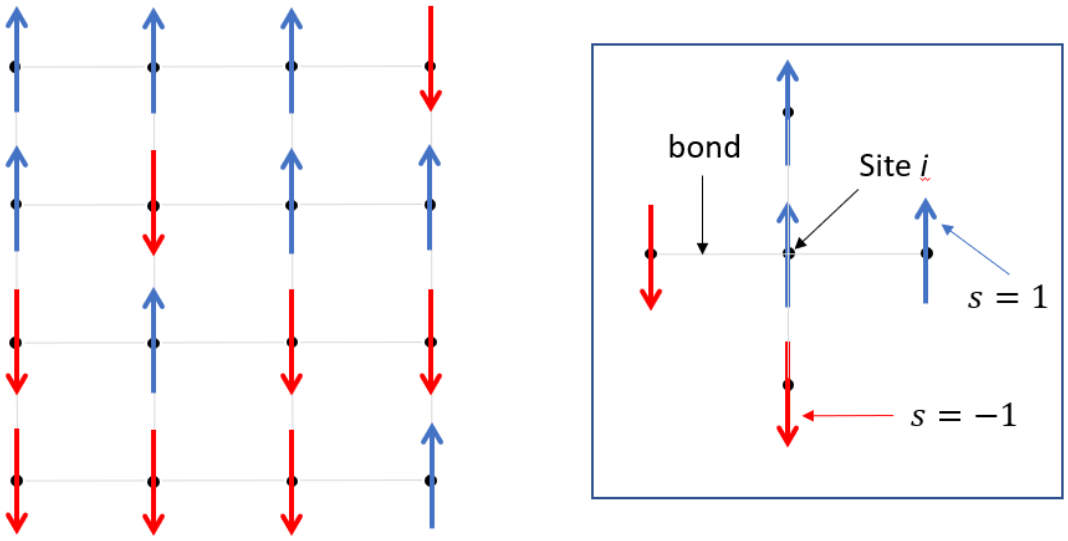

Figure 15.2: (Left) A schematic representation of the square lattice Ising model. (Right) On each site "spins" are either up or down, corresponding to $s_i = \pm 1$. Sites with a bond between them are included in the term which sums over neighboring sites in the Hamiltonian.

The Ising model

Although seemingly simple, this model is connected to some of the deepest insights in 20th century physics. It was introduced by Wilhelm Lenz [69], and Lenz's student's dissertation constructed the exact solution for a one-dimensional lattice [70]. Ising stopped doing research, partly because he thought he had proved that the model he spent so much intellectual effort on had no physical relevance[a]; only much later in life did he find out that this model had become incredibly influential and, indeed, a cornerstone that helped build modern statistical physics.

[a] His career was also tragically cut short by religious persecution. He was forbidden from teaching and conducting research, fled Germany, and performed forced labor dismantling the Maginot line railroad before eventually emigrating to the US [71].

15.2.1 A top-down design

Let's implement a Metropolis Monte Carlo simulation of the simple Ising model on a hypercubic lattice, but lets do so in a way that anticipates some of the natural ways we might want to extend it. Just as in Chapter 8, we'll think about a flexible top-down design: we'll first think about how we want to structure a single *metropolis_step!*, and what that design contract in turn guides the rest of our lower-level implementation. Throughout we'll lean on multiple dispatch to write general code that nevertheless efficiently specializes on the particular structures we want to work with.

We begin by thinking through what the general statement of the Metropolis algorithm tells us we will need. Clearly we will need something for the state of whatever system we have under

study, and we will need a way of implementing the proposal function, g, that can suggest a state μ' given the state μ_i. We also need a way of calculating differences in energy – and hence energies in the first place, and then an implementation of the acceptance function A. Without even knowing that we have a lattice model in mind, we can already directly write this high-level description:

```
function metropolis_step!(system, hamiltonian, move::AbstractMCMove;
β=1.0)
    trial_move_info = propose_move(system, hamiltonian, move)

    ΔE = calculate_ΔE(system, trial_move_info, hamiltonian, move)

    if ΔE <= 0.0 || rand() < exp(-β * ΔE)
        accept_move!(system, trial_move_info, move)
    end
    return nothing
end
```

Code block 15.1: A possible implementation of a general step in a Metropolis Monte Carlo simulation.

15.2.2 System data structures

As before, the very first thing our high-level API tells us we need is a "system," and we could go in many different directions here. This is a key point in our work where we could try to write hyper-specialized code for the square lattice Ising model above, or try to write something extremely generic but not necessarily particularly performant, and so on. In this case, let's allow ourselves to be guided by the knowledge that we'll certainly be studying lattice-based spin systems, but we *also* have spent time thinking about particle based systems. It might be interesting to be able to compare both an ODE- and Monte-Carlo-based approaches to a particle system, so even though this case study is for a lattice model, let's make sure we can handle different types of systems.

Not a problem. Let's begin by setting up some basic lattice functionality, and then build a system on top of that foundation. We'll start with an abstract lattice type, and some of the basic functions we will be sure to implement for any concrete lattice we want to study:

```
abstract type AbstractLattice end

function num_sites(lattice::AbstractLattice) end

function neighbors!(lattice::AbstractLattice, site::Int) end
```

Here we see the first concession to generality: in a fixed lattice there might not be a need for `neighbors!` function: the neighbors of each site in the lattice are fixed forever! Why, then, are

we defining a mutating function? It's because if we want to use our code to also work for a dynamical particle system, our eventual `calculate_ΔE` function needs an interface that can work for systems whose neighbors can change at every time step. This is a real-world design trade-off that both illustrates the "meet-in-the-middle" design process and the potential cost of abstraction: do we prioritze maximum performance for a specific case, or do we build a more general interface that can be reused for other problems?

```
struct HypercubicLattice{D} <: AbstractLattice
    dims::NTuple{D, Int}
    neighbors::Vector{Int}
    neighbor_list::Vector{Vector{Int}}
end

function HypercubicLattice(dims::NTuple{D, Int}) where {D}
  N = prod(dims)
  neighs = Vector{Int}(undef,2*D)
  neighbor_list = [Vector{Int}() for _ in 1:N]
  cartesian_indices = CartesianIndices(dims)
  linear_indices = LinearIndices(dims)
  for i in 1:N
    current_idx = cartesian_indices[i]
    for d in 1:D
      for offset in (-1, 1)
        mod_offset = mod1(current_idx[d] + offset,dims[d])
        neigh_idx = Base.setindex(current_idx, mod_offset, d)
        push!(neighbor_list[i], linear_indices[neigh_idx])
      end
    end
  end
    HypercubicLattice{D}(dims, neighs,neighbor_list)
end
num_sites(lattice::HypercubicLattice) = prod(lattice.dims)

function neighbors!(lattice::HypercubicLattice, site::Int)
    lattice.neighbors .= lattice.neighbor_list[site]
end
```

Code block 15.2: A concrete `Hypercubic` lattice with precomputed nearest neighbors. Please forgive the abuse of consistent indentation – I wanted all of these lines to fit onto a single line of text in these notes.

Code block 15.2 gives an implementation of a lattice of fixed (nearest-neighbor) topology that, nevertheless, is optimized to compute nearest-neighbor lists without allocations[100].

[100] Whether this is, in fact, the optimal approach might depend on the details. Perhaps precomputing neighbor

This is most of the work we need to do for now; with a lattice defined we can set up a simple system of spins that includes both the spins (of different possible types!) themselves and the lattice they live on.

```
abstract type AbstractSystem end

struct SpinSystem{T, L <: AbstractLattice} <: AbstractSystem
    spins::Vector{T}
    lattice::L
end
```

15.2.3 Monte Carlo moves

Next we could tackle the Hamiltonian and the corresponding energy function (which is the next argument in the `metropolis_step!` function), but since it's needed on the first line let's look ahead to the `move::AbstractMCMove` and `propose_move` functions. We discussed earlier that there is a tremendous flexibility in what the proposal function g in the Metropolis algorithm can actually be: it could propose completely new states uniformly regardless of the current state of the system, or it could propose the most minor changes to the current state, or anything in between. The choice will strongly influence *how quickly* the sequence of states converges to the stationary distribution, and how correlated the sequence is, but choosing a good proposal function is part of the art of Monte Carlo methods. We'll make sure we set up the technology to implement different proposed types of state changes, and then concretely implement the simplest possible MC move for an Ising model: picking any single spine and flipping its sign. The types might look like this:

lists is better, or perhaps using the logic of the lattice to recompute neighbor indices on the fly is better than memory lookups. As always, if you are interested in this level of optimization you should *measure and benchmark*.

```
abstract type AbstractMCMove end

struct SingleSpinFlip <: AbstractMCMove end

struct SpinFlipInfo
    index::Int
end

function propose_move(system::SpinSystem{T}, hamiltonian,
                      move::SingleSpinFlip)
    random_index = rand(1:length(system.spins))
    return SpinFlipInfo(random_index)
end

function accept_move!(system::SpinSystem{T}, move_info::SpinFlipInfo,
                      move::SingleSpinFlip) where {T <: Integer}
        system.spins[move_info.index] *= -one(T)
end
```

You can see that, while we were at it, we defined not only the idea of a "single spin flip" MC move, but also the information that gets returned from the `propose_move` function, and what it would mean to accept a proposed move.

15.2.4 Energy calculations

All that remains for a complete implementation are the structures that encode a Hamiltonian and the functions that compute energy differences! For the first, we can set up a standard type hierarchy of abstract and concrete ways of calculating the energy – concrete Hamiltonians are where we'll store the parameters themselves – along with the function we'll need to implement for each concrete type:

```
abstract type AbstractEnergy end
# function we'll need to define for each concrete Hamiltonian:
function calculate_ΔE(system::AbstractSystem, move_info,
                      h::AbstractEnergy, move::AbstractMCMove) end
```

For our nearest-neighbor Ising model, the corresponding concrete struct and implementation of the ΔE calculation when a spin is flipped is shown in code block 15.3.

And with that, we're done! We have a complete, still efficient implementation of a Metropolis simulation of the Ising model on a hypercubic lattice. It may have taken us slightly longer to build this than the monolithic script would have been, but still: in less than 100 lines of code we have a structure that, as we'll see in the next case study, is actually *very* easy to generalize to other systems!

```
struct IsingHamiltonian{T} <: AbstractEnergy
    J::T
    h::T
end

function calculate_ΔE(system::SpinSystem, move_info::SpinFlipInfo,
                      ham::IsingHamiltonian, move::SingleSpinFlip)
    neighbors!(system.lattice, move_info.index)
    spin_i = system.spins[move_info.index]

    neighbor_sum = sum(@view system.spins[system.lattice.neighbors])
    ΔE = 2.0 * spin_i * (ham.J * neighbor_sum + ham.h)
    return ΔE
end
```

Code block 15.3: A concrete `IsingHamiltonian` structure, and the corresponding (fairly simple!) calculation of how much the energy changes when a spin is flipped.

15.2.5 Analyzing MCMC data

We have a nice, modular framework, and running this simulation will produce a long sequence of states, $\{\mu_0, \mu_1, \mu_2, ...\}$. The central promise of the Metropolis method is that after a long enough number of iterations this sequence is a representative sample from the Boltzmann distribution. But how do we actually turn this raw sequence of microscopic configurations into meaningful measurements for macroscopic observables, such as the average magnetization $\langle M \rangle$? The naive approach would be to calculate the magnetization $M = \sum_i^N s_i$ for every state in our sequence and take the average. This is dangerously wrong.

The first problem is the same one we encountered in our particle-based simulations. Our simulation began from an artificial initial state – artificial because we probably did not pick a representative configuration that has a high likelihood of appearing in the steady-state distribution. As the simulation runs, the system must "relax" or "thermalize" as it moves from this unusual initial state to a statistical steady state. If we were to include this early set of states in our average, the memory of the initial conditions would bias the final result, but the solution is simple: as before, we discard these initial samples. As before, the standard practice is to monitor a bulk observable as a function of the number of Monte Carlo steps, measure a timescale associated with the decay of this observable to its equilibrium value (Fig. 10.3, and discard all states in the first several multiples of this time scale.

The second problem is more subtle. After discarding the initial equilibration data, it would be tempting to take the remaining samples and then calculate the average and standard error of the mean. But this is also wrong: by their very construction, Markov chains generate a sequence of *correlated* states[101]. For instance, in the single-spin-flip version of the code above, state μ_{i+1} is either *exactly the same* as state μ_i, or it is different in the value of exactly one spin.

[101] Indeed, this is a live issue when studying molecular dynamics: configurations separated by a short time are *also* highly correlated with each other.

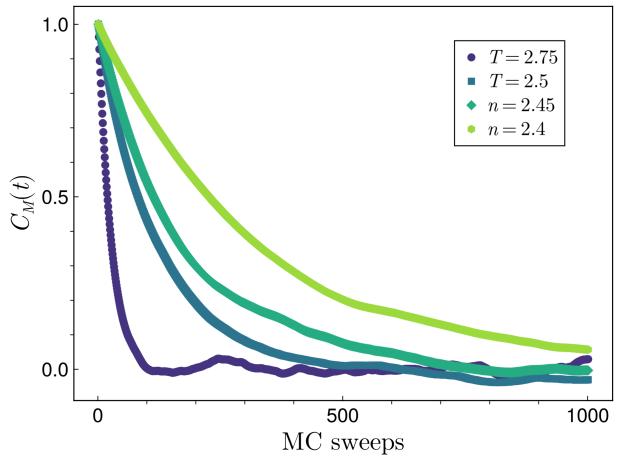

Figure 15.3: The autocorrelation function in the two-dimensional Ising model with $J = 1$, $h = 0$, at various temperatures, as measured via MC simulations on a 50×50 lattice.

These are hardly independent samples, and if treated them as independent we would radically underestimate the true statistical error in our measurements. That is: our error bars would be meaninglessly small, leaving us with a completely false sense of precision.

Instead, we have to actually (again) *measure* how long the memory of a typical state in the sequence persists. We can quantify this with an autocorrelation function, $C(t)$, associated with some observable. For an observable like the magnetization, $M(t)$, we can define the autocorrelation $C_M(t)$ by first defining the average $\langle M \rangle$, where $\langle \ldots \rangle$ means to average over all configurations at time t_0 within equilibrated set. We then define the instantaneous fluctuation of the observable, $\delta M(t) = M(t) - \langle M \rangle$. Finally, we measure the correlations of these fluctuations, which are then typically normalized so that the function begins at one:

$$C_M(t) == \frac{\langle \delta M(t_0) \delta M(t_0 + t) \rangle}{\langle (\delta M)^2 \rangle}, \tag{15.7}$$

Figure 15.3 shows a few such autocorrelation functions at different temperatures in a two-dimensional Ising model

This function measures how correlated the fluctuations in magnetization are with themselves, separated by a "time" of t MC steps. We have normalized the function so that $C(0) = 1$, but that is just a convention. Such functions typically decays exponentially[102], giving us a characteristic autocorrelation time τ. This is, roughly speaking, the number of MCMC steps we have to take before the system has "forgotten" an earlier state, and thus after which we can imagine that we have gathered another independent sample for our measurement.

The practitioner's analysis pipeline

Putting this all together, we have the following very standard pattern:

1. **Generate:** Run long MCMC simulations, saving the state (or the observables you care about) frequently.

[102] Perhaps you are detecting a theme, here

2. **Equilibrate:** Plot the observable versus the number of MC steps, identify the equilibration time t_{eq}, and ignore all data generated before this point in the following.
3. **Decorrelate:** Calculate the autocorrelation time, τ, for the remaining equilibrated data.
4. **Decimate:** Create a smaller data set of effectively independent samples by taking data points from the equilibrated trajectory only every τ (or every 2τ, or every 3τ, etc).
5. **Evaluate:** Calculate the average and the standard error of the mean only on this final data set.

15.3 Case Study II: Particle-based systems

While spin systems are a canonical use case for Metropolis Monte Carlo, the algorithms is much more general. We can apply the exact same MCMC machinery to the kinds of particle-based systems we studied in Module II, providing a powerful alternative to the molecular dynamics methods we developed there.

The goal, here, is fundamentally different. In MD we integrated the equations of motion to follow the true, physical evolution of the system over time, generating a *trajectory*. In MC, our goal is to correctly sample the equilibrium distribution of *configurations*. The sequence of states in an MCMC simulation is a carefully constructed random walk through phase space, and not a physical path. While we often talk about "Monte Carlo time" – typically the number of Metropolis steps taken – this is merely a measure of computational effort and not a faithful representation of the dynamics of the system over actual time.

Part of the beauty of the modular framework we have built is that we can implement this entirely new simulation paradigm by reusing almost all of our existing code. The logic for particle data structures, periodic boundary conditions, and potential energy calculations from Chapter 10 can and should be directly reused. With that as a foundation, we only need to write a few new methods to define the Monte Carlo "move" in this case, and we'll have a complete, functional simulation.

15.3.1 Implementation details

First, we define our `ParticleSystem` and `Particle` structs exactly as we already have. We'll even make the decision to keep the `velocity` field in the particles; while it will go unused in our pure MC simulation, keeping the data structure the same ensures compatibility[103].

[103] And allows for future extensions, such as "hybrid Monte Carlo" methods [72].

```
struct Particle{D,T}
    position::SVector{D,T}
    velocity::SVector{D,T}
    mass::T
end

struct ParticleSystem{D,T} <: AbstractSystem
    particles::Vector{Particle{D,T}}
    boundary_conditions::AbstractBoundaryConditions
end
```

Let's define the concrete MC move we will use for our particle-based simulations. One choice, analogous to the single spin flips in the Ising model, is to randomly displace a particle by some amount. This structure holds a key parameter – the maximum displacement in any direction that we will propose – which will be important to tune so that the system can explore a reasonable amount of phase space and that move get accepted a reasonable amount of the time.

```
struct SingleParticleMove{T} <: AbstractMCMove
    max_displacement::T
end
struct ParticleMoveInfo{D,T}
    index::Int
    new_position::SVector{D,T}
end
```

The proposal and acceptance functions are straightforward implementations that dispatch on this new move type, with the proposal function returning new structure that, as before, contains the information needed to accept a move if the Metropolis criteria is satisfied.

```
function propose_move(system::ParticleSystem{D,T}, hamiltonian,
                      move::SingleParticleMove) where {D,T}

    idx = rand(1:length(system.particles))
    p_old = system.particles[idx]
    random_vec = rand(SVector{D,T}) .* 2.0 .- 1.0
    displacement = random_vec * move.max_displacement

    return ParticleMoveInfo(idx, p_old.position + displacement)
end

function accept_move!(system::ParticleSystem,
    move_info::ParticleMoveInfo, move::SingleParticleMove)
    p_old = system.particles[move_info.index]

    p_new = Particle(move_info.new_position, p_old.velocity,
p_old.mass)

    system.particles[move_info.index] = p_new
end
```

To handle the physics, we introduce a `ParticleHamiltonian` that holds an instance of a new potential type. This allows us to plug in a `LennardJones` struct, a `HarmonicRepulsion` struct, or any other potential that conforms to our interface. It's one extra layer of indirection, but it allows us to decouple the physical law from the simulation algorithm, exactly as we might want

```
abstract type AbstractHamiltonian end

struct LennardJones{T} <: AbstractHamiltonian
    epsilon::T
    sigma::T
    cutoff_sq::T
end

struct ParticleHamiltonian{P<:AbstractHamiltonian} <: AbstractEnergy
    potential::P
    # Additional structures for, e.g. for cell list calculations
end
```

Finally, we implement the function to calculate the change in energy associated with our proposed move. We adopt the strategy of computing the contribution to the system energy due to the selected particle – not the total energy of the system! – and then recompute what would happen if we temporarily place the particle at the proposed new position. Even for this step we should be sure to reuse all of our (e.g.) cell list technology to make the calculation as efficient as possible, and then we should be *absolutely sure* to restore the system to its original state

before returning the energy difference. This allows us to reuse a set of helper functions while also correctly calculating the δE needed for the Metropolis acceptance criteria.

```
function calculate_ΔE(system::ParticleSystem, info::ParticleMoveInfo,
                      ham::ParticleHamiltonian, move::SingleParticleMove)

    particle_idx = info.index
    # A helper function we need to write
    E_old = _particle_energy(info.index, system, ham)

    p_old = system.particles[info.index]
    p_ghost = Particle(info.new_position, p_old.velocity, p_old.mass)

    system.particles[info.index] = p_ghost
    E_new = _particle_energy(info.index, system, ham)

    system.particles[particle_idx] = p_old
    return E_new - E_old
end
```

15.3.2 Comparing Methods: MC vs. MD

We have now spent time developing two complete simulation engines – MD and MC – that can be applied to study the same particle fluids. When we run both simulations at the same state point (NVT) and calculate structural properties like the radial distribution function, we expect to get results that are statistically identical. This is both a profound validation of both methods, and also a beautiful demonstration of statistical physics: two completely different microscopic processes (deterministic time evolution and a stochastic random walk) have converged to the *same* equilibrium structure.

Both methods work, so – you might ask – which is better? The answer depends on the scientific question you are trying to get at, as they provide fundamentally different kinds of information. MD generates trajectories, and so it is the correct way to study dynamical properties directly: diffusion, viscosity, relaxation times, and so on. It's main disadvantage is that, for system with large energy barriers between different states, it can take an extremely long time for the natural dynamics of the system to explore the entire relevant parts of phase space. In contrast, as we've emphasized, MCMC generates a sequence of configurations, and knows nothing about real time. However, because its "moves" are not constrained to lie along physical paths, you can invent Monte Carlo moves that *much more quickly* sample complex phase spaces. This can make MC methods much more efficient than MD for constructing phase diagrams, evaluating free energies, or finding the equation of state of some system.

In short: they are not really competitors – they are complementary tools. MD tells you how a system gets from state A to B; MC is a powerful shortcut for figuring out the properties of A and B themselves.

Chapter 16

Hamiltonian Monte Carlo and Bayesian inference

In the previous chapters we developed and compared two different simulation paradigms: molecular dynamics, which follows physical trajectories of a system, and Monte Carlo, which performs statistical random walks to explore the steady state distribution of configurations. Notably, in Metropolis Monte Carlo we are free to be wildly creative in how we craft the proposal function $g(\mu \to \mu')$. This freedom will be the key to our next advance.

The standard Metropolis algorithms we introduced in Chapter 15 used simple, local proposals – flipping a single spin, or translating a single particle. These are powerful, but also suffer from a potentially crippling inefficiency: they tend to perform very slow random walks around local neighborhoods of configuration space. Especially in high dimensions, it can take a large amount of time and computational effort for samples in our sequence of states to decorrelate. Perhaps we could start designing smarter proposals, that make take us to distant states that are, at the same time, very likely to be accepted? In this concluding chapter of Module IV, we'll explore an interesting marriage of the MD and MC techniques we've been studying. We'll use the resulting methods to solve a canonical scientific task: estimating the parameters of a model that best fit a set of data.

16.1 Hybrid Monte Carlo

In what may seem like a surprising detour, let's first extend the code that performs our Metropolis simulations of particle-based systems by introducing a new move. Rather than have the state μ' arise by picking a particle at random from configuration μ_i and translating it a little bit, Let's imagine starting at μ_i and generating a proposed state μ' by *performing a molecular dynamics simulation of some duration!* Rather than just moving a single random particle by a small amount as we did in Section 15.3, this new state involves all of the particles following some physical dynamics to move to a new position. This will likely decorrelate the sequence of MC states much more quickly than individual particle-based moves. We can also do this in a way that yields a high acceptance probability for our proposals: if we use *symplectic integrators* from Section 9.2 we know that the energy of the system will be approximately conserved, and so the energy differences between states will remain quite small.

This idea of using Hamiltonian dynamics as part of MCMC was originally called "hybrid Monte Carlo" [72, 73], and with the work we have do it is actually remarkably easy to implement. As shown in code block 16.1, the primary task is to extend the `AbstractMCMove` part of our type hierarchy by introducing a `HamiltonianMove` struct. Our strategy will be to store in this structure all of the components – a way of calculating forces, a method of integration, etc – that we need to plug into our existing API for running MD simulations. As has sometimes been the case, we will make a concession to performance in these definitions: since the "move information" would in principle need to contain the entire configuration of the system, and it would be expensive to copy that much data all of the time, we keep a scratch space for data in the `HamiltonianMove` struct itself.

```
struct HamiltonianMove{F <: AbstractForceCalculator,
                       I <: AbstractIntegrator, D, T} <: AbstractMCMove
    force_calculator::F
    integrator::I
    temperature::T
    time_step_size::T
    n_steps::Int
    scratch_system::ParticleSystem{D,T} #to avoid making copies
end

struct HamiltonianMoveInfo{T}
    delta_e::T
end
```

Code block 16.1: A kind of MC move that involves simulating a particle system via molecular dynamics.

The one potentially unexpected wrinkle we have introduced here is at the beginning of the `propose_move` function: before launching our simulation we randomize particle velocities by drawing them from the appropriate Maxwell-Boltzmann distribution. Why? Well, molecular dynamics is in the "Hastings" regime of Metropolis-Hastings, in that outside of extremely unusual pairs of states there is no reason to think that MD will result in a symmetric proposal distribution: $g(\mu \to \mu') \neq g(\mu' \to \mu)$. For such asymmetric proposals the acceptance probabilitiy must be [68]

$$A(\mu \to \mu') = \min\left(1, \frac{p(\mu')g(\mu' \to \mu)}{p(\mu)g(\mu \to \mu')}\right). \tag{16.1}$$

That seems to be a problem: if we maintain the particle velocities, MD is *deterministic* – from configuration μ there is a state μ' that MD *always* proposes, and in general if we have gone from state μ to μ' the odds that continuing the MD trajectory for the same amount of time will take you from μ' back to μ would require a laughably unlikely conspiracy. But if $g(\mu' \to \mu) = 0$, we are left with an acceptance probability of zero; that doesn't do anybody any good.

One elegant solution, outlined in code block 16.2 might be to have the proposed move integrate the equations of motion forward and then flip all of the particle momenta. This

```
function propose_move(system::ParticleSystem{D,T}, hamiltonian,
                      move::HamiltonianMove{F, I, D, T}) where {F, I, D, T}
    t = sqrt(move.temperature)
    for (i, p) in enumerate(system.particles)
        move.scratch_system.particles[i] = Particle(
            p.position, randn(SVector{D,T}) * t/sqrt(p.mass), p.mass
        )
    end

    old_energy = total_energy(move.scratch_system, hamiltonian)
    run_simulation!(move.scratch_system, move.force_calculator,
                    move.integrator, move.time_step_size, move.n_steps)

    new_energy = total_energy(move.scratch_system, hamiltonian)
    return HamiltonianMoveInfo(new_energy-old_energy)
end
```

Code block 16.2: A `propose_move` function for hybrid MC.

would bring us back to a symmetric proposal distribution, but without further tinkering we would end up constructing a Markov chain that only oscillated between two states. A better solution is to recall that in our MCMC approach we are not trying to trace system dynamics, we are trying to sample the phase space. The randomization of momenta at each step is what allows the Markov chain to explore the space effectively, preventing it from getting stuck in deterministic loops. That is, randomizing particle momenta before each MD run is a nice way of simultaneously giving us a symmetric g *and* introducing enough noise to actually explore phase space.

The acceptance step is then a standard Metropolis criteria, but because symplectic integrators *almost* conserve the total Hamiltonian, the change in $\Delta\mathcal{H}$ will be small and the acceptance probability will be high. This is the central trick of HMC.

16.2 Bayesian parameter inference

Forget something as specific as simulating particles for a moment. A much broader task that arises in all of the sciences is *inferring the unknown parameters of some model* from a set of (often noisy) experimental data. The traditional approach to this problem, which I suspect you have already learned, is to cast parameter inference as a point estimation problem: given a model and some data, what is the single best-fit set of parameters that minimizes the difference between the model and the data. Perhaps the most common version of this is to perform a least-squares fit, but more generally one can define a loss function on the difference between the model and the data and try to minimize the loss function with respect to the model's parameters.

Bayesian inference offers an alternate perspective. Rather than trying to find the single

best-fit set of parameters, the goal is to construct the entire *posterior probability distribution*, $p(\vec{\theta}|D)$, which represents the probability of the parameter values $\vec{\theta}$ given the observed data D. This distribution doesn't just give us the most likely parameter values – it gives us the full range of their uncertainties and any correlations that might exist between them.

The foundation for this approach is Bayes' theorem [74, 75]:

$$p(\vec{\theta}|D) = \frac{p(D|\vec{\theta})p(\vec{\theta})}{p(D)}. \tag{16.2}$$

You may not have seen this before, so let's be clear on the notation and on the physical meaning of each of these terms. We've already mentioned the posterior, $p(\vec{\theta}|D)$, and we read those symbols as "the probability of $\vec{\theta}$ given (or conditioned on) D" – this is the quantity we want to compute. The *likelihood* $p(D|\vec{\theta})$ asks, "Given a specific set of parameters $\vec{\theta}$ that go into a model $f(x, \vec{\theta})$, what is the probability of observing our actual data D?"

Answering that question is where our physical model and our assumptions about experimental noise enter. A common starting point is to assume that each of the N data points $y_i \in D$ all are subject to independent Gaussian noise of some standard deviation σ. Under that assumption, the likelihood is given by a product of probabilities:

$$P(D|\vec{\theta}) \propto \prod_{i=1}^{N} \exp\left(-\frac{(y_i - f(x_i, \vec{\theta}))^2}{2\sigma^2}\right) \tag{16.3}$$

The next components of Eq. (16.2) is the *prior*, $p(\vec{\theta})$. The prior represents our knowledge about the parameters *before* we see the data: we might have a "flat" prior if we really have no idea of what the parameters cold be, or an "informed" prior if we have some physical constraints on (e.g., by symmetry) or other knowledge of the values the parameters can take. Finally we have the *evidence*, $p(D) = \int p(D|\vec{\theta})p(\vec{\theta})\,\mathrm{d}\vec{\theta}$, which appears as a normalization constant.

The structure to really burn into your brain is **posterior** $\propto$ **likelihood** $\times$ **prior**. The evidence is a normalization constant that, much like the partition function Z in the previous chapter, is in generally impossible to actually compute. But just like Z, we don't actually need it for sampling! If we use MCMC to explore the space of model parameters, we can make sure that we only ever care about ratios of the posterior, for which the normalization factor cancels out.

While a standard Metropolis random walk could (in principle) explore this parameter space, for even modestly complex models it becomes incredibly inefficient. The simple local moves that work well for the Ising model turn out to be poorly suited to navigating the complex and often strongly correlated "landscapes" of posterior distributions. This inefficiency is not just an inconvenience: it is often the primary barrier to performing a Bayesian analysis at all. This sets the stage for our final application: using a surprising but physically-motivated MCMC algorithm for our Bayesian inference tasks.

16.3 Hamiltonian Monte Carlo

The core idea is going to be to map the Bayesian inference problem onto a Hamiltonian framework [72, 55, 73]. We imagine that the parameters of the model, correspond to the "positions"

of some fictitious particles (typically of unit "mass"), and we furthermore invent some fictitious "momenta," $\{p_i\}$ as addition degrees of freedom. We'll assume these fictitious momenta have a standard kinetic energy term, $K = \sum_i p_i^2/2$. Finally, we want the system to be drawn to areas in which the posterior is large (i.e., likely values of parameters given our data), so we will invent a "potential energy" for our system:

$$U(\vec{\theta}) = -\log p(\vec{\theta}|D). \tag{16.4}$$

This negative log-posterior function, indeed, has "energy minima" at probability maxima.

From here, we can directly plug into our hybrid/Hamiltonian Monte Carlo framework. The one caveat is that Eq. (16.4) is a potential energy rather different from the pairwise potentials we have studied in the context of more normal particle systems. Rather than being a pair-potential, the data almost certainly couple all of "positional" degrees of freedom together (c.f. Eqs. (16.2) and (16.3)). Thus, the calculation of the "force," $F = -\nabla U(\vec{\theta}) = \nabla \log p(\vec{\theta}|D)$, is often a bit thorny.

HMC for inference

In the context of our Bayesian inference problem, the HMC algorithm becomes:

1. Start at parameter state $\vec{\theta}$.
2. Assign random momenta to the parameters, drawing each from a zero mean, unit variance normal distribution.
3. Calculate the current $E_{\text{old}} = \mathcal{H}(\vec{\theta}, \vec{p}) = -\log p(\vec{\theta}|D) + K(\vec{p})$.
4. Simulate Hamiltonian dynamics for M time steps with $\Delta t = \varepsilon$ using a symplectic integrator (e.g., velocity Verlet), to get $\{\vec{\theta}', \vec{p}'\}$.
5. Calculate $E_{\text{new}} = \mathcal{H}(\vec{\theta}', \vec{p}')$.
6. Accept the state θ' with probability $\min\left(1, e^{-\Delta E}\right)$.

Key parameters that require tuning for specific models, priors, and data are the step size and the integration time, Δt and M.

16.3.1 Case study: inferring exponential decay

Let's pause and apply this framework to a common, simple problem we've discussed several times in the last few chapters: determining the parameters of an exponential decay from noisy data. Perhaps we've measured some quantity y at various times t, and we believe that the underlying process is actually exponentially decaying:

$$y(t) = ae^{-bt}. \tag{16.5}$$

Our goal is to infer the initial amplitude a and the inverse timescale b from a set of N data points, (t_i, y_i), assuming that we know the measurement noise (which we will here take to be independent Gaussian noise whose standard deviation is σ).

First, for this model we can transcribe Eq. (16.3) and (up to a constant) write the log-likelihood:

$$\log p(D|a,b) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}\left(y_i - ae^{-bt}\right)^2$$

We also need to *specify our priors*: what do we know or believe about the parameters a and b? Simple, relatively uniformative choices might be that they are positive numbers that we otherwise know nothing about:

$$p(a) \propto 1 \text{ for } a > 0; \quad p(b) \propto 1 \text{ for } b > 0.$$

Combining these and ignoring the constants, we want to sample

$$\log p(a,b|D) \approx -\frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - ae^{-bt_i})^2 \quad (\text{for } a > 0, b > 0) \tag{16.6}$$

This combination of model, likelihood, and prior is one of the few where we can analytically write down the gradient that we need in order to run our "molecular dynamics." We have:

$$-\frac{\partial U}{\partial a} = \frac{1}{\sigma^2}\sum_{i=1}^{N} e^{-bt_i}\left(y_i - ae^{-bt_i}\right) \tag{16.7}$$

$$-\frac{\partial U}{\partial b} = \frac{-1}{\sigma^2}\sum_{i=1}^{N} at_i e^{-bt_i}\left(y_i - ae^{-bt_i}\right) \tag{16.8}$$

$$\tag{16.9}$$

With this in hand, we can directly follow the HMC algorithm above. Starting with some relatively arbitrary initial parameter values, we choose our HMC parameters, implement a `LogPosteriorHamiltonian` for our specific model and then run the `metropolis_step!()` many times.

The output, as with any MC simulation, will be a long list of samples. We need to check for convergence, and measure autocorrelation times to get independent samples. But having done so, we are in a position concretely understand how the space of models intersects with our data. We could create a 2D histogram of the accepted $\{a, b\}$ samples – the densest region should be centered around the true values, and the shape of the distribution reveals both the uncertainties we should have and any correlations between a and b. One would typically find the *marginal posterior distributions* – the histograms of just the a values and of just the b values – and calculate / report the mean value along with other standard statistical measures.

Let's see what this starts to look like *in our code*. The first thing we want to do is add new structs that fit into our type hierarchy for *both* the metropolis acceptance step (for which we need a concrete `AbstractEnergy` which we will dispatch on to compute changes in energy) and for the MD step (for which we need a concrete `AbstractForceCalculator` on which we can dispatch to compute the forces).

```
struct LogPosteriorHamiltonian{M,P} <: AbstractEnergy
    data::Vector{Vector{Float64}}
    data_sigma::Float64
    model::M
    prior::P
end
struct LogPosteriorHamiltonianExpModel{E <: AbstractEnergy}
                                    <: AbstractForceCalculator
       U::E
end
```

These simple versions represent relatively hard-coded versions of these structs – our concrete energy struct has data, a kind of error term for the data, and space for model and prior functions, but the concrete force calculator is quite specific. Code block 16.3 shows an implementation of the `compute_acceleration!` function that dispatches on our exponential model. That code block in particular probably feels a bit different than most of the code we've been writing: rather than a generic function it is hyper-specialized (in this case, for (1) a simple exponential decay model, (2) errors in the data assumed to be independent Gaussians of the same variance, and (3) a flat prior on the model parameters).

```
"""
Specialized: exp model, indep gaussian errors, and a flat prior
"""
function compute_acceleration!(accelerations::Vector{SVector{D,T}},
           system::ParticleSystem{D,T,B},
           calculator::LogPosteriorHamiltonianExpModel) where {D,T,B}
    a = system.particles[1].position[1]
    b = system.particles[2].position[1]
    sigma_squared = calculator.U.data_sigma^2
    acceleration_a = zero(T)
    acceleration_b = zero(T)
    for d in calculator.U.data
        common_factor = exp(-b*d[1]) * (d[2] - a*exp(-b*d[1]))
        acceleration_a += common_factor
        acceleration_b -= a * d[1] * common_factor
    end
    accelerations[1] = SVector{1,T}(acceleration_a / sigma_squared)
    accelerations[2] = SVector{1,T}(acceleration_b / sigma_squared)
    return nothing
end
```

Code block 16.3: A `calculate_acceleration` function for HMC, concretely implementing the mapping from a very specific Bayesian log posterior to the gradient of a energy functional.

It's a little bit inelegant, and yet we can actually tie all of these components together! We can use the same, e.g., `VerletIntegrator` we introduced in Chapter 9 with our new "gradient of

the log posterior for the exponential model" `ForceCalculator` to do the molecular dynamics that represent each metropolis step, and tie it together with exactly the same MCMC code we wrote in Chapter 15. Figure 16.1 shows some of the results of such an analysis, in which the above machinery is applied to "fake data" generated by adding adding uncorrelated Gaussian noise on top of an exponential model.

Noise parameters

This has been a simplified, extremely idealized setting. For instance, it is rare that we have the luxury of actually knowing the true distribution of errors to use for our likelihood function in Eq. (16.3). Much more commonly the noise parameters (like σ) are treated as additional unknown parameters in our model, each with its own prior distribution. This additional complexity fits seemlessly into the HMC framework: we account for these extra parameters when defining the negative log-posterior and calculating its gradient, we we then aim to infer the joint posterior distribution for both the model parameters *and* the noise parameters.

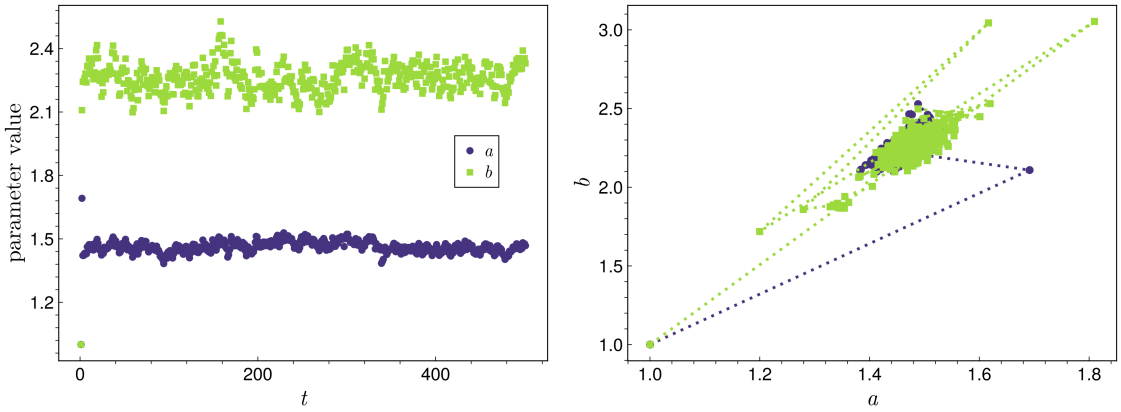

Figure 16.1: **Bayesian inference of model parameters** (Left) traces of the sequence of parameter values after each metropolis step in an HMC algorithm applied to data generated by adding noise on top of an exponentially decaying function. (Right) the trajectory of model parameter points after each metropolis step, with two different values of the HMC tuning parameters (Δt and M).

16.4 Flies in the ointment

Using Hamiltonian Monte Carlo for model parameter inference is a powerful, principled tool in the scientific toolbox. It is, however, not without some sticking points.

16.4.1 The trouble with tuning

First, the trouble with those tuning parameters: how exactly should we choose the integration time and the total run length? These are completely fictitious parameters – we are using artificial

Hamiltonian dynamics to sample space, but its not like the model for our data actually *has* Hamiltonian dynamics – so other than "the Markov chain generates independent samples cheaply" there are precious few objective criteria to help us determine them.

Manually tuning them is challenging. For Δt, too small a value means a very slow, computationally costly exploration of parameters space; too large a value means the integrator error grows and the acceptance rate goes down. For M, too few total steps just performs an inefficient random walk; too many total steps tends to result in trajectories that "U-turn" back on themselves, wasting a lot of our computational effort. What a headache.

A state-of-the-art solution to this problem are techniques like the "No-U-Turn Sampler" (NUTS) [76]. At a high level, the basic idea of it is to adaptively, automatically choose how long to run the Hamiltonian trajectory for *each* proposal. It does this by building a trajectory and stopping when it detects that the path is starting to "turn back" on itself (a tricky thing to quantify in high-dimensional model spaces!). The payoff, though, is that this completely removes the need for manual tuning of HMC parameters, making Bayesian inference accessible and robust for a wide range of problems[104].

16.4.2 The problem with partials

The other problem is related to computing the gradient of the negative log-posterior function we use as the potential energy in our Hamiltonian dynamics. Even in the simplest possible case – an exponential model, completely flat priors, and independent Gaussian errors in our experimental data – computing the relevant partial derivatives with respect to model parameters is at least a little bit tedious to do and error prone to encode. It also feels unsatisfying. One can imagine a vast zoo of potential models you might want to use to fit to data, a similarly large space of possible priors you might have, and a range of ways that the error in the likelihood function might work – do we really have to write down a different concrete implementation of the `compute_acceleration!` function for every single one of these variations? Fortunately, there is an unexpected computational tool that will swoop to the rescue, enabling us to compute the gradients of complex log-posteriors just as readily as we can write down new models. Prepare for Module V!

[104] So: you can and should turn to off-the-shelf solutions for your actual research.

Module V

Module 5: Machine learning in physics

In the previous modules we've focused on simulating physical systems from first principles, building models, integrating equations of motion, and sampling statistical properties. While powerful, this approach led us to a methodological wall: for any non-trivial model, we are stuck manually calculating the gradients of complex functions. This is a tedious, error-prone, and fundamentally unscalable "hand-crank" operation, whether we're writing finite-difference schemes for PDEs or trying to find the "forces" for Hamiltonian Monte Carlo.

This final module introduces a new class of computational tools, broadly grouped under the umbrella of "machine learning." We will begin in Chapter 17 by studying "automatic differentiation" – a way of thinking about gradients that powers modern machine learning and which can be used to replace the tedious hand-crank of manual differentiation. Then in Chapter 18 we will study a foundational machine learning model, the neural network. We'll explore the use of these as powerful general-purpose approximators, building up from the simple "perceptron" to multi-layer networks. Finally, in Chapter 19 we'll put these tools to use. We'll think about classifying the phases of the kinds of spin models we studied in Module IV, and about trying to infer the underlying force laws governing the motion of particles just by looking at particle trajectories.

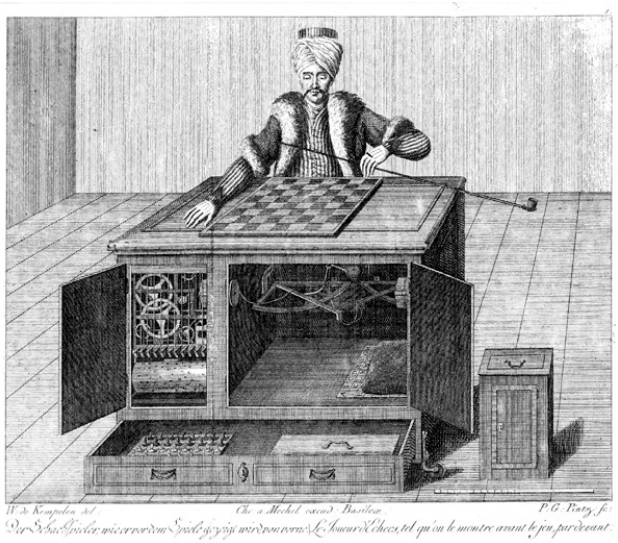

Figure V.1: Artificial intelligence, mechanical intelligence, or clever humans behind the scenes? "Briefe über den Schachspieler des Hrn. von Kempelen." Image in the public domain.

For a deeper dive into the methods discussed in this module, the review by Baydin et al. is the canonical starting point for automatic differentiation [77]. The "Deep Learning" book by Goodfellow, Bengio, and Courville is a comprehensive and standard text for neural networks [78]. For a review focused specifically on these techniques as applied to physics, the "High-bias, low-variance" paper by Mehta et al. is an excellent resource [79].

Chapter 17

Derivatives

Derivatives?

At the end of Chapter 16 we were confronted with a vast zoo of possible models that might reasonably arise, and perhaps you joined me in shuddering at the thought manually differentiating each of them with respect to all possible model parameters and then writing down all of that code. Perhaps more familiar than Bayesian inference of model parameters, we could also consider what modern large language models (LLMs) look like. We'll explore precisely how neural networks and related models are structured and how they work in Chapter 18, but for now it's enough to know that LLMs are often structured as a deep stack of ~ 100 "transformer" layers. Each of these layers is a combination of a "self-attention" mechanism (which, puts different weights on the importance of different words in the current context) and a "feed-forward" network that processes that weighted information. Inputs to the LLMs are converted to a numerical representation, and this information is passed sequentially through each layer.

The goal of model "training" is to make the LLMs final output predict text accurately, guided by a *cost function* that quantifies how wrong the model's predictions currently are. Just as finding a minimum-energy state in a complex physical system requires computing the gradient of an energy with respect to many particle coordinates ($F_i = -\nabla_i E$), training an LLM requires computing the gradient of the cost function with respect to each of the parameters of the model. The first "L" in "large language model" is apt: we could easily be talking about models that involve billions – or even trillions – of parameters that can be adjusted to try to produce high-quality outputs.

17.1 Symbolic differentiation

We're all pretty good at calculus, so your first instinct might be to wonder why we cannot just calculate the derivative of the cost function and code it up? This direct approach, indeed works perfectly well for many physics problems. When simulating particle-based systems in Module II, the derivative of a pairwise potential is often as simple to write down as the potential itself, and hence the computational cost of finding the force is similar to that of finding the energy. It's important to realize that it is not the overall scale – the number of variables to take derivatives with respect to – that is necessarily the problem: performing molecular dynamics simulations of billions of particles is within the scope of what people sometimes do [80].

The problem is the stacked structure used in the machine-learning models described above: parameters in one layer affect the calculations that each subsequent layer performs. This is quite different to, e.g., molecular dynamics simulations in which the position of one particle only affects the forces experienced by particles in some small, local neighborhood. In the layered setting, taking a derivative with respect to a parameter in the first layer requires applying the chain rule through *every subsequent layer*. This would create a symbolic expression for the derivative of literally unimaginable complexity.

Let's see what we mean by doing a quick Fermi estimate. The *Llama* models are a source-available family of large language models developed by Meta and whose architectural details are available [81]. The 70-billion-parameter version has $L = 80$ layers, and each layer's feed-forward network has an intermediate dimension of $D_{\text{ffn}} = 28672$. Even ignoring the complexity of the rest of the model's architecture, the number of terms in the symbolic derivative's "fan-out" scales with the dimensions of each subsequent layer. A lower bound on the number of terms in the expression for the derivative with respect to a parameter in the first layer as $D_{\text{ffn}}^{L-1} \approx 1.38 \times 10^{352}$.

I'm sure you agree that diligently writing down all of those terms isn't a great path forward[105].

17.2 Numerically approximate differentiation

A natural alternate approach is to consider the definition of the derivative,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

and imagine approximating this definition by choosing a small but finite value h for the relative separations of the function's arguments:

```
function finite_difference(f, x, h)
    return (f(x+h)-f(x))/h
end
```

Finite differences

This kind of discrete approximation to a derivative is, of course, quite familiar to us – this is the same forward finite difference we saw when we discussed forward integration of ordinary differential equations in Module II, and we equally well know that more sophisticated variations like centered finite differences are available to us. There are at least two issues. The first is that with this approach we always need to compute the function itself twice. Factors of two in compute time aren't necessarily prohibitive, but could become particularly problematic when we decide to compute the gradient of a function with respect to N variables, we've just signed ourselves up to compute the same function at least $N + 1$ times[106]

[105] Indeed, any time you have to write down more terms than the merely 10^{80} atoms in the universe you have to start asking hard questions. Like what pen you're going to use.

[106] Centered finite differences are more accurate, but that's signing yourself up to compute the function $2N$ times! I hope N isn't in the billions.

The second issue is in choosing a good value for h. It is sometimes convenient to forget that floating point numbers are not the same as real numbers, but this is a case where it matters. Recalling the standard names for the parts of a number written in scientific notation,

$$\overset{\text{sign}}{\overset{\frown}{+}}\underbrace{1.3806549}_{\text{significand}}\times 10^{\overbrace{-23}^{\text{exponent}}},$$

a floating point number allocates 1 bit for the sign and a set number of bits for the significand[107] and for the exponent. For the `Float64` type that we use so much, the significand has 52 bits and the exponent has 11 bits. Among other consequences, this means that in floating point arithmetic there are plenty of small numbers δ for which $1 + \delta = 1$: there is simply no way for a `Float64` to tell the two results apart. The largest such δ is called the floating-point unit, u, and for a 64-bit float $u = 2^{-52} \approx 2.2 \times 10^{-16}$.

The finite precision of computer arithmetic has important consequences for how accurate finite difference are. When h is small we are subtracting two nearly equal floating point numbers from each other, loosing many of our significant digits. This lose of accuracy is then magnified when we divide by the small value of h, leading to a round-off error of

$$E_{\text{round-off}} \approx 2\frac{u|f(x)|}{h}. \tag{17.1}$$

This competes with the error we make in using a finite h in the first place: Taylor expanding $f(x + h)$ about x, we get a truncation error of

$$E_{\text{truncation}} = \frac{h}{2}|f''(x)| + \mathcal{O}(h^2). \tag{17.2}$$

Thus, the optimal step size to use requires minimizing this sum of errors, i.e.,

$$h_{\text{opt}} \approx 2\sqrt{\frac{u|f(x)|}{|f''(x)|}}. \tag{17.3}$$

This is the origin of the common advice: in the absence of any particular information about the magnitude of the function and its second derivative, assume both are of order one and pick a step size close to $\sqrt{u}$. To demonstrate explicitly how the finite difference approach's error varies with h, I somewhat randomly[108] down this function:

```
function goofy_function(x)
    return (sqrt(x) + sin(x)*cos(x))/(log(1+tanh(x)))
end
```

The left panel of Fig. 17.1 shows that function and its derivative, and the right panel shows how accurately finite difference are able to approximate the true derivative as a function of h at two different values of x.

[107] Some of us grew up – like John Wallis in 1693 [82] and Euler in 1748 [83] – calling it the "mantissa." That derives from the Latin *mantisa*, meaning a worthless addition or makeweight; sometimes the more accurate words are less fun.

[108] A choice which I instantly regretted, as it involved a few more applications of the chain rule than I actually wanted to do by hand.

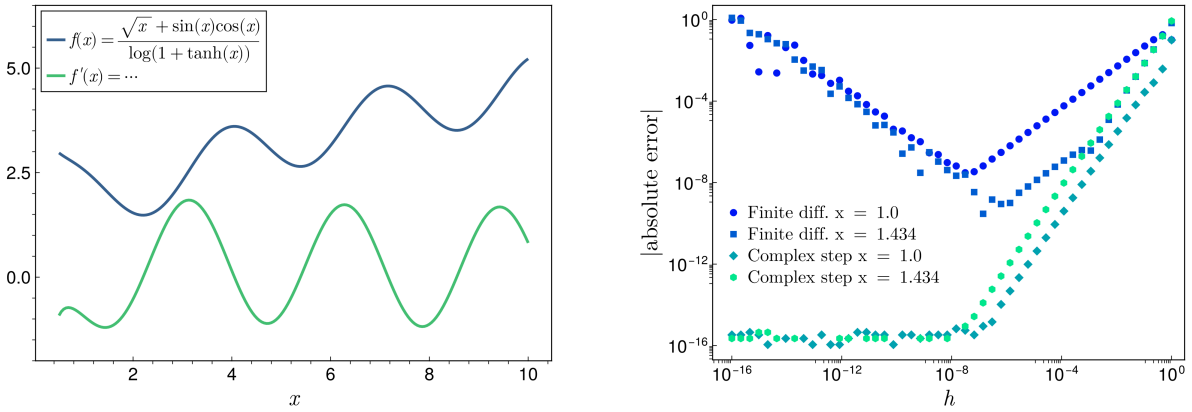

Figure 17.1: (Left) A plot of a goofy function, and its analytically computed derivative, as a function of x. (Right) The absolute error of the forward difference and complex step approximations to the derivative, as a function of the step size h.

Calculating finite differences is at least easy to do, and is sometimes the appropriate solution, but this leaves us in an unsatisfying situation. Even leaving aside the question of repeated function evaluations, choosing the optimal step size requires knowing the derivative of the function. But clearly if we knew the derivative we wouldn't have to resort to finite differences in the first place.

Complex step method

The origin of the problem with the finite difference method is known as "catastrophic cancellation," in which we the representative power of floating point numbers is destroyed by subtracting off two nearly equal numbers from each other. Could this somehow be avoided, perhaps by doing the calculation of the derivative in a way which is separate from the calculation of the function itself? One clever idea is to make use of the theory of complex variables – even if the functions we care about are themselves real. The "complex step method" [84, 85] begins by noting that the expansion of an analytic function $f(z)$ about a real point x is

$$f(x+ih) = f(x) + ihf'(x) - \frac{h^2}{2}f''(x) - i\frac{h^3}{6}f'''(x) + \cdots . \tag{17.4}$$

Notice that, in the argument of the function, it does not matter how small h is compared to x: a `Complex` type will store the real and imaginary components of a complex number as separate floating point numbers. The value x lives in the purely real part of the complex number, and the value h lives in the purely imaginary part. Also notice that, for $h \ll 1$, we can immediately read off the derivative:

$$f'(x) = \frac{Im\,(f(x+ih))}{h}. \tag{17.5}$$

Thus – provided that we implement our function f so that it works on complex numbers – we can immediately get the derivative of a function without worrying about the interplay between truncation and round-off errors. Julia has fantastic support for complex numbers – the global constant `im` represents the imaginary unit i – and the automatic conversion of promotion system it has for `Number` types does all of the work for us. We simply write:

```
"""
    complex_step_method(f, x::Real)

Calculates both the function value `f(x)` and its derivative `f'(x)`
using
the complex step method.
"""
function complex_step_method(f, x::T) where {T <: Real}
    h = floatmin(T)
    z = x + h * im
    return (real(f(z)),imag(f(z))/h)
end
```

Code block 17.1: The complex step method can compute compute the value and derivative of an analytic function at a real value with a single function evaluation.

```
function complex_step(f, x::Real, h::Real)
    return imag(f(x+h*im))/h
end
```

Figure 17.1 compares this approach with the earlier finite difference approach on the same goofy function, and we see that we are easily able to get to machine precision in our approximation of the derivative. Even better: with a *single* evaluation of the function on a complex argument we can get both the value of the function *and* its derivative! The imaginary part, as we saw above, is proportional to the derivative, and the real part is (to $\mathcal{O}(h^2)$) the value of the function itself. A straightforward implementation of this is shown in code block 17.1, where we use our freedom to choose an absurdly small step size to simultaneously obtain the function's value and its machine-precision accurate derivative.

While remarkably convenient, the complex step method runs into practical problems pretty quickly. For instance, the complex step method relied on a Taylor expansion of a complex function about some real value, but what if the function isn't analytic? This is not just an abstract concern; for instance, "rectified linear units" [86, 87] are not analytic, and they are one of the most common functions used as a building block of artificial neural networks. Also, what if the function we are working without doesn't naturally handle complex numbers? Perhaps in our function we had a conditional branch like `if x > 0`; this is perfectly sensible for the real-valued `x` we were probably imagining, but meaningless if we need to treat the variable as complex.

Analyticity matters

The complex step method will definitely return *something* if you apply it to non-analytic functions, it's just that what it returns no longer needs to be related to the derivative of the function! What happens, e.g., for $f(x) = \text{abs}(x)$?

17.3 Forward mode automatic differentiation

Complex numbers aren't going to do the trick, but is there another way of coming up with a description of a function so that (a) derivative information is carried along with the variable not as a small-number Taylor approximation but as a fundamental quantity and (b) the issue of catastrophic cancellation is avoided? One approach, which we'll implement in this section, is called *forward mode* automatic differentiation (AD) using dual numbers [77]. The term comes from work[109] by mathematician Eduard Study [89], and the *idea* originates in "A preliminary sketch of bi-quaternions" [90]. There Clifford explored a kind of algebra involving pairs of quaternions, one of which is multiplied by a special "dual" unit that we're about to describe; these ideas were first implemented as a computational technique by Wengert in the mid 1960s [91].

17.3.1 An Algebra for Derivatives

Just as the complex numbers extend the real numbers by defining the imaginary unit i with the property $i^2 = -1$, dual numbers extend the reals by defining the dual unit[110] o with the property $o^2 = 0$. Arithmetic with the dual numbers follows directly from this property (where we define division via the "dual conjugate" just as we do with the complex conjugate when dividing by a complex number).

$$\begin{aligned} \text{addition:} \quad & (a + bo) + (c + do) = (a + c) + (b + d)o \\ \text{multiplication:} \quad & (a + bo)(c + do) = ac + (ad + bc)o + bdo^2 \\ & \qquad = ac + (ad + bc)o \\ \text{division:} \quad & \frac{a + bo}{c + do} = \frac{a + bo}{c + do}\frac{c - do}{c - do} = \frac{a}{c} + \frac{bc - ad}{c^2}o \end{aligned} \tag{17.6}$$

Stare, in particular, at the result for multiplication and division. If we interpret the real part of a dual number as a value and the dual part as a derivative, it sure looks like the rule for multiplication of dual numbers has the same structure as the *product rule* of calculus. Perhaps even more surprisingly, the rule for division has the same structure as the quotient rule!

Indeed, the "nilpotent" property of duals, $o^2 = 0$, is not a coincidence; we can view the entire system as being deliberately constructed to reproduce the arithmetic of first-order Taylor series. If we formally substitute $x \to x + h$ and then replace h with our dual unit o, the Taylor expansion becomes

$$f(x + o) = f(x) + f'(x)o + \frac{f''(x)}{2!}o^2 + \cdots = f(x) + f'(x)o.$$

The algebra perfectly truncates the series for us.

[109] Of whose book a contemporary review said, "These pages contain a kernel of sterling value concealed within an immense husk of almost impenetrable material which would prove trying to even the strongest intellectual teeth..." [88].

[110] I'm using o in place of the ε you'll more commonly see in the literature because the entire dual number formulation of forward mode AD is so reminiscent of Newton's fluents and fluxions [92].

Thus, if we want to know the value of a function and its derivative at the real number x, we can seed our computations by feeding in the dual number $x + 1o$ (where the "1" represents the derivative of x with respect to itself, $dx/dx = 1$). We can proceed to evaluate functions in their natural order forward, and the arithmetic of dual numbers automatically takes care of the propagation of derivatives via the chain rule for us. That is, for $h(x) = f\big(g(x)\big)$, we first evaluate $g(x + o) = g(x) + g'(x)o$. Applying f:

$$f\big(g(x) + g'(x)o\big) = f\big(g(x)\big) + f'\big(g(x)\big)g'(x)o.$$

That is: the real part is precisely the composition of the functions and the dual part is precisely the application of the chain rule for evaluating the derivatives!

17.3.2 Implementing a `Dual` Type in Julia

Turning this idea into code is as simple as defining a new kind of data structure and defining the rules that we outlined above. Julia makes this extremely straightforward. First, we learned back in Section 2.1 that Julia has a rich system for automatically converting and promoting numerical types when they are mixed together in an expression – for instance, when we type `1 + 2.0` Julia sees that an `Int64` and a `Float64` are being combined, and it knows to automatically convert the integer type to a floating point type before performing the arithmetic operation. We can leverage this system by declaring our `Dual` structure to be a numerical subtype, and then telling Julia what to do when we mix together a `Dual` and any other real number. This is shown in code block 17.2.

```
struct Dual <: Number
    fluent::Float64
    fluxion::Float64
end
# Promote Reals to Duals when they are mixed together
Base.promote_rule(::Type{Dual}, ::Type{<:Real}) = Dual
# Treat Reals as a constant, setting them to a (val,0.) Dual
Base.convert(::Type{Dual}, x::Real) = Dual(Float64(x),0.0)
```

Code block 17.2: A simple implementation of a dual number.

Fluents and fluxions

For out implementation of the `Dual` type I've named the fields "fluent" and "fluxion," serving two purposes. The first is, again, in a nod to the similary between our algebra for differentiation and Newton's original methods. The second is that choosing goofy names (instead of a more sensible pair like "value" and "derivative," or something more technical like "primal" and "tangent") is a reminder: if we wanted to actually *use* AD in our research we should use the much more standard and production-grade autodifferentiation packages that have already been written.

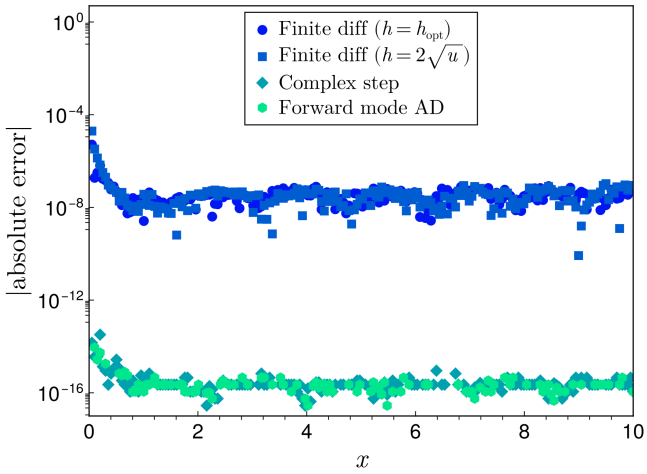

Figure 17.2: Absolute error in computing numerically computing the derivative of $f(x) = (\sqrt{x} + \sin(x)\cos(x))/(\log(1 + \tanh(x)))$ with different techniques: forward finite differences using either the optimal step size of Eq. (17.3) or a heuristic reasonable guess, the complex step method with $h = 10^{-100}$, and with forward mode AD.

Having defined this `struct`, we just need to teach Julia how arithmetic (and any other mathematical functions we might care to use) works with this type. That might look like this:

```
#Overload binary operations, for example:
Base.:+(x::Dual,y::Dual) = Dual(x.fluent+y.fluent,x.fluxion+y.fluxion)
Base.:*(x::Dual,y::Dual) = Dual(x.fluent*y.fluent,
                    x.fluent*y.fluxion+x.fluxion*y.fluent)
Base.:/(x::Dual,y::Dual) = Dual(x.fluent/y.fluent,
                    (y.fluent*x.fluxion-x.fluent*y.fluxion)/y.fluent^2)
#Extend unary functions, for example:
Base.log(x::Dual) = Dual(log(x.fluent),x.fluxion/x.fluent)
```

That's it – we're done! Julia's internal workings – multiple dispatch, conversion and promotion, etc. – take care of everything else for us. Figure 17.2 compares the application of this new way of evaluating derivatives with the finite difference and complex step method we already saw on the goofy function from before. We see that we get machine-precision accuracy without needing to worry about finite step sizes, and while this approach matches the behavior of the complex step method on this analytic function we can easily extend it to cover all of the cases where the limitations of complex variables impeded our progress.

17.3.3 Differentiating through algorithms

We should take a step back and think about what we have done. We implemented a dual number and demonstrated how it works when we apply it to a "goofy" function built out of various mathematical operators and elementary functions. But on a computer *all calculations are expressible as a sequence of such operators and functions!* We should expand our imagination, and think of automatic differentiation not just as unlocking a better version of finite differences

of specific functions, but as something we can apply to *arbitrary chunks of code*. In the context of the LLM example we used as motivation, perhaps we could evaluate the output of the entire LLM *and* its gradient at the same time – all without having to worry about the explosion of expressions a symbolic approach would generate.

```
function bakhshali_root(x::Number; root::Int = 2, iterations::Int = 6,
guess::Float64 = -1.0)
    x_n = (x + 1.0) / 2.0 # a reasonable initial guess
    if guess != -1.0
        x_n = guess
    end

    r = abs(root)
    for _ in 1:iterations
        a_n = (x - x_n^r) / (r * x_n^(r-1))
        x_n = x_n + a_n - (a_n * a_n) / (2.0 * (x_n + a_n))
    end

    if root > 0
        return x_n
    else
        return 1.0/x_n
    end
end
```

Code block 17.3: The generalized Bakhshali algorithm for taking integer roots. For clarity, I have stripped out all of the normal safety checks (Are we taking an even root of a negative number? Are we about to divide by zero?).

As a simple but concrete example, consider the algorithm in code block 17.3. It encodes the "Bakhshali square root" algorithm[111], which approximates square roots via an iterative procedure [94], and here it's been modestly generalized to cover any integer root. Notably, it has many of the common features of the code we often write: for loops, conditional statements, and so on. And yet, we can simply pass one of our dual numbers in and see both the approximate value of the root and the derivative of the corresponding function at that point:

```
julia> S = Dual(2.5,1.0);
julia> bakhshali_root(S; root = -5)
Dual(0.8325532074018731, -0.06660425659214984)
```

Both numbers are correct to within floating point precision.

[111]The birch bark leaves of "Bakhshali manuscript" where this algorithm appears date to sometime between the 9th and 11th century, and were randomly unearthed in an abandoned building in 1881 [93].

17.3.4 The direction of differentiation

The fact that the derivative information is "pushed forward" through the computation from the input to the output gives this form of AD its name. This method is remarkably efficient for functions with few inputs and many outputs (which we might denote as $f : \mathbb{R}^n \to \mathbb{R}^m$ for $n \ll m$). With a single function evaluation we can determine how all m outputs change with respect to any one input by promoting that input to a Dual whose "fluxion" is set to one. To compute the full gradient for a function with $n > 1$, we repeat this forward pass through the function once for each input variable[112].

Performance of forward mode AD

Define some reasonably interesting function $f(x)$ which combines a bunch of elementary operations and, perhaps, some trigonometric or other special functions. Analytically work out the derivative, and then use `BenchmarkTools.jl` (or other benchmarking approach) to compare the computational cost of forward mode AD with your implementation of the analytical derivative. Is one faster than the other?

17.4 Reverse mode AD

That scaling brings us back to our motivating problem. The loss-function of a large language model is basically the worst-case scenario for forward mode AD: while training we're trying to minimize a function that goes from an enormous domain (the billions of parameters) to a one-dimensional range (the scalar loss function) – $f : \mathbb{R}^{\sim 10^9} \to \mathbb{R}$. Calculating the full gradient in order to train the model would require us to run the entire – computationally expensive! – evaluation of the model a billion times. This is not merely impractical; it's impossible! Clearly for this case we need a different approach – an approach that propagates derivative information in the opposite direction so that it will work well when $m \ll n$.

That different approach – reverse mode AD – is the engine that powers all of today's large machine learning models; it was described in work from the 1970s [96] and given a published, algorithmic implementation shortly thereafter [97]. There are two main strategies for implementing reverse mode. One strategy uses is *source-to-source transformation* techniques, which read the source code of a function and programatically generate new, highly-optimized source code that will take the gradient of that function. The second, which we will focus on here, is based on recording a computational graph of a function and traversing it cleverly. Much like we did when implementing forward mode via dual numbers, we will define a new number type that records a trace of all operations involving that type onto a data structure we'll call a "tape[113]." We'll then walk backward along that recording, propagating derivative information

[112] One can extend the approach above to "hyper-dual" numbers in order to compute higher order derivatives or multiple components of the gradient with each function evaluation [95], but the general principle that this approach is best for $n \ll m$ remains.

[113] "Tape" in the sense of a narrow strip of something. From Ref. [98]: *We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1, q_2, \ldots, q_R$ which will be called "m-configurations". The machine is supplied with a "tape" (the analog of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol".*

from the output back to all of the inputs.

17.4.1 Reversing the direction of differentiation

To think about how such a method might work, let's take a moment to work through a toy example. Consider the simple, contrived function

```
julia> f(x1,x2) = x1*x2 + cos(x1)
```

A quick look with the `@code_lowered` macro tells us that we can think of this function as a computational graph where the nodes are the intermediate variables (v_i) in the calculation and the edges are various elementary operations. In this case, for $y = f(x_1, x_2)$ we have

1. $v_1 = x_1 x_2$
2. $v_2 = \cos(x_1)$
3. $y = v_1 + v_2$

At the risk of overexplaining, the chain rule is the fundamental mechanism by which we can find the derivative of the final output, y, with respect to one of the initial inputs x_i. For instance, x_1 influences y through two paths of this graph, so we sum the relevant contributions:

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial v_1}\frac{\partial v_1}{\partial x_1} + \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial x_1}$$

Forward mode corresponds to the intuitive, left-to-right (or, from the perspective of the function, from the inner-most expressions to the outermost) evaluation of this function, pushing the derivatives of all of the intermediate variables (like $\partial v_1/\partial x_1$) forward through the graph. By pushing everything forward like this we (a) immediately get the derivative of *all of the output variables* but (b) have to repeat for each *input* variable.

This is basically the key insight: to get the gradient of one output with respect to many inputs, what we need are quantities like $\partial y/\partial v_i$ that quantify how sensitive the final output is to one of the intermediate variables. Let's define the *adjoint* of a variable, denoted with an overbar, as the partial derivative of the final output with respect to that variable:

$$\bar{u} \equiv \frac{\partial y}{\partial u}.$$

Our goal is to compute the adjoints of the inputs, $\bar{x}_i$; we'll again rely on the chain rule. For some variable u that is used to compute a set of subsequent variables $\{v_j\}$, the chain rule says

$$\bar{u} = \frac{\partial y}{\partial u} = \sum_{j \in \text{children}(u)}^{n} \frac{\partial y}{\partial v_j}\frac{\partial v_j}{\partial u} = \sum_{j \in \text{children}(u)}^{n} \bar{v}_j \frac{\partial v_j}{\partial u}.$$

In forward mode AD we started with a known initial derivative to seed one of our dual numbers with $\partial x/\partial x = 1$. In reverse mode AD we start with the known *adjoint* of the output, $\bar{y} = 1$, and apply the chain rule backwards through the computational graph until we

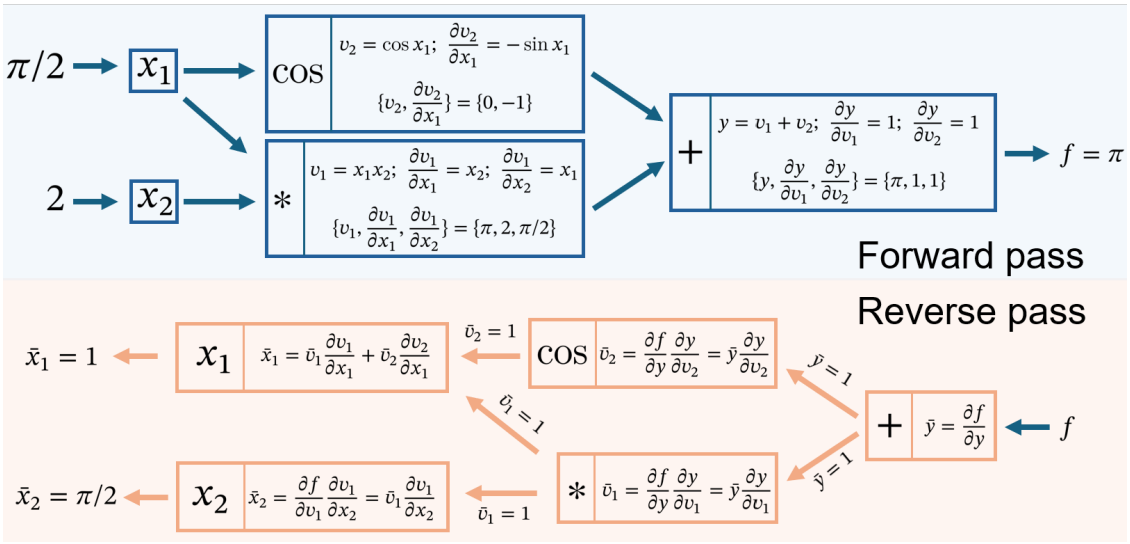

Figure 17.3: An illustration of a simple forward and reverse pass. The forward pass computes the value of the function and also stores information that will be needed in the reverse pass; in the reverse pass the computational graph is traversed from the output to the input in order to compute the derivative of the output with respect to many inputs simultaneously.

reach all of the inputs. Being able to do this backward trace through the calculation while also maintaining knowledge of the local derivatives of the elementary operations (the $\partial v_j/\partial u$ terms) requires a bit more work than forward mode, but it is not so bad. As illustrated in the context of a particularly simple function in Fig. 17.3, we proceed in two phases: During the **forward pass** we process the function from inputs to output, just as in the normal evaluation of the function. Along the way we store the values of all intermediate variables, along with information about the computational graph that is being constructed. During the **reverse pass** we start with $\bar{y} = 1$ and propagate the adjoints backward through the graph using the rule above. This operation often depends on the values of the local derivatives that we computed in the forward pass, and we accumulate the partial derivatives for each variable until we have the adjoints for all the inputs.

17.4.2 Implementing a tape-based reverse mode in Julia

To begin, we define the "tape" on which we'll record the nodes and derivative information we discussed above. The `Tape` will just be a vector of `Node` objects, and each node will hold a tuple of local derivatives and a tuple of indices that point to the potential parents of that node. For type stability and for convenience we'll say that *every* node has two parents – this is just one example of how we're writing less-than-optimally generic code here, but I hope that it makes the implementation easier to understand. For the purposes of making the following code concise I'm also going to use a global variable for the tape; this requires writing less code, but is yet another "code smell[114]." Code block 17.4 defines all of these basic objects.

[114]And, in this, case, a code-smell that implies concrete limitations of what we will be able to do with this implementation.

```
struct Node
    local_derivatives::NTuple{2, Float64}
    parents::NTuple{2, Int}
end

struct Tape
    nodes::Vector{Node}
end

const TAPE = Tape(Node[])
reset_tape!() = empty!(TAPE.nodes)
push_to_tape!(n::Node) = push!(TAPE.nodes,n)
```

Code block 17.4: Defining nodes and a global tape for reverse mode AD.

We next define in code block 17.5 a `TrackedVariable` structure which will hold a value, and will also keep track of an index (corresponding to where it is first defined on the global tape). Just as in our implementation of dual numbers, we declare these to be a subtype of `Number` so that we can access Julia's powerful automatic promotion and conversion functions. We also define a constructor for these tracked variables that places them on the end of the tape and assigns them their new index automatically, and define conversion and promotion rules so that (e.g.) multiplying a tracked variable by a constant behaves correctly.

```
struct TrackedVariable <: Number
    index::Int
    value::Float64
end
# Create a new leaf node on the global tape.
function TrackedVariable(x::Real)
    index = length(TAPE.nodes) + 1
    push_to_tape!(Node((0.0, 0.0), (index, index)))
    TrackedVariable(index, convert(Float64, x))
end

Base.convert(::Type{TrackedVariable}, x::Real) = TrackedVariable(x)
Base.promote_rule(::Type{TrackedVariable}, ::Type{<:Real}) =
TrackedVariable
```

Code block 17.5: A tracked variable structure that automatically gets added to the global tape.

In order to make this whole machinery work, we need to be able to transcribe what's happening with our tracked variables onto the tape. We'll do this by defining the way functions operate on `TrackedVariables`. To record a binary operation, we follow the pattern given in code block 17.6, in which the parents and local derivatives are correctly assigned and a new tracked variable is then added to the global tape.

```
function push_binary_operation!(parent_idxs::NTuple{2, Int},
                                local_derivatives::NTuple{2, Float64})
    index = length(TAPE.nodes) + 1
    push_to_tape!(Node(local_derivatives, parent_idxs))
    return index
end

function Base.:*(a::TrackedVariable, b::TrackedVariable)
    local_derivatives = (b.value, a.value)
    parents = (a.index, b.index)
    new_index = push_binary_operation!(parents, local_derivatives)
    return TrackedVariable(new_index, a.value * b.value)
end
```

Code block 17.6: Defining how binary operations transcribe tracked variables to the tape.

Binary operations fit very naturally into the structure of the `Node` we've defined, where each node has two parents. But obviously we *do* want to be able to handle unary operations, too, so we will adopt a convention in which the second "parent index" of a unary operation points back to the node itself. This is done in code block 17.7, implementing the `push_unary_operation!` function and showing its use in a specific case.

```
function push_unary_operation!(parent_idx::Int, local_deriv::Float64)
    index = length(TAPE.nodes) + 1
    push_to_tape!(Node((local_deriv, 0.0), (parent_idx, index)))
    return index
end

function Base.sin(x::TrackedVariable)
    local_derivative = cos(x.value)
    new_index = push_unary_operation!(x.index, local_derivative)
    return TrackedVariable(new_index, sin(x.value))
end
```

Code block 17.7: Defining how unary operations transcribe tracked variables to the tape.

At this point we have all of our recording technology set up: if we define methods of all of the other elementary binary and unary operations that might constitute the code we want to be able to take the derivative of, a record of the intermediate values and all of the local derivatives will be stored on the tape. Each node of the tape will, furthermore, contain information about the indices of that node's parent(s) – this is precisely the information we need to be able to accumulate the adjoints as described above. All that is needed to a function that actually walks through the tape backwards, seeding the adjoint of whatever output variable we are interested in knowing the gradient of with respect to all of the input variables and then propagating all of the adjoint and local derivative information successively to parents. This reverse traversal is

```
struct Gradient
    derivatives::Vector{Float64}
end
Base.getindex(g::Gradient, v::TrackedVariable) =
g.derivatives[v.index]

function grad(z::TrackedVariable)
    nodes = TAPE.nodes
    derivatives = zeros(Float64, length(nodes))
    derivatives[z.index] = 1.0 #seed the adjoint of the output

    for i in length(nodes):-1:1 # Traverse the tape in reverse
        node = nodes[i]
        adj = derivatives[i]
        # (adjoint of parent) += (adjoint of child)*(local derivative)
        derivatives[node.parents[1]] += adj * node.local_derivatives[1]
        derivatives[node.parents[2]] += adj * node.local_derivatives[2]
    end

    return Gradient(derivatives)
end
```

Code block 17.8: A simple implementation of traversing the global tape

implemented in code block 17.8.

Notice that this function computes the gradient of the tracked variable with respect to not only the input variables but all of the intermediate ones, too. As a convenience, then, in code block 17.8 we also define extend `getindex` so that we have a helper function that lets us extract the derivative information we want in a more idiomatic way.

With all of that in place, here is a silly example of a function that takes multiplies a bunch of numbers together, takes the sine, and then adds that to the sine of the sine of the product. Evaluating the gradient of this ridiculous function with respect to all of the input variables is now as simple as code block 17.9.

Of course, there are *many* reasons why we wouldn't want to use this pedagogical approach in production. For instance, if our functions has N operations, our current implementation involves a computation of $\mathcal{O}(N)$ time (that's good!) and $\mathcal{O}(N)$ space (that's bad – even the simplest function involving the billions of parameters of an LLM would take *terabytes* of memory!). More sophisticated approaches can make use of techniques like checkpointing (in which only some computations are written to the tape and others need to be recomputed during the reverse pass, costing $\mathcal{O}(N \log N)$ time but only $\mathcal{O}(\log N)$ space) or exploit the mathematical structure of (often sparse) matrix multiplications to radically reduce the amount of data the tape needs to store [99].

On the other hand, I still think this is remarkable! In less than 100 lines of code we've implemented all of the structure we need to automatically, algorithmically, take the gradient of

```
function f(x::AbstractArray{<:Number})
    v1 = prod(x)
    v2 = sin(v1)
    return v2 + sin(v2)
end

x=[TrackedVariable(0.5*i) for i in 1:5]
y=f(x)
g=grad(y)
println([g[xi] for xi in x])
```

Code block 17.9: Using our reverse mode AD to find the gradient of a not-especially-helpful function.

a function with respect to many inputs, and all in a way that involves only a small constant multiple of however many operations it took to compute that function in the first place. Although we will make use of this implementation in the coming chapters, it must be emphasized:

Attention!

Our implementation of reverse mode AD in Julia is straightforward (albeit slightly more complex than our implementation of forward mode). It is, however, extremely difficult to get all of the edge case correct and build it in a way which is efficient for a wide range of problems and generic enough to handle the dizzying array of computations we can imagine – for instance, what if we want to apply forward mode differentiation through a reverse-mode calculation? Thus, the implementation here should be treated as a pedagogical one, and *not* what you would turn to in your actual research.

Chapter 18

Neural networks

Under construction

This chapter is still under construction. Key points have been transcribed from the lecture notes, but not the code blocks, figures, or text.

Intro: from Chapter 17 we have the machinery to efficiently, algorithmically find the gradients of functions with respect to inputs. In this chapter we are going to apply this machinery to build up a cornerstone of machine learning: artificial neural networks.

We are going to think of NNs as powerful, general-purpose ways of learning how to approximate general functions $f : \mathbb{R}^n \to \mathbb{R}^m$. These are most useful when we don't know how to write down the correct function, and we are trying to let the data guide us to a reasonable approximation. We'll see how this is useful for tasks ranging from regression to classification to inference, and that we should view machine learning as a (relatively) new tool we should add to our toolkit as we attempt to build a physical understanding of the world around us.

18.1 From a "neuron" to a network

We'll be spending this chapter on neural networks, which have a design very loosely inspired by physical neurons. In the spirit of spherical cow models, in our context an individual "neuron" is going to be modeled by a unit which receives n inputs, $\vec{x} = \{x_1, x_2, \dots, x_n\}$, and produces an output y. The output is determined by first applying an affine transformation (i.e., computing a weighted sum with a bias),

$$z = \sum_i^n w_i x_i + b = \vec{w} \cdot \vec{x} + b,$$

and then applying a (typically nonlinear) "activation function", $y = f(z)$. This is schematically illustrated in Fig. 18.1.

Common activation functions: The sigmoid, $f(x) = (1 + \exp(-x))^{-1}$, which was historically important but is less used now. The hyperbolic tangent, $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, which is mathematically pleasing. The "rectified linear unit (ReLU), $f(x) = \max(0, x)$, which is simple, fast, and variations of which are the modern default.

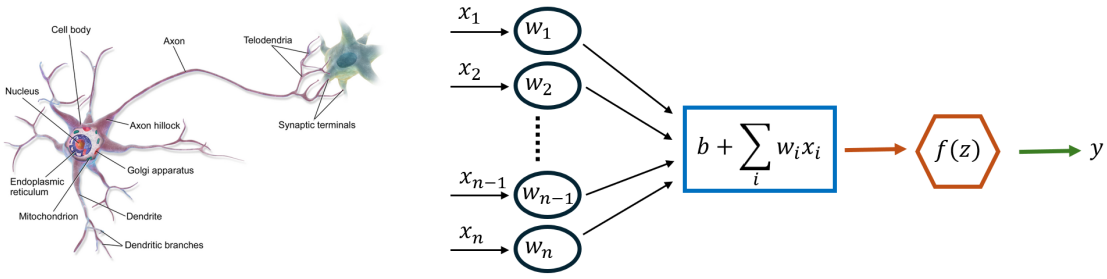

Figure 18.1: (Left) An illustration of the anatomy of a multipolor neuron – image from Bruce Blaus, Blausen Medical Communications, CC Attribution 3.0 unported. (Right) A spherical-cow model of a neuron whose output involves applying an activation function, f, to an affine transformation of its inputs.

18.1.1 The perceptron

Some historical color goes here: Rosenblatt 1957/1958 with 400 pixels and a 512 hidden layer. Binary classifier.

"perceptron" as a term for a single-layer network: the original had inputs directly mapped to a layer of outputs via a step function. Key limitation (minksy / papert): can only solve linearly separable problems. Fails, e.g., for the problem of predicting xor from two inputs.

18.1.2 Multi-layer perceptrons

The solution to the XOR problem: stack / glue layers together. Structure: for $f : \mathbb{R}^n \to \mathbb{R}^m$, start with an input layer of n neurons, which directly take the data $\vec{x}$. Add "hidden layer(s):" one or more layers that perform intermediate calculations. Add an output layer of m neurons, responsible for producing the final result.

Think of this as a composition of activation functions acting on the affine transformations. E.g, for one hidden layer and an output layer, we have

$$\vec{y} = f_{\text{out}}\left(\mathbf{W}_2 \cdot f_{\text{hidden}}\left(\mathbf{W}_1 \cdot \vec{x} + \vec{b}_1\right) + \vec{b}_2\right). \tag{18.1}$$

Universal approximation theorem: An MLP with just one sufficiently wide layer can approximate any continuous function to an arbitrary degree of accuracy – hence, NNs are "general-purpose function approximators" How to find the NN that approximates the function you want??

There are *parameters* (learnable/adjustable weights $\mathbf{W}$ and biases $\vec{b}$) and *hyperparameters* (non-learnable design choices: number and width of layers, activation function to use, etc.).

18.1.3 Training and gradient descent

Goal: for a given set of hyperparameters, find the optimal parameters $\theta = \{\mathbf{W}_1, \vec{b}_1, ...\}$ that make the network's output match the true data.

Quantification: define a loss function $L(\theta)$ that quantifies how "wrong" the network's predictions are.

Idea: perform gradient descent! Treat L as a high-dimensional landscape, in which we want to find the minimum (just like minimizing energy in a many-particle hamiltonian system). The negative gradient, $-\nabla_\theta L$, points in the direction of steepest descent, and – super! – we just developed the machinery to easily find the gradient of functions of many inputs! Update with a "learning rate" η:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_\theta L. \tag{18.2}$$

Here η is yet another hyperparameter.

An interesting caveat is that sometimes we have *a lot* of data: calculating the gradient of the loss function over all of it might be extremely slow. An interesting idea (SGD) is to *estimate* the gradient by using a small, random "mini-batch" of the data at each time step, and assuming that $-\nabla_\theta L \approx -\nabla_\theta L_{\text{batch}}$. Not only is this faster; empirically this "noise" helps escape local minima and train better networks. In SGD an "epoch" is one full pass through the entire training dataset (often many mini-batches).

18.2 Case study: learning a function

Goal: train a small MLP to learn $f(x) = \sin(2\pi x)$ on the interval $[0, 1]$. We'll stitch together a sigmoid or tanh with an identity-activation output layer ($y = z$ – what we typically want for regression problems).

Loss function will be MSE / L2 norm:

$$L(\theta) = \frac{1}{N}\sum_i \left(y_{\text{true},i} - y_{\text{pred},i}\right)^2 = \frac{1}{N}\sum_i \left(\sin(x_i) - \text{model}(x_i, \theta)\right)^2. \tag{18.3}$$

[code samples go here – our tape-based method will be more than sufficient to learn a sin function]

18.3 Case study: classifying data

One of the other major uses of machine learning is for classification tasks – is a given image a dog/cat/bat/hat? is our physical system in this phase or that phase? – in which we try to match not a continuous variable but make a prediction for a discrete label A special case of such tasks are *generative* ones: whereas standard classification tasks can be thought of as trying to assign a label to given data input, we can also try to build a classifier that accepts an input sequence and assigns a label corresponding to the predicted *next* word in that sequence.

18.3.1 MNIST data

Historical color: MNIST as the hello world of ML, and which kick-started a revival. The data came from NIST, scaled and centered, etc.

What is it: a large number of 28-by-28 grayscale pixel maps of the digits from zero to nine. We prep this data by flattening each input into a vector of 784 integers – *losing information*

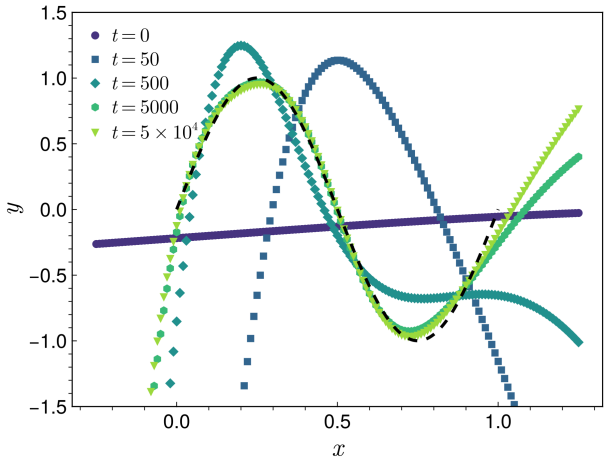

Figure 18.2: The output of a single-hidden-layer neural network trained on 100 input-output pairs of the function $\sin(2\pi x)$ on the domain $x \in (0, 1)$ (black dashed line). Points correspond to a 1000-neuron hidden layer with a sigmoid activation function, trained on mini-batches of size 10 for the indicated number of epochs, using the loss function in Eq. (18.3) and a learning rate of $\eta = 0.01$.

about the spatial correlation of those elements in the process – and keeping track of the known label attached to that vector. Our goal is thus to construct a function $f : \mathbb{R}^{784} \to \mathbb{R}^{10}$, and one way of interpreting the model is to choose the argmax to be the predicted class. A natural scaling would have an output vector of, e.g., $\{1, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$ corresponds to the certainty that the digit was a zero. This is known as a "one-hot" vector encoding of the output. More generally we will construct our function so that the outputs correspond to the predicted probability of the input belonging to each class.

18.3.2 Output and loss functions

Problem: using identity as the final activation function means that the output variables can be an real number, but we would much rather have the output be a vector of probabilities. Also, argmax tells us what class we think the answer is, but nothing about confidence of that prediction. Solution: "soft arg max" → "softmax." This is a function $\sigma : \mathbb{R}^m \to (0, 1)^m$ defined so that each component is:

$$\sigma(\vec{z})_i = \frac{\exp(z_i)}{\sum_j^m \exp(z_j)} \tag{18.4}$$

This is, indeed, positive and normalized so that the components sum to one. Physically, this is like interpreting the components z_i as the "negative energy" of the class, and the softmax function as returning the Boltzmann distribution at $T = 1$. This normalization ensures that components of the resulting vector sum to 1. Gives a probability vector.

Problem: What loss function should we use. MSE measures a geometric distance, but we want to measure the distance between *two probability distributions* (in this case, our certainty from the labels that the ground truth is some particular digit and the models uncertainty about

what the label should be). Solution: *categorical cross-entropy* ("cross-ent"). The definition is

$$L = -\sum_{j}^{m} y_{\text{true},j} \log\left(y_{\text{pred},j}\right), \tag{18.5}$$

which for a true distribution corresponding to a one-hot vector, simplifies to $L = -\log p_{\text{correct}}$. That is: minimizing the loss means maximizing the log-probability of predicting the correct class / category. Seems very reasonable.

Combined with the choice of applying softmax to the final output layer, this also leads to a particularly simple gradient. For

$$L = -\log p_i = -\log\left(\frac{e^{z_i}}{\sum_k e^{z_k}}\right) = -z_i + \log\left(\sum_k e^{z_k}\right), \tag{18.6}$$

we see that taking the derivative with respect to any of the output numbers is just

$$\frac{\partial L}{\partial z_j} = -\delta_{ij} + \frac{e^{z_j}}{\sum_k e^{z_k}} = p_j - \delta_{ij} = \left(y_{\text{pred,j}} - y_{\text{true},j}\right). \tag{18.7}$$

18.3.3 Overfitting, validation, and testing

We are building models with a large number of parameters – in the context of how we usually think about fitting models to data[115] we should be very worried about potentially overfitting our data. Commonly we might expect that as we train we continuously make the loss function smaller, but at some point we are doing this not by learning the true underlying structure of the data but by over-indexing on the particular bumps and wiggles it might have. If we applied such an overfit model to novel data, we would expect it to do *worse* than a model which had a nominally larger loss on the data we trained it on.

The traditional solution is to split the data we have into training, validation, and test sets. The *training set* is the data we show to the model while minimizing the loss function with our optimization protocol. We can think of this as the "homework" we give our model to learn from—repetitive practice to understand the material. The *validation set* is *not* given to the model to compute gradients of the loss function, but we periodically use it to know when to stop training our models – when the loss is decreasing but the performance on the validation set gets worse, we should stop. We also use it to help tune our hyperparameters We can think of this as a "practice quiz": it helps us adjust our study strategy and tells us when we are ready, but it doesn't count toward the final grade. The *test set* is the "final exam." This data is strictly hidden from the model (and the scientist!) until the very end. We use it only once to get an unbiased estimate of how well the model generalizes to completely new data.

Regularization as bridge between under and over fitting. Add regularization term, e.g., $\lambda \sum_i |\theta_i|$ to promote sparsity (L1) or $\lambda \sum_i \theta_i^2$ to force many parameters to be small but non-zero (L2). L1 is like automated feature selection, forcing irrelevant terms to be zero; L2 is good for highly correlated input variables, or when many things contribute a bit.

[115] *In desperation I asked Fermi whether he was not impressed by the agreement between our calculated numbers and his measured numbers. He replied, "How many arbitrary parameters did you use for your calculations?" I thought for a moment about our cut-off procedures and said, "Four." He said, "I remember my friend Johnny von Neumann used to say, with four parameters I can fit an elephant, and with five I can make him wiggle his trunk."* [100]

18.4 Engineering

Machine learning is a vast discipline, and multiple courses could be devoted to exploring variations of just what we have explored so far. How to set hyperparameters. NNs with other structures (CNN, RNN), the use of different activation functions, the topic of non-convex optimization How to optimize and parallelize the calculations – our examples, realistically, can be run in *seconds* rather than minutes.

Chapter 19

Applications: force inference and phase classification

Under construction

This chapter is still under construction.

19.1 Classification

19.1.1 Identifying phase transitions

19.2 Inference

19.2.1 Inferring force laws

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[2] Cristopher Moore and Stephan Mertens. *The nature of computation*. Oxford University Press, 2011.

[3] Werner Krauth. *Statistical mechanics: algorithms and computations*, volume 13. OUP Oxford, 2006.

[4] Daan Frenkel. Simulations: The dark side. *The European Physical Journal Plus*, 128:1–21, 2013.

[5] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*. Elsevier, 2023.

[6] Alex Gezerlis. *Numerical methods in physics with Python*, volume 1. Cambridge University Press Cambridge, UK, 2023.

[7] Kyle Novak. *Numerical Methods for Scientific Computing: The Definitive Manual for Math Geeks*. Equal Share Press, 2022.

[8] William Jones. *Synopsis Palmariorum Matheseos: Or, a New Introduction to the Mathematics*. J. Matthews for Jeff. Wale at the Angel in St. Paul's Church-Yard, 1706.

[9] William Oughtred. *Clavis Mathematicae denuo limita, sive potius fabricata*. Lichfield, 1631.

[10] Florian Cajori. *A history of mathematical notations*, volume 1. Courier Corporation, 1993.

[11] Tom Kwong. *Hands-On Design Patterns and Best Practices with Julia: Proven solutions to common problems in software design for Julia 1. x*. Packt Publishing Ltd, 2020.

[12] Lee Phillips. *Practical Julia: A Hands-on Introduction for Scientific Minds*. No Starch Press, 2023.

[13] Berni J Alder and Thomas Everett Wainwright. Studies in molecular dynamics. i. general method. *The Journal of Chemical Physics*, 31(2):459–466, 1959.

[14] Gregorii Aleksandrovich Galperin. Playing pool with π (the number π from a billiard point of view). *Regular and chaotic dynamics*, 8(4):375–394, 2003.

[15] F Chiappetta, C Meringolo, P Riccardi, R Tucci, A Bruzzese, and G Prete. Boyle, huygens and the 'anomalous suspension'of water. *Physics Education*, 59(4):045026, 2024.

[16] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[17] Greg Wilson, Dhavide A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. Best practices for scientific computing. *PLoS biology*, 12(1):e1001745, 2014.

[18] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K Teal. Good enough practices in scientific computing. *PLoS computational biology*, 13(6):e1005510, 2017.

[19] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[20] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[21] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

[22] Tony Freeth, Yanis Bitsakis, Xenophon Moussas, John H Seiradakis, Agamemnon Tselikas, Helen Mangou, Mary Zafeiropoulou, Roger Hadland, David Bate, Andrew Ramsey, et al. Decoding the ancient greek astronomical calculator known as the antikythera mechanism. *Nature*, 444(7119):587–591, 2006.

[23] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press, 2009.

[24] Ernst Hairer, Christian Lubich, and Gerhard Wanner. Structure-preserving algorithms for ordinary differential equations. *Geometric numerical integration*, 31, 2006.

[25] Leonhard Euler. *Institutiones calculi integralis*, volume 1. Impensis Academiae Imperialis Scientiarum, 1768.

[26] Carl Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895.

[27] Wilhelm Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901.

[28] John C Butcher. Implicit runge-kutta processes. *Mathematics of computation*, 18(85):50–64, 1964.

[29] Ch Tsitouras. Runge–kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Computers & mathematics with applications*, 62(2):770–775, 2011.

[30] Hermann Weyl. *The classical groups: their invariants and representations*, volume 1. Princeton university press, 1939.

[31] Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[32] Carl Størmer. Sur les trajectoires des corpuscules électriques dans l'espace sous l'action du magnétisme terrestre. *Archives des Sciences Physiques et Naturelles*, 24:5–18, 113–158, 221–247, 317–364, 1907. Published in four parts.

[33] Robert I McLachlan and G Reinout W Quispel. Splitting methods. *Acta Numerica*, 11:341–434, 2002.

[34] Isaac Newton. *Philosophiæ Naturalis Principia Mathematica*. Jussu Societatis Regiæ ac Typis Josephi Streater, Londini, 1687.

[35] Eugene Borisovich Dynkin. Calculation of the coefficients in the campbell-hausdorff formula. In *Dokl. Akad. Nauk. SSSR (NS)*, volume 57, pages 323–326, 1947.

[36] Mark E Tuckerman. *Statistical mechanics: theory and molecular simulation*. Oxford university press, 2023.

[37] William Camden. *Remaines of a greater worke, concerning Britaine, the inhabitants thereof, their languages, names, surnames, empreses, wise speeches, poësies, and epitaphes*. Printed by G. Eld for Simon Waterson, London, 1605.

[38] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[39] John Edward Jones. On the determination of molecular fields.—ii. from the equation of state of a gas. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 106(738):463–477, 1924.

[40] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. *SIGGRAPH sketches*, 10(1):1, 2007.

[41] Kurt Kremer and Gary S Grest. Dynamics of entangled linear polymer melts: A molecular-dynamics simulation. *The Journal of Chemical Physics*, 92(8):5057–5086, 1990.

[42] Roger W Hockney and James W Eastwood. *Computer simulation using particles*. IOP Publishing Ltd., 1988.

[43] Hans C Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *The Journal of chemical physics*, 72(4):2384–2393, 1980.

[44] Shuichi Nosé. A unified formulation of the constant temperature molecular dynamics methods. *The Journal of chemical physics*, 81(1):511–519, 1984.

[45] William G Hoover. Canonical dynamics: Equilibrium phase-space distributions. *Physical review A*, 31(3):1695, 1985.

[46] Glenn J Martyna, Mark E Tuckerman, Douglas J Tobias, and Michael L Klein. Explicit reversible integrators for extended systems dynamics. *Molecular Physics*, 87(5):1117–1157, 1996.

[47] Owen G Jepps, Gary Ayton, and Denis J Evans. Microscopic expressions for the thermodynamic temperature. *Physical Review E*, 62(4):4757, 2000.

[48] Albert Einstein. Über die von der molekularkinetischen theorie der wärme geforderte bewegung von in ruhenden flüssigkeiten suspendierten teilchen. *Ann. d. Phys.(Leipzig)*, 17:549, 1905.

[49] Stanley J Farlow. *Partial differential equations for scientists and engineers*. Courier Corporation, 1993.

[50] Sandro Salsa. *Partial differential equations in action*. Springer, 2016.

[51] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[52] Leonhard Euler. Principes généraux de l'état d'équilibre des fluides. *Mémoires de l'Académie des Sciences de Berlin*, 11:217–273, 1757. Eneström Index E225.

[53] Leonhard Euler. Principia motus fluidorum. *Novi Commentarii Academiae Scientiarum Imperialis Petropolitanae*, 6:271–311, 1761. Eneström Index E258.

[54] Daniel M Sussman and Daniel A Beller. Fast, scalable, and interactive software for landau-de gennes numerical modeling of nematic topological defects. *Frontiers in Physics*, 7:204, 2019.

[55] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

[56] Maurice G Kendall and B Babington Smith. Randomness and random sampling numbers. *Journal of the royal Statistical Society*, 101(1):147–166, 1938.

[57] WE Thomson. A modified congruence method of generating pseudo-random numbers. *The Computer Journal*, 1(2):83–83, 1958.

[58] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of sciences*, 61(1):25–28, 1968.

[59] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[60] Melissa E O'neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Transactions on Mathematical Software*, 204:1–46, 2014.

[61] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software (TOMS)*, 47(4):1–32, 2021.

[62] George EP Box and Mervin E Muller. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 29(2):610–611, 1958.

[63] A. A. Markov. Issledovanie zamechatel'nogo sluchaya zavisimyh ispytanij. *Izvestiya Akademii Nauk*, 1(3):61–80, 1907. Translated into French as "Recherches sur un cas remarquable d'epreuves dependantes", Acta Mathematica, 33, (1910), 87-104.

[64] A. A. Markov. An example of statistical investigation of the text eugene onegin concerning the connection of samples in chains. *Science in Context*, 19(4):591–600, 2006.

[65] Oskar Perron. Zur theorie der matrices. *Mathematische Annalen*, 64(2):248–263, 1907.

[66] Georg Frobenius, Ferdinand Georg Frobenius, Ferdinand Georg Frobenius, Ferdinand Georg Frobenius, and Germany Mathematician. Über matrizen aus nicht negativen elementen. 1912.

[67] William Shakespeare. *The Complete Works of William Shakespeare*. Project Gutenberg, 1994. Project Gutenberg EBook 100 Accessed October 8, 2025.

[68] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. 1970.

[69] Wilhelm Lenz. Beitrag zum verständnis der magnetischen erscheinungen in festen körpern. *Z. Phys.*, 21:613–615, 1920.

[70] Ernst Ising. Beitrag zur theorie des ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, 1925.

[71] Sigismund Kobe. Ernst ising—physicist and teacher. *Journal of statistical physics*, 88(3):991–995, 1997.

[72] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics letters B*, 195(2):216–222, 1987.

[73] Michael Betancourt. A conceptual introduction to hamiltonian monte carlo. *arXiv preprint arXiv:1701.02434*, 2017.

[74] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, 53:370–418, 1763. Communicated by Richard Price.

[75] Pierre Simon Laplace. Mémoire sur la probabilité de causes par les évenements. *Mémoire de l'académie royale des sciences*, 1774.

[76] Matthew D Hoffman, Andrew Gelman, et al. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014.

[77] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153):1–43, 2018.

[78] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[79] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre GR Day, Clint Richardson, Charles K Fisher, and David J Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics reports*, 810:1–124, 2019.

[80] Zhuoqiang Guo, Denghui Lu, Yujin Yan, Siyu Hu, Rongrong Liu, Guangming Tan, Ninghui Sun, Wanrun Jiang, Lijun Liu, Yixiao Chen, et al. Extending the limit of molecular dynamics with ab initio accuracy to 10 billion atoms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–218, 2022.

[81] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[82] John Wallis. *Opera mathematica*, volume 2. E Theatro Sheldoniano, Oxonii, 1693.

[83] Leonhard Euler. *Introductio in analysin infinitorum*, volume 2. MM Bousquet, 1748.

[84] James N Lyness. Numerical algorithms based on the theory of complex variable. In *Proceedings of the 1967 22nd national conference*, pages 125–133, 1967.

[85] William Squire and George Trapp. Using complex variables to estimate derivatives of real functions. *SIAM review*, 40(1):110–112, 1998.

[86] Alston S Householder. A theory of steady-state activity in nerve-fiber networks: I. definitions and preliminary lemmas. *The bulletin of mathematical biophysics*, 3(2):63–69, 1941.

[87] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[88] RWHT Hudson. Geometrie der dynamen. die zusammensetzung von kräften, und verwandte gegenstände der geometrie. von e. study.(leipzig, teubner, 1903.) pp. 603. m. 21. *The Mathematical Gazette*, 3(44):15–16, 1904.

[89] Eduard Study. *Geometrie der Dynamen. Die Zusammensetzung von Kräften und verwandte Gegenstände der Geometrie*. B. G. Teubner, Leipzig, 1903.

[90] William Kingdon Clifford. Preliminary sketch of biquaternions. *Proceedings of the London Mathematical Society*, 1(1):381–395, 1873.

[91] Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

[92] Isaac Newton. *The Method of Fluxions and Infinite Series; With Its Application to the Geometry of Curve-Lines*. Henry Woodfall, London, 1736. Originally written in Latin as 'De Methodis Serierum et Fluxionum' c. 1671.

[93] George Sarton. The bakhshâlî manuscript, 1929.

[94] Bill Casselman. Reading the bakshali manuscript. *American Mathematical Society*, 2018.

[95] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892*, 2016.

[96] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[97] Bert Speelpenning. *Compiling fast partial derivatives of functions given by algorithms*. University of Illinois at Urbana-Champaign, Urbana-Champaign, 1980.

[98] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[99] Adrian Hill, Guillaume Dalle, and Alexis Montoison. An illustrated guide to automatic sparse differentiation. In *ICLR Blogposts 2025*, 2025.

[100] Freeman Dyson. A meeting with enrico fermi. *Nature*, 427(6972):297, 2004.