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 – Please email any 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.

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 integration, solving ODEs, 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?

That's why this course will begin with a "foundations" module. 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 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. The answer has a few layers.

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 simultaneously a dynamically typed "scripting language" in which simple, expressive code can be written very quickly and with minimal boilerplate – even more so than in Python, often one can almost directly translate 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 underneath that surface answer. 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 – like "noun" or "past perfect tense" or "imperfect aspect" – 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 – and more personal – layer to this decision: I love learning. One of the true joys of academia is the constant opportunity to explore new areas and consume more and more knowledge. My own computational research almost entirely involves writing in C and CUDA/C++, and in the early spring of 2025 I saw a research talk that cited a Julia package. The talk was excellent, the code seemed to be doing some clever things, and I filed that memory away as an intriguing tidbit to come back to someday.

Well, when I first began to structure the notes for this class, I started to get a little worried – much of the course content was material I had thought too much about for too much of my research. Where would the opportunity be for me to learn something alongside you? That, ultimately, was the tie-breaking factor in choosing Julia: I selected a language that I was excited to learn more about myself. My hope is that by learning and navigating some of Julia's intricacies together, we not only master the course material but also model the rewarding process of continuous learning and of navigating new technical landscapes – skills that will serve you well long after this semester ends. I'm excited to be on this learning path with you, and I hope you find that enthusiasm infectious!

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.

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:

```julia
# 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].

[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.

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 time and consideration in order to select two 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).

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] Berni J Alder and Thomas Everett Wainwright. Studies in molecular dynamics. i. general method. *The Journal of Chemical Physics*, 31(2):459–466, 1959.

[12] 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.

[13] 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.

[14] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[15] 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.

[16] Leonhard Euler. *Institutiones calculi integralis*, volume 1. Impensis Academiae Imperialis Scientiarum, 1768.

[17] Carl Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895.

[18] Wilhelm Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901.

[19] John C Butcher. Implicit runge-kutta processes. *Mathematics of computation*, 18(85):50–64, 1964.