Lecture notes for Advanced Computational Physics (436)

Daniel M. Sussman

June 6, 2025

Contents

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

[Not needed right now]

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[1]. 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 –

[1] Unicode symbols for the math and all!

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[2] 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.*

A fascinating aspect of learning is that we often grasp the underlying rules of a system more profoundly when we encounter a different but related one. For example, many people find that they gain a radically deeper understanding of the grammar of their native tongue only after they study a second language. We learn to abstract and identify concepts – like "noun" or "past perfect tense" or "imperfect aspect" – that we had been using for years but that we didn't have a label or category for. By choosing a language that I anticipate most of you haven't encountered extensively before, the aim is to provide that "second language experience," but in the real 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 to voraciously consume as much knowledge as possible. 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. Author (*Source*; brief description of why the source is good) [citation]
2. Author (*Source*; brief description) [citation]
3. Author (*Source*; brief description) [citation]

[2] It pulls at me, too!

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), I'll use the following style of light backgrounds and syntax highlighting.

```
# 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; they will appear like this:

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] Especially those code blocks... when I'm working I usually prefer dark themes, but I thought in the context of these notes having a light theme would be more natural.

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[3].

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) uses JetBrains Mono, with all of the ligatures disabled and scaled in size to match the main text.

Colors – including code syntax highlighting and other visual elements – are based on the "Kanagawa" theme (Tommaso Laurenzi, (c) 2021, MIT License).

[3] 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.

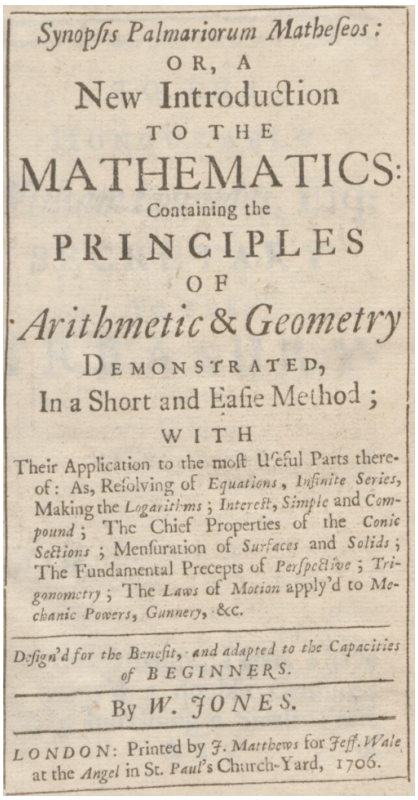
Module 0

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 introduction 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 basic programming concepts. In this course we'll be working with the Julia programming language; I suspect many of you have not used it before[4], and so this module aims to walk you through – 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 1: A page from the first book to use the symbol π with its modern meaning [1]. "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[5]. Earlier the symbol was used by William Oughtred to refer to the circumference of whatever circle was being considered at the time [2]. 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 [3] – a wrinkly in the Pi vs. Tau debate!

The structure of each of the following chapters will be 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.

[4] See the preface for the whole spiel.

[5] 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)!

Appendix A

Setting up Julia

Arguably the most basic method of finding the digits of π is to... ask someone who already knows the answer[6].

A.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.

A.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 the first thing I did when I opened Julia for the first time:

[6]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. [1], page 243).

```
julia> 1+1
2

julia> exit()
```

Powerful stuff.

The REPL also has a powerful "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.

A.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 Appendix E. 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>`".

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 Appendix E.1.

A.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[7]. 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 include this in your startup file:

```
# startup.jl
using Revise
using BenchmarkTools

if isinteractive()
    using OhMyREPL
end
```

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 next two lines just activate `Revise` and `BenchmarkTools` every time we start Julia, and the `if isinteractive() ... end` block is a conditional statement: the code inside this block (here, just `using OhMyREPL`) only executes if Julia is running in an interactive mode, such as when you launch the REPL directly.

A.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 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
```

[7] 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

The "`Printf`" module is part of Julia's standard library, so no separate installation is needed. The "`@printf`" function[8] 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 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 `double` in C++ or the default `float` in Python), 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!

A.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 latter, 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 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 was recently updated so that you can use it to run Julia rather than Python: 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:

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

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

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

```
$ julia ./HelloPi.jl
```

you should see a familiar result. However, *this pattern is discouraged*, because every time you start 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.

Appendix B

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.

B.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.

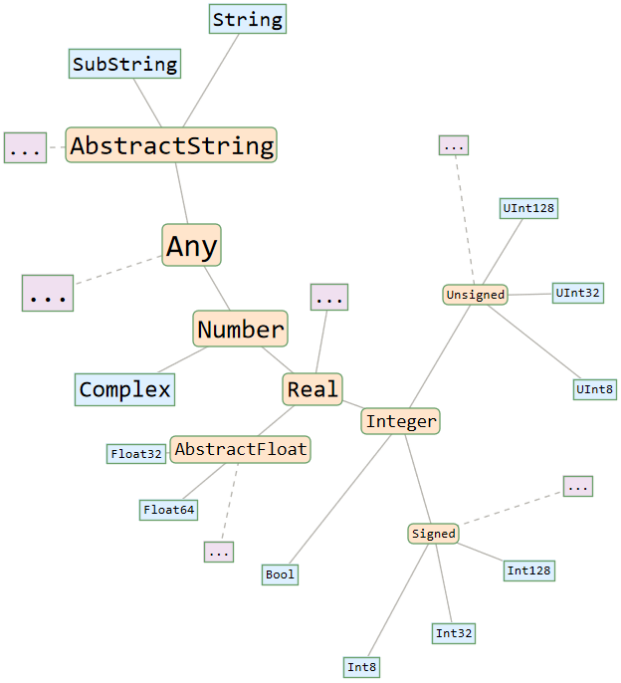


Figure B.1: A small subset of Julia's type tree. Abstract types are in orange, and concrete types are in blue. A potentially large number of nodes are implied by the ellipses in purple.

Julia's type system looks like a massive tree (a tiny portion of which is in Fig. B.1). At the root of this tree is a special type called "`Any`" – by default, values can, indeed, be any type that Julia knows about. This includes the types that are predefined by the language, and also any types that you (or another package

author) define and add to the type tree. Having values be of this special type is the kind of thing that allows this nonsense:

```
julia> x=1
1

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

Julia has a notion of "abstract" vs "concrete" types. Abstract types are merely nodes in the type tree, helping to organize related sets of types, and concrete types are those that the compiler can actually create values for. That means that at any moment a variable will always be some concrete type:

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

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

The above example helps emphasize that *variables* do not have types in Julia, only *values* do. Variables are just names that get associated with values. Here, by the way, we see that we can use a semi-colon in the REPL to suppress output, and the "`typeof()`" function is part of Julia's Core module. You can explore the type tree by using this and the "`subtypes()`" and "`supertypes()`" functions, also in the Core module.

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, which plays a few different roles. One thing we can use it for is *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 Appendices C and E.3, 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).

Type annotations can also be used to *constrain the arguments of functions* – this is a core part of Julia's killer "multiple dispatch" feature, in which specific versions of a function can be called according to the types of arguments passed to it at runtime. We'll explore this powerful feature in greater depth in Appendix E.4, but first let's learn about writing basic functions.

B.2 Functions and control flow

B.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[9], 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.

[9] Perhaps you expected this to be the role played by "plus" operator, but Julia notes that when both addition and multiplication are defined, and if one of them is not commutative, then by convention multiplication is typically the noncommutative one.

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?

B.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 Appendix E.4 – 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[10] formula used to compute the digits of π, which is due to John Machin[11]:

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

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.

B.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:

```
# MachinFunction.jl
function f(x)
    # How does x behave in this context?
```

[10] "Famous"

[11] The same Mr. John Machin we met in Footnote 6

```
    x = 16*atan(1/5) - 4*atan(1/239)
    return x
end
```

In this particular function[12] 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* values (for instance Ints and Floats) and *mutable* values (for instance, Arrays, which we will meet more properly in Appendix C). If a local name refers to a mutable object, you can change the internal state of that object (for instance, 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 an example, 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
```

We clearly see that rebinding the local argument name in the function doesn't affect the caller's variable.

[12] 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 Appendix E.2.

B.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[13] series expansion for the arc tangent,

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

B.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 atanSeries(x,numberOfTerms)
    if numberOfTerms <= 0
        error("numberOfTerms must be positive")
    end
    result = 0.
    for i in 0:numberOfTerms-1
        result += (-1)^i*x^(2*i+1) / (2*i+1)
    end
    return result
end
function f(x)
    result = 16*atanSeries(1/5,x) - 4*atanSeries(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 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 Appendix C.1, but for now we'll just use the above as a way of writing a `for` loop.

[13] 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.

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 - f(10)
8.881784197001252e-16
```

B.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 especially *expressive* of the overall intent[14]. 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 that was the 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, the following code that creates a vector of functions and iterates over them...works:

```
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
```

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

[14]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.

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[15] or if you write raw[16] loops? Nope. But programming, at its heart, is an act of composition: we articulate solutions by weaving together data structures and algorithms to accomplish some goal. When we translate a mathematical concept or a physical system into code, we have many choices to make. You might opt to write your code in an imperative style, with a sequence of explicit operations that change the program's state. You might lean into an object-oriented approach[17] to dividing up the state of your program and what pieces of it are responsible for acting on different components of that state. You 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> atanSeriesTerm(x,i) = (-1)^i*x^(2*i+1) / (2*i+1) ;

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

I really don't believe any of these approaches are inherently "better" than the others. 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.

Programming style?

Which version of `atanSeries` do you find most clearly conveys the mathematical summation here? Why? Would something else be even better? Reflecting on such choices, and why one form might appeal over another in different contexts, is crucial for developing your own programming style.

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

[16] If you've written a lot of verbose C++ code you might find Sean Parent's "C++ Seasoning" an insightful perspective

[17] We'll see that Julia doesn't use classes and inheritance in the same way as C++ or Python, but we'll see in Appendix E how we can use its powerful type system and multiple dispatch to enable robust and flexible OO-like designs.

Appendix C

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.

C.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.

C.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 [18]

[18] Don't worry: I appreciate the irony of having this module appear as "Module 0" in the table of contents.

enough:

Don't forget!

Julia is a 1-indexed language!

That is, you access the element of the tuple above (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[19]: 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
```

C.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:

[19] 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 ArrayInt64, 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 MatrixFloat64:
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.

C.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[20], 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:

[20] 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
```

C.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> indexingStyle = Dict("C++"=>0,"Python"=>0,"Scheme"=>0);

julia> indexingStyle["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> indexingStyle["Julia"]=1;

julia> push!(indexingStyle,"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...

C.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 testWhileLoop()
    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[21], 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 testForLoops()
    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. An iterator is an object that produces a sequence of values one at a time, often on demand, without necessarily storing all of them in memory at once.

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

C.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[22] 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
```

C.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 *comprehension* syntax. The idea is to write something like

```
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.

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

C.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
           funnyName::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, for instance:

```
julia> mutable struct ParticlePosition
           x::Float64
           y::Float64
           z::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 Appendix E.4, 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[23] constructors. These are simply functions (often using the same name as the struct) that return a new instance of the struct in question.

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. We'll further define a function that defines the norm of a PolygonVertex to be its distance from the origin. Finally, in a slight abuse of the intention of the data structure[24], create a new method for the binary subtraction operator.

```
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
```

C.4 Hello, π! (Method 3: Using geometry)

Figure C.1: Archimedes, or perhaps a self-portrait by Jusepe de Ribera; funny how it's hard to tell sometimes.

Instead of relying on modern[25] 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 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

[23] Yes, there are also inner constructors

[24] 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...

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

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 calculatePerimeter(polygon)
    result = zero(polygon[1].x);
    lastPoint = last(polygon)
    for currentPoint in polygon
        distance = currentPoint - lastPoint
        result += norm(distance)
        lastPoint = currentPoint
    end
    return result
end

function archimedes(n)
    angles=range(0,step=2pi/n,length=n)
    pts = [PolygonVertex(θ) for θ in angles]
    return calculatePerimeter(pts)/2
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 Appendix B.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 calculatePerimeter2(polygon)
    #That's a rotate!
    rotatedPolygon = circshift(polygon,1)
    displacements = polygon .- rotatedPolygon
    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 calculatePerimeter3(polygon)
    return mapreduce(i-> norm(polygon[i] - polygon[mod1(i-1,
length(polygon))]), +, 1:length(polygon))
end
```

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

C.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 iterativeArchimedes(doublings)
    numberOfSides::BigInt = 6
    sideLengthSquared::BigFloat = 1.0

    for i in 1:doublings
        numberOfSides *= 2
        sideLengthSquared = 2-2sqrt(1-sideLengthSquared/4)
    end
    return numberOfSides*sqrt(sideLengthSquared)/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). If this was indeed his approach[26], 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!

C.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

[26]The doubt basically being related to the existing booking technology for keeping track of the intermediate calculations while tediously evaluating square roots.

clear with the compiler about what you want to do, but also being clear with your collaborators[27]. 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> computeAcceleration(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` [28]). 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!

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! Speaking of... please don't name your function like `myFunction3`!

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

[27] which includes *your future self*

[28] Something I still struggle with! I personally find it hard to read code with mixed camel and all-lowercase and snake case conventions, and in many other contexts this mixing is discouraged. In the Julia community, however, it is idiomatic, and I should probably go back and re-write a lot of these names

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:

```
"""
    calculatePerimeter3(polygon::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
- `polygon`: 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> calculatePerimeter3(square)
4.0
```
"""
function calculatePerimeter3(polygon)
    return mapreduce(i-> norm(polygon[i] - polygon[mod1(i-1,
length(polygon))]), +, 1:length(polygon))
end
```

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 [29]! 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.

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

Appendix D

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[30], and we'll learn a lot more about these approaches later in this class (see **??**)!

D.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

[30]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:"

```
generatePoint(L) = rand(Float64,2) .*L .-L/2
throwPointsDown(n,L) = [generatePoint(L) 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.

D.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
```

D.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 `throwPointsDown` 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",throwPointsDown(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.

D.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.

D.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[31]: 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[32] 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.

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

[32] 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 Appendix E.1

D.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[33] which will let us determine which points are inside of and outside of the unit circle:

```
isInUnitCircle(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(isInUnitCircle,eachrow(dataRead))
exterior = filter(!isInUnitCircle,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`[34] 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.

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

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

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

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:

```
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)
```

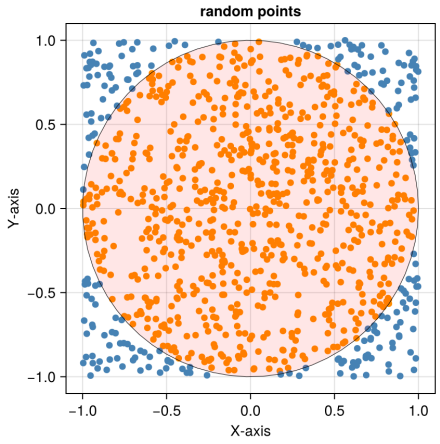


Figure D.1: Points scattered in a square close to the origin.

The result is in Fig. D.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.

D.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[35] that will take a set of points and determine the fraction of them that are inside the unit circle:

```
function fractionInUnitCircle(points)
    # return count(isInUnitCircle,points)/length(points)
    result = 0.0
    for p in points
        if isInUnitCircle(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 estimatePi(n,trials)
    data = [4*fractionInUnitCircle(throwPointsDown(n,2)) for trial 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:

```
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. D.2. It looks terrible, but at least we can tell that π is somewhere between 2.8 and 3.2.

[35] 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.

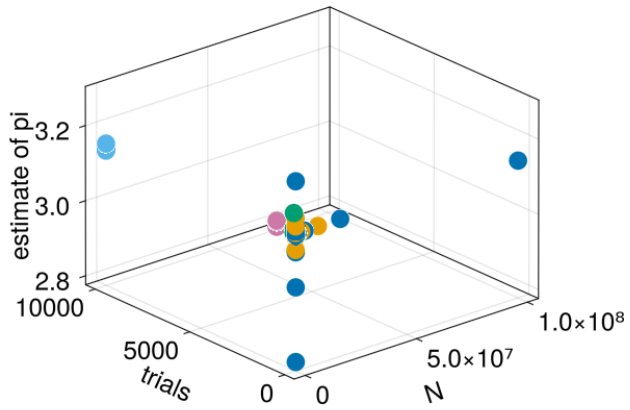


Figure D.2: A 3D plot estimating π as a function of number of points thrown down in a square and the number of trials. This plot is not meant to look good (and it doesn't).

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. D.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 D.3 is a step in the right direction – far from perfect, but better.

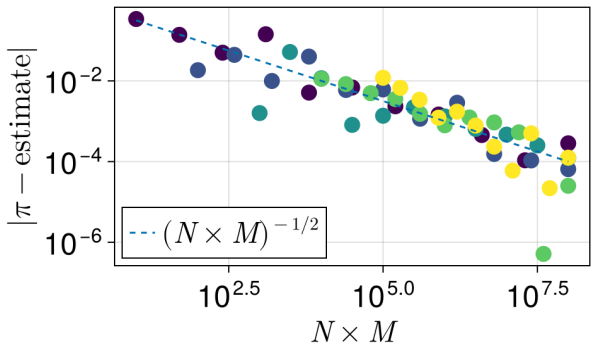


Figure D.3: Difference between π and its simple Monte Carlo estimate vs the product of the number of points (N) and the number of trials averaged over (M). Notice, also, that Makie let's us easily customize the aesthetics of our plots. Please don't pick tick mark label fonts and axis label fonts that clash as much as they do here.

D.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[36]. 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

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

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:

```
julia> const fineStructureConstant = 0.0072923525643;
```

Functions should consistently return values of the same type[37], 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.

D.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` 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[38] 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)
```

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

[38] We saw our first macro, `printf`, in Appendix A, 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

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 generatePoint!(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 estimatePiInPlace(n,trials)
    #pre-allocate arrays
    currentTrial = [Vector{Float64}(undef, 2) for i in 1:n]
    data = Vector{Float64}(undef,trials)
    for i in 1:trials
        for j in 1:n
            generatePoint!(currentTrial[j],2.0)
        end
        data[i] = 4.0*fractionInUnitCircle(currentTrial)
    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.

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[39]?

[39] You bet!

Appendix E

Modules, 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 learn about Julia's system for organizing code (and its excellent package management system), how its 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. These topics could each easily deserve their own chapter, so this might feel like a major ramp-up in information density. But don't worry – we'll digest these sections in chunks, and the structured Problems will hopefully help guide you through the material!

E.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 **??**!

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[40] 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. 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.

E.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

[40] 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.amazingFunctionDefinedInMyModule()
```

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[41]

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) [4] to estimate the value of π. EDMD is an alternative to standard molecular dynamics (something we'll spend much more time on in Module **??**) – 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:

[41] 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[42] 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
# 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[43], start Julia, activate a local environment with "`]activate .`" and import[44] our module. The functions in the package are now available to us using the module's namespace, for instance:

```
julia> import EventDrivenMolecularDynamics
```

After which we could do:

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

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

[44] 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> 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.

E.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[45]. 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 (like a function, or the body of a loop) to be able to see the broader context of an "outer" scope. But at the same time, we want *encapsulation*: we don't want an outer scope to be *accidentally* changed by what happens inside a function, nor do we want two separate functions to interfere with each other just because they both happen to 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.

[45] 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.

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 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 your REPL code with this extra keyword[46] 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.

[46] 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.

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).

E.3 Building type hierarchies: parametric and abstract types

E.3.1 Parametric composite types

Back in Appendix C.3 we defined a "`ParticlePosition`" mutable structure which held three 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
```

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.);
```

E.3.2 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], `<:`, like so:

```
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
```

[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`."

```
""" A concrete Archosaur (skipping a division)"""
struct SaltwaterCrocodile <: Archosauria
    name::String
    numberOfTeeth::BigInt
end
""" Another concrete Archosaur"""
struct BeeHummingbird <: Archosauria
    name::String
    lengthInMillimeters::Float64
end
```

Unlike in some other languages where classes can inherit member functions and variables from other classes, Julia's `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}
    normalVector::SVector{D,T}
    pointOnPlane::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.

E.4 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, 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(A::algorithm, x::Number, y::Number)`. 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. Julia generally does the thing that you expect: it selects the *most specialized*[50] method that matches the argument list.

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

[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.

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 Appendix C.3?

In the context of our EDMD simulation, we can write an `elasticCollision` 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:

```
function elasticCollision(p1::Particle{D,T}, p2::Particle{D,T}) where {D,T}
    #logic to define new Particles post-collision...
    #newParticle1=...
    #newParticle2=...
    return newParticle1,newParticle2
end

"""when a particle and an obstacle collide, the particle reverses its
velocity"""
function elasticCollision(p::Particle{D,T}, obstacle::AbstractObstacle{D,T})
where {D,T}
    newVelocity = -p.velocity
    newParticle = Particle{D,T}(p.position,newVelocity,p.mass,p.radius)
    return newParticle,obstacle
end
```

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 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:

```
"""For a call with o first, just call the method with the (p,o) order"""
function elasticCollision(obstacle::AbstractObstacle{D,T}, p::Particle{D,T})
where {D,T}
    newParticle, o = elasticCollision(p,obstacle)
    return o,newParticle
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!

E.5 Hello, π! (Method 5: Counting collisions)

If you haven't seen this calculation[51] 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. [5] 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. E.1), and you should spend some time thinking about why this might even be tangentially related to π.

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 `piPoolInitialization(massRatio)` 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:

[51] Which is, inarguably, the silliest calculation I know.

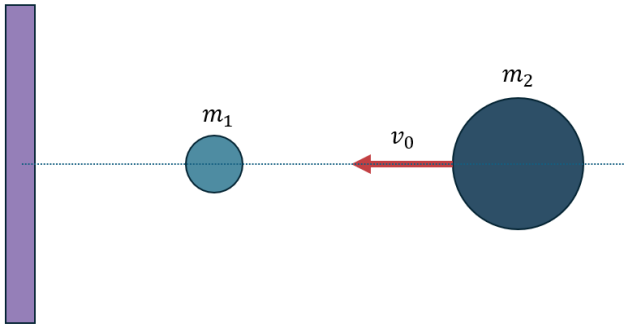


Figure E.1: Our physical setup for calculating π: an immovable wall (e.g., of infinite mass) is on the left. In the middle is an initially stationary particle of mass m_1. On the right is a particle of mass m_2 moving with initial velocity along the x axis towards the middle mass.

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

function piFromPool(massRatio::Float64)
    s::System = poolPiInitialization(massRatio)
    timeToPreviousCollision = 0
    while(timeToPreviousCollision != Inf)
        timeToPreviousCollision = evolveStep!(s)
    end
    return s.totalCollisions
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.piFromPool(1.)
3
julia> EventDrivenMolecularDynamics.piFromPool(100.)
31
julia> EventDrivenMolecularDynamics.piFromPool(10000.)
314
julia> EventDrivenMolecularDynamics.piFromPool(1000000.)
3141
```

```
julia> EventDrivenMolecularDynamics.piFromPool(100000000.)
31415
julia> EventDrivenMolecularDynamics.piFromPool(10000000000.)
314159
julia> EventDrivenMolecularDynamics.piFromPool(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.

E.6 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[52]). 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.

[52] 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.

- "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.

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.

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

E.7 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 Appendix A 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!

Bibliography

[1] 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.

[2] William Oughtred. *Clavis Mathematicae denuo limita, sive potius fabricata*. Lichfield, 1631.

[3] Florian Cajori. *A history of mathematical notations*, volume 1. Courier Corporation, 1993.

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

[5] 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.