

# High Performance Computing in Julia

from the ground up.



# High Performance Computing in Julia

from the ground up.

Jamie F. Mair

Jamie F. Mair  
Nottingham, England  
United Kingdom

© 2022 Jamie F. Mair

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording or information storage and retrieval) without permission in writing from the publisher.

This book was set in T<sub>E</sub>X Gyre Pagella by the authors in L<sup>A</sup>T<sub>E</sub>X.

*This book is an introduction to becoming a Research Software Engineer with particular focus on parallel and GPU programming.*



# Contents

*Preface* xi

*Acknowledgments* xiii

## PART I PRELIMINARIES

1	<i>Module Outline</i>	5
1.1	Aims	5
1.2	Prerequisites	5
1.3	Other Information	6
2	<i>Getting Started</i>	7
2.1	Installing Julia	7
2.2	Installing Visual Studio Code	8
2.3	Installing Git and GitHub Desktop	8
2.4	Creating a GitHub Account	10
2.5	Using GitHub Classroom	11
2.6	Getting Help with Git	11
3	<i>Hardware &amp; Software Basics</i>	13
3.1	Introduction to Hardware	13
3.2	Introduction to Software	19
4	<i>Julia Fundamentals</i>	37
4.1	<i>Why Julia?</i>	37
4.2	Basic Syntax	43
4.3	Advanced Syntax	49
4.4	How does Julia work?	56
4.5	Package Management	62

5	<i>Measuring Performance</i>	67
5.1	Timing	67
5.2	Benchmarking	68
5.3	Profiling	71
5.4	Memory Usage	72
5.5	Computational Complexity	77

## PART II HIGH PERFORMANCE CODE

6	<i>Optimising Serial Code</i>	87
6.1	Caching and Memory Locality	87
6.2	Reducing Allocations	98
6.3	Memoisation	105
6.4	Vectorising vs Loops	106
6.5	Views	108
6.6	Speed of Operations	109
6.7	Type Stability	111
6.8	Constant Propagation	122
6.9	Inlining	125
6.10	Generated Functions	126
6.11	Conclusion	129
7	<i>Introduction to Parallel Programming</i>	131
7.1	Dependency Graphs	133
7.2	Theoretical Expectations	137
7.3	Maps and Reductions	141
7.4	Thread Safety and Race Conditions	144
7.5	When to parallelise?	152
7.6	Summary	154
8	<i>Multithreading</i>	155
8.1	Multithreading in Julia	157
8.2	Task-based Parallelism	163
8.3	Packages	165
8.4	Summary	167
8.5	Exercises	169



9	<i>Multiprocessing</i>	171
9.1	Multiprocessing in Julia	172
9.2	Multiprocessing on a Cluster	178
9.3	Message Passing	180
9.4	Summary	181
9.5	Exercises	181
10	<i>Introduction to GPU Programming</i>	185
10.1	Modern GPU Hardware	185
10.2	GPU Vendor Overview	187
10.3	High Level Introduction to CUDA	187
10.4	CUDA Kernels	193
10.5	CUDA Libraries	198
10.6	Benchmarking & Profiling	199
10.7	Tips	201
10.8	Case Study: Monte-Carlo Simulations	202
10.9	Exercises	206

### PART III PROFESSIONAL SCIENTIFIC CODE

11	<i>Version Control</i>	209
11.1	What is Git?	210
11.2	Working with Teams	215
11.3	Keeping a Clean Repository	216
11.4	Learning Git	217
12	<i>Reproducibility</i>	221
12.1	Controlling for Randomness	221
12.2	Managing Software Versions	225
13	<i>Documentation</i>	229
13.1	Commenting is <b>not</b> Documentation	229
13.2	Better Variable and Function Names	235
13.3	Automatic Documentation	237
13.4	Summary	240
14	<i>Unit Testing</i>	243
14.1	Why test your code?	246
14.2	Writing Good Unit Tests	248
14.3	Bad Unit Testing	254

X CONTENTS

15	<i>Code Organisation</i>	257
15.1	Folder Structure	257
15.2	Modules	258
	<i>References</i>	261

# *Preface*

These lecture notes are intended for teaching some Advanced Programming and Optimisation techniques in Julia. All examples in the book use real code written in Julia.

JAMIE F. MAIR  
Nottingham, England.  
2023-01-26



# *Acknowledgments*

The author wishes to thank Mykel J. Jochenderfer and Tim A. Wheeler for providing this template to create these lecture notes, along with Edward Tufte - whose work inspired the style of this book.

The style of this book was inspired by Edward Tufte. Among other stylistic elements, we adopted his wide margins and use of small multiples. In fact, the typesetting of this book is heavily based on the Tufte-LaTeX package by Kevin Godby, Bil Kleb, and Bill Wood.

We have also benefited from the various open source packages on which this textbook depends. The typesetting of the code is done with the help of `pythontex`, which is maintained by Geoffrey Poore. Plotting is handled by `pgfplots`, which is maintained by Christian Feuersänger. The book's color scheme was adapted from the Monokai theme by Jon Skinner of Sublime Text (`sublimetext.com`). For plots, we use the `viridis` colormap defined by Stéfan van der Walt and Nathaniel Smith.



```
julia> include("support_code.jl")  
== (generic function with 308 methods)
```





**PART I:**

**PRELIMINARIES**



# 1 *Module Outline*

Welcome to the High Performance Computing in Julia module! This book will serve as the lecture notes to the module, covering and expanding upon the topics in the lectures. Check out the website<sup>1</sup> for an overview of the MPAGS<sup>2</sup> module.

<sup>1</sup><https://jamiemair.github.io/mpags-high-performance-computing/overview/>

<sup>2</sup>Midlands Physics Alliance Graduate School

## 1.1 *Aims*

The aim of this module is to provide the student with the tools to write fast and efficient code. We will touch on a few algorithms and basic examples, but most of the focus will be on the practical implementation. The module will teach the students how to program in Julia and how to take advantage of modern hardware, by being able to write parallel and GPU based code.

## 1.2 *Prerequisites*

In order to take this module, you should have the following prerequisites:

- Some recent programming experience (any language is fine).
- Basic mathematics skills. This includes basic algebra and some basic calculus which is needed later in the module.
- A willingness to learn something new!

### 1.3 *Other Information*

The information for this course will be posted on the website<sup>3</sup>. This includes details of the timetable, format and assessment of the module. The content of the lectures will roughly follow the content in this book, however, this book will aim to be more comprehensive.

<sup>3</sup><https://jamiemair.github.io/mpags-high-performance-computing/overview/>

## 2 *Getting Started*

This chapter is dedicated to getting set up with all the software you will need to follow along with the course and complete the assignments. The first assignment of the module will have you complete this section. In this chapter, the following will be covered:

- Downloading and installing Julia via Juliaup
- Installing Visual Studio Code
- Installing Git and GitHub desktop
- Creating a GitHub account
- Using GitHub classroom

### 2.1 *Installing Julia*

There are a few ways to install Julia. It is recommended to use the latest version of Julia, as there are frequent updates and improvements which are most likely non-breaking. Usually, it is safe to update to the most recent version without breaking your existing code. For this reason, we recommend using the `juliaup`<sup>1</sup> tool to download and manage which Julia version is installed on your machine. Visit the `juliaup` GitHub page and follow the instructions in the README to install this on your system.

<sup>1</sup><https://github.com/JuliaLang/juliaup>

Once installed, run open a terminal<sup>2</sup> and type `Julia`. This adds the most recent

```
juliaup add release
```

channel.

We also want to make this the default Julia installation, which can be done by running the command

```
juliaup default release
```

Once this is done, you should be able to type `julia` into your terminal and launch the REPL, showing the most recent version. At the time of writing the version of Julia used is Julia v.1.8.4 (Dec 23, 2022).

## 2.2 *Installing Visual Studio Code*

Visual Studio Code (or VS Code) is a lightweight Integrated Development Environment (IDE), which helps you write code. It supports a comprehensive development experience for almost any programming language. It is currently the most supported editor by the Julia developers.

Head to the download link<sup>3</sup> below and follow the instructions to download it for your platform. Visit the documentation for the VS Code Julia extension<sup>4</sup>. Follow the instructions to download the official Julia extension.

You can start the Julia REPL (this stands for Read Evaluate Print Loop) by pressing `Ctrl+Shift+P` to open the command palette and start typing in `Start REPL` and then select the option to start the Julia REPL.

The reason that we want to use the integrated REPL is so that plot viewer and other tools in Visual Studio Code can communicate with Julia. You can use any terminal to run your Julia code, but this makes the development experience a bit easier.

## 2.3 *Installing Git and GitHub Desktop*

In order to complete the assignments, you will need to use Git. No advanced Git skills are required for this course, but it is highly recommended that you learn

<sup>2</sup> On Windows, you can use either Powershell or Command Prompt, which can be searched in the Start Menu. On Mac, find a guide online on how to open a terminal if you do not know.

<sup>3</sup> <https://code.visualstudio.com/>

<sup>4</sup> <https://code.visualstudio.com/docs/languages/julia>

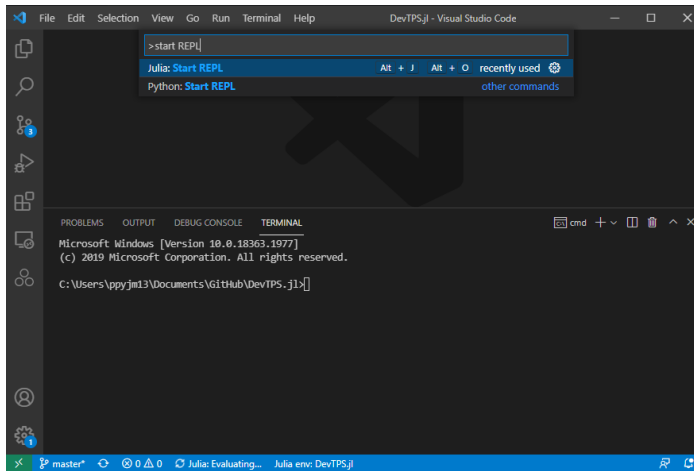


Figure 2.1. A screenshot of starting a Julia REPL within Visual Studio Code. This is done by using the command palette, accessible using Ctrl+Shift+P.

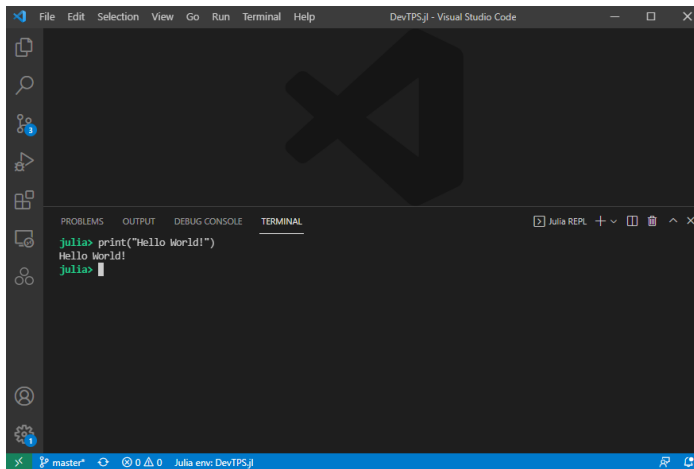


Figure 2.2. A screenshot of writing code in the Julia REPL from within Visual Studio Code. This REPL is located in the "Terminal" window, usually at the bottom of VS Code.

the basics as this is an **essential** skill for any programmer. This module does not require and actively discourages the use of Git via the command line. It is recommended to use a Git GUI client (there are many to choose from) to interact with Git. This section chooses GitHub Desktop, as this is one of the easiest Git clients to use and is available for both Windows and MacOS.

### 2.3.1 *GitHub Desktop*

GitHub Desktop is what is known as a "Git Client". Git is the underlying software that tracks changes in your code. It works by looking at all the files in folder on your computer and seeing if there are any changes. A repository is just a folder for your project, which is tracked by the Git software. GitHub desktop gives you a user-friendly way to interact with Git, instead of the user having to learn the command line tools. Head to the GitHub desktop website<sup>5</sup> to download GitHub desktop for your specified platform. Install the software and create a GitHub account and login here.

<sup>5</sup> <https://desktop.github.com/>

### 2.3.2 *Git*

Installing the Git tool separately is a good idea since it can integrate with all of your software tools. Most importantly, this allows you to use some of the source control features inside VS Code. Head to the Git download page<sup>6</sup> to install Git on your platform.

<sup>6</sup> <https://git-scm.com/download>  
5

## 2.4 *Creating a GitHub Account*

GitHub will be used to store a copy of your code on the cloud, acting like a backup and as a "source of truth" for the proper version of your code. This will act as a "remote" for your repository. You will also hear the GitHub version of your repository called the "origin". You want to keep the GitHub version of your code and your local version as close as possible, make sure to use the "fetch" and "sync" buttons in the Git client as frequently as possible.

Follow the instructions to create an account or use your old one. You will need this to login to the GitHub desktop client. This allows you to clone your private repositories to your own machine.



## 2.5 *Using GitHub Classroom*

We are going to use GitHub classroom to assign workshops and projects to each student. Every student will have their own separate repository for each assignment. This will be hosted under our organisation, and not privately on your account. If you want to keep a copy of this code after you graduate, make a local copy. We would prefer if you not make any code from this module public, for academic integrity purposes.

You will need a GitHub account to use GitHub classroom. Make sure you are logged in on your browser and copy the assessment URL in a format like `https://classroom.github.com/a/XXXXXX`. Accept the assignment and find your name in the Moodle list, so we know whose GitHub accounts is whose. The repository will be created in the organisation page which will be linked in the Moodle page. Search for your username to find the repositories that you have access to. Clone this repository to your local machine.

## 2.6 *Getting Help with Git*

This module will not focus on the specifics of Git, and we highly recommend that you make use of Google to find answers to specific questions. During this course, the Git skills that are required are:

1. Cloning a Git repository to your local machine
2. Making local changes to your code, staging and committing them locally
3. Pushing your commits to the cloud<sup>7</sup>
4. Pulling changes from the cloud to your local machine
5. (Optional) Working in a separate branch and making Pull Requests to merge your code into the main branch

<sup>7</sup> Sometimes called the origin. This is where the code is hosted and backed up on GitHub.

Use these keywords to help your Google searches.



## 3 *Hardware & Software Basics*

This module was designed with students from non-Computer Science STEM backgrounds, who may have never had any formal training in fundamental Computer Science (or Software Engineering) topics. As being able to write high performance code is highly dependent on a solid understanding of how a computer works, we will start with a lightning tour through the fundamentals of modern hardware and software. Here, we do not aim to provide a deep understanding of all the nuances of these topics, but provide a guiding framework which we can use to reason about the operation and optimisation of our own computer programs.

As modern computers are intractably varied and complex, it is often futile to try and understand the operation of a specific computer in minute detail, however, it certainly is possible to construct a useful mental model of a computer, to help us ask questions and effectively investigate and find the answers to these questions. This mental model will become essential when we start thinking about optimising our code and making sure the code we write is correct.

### 3.1 *Introduction to Hardware*

Virtually all modern computers use a similar hardware architecture. There are many universal standards which manufacturers use to ensure they have high compatibility across multiple devices. Today, it is extremely easy to build up a computer from a collection of components, which can all be produced from different manufacturers - it essentially amounts to having a very expensive Lego kit. We will briefly discuss the most important components which make up a modern desktop computer.

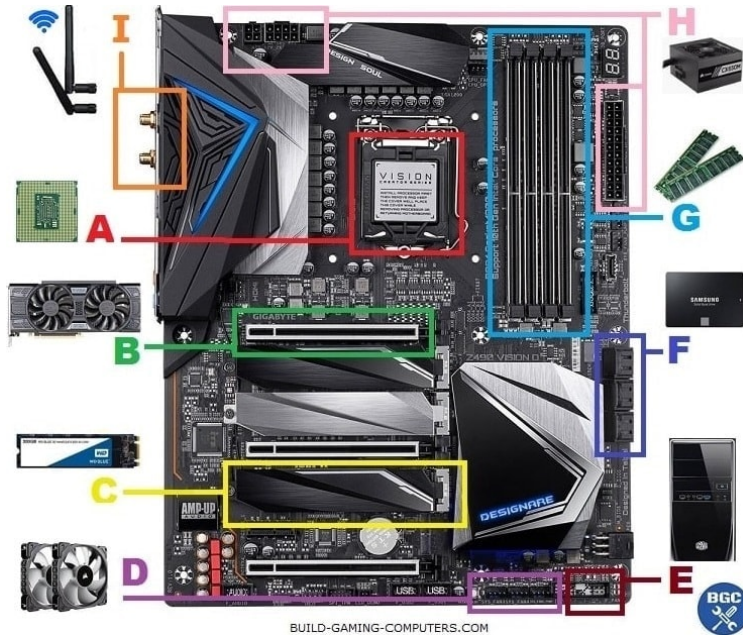


Figure 3.1. A labelled diagram of a typical modern motherboard.

### 3.1.1 Motherboard

Let's take a closer look at Figure 3.1. This picture depicts a modern motherboard with many sections labelled. A motherboard acts as the *glue* which connects all the disparate parts of the computer together. It helps facilitate power delivery to each component<sup>1</sup>, along with data transfer to facilitate communication between all the different parts. Motherboards also provide most of the external connections to the outside world, and handle the communication to the parts inside. Most of the ports you will see on the back of a computer are directly connected to the motherboard.

### 3.1.2 Central Processing Unit (CPU)

The CPU is the central part of any computer. It acts as the “brain” of the computer and as the name suggests, handles all the processing of the computer. This device usually sits near the centre of the board. In Figure 3.1, the CPU is housed in socket A. CPUs often come in different form factors and are designed to fit in

<sup>1</sup> Note that some components require a lot of power which also draw power directly from the PSU (Power Supply Unit) as well as from the motherboard.

different sockets, however, there is usually overlap between the same products that a manufacturer produces.

Modern CPUs are often designed to be to processes multiple tasks at the same time. For example, when you use your computer, there are usually multiple programs all running at the same time, such as browsing the web, writing a document and listening to musics. The way that CPUs used to handle doing these tasks at the same time is by using a scheduler to switch between the tasks so quickly that the user would not notice that they were not happening at the same time. This is called **concurrency**, and it is the illusion of work being done in parallel. However, modern CPUs have truly parallel capabilities since there are multiple processing cores contained within the computer, that can all process information in parallel.

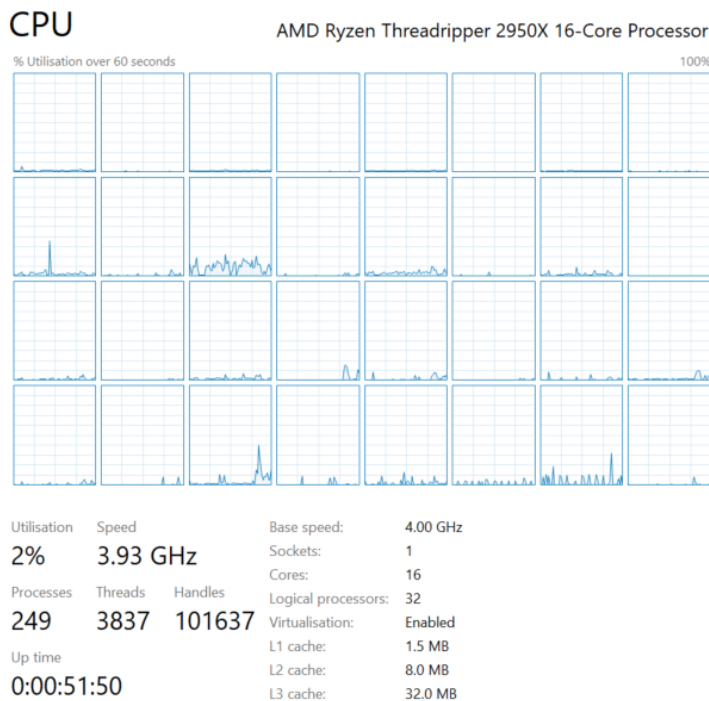


Figure 3.2. A screenshot of Windows Task Manager, showing the information about an AMD Threadripper processor.

If you look at Figure 3.2, you can see that this processor has 16 cores. This means the entire CPU is made up of 16 processing units, that can individually work on

a task. Task manager shows 32 “logical processors” on the computer, which may be confusing at first. This is quite common for many higher-end chips. This is known as **Simultaneous Multithreading** (SMT) or **Hyperthreading**<sup>2</sup>. SMT is a special type of technology which increases the throughput of each individual core, allowing an increased amount of parallel processing. Not all workloads can benefit from this increased performance, but highly parallelised workloads may see increased performance on these chips when using all the logical processors, and not just each physical core.

<sup>2</sup> The name when it is used in Intel Chips

Most modern CPUs also contains a small amount of *very fast* working memory called the cache. The cache of a CPU is usually arranged in levels from 1 to 3, 1 being the fastest and closest to the processor and 3 being the slowest. These levels are called L1, L2 and L3 cache. This cache is usually very small - the Threadripper processor shown in Figure 3.2 has only 96 KB per core on my CPU. This is only enough to store around 1500 64-bit floating point numbers. While that may seem like a lot, compared to main memory, it is barely a rounding error. Despite this very low capacity, it is essential for reaching optimal processing speeds, as it is the fastest way to get information to the processor. Many workloads are often bottlenecked by the amount of cache available on a given CPU.

### 3.1.3 *Random Access Memory (RAM)*

The RAM of a computer acts as the “working memory” part of the computer. Any information stored here is “volatile”, which means that its contents are erased when the computer is powered off. It is usually many orders of magnitude faster than permanent storage to both read and write to. It is located near the CPU, in sockets G in Figure 3.1. As computing requirements have increased, the total RAM is usually spread over multiple chips in multiple sockets. It should be noted that the speed of RAM is usually much slower than the cache. We usually see an inverse relationship between capacity and speed. In this case, RAM has more capacity than CPU cache but at the cost of being much slower.

### 3.1.4 *Physical Storage*

A computer needs a place to store information long-term, such as the operating system and user files permanently, even when the computer is turned off. As it is not appropriate to store this in RAM, a separate device is needed. The traditional

medium for this storage is known as a Hard Disk Drive (HDD), and is made up of a magnetic disk which physically spins. Seen in Figure 3.3, you can see that this device looks a bit like an old-fashioned record player and works similarly. The main disk spins and a head that can read and write rotates to find the desired information.



Figure 3.3. A spinning hard disk drive.

HDDs are usually very high capacity, while an average desktop computer today will have around 8 GB of RAM, the average hard drive starts at around 1 TB, almost 100 times the capacity. The storage on this drive, however, is very slow. It is so slow that CPUs have dedicated hardware built in to communicate with these devices asynchronously, so that the CPU can continue to do other things while it waits for the information from the hard drive. As this module is about optimisation, the main thing to remember that any communication with a hard drive is very, very slow, and should only be done when necessary. This can be a serious bottleneck of most poorly designed algorithms.

While HDDs were traditionally the main storage devices used, there has been a huge uptake of a newer technology - Solid State Drives (SSDs). These new devices have no spinning parts and consequently, draws far less power than the spinning alternative. These devices also have a huge increase in speed, being anywhere from 10 to 1000 times faster than spinning disks. However, the price is far higher for the same capacity. The “Price Per GB” is usually at least 4 times higher for SSDs than for spinning disk, however, the speed of the SSD is usually far higher.

Despite SSDs being much faster than HDDs, they are still incredibly slow next to accessing RAM. Anything that can be loaded into RAM before processing is usually better to do, than to load the information while you are processing it.

### 3.1.5 Graphical Processing Units (GPUs)

A GPU is a special type of co-processor<sup>3</sup> that is primarily designed for graphical workloads, such as, deciding which colour each pixel on your screen should be in a given application. A GPU has its own set of fast memory, separate to main memory. The traditional graphics workloads that a GPU was designed for is massively parallel, but doing very similar things. This type of workflow is usually called SIMD which stands for Single Instruction Multiple Data. As such, that GPU usually has many thousands of small processors inside that can perform the same instruction over many pieces of data at the same time. The programs that run on each of these cores is usually called a **kernel**<sup>4</sup>.

More recently, many of these devices are referred to as GPGPU - General-Purpose Graphics Processing Unit - as modern requirements usually have a lot of SIMD workflows which are practical to run on a GPU. In general, the speed of an individual processor inside a GPU is a lot slower and more restricted than a CPU, however, the sheer number of these cores can make processing a huge quantity of data much faster. You can imagine having 1 highly skilled worker that is 10 times as efficient as one unskilled worker. Imagine that in total you have 1 highly skilled worker and 100 unskilled workers. Provided that all workers, regardless of skills, can do a task, if you have 100s of tasks to do, then it is often better to outsource to the 100 workers, since they can work on the task individually, whereas the highly skilled worker can only go through each task one by one. This means that GPUs are usually only suitable for massively parallel tasks.

A GPU is connected to the computer via a PCIe bus<sup>5</sup>. A program is executed on the CPU first. If the program is designed to make use of a GPU, it must communicate with the GPU. Any data that is required for a computation must be copied from main memory to the dedicated GPU memory. A program which executes on the GPU is called a kernel. These kernels are usually written in a special language dependent on the manufacturer such as CUDA<sup>6</sup>.

The architecture of a GPU and the interaction with the CPU will be discussed in more detail in a later section. But for now, remember that a GPU is a lot slower than a CPU for a single task, the benefit only arises when you need to perform a similar task across a huge set of data.

<sup>3</sup> A co-processor is a device which is able to perform computations alongside the main processor, which is usually specialised on certain workloads.

<sup>4</sup> The name - *kernel* - is shared with many other distinct concepts in Computer Science, so one should pay attention to the context in which it is used.

<sup>5</sup> Peripheral Component Interconnect Express - A connection standard used on the majority of modern systems to connect devices to the motherboard, with direct communication access to the CPU.

<sup>6</sup> Compute Unified Device Architecture - An NVIDIA proprietary programming interface which provides a software layer to gain direct access to the GPU's virtual instruction set.



## 3.2 Introduction to Software

### 3.2.1 Operating Systems

Almost every modern computer system runs an operating system, which manages computer hardware and software resources and provides common utilities to software running on the system. There are 3 main types of operating system that you will need to be aware of. Most desktop computers run Windows, which is usually Windows 10, or more recently, Windows 11. A substantial part of the market also uses macOS. Finally, a huge amount of the worlds computing infrastructure runs on Linux<sup>7</sup>. You may have heard of some Linux *distributions*, such as *Debian*, *Fedora* and *Ubuntu*. Most supercomputing clusters run on Linux, commonly a version of *Red Hat Enterprise Linux*.

For us as researchers, it is important that our code can be run on multiple different operating systems, as you will likely use your home machine (likely Windows or macOS) to develop your code and then move to a Linux cluster to run your code at scale. It is now easier than ever to write cross-platform code which can execute on a wide variety of systems. This is why a lot of scientific oriented code is written in C/C++ or Python, as there is much support across all operating systems. Most major programming languages today will be able to run on any platform. Julia is no different, and can run on most platforms.

Now we know what operating systems are out there, let us talk about what they do for you. The Operating System handles the basic operations that programs use regularly. They handle interacting with files on the permanent storage, they handle the memory allocation in RAM and most importantly, they handle the scheduler for the processor. An OS acts as an intermediary between the hardware on a computer and the programs and applications themselves. Its role can be seen in Figure 3.4.

An operating system is incredibly useful as it abstracts away the hardware details and allows developers to deploy their application across a huge range of computers, with high confidence that they will work.

Today, most languages provide abstracts to perform common tasks independent of the specific operating system. For example, the `os` library in Python provides functions for constructing file and folder paths, checking whether directories or files exist etc. These are functions typically handled by the OS directly, but when using these abstractions, the same code can work on Linux, Windows

<sup>7</sup> Linux is an open source project which refers to a family of operating systems based on the Linux Kernel. There is a related project called OpenBSD which is also very similar to a Linux based system.

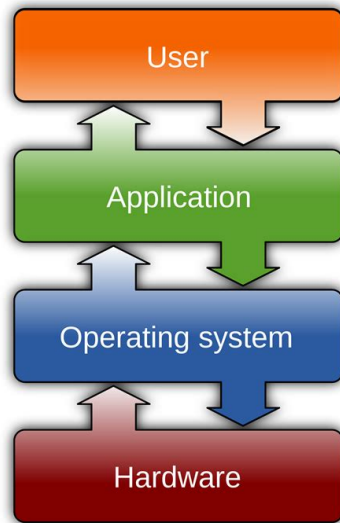


Figure 3.4. Source: Wikipedia

and macOS with no requirement to change it, despite the operating systems using different formats for file paths.

### 3.2.2 Compilers and Machine Code

While modern architecture is built through abstraction, when it comes time for the code to run, the hardware must eventually be able to execute the program. In order for a program to be executed, it must be *translated* to machine code. Machine code is made up of very simple, processor dependent instructions, which manipulate data at the most atomic level, sometimes a single bit at a time. Machine code is very difficult for humans to read and write, which is why we write source code in a higher-level language. This higher level language is human-readable and is called the **source code**. Another piece of software is then required to take the human-readable source code and translate it, sometimes through multiple layers of abstraction, to eventually become machine code which is then executed on the system.

There are many varied approaches to take source code and translate it into machine code. Older languages tend to use a static compiler which translates the entire set of source code into a **binary**<sup>8</sup> This method will need to be repeated on

<sup>8</sup> An executable bundle of instructions, specific to the hardware architecture and operating system it will be executed on.

each platform on which you wish to run this software. For this reason, languages like Java or C# have taken the approach of creating an intermediate language which is designed to be executed on a *virtual machine*. This allows the partially compiled byte code (often called an Intermediate Language) which can be **just-in-time** compiled into native code<sup>9</sup>. Languages like Python are usually executed via the use of an **interpreter** which line-by-line translates the source code into native code and executes this as it goes.

The process of translating source code into native code has a huge variety of approaches, but eventually all source code needs to be translated into machine/-native code to run on the hardware.

At this point, it is important to make the distinction between a programming language and the toolchain used to execute code written in that language. A language is just a standard set of definitions for how a piece of source code will execute. However, for brevity, we often refer to a language and its most common (and sometimes only) toolchain as a bundle. E.g. we say that C/C++ is a compiled language, however, it is possible to write an implementation of a toolchain which interprets C/C++ code instead of compiling it. In the case of Julia, the only real implementation is using the LLVM compiler backend, and so when we refer to Julia's compilation processor, we are referring to the LLVM compiler. This is similar for the vast majority of languages, however C/C++ or Fortran have a vast array of different compilers as they have been around for a very long time. From this point on, any reference to execution of C/C++ or Python etc assumes we are using the most common toolchain for the language<sup>10</sup>.

For languages such as C/C++, this compilation happens ahead of time, before the code is executed. This is what makes it a compiled language. The output of the compilation is sometimes referred to as a **binary**. This refers to the nature of machine code being written in binary. On Windows, this file has the .exe extension.

For languages like Julia, things get a bit more complicated. It uses a **Just-Ahead-of-Time** compilation model. This works by having a Julia *runtime* which translates source code on the fly into machine code, just before it is needed. Once the compilation happens once, it is stored (cached), which means the compilation process can be skipped on the next execution. This allows the language to be very dynamic and flexible, giving more expressive power to the developer at a small cost of startup time<sup>11</sup>.

<sup>9</sup> Native code is native to the architecture of the machine running the program. For example, using x86 instructions on modern Intel or AMD processors.

<sup>10</sup> For Python, we refer to CPython. For C/C++ we refer to the any compiled toolchain, such as *Clang* which uses LLVM to compile the code much like Julia and also Rust

<sup>11</sup> There is much work being done to be able to precompile code to reduce the startup time of the Julia runtime. Currently, it is very difficult to statically compile a standalone binary, but packages such as *PackageCompiler.jl* are making huge progress. The main benefits of more recent Julia versions is the great improvement in loading

In general, most modern compilers are very smart, and are able to optimise source code written to be as efficient as possible, with little input from the developer. Only recently has an attitude of “trust the compiler” become prevalent in developers. Most of the time, the compiler has tricks to make the code fast, and often times you can focus on writing clean code, without worrying too much about optimisation. For example, if you are calculating a sum, which is known at compile time, the compiler will often precalculate your value and not run the calculating every time.

```
function my_large_sum()
    s = 0
    for i=1:100000000000
        s += i
    end
    return s
end
```

Algorithm 3.1. A simple function which performs  $10^{11}$  many additions.

Let’s take Algorithm 3.1 for example. When we run the method we get the following:

```
julia> import BenchmarkTools: @btime;
julia> @btime my_large_sum()
2.063 ns (0 allocations: 0 bytes)
932356074711512064
```

This means it only took a couple of nanoseconds to calculate the answer. This does not seem feasible since the function has to iterate through  $10^{11}$  integer additions. However, the compiler is very clever - it knows that this function only operates on constant values, and so it can precalculate the output, removing the need to make the calculation each time. The compiler collapsed all the code to just return the result of the calculation, without having to go through the for loop every time.

We can see that if the compiler has access to more information, such as a value being constant, it can make sophisticated optimisations to avoid having to perform certain calculations at runtime.

Julia’s compiler is called LLVM which is a very established and popular compiler. You can see the actual assembly code (analogous to machine code) produced by this function:

```

julia> @code_llvm debuginfo=:none my_large_sum()
define i64 @julia_my_large_sum_1682() #0 {
top:
    ret i64 932356074711512064
}

```

This might be slightly difficult to read, but it is clear that there are no calculations happening here. The function simply returns the precomputed value.

This sort of behaviour is just one of the many compiler tricks that are used to make your code faster. If you are every wondering what your code is doing, then you can check various levels of output in the translation process to see what is happening. Some helpful macros are `@code_typed`, `@code_llvm` and `@code_native`.

### 3.2.3 *Threading and Multiprocessing*

Modern CPUs have access to multiple cores. It is standard for most modern desktop computers to have a CPU with at least 4 cores inside. High-Performance Computing clusters are made up of a set of computing “nodes” (a term for a single machine in a cluster), which are networked together. Typically, each node has 1 or 2 CPUs with anywhere from 20-128 cores each. It is important to know how to harness the computing power of machines with multiple cores, as the speed of individual cores is not really improving over time, and the only way to access more compute is to scale horizontally.

There are many approaches to parallelism (on the CPU) which you will need to understand. The first of which is called “multithreading”. A thread can be thought of as a small unit of instructions which is part of an overall program/process. A process can have multiple threads inside, which usually have access to shared resources, such as memory. Different processes do not share these resources, such as memory. What is important to understand is that different threads can execute at the same time, and have access to the same data, so it is very important to make sure that only one thread can access (read/write) to the same piece of data at any one time.

Processes on the other hand, do not have access to shared memory resources, and are isolated from one another. However, one can communicate information between different processes via ports or pipes. You can think of these as a communication channel between different workers. These are implemented in many

ways, but it allows for a workload to split up across different processes. Since these processes only need a communication channel to function, you can split up the workload across multiple computers and scale up to use 1000s of computers if needed.

Most programs often use threading as it is much simpler to use and implement, and is very efficient for sharing resources across each thread. However, some languages like Python have restrictions on which data can be accessed by which thread, which makes threading in Python very slow. Instead, Python uses multiprocessing to split work across multiple different processes to gain a speed-up on systems with multiple cores and parallel workloads.

### 3.2.4 Hardware Instructions

Some modern CPUs have built into them the ability to manipulate multiple pieces of data at the same time, provided they are in a suitable format. Take the problem of adding two vectors together.

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, \quad (3.1)$$

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}, \quad (3.2)$$

$$\vec{x} + \vec{y} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \\ x_4 + y_4 \end{bmatrix} \quad (3.3)$$

Notice that there are 4 add operations here, but they are all independent of one another. Modern CPUs tend to have advanced instructions for doing this operation all at once. It works by taking the 4 values from the first and second vector all at the same time, and putting them in a special arithmetic unit on the CPU which can add numbers in parallel. There are many implementations in hardware, but the most common ones are *AVX*<sup>12</sup> and *SSE*<sup>13</sup>. These specialised

<sup>12</sup> [https://en.wikipedia.org/wiki/Advanced\\_Vector\\_Extensions](https://en.wikipedia.org/wiki/Advanced_Vector_Extensions)

<sup>13</sup> [https://en.wikipedia.org/wiki/Streaming\\_SIMD\\_Extensions](https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions)

instructions are known as **hardware SIMD** (Single Instruction Multiple Data).

Let’s look at an example implementation of SIMD in Algorithm 3.2.

```
function custom_sum(numbers)
    total = zero(eltype(numbers))
    @inbounds for x in numbers
        total += x
    end
    return total
end
function custom_sum_simd(numbers)
    total = zero(eltype(numbers))
    @inbounds @simd for x in numbers
        total += x
    end
    return total
end
```

Algorithm 3.2. A simple example of two implementations of a function which performs a sum over an input array of elements.

The `@inbounds` macro simply tells the compiler to ignore bounds checking (making sure that you index into an array correctly). If we benchmark these functions we see the following results:

```
julia> numbers = rand(Float32, 256);
julia> @btime custom_sum($numbers)
162.618 ns (0 allocations: 0 bytes)
130.38246f0
julia> @btime custom_sum_simd($numbers)
6.161 ns (0 allocations: 0 bytes)
130.38245f0
```

We can see that the SIMD version has a significant speed-up (around 18x speedup), but requires very little effort on our part.

We can inspect the code produced, using the `@code_llvm` macro. Consequently, we will see some directives with `< 8 x float >` being produced. This shows that 8 32-bit floating point numbers are being loaded and processed in a single CPU instruction. There are some other optimisations as well, such as using the “fast” math versions of the addition, but overall, the majority of the speed-up comes from using the special instructions on the modern CPU. The hardware on the processor allows for processing 256 bits of information at the same time<sup>14</sup>. Since we used 32 bit numbers, we were able to perform 8 simultaneous additions. If we

<sup>14</sup> Many modern CPUs, such as Apple’s M1 processor, supports AVX-512 which allows processing on twice the number of bits as my machine. This allows for huge acceleration in many computationally intensive workloads

were to use double precision floating point numbers (64 bits each) we could only fit 4 numbers into the SIMD register.

It should be noted that not all operations are suitable for a SIMD speed-up and this optimisation should only be considered for performance critical code.

We can compare the implementation to the built-in `sum` function to see that our implementation is actually a lot faster:

```
julia> @btime custom_sum_simd($numbers)
 6.172 ns (0 allocations: 0 bytes)
130.38245f0
julia> @btime sum($numbers)
22.906 ns (0 allocations: 0 bytes)
130.38248f0
```

We can see that our implementation is even faster than the inbuilt `sum` function. This is likely due to the `sum` function using a slower, but more accurate version of the addition.

### 3.2.5 Data Types

We know that all information on a computer is stored as binary data, simply a series of 1s and 0s called **bits**. As all information is stored this way, it is essential to be able to know what a set of these bits actually means. How does the computer interpret this information and know exactly how to manipulate it? This is exactly what a data type is (or rather a primitive data type) - a specification of how to interpret the bits of some data. The operation for adding two integers is very different from the operation of adding two floating point numbers. The type of your data therefore determines *which* machine instruction needs to be executed. This is essential information. If the code does not know the type of data ahead of time, the processor must do some work to look up or determine the type which specifies how to handle the bits of data. This can be incredibly slow and costly, which is why later in the book, we will focus on giving the compiler information about the types so that it can avoid these costly checks. For the rest of this chapter, we will discuss the different common primitive data types which are common to most programming languages. Modern CPUs have dedicated hardware and instructions to process each of these types.

#### Unsigned Integers



The most basic type of data is an integer. This has the same meaning as in maths, where this represents a natural number. However, for efficiency, we break up integers into different sizes depending on the expected range of values the data will have to cover. In Julia, the unsigned integer types are

```
UInt8 # 8 bits used per number
UInt16
UInt32
UInt64
UInt128
```

We go up by double the number of bits each time. With  $n$  bits, you can store the values from 0 to  $2^n - 1$ . As the range of numbers is limited, we must make sure that the values to be stored in these variables will not exceed the maximum size. If this happens, we get an **overflow error** which can cause logic errors in a program, leading to incorrect results or even program crashes.

You may have realised that a computer (or specifically its processor) is a 64 or 32 bit processor (most modern CPUs are 64 bits). This refers to the **word size** of the registers and communication channels, or in other words, the width in bits of how much the CPU can process/retrieve in one go. The default size of variables usually depends on the **word size** of the CPU, which will be 64 bits on most systems. If you were to specify the `UInt` type in your source code, this will refer to `UInt64` on a 64-bit machine and `UInt32` on a 32-bit machine.

### Signed Integers

In order to store negative integers, we need to alter our representation to allow for negative numbers. One option is to designate the leading bit as a *sign* bit which indicates a positive number when this bit is 0 and negative when it is 1. However, this is not a very efficient approach. One of the most common approaches is to use the **twos complement** method. If we have an  $n$ -bit signed integer, then the leading bit represents the value of  $-2^{n-1}$ . Take a 4 bit number for example. Usually the bit representation of 7 in binary is 0111 as  $7 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ <sup>15</sup>. However, instead of the leading bit representing  $+8$  ( $2^3$ ), we represent it with  $-2^3$ . This means that representing  $-7$  becomes 1001. This method of representation allows for easy expression of addition, despite the negative numbers.

If you find a bug in software, where a number that should always be positive gets very large and then suddenly turns negative, this is because when the 1 gets carried to the leading bit position, the number suddenly becomes negative. This is because the developer has used a signed integer instead of an unsigned

<sup>15</sup> Each of the columns in the binary number represent  $2^x$  where  $x$  is the index of the column starting from 0 on the right-hand side

integer and also used a representation with too few bits to accurately represent the variable.

As signed integer are the most common used, specifying `Int` in Julia refers to a signed integer. Additionally, this will refer to a 64 bit integer on 64-bit systems just like the signed integers in the previous section.

### Boolean Values

Boolean values are simply *true* or *false* values. In most languages, this is represented as an 8-bit unsigned integer. This is because, on most systems, the smallest block of memory which can be addressed is a single byte<sup>16</sup>. We restrict these 8 bits to be either all zeros (representing false) or a single 1 in the trailing bit position to represent true. This is why often languages allow 1 and 0 to be interpreted as *true* or *false* respectively. This is also why math with booleans make sense, as `true * 5 == 5` and `false * 5 == 0`.

<sup>16</sup> 1 byte is equivalent to 8 bits.

Note that storing an array of booleans will likely take up more memory than you expect. In Julia, we can create an array of boolean values with

```
julia> bool_vals = Vector{Bool}(undef, 64);
julia> sizeof(bool_vals) # Outputs number of bytes stored
64
```

Note that we have 64 bytes or 512 bits to store 64 boolean values, which is not very efficient. This is why we can also use a `BitVector` which is designed for this purpose, to allow us to use the minimum amount of memory required. The same array, converted to a `BitVector` has a size of

```
julia> sizeof(BitVector(bool_vals))
8
```

which gives 8 bytes and 64 bits, the expected amount.

### Characters and Strings

Programs often need to be able to interpret text, which is usually stored as an array of characters (often called *chars*). Traditionally we used ASCII to represent characters such as the alphabet (lower and upper case), punctuation and numbers, along with a few other special characters. This used 8 bits per character with a specific mapping from binary representation to character. For example, the character 'a' is represented by the number 97 or the binary representation 01100001. As the use of computers has grown, we have needed more and more characters. Most encoding schemes now use UTF-8 (Unicode Transformation Format) which allows for variable length encoding. Most text using the English

language will revert to using standard ASCII encoding, but less common characters may use multiple bytes to represent a single character. This allows for a great deal of compression instead of extending each character to use 2 or even 4 bytes. UTF-8 is capable of encoding all 1,112,064 possible valid character codes in the Unicode standard, including emojis!

A string is just a collection of characters, essentially like a special type of array. Due to UTF-8 encoding, it is often difficult to index into a string the same way one would index an array, since each character may use a variable number of bytes. Fortunately, the low level details are usually abstracted away from the programmer.

### Floating Point Numbers

Floating point numbers are necessary to store real numbers, including those with decimal values. While integer representations are not capable of storing decimal values, a floating point number can handle this. Julia, like most other programming languages, uses the IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008). This is made up of 3 parts, similar to standard form notation

1. The **sign** of the mantissa. The leading bit of the number represents the sign, such that 0 is positive and 1 is negative.
2. The **exponent**. The next set of bits (8 for a single-precision (32 bit) float) represents the size of the exponent. This includes positive and negative values.
3. The **mantissa**. This represents the significant digits of the number, which is usually normalised. This has columns to represent values like a half, a quarter, an eighth etc.

Overall, the number is calculated as  $x = s \times m \times 2^e$ , where  $s = \pm 1$  is the sign,  $m$  is the mantissa and  $e$  is the exponent. This is core to scientific computing as this severely limits the accuracy of calculations using floating point values, and we need to account for floating point behaviour. Note that we cannot represent every real number, but only approximate it. For example, look at the result of the following calculation:

```
julia> 0.1+0.2
0.30000000000000004
```

This is obviously wrong. This is called a **floating-point math error**. The topic of issues with floating point maths is very deep and nuanced, and we will discuss this throughout the module. For simplicity, we should recognise that multiplication is quite accurate, and even quite fast, whereas addition can lead to inaccuracies if the values being added are of significantly different magnitudes.

Note that floating point numbers can cover a vast range of magnitudes. For example, a 32 bit floating point number can represent values from  $1.1754944 \times 10^{-38}$  to  $3.4028235 \times 10^{38}$ , whereas a 64 bit floating number can go from  $2.2250738585072014 \times 10^{-308}$  to  $1.7976931348623157 \times 10^{308}$ . On computers, you will often see numbers written like  $2.2e12$ , one can interpret the  $e$  as “times 10 to the ...”, so the number becomes  $2.2 \times 10^{12}$ .

### 3.2.6 Memory Management

All programming languages need some mechanism to handle memory, as it is a finite resource. There is a vast array of approaches to handle memory during the execution of a program.

Older languages like C/C++ tend towards having manual memory management - when you are done with using some memory, you must free it so that it can be reused later. Failure to handle this properly causes **memory leaks** which can fill up the computer’s memory and eventually cause the program, or the computer, to crash. Manual freeing of memory is a technique that can be used for optimisation, but is not as important as discussing the more common approach taken by languages such as C#, Java or Python: **Garbage Collection** (GC).

Garbage collection works by having a separate process tag the parts of memory that are no longer being used and assign them to be freed so that that memory can be reused. This approach makes programming a lot easier for the developer and helps to make programs more safe by virtually eliminating memory leak bugs<sup>17</sup>. There are lots of different ways in which garbage collection is implemented in all of the different languages. As Julia is the focus of this book, we will give a brief overview of the GC strategy:

- Objects that require GC management are “tagged” with some additional information for marking whether they need to be collected, and their “age”.
- A sweep is conducted through the set of managed objects to see which can be freed, marking them for collection.

<sup>17</sup> These are still possible in some languages even with a GC.

- A heuristic is used to tune the amount of memory collected by the GC with the aim of balancing the amount of memory allocated, and the amount collected. The interval of time until the next collection is adjusted based on the amount collected.

For this process to be safe and predictable, the memory must be frozen in place to avoid errors. When the GC runs, all other execution is paused to ensure the memory does not change during the GC sweep. As you can imagine, this has a severe and adverse effect on the performance of your program if it spends a lot of time trying to reclaim memory. If you are trying to allocate more memory, once the program is almost full, this will cause a lot of what is known as **GC Pressure**. This can cause a tremendous amount of slowdown in your program. Some GC strategies cannot deal with a high amount of GC pressure, and tend to fail, returning *out-of-memory errors*<sup>18</sup>.

Obviously, having a GC can be a huge advantage, as it outsources part of the programmers responsibilities. However, there are significant performance implications of using this feature, and often, it is best to write your performance intensive code without relying on the GC at all. This will be spoken about later in the book, when we dive into practical optimisation strategies.

<sup>18</sup> Later, we will learn to use *CUDA.jl*, which implements its own custom GC for memory on the GPU. This GC has huge issues with GC pressure and can fail to free memory fast enough, causing crashes in your program.

### 3.2.7 The Stack and the Heap

During the previous section, we spoke about “allocating” memory. What does this mean? If we create a variable in our code, we must *allocate* some memory to store the data for that variable. For example, take the following code:

```
x = 5
x = x + 1
```

On the first line, we create a variable with the label<sup>19</sup> `x` and then store the integer value of 5 here. There are a few things going on here. First of all, the syntax of Julia abstracts away one detail, how did the program know how much memory to allocate for the variable, as this happens first? Implicitly, we can infer the type of the variable 5, and we know this is an integer (by default 64 bits), and so we know how much memory to reserve for this variable. Since the variable `x` is already assigned, on the next line, it is completely valid to write `x=x+1`, since we are saying: *Fetch the value from the memory which the variable `x` points to, add the constant 1 to this value and then write the result back into the memory referenced by the*

<sup>19</sup> Also known as an identifier.

*variable x*. During this entire process, the source code made it clear exactly how much memory was needed at each point, there were no guesses that need to be made.

Take another example, take reading in a file from the disk:

```
words = readlines("myfile.txt")
```

The computer has no way of knowing the contents of the file before execution. Therefore, it does not know how much memory needs to be allocated ahead of time.

These two different cases are handled by concepts known as the **stack** and the **heap**. You may have heard of a stack before, as it is a well known data structure<sup>20</sup>. A stack is a convenient way to organise information (such as the data in your variables) by stacking them on top of each other. Adding an item to the stack is called a *push* and taking an item off the top is called a *pop*<sup>21</sup>. A nice feature of the stack is that the information that is no longer needed can be *popped* from the top, without the need to free up the memory. This amounts to simply changing a pointer which points to the top of the stack, without having to clear the memory used above. However, in order for this to be efficient, there are some restrictions on what sort of objects can live on the stack. The most important restriction is that you need to know the size of an object at compile time, or else it cannot live on the stack. Additionally, there are some size restrictions on the stack as well. Objects that are too large are often better left out of the stack as well. As we have already seen with our example above, there are clearly common cases when knowing the size an object ahead of time is not possible. For this reason, programs also tend to use a **heap**.

A *heap* is a section of memory that can be used to store objects of variable sizes, such as arrays. It is usually a lot larger than the stack in size. At the beginning of a program's lifecycle the heap is usually empty with no memory around. When a request to store information on the heap comes, the program must search through the memory to find a space large enough for the object to fit, which is not already being used by another object. This is the process of "allocation", finding a section of the heap memory large enough for the object to be stored. Once this piece of memory is allocated, it is marked as unavailable for future allocations until it is freed. This is also what it means to "free" memory, we are just marking it as available for new allocations. Over time, this process tends towards fragmentation, with small blocks of free memory dispersed between allocation blocks. If there

<sup>20</sup> The "stack" in *Stack Overflow* comes from this very data structure, and a common error that people consider when incorrectly implementing a recursive function.

<sup>21</sup> Another common operation is called a *peek* which just looks at the top of the stack without removing it.

are no contiguous blocks of memory big enough for a new object, the program must de-fragment the heap to combine the smaller free chunks together and move existing allocated chunks around to make space. These issues have performance implications on your program. Sometimes you may need to think about this process in order to optimise your program.

One important thing to remember is that the stack and the heap (and even the machine instructions) are all stored in the same memory. This means the stack and the heap have the same access speed and one is not inherently faster than the other. However, we have gained a small insight into the additional management of the heap, and how using it can lead to additional work for the program, causing a slower performance. We have also gained an insight into what exactly we mean when we say we are *allocating* memory - reserving space on the heap for information. Even though storing data on the heap is technically also allocating memory, this usually has no performance implications, and we often do not refer to this process as allocation. For this reason, the term *allocation* should be taken to mean **heap** allocation, at least in the context of this book.

### Stack vs Heap example

Let's take a simple function:

```
function simple(x)
  x = x + 1
  x = x * x
  return x
end
```

Let's walk through the steps of evaluating this for an input of 2:

1. Push the input argument onto the stack (Stack: [2])
2. Push the constant 1 onto the stack (Stack: [2, 1])
3. Pop the top 2 items of the stack and add them together and push the result to the stack (Stack: [3])
4. Peek the value of the top of the stack and push a copy onto the stack (Stack: [3, 3])
5. Pop the top 2 items and multiply them, pushing the result back onto the stack (Stack: [9])
6. The return value is the last item on the stack.

Stacks are useful because evaluating nested functions becomes very straightforward. This does not need to be done manually, as it is all handled by the compiler. The aim of this example is to demonstrate that the very nature of the stack handles the “memory management” itself.

On the other hand, take another function:

```
function simple_heap(n::Int)
    v = rand(n)
    s = 0.0
    for val in v
        s = s + val
    end
    return s
end
```

This function takes in an integer variable `n` and creates an array `v` of random numbers using another function. For now, we do not need to focus on the specifics of this function call, but since `n` is not known at compile time, the array of random values **must** be stored on the heap. Let’s see this process from start to finish using an input of 4:

1. Push the value of the input argument onto the stack (Stack: [4])
2. Call the `rand` function with the item on the top of the stack. This creates a **pointer**<sup>22</sup> to the memory on the heap. (Stack: [ $p_v$ ])
3. Pop the stack and follow the pointer to the first value of the array in memory and push this to the stack. Increment the pointer by 4 bytes and push this back to the stack. (Stack: [ $v_0, p_v + 4$ ])
4. Pop the pointer from the stack and extract the data there and push this to the stack. Increment the pointer by another 4 bytes, and save in a CPU register. (Stack: [ $v_0, v_1$ ])
5. Pop the two top values and add them together and push the result to the top of the stack.
6. Push the saved pointer to the stack. (Stack: [ $v_0 + v_1, p_v + 8$ ])
7. Repeat the process until you reach the last element in the array, which will have the sum of all values in that variable. Do not push the last pointer variable onto the stack.

<sup>22</sup> A pointer is just a number which specifies a single location in memory, like an address. An array pointer points to the start of the array only.



8. The pointer  $p_v$  is no longer referenced in any variables and therefore can be marked for collection by a future GC sweep.
9. The final item in the stack is there return value of the function.

Note that this function has the side effect of leaving memory allocated. In a language like C, this is usually when you would manually free the memory allocated here. The majority of algorithms can be rewritten to avoid heap allocations or if this is not possible, to only allocate memory once and then reuse this memory. This minimises the number of heap allocated objects and relieves pressure on the GC. This technique is known as preallocation and will be covered in more detail later in the book.



## 4 *Julia Fundamentals*

There is no doubt that many people taking this course will ask the question - *why are we bothering to learn Julia?* Before we dive into the fundamentals of this new language, we will take some time to answer this question and, hopefully, convince you that learning this new language is worthwhile.

### 4.1 Why Julia?

Learning a new language can often be daunting, especially for those who are only just starting to learn programming. However, being able to quickly pick up a new language is an essential skill for any aspiring professional who writes code on a daily basis. Once you have experienced developing code in a few languages, you will realise that most languages share a common heritage, and transitioning between languages is relatively straightforward. While every language has its own quirks and technicalities, often it takes less than an hour to begin writing simple code, with the assistance of your favourite search engine of course. Julia, in particular, was designed to be easy to understand and read and especially easy to write.

In the most popular modern languages used in research (e.g. Python, MATLAB, C/C++, C#, Java, JavaScript... etc), there are common patterns that are shared, representing common operations and ideas with only minor differences in syntax. One can think of syntax as a mix of the *grammar* and *vocabulary* of a particular language. While some languages have notoriously high learning curves, one should focus on the logic of a program, and not worry as much about the syntax. Many research papers present their algorithms in the form of pseudocode, which does not represent any actual programming language. Syntax is just an implementation detail which often distract from the logic of the algorithm. This

being said, to get something working, we have to address questions of syntax. Thankfully, writing very simple programs can often be achieved by being able to answer the following fundamental questions:

- How do you declare variables in language X?
- How do you write an if statement in language X?
- How do you write a for loop in language X?
- How do you write and call a function in language X?
- How do you write an executable program/script in language X?

These are often the fundamentals which you will want to pick up first when writing code in a new language. Becoming proficient in the language will come in time, through practice and exposure. Once you have put effort into learning a second language, each additional language you learn takes less time and effort. This allows you to be very flexible and have more options when choosing a tool for each specific problem<sup>1</sup>

Before we specifically talk about using Julia, we should first consider what options already exist now, some of which you may be using now.

<sup>1</sup> For example, even though MATLAB allows you to build GUIs, this is often better left to more established languages (and their frameworks) like C#, Java or JavaScript.

#### 4.1.1 Caveats for Language Comparisons

We should take pains to differentiate a language (and its syntax) from the implementation (what we will call the toolchain) that you use. Python, in particular, has many different variants (*PyPy*, *CPython*, *IronPython* to name a few), each with their own particular capabilities. While differentiating the language (syntax & features) from the implementation is an important caveat, we can simplify our discussions of comparisons to the most common toolchain. In particular, we will be referencing the *CPython* implementation of Python here, as it is the most common. When we discuss other languages like C/C++, we will assume one is using the appropriate toolchain compilers for the job, whether it be vendor specific (such as the Intel compilers) or a general compiler such as *Clang*. In Julia, there is really only one implementation, using LLVM as the compiler, and so when we refer to Julia, we do not mean only the language, but the entire toolchain, including the compiler.

Further discussions of *languages* are assumed to include their most common toolchain, to make easy and general comparisons between languages.

### 4.1.2 Python

The first language, which is rapidly spreading throughout industry and academia is Python. Python is an incredible tool, and certainly an asset if used correctly, however, it has some major downsides which make it unsuitable for writing highly optimised code.

Native Python code can be extremely *slow*<sup>2</sup>.

This is the first major downside of the language. Executing native Python code is slow, being any from 10 – 1000 times slower than most other languages. The way that the language gets around this problem is by using packages written in other languages. For example, most of *numpy* is actually written in C, but uses a Python wrapper<sup>3</sup>. This is because C code can be *compiled* into code that runs quickly on the machine you are using. The speed-up from outsourcing the processing only comes when you can batch enough processing to do in the other language that the overhead of switching between the languages is negligible<sup>4</sup>. The massive success of Python for scientific computing comes down to the fact that many necessary calculations can be efficiently batched so that Python need only act as the glue, connecting various functions actually written in C.

However, what happens when you do not have the necessary functionality already implemented in a fast Python package like *numpy*? There are a few choices you have, such as using *Just-In-Time* compilation with *numba* or using *jax* at the cost of learning a whole new style of coding. You can even learn C and create your own package. However, these approaches are not perfect solutions and require a significant time investment to get working. If, after finishing this course, you wish to continue using Python, transferring the skills learn here can be done when using the aforementioned tools to gain high performance.

The main reason why we have chosen Julia over Python (and the tools like *jax* and *numba*) is because Python was not designed to be a high performance language. It was designed in the days of single core processors, and when programmers would turn to languages like C or Fortran if they really wanted to write fast code. It is only a somewhat recent development that Python has surged in popularity in the numerical computing sphere, and so the language has ended up being used in a way it was not purpose designed for. Where this is most clearly apparent is the existence of the Global Interpreter Lock (GIL)<sup>5</sup>. This is a design choice made to make sure that the code written is “thread-safe”. We will learn later exactly what this means, but the main takeaway is that the programmer is

<sup>2</sup> We are referring to the CPython implementation.

<sup>3</sup> A wrapper is a piece of code that acts as an interface between the user of the wrapper and some underlying, more abstract code, such as code written in a different language.

<sup>4</sup> Interestingly, this is where the optimisation technique popular in Python with *numpy* and MATLAB is to *vectorise* your calculations, instead of using a for loop. This just means that you execute the for loop in C instead of in the slower language, with a few other optimisations.

<sup>5</sup> Some Python implementations do not have a GIL, but the main one, CPython, does.

severely limited when trying to scale their code to use multiple processors, and is forced into using multiprocessing over multithreading<sup>6</sup>, or use a package which uses a different language to allow for parallel processing. As progress in increasing the single core performance of processors has been drastically slowed in the last decade, one is almost forced into writing code that scales across multiple cores to get increased performance on modern day problems. While it is certainly possible to write parallel code in Python, it is a much less attractive choice.

<sup>6</sup> Each of these techniques are covered in detail in their own chapters later in this book.

### 4.1.3 MATLAB

MATLAB is both a language and a set of libraries and frameworks that provide the majority of the building blocks needed for modern scientific computing. As the name suggests, one of the early primary focuses of MATLAB was handling matrix manipulations - i.e. providing fast linear algebra routines. There are many additional frameworks such as *Simulink* used for model based design and other engineering applications.

MATLAB also provides great plotting capabilities and is somewhat of an industry standard for scientific computing. Until recently, all the numerical computing taught at the University of Nottingham was in this MATLAB, being usurped by Python in last few years.

While we do not go into much depth on the language, there are a few factors which make it less attractive for use in this module. The most important of these is the licence, which is proprietary. While most universities provide a licence to their students, it provides too high a barrier for the aims of this module. Additionally, the facilities for fine-grained optimisation and parallel/GPU computing are very limited, especially when compared with Julia.

It should be noted that MATLAB had a huge influence on Julia, and if you are familiar with the syntax of MATLAB, you will recognise much overlap when we learn the Julia syntax.

### 4.1.4 C/C++ & Fortran

Traditionally, when people have realised that Python/MATLAB was too slow for their needs, developers turned to languages like C/C++ or Fortran and their compilers. However, as many people will tell you, these languages have an incredibly high learning curve. They take a long time to develop the software and requires

highly skilled developers to produce highly performance code. The main reason for avoiding it in this course, is due to the difficulty of learning these languages in a short time frame, the learning curve is far too steep<sup>7</sup>.

Hopefully these sections should hint at a common problem in numerical computing, encapsulating what is known as the “**two-language problem**”. While, for example, Python is easy to write and develop your code in, it is often far too slow to be useful at scale. This often leads to researchers and developers experimenting with their code in Python and use libraries like *numpy* to speed up common tasks such as linear algebra. Depending on the application, there is often a limit to how performant code written in Python can be, even when using highly optimising libraries like *numpy*. This often results in researchers having to rewrite the performance critical parts of their code in C/C++ or Fortran. Clearly, this is a suboptimal setup. It should be possible to have a language that is flexible and easy to learn and write, while also delivering the same performance as languages like C/C++ and Fortran. The founders of the Julia language believed this to be the case, and began about bringing such a language into existence.

<sup>7</sup> Also, I do not know these languages well enough to teach them.

#### 4.1.5 Julia

Julia is a relatively recent language<sup>8</sup>, whose user-base has been growing exponentially over the past few years. It was specifically designed to be an easy-to-use, dynamic language, suitable for fast-paced research and development, while also delivering performance comparable to traditional, statically-typed, languages such as C/C++. There are numerous ways in which Julia achieves this, which will be explored throughout our examples in the book. Most importantly, this is a modern language, purpose designed for high performance scientific computing.

In technical terms, Julia is a “high-level, high performance, dynamic and Just-in-Time compiled language”. It has a multi-paradigm approach whose main features are multiple dispatch and procedural, functional, metaprogramming and multistaged patterns. Since many readers will be unfamiliar with most of these words, unless they come from a Computer Science background, we will break down what each of these mean.

The first term, “high-level”, means that Julia has human-readable code. Specifically, this means that the language requires some tools to convert the high level source code into low level machine code before it can be actually executed. When

<sup>8</sup> Recent when compared with languages discussed previously. Julia has been around for over a *decade* now.

we talk about “high” and “low” level code, we are referring to how close the code is to machine code, “low” being the closest and “high” being the furthest away.

**High performance** is self-explanatory, however, **dynamic** is more nuanced. Dynamic in the simplest sense, refers to the way that the way typing is handled in the language. In Julia, one can reassign the type of a variable: for example, setting a variable which stored an integer, to then storing a string. This is traditionally disallowed in statically-typed languages like C/C++, which require the type of variables to stay fixed (or static) during their lifetime. Additionally, statically typed languages usually require that the types of all variables be known at compile time. Python is also a dynamic language, which is why you almost never see the types of a variable in the source code, because they can be reassigned to be anything.

**Just-in-Time** refers to when the code is actually compiled. Julia compiles the code when it is actually needed, which means the compiler can have a lot more information about the code being run (such as the types of the variables being passed into a function) and therefore can often perform more optimisations. This also allows for code to be dynamically generated on the fly, which enables metaprogramming.

**Metaprogramming** is when you have code which can write more code. In Julia, this takes the form of writing **macros**, which allow you to write easy and flexible code.

One can read more about the difference between procedural and functional paradigms in other sources, but both have their advantages and disadvantages and so having a language which allows for a mix of approaches gives one the best of both worlds.

Finally, the poster child of Julia is **multiple-dispatch**, one of the core differentiators against other languages. *dispatch* refers to choosing the specific implementation of a function <sup>9</sup>. Multiple-dispatch is a feature which allows a function or method to be dynamically dispatched to based on the *run-time* (dynamic) type of more than one of its arguments. It is similar to the concept of *polymorphism* in other languages, but a generalisation of it. In a nutshell, multiple-dispatch routes a call to a function dynamically based on the characteristics of every input argument of the function, allowing specialisation on every argument type. It is a topic which will be covered in far more detail later on in this chapter, but it is essential to understand to be able to understand code written in Julia, and it also holds the key to the two-language problem.

<sup>9</sup> Often called a method in the case of Julia.



## 4.2 Basic Syntax

In order to read and write Julia, we first need to understand the syntax of the language. For a comprehensive and well written guide to the design and syntax of Julia, it is highly recommended that you read through the official Julia documentation<sup>10</sup>. The official documentation is extremely high quality and comprehensive. If you are serious about learning Julia, it is worth sitting down and reading/skimming through the entire manual<sup>11</sup>. To make the transition from another language very easy, the documentation also includes a section on noteworthy differences from other languages<sup>12</sup>, which can help bootstrap your learning.

While the documentation linked above is definitely the recommended place to learn the syntax of the language, we have included a distilled version of the key points in the manual to help you get started.

<sup>10</sup> <https://docs.julialang.org/>

<sup>11</sup> <https://docs.julialang.org/en/v1/manual/getting-started/>

<sup>12</sup> <https://docs.julialang.org/en/v1/manual/noteworthy-differences/>

### 4.2.1 Declaring variables

As in most languages, declaring a variable is simple:

```
a = 0
```

This syntax creates a new variable, `a`, and assigns the value of `0` to it. Just like in Python, one can reassign a variable to a different type:

```
a = "Hello, World!"
```

This is perfectly valid, but as we will discuss later, it is often a bad idea to re-assign types.

Since this is a numerical computing module, we need to discuss how to declare an array. Arrays are usually built using functions. Just like in Python's *numpy*, we usually use the `zeros`<sup>13</sup> function to create a new array:

```
arr = zeros(10, 10)
```

This creates a 10 by 10 array. By default, this function creates the array with the type `Float64` (on a 64-bit system). We can specifically tell the function which type we want the elements of the array to be, with:

```
arr = zeros{Int64}(10, 10)
```

This makes sure each element is a 64-bit integer. As an additional note, arrays use one-based indexing<sup>14</sup> and to access the first element of an array one uses:

<sup>13</sup> There are many, many, ways of initialising an array, which will be explored in examples later in this book.

<sup>14</sup> As Julia is a flexible language, one can use array types with 0-based indexing. As such, it is best to write your functions to be independent of one-based vs zero-based arrays, and so this point should not affect the majority of the code you write.

```
arr[1]
```

Linear indexing like this will work, even for multidimensional arrays, since all arrays are stored as a contiguous 1D vector in memory. Having multiple dimensions is just syntax sugar. For our  $10 \times 10$  example, the syntax `arr[i, j]` is exactly equivalent to `arr[i+j*10]`, as arrays in Julia are stored in *column-major* order. This will be covered in more detail later.

### 4.2.2 If statements

An **if** statement controls the flow of your program and chooses which code will be executed. This is sometimes referred to as a “conditional”. In Julia the syntax is:

```
if statement
    # execute code here if statement is evaluated as true
end
```

Here, we see the first major bit of difference between Julia and Python. Julia, like MATLAB, requires the use of the **end** keyword to close all statements. A general rule of thumb is that most statements that cause an increase in indentation, also require an **end** keyword.

We can extend this to have a control flow with many options with an **elseif** keyword. This comes in the form:

```
if statement1
    # code here for statement1 block
elseif statement2
    # code here for statement2 block
else
    # code if neither statement1 or statement2 is true
end
```

What is special about this syntax is that it can *short-circuit*, which means that if **statement1** is evaluated to be true, then the rest of the *if-elseif-else* block is not evaluated. The **else** keyword will contain the code that is executed when all other conditions in the block are evaluated as false.

One can combine multiple statements together on a single line by using either an “and” operator or an “or” operator, in Julia these are written as `&&` and `||` respectively. These are somewhat special and are lazy in their evaluation. Take the following examples:

```

if statement1 && statement2
    # code if statement1 and statement2 are both true
end
if statement1 || statement2
    # code if either statement1 or statement2 is true
end

```

In the “and” example, if `statement1` is false, then `statement2` is not even evaluated since there is no point, as the combined statement will definitely be false since both arguments need to be true in order to return true. In the “or” example, if `statement1` is evaluated as true, then `statement2` need not be evaluated, since the combined statement is known to be true. This optimisation leads to a form of syntax common throughout a lot of Julia source code in the wild, known again as *short-circuiting*, which removes the need for explicit if statements in the code. The syntax for this is:

```
statement1 && conditionalcode()
```

If `statement1` is true, then `conditionalcode()` is evaluated, otherwise it is not even evaluated. This syntax is equivalent to the following:

```

if statement1
    conditionalcode()
end

```

A similar thing can be done with the `||` operator, but this is far less common.

This section would be slightly incomplete if we did not mention the ternary operator. In order to describe this, let’s take a look at a common example:

```

# var is an existing variable
if statement1
    var = 1
else
    var = 0
end

```

This type of control flow is so common, that many languages implement syntax for a ternary operator, which allows one to write this type of code on a single line. The equivalent code using the ternary operator is this:

```
var = statement1 ? 1 : 0
```

This is far more concise. The operator is `?`, and evaluates the statement to the left for truth, and if true, returns the value to the right of the operator. If false, then the statement returns the final value, given after the colon<sup>15</sup>. This syntax also has an advantage in that the variable `var` does not need to be declared before the `if` statement to ensure that the value survives beyond the scope of the `if` statement.

Finally, we need to talk about the “not” operator which negates a statement. This is given by `!`, in Julia. Take the following:

```
var = !(statement1) ? 0 : 1
```

This code is equivalent to the ternary example, since the two outputs have switched place.

### 4.2.3 Loops

A for loop in Julia is very similar to Python and MATLAB, the syntax is:

```
for state in iterator
    print(state)
end
```

Like before, one needs to use an `end` keyword to close out the for loop. What is necessary to discuss is what form the `iterator` can take. In Julia, one can write a very generic iterator (see the ‘Iterators’ section in the Julia documentation<sup>16</sup>), but there are a collection of useful iterators already implemented in the base library<sup>17</sup>. Before we talk about these utilities, we should cover the most basic iterator, the array iterator. If we have an array, this acts as an iterator by default, as seen below:

```
arr = [3,2,1]
for a in arr
    println(a)
end
```

What is more interesting is the following use case, which is very common, especially when doing a parameter search. Let’s say we want to have a nested for loop, which prints out every combination of elements from two arrays. Take the following:

<sup>15</sup> Since colons are important to this syntax, if you are using a range in one of the return values, you must wrap the value in brackets, to ensure it is evaluated first.

<sup>16</sup> <https://docs.julialang.org/en/v1/manual/interfaces/>

<sup>17</sup> <https://docs.julialang.org/en/v1/base/iterators/>

```

as = [2,1]
bs = [4,5]
for a in as
    for b in bs
        println("a: ", a, ", b:", b)
    end
end
end

```

This syntax can get difficult to read very quickly, and what if the number of arrays you want to iterate through is variable and can change? Clearly this requires a smarter iterator. This is where the base libraries "Iterators" package comes in. This defines the `product` function, which allows us to collapse a nested for loop into a single loop:

```

using Base.Iterators

as = [2, 1]
bs = [4, 5]
for (a,b) in product(as, bs)
    println("a: ", a, ", b:", b)
end

```

This code is equivalent to before, but is more readable and concise. The `product` iterator works with any amount of inputs, so one should take the opportunity to experiment with it in your console to understand how it works.

Another common situation is when you want to pair two arrays with the same number of elements. Usually, the situation is as follows:

```

as = [1,2,3]
bs = [4,5,6]
s = 0
for i in eachindex(as, bs)
    s += as[i]*bs[i]
end

```

This method requires manual indexing of the arrays. The `eachindex` method is very powerful and makes sure that you can jointly index the arrays `as` and `bs` properly and raises an error if the dimensions of the arrays do not match. The traditional approach is to manually index the arrays, but this is discouraged as the code will not work with arrays with non-standard indexing<sup>18</sup>. Instead of indexing the arrays, one can use the `zip` method. This works the same way as it does in Python, and pairs each element from across multiple lists (arrays) as follows:

<sup>18</sup> While standard arrays in Julia use 1-based indexing like MATLAB, it is entirely possible to define one's own array type that uses arbitrary offset array indexing, so it is recommended to write your code generically to avoid preferring one indexing scheme over another.

```

as = [1,2,3]
bs = [4,5,6]
s = 0
for (a, b) in zip(as, bs)
    s += a*b
end

```

This gives you the exact same code, but without needing to explicitly index either of the arrays. But what if you still want the index? This is where the `enumerate` method comes in. This, again, works the same way as in Python, by providing the index along with the original state. An example is shown below:

```

as = [6.0,3.0,12.0]
for (i, a) in enumerate(as)
    println("i: ", i, ", a:", a)
end

```

It is worth exploring the documentation to see the other methods that are built into the Julia base library.

#### 4.2.4 Writing Functions

A function can be written in many ways in Julia. The first way is writing a function exactly how you would write down a mathematical function. Take for example:

```

julia> f(x)=2x^2+5x-10
f (generic function with 1 method)
julia> f(1.0)
-3.0

```

This function syntax is great for functions that can be expressed in a single line. If your function needs to be spread across multiple lines, you can use the `function` and `end` keywords to define the scope of your function:

```

julia> function f_multiline(x)
    a = 2*x^2
    b = 5x
    c = -10
    return a + b + c
end
f_multiline (generic function with 1 method)
julia> f_multiline(1.0)
-3.0

```

Notice that the syntax starts with writing the `function` keyword, followed by the name of the function, `f_multiline`, and then some parenthesis. Inside the parenthesis we list the arguments that the function takes, these arguments are named and should be separated by commas. A special type of argument, called a keyword argument, can be written at the end of the set of normal arguments, but separated by a `;`, like below:

```
julia> function f_manyargs(x, y, z; kwarg1, kwarg2)
    return (x*y+z) * kwarg1 + kwarg2
end
f_manyargs (generic function with 1 method)
julia> f_manyargs(1.0, 2.0, 3.0; kwarg1=5.0, kwarg2=-2.0)
23.0
```

What is important to remember when writing functions is *scope*. Scope defines the “lifetime” of the variables define within the current scope. As a general rule of thumb, one can infer the scope via the current level of indentation. The parameters declared in the function signature, only survive until the end of the function, along with any local variables defined within. One can extend the lifetime by mutating the state of the parameters passed in by reference, or by assigning them to an outer scope by returning the parameters from the function.

What is special about Julia functions, is that you need not specify the `return` keyword as one needs to in other languages. Take the following example:

```
function mutate_argument!(parameter)
    parameter.state = 0
    nothing
end
```

This function mutates the variable `parameter`, and sets the `state` field to 0. The final line of the function is just written as `nothing` and this will be returned when the function is called. Notice the use of the `!` at the end of the function name, this is a convention to indicate the function will mutate the input argument (usually only the first one).

### 4.3 Advanced Syntax

While it is possible to write even complex programs with the syntax discussed previously, it can be very beneficial to invest in learning some more advanced syntax to help write more general and effective code.

### 4.3.1 Broadcasting

It is very common to want to apply an arbitrary operation to each element in an array. This is known as a *map* operation, since we want to map an input to some output. Take the following example:

```
numbers = rand(100)
results = similar(numbers)
for i = 1:length(numbers)
    results[i] = sin(numbers[i])*numbers[i]
end
```

Here, we want to apply the same operation to each element of `numbers` and store the result in another array called `results`. This operation can be simplified in Julia to the following syntax:

```
numbers = rand(100)
results = sin.(numbers) .* numbers
```

This for loop can be expressed in a single line, and what is better, one does not need to worry about preallocating an array and indexing the right location. Even more impressive, is that Julia can fuse the broadcast operations together, which allows the compiler to group these operations together to avoid having to store intermediate results, which can use up a lot of memory<sup>19</sup>.

The way to apply broadcasting is to put a `“.”` before an operator, even an arbitrary function. There are some macros to help as well, such as `@.`, which will take a normal statement which works for a scalar and add in the dot operator to make the statement a broadcast instead.

<sup>19</sup>This is one downside of using *numpy* as this often allocates a lot of intermediate arrays.

### 4.3.2 Structs

It is often necessary to define your own types which combine more primitive types together. This allows for the code to be readable. Take the example of a complex number. Pretend for the moment that Julia does not provide a complex number, let us see how we can write this part:

```
struct Complex
    real
    imag
end
```



One can then instantiate an instance of this object just by writing `Complex(5.0, -2.0im)`. One should note that `Complex`

An important point in Julia is that structs are *immutable* by default. This means that after the instantiation, one cannot edit the components. One might think that this makes structs useless, but operations that need to alter the inner values, can instead return a new struct with the new values. This has a number of advantages, but that is beyond the scope of this section. If you want to be able to mutate the inner variables of a struct, you can declare the struct to be mutable, with the `mutable` keyword:

```
mutable struct Complex
    real
    imag
end
```

One final point to mention here is that this struct has no type declarations, so the real and imaginary parts can be any types. This is probably not what you want for this, especially since you likely want the real and imaginary part to be of the same underlying type. We can enforce this by specifying exact types:

```
struct Complex
    real::Float64
    imag::Float64
end
```

Once you learn more about Julia, you will see that this is not a good way to write a struct. Mainly because this type is only useful if you want the inner values to be 64-bit floating point numbers. What if you wanted these to be integers instead? Or even 32-bit floating point numbers? This would require more types. However, a better way to solve this problem is by using generic parameters. A generic parameter acts as a placeholder for an input type, which is specified when one instantiates an instance of the struct. The syntax for this is written as:

```
struct Complex{T}
    real::T
    imag::T
end
```

If we create a variable `Complex(1, 2)`, it will now be a different type to `Complex(1.0, 2.0)`, but this enforces that both variables must be of the same type. This generic parameter specification adds some performance to our code that uses these complex numbers, since the type of the inner elements is now mixed with the type of the struct. In our example, the first variable will be of type `Complex{Int64}` and the second will be of type `Complex{Float64}`. This means that if we pass these objects into a function, the Just-in-Time compiler will know about the type of the variables inside the parameter, and therefore, is able to write specialized machine code, without needing to “box” the variables. Boxing will be described more at a later time, when we talk about performance, but one can think of it as having to check the type of a variable every time it is used, since the type specifies which code is executed when the program runs.

As a final example, we can restrict the generic type of a parameter using the `<:` operator which is the inheritance operator which evaluates whether a type is equal to, or a subtype of another type. It is also used to restrict the types that are allowed. If we only want `Real` numbers to be allowed as types in our complex number, we can write:

```
struct Complex{T<:Real}
    real::T
    imag::T
end
```

These restrictions are not necessary, but they do restrict how your code can be used. This often allows for errors to be thrown at compile time when you are using types that you do not expect.

### 4.3.3 List Comprehensions

A feature which Julia inherited from Python is the list comprehension. This is a very powerful syntax which makes defining a map and/or filter operation over an array. For example, if we would like to create an array in which each element’s value is equal to its index, then traditionally we would need to write:

```
n = 10
numbers = zeros(n)
for i = 1:n
    number[i] = i
end
```

Instead, we can write a list comprehension, which is much shorter:

```
numbers = [i for i = 1:10]
```

To break down the syntax here, we use the square brackets to specify an array (or a list) to be created. Inside we can specify a loop via the `for i = 1:10`, and then on the left-hand side of the `for` statement and after the opening square bracket, one states the value of the element, dependent on the state of the loop, `i`.

One can also specify a filter condition, based on the loop by adding an `if` statement to the end of the comprehension:

```
numbers = [i for i = 1:10 if i>5]
```

This will create a shorter list, only when the index is greater than 5.

The most important thing to remember about list comprehensions is that they are just syntactic sugar for writing a very common type of operation. They do not have the most readable syntax, and it might be beneficial to write out your code, especially if the equivalent list comprehension is complex. One can equivalently write the logic of a list comprehension without the use of one.

#### 4.3.4 Lambda Functions

Lambda functions are also known in other languages as *anonymous* functions. These are simply functions without an explicit name. The syntax uses the `->` notation, which signifies mapping from some input parameters to an output. Take our previous example:

```
numbers = filter(i->i>5, collect(1:10))
```

This syntax creates a lambda function via the `i->i>5` syntax and then passes this function to the `filter` function which is built into Julia. This syntax can be read as “`i` is mapped to `i>5`”, which returns either `true` or `false`.

#### 4.3.5 Closures & Functors

Building on lambda functions, a closure is a function which *captures* a variable from the same scope as it was declared in. In Julia, and many other languages, you can write a function inside another function and return it as an object<sup>21</sup>. As an example, let’s take an arbitrary example:

<sup>21</sup> Functions are 1<sup>st</sup> class objects in Julia.

```

function add_n_fn(n)
    function _add_n(x)
        return x+n
    end
end
add_5 = add_n_fn(5)
add_5(10) == 15

```

Inside `_add_n`, we *capture* the local variable `n`, and use that value when `_add_n` is called in the future. This is a method used to simplify the arguments of a function call by capturing all the necessary inputs. However, this approach can cause huge performance issues, as the types of the captured variables cannot always be known. Additionally, there is some state associated with the function now, which may lead to hard-to-detect race conditions.

An alternative to closures is writing an explicit **functor**, which is a **struct** like object which acts like a function, with some state. Let's re-write the previous example using a functor:

```

struct AddNFn{T} <: Function
    n::T
end
function (fn::AddNFn)(x)
    return x + fn.n
end
add_5 = AddNFn(5)
add_5(10) == 15

```

Here, we say that the object `AddNFn` inherits from `Function`, and contains a generically typed variable `n`. We also define calling that method by specifying the type as the *function-name*, along with an identifier for the type so that we can access its properties inside the function. This can be especially useful for writing re-useable code, which is also very performant.

#### 4.3.6 Modules

From what you have seen so far, Julia does not have any classes, or namespaces and all functions are defined in the global scope. This seems very unmanageable? How can Julia scale to large projects if the programmers don't have the ability to segment their code?

The answer to this is *Modules*! Julia has a very simple syntax to encapsulate a series of functions and variables inside a barrier, which can be used to divide up sections of a code base, and allow massive amounts of code reuse without clashing with other peoples code.

This section will not go over all the details, and if you are pursuing a larger project, you are highly encouraged to read the documentation about how modules work in Julia.

The basic syntax is as follows:

```
module MyModule

function myfunc(x)
    println(x)
    nothing
end

end # end MyModule
```

If one executes this in the Julia REPL, or includes the file in which it is defined, they will have to specify the module name to access the `myfunc` function - `MyModule.myfunc("Hello, World")`.

One can even specify a module, within a module, known as a *sub-module*:

```
module MyModule

module SubA
    function subafunc(x)
        x*2
    end
end

function myfunc(x)
    println(x)
    nothing
end

end # end MyModule
```

In order to access this new function, you simply write `MyModule.SubA.subafunc(x)`. Hopefully, you can see that this approach may lead you to write very verbose code. The writers of Julia do not want this to happen. For this reason they have supplied the `export` keyword, which lets a module expose some chosen functions

as its public API, and when someone uses the module, the exported items get put into the global (or current) scope.

We have seen this already, we have been using the `@btime` macro from `BenchmarkTools.jl`, and all we needed to do was include a `using BenchmarkTools` at the top of the file/REPL. An example of how this works is below:

```
module MyModule
  f(x) = x
  f2(x) = x*x
  f3(x) = x*x*x

  export f2, f3

  end # end MyModule
```

When users write `using MyModule` in their code, they will now have access to the `f2` and `f3` functions without having to specify the module like before. Note that we did not expose the `f` function, but one can still access this by specifying the module - `MyModule.f`.

## 4.4 How does Julia work?

Now that we know more about the syntax of Julia, it is very important to have an understanding of how the language works. This is especially important as there are a few key differences to other languages that you will have seen before.

### 4.4.1 Running a program

Julia is designed around a REPL<sup>22</sup> workflow. However, you can organise your code into scripts and execute them via the command line using `julia myscript.jl`, much like in Python. A REPL-based workflow allows you to keep the variables and function definitions alive, even after multiple code changes. This is very important in Julia, as it is Just-In-Time compiled, and you have to pay a cost on the first execution of a given function, or first import of a package. You do not want to have to wait for re-compilation of functions which have not changed on every iteration cycle. Packages like `Revise.jl` are frequently used to make this workflow extremely streamlined. You may have seen a similar workflow using MATLAB, or Python (with Spyder or Jupyter notebooks).

<sup>22</sup> Read-Evaluate-Print-Loop

One caveat that must be followed for beginners is that performance sensitive code should only be executed inside of a function and not in global scope. For example, if you want to sum up over an array (called `arr`) manually, you should not write a script with

```
s = zero(eltype(arr))
for a in arr
    s += a
end
```

You should put this code inside a function, and call the function:

```
function my_sum(arr)
    s = zero(eltype(arr))
    for a in arr
        s += a
    end
    return s
end
s = my_sum(arr)
```

Additionally, notice how we pass in the arguments to the functions, instead of using global variables? This is extremely important as global variables often have severe performance implications, and are often very bad programming practices.

In general, running a Julia program just means opening up a REPL and typing out some code. It is recommended to use the VS Code IDE to write out your scripts and use the inbuilt tooling to “send” your code to the REPL, or use something like *Revise.jl* to automatically update the REPL with your script every time it is saved.

Having a good workflow in Julia is crucial to having a good development experience. This is often a pain-point for new (and even advanced) users, but most of the issues have mitigations. However, there is no one workflow that works well for everyone, and there are frequent changes and improvements being made in the Julia ecosystem.

#### 4.4.2 Type System

The first thing to go over is the type system. You will have heard a lot about types in the first part of this book, and probably will have a much better understanding of what a type is. This section is written to give you a deeper understanding of types in general, and how the type system works in Julia.

The first thing to note in Julia, is that while the language is dynamic (i.e you can change the type of a variable during execution), the language still has a very strict and robust type system. The dynamic typing of Julia allows the programmers to be very flexible and fast when they write code, but the robust underlying type system allows programmers to optimise their programs if they respect the type system. The first question to ask is, how do types allow code to run faster?

This question has a very practical answer. In your computer, every piece of data has to be stored in binary format in memory. This means that every integer has to be represented in binary bits, usually with a fixed amount of bits per number. Integers are rather simple as the bit storage for an integer is usually equivalent to how you would write the integer in binary, except for negative numbers which use a *two's compliment*<sup>23</sup> format. Floating point numbers (real numbers) are usually representing in binary standard form. There are many choices on how to represent these numbers, but they are all usually based on the IEEE 754 standard<sup>24</sup>. Clearly, operating on a completely different set of bits for integers and floating points mean that one requires different operations (and literally different sets of hardware on your CPU) to efficiently process them. This means that at some level on your computer, it **has** to know what type a variable is, in order to know how to manipulate the bits. If the programming language is dynamic, and the code does not know what type a variable is, it **must** at some point, find out. This check is often very expensive. If one of your variables is boxed (i.e. the type information of the underlying data is unknown or obscured) then the computer must put in effort to “unbox” the variable and process it in the correct way.

If we move away from machine types for a second, we can use type information to also select the most efficient algorithm for a certain problem. Take a function such as `sum`:

```
function sum(arr)
    s = zero(eltype(arr))
    for x in arr
        s += x
    end
    return s
end
```

In this implementation, we have made sure that `s` is the same type as the elements of the input array to stop any type instability<sup>25</sup>. While this implementation is

<sup>23</sup> In two's compliment, the leading bit is used to represent the negative of its unsigned counterpart. For example, representing a number with 8 bits has a leading bit representing  $2^7$ , but we can change this to represent  $-2^7$  to be able to store negative numbers, which turns out to be a very efficient way of allowing negative numbers

<sup>24</sup> [https://en.wikipedia.org/wiki/IEEE\\_754](https://en.wikipedia.org/wiki/IEEE_754)

<sup>25</sup> Type instability means that the compiler cannot predict the types within a function, and is usually the most common cause of performance loss in Julia. One can detect type instability by using the `@code_warntype` macro.



very general, suppose for a minute that we have a sparse array with the following implementation:

```
struct SparseArray{T}
    elements::Vector{T}
    indices::Vector{Int}
end
```

Since this is a sparse array, we can implement a more efficient form of the `sum` function<sup>26</sup>:

```
function sum(arr::SparseArray{T}) where T
    s = zero(T)
    for x in arr.elements
        s += x
    end
    return s
end
```

<sup>26</sup>This implementation uses generic type parameters and captures them in the constant `T` which can be used in the source code of the function.

This will save on a lot of computation, as one does not need to loop through all the zeros.

Now we know why types are important, we should gain an understanding of how they work in Julia. We can start this by realising that there are two types of types: *concrete* and *abstract*. All variables in Julia have a concrete type to represent them. A concrete type has a specific representation of data. All machine types, such as `Float64` and `Int64`, are concrete types. Our example of the `SparseArray` is also a concrete type. An instance of an object in Julia is always a concrete type, as it has some definite memory layout. A concrete type is at the very bottom of a type-hierarchy, as a concrete type cannot be inherited from. Julia avoids most of the issues with inheritance by not allowing actual implemented types to be inherited from.

However, some aspects of inheritance are very useful, and help to reuse code throughout the code base. So Julia provides the `abstract` type. An abstract type is only a name, and holds no actual data. Abstract types exist only to define the behaviour of a type - not the implementation! Abstract types provide the bulk of the structure to a type-hierarchy, and allow for massive reuse of code, while providing structure and grouping similar types together.

One should remember that the type hierarchy and inheritance system is all about inheriting behaviour and not about inheriting implementation details. In practice, the type system is used to guide and direct multiple-dispatch to choose

the most specific method implementation. The power of the type system comes in **restricting** the types that are allowed in your method implementations. If you are familiar with a language like C#, an abstract type should remind you of an “interface”.

#### 4.4.3 *Number type hierarchy*

Let’s take a look at how the different number types are organised together in Julia using abstract types. To define an abstract type, we write the following:

```
abstract type Number end
```

Implicitly, this type inherits from the supertype `Any`, as all other types inherit from, and so we do not need to write this in.

Imagine we want to distinguish between real and complex numbers, we can define two abstract types for each different behaviour. However, these two types of numbers share a lot of behaviour and are both numbers, and hence they should inherit from the `Number` abstract type. We define this relationship using the operator `<:` which can be read as *is a subtype of*:

```
abstract type Real <: Number end
abstract type Complex <: Number end
```

We can also use `<:` as an operator to evaluate whether a type inherits from another type:

```
julia> Real <: Number
true
julia> Complex <: Number
true
julia> Complex <: Real
false
```

Note that Julia only has **single inheritance** and so each type can only have a single parent, but an abstract type can have many children. This is not as limiting as it can be in other languages, due to the multiple dispatch system described later in this chapter.

The type hierarchy ends at the concrete types which can inherit from an abstract type. A concrete type is either a composite type, known as a **struct**, or a **primitive** type. For example, if we have a `AbstractFloat` type, we can define a 32 bit integer using

```
primitive type Float32 <: AbstractFloat 32 end
```

where the second number 32 defines the number of bits required to store the type. It is not very common to declare one's own primitive types, but Julia gives you the option.

#### 4.4.4 Multiple-Dispatch

The choice of which method (function implementation) to execute when a function is applied to a set of arguments is called *dispatch*. Since Julia allows this choice to depend on all of the input arguments, we call this process multiple dispatch. Most of the languages you have likely encountered in the past have *single dispatch*, which means that the choice of implementation depends only on the first argument<sup>27</sup>. Some languages have function overloading (also called polymorphism), but multiple dispatch is a generalisation of this technique. In these languages, the functions are tightly coupled to their first argument. This does have many advantages, particularly when filtering the list of available methods for an object, but is an arbitrary distinction, which does not often make sense in mathematical code as the implementation usually depends on the types of all the input arguments<sup>28</sup>.

The Julia runtime selects which implementation of the function to execute, depending on the types of the input arguments of the function, and compiles code for the concrete types passed into the function. That is all it is, a way of deciding which implementation of a function to execute.

Take our example in the type section about the sum, with the sparse array:

```
function sum(arr::AbstractArray)
    s = zero(eltype(arr))
    for x in arr
        s += x
    end
    return s
end

function sum(arr::SparseArray{T}) where T
    s = zero(T)
    for x in arr.elements
        s += x
    end
end
```

<sup>27</sup> The first argument is usually the object itself, think of `self` in Python, where you find the function you wish to execute by using `self.function()`.

<sup>28</sup> Think of the addition operation - the function only makes sense when considering the input types of both arguments equally.

```

return s
end

```

Here, we have two implementations for a given function. In the first example, any variable that inherits from the `AbstractArray` abstract type should be able to be passed into the first sum function. This provides a fallback implementation, so that most code can be executed. However, we can specify a more *specific* type (a concrete type in this case, which is the most specific possible type), which allows for a more optimal implementation. If we were to call the function via `sum(array)`, Julia chooses a different implementation, depending on the actual type of `array`. The Julia runtime attempts to pick the most specific implementation possible, which matches the type of input. If no functions are found with a matching type implementation, then an error is thrown.

The rules are slightly more complicated when a function has many parameters, there are rules to decide on which function is the “most specific” in those cases<sup>29</sup>. In general, “more specific” equates to “further down the type hierarchy”. The most specific implementations specify concrete type arguments.

This method specialisation allows us to extend our code to more specific use cases without modifying the original source code. The only thing that changes is the **type** of the input arguments. One will find that an extremely large amount of packages through the Julia ecosystem are compatible with one another, despite having no “glue” code written to allow these packages to communicate. Additionally, it is incredibly easy to extend the functionality of a given package (or the Julia base libraries themselves) by defining your own types. This really captures the **O** from the **SOLID** design principles<sup>30</sup> which states that code should be “*Open for extension, but closed for modification*”.

## 4.5 Package Management

Almost all research nowadays will rely on using other people’s code, outside the base libraries provided by the language you are using. The code you use will often come in the form of packages (such as *numpy* in Python), which are fluid projects which get updated over time. It is important to be able to specify which packages you have used to develop a certain piece of code so that other people can easily run the code with the same packages and reproduce your results. This

<sup>29</sup> These rules can be found in the documentation - <https://docs.julialang.org/en/v1/manual/methods/>

<sup>30</sup> A set of principles for professional software developers to help write clear and maintainable code, introduced by Robert Martin (also known as *Uncle Bob*).

is also helpful for revisiting an old piece of code you have written which should still work, even years after it was written.

To ensure reproducibility of your code over time, and to make sure that anyone can run your code in the same way without any trouble, it is important to keep track of the external dependencies of your code. In Python, you may be familiar with a `requirements.txt` file which lists the packages required and (optionally) the version number requires. Certain features of packages are only available in later versions of the package, or some functionality may be changed or removed in future versions, restricting the number of compatible versions of the project. Julia has a built in system for managing the packages required for a given project which we call an *environment*.

Environments are reminiscent of *conda* environments, used to isolate the packages installed for different projects in Python. In Julia, these environments are far more lightweight and are specified in a single text file called `Project.toml`. This is managed by the built-in package manager which can be accessed via the Julia REPL. A folder containing a `Project.toml` file is an environment. This environment lists all the packages required to run code within the project contained in the folder.

Environments are **essential** for producing reproducible projects, and ensuring that anyone can run your code. It is highly discouraged to install all the packages you use in your default environment and reuse these packages between projects as this becomes incredibly cluttered and can lead to severe compatibility issues<sup>31</sup>. Instead, it is recommended to start each new environment with a clean start, with no existing packages and only add what you need and what is used. If a package is no longer used, it should be removed from the environment.

#### 4.5.1 Creating an environment

In order to create an environment, open up the terminal in the root directory of your project and start the Julia REPL using the `julia` command.

Once the terminal is open, you can press the “close square-bracket key”, `]` to open the package management interface<sup>32</sup>. From here, you can activate the environment of the current folder by typing

```
] activate .
```

<sup>31</sup> There are a few packages like *Revise.jl* or *BenchmarkTools.jl* which are often installed in the global environment since they only affect REPL usage and code development and are often not used in any source code.

<sup>32</sup> Pressing the backspace key will go back to the normal REPL. You can also access the shell with the `;` key, but this does not work very well on Windows.

where the `.` symbolises the current folder. Activating an environment directs the package manager to the correct location of the `Project.toml` file. Once activated, you should see the name of the folder show up in parentheses on the left of the cursor. From here, you can add a package by typing `add PackageName`. For example, the package `Plots.jl` can be added using

```
] add Plots
```

Notice that the `.jl` extension is left off. The package manager has access to a registry of all public Julia packages which is used to find the right dependency to be added to your project. Even unregistered packages can be added if you have them in a folder, in which case you use the path to the folder, or in a GitHub repository.

Once the package is added, it will usually do some precompilation to help speed-up usage later. This process may take a while, but can be sped up by starting Julia with more threads (discussed in the workshops). After precompilation has finished, you can use the package:

```
using Plots
plot(rand(10))
```

If you type this into the REPL, you should see line plot with random data show up.

#### 4.5.2 *Installing packages*

If you have downloaded someone else's project repository and you want to install the packages, you can simply open the REPL in the folder of the project and make sure it is activated. You can do this in one go by using the command `julia --project` which opens the REPL in the current folder and activates the environment in that folder.

Once activated, type `instantiate` into the package manager console:

```
] instantiate
```

This will start downloading and installing all the packages listed in the environment. Sometimes, you will also want to use the command `resolve`.

### 4.5.3 *Removing packages*

You can remove a package by entering the package manager console of the REPL and using the command `rm`, which stands for remove:

```
] rm Plots
```

### 4.5.4 *Getting help*

Typing in the command `help` to the package manager console in the REPL will show you a list of commands which are useful for managing your environment.





## 5 *Measuring Performance*

If we are going to learn about optimisation, we need to understand how to quantify the performance of our algorithms. All resources in a computer system are finite, but the ones that we usually care most about are *time* and *memory*<sup>1</sup>. Here, we will discuss various approaches of quantifying how many resources are required to perform a specific task.

<sup>1</sup>For some cases, there are other concerns such as energy usage, storage requires etc.

As a general rule of thumb in Julia, one should only ever benchmark/profile code **within a function**, and not executed at global scope, this will give the most accurate results. This is inline with the performance tips of the language where all time sensitive code is put inside a function.

Measuring the runtime of a program can be done a few different ways. Here, we will talk about timing, benchmarking and profiling which are slightly different approaches to quantify the performance of an algorithm.

### 5.1 *Timing*

We can start with the most basic form of measuring performance: timing how long the code takes to run. Modern computers have internal clocks to keep track of time. This is often referring to as measure the **wall-time** of an algorithm. These clocks are usually accurate on the scale of nanoseconds. This is because the CPU has to synchronise its behaviour very precisely. We can take advantage of this built-in feature and measure the current time before a piece of code has executed, and after it has finished. Taking the difference of these measurements will give you the elapsed time of the function.

Let's write an example in Julia. Take the example of creating a 1000 by 1000 array of random numbers. We can do this with the `rand(1000, 1000)` function call. We are not interested in the result of this function, but rather the time it takes

to run this code. In Julia, there exists the `@time` macro, which measures the time taken for given code to execute<sup>2</sup>. Let's run this code multiple times and see what the results give.

```
julia> @time rand(1000,1000);
0.015791 seconds (2 allocations: 7.629 MiB)
julia> @time rand(1000,1000);
0.015770 seconds (2 allocations: 7.629 MiB)
julia> @time rand(1000,1000);
0.002748 seconds (2 allocations: 7.629 MiB)
julia> @time rand(1000,1000);
0.002724 seconds (2 allocations: 7.629 MiB)
```

Notice that the time taken is different on each execution, even though the piece of code is the same. This is where we encounter the first issue of timing - the execution time cannot be guaranteed to be consistent between runs. This is because a modern computer is a complex machine, which is usually running the operating system and multiple applications at the same time. The operating system controls a scheduler, which tells the CPU which bit of code to run at any time. What is important here is that a program can be interrupted (the execution is paused), while the CPU switches to process a different task. This can cause the same piece of code to have different measured times as seen above.

Another reason for the discrepancy is that modern CPUs dynamically adjust the speed at which they operate, especially on devices such as laptops and phones. This is because running at 100% speed uses a lot of power, even when the CPU is idle. The CPU will try and operate at full speed when it is required, and switch back to a low power mode when the performance is not needed.

This method of timing is also unsuitable in Julia as the language is Just-in-Time compiled, which means that the first call to a function will include some additional compilation time. This should only happen on the first call, and subsequent calls will be much faster. However, it is strongly discouraged to time your code using this primitive method, and instead, benchmark the function - this is the topic of the next section.

## 5.2 Benchmarking

If the changes in measurement are only small fluctuations, we can average over a number of repeats to get a better idea of how long the code takes to run. This is

<sup>2</sup> One can also use the `@elapsed` macro to store the execution time (in seconds) of some code which can be used to plot a graph

where the idea of benchmarking comes in. Benchmarking can be thought of as a more systematic approach to timing a piece of code. In Julia, benchmarking can be handled by a package called *BenchmarkTools.jl*, which will be used frequently throughout this book. Just like the `@time` macro from before, this package provides a few more macros which are used to systematically time a piece of code. Let's apply the `@benchmark` macro to our `rand` function:

```
julia> import BenchmarkTools: @benchmark
julia> @benchmark rand(1000,1000)
BenchmarkTools.Trial: 2555 samples with 1 evaluation.
Range (min ... max):  898.646 μs ... 20.912 ms | GC (min ... max): 0.00% ... 0.00%
Time (median):       910.509 μs | GC (median): 0.00%
Time (mean ± σ):    1.951 ms ± 3.459 ms | GC (mean ± σ): 4.82% ± 12.29%
█
899 μs          Histogram: frequency by time          13.9 ms <
Memory estimate: 7.63 MiB, allocs estimate: 2.
```

This outputs a much more comprehensive breakdown of the performance of the `rand` function. One can see that this benchmark yielded a huge range of results. Notice that this benchmark also talks about memory and allocation data which is usually highly correlated with the execution speed of a function, which is discussed in more detail later in the chapter.

Why are some executions of this function so much faster than what we have seen before? To answer this, we must remember that Julia is a “Just-in-Time” compiled language. This means that when Julia executes a piece of code for the first time, it has to compile it into something the computer can process. Most of the in-built timing methods in Julia also include this compilation time. This is the main reasons for using the external package, as it will screen out the compile time and only show you the results that are important.

Why are some executions of this function so much faster than what we have seen before? To answer this, we must remember that Julia is a “Just-in-Time” compiled language. This means that when Julia executes a piece of code for the first time, it has to compile it into something the computer can process. Most of the in-built timing methods in Julia also include this compilation time. This is the main reason for using the external package, as it will screen out the compile time and only show you the results that are important.

Usually, one favours the `@btime` macro over the `@benchmark` since it gives a much more succinct output:

```

julia> import BenchmarkTools: @btime
julia> @btime rand(1000,1000);
904.087 μs (2 allocations: 7.63 MiB)

```

**Note:** This macro will return the **minimum** time taken over a set of evaluations. This can give misleading results for code that may be intermittently slowed down, such as code with heavy amounts of allocations. Specifically, for code with a high number of allocations, it is often better to look at the histogram of results from a full `@benchmark`. If you are embedding the function you have benchmarked with `@btime` inside a loop, calling it  $N$  times, you may expect the result to take  $N$  times longer than the number given you by `@btime`, but this is often an **underestimate**. This is also key when comparing two implementations, as it is often important to compare the averages and not just the *best case* scenario.

All timings are usually given with an SI prefix, such as  $n$ ,  $\mu$  or  $m$  to indicate *nano*, *micro* or *milli* respectively. Remember, these stand for  $10^{-9}$ ,  $10^{-6}$  and  $10^{-3}$  respectively.


If you want to be able to time a function and store the result in a variable, you can use the `@beLapsed` macro to simply return the timing, aggregated (via the minimum) over many samples. If you need more precise control over how the benchmark is run, you should consult the documentation for *BenchmarkTools.jl*.

One very important factor when benchmarking using the macros from *BenchmarkTools.jl*, is to ensure that you are properly **interpolating** variables. There are usually allocations when using variables to call the functions, as these are often global variables, which are heap allocated. Let's take a look at an example:

```

julia> my_arr = rand(1024);
julia> @benchmark sum(my_arr)
BenchmarkTools.Trial: 10000 samples with 976 evaluations.
Range (min ... max): 68.184 ns ... 9.004 μs      GC (min ... max): 0.00% ... 98.88%
Time (median):       74.888 ns                  GC (median):      0.00%
Time (mean ± σ):     75.344 ns ± 89.354 ns      GC (mean ± σ):   1.18% ± 0.99%

```



```

68.2 ns      Histogram: frequency by time      82.7 ns <
Memory estimate: 16 bytes, allocs estimate: 1.

```

Here, we have an allocation of just 16 bytes, but it does exist. We don't expect this sum to allocate at all. The reason for this, is using the global variable to pass into our function. Instead of using this global variable, we should interpolate the contents inside the benchmark:

```

julia> @benchmark sum($my_arr)
BenchmarkTools.Trial: 10000 samples with 987 evaluations.
Range (min ... max): 49.172 ns ... 98.589 ns | GC (min ... max): 0.00% ... 0.00%
Time (median): 51.405 ns | GC (median): 0.00%
Time (mean ± σ): 51.660 ns ± 1.374 ns | GC (mean ± σ): 0.00% ± 0.00%

Histogram: frequency by time
49.2 ns 57.1 ns <
Memory estimate: 0 bytes, allocs estimate: 0.

```

Notice that this removed the mysterious allocation, while also seemingly improving the performance. It is critical that when passing in arguments to a function to be benchmarked via *BenchmarkTools.jl*, you **must** interpolate the values being passed into the function. Similarly, if you want to avoid creating a temporary variable, but do not want to benchmark its creation, you can interpolate:

```

julia> @benchmark sum($(rand(1024)))
BenchmarkTools.Trial: 10000 samples with 987 evaluations.
Range (min ... max): 49.558 ns ... 122.961 ns | GC (min ... max): 0.00% ... 0.00%
Time (median): 51.395 ns | GC (median): 0.00%
Time (mean ± σ): 51.656 ns ± 1.506 ns | GC (mean ± σ): 0.00% ± 0.00%

Histogram: frequency by time
49.6 ns 57 ns <
Memory estimate: 0 bytes, allocs estimate: 0.

```

Interpolation is key to accurate benchmarking using the macros. There are many options to expand on this capability, such as setup arguments, which may be of interest to low level benchmarking.

### 5.3 Profiling

Let's say you have a more complicated piece of code that you are trying to optimise. Many functions are very complex and call other nested functions, and it would be tedious to try and benchmark each individual line of code in this algorithm, and maybe that's not even possible, since you are using some else's code and cannot edit their source code directly. What can you do?

This is where profiling comes into play. Profiling can allow you to diagnose poor performing parts of your code base. It does this by running your program as normal and inspecting which parts of the code get called and for how long.

There are many implementations for achieving this, but we will only talk about statistical profilers.

A **statistical profiler** works by frequently asking the operating system which part of the program is running at any different time. This is used to build up a picture of which part of the code is being executed for the longest amount of time during the lifetime of the program. This method can be very good as it allows the code to run at close to full speed while the profile is being collected. The downside is that this method is often prone to numerical errors and mistakes and is often unsuitable for some tasks. This is the main way that Julia allows for profiling in the language. The built-in library *Profile.jl* contains tools for profiling your code and the external package *PProf.jl* contains tools to let you visualise the profile results. An example of using these tools to diagnose memory allocations is provided in the next section, but applies to testing speed by removing the `Allocs` submodule from each of the commands.

A profiler is quite good at identifying the relatively slow parts of your code. If you have a block of code that takes 95% of the execution time, it is worth focusing on that block instead of any code in the other 5%. This is important to recognise, since profiling can be very informative, often times independent of the hardware the program is being run on. This approach will be used frequently to identify slow running parts of your code. Often times, most programs have a **hot loop**, which repeatedly executes code many times. It is often work identifying this hot loop and optimising the inner function before anything else.

## 5.4 Memory Usage

All resources on a computer are usually finite. The resource which is usually considered first is time. How long will it take to run a certain piece of code? An equally important question is “how much memory will this algorithm use?”. Memory is also a finite resource and a developer should be considerate of the memory footprint his code has. Similar tools to what we have already discussed can be repurposed to measure how much memory has been used.

In Section 3.2.7, we spoke about the **stack** and the **heap**. Here, it is important to distinguish between them during profiling. The stack tends to be a lot smaller than the heap, and one should avoid allocating large objects on the stack. Often this can lead to crashes or poor performance. In the profiling techniques we have discussed, you will have seen that these techniques also measure the number

of *allocations*, which refers to allocations on the heap. Additionally, most of the timing includes an estimate of the GC <sup>3</sup> time spent trying to the clean-up the memory allocated by the function.

<sup>3</sup> Garbage Collection

Let's take a look at profiling the function given:

```
julia> function mem_func(n, pow)
    arr = rand(n)
    s = 0
    for x in arr
        s += x ^ pow
    end
end
mem_func (generic function with 1 method)
julia> n = 1024;
julia> pow = 3;
julia> @btime mem_func($n, $pow);
2.496 μs (1 allocation: 8.12 KiB)
```

Notice that we are *interpolating* the variables `n` and `pow` by using the `$` character. This should always be done to accurately benchmark a function with input arguments which come from variables. Secondly, notice that the output of this benchmark specifies that there was one allocation with a total of 8.12 KiB (which is  $1024 \times 8 = 8192$  bytes). However, it does not tell us where the allocation is. For our function, it is easy to see that creating the array caused the allocation, however, in some code addressing where this comes from can be a challenge.

Julia v1.8 introduced an allocation profiler which allows you to profile your code and visually inspect where allocations are coming from. This new functionality is covered and demonstrated in a great talk from JuliaCon 2022 called "Hunting down allocations with Julia 1.8's Allocation Profiler"<sup>4</sup>. I will use the same example given to demonstrate how this works in practice. Take the example of serialising a CSV<sup>5</sup> file with some data: We can benchmark this code with some data:

<sup>4</sup> <https://live.juliacon.org/talk/YHYSEM>

<sup>5</sup> Comma Separated Values

```
julia> rows = [rand(10) for _ in 1:128];
julia> @benchmark TestAllocs.serialise($rows);
```

We can see that after compilation, we still have a huge amount of allocations present (over 5000 allocations), and a huge amount of time is spent garbage collecting. Instead of manually trying to figure out which lines are allocating, we can instead use a profiler to find this information out automatically. In order to do

```

module TestAllocs
function serialise(rows::Vector{Vector{T}}) where {T}
    output = ""
    for row in rows
        for col_idx in eachindex(row)
            val = row[col_idx]
            output = output * string(val) # string concat
            if col_idx < length(row)
                output = output * ", "
            end
        end
        # add a newline
        output = output * "\n"
    end
    return output
end
end

```

Algorithm 5.1. An example algorithm to show how to profile memory allocations.

this, we must use the `Profile.jl` package, which is included in the base library.

You can run

```
using Pkg; Pkg.add("Profile")
```

to add this to your environment. Now we can run the profiler, to specifically track the allocations<sup>6</sup>:

```
using Profile
Profile.Allocs.@profile sample_rate=1 TestAllocs.serialise(rows);
```

One can choose a sample rate between 0 and 1. It is recommended to use `@time` first to figure out how many allocations there are, and if there are many, to use a small sample rate. If you choose a number that is too large, this profile will take a very long time to finish. In order to visualise these allocations we will also use a package called `PProf.jl` (created by Google, which you will also need to install). You can run the following code to visualise the allocations in a web browser:

```
using PProf
PProf.Allocs.pprof(from_c=false)
```

The `from_c` parameter will allow you to filter out any stack frames internally allocated by the Julia runtime, which usually just clutters up your results. You

<sup>6</sup> You should make sure that you have run the code to be profiled at least once to avoid profiling the compilation process of the function as well.



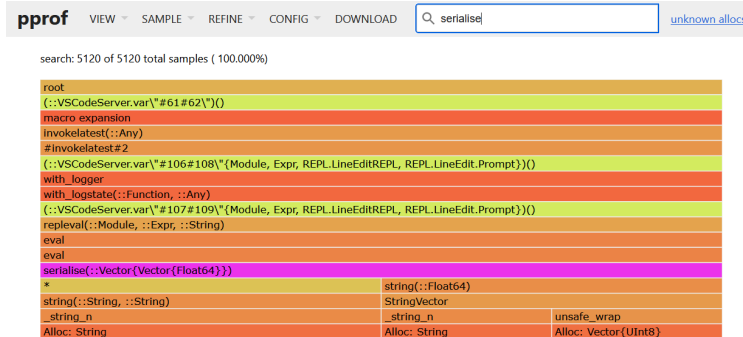


Figure 5.1. A flame graph showing the number of allocations of the `serialise` function.

will get a link to a localhost website (locally hosted on your machine), which lets you view the results.

If you change the view to a flame graph (View->Flame Graph) you can see something like Figure 5.1. Changing the sample from `allocs` to `size` (Sample->size) will weight the size of each block by the size. The bars at the bottom are the most useful as these refer to the lowest level calls. You can see that the problem function is our `serialise` function. Use the search bar to highlight this function call. Once highlighted, you can switch to the source view, which will highlight for us where all the allocations come from.

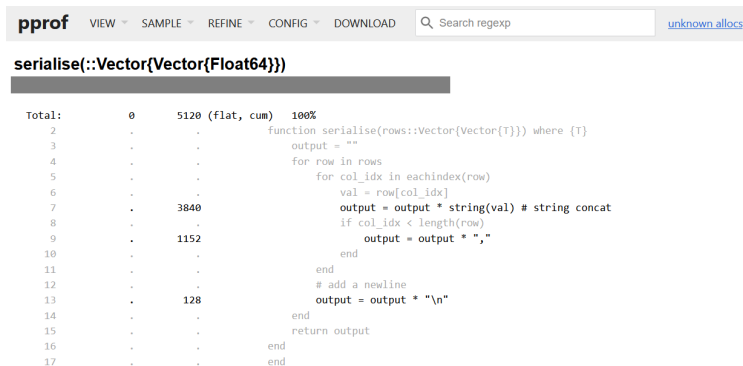


Figure 5.2. A source view showing the number of allocations of each line in the `serialise` function.

Looking at Figure 5.2, we can see that we are allocating a lot whenever we are concatenating a string, and also when converting a value to a string. Each concatenation produces a new string which causes another allocation. Instead

of concatenating, we want to print to a buffer instead. Let's look at an improved version of this code, shown in Algorithm 5.2.

```

module TestAllocsFast
function serialise(rows::Vector{Vector{T}}) where {T}
    buffer = IOBuffer()
    for row in rows
        for col_idx in eachindex(row)
            val = row[col_idx]
            print(buffer, val)
            if col_idx < length(row)
                print(buffer, ",")
            end
        end
        # add a newline
        print(buffer, "\n")
    end
    return String(take!(buffer))
end
end

```


Algorithm 5.2. An example algorithm to show how to profile memory allocations, but fixing the performance issues of Algorithm 5.1.

If we rerun the results, we can see a huge performance improvement:

```

julia> @benchmark TestAllocsFast.serialise($rows)
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
Range (min ... max): 159.687 μs ... 8.916 ms | GC (min ... max): 0.00% ... 93.63%
Time (median): 270.199 μs | GC (median): 0.00%
Time (mean ± σ): 270.715 μs ± 584.493 μs | GC (mean ± σ): 14.67% ± 6.62%

```



```

Histogram: frequency by time
Memory estimate: 568.76 KiB, allocs estimate: 2570.

```

We can see from Figure 5.3 that the allocations are much improved in terms of size, but there are still lines which allocate. This implementation is not completely optimised, but it has improved hugely on the previous implementation. The advantage of this profiling approach is that you can find allocations that you may not have realised are there.



Figure 5.3. A source view showing the number of allocations of each line in the **faster serialise** function from Algorithm 5.2.

### 5.4.1 Conclusion

While these practical methods give you a wide array of tools, which can be used to analyse runtime metrics, they do not give you the full picture about the performance of the studied algorithms. For one, each of the performance metrics are specific to the hardware on which the benchmarks were run. This makes it almost impossible for different researchers to compare their results via these empirical methods, unless they are running each algorithm on the same hardware. Alongside empirical benchmarks, we must also have a set of theoretical tools which can be used to quantify the performance of an algorithm. This is going to be the topic of the next section.

## 5.5 Computational Complexity

When timing our algorithms we are implicitly asking how many resources an algorithm needs. As mentioned before, these resources can be in terms of time or space requirements. Space refers simply to the amount of information the algorithm needs to “remember” at any one time. We will look at both of these type of resources here.

First, we need to provide a distinct approach to the practical measurements of the previous section. The failings of this approach is that it was often very abstract to compare algorithms on different hardware, as the numbers themselves are somewhat arbitrary and highly coupled to the system running the benchmarks.

We need a way to describe the typical performance of an algorithm, abstracted away from the absolute time taken to run the algorithm. The way we can analyse this is by asking about scale. In order to do this, let's create a very simple example algorithm as seen in Algorithm 5.3.

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a) == size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```

Algorithm 5.3. A simple and general implementation that calculates the element-wise multiplication of two arrays,  $a$  and  $b$ .

This algorithm is only correct when the inputs are arrays of the same size. The `eachindex` function is an iterator which allows the for loop to iterate through each element in the array `a`. The `similar` function is very helpful, as it allows us to create a new array which is the same size and type as `a`, but with no values specified. The values in this array are often random, as the elements are not set when created.

What do we mean when we talk about scale? Let's think about the *size* of the inputs. As an example, let's say that  $a$  and  $b$  are  $n$ -dimensional vectors. How will the resources used scale with  $n$ ? In order to answer this, we need to count how many lines of code are executed. If we ignore the size check, we first arrive at allocating the new array. Allocating is the process of carving out a section of your memory in which to store some results during the runtime of your program. An array can be deallocated to free up space for other objects during the runtime. Allocating is usually an expensive operation, but we don't know how it scales with the input size, so let's just denote this with  $s(n)$ , which is a function describing the *time* cost of allocating an array using the `similar` function; for now, let's assume that this can be done in  $\mathcal{O}(n^k)$  where  $k \leq 1$ . Next of all, we have the for loop. We can ignore initialising the for loop, since this is not dependent on the number of times the for loop will be executed. However, the code inside the for loop will iterate many times. In this case, there is an iteration for each element in the  $a$  (or  $b$ ) array. This means that this piece of code will execute  $n$  times. The return at the

end of the function also does not scale with  $n$ . We can conclude that this piece of code will likely scale linearly with  $n$ . This means that the time taken to run this code should be of the form:

$$T(n) = c_1n + c_0 \quad (5.1)$$

This is the equation of a straight line, and what we can do is actually time this function using the `@elapsed` function, for different sized inputs and graph this to test our analysis.

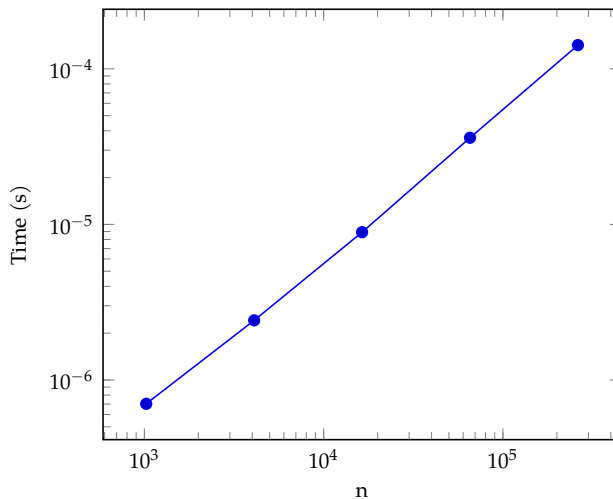


Figure 5.4. Time taken when using `elementmul` for several values of  $n$ . The time taken is measured several times for each value of  $n$  using the `@elapsed` macro from `BenchmarkTools.jl`.

In Figure 5.4, you can see that the time is roughly linearly proportional to the size of the vectors in log-log space. Since we can model this relationship as a straight line in log-log, we can write plot is  $\log(t - t_0) = m \log(n) + c$ , where  $m$  is the gradient of the line and  $c$  is the  $t$ -intercept. This is equivalent to

$$t = t_0 + An^m, \quad (5.2)$$

where  $A = e^c$  and  $t_0$  is some constant overhead time. If we were to calculate this, we would see that  $m$  is roughly 1, which means this function scales linearly. This means that our assumption about the `similar` function was correct, at least in the range of  $n$  values seen.

**Exercise 5.1.** Can you design an experiment to test the time complexity of the `similar` function? Do the results change when comparing the distribution of times taken, when compared with taking the minimum time?

Almost any computer that we run this experiment on will give us the same result, that the time taken for `elementmul` to execute is linearly proportional to vector size  $n$ . The results will not be exactly the same, since the gradient and offset of the line will be different. These are the constants  $c_1$  and  $c_2$  in Equation (5.1). For this reason, Computer Scientists often use *asymptotic* notation, or more fondly referred to as *Big O* notation. This type of notation drops the constants from the equation, but also simplifies the equation to only include the fastest scaling terms of  $n$ . This means that if we have a time relationship of  $T(n) = c_2n^2 + c_1n + c_0$ , we would only keep the  $n^2$  term as this term grows the fastest. For this example, the *Big O* notation would be  $\mathcal{O}(n^2)$ . The way we calculate the important terms is by taking limits as  $n$  approaches  $\infty$  as shown below:

$$\lim_{n \rightarrow \infty} T(n) = \lim_{n \rightarrow \infty} n^2 \left( c_2 + \frac{c_1}{n} + \frac{c_0}{n^2} \right) = n^2(c_2 + 0 + 0) = c_2n^2 \quad (5.3)$$

One can see that we can factor out the biggest term  $n^2$  and the remaining terms will approach 0 as  $n$  approaches  $\infty$ . This allows us to ignore these terms, leaving only the constant multiplied by  $n^2$ . Since we are also not interested in the constant, we drop this to leave us with the asymptotic time complexity being  $\mathcal{O}(n^2)$ .

### 5.5.1 Examples

The process of assessing runtime is a skill that can be easily learnt with practice. For this reason, let's go through a few algorithms and assess their computational complexity in both time and space.

#### *Straightforward Nearest Neighbour Calculation*

To start with, take a nearest neighbour algorithm. Given many points, forming a "point cloud", and an input point, the algorithm calculates which point is closest to every other point. This algorithm is commonly used in machine learning classification problems. To define this problem, we have a cloud of  $D$  dimensional points. We have  $N$  of these points. The point cloud can therefore be represented by an  $N \times D$  array, since we need to store  $D$  numbers for each of the  $N$  points. The output is an  $N$ -dimensional array. The value of index  $i$  in the array corresponds to the index of the closest point to the  $i^{\text{th}}$  point in the point cloud. We also need

to define how we measure the distance between points; to keep this simple, let's use the Euclidean distance.

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Algorithm 5.4. A simple implementation of a nearest neighbour algorithm, using a Euclidean distance function and calculating the nearest neighbour to each point in the cloud.

Now, let's calculate the computational complexity of Algorithm 5.4. First, let's identify how the algorithm scales with the number of points,  $N$ . We first notice that we allocate an array of  $N$  integers. If we are on a 64-bit system, then this takes up  $64N$  bits of memory. After this, we hit our first for loop. The variables here do not have to be stored in memory and hence are not allocated on the heap, but in fact, are allocated on the stack. This means that their allocation is trivial. However, this takes a linear amount of time to do. Keep note of this. Finally, we enter into our second nested for loop, which occurs  $N$  times. Here, we are calculating the distance between two points and then possibly writing to memory. The operations here are linear (dependent on  $D$ ) and hence we can count this as a single "unit of computation"; it must be realised that this calculation happens  $N^2$  times. This is the only place where the algorithm scales with  $N$  and hence we can conclude that the complexity of this algorithm is  $\mathcal{O}(N^2)$ . In terms of memory footprint, this algorithm scales linearly with  $N$ . The important takeaway here is that each

nested for loop makes you multiply by the number of times the loop occurs. We can explore this idea further with a recursive algorithm.

### *Recursive Complexity*

```
function recursivecalc(n, a)
    if n==1
        return a*a
    end

    s = 0.0
    for i in 1:n
        s += recursivecalc(n-1, i)
    end

    return s
end
```

Algorithm 5.5. An arbitrary algorithm to illustrate the rules of calculating computational complexity.

Before we go into details, try and have a go at calculating the complexity of this algorithm.

Since this is a recursive algorithm, let's write down what happens at one step of this computation. Let us calculate the complexity of `recursivecalc(k, 1)`, where  $k > 1$  and  $k \in \mathbb{Z}$ . We ignore the first check (however, we must remember that this takes a constant amount of time). Then we go into a `for` loop with the inner loop taking  $\mathcal{O}(f(k-1))$  amount of time complexity, where  $f$  is the unknown complexity of the algorithm. This means that the computational complexity can be written as

$$\mathcal{O}(f(k)) = \mathcal{O}(k \times f(k-1)). \quad (5.4)$$

This fundamentally is a recursive calculation. If we substitute this equation back into itself, we find that  $\mathcal{O}(f(k)) = \mathcal{O}(k(k-1) \times f(k-2))$ . Each time we substitute the equation back into itself, we multiply another term, until we are left with  $f(1)$ , which can be calculated exactly, as this takes linear time, which therefore has a value of just 1. The complexity function is therefore given by  $f(k) = k!$  and hence the computational complexity of this algorithm is  $\mathcal{O}(k!)$ . If we were to do a similar substitution process in the code, as we did with the equation, we would find that we would have  $k$  nested loops. Each loop multiplies the complexity by the number of times the loop is iterated through. This is a simple rule to use when calculating complexity, which is just derived from the definition of multiplication.



*Tree-like Complexity*

A tree is a very common data structure in computer science, since they often have algorithmic implementations which have far better computational complexity. Let's take an example of inserting a number into a sorted list. Let's say we have a sorted array of  $N$  numbers and want to insert a new number into the correct location in this list. An implementation of this algorithm could look like the following:

```
function insertintosorted!(numbers, num)
    n = length(numbers)
    startpoint = 1
    endpoint = n
    while startpoint < endpoint > 1
        midpoint = (startpoint+endpoint)÷2
        if num < numbers[midpoint]
            endpoint = midpoint
        elseif num > numbers[midpoint]
            startpoint = midpoint
        else
            insert!(numbers, midpoint, num)
            return
        end
    end
    insert!(numbers, startpoint, num)
    nothing
end
```

Algorithm 5.6. A simple algorithm which inserts a number into an already sorted list, making sure the list is still sorted after insertion. This algorithm assumes the list is sorted in ascending order.

Here, since we have a while loop instead of a for loop, so it is not immediately obvious how many times it is executed. However, we know that the range being checked halves upon every iteration of this while loop. The while loop ends either when the number is prematurely inserted, or when the range has contracted to 1. This means that we need to ask the question - how many times do you have to halve  $n$  to get 1? In mathematical terms, we are asking, which  $x$  satisfies the equation  $n \times (\frac{1}{2})^x = 1$ , which is solved by  $x = \log_2(n)$ . This means that this algorithm scales with the logarithm of  $n$ , and hence the complexity is simply written as  $\mathcal{O}(\log(n))$ .

This process of cutting down a search space by a constant factor is surprisingly common in many tree-like algorithms. Having an  $\mathcal{O}(\log(n))$  algorithm is far more performant than even linearly complex algorithms ( $\mathcal{O}(n)$ ). Consider a linear

implementation for this same problem that iterates through each of the numbers in the list in order, until the correct place to stop is found. This implementation would, on average, take around  $\frac{n}{2}$  iterations, and hence would be a linear algorithm. If we had  $10^6$  elements, the tree-like implementation would be many orders of magnitude faster than any linear implementation.

### 5.5.2 *Conclusion*

Hopefully, one has gleaned an intuition for calculating computational complexity. While the topic may at first appear somewhat intimidating, one need only be able to count how many times each line is being run, which is usually rather straightforward. This process can become arbitrarily more difficult for more obscure and complex algorithms, especially when the algorithm relies on random numbers. The main takeaway here, is that nested for loops incur a huge computational cost, especially when processing large datasets. If one can avoid this cost with a more efficient algorithm, then it is usually worth the effort to use it, especially if you need a faster speed.

**PART II:**

**HIGH PERFORMANCE CODE**



## 6 Optimising Serial Code

In this chapter, we will learn about optimising the code you write. Here, we will only talk about sequential performance, as we save the discussion of parallelising your code to later chapters. It is important to understand that when we talk about optimisation here, we are not talking about the use of the right algorithm. Here, we are entirely focused on the practical performance of our algorithms on specific hardware, not with the theoretical performance of an algorithm. This is a way of bringing back the constants from the Big  $\mathcal{O}$  notation, which tell you the latency of the operations and the speed at which they perform. It is entirely possible that an  $\mathcal{O}(n)$  algorithm outperforms an  $\mathcal{O}(\log n)$  algorithm, if the system size is small enough. For this reason, we must take a practical view when talking about optimisation.

We will rely on benchmarks using the *BenchmarkTools.jl* package to see performance differences.

### 6.1 Caching and Memory Locality

In Section 3.1.2, we briefly discussed how modern CPUs are structured. This topic usually takes a backseat when talking about performance comparisons of various algorithms. However, it is of paramount importance when writing performance critical code.

In Figure 6.1, one can see a model of how physical cache on the CPU is structured. It is important to keep in mind that L1-L3 caches are physically stored on the CPU die. The CPU and the RAM (the main memory) are connected via the motherboard, making accessing the RAM a much slower operation than accessing the onboard L caches. It is important to remember that the caches closest to the

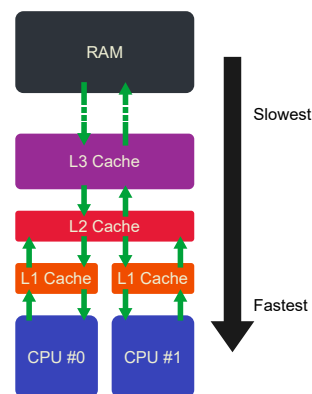


Figure 6.1. A diagram of a typical cache setup for a dual-core processor. The green arrows show the direction of memory travel. The closer the memory is to the CPU, the faster the memory transactions are.

CPU are the fastest, but lowest capacity, with increasing capacity and latency moving further away from the CPU.

An important observation is that if the memory required by the CPU is stored in the L1 cache, then the operation to retrieve that memory is extremely fast (low latency). Again, if it is stored in the L2 cache, then it is a bit slower to access, but the CPU is still able to access the memory without having to go to RAM. The story is the same for the L3 cache, which is shared amongst all CPU cores. The decrease in speed comes from latency in retrieving the information, if you have to check more places, it will take longer, even if checking each place takes the same amount of time.

It would be very beneficial if all the data required in a program were able to be stored in one of the caches, and the processor would try and minimise the number of times memory has to be retrieved from RAM. A **cache miss** happens when memory has to be retrieved from RAM, instead of being available in the cache. If the CPU is able to predict what memory will be needed next, it is able to store this memory inside the cache so that cache misses can be avoided.

Effectively handling cache relies on knowing as much about the program's effect on memory as possible - so that the compiler and the produced machine code can lay out the memory in a way that is cache friendly. A CPU contains heuristics on what memory to cache, and it is important to align your code with the expectation of these heuristics. This is also why types are so important, since a type essentially specifies how much memory an object takes up and what the physical bits mean. If a compiler knows the type of the object, it can write instructions to reserve the exact amount of space required for that object, and make it more likely to be available in the cache when it is needed.

One trick that computers use to minimise cache misses, is to retrieve entire blocks of memory when iterating over an array, instead of a single value: This is known as a **cache line**. The first time you access an array, and try to access the first element, the program can guess that the next few values in the array may be needed and hence will copy the next few items into the cache as well. This way, when the program iterates to the next element, it is already in the cache, and a journey back to RAM is not required. The next section will discuss a specific form this takes.

Finally, code may be written as to minimise the amount of memory that needs to be stored. If we are only operating on a few bits of memory at once, the machine

instructions can keep all these memory inside registers on the CPU - avoiding the need for storing anything in cache!

### 6.1.1 Multidimensional Array Indexing

As Julia is designed around numerical computation, it contains an implementation for multidimensional arrays in the core library. Multidimensional array support is critical for scientific computing. Python's main package for this is *numpy* and MATLAB is centred around the multidimensional array. It is important to understand how these arrays are actually stored in memory, so that we can avoid introducing performance hits to our applications.

You will remember that one dimensional arrays are stored in a contiguous block in memory and the variable which "stores" the array is, in fact, just a pointer to the starting point of this contiguous block. The memory address of any value in the array can be calculated by knowing the index of the element and appending this to the value of the pointer. The question then arises, if an array is stored as a contiguous block of memory, how is a multidimensional array stored? The answer is simple, it is still stored as a single 1D block of memory, the only thing that changes is how one calculates the index into that 1D array, based on the indices from each dimension. One can think of this as splitting up a matrix (or a tensor) into rows or columns and placing them side by side, and labelling the final 1D string of elements with their position in that array.

Take the example of a matrix. A matrix has two numbers  $i$  and  $j$  which we will take to represent which row and column the element lives in respectively. The element of the matrix  $A$ , indexed as  $A_{ij}$  is the number in the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column. There are two options here for storing this array in linear memory. The one which Julia chooses is known as **column-major order**, which calculates the linear index,  $k$ , as follows:

$$k = j \times N_{\text{rows}} + i, \quad (6.1)$$

where  $i$ ,  $j$  and  $k$  are using zero-based indexing and  $N_{\text{rows}}$  is the total number of rows in the matrix. The alternative equation which uses row-major ordering (as in *numpy*'s implementation) uses the following equation:

$$k = i \times N_{\text{cols}} + j, \quad (6.2)$$

where  $N_{\text{cols}}$  is the total number of columns. The difference between these orderings can be seen in Figure 6.2.

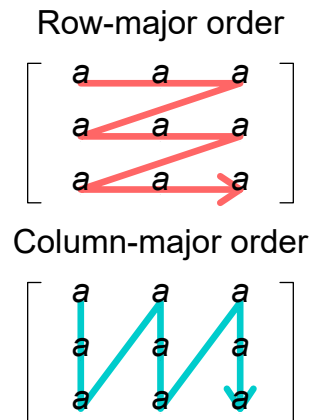


Figure 6.2. A diagram from Wikipedia ([https://en.wikipedia.org/wiki/Row\\_and\\_column-major\\_order](https://en.wikipedia.org/wiki/Row_and_column-major_order)) which shows the way in which multidimensional arrays are stored in linear memory under row-major and column-major ordering schemes.

These schemes can be extended into multiple dimensions. The general rule of thumb for column-major indexing is that one should begin incrementing from the left-most index first, and then increment each subsequent index once one reaches the end of that index.

We can design an experiment to compare the same algorithm, with the only difference being the order in which data is iterated over through an array. For this experiment, we will take the problem of implementing matrix addition. The implementations are given in Algorithm 6.1 and Algorithm 6.2 for row-major and column-major ordering respectively.

```
function row_major_matrix_add!(C, A, B)
    @assert size(C)==size(A)==size(B)
    @inbounds for i in axes(A, 2)
        for j in axes(A, 1)
            C[i, j] = A[i, j] + B[i, j]
        end
    end
    nothing
end
```

Algorithm 6.1. This algorithm shows the matrix addition of two 2D arrays, with the result being stored in a third array  $C$ , which is mutated. The bounds checking is turned off for performance reasons. In this example, the loop is executed following row-major ordering, such that the right-most index is incremented first.

```
function column_major_matrix_add!(C, A, B)
    @assert size(C)==size(A)==size(B)
    @inbounds for j = axes(A, 2)
        for i in axes(A, 1)
            C[i, j] = A[i, j] + B[i, j]
        end
    end
    nothing
end
```

Algorithm 6.2. This algorithm shows the matrix addition of two 2D arrays, with the result being stored in a third array  $C$ , which is mutated. The bounds checking is turned off for performance reasons. In this example, the loop is executed following column-major ordering, such that the left-most index is incremented first. This is following the linear indexing pattern.

These two algorithms are theoretically exactly equivalent, since the order in which the operations happen has no effect on the output. Each algorithm performs identical operations on the data, the only difference is the order in which these operations are calculated, which has no effect on the output. One would expect these algorithms to have the same performance, but let's test this.

```
julia> N = 1024; A = rand(N, N); B = rand(N, N); C = similar(A);
julia> @benchmark row_major_matrix_add!($C, $A, $B)
```



```

BenchmarkTools.Trial: 691 samples with 1 evaluation.
Range (min ... max):  7.176 ms ... 7.330 ms | GC (min ... max): 0.00% ... 0.00%
Time  (median):       7.222 ms           | GC (median):      0.00%
Time  (mean ± σ):     7.224 ms ± 17.390 μs | GC (mean ± σ):  0.00% ± 0.00%

7.18 ms      Histogram: frequency by time      7.29 ms <
Memory estimate: 0 bytes, allocs estimate: 0.
julia> @benchmark column_major_matrix_add!($C, $A, $B)
BenchmarkTools.Trial: 6647 samples with 1 evaluation.
Range (min ... max):  717.298 μs ... 880.502 μs | GC (min ... max): 0.00% ... 0.00%
Time  (median):       736.976 μs           | GC (median):      0.00%
Time  (mean ± σ):     735.259 μs ± 7.836 μs | GC (mean ± σ):  0.00% ± 0.00%

717 μs      Histogram: frequency by time      750 μs <
Memory estimate: 0 bytes, allocs estimate: 0.

```

There is a huge difference between the performance of these two algorithms, about a 40× difference! Simply swapping the order in which one iterates over rows or columns can have a big impact on performance. This is because the memory is aligned in order to avoid as many cache misses as possible.

### 6.1.2 Hardware Vectorisation

One of the big selling points of Julia, is that we do not **need** to write our code in a vectorised way for performance. Here, we are explicitly reserving **vectorisation** to mean performing bulk, array-level operations (often element-wise). Let's take the example of adding two matrices together, as given in the previous example. There, we used a **for** loop to achieve this. We can do the same by using the **broadcasting** notation in Julia, which operates element-wise: We can benchmark this to see

```

function broadcasting_add!(C, A, B)
    C .= A .+ B
    nothing
end

```

Algorithm 6.3. This algorithm shows the element-wise addition of two arrays, **A** and **B**, and storing the result in another, preallocated array **C**.

it is very similar to the previous implementation, while being a much simpler written implementation:

```

julia> @benchmark broadcasting_add!($C, $A, $B)
BenchmarkTools.Trial: 6638 samples with 1 evaluation.
Range (min ... max): 717.718 μs ... 826.287 μs | GC (min ... max): 0.00% ... 0.00%
Time (median): 737.687 μs | GC (median): 0.00%
Time (mean ± σ): 735.827 μs ± 7.577 μs | GC (mean ± σ): 0.00% ± 0.00%

Histogram: frequency by time
718 μs 750 μs <
Memory estimate: 0 bytes, allocs estimate: 0.

```

We can also write the equivalent `for` loop, taking advantage of Julia’s linear indexing, even for multidimensional arrays: We can benchmark this example to

```

function vector_add!(C, A, B)
    @inbounds for i in eachindex(C, A, B)
        C[i] = A[i] + B[i]
    end
    nothing
end

```

Algorithm 6.4. This algorithm shows the element-wise addition of two arrays, `A` and `B`, and storing the result in another, preallocated array `C`, using a native `for` loop.

see it has roughly the same performance as our array implementation:

```

julia> @benchmark vector_add!($C, $A, $B)
BenchmarkTools.Trial: 6649 samples with 1 evaluation.
Range (min ... max): 717.708 μs ... 842.528 μs | GC (min ... max): 0.00% ... 0.00%
Time (median): 736.665 μs | GC (median): 0.00%
Time (mean ± σ): 735.034 μs ± 7.562 μs | GC (mean ± σ): 0.00% ± 0.00%

Histogram: frequency by time
718 μs 749 μs <
Memory estimate: 0 bytes, allocs estimate: 0.

```

This means that there is only really an aesthetic difference between using the broadcasting (vectorised) notation, and a `for` loop. Even if we are use `@simd` to take advantage of **hardware vectorisation**, we will see no difference, as the Julia compiler is smart enough to do this automatically where it is safe. Benchmarking this leads to the same results:

```

julia> @benchmark vector_simd_add!($C, $A, $B)
BenchmarkTools.Trial: 6650 samples with 1 evaluation.
Range (min ... max): 715.885 μs ... 824.624 μs | GC (min ... max): 0.00% ... 0.00%
Time (median): 736.625 μs | GC (median): 0.00%
Time (mean ± σ): 734.902 μs ± 7.460 μs | GC (mean ± σ): 0.00% ± 0.00%

```

```
function vector_simd_add!(C, A, B)
    @inbounds @simd for i in eachindex(C, A, B)
        C[i] = A[i] + B[i]
    end
    nothing
end
```

Algorithm 6.5. This algorithm shows the element-wise addition of two arrays, *A* and *B*, and storing the result in another, preallocated array *C*, using a native `for` loop with `@simd`.



The `@simd` macro is usually **not** needed to speed up your `for` loops, but can be helpful to give the compiler some additional leeway when the ordering of the results may be important, for example when performing a **reduction**. From the help documentation, this macro asserts several properties of the `for` loop:

- It is safe to execute iterations in arbitrary or overlapping order, with special consideration for reduction variables.
- Floating-point operations on reduction variables can be reordered, possibly causing different results than without `@simd`.

The `@simd` just gives the compiler additional flexibility in auto-vectorising a `for` loop. Some constraints that should be followed:

- The loop must be an innermost loop.
- The loop body must be straight-line code. Therefore, `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short `&&` (and), `||` (or), and `?:` (ternary) expressions into straight-line code if it is safe to evaluate all operands unconditionally. Consider using the `ifelse` function instead of `?:` in the loop if it is safe to do so.
- Accesses must have a stride pattern and cannot be “gathers” (random-index reads) or “scatters” (random-index writes).
- The stride should be unit stride.

### 6.1.3 Where does memory live?

In Section 3.2.7, we introduced the idea of the **stack** and the **heap** as two data structures a program uses to organise its data. Again, these are both stored in RAM, and there is no physical difference between the memory in the stack and the heap. As a rough approximation:

- The **stack** is a special area of memory which stores temporary variables (such as local variables of a function) which are deleted (or forgotten) once a function has finished executing. This is temporary storage.
- The **heap** is used for storage which has variable length, or for objects that are too large for the stack.

It is important to be able to identify which variables will be allocated on the stack, and which will be allocated on the heap. We will talk in later sections about “allocating” memory, but this almost exclusively refers to storing memory on the heap. In this section, we will talk about the key difference between these structures.

Let us analyse the following function:

```
function stack_only_function(x:T) where {T<:Number}
  a = x + 2
  b = sin(x)+10
  c = cos(x) -5
  d = a*b*c
  return d
end
```

Algorithm 6.6. An example function which does not allocate any memory on the heap.

Algorithm 6.6 seems to create 4 different variables. It is critical to understand that this function does not allocate any memory, when the input is just a number. But it is obviously using memory, since we create many different variables. The answer to this apparent contradiction is that it allocates memory on the stack. The reason these variables can exist on the stack is that the size can be predicted at compile-time and hence, space in the stack can be made for these variables. These variables are likely to be able to be stored in physical cache as well, since the stack is so frequently accessed that it is likely to be in the cache.

```

function heap_function(x::T) where {T<:AbstractArray}
    a = x .+ 2
    b = sin.(x) .+ 10
    c = cos.(x) .- 5
    d = a.*b.*c
    return d
end

```

Algorithm 6.7. An example function which a lot of temporary memory on the heap, under the same logic as Algorithm 6.6.

Instead, let us change the example, so that instead we pass in an array with a single element, shown in Algorithm 6.7.

We can benchmark these two functions on an array:

```

julia> A = rand(128); B = similar(A);
julia> isapprox(stack_only_function.(A), heap_function(A))
true
julia> @btime begin
    ($B) .= stack_only_function.($A)
    nothing
end
887.104 ns (0 allocations: 0 bytes)
julia> @btime begin
    ($B) .= heap_function($A)
    nothing
end
1.022 μs (4 allocations: 4.25 KiB)

```

Remember that we are using interpolation with the `$` character to ensure accurate benchmarking. Looking at the results, we see that the stack only function does not allocate any memory, but the heap function allocates some memory four times. These algorithms are theoretically identical and yield the exact same results, so why does one algorithm allocate memory and the other does not? Additionally, why does the one with heap allocations run slower than the stack function?

In each example, we are using the broadcasting notation. Notice that in the stack function we are applying the entire function element wise to the array `A` and then storing the result element wise in the array `B`. As mentioned previously, the broadcast notation will simply compile down into a `for` loop. Inside this loop, the temporary variables `a`, `b`, `c` and `d` are stored on the **stack**. Since we have specified where to put the result<sup>1</sup>, no memory needs to be heap allocated.

<sup>1</sup> We have preallocated the array `B`.

In the heap function, we instead take in the entire array. Each line uses the broadcast notation, but only on the right-hand side. This means that a temporary

array must be allocated to store the result. Each of the variables is a new array allocated on the heap, hence the 4 observed allocations from benchmarking. To reduce this, one could write the entire expression on one line, which would certainly help, but it is often better to write a function for a scalar and use broadcasting instead.

The performance difference between the two implementations is actually somewhat understated here. We are using `@btime` which reports the **minimum** time taken. This will exclude any evaluations which are interrupted by the Garbage Collector, which is likely to happen if this code is used many times within the **hot loop** of your program. Here, we are only measuring the time taken to allocate these new arrays, but ignoring the cost of having to clean them up, which can often be significant, especially in parallel code.

The reason that the stack is able to store all the temporary information needed is because the size of the intermediate values are known at compile time, whereas the size of the array is not known and therefore **must** go on the heap.

One should remember that even though the data of the arrays are allocated on the heap, the pointer<sup>2</sup> is stored on stack. Accessing the array information requires dereferencing that pointer, causing some additional overhead, even if the array only has a single element. It is much easier for the compiler to optimise the machine code if the memory size and layout is known at compile time, instead of having to introduce pointers.

What if the array we are using is of a known size? For example if we write a function to calculate the length of a three-dimensional vector as in Algorithm 6.8. This function happens to work for vectors of any length, as long as they are stored in an array, with each element representing a component of the vector.

```
function cross(a, b)
    [
        (a[2]*b[3]-a[3]*b[2]),
        (a[3]*b[1]-a[1]*b[3]),
        (a[1]*b[2]-a[2]*b[1])
    ]
end
```

<sup>2</sup> Remember that a pointer is just the address to the place in memory where other information is stored.

Algorithm 6.8. A simple implementation of calculating which calculates the cross product between two vectors `a` and `b`, which can be any abstract arrays. This function does not have any size checks, as we are only focused on the performance here.

We can benchmark this function:

```
julia> a = rand(3);
julia> b = rand(3);
```

```

julia> @btime cross($a, $b)
 22.895 ns (1 allocation: 80 bytes)
3-element Vector{Float64}:
 0.18242146759034802
-0.428829354451039
 0.053461858289415076

```

This function runs very quickly, but there is a way we can make this even faster. If we know we are using 3D vectors a lot, we can make use of a package called *StaticArrays.jl*, which provides a nice utility for implementing fast algorithms which allow arrays to be stack allocated since they are forced to be of a fixed size. Let's have a look at the implementation for this type of static array. Since this is an external package, you may need to add the package with `pkg"add StaticArrays"` in the console. To be able to use the package, one needs to import it:

```

julia> using StaticArrays;

```

From here, we can add a specific implementation of the cross product function, which returns a static vector with three dimensions. This implementation is similar to our previous one, except we have added type information to the variables and additionally swapped to using the `SVector` constructor, instead of the array notation.

```

function cross(a::SVector{3, T}, b::SVector{3, T}) where {T}
    SVector(
        (a[2]*b[3]-a[3]*b[2]),
        (a[3]*b[1]-a[1]*b[3]),
        (a[1]*b[2]-a[2]*b[1])
    )
end

```

Algorithm 6.9. A concrete implementation for a cross product when the inputs are two static vectors.

We can benchmark this function as well, by first converting our test vectors to static vectors:

```

julia> sa = SVector(a...);
julia> sb = SVector(b...);
julia> @btime cross($sa, $sb)
 2.274 ns (0 allocations: 0 bytes)
3-element StaticArraysCore.SVector{3, Float64} with indices SOneTo(3):
 0.18242146759034802
-0.428829354451039
 0.053461858289415076

```

We can see that this implementation is a lot faster! This is because there are zero allocations and the entire vector fits into the stack, and hence, is very likely to be in the cache. Since the cross product is just 6 multiplications and 3 subtractions, this is designed to be calculated locally. In this particular benchmark, this small switch led to around a  $11.5\times$  speed improvement.

One thing to remember about static vectors is that they are best used statically, and one should avoid mutating the elements. This is because a lot of the speed comes from their immutability. Instead, one should create a copy of the vector, with the value you want changed. Surprisingly, this is actually not that slow, as the copy exists on the stack, and not in the heap.

This method acts as a nice route into the next section, which is specifically dedicated to reducing heap allocations.

## 6.2 *Reducing Allocations*

By far, the easiest thing to look for during optimisation is the number and size of allocations made in your code, and seeing if this can be reduced. In this context, we refer to “allocation” as reserving space in memory (RAM) to store data. Most algorithms can be written in a way that allocates the space required at the beginning of the execution, and reuses this memory throughout the execution of the algorithm. Obviously, we as developers do not always know the amount of space in memory when writing the code, however, when it is known, there is no reason to take advantage of this.

### 6.2.1 *Appending vs Preallocating*

In languages like Python and MATLAB, it is very common to see developers appending information to their arrays, instead of *preallocating*. Appending to an array is when you add an element to an array, changing the size of the array. In some languages, there is distinct nomenclature around which collections of elements are static and which are dynamic (can have their size dynamically change). For example, C++, which reserves the term ‘array’ to refer to a statically sized array and ‘vector’ to refer to a dynamically sized collection. On the other hand, C# uses the term “List” for a resizable array, and arrays are strictly statically sized.



The reason for the distinction in many languages is made on purpose, to discourage the use of dynamically sized arrays whenever possible. When MATLAB notices code that resizes an array (i.e. appends a value to the array), it warns the user of the performance hit this process causes. Let's look at an example below of an algorithm which appends to an array:

```
numbers = rand(100)
cumulative_sum_array = []
total_sum = 0
for a in numbers
    total_sum += a
    push!(cumulative_sum_array, total_sum)
end
```

Every time we append to `cumulative_sum_array`, the computer needs to find a free contiguous block of memory with  $(n + 1) \times 64$  bits of space, in which a new array, consisting of the old array and the new appended value, can be stored. This also involves copying the information from the old array into a new location. If this process is done 100 times, then you will have to allocate that many times, and copy that many times, instead of just 100 reads and writes that this algorithm requires.

Many languages, such as C#, implement a buffer system which allocates more memory than is actually required for the array. When elements are added to the array, more of the buffer is used. When the buffer runs out, the new array will have space for all the elements and another, larger, buffer. Usually the size of the array grows by a factor of  $\sqrt{2}$  each time the buffer runs out.

The same algorithm can be written without appending to an array, since we know the size of the output:

```
numbers = rand(100)
# The 'similar' function allocates an array,
# which is the same size and type as the input.
result_arr = similar(numbers)
total_sum = zero(eltype(numbers))
for i in eachindex(result_arr, numbers)
    total_sum += numbers[i]
```

```

    result_arr[i] = total_sum
end

```

Notice that this only required a small number of changes to the algorithm, but this can have a huge impact on performance. Let us turn this algorithm into a function, and see the benchmark performance. The preallocated algorithm is implemented in Algorithm 6.10, while the appending algorithm is implemented in Algorithm 6.11.

```

function cumulative_sum_preallocated(numbers)
    results = similar(numbers)
    total_sum = zero(eltype(numbers))
    for i in eachindex(numbers)
        total_sum += numbers[i]
        results[i] = total_sum
    end
    return results
end

```

Algorithm 6.10. A simple implementation of the cumulative sum calculation for an array of points, which uses preallocation instead of a dynamically changing array.

```

function cumulative_sum_appending(numbers)
    results = (eltype(numbers))[]
    total_sum = zero(eltype(numbers))
    for i in eachindex(numbers)
        total_sum += numbers[i]
        push!(results, total_sum)
    end
    return results
end

```

Algorithm 6.11. A simple implementation of the cumulative sum calculation for an array of points, which appends to an array instead of preallocating.

Looking at Figure 6.3, remembering that it is plotting in log-log scale, we can see that the lines are roughly parallel, meaning that they have the same computational complexity (in this case it is linear). However, there is a constant difference between the lines, showing that the preallocated code is around one order of magnitude faster for every input of  $n$ . The constants that predict the speed difference are left out in Big  $\mathcal{O}$  notation, since they are merely implementation details. This example clearly shows they are of great practical importance, and show that computational complexity is not a complete toolkit when analysing algorithms.

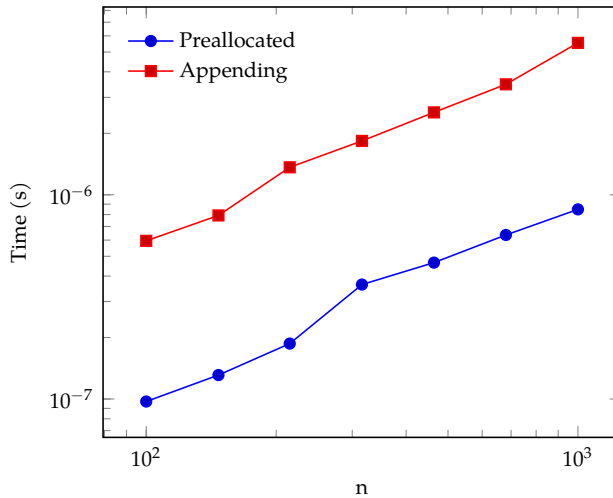


Figure 6.3. Time taken when using `cumulative_sum_preallocated` and `cumulative_sum_appending` for several values of  $n$ . The time taken is measured several times for each value of  $n$  and the minimum time is taken, which dramatically **under-estimates** the time taken in the allocating function. It is important to notice the *log-log* scale used for plotting. We can see that the preallocated version is around an order of magnitude faster for all values of  $n$ , in the best case scenario of the **allocating** version.

**Note:** Many programming languages implement variable sized vectors in a more efficient way. When the vector is appended to, the runtime allocates  $n\sqrt{2}$  spaces, and retains this additional space, in case the vector is appended to in the future, in which case, there is space to allocate more values without needing to copy the array. This significantly improves the performance of vectors, while having them not allocate too much memory.

### 6.2.2 Using a cache

Many parts of your code will involve calculating a formula. For readability, many programmers will separate out different terms in an equation into separate variables and then sum them together at the end. This has benefits for the programmer, as the code becomes more readable and maintainable, but it can introduce pernicious performance bugs in the process. Let's take a look at the following example of coding up Equation (6.3):

$$y(x) = \frac{5x^5 \sin(x^2) + 20}{\exp(-4x) - x^2} \quad (6.3)$$


Suppose that we had to write a function, which calculate a series of  $y$  values for each element of an array  $x$ . An initial implementation could look like Algorithm 6.12.

```
function example_equation_no_cache(x)
    numerator = 5 .* x .^ 5 .* sin.(x.^2) .+ 20
    denominator = exp.(-4 .* x) .- x .^ 2
    y = numerator ./ denominator
    return y
end
```

Algorithm 6.12. A simple function to represent a vectorised version of Equation (6.3).

In this first implementation, we have split up the equation into two variables, one for the numerator and one for the denominator of the fraction. First of all, notice that we are using Julia’s broadcasting notion, introduced in Section 4.3.1. Let’s benchmark this function with some data:

```
julia> x = rand(8192);
julia> @benchmark example_equation_no_cache($x)
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
Range (min ... max):  99.862 μs ... 1.648 ms  | GC (min ... max): 0.00% ... 92.05%
Time (median):        102.988 μs             | GC (median): 0.00%
Time (mean ± σ):     105.738 μs ± 59.301 μs  | GC (mean ± σ): 2.13% ± 3.56%
```




99.9 μs                      Histogram: frequency by time                      110 μs <

Memory estimate: 192.14 KiB, allocs estimate: 6.

Notice that there are many more allocations than actually required for this method, which should only allocate as much memory as is stored in  $x$ . Not only is this implementation wasteful in terms of memory, but it is also slower since the computer has to find space to store these numbers. We can modify the algorithm to store intermediate values directly in the  $y$  array so that we only allocate once. This is shown in Algorithm 6.13.

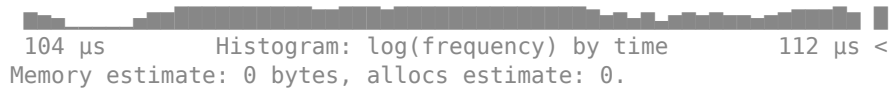
```
julia> y = similar(x);
julia> @benchmark example_equation_cache!($y, $x)
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
Range (min ... max):  104.251 μs ... 180.727 μs  | GC (min ... max): 0.00% ... 0.00%
Time (median):        106.545 μs                | GC (median): 0.00%
Time (mean ± σ):     106.837 μs ± 1.753 μs      | GC (mean ± σ): 0.00% ± 0.00%
```



```

function example_equation_cache!(y, x)
    # Set y to the value of the numerator
    y .= 5 .* x .^ 5 .* sin.(x.^2) .+ 20
    # Divide out the denominator
    y ./= exp.(-4 .* x) .- x .^ 2
    return y
end

```



Algorithm 6.13. A function to represent a vectorised version of Equation (6.3), implemented with a cache variable, so that intermediate results are not stored. We pass in the cache, relying on the caller to correctly initialise the result. Notice the use of the ! notation (called the “bang” notation), which is a convention to let the users of the function know the function mutates the input, usually the first argument.

While the speed difference between these two implementations is not that large on the lower end, the non-cache version has a significant tail on the timings. Using this within a hot loop can lead to a huge performance degradation.

One major benefit to using broadcasting notation is that Julia’s compiler can *fuse* broadcasted operations together, so that the intermediate results do not allocate any memory. Additionally, we can specify where the results of these operations should be stored. This is a significant advantage over Python, where each intermediate result in an equation must allocate memory, since the equation is interpreted and the operations cannot be fused together.

There is an obvious disadvantage to this approach - one need keep track of all the cache variables, and the code can become verbose very quickly. For this reason, there are a few rules which can be used to make this process simpler:

- Write out formulae and equations in scalar functions which are in turn, broadcasted so that one need not deal with adding `.` before each operation, and so that one avoids unnecessary allocation when trying to refactor code. This is only usually possible with `map` operations.
- Frequently use broadcasting, so that the operation can be fused to avoid intermediate results.
- Identify whether using a cache would have a significant performance difference in the application. This usually involves functions that are called in a loop many times. If the cache is only being used once (or only a few times) throughout the lifetime of your program, this is probably a sign that you do not need to cache the variable.

- If many cache variables are required, consider using a struct to group all the cache variables together and then pass this cache to the functions.
- If several sizes of cache are required (all of the same type), but only one cache is needed at a time, then one can create a single variable which is of the largest size required, and then create views of this variable to act as the cache for cases where smaller values are needed.
- If the same sized cache variable is required many times, consider using a *functor* (introduced in Section 4.3.5) to wrap the cache away so that one need not keep track of it. Be careful when using functors (or closures) in multithreaded, as multiple CPUs may be accessing the same cached memory at the same causing **race conditions**, discussed in more detail later in this book.

On this last point, a *functor*, is a design pattern which stores local variables inside a struct, with an override for calling the struct, such that one need not pass the stored variables to the function. This is a more performant way of creating a closure, giving you control over the types of the variables - which will help avoid type instability<sup>3</sup>.

As one may come across closures in their code, an example of creating a *closure* is given in Algorithm 6.14.

```
function create_cached_closure_fn(cache_type, cache_size, fn)
    cache = zeros(cache_type, cache_size)
    cached_fn(x) = fn(cache, x)
    return cached_fn
end
```

We have to create this function in order to use it:

```
fn = create_cached_closure_fn(eltype(x), length(x), example_equation_cache!);
y = fn(x); # Uses the cache created in the line above (non-allocating)
```

It must be stressed that this function is no longer thread-safe and cannot be used in parallel. Closures can still work in parallel, but require some careful consideration.

<sup>3</sup> See Section 6.7 later in this chapter for more details.

Algorithm 6.14. An example pattern for creating a closure to hide a cache variable in a function definition. Closures should be avoided for performance critical code, as they can often result in poor type inference at compile time. Functors are usually much better than closures for this purpose.

Usually, it is better to provide two methods, one with a “bang” which uses some cached memory, and a second function, without a “bang” which creates the cache to pass to the first method; this way, one can have a single implementation of your algorithm which is memory efficient, while also providing an easy-to-use function that does not require the user to keep track of cache variables, at the cost of some performance. This is a good trade off, particularly when the function will only be called once and a cache will not improve performance.

Note that closures are implemented much like functors behind the scenes. Defining your functor and specifying the variables is much preferred as it is clear what variables are being captured, and it is easier to guarantee better performance due to type inference.

### 6.3 Memoisation

Sometimes, we have pure functions that will return the exact same output for a given input. Instead of having to re-calculate these values again and again, we may want to store these values so that they returned quickly, instead of having to recompute the result again. This only makes sense when we are calling the same method many times with the same inputs, and we have the memory resources to keep track of the results.

This technique is known as memoisation, which *caches* the result of a function call based on the input arguments. Let’s take a very simple example of calculating a number in the Fibonacci sequence: We can create a *functor* to combine the

```
function fib(n)
    if n==1
        return 1
    end
    return n * fib(n-1)
end
```

Algorithm 6.15. An algorithm to calculate the  $n^{\text{th}}$  Fibonacci number via recursion.

functor and the storage of the results in a dictionary: We can now create this functor and use it to calculate our values:

```
julia> cached_fib_fn = CachedFib();
julia> cached_fib_fn(4)
24
```

```

struct CachedFib
    cache::Dict{Int, Int}
end
# Create an argument-less constructor
CachedFib() = CachedFib(Dict{Int, Int}())
function (f::CachedFib)(n)
    if n == 1
        return 1
    end
    cache = f.cache
    # Check if the cache contains the result
    if haskey(cache, n)
        return cache[n]
    end
    # If not, actually calculate the result
    result = n * f(n-1)
    # Store the result in the cache
    cache[n] = result
    # Return the result
    return result
end

```

Algorithm 6.16. An algorithm to calculate the  $n^{\text{th}}$  Fibonacci number via recursion.

Not that the backing cache does not have to be a dictionary, or even stored in main memory, one can just as easily use a disk cache as well for results that take a long time to run. This pattern can be very useful for long-running calculations that may be interrupted, such as a HPC job. Using the disk as a cache can easily allow you to recover from stops, avoiding having to recalculate everything from the beginning.

## 6.4 Vectorising vs Loops

Even though we have already covered this in some detail, we must drive home the point of comparing vectorised code and using a loop.

If you have come from MATLAB and Python, the gospel of optimisation is to **vectorise** your code. Let's first break down what this means. Take the following two implementations of the same method:

```

function add!(c, a, b)
    @inbounds for i = 1:length(a)
        c[i] = a[i] + b[i]
    end

```



```

    end
    nothing!
end
# OR:
function add!(c, a, b)
    c .= a .+ b
    nothing
end

```

These two implementations are essentially the same in Julia, except that the first implementation does not check the bounds of the inputs before the loop, which should be done but is removed for simplicity. The first function, containing the `for` loop, would not be considered to be vectorised, while the second one is the vectorised form. Vectorising your code, simply means writing your code without these `for` loops, and in terms of only array operations.

In Python or MATLAB, vectorising your code makes a huge difference to performance, despite the actual contents of the computation being essentially the same. Why is there a difference? Well, the difference is because Python and MATLAB rely on array calculations from a library written in a different, faster language<sup>4</sup>. For example, Python extensively uses *numpy*, which is mostly written in C, and only has Python bindings. When you write  $C = A + B$ , when  $A$  and  $B$  are *numpy* arrays, then this addition happens “inside C”, and not Python, which is much slower as it is not compiled. Vectorising then, just means offloading most of the computation into a lower level library (such as *numpy*), having the program spend as much time as possible in the faster language and avoiding doing any bulk of the calculation in the interpreted language.

<sup>4</sup>Or rather, rely on highly optimised routines which are compiled in a language like C or Fortran.

So what about Julia? Well Julia is a compiled language, and hence is a fast language and so does not need to make this distinction between vectorisation and loops and both operations compile down to practically the same machine code. For this reason, both implementations are practically the same in terms of performance. This means you are free to write loops if it is easier for your code.

However, there are still reasons for preferring vectorisation in Julia. The first reason is readability and maintainability. The broadcast notation (the vectorised notation) is very powerful and can make the code look much cleaner, and hence easier to maintain over time due to the simplicity. On the other hand, even though running native code has practically negligible differences between vector and loop form, if one would like to extend one’s algorithm to run on the GPU, then the vectorised form can be easily (and automatically) translated into efficient

GPU kernels. Usually, this can be done without having to change a single line of the source code, only changing the types of the arrays. This means you can have the same code executing on both the CPU and the GPU, which is incredibly powerful. This expressive power massively reduces developer time (since there is only one implementation of an algorithm), while also reducing the effort spent on testing. We will cover writing GPU code in more detail later.

## 6.5 Views

We have seen that allocating memory on the heap can negatively impact performance, but what if we already have allocated memory, and simply want a convenient way to look at a part of an array, without increasing code complexity. This is where the `view` comes in handy. If you have used Python or MATLAB, you will be familiar with slicing syntax:

```
julia> a = rand(3,3)
3×3 Matrix{Float64}:
 0.398328  0.276889  0.282382
 0.783487  0.979941  0.328688
 0.218223  0.141403  0.113252
julia> b = a[1:2, 1:2]
2×2 Matrix{Float64}:
 0.398328  0.276889
 0.783487  0.979941
```

This syntax, allows us to take a section of an array and manipulate it. The only issue is that we have just created a copy of that part of the array! To prove this, let's mutate part of `b` and then look at the values of `a`:

```
julia> b .= 0
2×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
julia> a
3×3 Matrix{Float64}:
 0.398328  0.276889  0.282382
 0.783487  0.979941  0.328688
 0.218223  0.141403  0.113252
```

Notice that `a` is unchanged. This is a good behaviour by default, as if any other behaviour were implemented, this would lead to countless bugs. It is important to remember that simply assigning a variable to an array, will assign by *reference* and hence not make a copy, since the array variable is simply a pointer to the start of the array:

```
julia> c = a; # c and a point to the same memory now
```

However, if we want to mutate part of an array, while not copying the memory, we can just use a `view`. There is a handy macro that does this for us. Take the first example using the `@views` macro<sup>5</sup>:

```
julia> b = @views a[1:2, 1:2]
2×2 view(::Matrix{Float64}, 1:2, 1:2) with eltype Float64:
 0.398328  0.276889
 0.783487  0.979941
```

<sup>5</sup> There is also the `@view` macro, but `@views` will convert all array slicing into a view for an entire expression.

Now, `b` points to the same four elements in memory as the top-left square of `a`. To show this, let's manipulate `b` and check `a`:

```
julia> b .= 0
2×2 view(::Matrix{Float64}, 1:2, 1:2) with eltype Float64:
 0.0  0.0
 0.0  0.0
julia> a
3×3 Matrix{Float64}:
 0.0      0.0      0.282382
 0.0      0.0      0.328688
 0.218223  0.141403  0.113252
```

Notice that we have propagated this information back to `a`. This can be very useful when one would like to avoid making copies of arrays, but needs to access only part of the array.

## 6.6 Speed of Operations

This section is based off information from <http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/>, whose article is very well written and should be viewed. They provide the infographic in Figure 6.4.

Taking a look at this graph, we can see that the basic operations, such as addition, multiplication and even memory writes are very fast. A lot of the details

**Note:** One should still be very careful, since the memory layout has not changed, but the indexing scheme has changed. This means that even though one may loop through the view using the correct indexing scheme, it is likely that the memory is not contiguous and hence *can* have severe performance penalties for using a view. In situations where memory locality is more impactful than allocating memory, a copy may be worth making. Again, this trade-off comes down to benchmarking, necessary for making an informed decision.

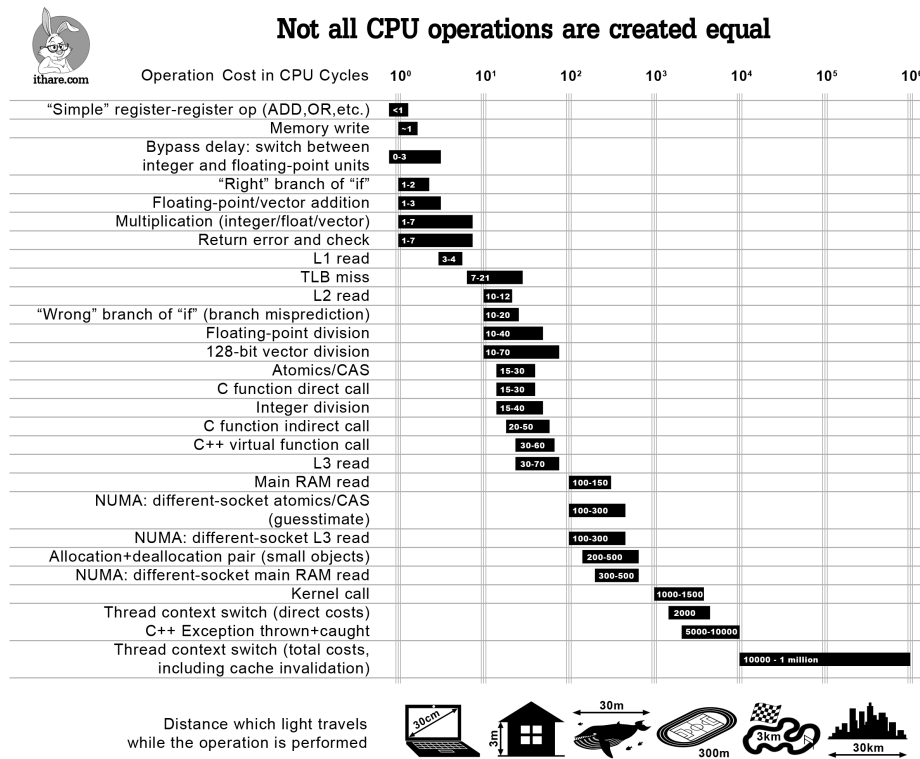


Figure 6.4. This table is an approximate comparison of the speeds of different operations on a typical CPU. There is a lot of uncertainty here, but it can be used to get a few heuristics for which operations are slow and which are fast.

of this graph are too complex to see, but one heuristic to see is the different speeds of cache misses and allocation. One can see that allocation tends to be orders of magnitude slower than simple cache read and writes, or simple numeric calculations. Additionally, one can see that branching operations (if statements) can be very costly when mispredicted. Modern CPUs often use branch prediction to start execution one of the branches of an if statement while the statement is checked. If the CPU gets the prediction wrong, then it has to backtrack which costs many cycles. This is one of the reason you see many people opting for “branchless” programming styles, where control loops are kept to a minimum. This usually involves using boolean numerics to set a value to zero if something is false and one otherwise and then summing both results together. Branchless programming can have significant performance improvements if done correctly.

## 6.7 Type Stability

While Julia is a dynamic language, it can be very important for the compiler to know the **type** of the variables being used, so that it can specialise on it. We have been alluding to this throughout the book so far, but getting this wrong may have huge performance impacts. Let’s get a concrete example:

```
function get_fibonacci(n)
    a = 1
    b = 1
    fibonacci_nums = []
    push!(fibonacci_nums, a)
    push!(fibonacci_nums, b)
    for i = 1:n-2
        a, b = b, a+b
        push!(fibonacci_nums, b)
    end
    return fibonacci_nums
end
```

Algorithm 6.17. An example of calculating the Fibonacci sequence by appending to an array.

This calculates the first *n* Fibonacci numbers and stores the results in an array. Here, we get the result of the vector being of type `Vector{Any}`. Let’s see what effect this has on performance:

```

julia> fibonacci_nums = get_fibonacci(50);
julia> typeof(fibonacci_nums)
Vector{Any} (alias for Array{Any, 1})
julia> @btime sum($fibonacci_nums)
 655.310 ns (38 allocations: 608 bytes)
32951280098

```

We can see that this sum had many allocations, and took around 1 microsecond. If we simply make another array with the correct type, let us see the difference in performance:

```

julia> fibonacci_nums_with_type = [f for f in fibonacci_nums];
julia> typeof(fibonacci_nums_with_type)
Vector{Int64} (alias for Array{Int64, 1})
julia> @btime sum($fibonacci_nums_with_type)
 5.951 ns (0 allocations: 0 bytes)
32951280098

```

Notice that the allocations were reduced by orders of magnitude. Additionally, the runtime of these algorithms went from around 1.2 microseconds, to around 10 nanoseconds. In this example, when Julia knew the type of the variables, the performance increased by a factor of almost 120. The reason why performance was so low for the first method, is that Julia had to check the type of each individual element in the array to make sure that it was the right type. In order to store the array, Julia “boxes” the elements into the `Any` type, instead of storing the raw binary in a contiguous array. Each element of the array is actually just a pointer which references another location in memory, which could store information of any time (and hence any size), and so Julia is forced into storing pointers, instead of the raw values, as it does not know ahead of time how much space to reserve in memory for each element. This brings many other issues, such as the data not being stored together predictably, and hence entire blocks of memory cannot be copied into the CPU cache, reducing the need to access memory. Not only does the runtime have to manually check the type of each element in the array (unboxing the elements), but it also has to find them one by one in memory by dereferencing each pointer, sacrificing any opportunity to use the CPUs cache.

Why did this happen in the first place? This is one of the consequences of appending to an array, starting from an empty array. It is not obvious what the type of the array is. It is possible to initialise an empty array with a given type. The correction to the code above is the following:

```
fibonacci_nums = (typeof(a))[]
```

Otherwise, one could just preallocate an array with the appropriate type, which would also give the array a concrete type.

This leads us into discussing type stability. If the compiler cannot trace the types of the variables used throughout a function (each variable having a known concrete type at compile time), then the function is said to be *type unstable*. This means that the compiler will have to perform additional checks on variables with unknown types at runtime, causes performance hits. These checks are not only costly, but not knowing the types also removes the ability for the compiler to specialise on the types beforehand.

An analogy is having an optimal plan before performing a task, compared to “winging it” on the fly, depending on the information you get. While one will most likely perform the same subtasks at the same speed, having a plan can save a lot of time and energy when moving between tasks, and knowing what resources you will need ahead of time. Additionally, you will not need to perform basic checks to make sure you are doing the right thing, as you will have organised everything to be in the right place when you plan ahead.

Let’s take a look at an example of a type unstable function, shown in Algorithm 6.18 below.

```
function example_type_unstable_fn(x)
    s = 0
    for x_i in x
        s += x_i
    end
    s
end
```

Algorithm 6.18. An example of a type unstable function, which sums all the elements in the array  $x$ .

This looks like a normal implementation. We can even run it and benchmark to see if it works:

```
julia> x = collect(1:100);
julia> @btime example_type_unstable_fn($x)
5.751 ns (0 allocations: 0 bytes)
5050
```

```
function calc_summary(rows)
    return sum(example_type_unstable_fn, rows)
end
```

Algorithm 6.19. Using the previous type unstable function within a calculation.

There are actually no issues with this code so far that we can see, but let's show an example where some performance issues come out.

Let's run this with some random test data:

```
julia> rows = [rand(rand(0:10)) for _ in 1:100];
julia> @btime calc_summary($rows)
 259.697 ns (0 allocations: 0 bytes)
247.12173352279768
```

We see that the performance of this function is actually pretty good, and there are even no allocations. However, there is actually a performance bug which comes to light when we use the `@code_warntype` macro provided by the base library:

```
julia> @code_warntype calc_summary(rows)
MethodInstance for Main.WeaveSandBox0.calc_summary(::Vector{Vector{Float64}})
  from calc_summary(rows) in Main.WeaveSandBox0 at /nfsshare/home/ppyjml3/dev/fictional-computing-machine
Arguments
 #self#::Core.Const(Main.WeaveSandBox0.calc_summary)
 rows::Vector{Vector{Float64}}
Body::UNION{FLOAT64, INT64}
1 - %1 = Main.WeaveSandBox0.sum(Main.WeaveSandBox0.example_type_unstable_fn, rows)::UNION{FLOAT64, INT64}
└─ return %1
```

Look for the sections that show the return type to be a `Union{Float64, Int64}`. Unfortunately, this book does not show highlighting, however in your console the parts with `Union` should be highlighted red so they can be spotted easily. This means that the compiler cannot work out whether the return type should be a float or an integer. A union type symbolises that the type can any of the types in the union. This seems very strange, as we can see the type of the inputs is fixed:

```
julia> typeof(rows)
Vector{Vector{Float64}} (alias for Array{Array{Float64, 1}, 1})
```

We know we are only dealing with floats, however, the return type potentially could be an integer. This is due to Algorithm 6.19 being type unstable, which propagates out to our `calc_summary` function. We can check this by running `@code_warntype` again:



```

julia> @code_warntype example_type_unstable_fn((rows[1]))
MethodInstance for Main.WeaveSandBox0.example_type_unstable_fn(::Vector{Float64})
  from example_type_unstable_fn(x) in Main.WeaveSandBox0 at /nfsshare/home/ppyjml3/dev/fictional-comp
Arguments
  #self#::Core.Const(Main.WeaveSandBox0.example_type_unstable_fn)
  x::Vector{Float64}
Locals
  @_3::UNION{NOTHING, TUPLE{FLOAT64, INT64}}
  s::UNION{FLOAT64, INT64}
  x_i::Float64
Body::UNION{FLOAT64, INT64}
1 -      (s = 0)
   |
   | %2 = x::Vector{Float64}
   |   (@_3 = Base.iterate(%2))
   | %4 = (@_3 === nothing)::Bool
   | %5 = Base.not_int(%4)::Bool
   |     goto #4 if not %5
2 ... %7 = @_3::Tuple{Float64, Int64}
   |     (x_i = Core.getfield(%7, 1))
   |     %9 = Core.getfield(%7, 2)::Int64
   |         (s = s + x_i)
   |         (@_3 = Base.iterate(%2, %9))
   | %12 = (@_3 === nothing)::Bool
   | %13 = Base.not_int(%12)::Bool
   |     goto #4 if not %13
3 -      goto #2
4 ...      return s

```

We can see that the variable `s` is type unstable (i.e. has a union type). This is because the input array has the possibility of being empty. If this array is empty then the loop gets skipped and the function will return 0, which is an integer. If the array is not empty, then the first assignment to `s` will promote the type to a floating point number, making `s` now a float. This is one of the issues with a dynamically typed language, the types of the variables are allowed to change. We can fix this error and see how it affects the benchmark.

In this example, the fix is very easy, since we can use the `zero` and `eltype` functions. These are usually compiled away into constants when the function is compiled, since Julia knows the type of the input at runtime due to multiple dispatch and Just-in-Time compilation. The fixed example is shown in Algorithm 6.20.

```
function example_type_stable_fn(x)
    s = zero(eltype(x))
    for x_i in x
        s += x_i
    end
    s
end
```

Algorithm 6.20. An example of a type unstable function, which sums all the elements in the array `x`.

First, we notice that any red flags from the `code_warntype` macro (the union types) have disappeared:


```
julia> @code_warntype example_type_stable_fn(x)
MethodInstance for Main.WeaveSandBox0.example_type_stable_fn(::Vector{Int64})
 from example_type_stable_fn(x) in Main.WeaveSandBox0 at /nfsshare/home/ppyjml3/dev/fictional-computing-
Arguments
 #self#::Core.Const(Main.WeaveSandBox0.example_type_stable_fn)
 x::Vector{Int64}
Locals
 @_3::UNION{NOTHING, TUPLE{INT64, INT64}}
 s::Int64
 x_i::Int64
Body::Int64
1 - %1 = Main.WeaveSandBox0.eltype(x)::Core.Const{Int64}
   |   (s = Main.WeaveSandBox0.zero(%1))
   |   %3 = x::Vector{Int64}
   |   |   (@_3 = Base.iterate(%3))
   |   |   %5 = (@_3 === nothing)::Bool
   |   |   |   %6 = Base.not_int(%5)::Bool
   |   |   |   goto #4 if not %6
2 ... %8 = @_3::Tuple{Int64, Int64}
   |   |   (x_i = Core.getfield(%8, 1))
   |   |   %10 = Core.getfield(%8, 2)::Int64
   |   |   |   (s = s + x_i)
   |   |   |   (@_3 = Base.iterate(%3, %10))
   |   |   |   %13 = (@_3 === nothing)::Bool
   |   |   |   |   %14 = Base.not_int(%13)::Bool
   |   |   |   |   goto #4 if not %14
3 -   |   goto #2
4 ...   return s
```

Now we can propagate this fix onto the outer function.  
Now, let's benchmark:

```
function calc_summary_stable(rows)
    return sum(example_type_stable_fn, rows)
end
```


Algorithm 6.21. Using the fixed type stable function to perform the map reduce operation.

```
julia> @benchmark calc_summary($rows)
BenchmarkTools.Trial: 10000 samples with 337 evaluations.
Range (min ... max): 263.831 ns ... 441.976 ns | GC (min ... max): 0.00% ... 0.00%
Time (median):      281.015 ns | GC (median): 0.00%
Time (mean ± σ):    282.294 ns ± 5.236 ns | GC (mean ± σ): 0.00% ± 0.00%
```



```
264 ns Histogram: frequency by time 297 ns <
Memory estimate: 0 bytes, allocs estimate: 0.
```

```
julia> @benchmark calc_summary_stable($rows)
BenchmarkTools.Trial: 10000 samples with 343 evaluations.
Range (min ... max): 254.834 ns ... 373.956 ns | GC (min ... max): 0.00% ... 0.00%
Time (median):      263.653 ns | GC (median): 0.00%
Time (mean ± σ):    264.754 ns ± 4.822 ns | GC (mean ± σ): 0.00% ± 0.00%
```



```
255 ns Histogram: frequency by time 286 ns <
Memory estimate: 0 bytes, allocs estimate: 0.
```

This performance difference is not huge on the most recent versions of Julia, but can be significant if the type instabilities are large unions, or worse, of type `Any`. Recent work on the Julia compiler has introduced **union splitting** which drastically improves the performance of type unstable code provided the unions are small.

**Exercise 6.1.** Install a previous, older, version of Julia (via `juliaup`) and benchmark how these implementations differ over Julia development. Show these changes graphically over time.

Making our code type stable also ensures our code is very generic and reusable across many types of numbers. In general, it is a good idea to avoid constants in your code, and instead use the `zero`, `one`, `eltype` and `typeof` functions to construct your constants. One will frequently see this practice used throughout this book, for this reason of performance.

Type stability is much broader topic with many nuances. If you are experiencing poor performance, it is often a good idea to check whether your functions are actually type stable, using the `code_warntype` macro.

### 6.7.1 Barriers

There are some occasions when you cannot predict the type being returned from a function, which leads to type instability. There are still some tools one can use to mitigate the performance hits of this. The antidote to this problem is to break up larger functions into several smaller functions, which act as type barriers so that performance critical parts of your code are not affected, while giving you the required flexibility.

Let's take the example of using an array to store parameters which are used within a function:

```
function params_in_array_test(x)
    a = 0
    b = 1
    for i = 1:100
        c = x[1] + a
        d = x[2] / b
        a, b = c, d
    end
    return a, b
end
```

This function works with the input `x=[2.0, 1]`. However, this is because the type of the input is stable:

```
julia> x = [2.0, 1]
2-element Vector{Float64}:
 2.0
 1.0
julia> @btime params_in_array_test($x)
419.548 ns (0 allocations: 0 bytes)
(200.0, 1.0)
```

The compiler has “promoted” the integer `1` to a floating point number, so that the numbers in the array can match. Now, let's imagine that we add another parameter to this array, which is of a different type:

```

julia> x = [2.0, 1, "third"]
3-element Vector{Any}:
 2.0
  1
 "third"
julia> @btime params_in_array_test($x)
 3.425 μs (201 allocations: 3.16 KiB)
(200.0, 1.0)

```

Now, even though the two parts of the `x` array are unchanged, all we did was add a third parameter, we suddenly have around a 17 times decrease in speed! This is because `x` now has to be an array of type `Any`, which means that the compiler no longer knows how many bits to reserve when processing elements of the array. It first has to check how much space is needed, and then allocate that space. This must be done on the **heap**, because the size is unknown at compile time. Having to chase pointers to find the actual data makes this code incredibly slow.

We can remedy this hit, without changing the input types, or the function signature at all. We can do this by refactoring our function and using a barrier function:

```

function _params_in_array_test_loop(x1, x2, a, b)
    for i = 1:100
        c = x1+a
        d = x2 / b
        a, b = c, d
    end
    return a, b
end
function params_in_array_test_with_barrier(x)
    a = 0
    b = 1
    return _params_in_array_test_loop(x[1], x[2], a, b)
end

```

Algorithm 6.22. This is an example of a *barrier* function, which refactors the inner loop of the main body into a separate function with named arguments, instead of indexing the array `x`. This makes sure that the bulk of the processing can occur in a type stable manner, avoiding the performance hit when it is not possible to completely remove type instabilities.

This re-factor amounted to simply a copy and paste and a few renames. However, if we now benchmark the performance on the same input as before:

```

julia> @btime params_in_array_test_with_barrier($x)
 460.680 ns (1 allocation: 32 bytes)
(200.0, 1.0)

```

This is not a perfect solution, as we had to perform a **dynamic dispatch**. A dynamic dispatch happens when the compiler cannot infer which method of a function should be called based on the input types of a function. When `params_in_array_test_with_barrier` is compiled, the inner function call cannot be compiled as the types of the inputs are not known. Instead, the compiler will dynamically look up the most specific method at run-time (i.e. dynamically dispatching to the correct method). Dynamic dispatch can be a useful tool when you have no other choice, but if occurring in the hot loop of your code, can be a source of performance loss. Note that the allocation in the benchmark is due to not knowing the return type ahead of time and hence needing to allocate the result on the heap.

Looking at Section 6.6, it may be tempting to remove functions calls, as it shows that they can be relatively expensive. However, we do not need to worry about in Julia, as the compiler can always inline the function if it will improve performance, which increases runtime performance at the cost of a longer compile time. The lesson to take from this is that it is more important to write readable and maintainable code, than to worry about performance, since Julia's compiler will usually be able to fix most performance problems that arise from refactoring your code into smaller functions.

### 6.7.2 *Type Inference and Branching*

Taking a look at Section 6.6, branching (using an `if` statement) can be expensive if the compiler mispredicts the correct branch. However, if one writes branches conditions as a function of compiler constants - which usually include the types of variables - one can always predict the correct branch, and even compile away the branch. As a quick example:

```
function branch_predict_with_types(x)
    if x isa Int
        return x % 2
    elseif x isa AbstractFloat
        return (x-1.0) % 2
    else
        return Nothing
    end
end
```

Algorithm 6.23. An example function that will compile away the branches as they can be evaluated at compile time.

The above function would normally seem expensive as it has to do a branching operation. However, Julia will compile away the branch information, since at compile time it usually can infer the type of the input `x`. Do not be afraid to branch based on type, as this will have little to no effect on performance<sup>6</sup>. However, usually this type of branching is better done using *multiple dispatch*, so consider alternative patterns.

This type inference at compile time is also why using the `typeof` function involves little penalty.

<sup>6</sup> Provided that your function is type stable and the type is actually known at compile time and does not need to be dynamically checked.

### 6.7.3 Using composite types

In Julia, there are no classes. However, one can define one's own composite types. By a composite type, we specifically mean a type that is made up of a composition of other types. In Julia, we call this object a `struct`. By default, structs are immutable.

Structs are a common place that people introduce type instability, and let us explore why. Take the example of defining one's own complex number:

```
struct MyComplex
    real
    imag
end
```

One can even start to define some methods on this type to make it useful:

```
+(a::MyComplex, b::MyComplex) = MyComplex(a.real + b.real, a.imag + b.imag)
+(a::MyComplex, b::Number) = MyComplex(a.real + b, a.imag)
*(a::MyComplex, b::Number) = MyComplex(a.real * b, a.imag * b)
# ... add all other operations
```

However, this type is not type stable. When we defined the type previously, we implicitly labelled both the `real` and `imag` variables as type `Any`:

```
struct MyComplex
    real::Any
    imag::Any
end
```

This now means that this type must be heap allocated, since the variables can be any type and hence any size. Additionally, the type of the variables cannot be inferred at compile time and so this object will have terrible performance. We can fix this, by specifying a type:

```

struct MyComplex
    real::Float64
    imag::Float64
end

```

However now, this type will only work with type `Float64`. Instead, we can use a feature of Julia called *generics*, which allows us to specify the type of the parameters inside a struct:

```

struct MyComplex{T}
    real::T
    imag::T
end

```

The main difference here, is that the real and imaginary parts are forced to be the same type. Subtly, this has created multiple definitions of the struct, one for each type of the real and imaginary parts. These types can still be of type `Any` though, and should likely be restricted. This can be done with the following syntax:

```

struct MyComplex{T<:Number}
    real::T
    imag::T
end

```

If one writes the method functions on this type generally, it will not matter if you have two complex numbers, one with integers and the other with floats, since the compiler should know how to promote the types.

If you have a performance critical struct, make sure that the types are well-defined. If the struct is a collection of numeric or, more generally, “`isbitstype`” types, the resulting composition will also be an “`isbitstype`” type and hence allowed to be stack allocated. Additionally, making a struct mutable, will make the object heap allocated, which can hurt performance.

## 6.8 Constant Propagation

We have already seen that some code can get compiled away if all the constants are known at compile time. Take the following example:



```

julia> compiled_fn() = sum(1:1000);
julia> @code_typed compiled_fn()
CodeInfo(
  1 -      return 500500
) => Int64

```

One can see that the actual code simply returns the constant value which was calculated. The code never actually performs the sum at runtime, since it can be precomputed at compile time. This opens up a whole new world of possibilities. If you can give the compiler information about constants during compile time, it can propagate that information forwards to avoid costly computations down the line.

Let's take the following example, taking from a real world problem involving cellular automata, shown in Algorithm 6.24. This code is quite complicated, per-

```

function dynamics_rule_150_no_val(u, N)
    unit = one(typeof(u))
    mask = ~(~zero(typeof(u)) << N)
    u_left = (u << 1) | ((u & (unit << (N-1))) >> (N-1))
    u_right = (u >> 1) | ((u & unit) << (N-1))
    return (xor(xor(u_left, u), u_right) & mask)
end

```

Algorithm 6.24. An example taking from a program that calculates cellular automata updates to a 1D array of spin  $\frac{1}{2}$  particles.

forming many bitwise operations on the input  $u$ . The specifics of the calculation are not very important here, you should just notice that the `mask` variable calculated using the variable  $N$ . This is one additional calculation that does not need to be performed if  $N$  is constant. However, we want our code to be able to work with different values of  $N$ . We want to introduce the variable  $N$  as a compile-time constant, but allowing the function to be used with different values of  $N$ .

<sup>7</sup>  $N$  is the number of particles in the spin chain.

How can we achieve this? The answer is by taking advantage of Julia's type system. Since the type information is available to the compiler, we can simply insert the value of  $N$  into some type information. There is a special type in Julia called `Val` which allows us to insert data into the type information. For example we can construct an object with the number 5 in the type:

```

julia> Val(5)
Val{5}()

```

```

function dynamics_rule_150_with_val(u, ::Val{N}) where {N}
    unit = one(typeof(u))
    mask = ~(~zero(typeof(u)) << N)
    u_left = (u << 1) | ((u & (unit << (N-1))) >> (N-1))
    u_right = (u >> 1) | ((u & unit) << (N-1))
    return (xor(xor(u_left, u), u_right) & mask)
end

```

Algorithm 6.25. An example taking from a program that calculates cellular automata updates to a 1D array of spin  $\frac{1}{2}$  particles, but using a compile time constant  $N$ , provided in the type information.


Notice the  $\{5\}$  which shows this is the type information. We can extract that information using a generic function definition:

We can benchmark the two functions to see the difference:

```

julia> u = rand{Int}
4763098327049107497
julia> N = 32
32
julia> @benchmark dynamics_rule_150_no_val($u, $N)
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
Range (min ... max):  3.416 ns ... 7.404 ns      GC (min ... max): 0.00% ... 0.00%
Time (median):        3.476 ns                   GC (median):      0.00%
Time (mean ± σ):      3.486 ns ± 0.137 ns         GC (mean ± σ):   0.00% ± 0.00%


```



```

3.42 ns      Histogram: frequency by time      4.26 ns <
Memory estimate: 0 bytes, allocs estimate: 0.
julia> @benchmark dynamics_rule_150_with_val($u, $(Val{N}))
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
Range (min ... max):  2.043 ns ... 10.260 ns     GC (min ... max): 0.00% ... 0.00%
Time (median):        2.074 ns                   GC (median):      0.00%
Time (mean ± σ):      2.082 ns ± 0.118 ns         GC (mean ± σ):   0.00% ± 0.00%

```



```

2.04 ns      Histogram: log(frequency) by time  2.08 ns <
Memory estimate: 0 bytes, allocs estimate: 0.

```

Comparing the minimum times here, we have about a 50% speed-up by pre-computing the mask (and also the  $N - 1$  values). Note that we could precompute the mask and pass this in manually, but this makes the code much more verbose. Instead, making use of the `Val` type can save a lot of time and effort.

As an alternative to this, we can also instruct the Julia compiler to propagate some constants through function definitions when compiling. An example on the

Julia Discourse forum<sup>8</sup> shows some allocation differences when separating out your code into separate functions:

```
function compute!(r, a, b, p, s)
    mul!(r, a, b, s, p)
    mul!(r, b, a, -s, true)
    return nothing
end
function calling_fn(r, a, b)
    p = false
    s = 2.0
    compute!(r, a, b, p, s)
end
function manual_inlining_fn(r, a, b)
    p = false
    s = 2.0
    mul!(r, a, b, s, p)
    mul!(r, b, a, -s, true)
    nothing
end
```

The two functions are identical in which operations are performed, but one allocates and the other does not. The main reason for this is that the constants `p` and `s` do not automatically get propagated through the function barrier of `compute!`. Fortunately, instead of having to resort to using `Val` in our function definitions, we can instead use a macro from the Base library to encourage the compiler to propagate our constants through a function barrier:

```
Base.@constprop :aggressive function compute!(r, a, b, p, s)
    mul!(r, a, b, s, p)
    mul!(r, b, a, -s, true)
    return nothing
end
```

The `Base.@constprop` can be used if you notice that refactoring your code leads to allocations that you do not understand.

## 6.9 Inlining

If a function is short enough, one can encourage the compiler to always inline the function so that at runtime, one does not have to pay the cost of calling another function. This can be done in Julia using the `@inline` macro:

<sup>8</sup> <https://discourse.julialang.org> - the main place to ask Julia related questions. This has a much bigger community than Stack Overflow for the language.

```
@inline function +(a::MyComplex, b::MyComplex)
    MyComplex(a.real + b.real, a.imag + b.imag)
end
```

This makes sure that everywhere the addition function between two complex numbers is used, the code will be inlined. We should note that the compiler will inline your code for you, and using the `@inline` macro is only a suggestion to the compiler. Most of the time this is not necessary.

### 6.10 *Generated Functions*

We have not covered Julia's amazing meta-programming capabilities, other than showing the use of a few macros. Remember that meta-programming is writing code that will write other code. We are not covering this in detail, but this is a very short overview just in case you come into contact with these concepts in the wild. It is very rare that you need to use these facilities, as they are often just syntactic sugar to help code expressibility and reuse, without sacrificing performance.

One special way of creating a function is by using a generated function, which uses the types of the inputs to change how the code is written. Let's give a very basic example of creating a function which evaluates a polynomial, taken from the Julia base library:

```
@generated function my_evalpoly(x, p::Tuple)
    N = length(p.parameters::Core.SimpleVector)
    ex = :(p[end])
    for i in N-1:-1:1
        ex = :(muladd(x, $ex, p[$i]))
    end
    ex
end
```

Algorithm 6.26. This function is taken from the Base Julia library. Given parameters in a tuple  $p$ , the polynomial  $p_1 + p_2x + p_3x^2 + p_nx^{n-1}$  will be evaluated. This function is a generated function which changes the expression based on the input tuple (and its length).

This is meta-programming, as we have written code which generates an expression, which then becomes the function that is evaluated. Here, we are saying that if the coefficients of the polynomial are given in a tuple (whose size is known at compile time), then we construct a nested expression. When this function is compiled, it will generate another function that is used in place of this generated function. We can call this function as expected:

```

julia> x = 5.0;
julia> p = (1.0, -2.0, 5.0);
julia> @benchmark my_evalpoly($x, $p)
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min ... max):  2.053 ns ... 6.553 ns   | GC (min ... max): 0.00% ... 0.00%
  Time (median):      2.084 ns                 | GC (median):      0.00%
  Time (mean ± σ):    2.085 ns ± 0.101 ns     | GC (mean ± σ):   0.00% ± 0.00%

Histogram: frequency by time
2.05 ns          2.08 ns <
Memory estimate: 0 bytes, allocs estimate: 0.

```

We can inspect the lowered code using the `@code_typed` macro:

```

julia> @code_typed my_evalpoly(x, p)
CodeInfo(
  1 - %1 = Base.getfield(p, 3, true)::Float64
    | %2 = Base.getfield(p, 2, true)::Float64
    | %3 = Base.muladd_float(x, %1, %2)::Float64
    | %4 = Base.getfield(p, 1, true)::Float64
    | %5 = Base.muladd_float(x, %3, %4)::Float64
    |   return %5
) => Float64

```

Notice that we only have a few multiply add expressions. Compare this to a naive implementation:

```

function my_evalpoly_simple(x, p)
    N = length(p)
    s = p[end]
    for i in N-1:-1:1
        s = muladd(x, s, p[i])
    end
    s
end

```

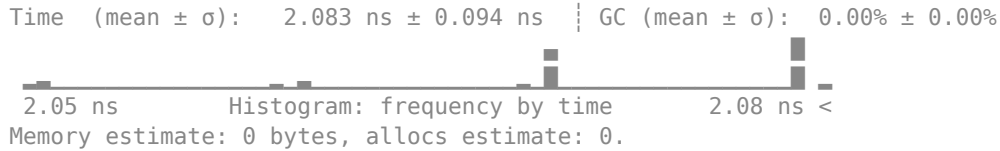
Algorithm 6.27. Same as Algorithm 6.26, but not using a generated function.

We can benchmark this to see that the performance is pretty much the same, as the compiler is smart enough to be able to unroll the loop:

```

julia> @benchmark my_evalpoly_simple($x, $p)
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min ... max):  2.053 ns ... 5.931 ns   | GC (min ... max): 0.00% ... 0.00%
  Time (median):      2.084 ns                 | GC (median):      0.00%

```



This is another case of pre-emptive optimisation, assuming that the compiler is not smart enough to generate the most appropriate, and performant, machine code. In older versions of Julia, this optimisation could be very important. However, usually it is not worth it to write these complicated generated functions, unless you know what you are doing.

### 6.10.1 Performance Annotations

#### 6.10.2 Fast Math

We can allow the compiler to include floating point optimisations that are correct for real numbers, but may lead to differences for IEEE encoded floats. These may change numerical results and accuracy, but may improve the performance of your code. This is enabled via the `@fastmath` macro in Julia:

```
function nofastmath_example!(y, x)
    @inbounds for i in eachindex(y, x)
        y[i] = sin(x[i])
    end
    nothing
end
function fastmath_example!(y, x)
    @inbounds @fastmath for i in eachindex(y, x)
        y[i] = sin(x[i])
    end
    nothing
end
```

Algorithm 6.28. An example of two algorithms, which only differ by the use of `@fastmath` macro.

We can benchmark these two algorithms:

```
julia> x = rand(1024); y = similar(x);
julia> @benchmark nofastmath_example!($y, $x)
BenchmarkTools.Trial: 10000 samples with 8 evaluations.
Range (min ... max): 3.333 μs ... 4.199 μs | GC (min ... max): 0.00% ... 0.00%
Time (median): 3.412 μs | GC (median): 0.00%
```

```

Time (mean ± σ):  3.432 μs ± 82.281 ns  | GC (mean ± σ):  0.00% ± 0.00%
Histogram: frequency by time
3.33 μs  3.85 μs <
Memory estimate: 0 bytes, allocs estimate: 0.
julia> @benchmark fastmath_example!($y, $x)
BenchmarkTools.Trial: 10000 samples with 8 evaluations.
Range (min ... max):  3.223 μs ... 11.929 μs  | GC (min ... max): 0.00% ... 0.00%
Time (median):        3.313 μs                | GC (median):      0.00%
Time (mean ± σ):     3.331 μs ± 155.207 ns    | GC (mean ± σ):   0.00% ± 0.00%
Histogram: log(frequency) by time
3.22 μs  3.73 μs <
Memory estimate: 0 bytes, allocs estimate: 0.

```

We can see that the `@fastmath` example is slightly faster. These operations can violate strict IEEE semantics, making some operations undefined behaviour. For this reason, it is often avoided in many scientific applications and is an *opt-in* performance enhancement.

### 6.10.3 Bounds Checking

We have already seen the use of the `@inbounds` macro throughout this book. This is one of the easiest optimisations to make, as long as you are confident that you are accessing memory in a correct way. Turning off bounds checking and accessing incorrect areas of memory may lead to undefined behaviour, memory corruption and crashes. This can be mitigated by proper use of methods like `eachindex` or `axes`.

## 6.11 Conclusion

Optimisation is a very broad topic, and we are only able to cover the introduction to this topic here. Many of the tips discussed in this chapter are applicable to other languages. Many tips specific to Julia are given in the “Performance Tips” section of the Julia manual<sup>9</sup>, which are certainly worth reading, but the main points are included in this chapter.

<sup>9</sup> <https://docs.julialang.org/en/v1/manual/performance-tips/>

We have opted not to talk about specific optimisations by using the correct algorithm for a task, as this is often too specific to a certain problem. Instead, it should be your responsibility as the developer to choose the appropriate algorithm for the job. You can then use the techniques in this chapter to make your implementation of that algorithm as performant as possible, using a single core.

When writing high performance code, it is imperative that you first optimise the existing code as described in this chapter, before moving onto optimising via parallelism. The next few chapters will cover the basics of parallelism over the main paradigms available to programmers today. It is important to remember that the main source of speed up comes from optimising the serial version of your code, and these benefits can often be compounded by using the right parallel paradigm to further speed up the execution of your code.



## 7 Introduction to Parallel Programming

Now that we have gained an understanding of how to write fast code in Julia, or more accurately - how to avoid writing slow performing code in Julia, we can move onto to trying to utilise modern hardware to accelerate our software.

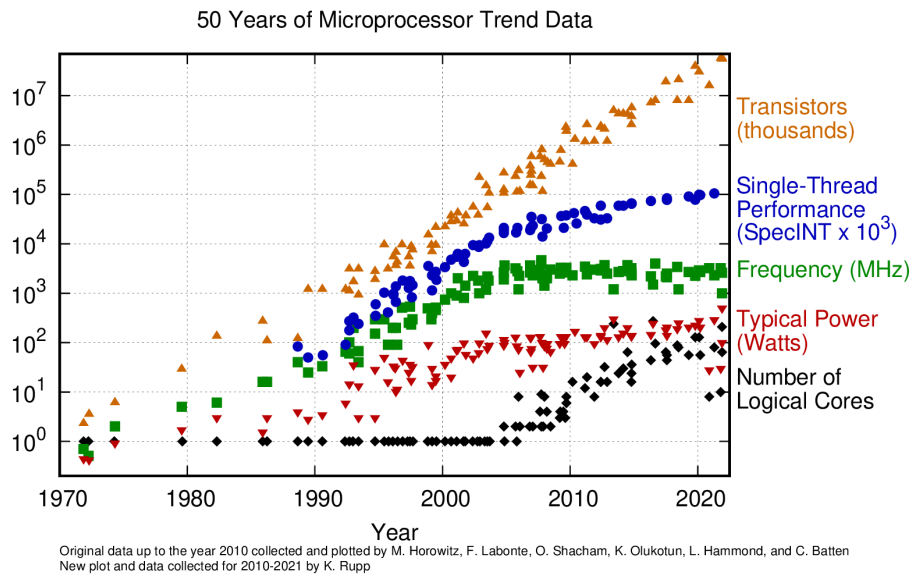


Figure 7.1. Shows the trends of CPUs over time. Notice that the transistors double roughly every two years, as Moore predicted, however single-thread performance and frequency have begun to stagnate. Figure sourced from

Over the past few decades, single-core performance has not been improving at the rate it once was. We can see this in Figure 7.1, as it clearly shows this

performance stagnating, along with clock speed frequency. Now, the way that manufacturers have sought to increase the performance of their chips, was simply to add more cores to a single CPU, you can see this trend beginning in the mid 00's. While each core might only be 10% faster than the last generation, it may now have twice as many cores, and theoretically, able to at least double the performance, given the right task and software. Now, some manufacturers are able to fit 128 cores onto a single chip. Additionally, some motherboards are dual socket, which means they can fit two CPUs and have twice the number of cores available.

While in theory, one can increase the amount of computational power available, there are quite a few challenges:

- Some tasks are inherently sequential, and cannot be divided up amongst multiple cores and are limited to sequential processing. An example of this is in a Physics engine, where one has to work out the current state of the simulation before being able to calculate the next state. While the forces can often be calculated in parallel, each distinct step relies on the previous step and has to be done in sequence.
- Developers need to write the code in such a way that it can efficiently utilise multiple cores, while working across many different computers and architectures. This adds a lot of complexity to the code base and requires a much higher level of expertise, let alone the increased amount of testing required to ensure the software performs as expected. Many applications today under-utilise the hardware due to this increased burden on the developers.
- In order to parallelise the code correctly, the algorithms can be completely different, requiring huge amounts of developer time to rewrite and test these new algorithms. This process is fraught with bugs which make software less reliable. Reliability is often preferred over speed, and so parallelising code is often overlooked.
- If done incorrectly, parallelised code can often be **slower** than the serial alternative. It is important to know when to parallelise and when not to.

This chapter will aim at exploring these challenges and give you the theoretical tools to understand the framework of parallelisation.

## 7.1 *Dependency Graphs*

In order to be able to parallelise, we first need to formalise how to describe a process that benefits from parallelism. It is easy to build up an intuition of this. Firstly, let's define what a **worker** is:

In the context of parallelisation, a *worker* is an entity which can process information and perform certain tasks.

Purposefully, we have kept this notion as abstract as possible. When thinking about algorithms in an abstract sense, it is often useful to imagine the process being performed by hand, and humans filling the role of the worker. While an algorithm, when implemented on a computer, will be executed by a CPU, it is possible that each core will act like a separate *worker*. We will often talk about a worker as if they are a human with agency, but this is only for illustration purposes.

We can imagine breaking up a large task into many smaller tasks. A convenient way to organise these subtasks is in a hierarchy, in which tasks at the top do not depend on any other tasks. Moving down the hierarchy, tasks depend on the completion of tasks higher in the hierarchy. This structure is often seen in a tree-like structure. Let's take a trivial example of a simple stir-fry recipe to illustrate the point:

1. Slice chicken/protein into thin strips.
2. Julienne (slide into strips) an onion.
3. Julienne a bell pepper.
4. Mince Garlic.
5. Peel and grate ginger.
6. Cook the rice according to instruction.
7. Pre-heat a wok to high heat.
8. Add half oil for frying.
9. Brown protein in the wok, stirring frequently.
10. Remove protein on a plate when slightly browned.

11. Add onion to the wok and fry until slightly browned.
12. Add garlic, ginger and pepper.
13. Add in the cooked protein (with seasoning and soy sauce) and cook for a couple of minutes.
14. Serve the stir-fry on the bed of cooked rice.

This recipe has many items, but we can draw a diagram of the process. In order to trace dependencies, it is often easiest to go to the last item in the task and work backwards for what is needed. We can draw this process in a dependency graph, shown in Figure 7.2.

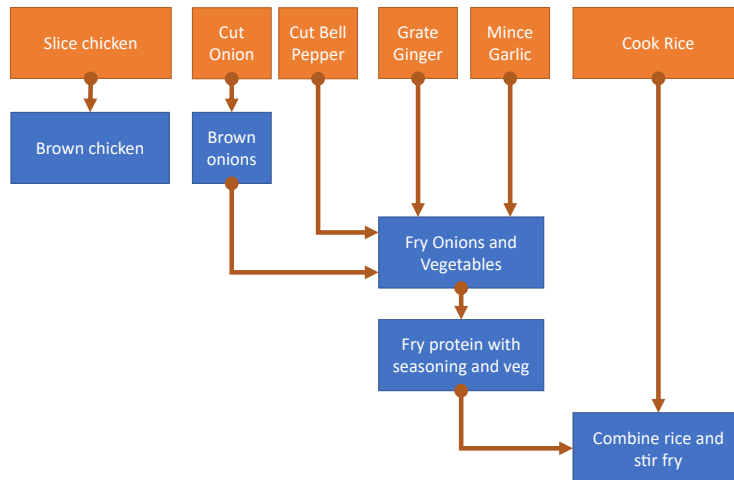


Figure 7.2. This is a dependency graph depiction of making a basic stir-fry. Arrows going into a task show a dependency on the task at the origin of the arrow. This type of graph roughly shows the necessary temporal aspect of a task.

Notice how all the tasks in the top row can theoretically be done at the same time. Obviously, we are ignoring the restriction that food loses heat once cooked, and so cooking tasks should only be completed “Just-in-Time”. If we had enough resources (enough workers, knives and equipment), we could have one worker cutting an onion while another cuts a bell pepper. Notice how in doing this, we have not only increased the number of people needed, but we have also increased the resources required (one knife to two knives). Additionally, what is not obvious from the recipe, is that given two woks and two people, we could brown the chicken and the onions at the same time in each wok. This is because they are not inherently dependent on each other. If we only had a single wok, then suddenly,

browning the onions would depend on having already browned the chicken. If the rice starts cooking at the beginning of the process, it will be ready and finished by the time the rest of the meal has cooked, this process can happen in parallel too. The important aspect of this diagram, is that we can visually group tasks by which ones can be done in parallel. This gives us an idea of how to speed up the process, given more workers and resources.

Imagine a kitchen of professional chefs, who work together on a single meal. If managed correctly, this will usually speed up the process as some tasks can be done in parallel. However, if each task in the preparation depends on the previous task, then having all the additional resources and workers is not helpful, as only one task can be completed at a time.

A dependency graph shows how a process can be parallelised, and which resources are required at which points during processing. However, we have left out any concept of time. We are not trying to solve a scheduling problem - which inherently would involve knowing about the time taken to complete each task. A dependency graph tells you only if a task can be parallelised, not whether it should be, as there is not enough information to answer that question.

One may recognise a dependency graph as having similarities to a flowchart. A flowchart encapsulates the logic of an algorithm, independent of implementation details. One may use flowcharts with small tweaks to act as a dependency graph, to visualise the structure of a task.

While a flowchart is visually very easy to understand, when talking about dependency graphs we must remember that a dependency graph should never have a cycle - i.e these graphs should be *acyclic*. These type of graphs have a special name: **Directed Acyclic Graph** (DAG). While we may show flowcharts with loops (as in a sort of `for/while` loop), each cycle of this is inherently dependent on the past as is only grouped into a cycle for illustration purposes.

For clarity, let's take a simple algorithm, written in Algorithm 7.1. This program simply maps each input of an array to another value, which is stored in another array that is preallocated.

If we label the number of elements in `x` and `n`, then we know that we are calling the function `_inner_solve` `n` times, once for each of the elements. We know that

```

function _inner_solve(x)
    rate, limit, x = promote(0.01, 2.0, x)
    for t in 1:100
        x += rate*clamp(x, -limit, limit)
    end
    return x
end
function map_solve!(y, x)
    @inbounds for i in eachindex(x, y)
        y[i] = _inner_solve(x[i])
    end
    nothing
end

```

Algorithm 7.1. An example function which calculates an expensive operation over an array of elements and stores the results in a preallocated array. The function `promote` is used to make sure the compiler converts the constants into the same, compatible, types during compilation, ensuring that no type conversion happens at runtime. It also allows us to name the variables.

processing the inner loop requires loading the  $i^{\text{th}}$  element of the array `x`, and then processing this value in another function and finally storing the output in an array.

We can see a representative dependency graph in Figure 7.3. Firstly, we can see that processing one element of the inner loop, first requires that we have a pointer to the array `x`. We can see that each evaluation of the inner loop is independent of all other inner loop evaluations. Another way of saying this is that these inner operations can safely be done in any order without changing the final result. Finally, once all the processes in the inner loop have completed, then the algorithm is complete.

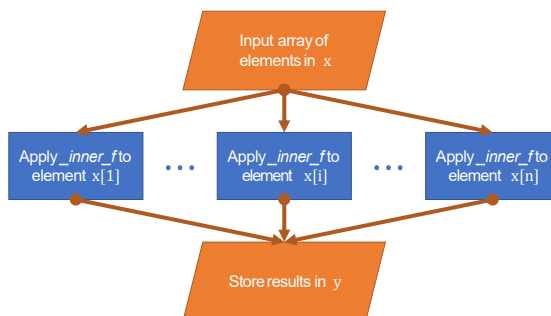
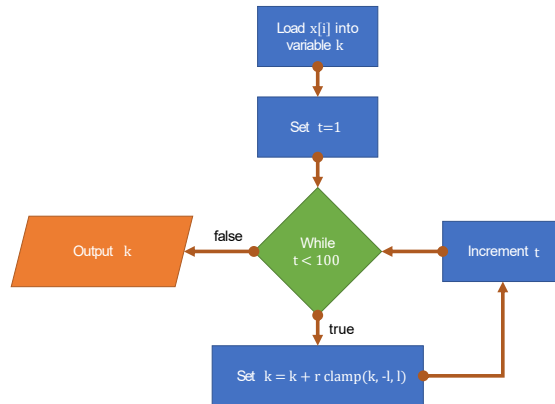


Figure 7.3. This diagram shows a flow chart like DAG. This shows the processing dependencies between each part of the program. As the graph is directed and there exist no connections between each of the blue processing steps, we can conclude that these processes do not overlap with data and hence can be processed at the same time.

For clarity, the diagram of the inner loop has been included in Figure 7.4. Note that this graph does not have any independent processes and so cannot be

parallelised. We can roll this process into a single processing step as shown in Figure 7.3.



What is important to see is that each evaluation of the inside of the loop is completely independent of all other processing. This means that it is safe for us to work on multiple inner loop evaluations at the same time - processed in parallel. This type of parallelism is extremely common and has a name - *embarrassingly parallel*. There are many problems that fit into this category, and thankfully, are the easiest to deal with.

## 7.2 Theoretical Expectations

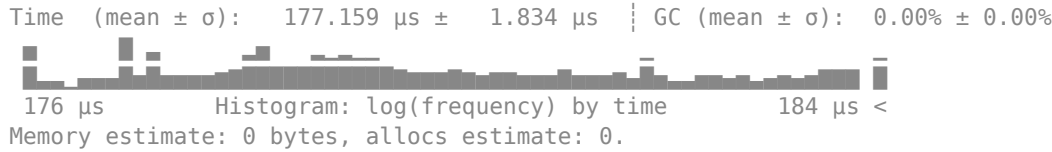
### 7.2.1 Amdahl's Law

Let's benchmark the algorithm from the previous section (Algorithm 7.1) with an array of 1024 numbers:

```

julia> x = rand(1024);
julia> y = similar(x);
julia> @btime _inner_solve($(x[1]))
 170.529 ns (0 allocations: 0 bytes)
2.481729241528297
julia> @benchmark map_solve!($y, $x)
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min ... max):  175.657 μs ... 242.796 μs  | GC (min ... max): 0.00% ... 0.00%
  Time (median):      176.660 μs                  | GC (median):    0.00%
  
```

Figure 7.4. This shows the expanded dependency graph for the blue processes in Figure 7.3. While this graph clearly contains a cycle in the while loop block, it is possible to unroll this loop into a DAG like structure, since the loop is finite and eventually ends. This process has been depicted in this way for convenience, but one should think of this as being unrolled in terms of dependencies. It is clear that each subsequent result of  $k$  requires the result of the previous iteration. Unlike the process in Figure 7.3, this process cannot be parallelised, as each block is dependent on the last.



We can see that the processing scales linearly with the size of the array. If we had 1000 processors, we could theoretically speed up this function call, by having each processor calculate the inner loop for each element and put the result in the final array. However, we would still have to spend the 210 nanoseconds required to process a single element. This brings up the concept of Amdahl's law<sup>1</sup>, which states:

<sup>1</sup>[https://en.wikipedia.org/wiki/Amdahl%27s\\_law](https://en.wikipedia.org/wiki/Amdahl%27s_law)

$$S = \frac{1}{(1 - p) + \frac{p}{s}}, \quad (7.1)$$

where  $S$  is the speed-up factor,  $p$  is the proportion of the program which can be parallelised, and  $s$  is the speed-up of the part of the task that is parallelisable. We can interpret the  $(1 - p)$  factor as the proportion of the program which cannot be improved.

Amdahl's Law tells us the theoretically maximum speed-up we can achieve by parallelising part of our program. In our example, we can know that  $p = 1$  as all  $N$  tasks can be done in parallel. However, the speed-up factor is not as trivial. We know that each worker can process one element at a time, and so the minimum amount of time needed is proportional to  $\left\lceil \frac{N}{w} \right\rceil^2$ .

<sup>2</sup>  $\lceil \cdot \rceil$  represents the *ceiling* function, which rounds up to the nearest integer. For example,  $\lceil 3.2 \rceil = 4$ , but  $\lceil 3 \rceil = 3$

If we put these equations together, we will see that an embarrassingly parallel problem (where each element takes the same amount of time) will have a speed-up given by

$$S = \frac{N}{\left\lceil \frac{N}{w} \right\rceil}, \quad (7.2)$$

If we take the limit as  $N \rightarrow \infty$ , we know that  $\left\lceil \frac{N}{w} \right\rceil = \frac{N}{w}$  and hence  $S = w$ , which is what is expected.

Let's say that we also need to read in the  $x$  data from the disk, which takes around  $1\mu$ s per element and must be done sequentially. If the operation on each element takes 200ns, then the factor  $p$  changes to  $p \frac{200\text{ns}}{1\mu\text{s}} = 0.2$ . This will change



our maximum speed-up:

$$S = \frac{1}{0.8 + 0.2 \frac{\lceil \frac{N}{w} \rceil}{N}}. \quad (7.3)$$

If we take limits as  $N \rightarrow \infty$  and assume we have unlimited resources (i.e  $w \rightarrow \infty$ ), then the maximum speed-up is 1.25. This shows us that no matter how many resources we have, if the sequential part of an algorithm takes much longer than the part which can be parallelised then we are severely limited in how much we can speed-up a task.

### 7.2.2 Embarrassingly Parallel Problems

For our practical example, we need to know the number of workers we have available. We can see the number of processors available with the built-in function

```
julia> Threads.nthreads()  
16
```

If this is set to 1, you most likely have to start Julia with more threads. It is common to set the number of threads to the number of physical cores available on your processor. You can use `julia -t auto` to start Julia with the number of threads equal to the number of logical processors. Due to Simultaneous Multithreading (SMT) this can be double the number of your actual physical cores.

Let's quickly implemented a parallel version of our algorithm using multithreading. We will discuss this in more detail later. In Julia, one can perform tasks in parallel by simply adding a macro on the beginning of a `for` loop:

```
function map_solve_parallel!(y, x)
    @inbounds Threads.@threads for i in eachindex(x, y)
        y[i] = _inner_solve(x[i])
    end
    nothing
end
```

Algorithm 7.2. Same as Algorithm 7.1, but using the `Threads.@threads` macro in the Julia base library. Note that the `@inbounds` is not required, but speeds up performance.

We can benchmark this algorithm the same as before:

```

julia> @benchmark map_solve_parallel!($y, $x)
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
Range (min ... max): 23.505 μs ... 119.159 μs      GC (min ... max): 0.00% ... 0.00%
Time (median):       37.627 μs                    GC (median):      0.00%
Time (mean ± σ):    39.818 μs ± 9.371 μs          GC (mean ± σ):   0.00% ± 0.00%

Histogram: frequency by time
23.5 μs                                     66.7 μs <
Memory estimate: 8.42 KiB, allocs estimate: 97.

```

Notice how the number of allocations has increased. This is because the threading macro rewrites your code and inserts some heap allocations. However, the performance of this algorithm is much better, around 6 – 7 times faster. This is lower than our theoretical maximum for many reasons. The first of which is that running code in parallel requires a lot of overhead and synchronisation to happen across the cores. The second reason is that each core may access the same cache line in memory, and even worse, each core can be modifying the same cache line, invalidating it and causing a huge slowdown. This is what is known as **false sharing**. Additionally, the memory access is not uniform and the compiler cannot vectorise the code, negatively impacting performance. We will visit better ways of parallelising this sort of algorithm in a later chapter.

### 7.2.3 Gustafson's Law

In the previous section, we talked about the theoretical maximum speed-up of an algorithm, given some amount of resources. However, this law does not give the entire picture. Today, many engineers and scientists are more interested in scaling up algorithms to bigger input sizes, and seeing how parallelisation scales with the problem size.

Gustafson's Law states that the estimated speed-up of a program gained by using parallel computing is given by:

$$S = (1 - p) + pN, \quad (7.4)$$

where  $N$  is the number of available workers (or processors) and  $p$  is the fraction of time spent executing the parallel parts of the program.  $S$  represents the theoretical **slowdown** of executing an already parallelised algorithm on a serial-only machine. This law proposes that engineers and scientists tend to increase the size of problems to fully exploit the computing power that is available on workloads of increasing size. Another way that people look at this, is asking

If a task, quantified by a workload size of  $w(n)$ , takes  $x$  amount of resources and  $t$  time to complete, then given  $y$  amount of resources, what size,  $n'$  can be scaled up to, resulting in the same competition time.

An example from the Wikipedia article on Gustafson's Law gives the example of an operating system's boot-time. If a user is okay to wait 30 seconds for a computer to boot-up, and we have additional resources, which additional features and tasks can be done during boot, making sure that the boot time is not increased?

### 7.3 Maps and Reductions

The example given in Algorithm 7.1 is the essence of what we call a *map*. This is a term widely used in functional programming. Essentially a map, simply applies the same operation (or function) to all inputs. It is an extension of SIMD (Single Instruction Multiple Data). This form of action is “embarrassingly parallel”. In Julia, whenever we use the broadcast operator, we are applying a map:

```
julia> x = [1,2,3,4];
julia> is_odd(x) = x % 2;
julia> is_odd.(x)
4-element Vector{Int64}:
 1
 0
 1
 0
```

Here, we have mapped the `is_odd` function to each element of the input array and then returned the output of that array.

Mapping is a very useful concept, especially since it can be so elegantly expressed in Julia, however, what happens when we need to combine elements together. For example, if we want to sum all the elements in an array? Traditionally, we have a variable to keep track over the total sum, initialised at zero, and then iterate over all the elements, adding each element to the sum. This sort of operation is called a *reduction*, aptly named for the effect of producing an output with fewer terms than the input - i.e. reducing a collection to a single element.

Let us take the example of the `sum` function. The dependency graph of the serial algorithm is shown in Figure 7.5. However, we know that addition is associative (i.e.  $a + (b + c) = (a + b) + c$ ), which means that we are free to rearrange the order of additions, and transform the dependencies in a way that aids parallelism.

However, inside a computer, numbers are represented with a finite amount of bits. Provided there is no overflow with integers, the addition should still be associative. However, floating point numbers are represented in standard form notation, with a sign, mantissa and exponent. Necessarily, these representations are limited and make approximations on most numbers. These approximations lead to the loss of associativity in operations of floating point numbers. For this reason, different implementations of the same sum of floating point numbers may be slightly different.

A simple way of making the algorithm parallel, is to divide up the array into  $k$  chunks, one for each worker, and have each worker serially reduce their chunk of the array and then have a final step which combines the results of all  $k$  chunks serially. Provided that the number of elements in the array is much larger than the number of workers, this will be a very easy task to parallelise.

Another way is to fundamentally change the algorithm. We can assign two pairs of numbers to each worker to add together. This will produce an intermediate result with around half the number of elements in the initial array. From here, the workers repeat the process, combining pairs of numbers, until there is only a single number left. This algorithm actually has practical benefit when applied to a sum reduction of elements of similar magnitude. This relates to the floating point problem. Additions of floating points are most accurate when the exponents of the floats are roughly equivalent (similarly sized numbers). If the exponents are different, then the mantissa of one number needs to be manipulated to match the magnitude before addition, propagating any errors resulting from finite bit size forwards. This effect is magnified as the total sum variable gets larger and larger compared to the elements in a sum, as the algorithm works through the elements.

This adapted algorithm, given enough resources, can vastly improve the parallelism of a reduction. In the serial case, the algorithm scales as  $\mathcal{O}(n)$ , where  $n$  is the number of elements in the array. However, in the parallel case, with enough resources, the algorithm scales as  $\mathcal{O}(\log_2(n))$ . This can be derived easily, since each halving of the problem set can be computed in constant time (given enough resources) and it takes proportionally  $\log_2(n)$  iterations to halve the initial  $n$  elements down to 1. This is the solution to the equation  $\left(\frac{1}{2}\right)^t n = 1$ .

This is just a simple example of a reduction for a sum, but there are many interesting and varied implementations, usually relying on associative (and also commutative) properties of the underlying operations.

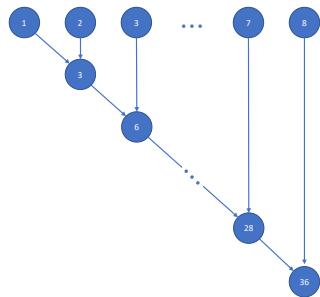


Figure 7.5. Shows a dependency graph (and execution graph) of a reduction (in the form of a sum), on the numbers from 1 to 8. Some steps have been excluded for clarity.

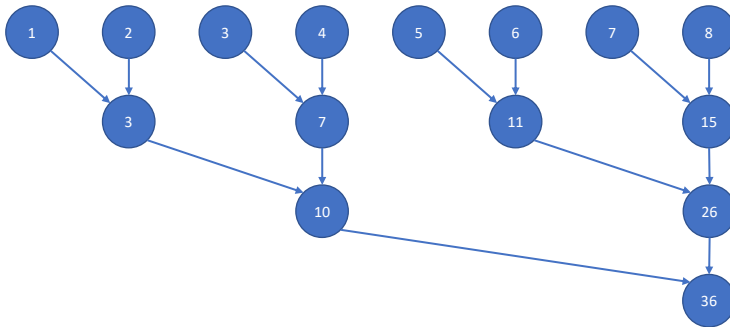


Figure 7.6. Shows a dependency graph of a binary, parallel, reduction algorithm, which sums the numbers from 1 to 8. This is a fundamentally different algorithm from Figure 7.5, however, it is equivalent under the assumption that addition is associative, which is true for integers, provided no overflow.

Julia provides default implementations for `map` and `reduce` functions. `map` takes in a function and collection, and applies the function to each element in the collection. For example, if we want to square all elements of a list and add 1 we would write:

```
julia> x = LinRange(0, 1, 5);
julia> x2 = map(y->y*y + 1, x)
5-element Vector{Float64}:
 1.0
 1.0625
 1.25
 1.5625
 2.0
```

Note that `y->y*y + 1` is an example of an `anonymous` function, these are frequently used in Julia when following a more functional style such as the “map/reduce” format.

If we now want to find the product of this array, we can use the `reduce` method with the multiplication operator:

```
julia> reduce(*, x2)
4.150390625
```

This reduction function takes in a two argument function and a collection. It iterates over the collection, combining results until there is a single scalar result left. It should be said that this will only return the correct answer if, and only if, the function is commutative and associative.<sup>3</sup>

As it is extremely common to first map something and then apply a reduction, a fused method is provided:

<sup>3</sup> It should be noted that one can use `foldl` and `foldr` much like the `reduce` function but guaranteeing left and right associativity respectively.

```
julia> mapreduce(y->y*y+1, *, x)
4.150390625
```

The first argument of this function is the mapping function, the second is the reduction function and the third is the collection of elements this will be applied to. We can benchmark the difference between these when put into a function.

```
julia> map_then_reduce(x) = reduce(+, map(y->y*y+1, x));
julia> map_and_reduce(x) = mapreduce(y->y*y+1, +, x);
julia> x = rand(10_000);
julia> @btime map_then_reduce($x)
 5.218 μs (2 allocations: 78.17 KiB)
13333.840517953066
julia> @btime map_and_reduce($x)
 949.708 ns (0 allocations: 0 bytes)
13333.840517953066
```

Note that the reduction operation has been changed to a `+` for avoiding getting an infinity overflow. One can see that the second fused algorithm is much more efficient as one did not need to allocate an intermediate array for the `map`. It is almost always better to use the fused version if you do not need the results of the intermediate `map`.

## 7.4 Thread Safety and Race Conditions

You may have noticed something peculiar about the previous section, particularly in Figure 7.6. This algorithm is broken up into different stages. What one may notice, is that this structure is somewhat arbitrary. One can consider alternative ways in which to break down the sum into pairwise operations. Why were the pairs chosen adjacent, not in an arbitrary order? Another way to break down the sum of  $n$  elements using  $k$  workers is to have each worker sum up  $\lfloor \frac{n}{k} \rfloor$  elements (with one worker taking any left-over elements). As  $k \ll n$  (usually), one can then assign one of the workers to sum the remaining  $k$  elements to produce a final output. The reason we chose this process is to make sure that each worker does not access the same memory at the same time.

Why is this so important? Let us inspect the following example:

Let's run this code with an example:

```
julia> arr = rand(1024);
julia> real_sum = sum(arr)
523.890365972938
```

```

function naive_parallel_sum(arr)
    sum_arr = zero(eltype(arr))
    @inbounds Threads.@threads for i in eachindex(arr)
        sum_arr += arr[i]
    end
    return sum_arr
end

```

Algorithm 7.3. A naive implementation of a parallel reduction. This code should not be used without modifications.

Now we know what to expect, let's see what is returned from our sum.

```

julia> naive_parallel_sum(arr)
523.8903659729372

```

Now this result is different, let's do a few runs to see if we get the same incorrect result:

```

julia> naive_parallel_sum(arr)
134.89395395564378
julia> naive_parallel_sum(arr)
148.8939002174354
julia> naive_parallel_sum(arr)
128.58731755433683

```

Each time the sum is called, we get a different result. This is the quintessential example of a race condition in our code. We only have one piece of memory, the variable represented by `sum_arr`, which is being read from and written to by each of our workers in parallel.

Each worker first reads the value of `sum_arr` and stores this value in a register inside the Arithmetic Logic Unit (ALU) of the current worker. It then reads the value from the input array which is read-only and does not change. Once these two values are inside the ALU, their sum is calculated and then written into the memory of the variable `sum_arr`. This whole process occurs in parallel, which means that another worker may have changed the value in `sum_arr` while another worker has an outdated value inside their ALU. Consequently, when the worker with the old value has completed the calculation, it will **overwrite** the intermediate result from the previous worker. For this reason, the final sum usually only accounts for a much smaller number of elements since most contributions are overwritten. We call this a race condition, since the behaviour of the algorithm depends on the order and the speed of execution, each worker “racing” to read and write to the same piece of memory.

This idea of race conditions overlaps with the idea of thread safety. A thread-safe algorithm can be executed in parallel without introducing any race conditions. An example of an algorithm which is often not thread-safe is a random number generator (RNG), as these are usually all implemented as *pseudo*-random number generators, which rely on an internal state to generate the next number in a deterministic process that has good enough properties to mimic a random process, while still being repeatable. If you parallelise a random process (e.g. a Monte-Carlo simulation), you should make sure that the random number generation algorithm is thread-safe, or the code may run into race conditions when the RNG reads and mutates its internal state.

The golden rule for thread safety is to check whether multiple cores are accessing the **same** piece of memory at the same time. If this memory is constant throughout the parallel process, this is usually fine (e.g. all cores have access to your constant data). However, if each worker tries to write to the same piece of memory, this causes a race condition.

#### 7.4.1 *Mitigating race conditions*

As race-conditions are very common and can have very bad results, such as memory corruption, one would like a way to have the benefits of parallel processing, without having to alter the algorithm that much. Computer scientists and software engineers have come up with ways to mitigate this happening.

##### **Atomics:**

One core concept to understand is the idea of *atomics*. An *atomic* operation is one which cannot be broken up into smaller parts done by different processors. An atomic operation must be executed serially or will break. The example from the previous section included an example of an atomic operation - incrementing a variable with another value. The one line of code -

```
sum_arr += arr[i]
```

- can be broken down into 4 distinct operations:

1. Fetching the value `arr[i]` and loading into register in the ALU<sup>4</sup>
2. Fetching the value `sum_arr` and loading into register in the ALU
3. Performing the sum of the two fetched values

<sup>4</sup>One can argue that the first operation is not part of the atomic since this fetch operation is from read-only memory (with respect to the task).



4. Writing the result of the sum back into the memory, represented by `sum_arr`

You know that the end goal of adding a value from an array to a variable is only valid if `sum_arr` stays constant during the operation. The operation consisting of these 3 tasks (excluding task 1), can be said to be an atomic operation. Therefore, one require that it be performed in serial, rather than in parallel. We need a way of expressing in code the need to perform atomic operations. In many languages there are special functions and libraries available to allow one to write common atomic operations (such as incrementing a variable) in a readable and thread-safe way. In Julia, there exists native support for atomics in the Threads library, so we can start with:

```
julia> using Base.Threads;
```

We can alter the previous algorithm to include this:

```
function naive_parallel_sum_with_atomic(arr)
    sum_arr = Atomic{eltype(arr)}(zero(eltype(arr)))
    @inbounds Threads.@threads for i in eachindex(arr)
        atomic_add!(sum_arr, arr[i])
    end
    return sum_arr[]
end
```

Algorithm 7.4. A naive implementation of a parallel reduction, using atomics.

```
julia> naive_parallel_sum_with_atomic(arr)
523.8903659729365
julia> naive_parallel_sum_with_atomic(arr)
523.8903659729378
julia> naive_parallel_sum_with_atomic(arr)
523.8903659729372
```

Now, if we run this algorithm we will get fairly consistent results. Any errors in the output will be because, unlike normal addition, floating point addition is not associative. However, if we were to benchmark this solution, we would find that it is severely lacking:

```
julia> @btime naive_parallel_sum_with_atomic(arr)
 15.269 μs (99 allocations: 8.45 KiB)
523.890365972937
julia> @btime sum(arr)
 67.113 ns (1 allocation: 16 bytes)
523.890365972938
```

This implementation is around 300 times slower than the native implementation of `sum`. Using atomics is incredibly slow as it forces all the operations to happen sequentially. Additionally, it requires that most threads sit around waiting. In later chapters, we will revisit this problem and implement a much faster algorithm.

If atomics are not very performant, when should they be used? The answer is when the operation running in parallel contains only a small section which needs to be performed atomically. An example would be running an expensive simulation in parallel and aggregating statistics during/after the simulation. These simple operations of aggregating the statistics are likely to be far less expensive than the simulation itself, and each thread spends most of the time in the simulation, and not that much time waiting to access the memory to change the statistics.

The `Base.Threads` module provides the following functions:

- `atomic_or!`
- `atomic_xor!`
- `atomic_sub!`
- `atomic_min!`
- `atomic_max!`
- `atomic_cas!`
- `atomic_and!`
- `atomic_add!`

These cover a large array of operations you may need, however, if you require more flexibility, you can use a *mutex* or a *semaphore*.

### **Mutexes and Semaphores:**

Mutexes and Semaphores are primitives for synchronising operations and processes. A *mutex* provides “mutual exclusion”, which means that only one task can have access to a mutex at a time, while all other tasks are blocked until control of that mutex is released. You can think of a mutex as being a lock on a door, which provides access to a room (which will act as the metaphor for the resources/operations that can only be accessed by one person at a time). Initially, the door is open, so the first person to use the room can walk in and lock the door behind them. Any other people looking to use that room will be blocked and will have to wait until the first person has finished. Once that person is finished, they need to unlock the door to leave, allowing the next person in the queue to go inside.

A mutex is usually implemented by scheduling blocked threads after the current thread with control of the mutex has been completed. This puts the other threads to sleep until they are able to continue. This can cause performance issues as waking a thread from sleep can be expensive. Alternatively, it can be implemented with a “spin lock”, which has each blocked thread keep checking whether the mutex is available over and over in a continuous cycle. This keeps the thread awake, but wastes many CPU cycles and is much less efficient than putting a thread to sleep if the thread has to wait for a longer time.

A semaphore is slightly different in that it is a signalling method which can symbolise the availability of limited resources. A semaphore is essentially an integer variable which has two atomic operations a *wait* and *signal* operation that atomically decrements and increments respectively. The semaphore is initially set to the number of resources available and cannot be decremented below 0. A thread that tries to perform the wait operation when the value is 0, has to wait until another thread releases control and increments the value. The key idea is that a resource can be given and taken by different threads. For example a producer thread can put data into a shared resource, which can then be consumed by another process when the producer sends the “signal” command.

While a mutex can provide mutual exclusion to a single resource / group of resources, a semaphore can represent a buffer or a pool of resources.

In Julia, one should look to use a [ReentrantLock](#) to act as a Mutex. Generally, semaphores are far less common, but can be found in additional packages, or easily written oneself using a [ReentrantLock](#). The syntax can be gleaned by the following:

```

mutex = ReentrantLock()
Threads.@threads for i in 1:length(results)
    # Long running processing
    results[i] = some_function(i)

    lock(mutex) do
        # process the results of results[i] serially
        aggregate_results = add_aggregate!(aggregate_results, results[i])
    end
end
end

```

It is important to remember that this method of programming is usually discouraged, due to the performance hit, as there are usually better implementations.

#### 7.4.2 *Producer and Consumer*

It is very common to have two tasks running in parallel, one which produces data and the other that consumes that data in some way. One example is a simulation which is expensive and run on a background thread and another consumer thread which processes this simulation and controls a live plot. This allows for a split responsibility, which makes the code more reusable, since the simulation code does not need to be hooked up to the plotting code directly.

Julia provides a data structure called a `Channel`, which makes implementing this pattern very easy. A channel is a data structure for storing information, which internally uses locks (mutexes and semaphores) to synchronise data access between different threads.

When constructing a channel, we can specify a capacity along with the data type of the elements stored in the channel:

```

julia> capacity = 8;
julia> buffer = Channel{Float64}(capacity)
Channel{Float64}(8) (empty)

```

Here, we can store a maximum of 8 floating point numbers in the channel called `buffer`.

Now let's write a function which will send data into the channel. An example function is given in Algorithm 7.5.

The function `put!` is similar to `push!`, as it sends the data in the second argument into the channel. However, if the channel is currently full, it will cause the calling thread to hang until there is a free space to put the data in. The final

```
function producer_fn(buffer::AbstractChannel, total_items)
    for i in 1:total_items
        sleep(0.02) # simulate work
        put!(buffer, rand(Float64))
    end
    close(buffer)
    nothing
end
```

Algorithm 7.5. A function to produce data and store it in a buffer.

line calling the `close` function makes sure that the channel cannot accept any more inputs. Also, closing the channel ensures that the consumer knows that the stream of data has ended when all the elements are used up.

We need to also consume the data, which we will also write in a function given in Algorithm 7.6.

```
function consumer_fn(buffer::AbstractChannel)
    s = zero(eltype(buffer))
    for item in buffer
        s += item
    end
    s
end
```

Algorithm 7.6. A function to consume the data produced by Algorithm 7.5. This is done using a very simple summation.

Here, we are safely iterating through the buffer with a `for` loop. This is a safe way to iterate. One can manually iterate through the channel using the `take!` command, but using a `for` loop like this tends to be a better option.

We can finally write some code to see this in action, using the `Threads.@spawn` macro to start work on a different thread.

```
julia> Threads.@spawn producer_fn(buffer, 50);
julia> @time result = consumer_fn(buffer)
1.083228 seconds (32.19 k allocations: 1.636 MiB, 2.96% compilation time)
21.185664445659256
```

This pattern is very useful when you want to read data from a file and start processing it immediately, without having to wait for the file to finish reading.

Additionally, if producing data and consuming data take very different amounts of time, one can have more producers than consumers and vice versa to scale up the entire process. For example:

If we want to schedule the consumer on a different thread as well, the return value from the `Threads.spawn` value is a `Task` object, not the result. We have to manually `fetch` the result to consume it. For example:

```
result_task = Threads.spawn consumer_fn(buffer)
result = fetch(result_task) # hangs until the task is complete
```

```
julia> buffer = Channel{Float64}(capacity)
Channel{Float64}(8) (empty)
julia> [Threads.@spawn producer_fn(buffer, 10) for _ in 1:5];
julia> @time result = consumer_fn(buffer)
0.190185 seconds (284 allocations: 8.922 KiB)
25.797803529808906
```

Here, the result is roughly what we expect still, but we were able to reduce the time waiting for the buffer to fill up since we had multiple consumers. Note that we must re-open the buffer before execution.

When using channels (and mutexes and semaphores generally), one should try to avoid **deadlocking** your code. A deadlock occurs when one thread is endlessly waiting for something that will never happen, usually because another thread is also in a deadlock. This usually happens because one thread is waiting for the results of another thread, but the other thread is waiting on the first thread.

## 7.5 When to parallelise?

The first question that a developer must fully understand is when to parallelise. Since a parallel implementation can often be difficult and sometimes even detrimental to performance, one must answer this crucial question.

### 7.5.1 Profiling and Benchmarking

Usually, one already has an implementation in serial code, which is the target of optimisation. Before deciding to optimise this function and make it parallel, one should begin by benchmarking the function on a typical workload. This benchmarking, ideally, should be written as a function so that it can be repeated throughout the optimisation process to make sure that improvements are being made. Benchmarking the process, usually done via profiling, has to reveal that

optimising this function is worth it, as it will have a large effect on the overall performance of the application. If the application spends 0.001% of its time in this function, making it twice as fast will have little overall effect (as stated by Amdahl's law discussed earlier). We, as developers, have a limited amount of time we can spend on our code, and it is important to use it as efficiently as possible.

If we have decided that this function should be optimised and will have a significant effect, then make sure that benchmarks are clearly written out in functions, so they can be repeated in the future.

### 7.5.2 *Identifying Speed-up Candidates*

Once one has an idea of which parts of the code will have the largest effect on the overall computation time of the program, one should then use the techniques and laws from the previous sections of this chapter to analytically find out whether any parallelisation is likely to increase performance.

*Firstly*, one should think about the dependency graph of this part of the algorithm and see if there are significant sections which can be done at the same time.

*Secondly*, one should ask if the identified parallel sections make up a significant enough fraction of the whole to make a worthwhile impact on the overall execution of the program. This can be estimated using Amdahl's Law.

*Lastly*, one can ask if increasing the problem size would change the answer of the previous question. If the speed-up becomes more efficient with larger and larger problem sizes. This comes with the assumption that increasing data throughput has tangible benefits.

These are all clues that will help you to avoid spending time parallelising a problem which will have little effect on the overall performance of your program. However, we have adequately discussed the "how" of parallelising code. That will be the topic of the next few chapters.

### 7.5.3 *Practical Slowdowns*

Whenever one wishes to parallelise an algorithm, there are significant factors that can get in the way of gaining the maximum speed-up. The main factors can be summarised as:

1. Time taken to spawn multiple threads/workers.

2. Time taken to schedule the work and divide it amongst the workers.
3. Time spent creating additional storage for the workers to avoid race conditions.
4. Time wasted due to idling because of poor **load-balancing**<sup>5</sup>.
5. Idle time of workers due to locks (mutexes, semaphores and atomics).
6. When more cores of a processor are being used, the energy used by the chip is vastly increased. In order for the CPU to stay at a safe temperature, the firmware may reduce the clock speed of each core. Lowering the clock speed will lower the overall power draw and the temperature of the CPU. Additionally, single-core may have a turbo mode which ramps up the clock speed of a single core, which is reduced when multiple cores are being used. For this reason, one may not see as big a speed-up as expected.

<sup>5</sup> Load balancing is scheduling the tasks amongst the workers so that each is as busy as possible. If the tasks take different amounts of time, then this strategy can become more complicated.

We will spend time in the next chapters discussing not only how to parallelise the code, but also strategies for mitigating these issues which cause performance hits.

## 7.6 *Summary*

Throughout this chapter, we have looked at the theoretical basis of parallel computing. Many of the examples shown use the **multithreading** paradigm (discussed more in Chapter 8), however, we will also have an in-depth look at **multiprocessing** in Chapter 9 and even GPU parallelism in Chapter 10.



## 8 *Multithreading*

As there are many types of parallelism, we must first make a distinction about multithreading. Multithreading is the most common parallel programming paradigm you will come across, at least in languages like Julia, C#, etc. It is usually the easiest paradigm to implement, and can usually be added into serial code with only a small amount of tweaks. Multithreading has the following traits:

- A process using can have many threads, each thread being a distinct, self-contained, sequence of instructions that can execute in parallel or concurrently<sup>1</sup>.
- A thread is an abstract unit of work which is usually mapped one to one with a CPU core. A CPU is oversubscribed if the number of concurrent threads being executed is larger than the number of CPU cores, in which case the CPU must spend time switching between the threads to give the illusion of full parallelism, but at the cost of degraded performance.
- In the multithreading paradigm, threads can access **shared memory**, and so can read from and write to the same variables.
- As each CPU core has its own L1 cache, and some cores do not share L2 cache, this means that cache should be considered to be local to each thread.

In comparison with other parallel programming paradigms, multithreading is distinct in that is a *shared memory* paradigm, meaning that multiple workers can work on the same piece of memory. Shared memory between workers is both a blessing and curse. A blessing as one does not need to manage communication of memory between the workers. A curse since multiple workers can access the same memory, leading to potential race conditions which can be hard to detect, but easy to introduce.

<sup>1</sup> Concurrency notes the ability to execute parts (in this case threads) out-of-order or partial processing of one and stopping to work on another. One can think of concurrency as what will happen to parallel code when there is only a single worker, where the worker quickly switches between many tasks, giving the **illusion** of working on the items in parallel.

As an additional complication, many modern CPUs now have Simultaneous Multithreading (SMT - also known as Hyper-Threading by Intel). SMT introduces new registers on each CPU core that effectively “double” the number of threads that a CPU core can process. The main idea behind this, is that multiple CPU tasks may need different units within the CPU core. One task may require the ALU (Arithmetic Logic Unit - the unit that handles mathematical operations), while another task only needs to manage memory. With SMT, we can effectively have both tasks running in parallel on the core, as if we had two physical cores. This speed-up only works if the resources that each task need at any one time do not overlap. Adding SMT to a chip is a cheap way of increasing the throughput through a chip. Usually, SMT will only increase the performance of some code by around 20 to 30%, but costs the manufacturers very little to add on. Unfortunately, in scientific computing we rarely have tasks that require different resources on each CPU core, rendering SMT to be of little practical benefit. When choosing the number of threads to use, a safe bet is to use the same number as you have physical cores. On Windows, task manager will show you that you have double the number of cores you actually have, as these show **logical cores** and not **physical cores**.

## 8.1 Multithreading in Julia

Fortunately, since Julia v1.3, multithreading has been fairly simple to use inside of Julia. Performance of multithreading has also increased as Julia develops. For this reason, results shown in this book may vary greatly, depending on when it is read, and the current version of Julia you are using.

This section will cover a basic use of multithreading to speed up an embarrassingly parallel algorithm. An embarrassingly parallel problem is one where we have a one-to-one relationship between our inputs (parameters) and our desired outputs (results). This sort of problem can be distilled down to repeating the following operation  $n$  times:

$$r_i = f(x_i), \quad (8.1)$$

where  $x_i$  represents one element of the  $n$  elements that are to be processed,  $f$  is the common function that performs the mapping and  $r_i$  represents the  $i^{\text{th}}$  element of the results vector. To use Julia specific language, any operation that can be written in vector format (broadcasted) is likely a good candidate to be embarrassingly parallel.

For a case study, let's look at a way that we can calculate  $\pi$ . We will choose a Monte-Carlo method of estimation, since this will prove to be an excellent case study for multithreading. Monte-Carlo methods are sampling techniques that can be used to estimate quantities via random simulations. They are often very easy to implement and can be used as a simple technique to provide numerical estimates for quantities that difficult or impossible to calculate analytically. In the case of calculating  $\pi$ , one can estimate its value by playing darts on a square block, with a circle touching the edge of the square. If the radius of the circle is  $r$ , then the area is  $\pi r^2$  and the area of the square it lies within is  $4r^2$ . We know that the ratio of the area of the circle compared with the square is  $\rho = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$ . If we can calculate an estimate for  $\rho$ , we can approximate  $\pi$  as  $\pi = 4\rho$ .

We can now randomly play darts on this setup, making sure that each point in the square is equally likely to be somewhere a dart hits. Then we can count how many of the darts land in the circle, compared with landing outside the circle and in the square. If we throw  $n$  darts uniformly randomly in the square and only  $n_c$  land inside the circle, then we know that  $\rho \approx \frac{n_c}{n}$ , which will approach the true value of  $\rho$  when  $n \rightarrow \infty$ , or in other terms:

$$\pi = 4\rho = 4 \lim_{n \rightarrow \infty} \frac{n_c}{n} \quad (8.2)$$

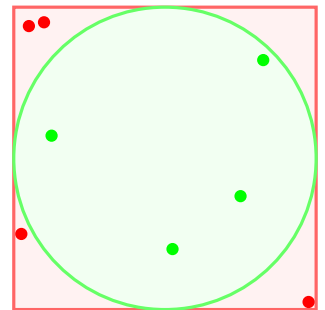


Figure 8.1. The dart board setup, with a rectangular backdrop and a circle matched to the edges of the backdrop. The green circles show a dart hit and the red circles show a miss outside the circle.

First, let's write an algorithm to do this in serial.

```
function est_pi_mc_serial(n)
    n_c = zero(typeof(n))
    for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_c += 1
        end
    end
    return 4 * n_c / n
end
```

One can see that the estimate for  $\pi$  improves with the number of darts thrown, by using a much higher number of points:

```
julia> pi
π = 3.1415926535897...
julia> est_pi_mc_serial(100)
3.28
julia> est_pi_mc_serial(10_000)
3.1356
```

Additionally, one can plot the standard deviation of the relative errors of each estimate of  $\pi$  against the number of darts thrown. This can be seen in Figure 8.2, which shows that the relative error approaches zero as  $n \rightarrow \infty$ .

While this is not the most efficient way of calculating  $\pi$ , heavily relying on the quality of the random number generator, it does provide a case study for a parallel speed up.

Each dart thrown requires two randomly generated numbers, mapped to be between 0 and 1. Additionally, they must calculate the square of the distance from the dart to the origin and check to see whether it is less than 1. All of these steps can be done in parallel, however, the variable `n_c` is shared between the threads and so provides a race condition if multiple workers were to try and alter it at the same time. In order to solve this, we can have a variable for each thread and then separately sum these at the end.

Algorithm 8.1. A serial implementation of a Monte-Carlo estimate for  $\pi$  using Equation (8.2).

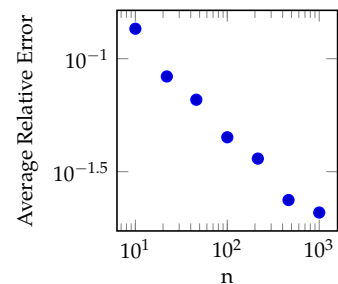


Figure 8.2. The graph shows the estimate of the relative error of the estimated value of  $\pi$ . This graph shows the standard deviation (using 30 repeats) of the quantity  $\frac{\pi'(n) - \pi}{\pi}$  where  $\pi'(n)$  represents an estimate for  $\pi$  constructed using Equation (8.2) with  $n$  samples.

```

function est_pi_mc_threaded(n)
    n_cs = zeros(typeof(n), Threads.nthreads())
    Threads.@threads for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_cs[Threads.threadid()] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end

```

Algorithm 8.2. A basic multithreaded implementation of a Monte-Carlo estimate for  $\pi$  using Equation (8.2).

Here, we have used very little memory to keep track of the counts, and made sure the implementation was thread-safe as each thread accessed a different part of memory. It should be noted here that `Threads.threadid()` returns an ID for the currently executing thread within the threaded loop. Additionally, the `rand()` function call is thread-safe on recent versions of Julia (v1.6+). Note that this implementation, even when seeded, will not be deterministic. Even if each thread had its own seeded RNG, then additional randomness is introduced when scheduling the threads for execution. This is usually not an issue, but it is something that should be kept in mind, especially when one would like to write testable code.

Now, we can benchmark these algorithms and see which is fastest. We can calculate the efficiency based on the number of threads:

```

julia> num_threads = Threads.nthreads()
16

```

The benchmarks:

```

julia> n = 100_000_000;
julia> mc_pi_serial_time = @belapsed est_pi_mc_serial($n)
0.331897864
julia> mc_pi_threaded_time = @belapsed est_pi_mc_threaded($n)
0.16878274

```

```
julia> mc_pi_serial_time/mc_pi_threaded_time
1.9664206423002732
```

One would expect the ratio of those times approach the number of available threads, however, even for this very large value of  $n$ , the parallel efficiency is still very small. This is very likely to be a case of **false sharing**.

### 8.1.1 False Sharing

When a CPU requests data from memory, it is usually gathered as an entire **cache line**, which is usually large enough to gather several adjacent values in memory. If one CPU core acquires this cache line in their L cache, it may store values that other CPU cores are currently writing to, even if the current CPU core is not using it. If the cache line is invalidated by another write (such as incrementing another value), extra processing time has to be spent synchronising the operations.

Let's modify Algorithm 8.2 to conduct an experiment on false sharing by spreading out the memory of each calculation to avoid accessing the same cache lines. We will do this by adding a `spacing` variable, implemented in Algorithm 8.3.

```
function est_pi_mc_threaded_spaced(n, spacing=1)
    n_cs = zeros(typeof(n), Threads.nthreads()*spacing)
    Threads.@threads for _ in 1:n
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        r2 = x*x+y*y
        if r2 <= 1
            n_cs[Threads.threadid()*spacing] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

Algorithm 8.3. An implementation to show the affect of false sharing on performance.

We can clearly see that increasing the spacing between elements has a huge impact on performance. We can see that the performance increases stop around a spacing of 8, suggesting that each cache line is around 512 bits wide.

The main reason that this false sharing is such a huge problem for performance in this problem is due to the frequency at which each core is accessing and writing to that memory. This problem is not as bad if each core is only *reading* memory, as the memory does not change and no synchronisation needs to occur. This is more of our worst case scenario. It should be noted that one should almost **never**

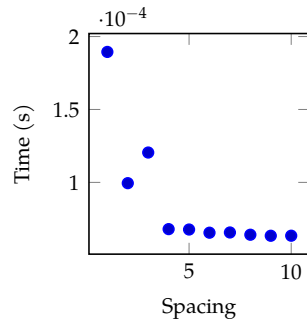


Figure 8.3. Shows the benchmarked timings of the parallel Monte Carlo estimation using Algorithm 8.3 with  $n = 10^6$  and changing the spacing between elements in the storage array.

implement an algorithm like Algorithm 8.3 to avoid false sharing. There are much better methods of doing this, discussed next.

### 8.1.2 Chunking

The problem of false sharing can be avoided by chunking the larger calculation into much smaller ones. An additional benefit of this method is that we can use the original implementation to maximise code reuse. If we make any performance improvements to the serial algorithm, this will also improve the parallel implementation.

The algorithm for this chunking is shown in Algorithm 8.4. A very helpful method for chunking the operations is from the base `Iterators` library called `Iterators.partition` which will break up a collection into a specified number of smaller blocks.

Now, we can benchmark this final algorithm and compare it to the others:

```

julia> mc_pi_threaded_chunked_time = @belapsed est_pi_mc_threaded_chunked($n)
0.021433614
julia> mc_pi_serial_time/mc_pi_threaded_time
1.9664206423002732
julia> mc_pi_serial_time/mc_pi_threaded_chunked_time

```

```

function est_pi_mc_threaded_chunked(n)
    n_threads = Threads.nthreads()
    num_inside = zeros(Float64, n_threads)
    # Calculate maximum chunk size
    chunk_size = div(n, n_threads, RoundUp)

    # Create an iterator and collect to turn into an array
    iter = collect(enumerate(Iterators.partition(1:n, chunk_size)))
    Threads.@threads for info in iter
        i, idx_range = info # Unpack the tuple from enumerate
        n_block = length(idx_range)
        pi_est = est_pi_mc_serial(n_block)
        num_inside[i] = pi_est*n_block/4
    end

    n_c = sum(num_inside)
    return 4 * n_c / n
end

```

Algorithm 8.4. A multithreaded, partitioned, implementation of a Monte-Carlo estimate for  $\pi$  using Equation (8.2).

```
15.484923074568758
```

```

julia> mc_pi_threaded_time/mc_pi_threaded_chunked_time
7.874674798193155

```

We see that the chunked approach was much faster, and almost reached the theoretical maximum performance of  $16\times$ . In general, it is better to split parallel tasks into large chunks that can be sequentially processed by that chunk. This avoids a lot of scheduling and orchestration overhead when managing the threads, as most variables can live inside the stack with little need to coordinate execution. Additionally, we massively reduced the number of writes to memory with the chunked approach as the count could live in registers close to the CPU and only be saved to memory once the bulk of the calculation was completed.

However, it should be noted that the overhead of parallel execution can be significant. The only way to see the effect is to measure the performance relative to the input size  $n$ . We can already assess that this algorithm has a time complexity of  $\mathcal{O}(n)$ . Instead of plotting these lines together, we will plot the  $S$ , compared to the theoretical maximum given by Amdahl's law. Inspecting Figure 8.4, we can see that the chunked implementation approaches the maximum speed-up for this algorithm, but suffers at lower values of  $n$ .



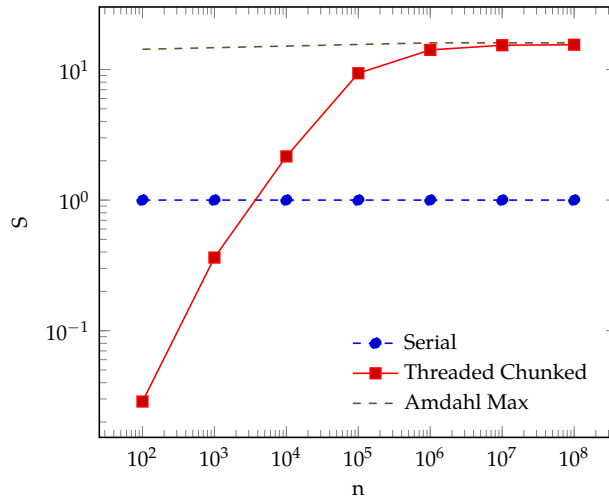


Figure 8.4. Shows the relative speed-up  $S$  of using Algorithm 8.4 (chunked parallel) over the serial implementation Algorithm 8.1. We have also plotted the theoretical maximum performance increase given by Amdahl's law, given by Equation (7.2).

One can infer that the cost of using multithreading is quite high, especially when the contents of the for loop are computationally inexpensive. However, this cost is more or less constant and increasing the throughput will minimise relative size of this cost. In ??, we see that it takes at least  $10^5$  to  $10^6$  samples to make the switch worth it, as the overhead of the threaded approach is much greater than the time taken to perform the calculations.

## 8.2 Task-based Parallelism

It is important to acknowledge that using the `Threads.@threads` macro is not the only way to achieve parallelism in Julia. There are other libraries that provide their own macros such as `LoopVectorization.jl`, which provides the `@tturbo` for thread-based speed-ups. However, one can also manage the threads directly, using `Threads.@spawn`.

`Threads.@spawn` works by creating a task that is scheduled to run on any available thread. This returns a task object that can be used to gather results or for synchronisation between the threads. For example, we can write the chunk based parallelism using the `Threads.@spawn` convention, with a bit more boilerplate code. The example is written in Algorithm 8.5.

```

function est_pi_mc_spawn(n)
    num_threads = Threads.nthreads()
    block_size = div(n, num_threads, RoundUp)

    task_handles = Vector{Task}(undef, num_threads)
    for i in 1:num_threads
        if i == num_threads
            num_darts = n-block_size*(num_threads-1)
        else
            num_darts = block_size
        end

        task_handles[i] = Threads.@spawn est_pi_mc_serial($num_darts)*$num_darts/4
    end

    n_c = 0.
    for task in task_handles
        n_c += fetch(task)
    end

    return 4 * n_c / n
end

```

Algorithm 8.5. A task-based multithreaded implementation using Equation (8.2) as an estimate for  $\pi$ .

One can see that we now have two `for` loops, the first actually spawns the threads to do the work, and the second has to fetch the results from each task handle. Spawning the task simply schedules it to commence, but fetching it waits for the thread to finish. This is a much lower-level approach to threading, and can be useful when you do not know how many times one needs to loop. One will also notice that some values were escaped using the `$` character, this makes the macro evaluate and print the value of the chosen variable directly into the expression to allow for better type analysis.

We can call this method to see it works, and compare it to the threaded and chunked version we previously coded.

```
julia> n = 1_000_000
1000000
julia> @btime est_pi_mc_threaded_chunked($n)
 229.932 μs (98 allocations: 8.77 KiB)
3.144808
julia> @btime est_pi_mc_spawn($n)
 228.148 μs (131 allocations: 8.69 KiB)
3.141232
```

One can see that the performance is slightly better in this example, but at the cost of added complexity. Using `Threads.@spawn` should be done very sparingly, as it can very easily introduce errors at runtime. It is strongly recommended that one sticks to using the `Threads.@threads` macro for multithreading parallelisation, and only look at using the `Threads.@spawn` if it cannot be implemented otherwise, or one finds that the performance is much better otherwise. For most use-cases, `Threads.@threads` is as fast or even faster than an equivalent `Threads.@spawn` implementation.

### 8.3 Packages

A very helpful package for cleaning up the multithreaded implementations is the `ThreadsX.jl` package. This provides a few functions which can help to write concise code for implementing common operations in parallel. Some common, helpful operations are

- `map` and `map!`
- `reduce`

- `mapreduce`
- `sum`
- `findall`
- `findfirst`
- `findlast`
- `foreach`
- `sort`
- `sort!`

Another package which can be used is *Transducers.jl* which allows for piping operations to be easily composed. Let's take an example of the Monte-Carlo  $\pi$  estimation. We will create a function which "throws a dart" and returns a "hit" or a "miss" encoded as 1 or 0.

```
square(x) = x*x
throw_dart() = square(rand()) + square(rand()) <= 1
```

Algorithm 8.6. A simple implementation of a single inner loop of Algorithm 8.1.

We can create a lazy mapping of values using the pipe operator `:|>`. This array is then passed into the `sum` function:

```
julia> using Transducers
julia> n = Int(1e6);
julia> pi_est_transducers(n) = 4 * sum(1:n |> Map(_ -> throw_dart())) / n;
julia> @btime pi_est_transducers($n)
 4.352 ms (0 allocations: 0 bytes)
3.14222
```

As `sum` is just a reduction operation, we can also use the generic, threaded, reduction method `foldxt` from *Transducers.jl* to perform the same operation in parallel:

```
julia> pi_est_transducers_t(n) = 4 * foldxt(+, 1:n |> Map(_ -> throw_dart())) / n;
julia> @btime pi_est_transducers_t($n)
 315.496 μs (563 allocations: 19.59 KiB)
3.140908
```

Notice that this is a nearly perfect speed-up, without having to use a chunking approach. Additionally, this uses much less code than before, while being of a similar performance.

Check out the documentation for both of these packages to see what they can be used for. However, it should be noted that most of this functionality is mostly only for syntax. One can write the same code in pure Julia, without using these packages.

## 8.4 Summary

### 8.4.1 Advantages

- Multithreading uses shared memory. Results from each thread can easily be shared with one another, without the need of explicit communication.
- Shared memory allows data to be stored only in one place, reducing the memory footprint. Each worker does not need a copy of the data it is working on.
- Each thread has access to the local runtime of the process in memory, including all libraries and pieces of code, instead of each thread having a copy of the local process. This significantly reduces memory overhead, and more importantly the **latency**, of this pattern.
- Latency of utilising multiple threads is usually the fastest of all the parallel paradigms covered in this book, except for hardware level parallelism (hardware SIMD instructions).
- If a language has a multithreading implementation (like Julia or C#), it is usually the easiest approach to leverage to allow a program to execute in parallel.

### 8.4.2 Disadvantages

- Multithreading can only be scaled up to the size of one machine, the number of threads limited by the number of logical cores on a CPU.

- Shared memory introduces the danger of race conditions occurring in your code, in some cases requiring the need for atomics, mutexes and semaphores, drastically increasing the complexity of some code.
- Some algorithms will need a complete redesign to effectively use the multithreading paradigm.
- Latency of spinning up additional threads and coordinating execution across all threads incurs a significant overhead cost. Unless the computation can be effectively parallelised, using multithreading may even **slow down** one's code.
- Execution in parallel necessitates an unpredictable order of execution, which often times greatly impacts the ability of the compiler and the runtime to pre-empt which memory is needed often and therefore has worse cache management than a serial approach. This can be mitigated, but usually requires an extension.
- Existing routines or libraries used by your code may not be "thread-safe", which means they should **not** be executed concurrently. If two parallel pieces of code run something that is not thread-safe, it could lead to unpredictable results at best, memory corruption and runtime errors at worse. Additional care by the developers must be taken to ensure that using multithreading is safe. Sometimes, if the non thread-safe code is essential, this paradigm may not be available.
- Experiments that rely on a deterministic and predictable outcome, such as ones that use random number generation with a seeded pseudo random number generator often require intervention to make sure that results can be duplicated. This may involve re-writing large parts of the code base to ensure reproducibility.
- Each thread usually requires its own stack and memory space in order to perform calculations. Additionally, used by each thread is often not allowed to overlap due to the risk of race conditions, and so more memory is usually required.

## 8.5 Exercises

Before beginning the exercises, check that your computer has multiple cores, and then run Julia with access to additional threads, either by setting the environment variable `JULIA_NUM_THREADS` and restarting Julia, or by running Julia with the command `julia -t 4`, which starts the Julia REPL with 4 threads<sup>2</sup>. The number of threads used should be set to the number of logical cores on your computer. One can check the number of available threads by using `Threads.nthreads()` in the REPL.

<sup>2</sup> Note that one can also use `auto` as an option for the number of threads to use all logical cores.

**Exercise 8.1.** Consider the function:

```
function generate_random_numbers(k)
    """Generates n random numbers"""
    for _ in 1:k
        rand()
    end
    nothing
end
```

This function can simulate a variable amount of work. This function can be called via `generate_random_numbers(k)`, where  $k$  is chosen to mimic short or long workloads. One can imagine having to repeat this work  $n$  times. Empirically measure the time taken for these variable work loads as a function of  $n$  with and without `Threads.@threads`. Use these times to create a plot resembling ?? to show the performance of the multithreaded task and serial task vs task size ( $n$ ). Reproduce this graph for varying workload size,  $k$ , e.g.  $k = 10$  (short),  $k = 10^5$  (medium) and  $k = 10^9$  (long), adjusting the range of  $n$  used to compensate for increased workload of the loop.

**Exercise 8.2.** Continuing on the results from the previous question, assuming that  $n$  is equal to the number of available threads, what is the minimum workload size needed to see improvements from using multithreading.

**Exercise 8.3.** Write your own multithreaded `mapreduce` implementation, using the same format as the following serial implementation:

```
function custom_mapreduce(map_fn, reduction_op, array)
    @assert length(array) >= 2
    mapped_results = map_fn.(array)
    acc = reduction_op(mapped_results[1], mapped_results[2])
end
```

```

    for i in 3:length(array)
        acc = reduction_op(acc, mapped_results[i])
    end
    return acc
end

```

One can assume that the reduction operation is associative.

**Exercise 8.4.** Take the following function:

```

function randomised_workload(min_time, max_time)
    # Generate a random amount of time to wait
    time = rand() * (max_time-min_time) + min_time
    sleep(time)
    nothing
end

```

This will allow you to simulate a process whose competition time is uniformly random. Use this in an experiment to compare the effectiveness of chunking a serial implementation (as in Algorithm 8.4) vs a simple threaded implementation when the workload size is randomised. How large does the workload size variability need to be for the simpler implementation to be more effective?

**Exercise 8.5.** Imagine a function  $f(n)$  whose runtime can be predicted via  $c(n) = an + b$ , where  $a$  and  $b$  are real positive constants and  $a \gg b$ . One needs to evaluate  $f(n)$  at  $n = 1, 2, \dots, 9999, 10000$ . If one has  $k$  workers available to calculate the results of  $f(n)$  for each of the required  $n$ , how must the tasks be scheduled amongst the workers to minimise runtime? Repeat the same analysis, now using  $c(n) = an^2 + b$ .

Note that answers need not be exact, just more efficient than a naive scheduler.



## 9 *Multiprocessing*

In the last chapter, we talked about scheduling tasks across different threads in a program, all of which have access to the same set of shared memory. Multiprocessing takes a different approach, and isolates each worker in their own process. Isolation means that each worker cannot access another's memory. Any sharing of information has to be done via communication between each process. This communication can happen over sockets if each process lives on the same machine. If the processes exist across multiple nodes, this communication is facilitated via network communication over ports.

In multiprocessing, one starts  $n$  processes independently, all of which have their own isolated memory space for storing a copy of their code as well as space for the heap and the stack. One must implement some framework or library to facilitate communication of these different processes. The way in which the processes are connected is called the **topology**. There are different models that can be used to structure the processes, but one of the most common is the **master-worker** configuration, which has one central process (the **master**) which coordinates the **worker** processes.

Multiprocessing is the commonly found parallel programming paradigm in languages like MATLAB and Python. In MATLAB, using `parfor` instead of `for` will distribute the calculation of the inner loop across multiple processes. While this is a very easy way to parallelise code, if one does not have a parallel pool (i.e. a group of MATLAB processes) running, it can take several **minutes** to start executing the code. Multiprocessing can have a huge latency overhead, as each worker has to start an entirely new process. This involves loading an entire copy of the language runtime, and orchestrating the communication channels between each process so that tasks can be appropriately scheduled. In the case of MATLAB,

you are loading up an entire copy of the MATLAB runtime (excluding the GUI) for each worker requested.

Fortunately, Python's runtime is much smaller, and so the overhead is actually very small when using multiprocessing. Python<sup>1</sup> must use multiprocessing due to the built-in Global Interpreter Lock (GIL), which interferes with threads that try to run at the same time, effectively allowing only *concurrency* but not *parallelism* inside a single process.

<sup>1</sup> As in the reference *CPython* implementation.

In Julia, the standard way to use multiprocessing is via the *Distributed.jl* package, included in the base library.

We can summarise the main traits of multiprocessing below:

- A multiprocessing paradigm uses several copies of the runtime to perform tasks in parallel. Each copy in a separate process, unable to access the memory space of another process.
- Any communication between processes must occur via sockets or over the network, and is usually very high latency. This includes sharing the results of calculations.
- Each process can have access to multiple threads allowing the combination of multiprocessing and multithreading paradigms.
- All the processes used need not exist on the same machine, but instead, can be spread across several machines. We often refer to a group of machine networked together as a **cluster**, and is usually where multiprocessing is used the most.

## 9.1 Multiprocessing in Julia

The easiest way to start using multiprocessing in Julia is to use the *Distributed.jl* package. This is the standard package to facilitate the multiprocessing paradigm. To add the package, type the following into the REPL:

```
using Pkg; Pkg.add("Distributed");
```

As with multithreading, we should check how many processes are available to do work. This is done with the command:

```
julia> using Distributed;
julia> nprocs()
```

```
1
```

By default, Julia launches only a single process when running. There are a few ways in which we can use more, the simplest is to use the `addprocs` function:

```
n_processes = 16;
addprocs(n_processes);
```

This will spawn `n_processes` locally to act as workers. The total number of processes with `nprocs()` will be one higher as this includes the master control process. We can check the number of processes and workers available again to make sure:

```
julia> nprocs()
17
```

```
julia> nworkers()
16
```

One can start Julia with multiple processes using `julia -p 3`, which will start Julia with 4 total processes. Be aware that one should also run the following, to make sure that each worker has access to all the packages:

```
@everywhere using Pkg; Pkg.activate(".");
```

The `@everywhere` macro execute the command on all workers.

The bread and butter of *Distributed.jl* are the following macros and functions:

- `addprocs(n)` - Spawns `n` available processes. This command launches processes locally by default, but can also be used to launch processes on other machines via SSH.
- `@everywhere` - A macro that runs the subsequent expression on all connected processes. This is most useful for `using` statements and `include` calls to load function definitions on the workers.
- `pmap` - A multiprocessing, parallel implementation of the standard `map` function in Julia. This enables easy use multiprocessing for an embarrassingly parallel problem.

- `@distributed` - A macro which is useful for parallelising a for loop, with an optional, built-in, mechanism for performing reductions.
- `@sync` - A macro that halts progression until all enclosed uses of `@async`, `@spawn`, `@spawnat` and `@distributed` are complete.

Let's start off with implementing Algorithm 8.4 in the multiprocessing paradigm. We can start off by having an array of numbers representing the number of darts to throw on each worker. Once this array is calculated we can map those numbers onto an estimate of  $\pi$ . The exact implementation is shown in Algorithm 9.1.

```
function est_pi_mc_multiprocessing_chunked(n)
    num_workers = nworkers()

    iter = Iterators.partition(1:n, num_workers)
    # Create a mapping function from a range to an estimate
    _f(range) = est_pi_mc_serial(length(range)) * length(range) / 4
    num_inside = pmap(_f, iter)

    n_c = sum(num_inside)
    return 4 * n_c / n
end
```

Algorithm 9.1. A chunked multiprocessing implementation of a Monte-Carlo estimate for  $\pi$  using Equation (8.2).

If we try and use this algorithm straight away, we will encounter an error since the function, `est_pi_mc_serial`, does not exist on all processes. This is a common error when people use the multiprocessing paradigm, as we have to **load** the function definitions on each worker before they are used. To simplify the processes, we can abstract away all the functions in a separate file/module and load the file. In the case of this book, I have called this file `support_code.jl`. We can give all processes access to these functions by running the command:

```
julia> @everywhere include("support_code.jl");
```

Now that we have the code loaded, we can benchmark the multiprocessing approach:

```

julia> n = 100_000_000;
julia> mc_pi_mp_chunked_time = @belapsed est_pi_mc_multiprocessing_chunked(\$n)
0.0267902
julia> println("MP Speedup vs Serial: ", mc_pi_serial_time/mc_pi_mp_chunked_time)
MP Speedup vs Serial: 15.041112048435622
julia> println("Threaded/MP Relative Speed: ", mc_pi_threaded_chunked_time/mc_pi_mp_chunked_time)
Threaded/MP Relative Speed: 0.9632328239430836

```

We can see that the performance difference between the multithreaded and multiprocessing implementations is mostly negligible, as  $n$  is very large, and the calculation is dominated by time spent in the loop. Notice that the multiprocessing implementation was slower than the multithreaded implementation, which is typical in these cases. One should remember that latency costs are much higher in multiprocessing than in multithreading.

Since we are effectively performing a map of empty arguments into the sum of darts in and out, we can look at an alternative implementation using the `@distributed` macro:

```

function est_pi_mc_multiprocessing(n)
    n_c = @sync @distributed (+) for _ = 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        r2 <= 1 # Last line indicates term to reduce
    end
    return 4 * n_c / n
end

```

Algorithm 9.2. A simple multiprocessing implementation of a Monte-Carlo estimate for  $\pi$  using Equation (8.2). Uses the `@distributed` macro to perform a map-reduce on a Monte-Carlo process.

Running this implementation, we get:

```

julia> mc_pi_mp_time = @belapsed est_pi_mc_multiprocessing(\$n)
0.0301067
julia> println("MP Simple/Chunked Relative Speed: ", mc_pi_mp_chunked_time/mc_pi_mp_time)
MP Simple/Chunked Relative Speed: 0.8898417960121834

```

We see that this implementation is much simpler, but is about 10% slower than the chunked implementation. This makes sense as the bulk of the reduction in the chunked case is done locally and can be locally optimised and cache-friendly.

Meanwhile, the simple approach requires the coordinated reduction of  $n$  separate elements.

### 9.1.1 Suggested multiprocessing pattern

During development, one may switch between using serial, multithreading and multiprocessing patterns. For this reason, it is often best to write the inner loop of an algorithm in a separate function. This avoids having to maintain several similar copies of implementations of the bulk of your code. For this reason, we suggest developing your code in the following way:

1. Write all inner loop functions, e.g. a single run of a Monte-Carlo simulation, inside a single (or small number of) file(s). Make sure you have a single file (which could simply include various other files), which loads all function definitions of any functions that may be run in parallel.
2. Have a separate file for executing your code, which runs all the `@everywhere` macros. This file should run `@everywhere include("allfunctions.jl")`, where `"allfunctions.jl"` is the relative path to the file with all function definitions that are needed by the parallel code.
3. Make use of packages like *Transducers.jl* or *Folds.jl*.

### 9.1.2 General Parallel Pattern

As a quick example, let's write a function which emulates `pmap` with a custom flag to indicate whether to use multithreading, multiprocessing or serial implementations:

We can then implement our naive Monte-Carlo algorithm with a single implementation:

We can benchmark each type of parallelism on the same algorithm to make sure it works:

```
julia> n = 1_000_000;
julia> @btime estimate_pi($(SerialEx()), $n)
 7.852 ms (8 allocations: 224 bytes)
3.14236
julia> @btime estimate_pi($(MultithreadingEx()), $n)
541.300 μs (98 allocations: 10.81 KiB)
```

```

# Declare an enum for the different map types
abstract type AbstractExecutionMethod end
struct MultiprocessingEx <: AbstractExecutionMethod end
struct MultithreadingEx <: AbstractExecutionMethod end
struct SerialEx <: AbstractExecutionMethod end

custom_map(::MultiprocessingEx, mapping_fn, c...) = pmap(mapping_fn, c...)
function custom_map(::MultithreadingEx, mapping_fn, c...)
    # Do some work to infer return type of function
    return_types = Base.return_types(
        mapping_fn,
        typeof(c)
    )

    if length(return_types) == 1
        return_type = return_types[begin]
    else
        return_type = Union{return_types...}
    end
    # Create a container to store results
    container = Array{return_type}(undef, size(c))
    Threads.@threads for i in eachindex(arguments)
        container[i] = mapping_fn(arguments[i]...)
    end
    return container
end
custom_map(::SerialEx, mapping_fn, c...) = map(mapping_fn, c...)

# Change the default implementation in your code
custom_map(mapping_fn, c...) = custom_map(SerialEx(), mapping_fn, c...)

```

Algorithm 9.3. An example way of implementing a custom map function which can switch between multiprocessing, multithreading and serial maps. Note that there are existing packages which better implement this functionality.

```

function throw_darts(n)
    total = zero(typeof(n))
    for _ in 1:n
        total += throw_dart()
    end
    return total
end
get_blocks(::SerialEx) = 1
get_blocks(::MultithreadingEx) = Threads.nthreads()
get_blocks(::MultiprocessingEx) = nworkers()
function estimate_pi(parallel_type, n)
    n_blocks = min(n, get_blocks(parallel_type))
    iter = Iterators.partition(1:n, n_blocks)
    num_darts = [length(iter) for range in iter]
    dart_results = custom_map(parallel_type, throw_darts, num_darts)
    return 4 * sum(dart_results) / n
end

```

Algorithm 9.4. A simple implementation using the custom map paradigm, shown in Algorithm 9.3. This will perform a chunked implementation of a Monte-Carlo algorithm which estimates  $\pi$ . It is vital that `throw_dart` and `throw_darts` are loaded on every single worker. The implementation of `throw_dart` comes from Algorithm 8.6.

```

3.142464
julia> @btime estimate_pi($(MultiprocessingEx()), $n)
1.387 ms (872 allocations: 36.80 KiB)
3.142812

```

These implementations are not always optimal, and one can specialise an implementation that can be improved using a different method. For example, if the input is smaller than a certain size, one will always use the serial version of the code.

## 9.2 Multiprocessing on a Cluster

Multiprocessing is your go-to parallelisation method on a cluster. A cluster is a collection of networked machines (often times referred to as “nodes”), which are usually similar in hardware architecture. A cluster typically gives you access to hundreds, if not thousands of CPU cores. As clusters involve many machines, it can often be very difficult to network each of the processes together and facilitate communication. Fortunately, there is a package, *ClusterManagers.jl*, which removes most of the hard work.



In this book, we will focus on using the SLURM<sup>2</sup>. Other scheduling systems may have support inside of *ClusterManagers.jl*, and it is likely trivial to switch to a different cluster.

<sup>2</sup>SLURM Workload Manager - <https://slurm.schedmd.com/>

As a starting point, let's introduce a bash script, which is used to request resources from the scheduler:

```
#!/bin/bash

#SBATCH --ntasks=32
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=2048
#SBATCH --time=00:10:00
#SBATCH -o test_job_%j.out
```

```
julia --project run_code.jl
```

This will request resources for 32 CPUs and around 64 GB of memory for a maximum time of 10 minutes. It also starts the process with running the *run\_code.jl* julia file. We can take a look inside to see the process for setting up the nodes:

```
using Pkg
using Distributed
using ClusterManagers
using BSON

println("Setting up SLURM!")
num_tasks = parse{Int, ENV["SLURM_NTASKS"]}
cpus_per_task = parse{Int, ENV["SLURM_CPUS_PER_TASK"]}
ENV["JULIA_NUM_THREADS"] = cpus_per_task
current_project = dirname(Pkg.project().path)
addprocs(SlurmManager(num_tasks);
    exeflags=[
        "--project=\"$current_project\"",
        "--threads=$cpus_per_task"
    ]
)
println("Workers: $(length(workers()))")

@everywhere include("allfunctions.jl")

parameters = get_parameters()
```

```
results = get_results(parameters)
BSON.@save "results.bson" results
```

At first, we have to perform a bit of prep work, by adding a separate process for each CPU and calling the correct cluster manager - `SlurmManager`. This also sets the number of threads Julia can use, such that each process has the right amount of threads. We have a separate file which defines the functions we need in our experiment. The `get_results` function internally calls functions like `pmap` which make use of multiprocessing.

We make use of the `BSON.jl` library to serialise our results and save them in a file. BSON is similar to JSON but stores the data structure in binary, instead of in plain-text. This allows a compressed storage of information, at the cost of losing human readability. One does not need to implement their own serialisation and de-serialisation methods, but instead make use of `BSON.jl` through the `@load` and `@save` macros.

### 9.3 Message Passing

Arguably the most common approach to multiprocessing is by using a message passing approach. MPI (Message Passing Interface) is a standardised specification of facilitating message-passing for parallel computing in distributed systems. As this is only a specification, there exist several implementations such as `OpenMPI` and `MPICH`. Some of these implementations are specialised for the network hardware used on a specific cluster.

The MPI approach has the developer write a program which will execute simultaneously across each process. The MPI library exposes routines which allow each process to be able to identify itself and send data to and from each process. Additionally, the MPI library provides a barrier function which allows all the processes to be able to synchronise their execution.

In Julia, one can use the `MPI.jl` package to follow this programming model which may be effective for your use case. However, `Distributed.jl` remains the standard for multiprocessing, as it has a much less verbose programming model, making it easier to get started.

## 9.4 Summary

### 9.4.1 Advantages

- As each process need not live on the same machine, a multiprocessing approach can be practically scaled up to almost any size. This means it is better able to utilise huge amounts of compute via supercomputing clusters (i.e. your local HPC).
- As each process must use its own memory, race conditions are much rarer. Lack of race conditions eliminates the needs for locks and can improve performance.
- Each process can use libraries and resources that are not thread-safe in parallel.

### 9.4.2 Disadvantages

- Each task must be communicated to each process, including all dependencies such as functions and data. Functions and required data used by each worker must be loaded on every individual process.
- Multiprocessing implementations usually require external libraries to work effectively.
- Cannot share memory between nodes, care must be taken to explicitly send and request information between nodes.
- More resources, particularly memory, are required to support all the individual processes<sup>3</sup>.
- As with all parallel approaches, ensuring reproducibility can be very tricky when using multiprocessing.
- With Julia in particular, due to the JIT compiled runtime, each process must compile all the code required to run their assigned tasks, which is a lot of redundant processing. This can be reduced with efforts in precompilation, but should be mentioned.

<sup>3</sup> In the case of MATLAB, each process can take around 1 to 2GB of memory before any data is loaded. If you have many cores, you can only utilise them if you have enough memory for the additional processes.

## 9.5 Exercises

**Exercise 9.1.** Take the following workload that scales linearly:

```

function example_inner_loop(k)
    s = zero(Float64)
    for _ in 1:k
        s += rand(Float64)
    end
    s
end
function example_workload(n, k=100)
    results = Vector{Float64}(undef, n)
    for i in 1:n
        results[i] = example_inner_loop(k)
    end
    return results
end

```

(i). What does the parameter `k` control? (ii). Implement a multithreaded version of `example_workload`. (iii). Implement a parallel version of `example_workload` using multiprocessing. (iv). Design an experiment to compare the two implementations of `example_workload`, and the original, serial, implementation, as you vary `n`. (v). Use the results of the previous experiment to estimate the latency relationship of both `Threads.@threads` and `Distributed.pmap` to the variables `n` and `k`.

**Exercise 9.2.** If you have an embarrassingly parallel problem, which relies on non-thread safe code, which parallel programming paradigm is most suitable?

*Solution:* Multiprocessing - as each process will have its own isolated memory copy which can ensure parallel processing without race conditions.

**Exercise 9.3.** Look at the following struct:

```

struct RunningStats{T}
    min::T
    max::T
    mean::T
    num_samples::Int
end

```

(i). Define a function which reduces two samples of type `T` into a single `RunningStats`. (ii). Extend the previous function to define reductions between single values of type `T`, and `RunningStats`, in any order. (iii). Test this custom reduction using the `reduce` function and the `Statistics.jl` on a vector of random floats. Calculate the statistics of this vector and compare them to the reduced statistics output. (iv).

Write a multiprocessing parallel implementation of the reduction calculated in the previous part, using a Monte-Carlo process of your choice which produces a single value.

**Exercise 9.4.** Take the following setup:

```
using Random
using LinearAlgebra
using Statistics

const rng = Random.Xoshiro(1234)
function generate_seeds(rng, n)
    seeds = zeros{Int, n}
    for i in 1:n
        seeds[i] = rand(rng, Int)
    end
    return seeds
end
const n = 32
const seeds = generate_seeds(rng, n)

function long_calculation(seed; l=10)
    c_rng = Random.Xoshiro(seed)

    k = 12
    s = 2^k
    matrix_size = (s,s)

    matrix = rand(c_rng, Float64, matrix_size) .* 2 .- 1
    cache = similar(matrix)

    for j = 1:l
        mul!(cache, matrix, matrix)
        matrix .= cache ./ s .- matrix
    end

    return Statistics.mean(cache)
end

const results = map(long_calculation, seeds)
```

(i). Use multiprocessing to parallelise this calculation and make sure the number of BLAS threads is set to 1. (ii). Implement memoised caching on the

long calculation, which uses the disk as a cache. **(iii)**. Use the previous parts to create a script which can recover from failure, without losing too much progress.

## 10 Introduction to GPU Programming

As mentioned in Section 3.1.5, this chapter will be a deeper dive on using GPUs for computational workloads. While GPUs were originally developed for accelerating rendering workloads, their massively parallel architecture lends itself to accelerating scientific workloads.

### 10.1 Modern GPU Hardware

Modern GPUs are not limited to graphical applications, such as gaming and 3D rendering, but can also be very effective at accelerating scientific simulations. A modern user with access to a consumer level GPU (e.g. shown in Figure 10.1) now has access to a comparable level of compute as supercomputing clusters from the early 2000s (as measured by peak FLOPS<sup>1</sup>).

A GPU is a type of co-processor which connects to the CPU via the motherboard. This connection is usually through a PCIe bus<sup>2</sup> and often requires a separate power connection to run. The GPU has its own on-board memory and contains many processor cores, typically on the order of 1000s.

<sup>3</sup> Many modern high performance clusters now include many GPUs. Take the Sulis HPC based at the University of Warwick as an example. Sulis has 90 *NVIDIA A100 Tensor Core GPUs*, which can cost upwards of £20,000 each. Let's take a closer look at the hardware specifications of one of these cards:

- The A100 supports the PCIe Gen4 standard which is capable of a maximum of 64 GB/s memory bandwidth between the GPU and the CPU.
- The A100 has 40 GB of HBM2<sup>4</sup>.



Figure 10.1. The NVIDIA 3090 GPU, designed for high-end consumer desktop machines.

<sup>1</sup> Floating-Point Operations Per Second.

<sup>2</sup> A PCIe (Peripheral Component Interconnect Express) bus is a standard, high-speed, interface for connecting external hardware to a motherboard

<sup>3</sup> R. Krashinsky, *NVIDIA Ampere Architecture In-Depth*, <https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/>, Accessed on 15th January 2023, 2020.

- The A100 has a theoretical maximum Single-Precision Performance of 19.5 TFLOPs<sup>5</sup>. Single-Precision (FP32) refers to 32 bit floating point numbers, with 8 bits to represent the exponent (the range) and 23 bits for the mantissa (the precision).
- This card also supports using the Tensor Float (TF32)<sup>6</sup> standard which uses the same number of bits as FP32 for the exponent, while only using 10 bits for the mantissa, like FP16. The TF32 precision allows for reaching a maximum of 156 TFLOPs of performance.

<sup>5</sup> Teraflops -  $10^{12}$  floating-point operations per second.

<sup>6</sup> P. Kharya, "TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x," 2020.

If we assume that each CPU thread is capable of performing a single floating point operation per clock cycle and runs at 4 GHz, and a CPU has 64 cores and 128 threads, we can estimate that this machine is roughly capable of 0.512 TFLOPs of performance. While this calculation is not completely accurate, it is clearly at least an order of magnitude slower than our GPU example.

In Figure 10.3, we can see a rough depiction of the structure of the A100 chip. This contains many thousands of individual processors along with separate on-device memory. Processors are grouped into SMs (Simultaneous Multiprocessors), which are also grouped into blocks. There are multiple separate memory chips with their own memory controllers to facilitate both a high capacity of memory and a high bandwidth.

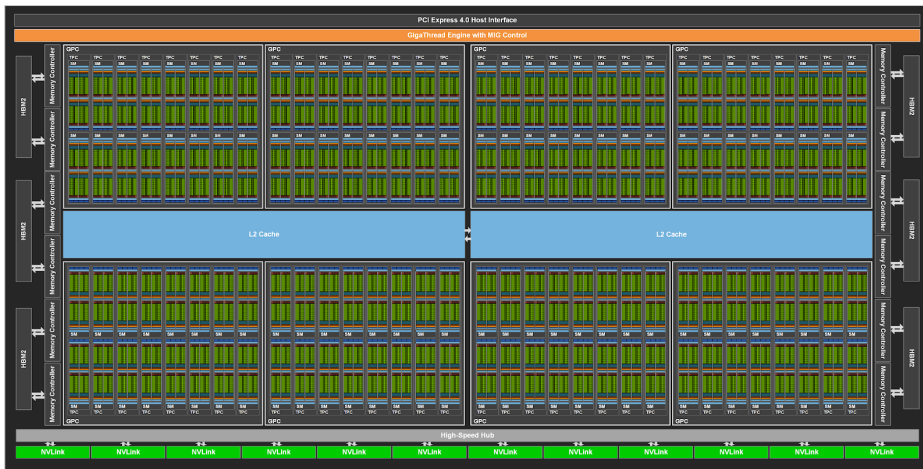


Figure 10.3. This shows a hardware diagram of the A100 GPU. One can immediately see there are many individual processors (shown in small green blocks), which are called CUDA cores or SPs. Additionally, one can see that the GPU has its own dedicated memory on-device, along with shared cache layers between the processors. Processors exist in a group called an SM - Simultaneous Multiprocessor.



The diagram above can be zoomed in to show a single SM unit, as seen in Figure 10.4. This SM contains its own lower level caches, independent of other SMs. Additionally, each unit has its own set of registers allowing them to execute independently. The processing units (in green), are optimised for high throughput arithmetic, and are not as capable as a generic CPU core. However, the purpose of these cores is to maximise throughput, not generality or latency.

## 10.2 GPU Vendor Overview

Each manufacturer has differing architectures and programming models to support their GPUs. The most popular of these is NVIDIA and the CUDA programming model, which is NVIDIA's proprietary platform for programming and using their GPUs. AMD has ROCm, which is a software stack for programming on their GPUs. Intel has also been developing oneAPI, which they intend to be useful across different hardware applications.

The main difference between CUDA and the other software stacks is that CUDA is proprietary, unlike the open source ROCm and oneAPI. There is also the OpenCL project which aims to provide an API for programming across many devices, including GPUs from different manufacturers and even CPUs.

For this book, we are choosing CUDA as our vehicle to explore GPU programming, as it is the most fully-featured and supported of the software stacks. Many modern HPC clusters will have NVIDIA GPUs available, and they have the vast majority of the market share of the desktop GPU market.

Skills from learning to use CUDA, should be applicable to other software stacks, but know that there is a learning curve when transitioning from one platform to another.

## 10.3 High Level Introduction to CUDA

CUDA is an API primarily designed to work with languages such as C, C++ and Fortran. However, the Julia ecosystem has maintained, developed and built the *CUDA.jl*<sup>7</sup> package. We will extensively use this package to interact with our GPU.

<sup>7</sup> T. Besard, C. Foket, and B. De Sutter, "Effective Extensible Programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2019.



Figure 10.4. A single SM showing multiple cores contained within.

### 10.3.1 Installing *CUDA.jl*

Begin by verifying your current machine has an NVIDIA GPU. On Windows, one can use the Task Manager to do this:

- Open Task Manager (Ctrl+Shift+Esc).
- Expand to “More Details” with the button in the lower left-hand corner.
- Switch to the “Performance” tab
- In the left-hand column, there should be an option called GPU. Check the name of the GPU and ensure it says “NVIDIA” and note the device name.
- If an older card, or a mobile card for a laptop, search online<sup>8</sup> for the CUDA Compute Capability of your card (e.g. the *NVIDIA 1080Ti* has a compute capability of 6.1.)

<sup>8</sup> A list can be found here - <https://developer.nvidia.com/cuda-gpus>.

On other operating systems, one can use the command line tool - “*nvidia-smi*” - to check the installed device. If this is not available, then you likely do not have an NVIDIA GPU, or you have not installed the NVIDIA proprietary drivers. If one has access to a machine with an NVIDIA GPU, you can remote into the machine with SSH and follow along there.

If you have a much older machine, you should find out the compute capability of your device. A low compute capability can severely limit the available features one can use. Anything above a CUDA Compute Capability of 5 should be enough for our purposes in this book.

To add the package, run:

```
using Pkg; Pkg.add("CUDA")
```

This will add the Julia CUDA libraries, however, it will not install the necessary NVIDIA libraries. Fortunately, instead of having to manually install the CUDA SDK, Julia’s package manager enables artefacts to be downloaded that can replace this step. These artefacts are known to be compatible with *CUDA.jl* and only need to be downloaded once. To start this process, run the following:

```
using CUDA
CUDA.versioninfo()
```

This will download all the necessary dependencies needed for *CUDA.jl* to function.

**(Optional):** If you want to make sure everything will work correctly on your machine, you can run the tests for the package using:

```
using Pkg;
Pkg.test("CUDA")
```

However, one should make sure that `Threads.nthreads()` is greater than 1 for this test as it can take a **very long time** to complete, even with multiple threads.

### 10.3.2 First Steps with `CUDA.jl`

Once installed, we can begin to explore how to use CUDA. Let's test out a vector addition by using the in-built `map!` function.

```
julia> n = 1_000_000;
julia> a = rand(Float32, n);
julia> b = rand(Float32, n);
julia> c = similar(a);
julia> array_test!(c, a, b) = map!(+, c, a, b)
array_test! (generic function with 1 method)
julia> @btime array_test!($c, $a, $b);
467.979 μs (0 allocations: 0 bytes)
```

Here, we are storing the result of the addition of `a` and `b` in the array `c`.

In order to use the GPU, we need only convert the CPU arrays (`a`, `b` and `c`) into CUDA arrays, using the `cu` function:

```
julia> a_gpu = cu(a);
julia> b_gpu = cu(b);
julia> c_gpu = similar(a_gpu);
```

This function copies an array over into the local memory on the GPU. We can now perform the same operation, but instead using our GPU arrays:

```
julia> array_test!(c_gpu, a_gpu, b_gpu);
```

Note that this will take a long time on the first execution due to the compilation. After the first run, it should be fast.

We can copy the `c_gpu` array back to the CPU using `Array` to compare the results to see if we get the correct answers:

```

julia> using Test
julia> c_cpu = Array(c_gpu);
julia> @test isapprox(c, c_cpu)
Test Passed

```

We use the `isapprox` function since we are likely to encounter some floating point differences between the CPU and the GPU. This function ensures that the values are within acceptable limits to be considered equal.

One thing to notice here is that we did not change the function that performed the calculation. Instead, we only used broadcasting (via the `map!` function) and regular Julia functions. Under the hood, `CUDA.jl` defines specialised functions for performing these operations and is able to compile a kernel<sup>9</sup> from the native Julia code. This is one of the core strengths of using Julia to program GPUs, one can write generic code that can run on both CPUs and GPUs, simply by changing the types of the array.

<sup>9</sup> A kernel is a program/algorithm that operates on a GPU.

Before we benchmark this function, it is important to discuss how operations occur on a GPU. Firstly, most operations that occur on a GPU are performed concurrently (or asynchronously) to operations on the CPU. The CPU simply launches a kernel (a program) to run on the GPU and continues with operations. One must perform a blocking operation to wait until a kernel has finished computing to properly benchmark the execution time. To start with, let's just benchmark the function how we always would:

```

julia> @btime array_test!($c_gpu, $a_gpu, $b_gpu);
4.903 μs (41 allocations: 2.19 KiB)

```

However, if we wrap our code in a function which forces synchronisation, then we can measure the true cost of using the GPU:

```

julia> @btime CUDA.@sync array_test!($c_gpu, $a_gpu, $b_gpu);
21.702 μs (41 allocations: 2.19 KiB)

```

We can see that the time taken is now much longer. This is a more accurate benchmark for this function. One should always make sure that when benchmarking, one is not just measuring the time taken to launch the kernel.

From the benchmarks, one can see that the GPU was able to perform this task around 12 times faster than the CPU (single thread). Figure 10.5 shows the comparison between the single core CPU speed and the GPU performance, varying with system size.

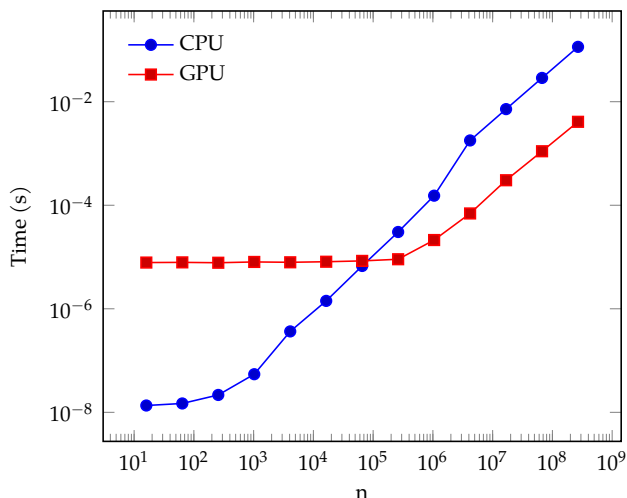


Figure 10.5. Comparing FP32 vector addition on a CPU against a GPU. The GPU used was the *NVIDIA 1080ti*.

From Figure 10.5, we can also deduce that the time taken to launch a kernel is between a hundredth of a millisecond and a tenth of a millisecond. It is not worth using a GPU to perform operations on this scale unless you have a significant amount of work for it to do. We can see that we need to use vectors of a size of around  $10^6$  to see a significant difference for the addition.

The performance increase may not seem like much on this graph, but it is important to remember the log-log scale. We are often seeing a 10 to 20 factor of performance increase on the GPU. This sort of performance difference throughout an application can mean the difference between the code taking multiple weeks to complete, to being able to be executed in a single day.

Hopefully this example demonstrates that Julia makes it relatively straightforward to write code for the GPU, provided that we can translate our algorithms to use broadcasting operations. This has the added benefit of being able to switch the types of our data and run code on the different devices. The *CUDA.jl* library can even compile native Julia functions into a CUDA kernel.

Why was it important to use broadcasting? This is because accessing elements of a CUDA array is very slow as indexing operations are calculated on the CPU, requiring synchronisation between the CPU and the GPU. This causes a huge performance hit. Scalar indexing is often disallowed by default because it is almost

always not what you want to be doing and causes huge performance hits. One may see this error when switching your data types into CUDA arrays:

```
julia> a_gpu[1]
Error: Scalar indexing is disallowed.
Invocation of getindex resulted in scalar indexing of a GPU array.
This is typically caused by calling an iterating implementation of a method.
Such implementations *do not* execute on the GPU, but very slowly on the CPU,
and therefore are only permitted from the REPL for prototyping purposes.
If you did intend to index this array, annotate the caller with @allowscalar.
```

This is the most common reason why functions from other libraries are not compatible with CUDA. For example, using the `eig` function from *LinearAlgebra.jl* to find eigenvalues of a matrix does not work, as the underlying implementations use scalar indexing. This incompatibility is the largest downside of using *CUDA.jl* and GPUs in general. Often, one will have to find the correct library or function to use, which is compatible with CUDA. Thankfully, the *CUDA.jl* library includes the entire set of C functions which can be used with your arrays, which means there is no reason to move to C to find any functionality missing in Julia.

If an algorithm explicitly requires scalar indexing, one can write one's own CUDA kernel, this will be covered in the next section.

## 10.4 CUDA Kernels

When writing code for the GPU, the best practice is to first try and use array programming, which can be executed on either the CPU or the GPU. This code is easy to test, and the tests can be run on any machine, without the GPU. If the code works correctly on the CPU, it is highly likely to also work on the GPU - as long as there are no errors. However, there will always be edge cases when your algorithm cannot be implemented in terms of array and broadcast operations, or it is highly inefficient to do so. In these cases, you may have to write the GPU kernel itself.

A GPU kernel is a self-contained program that can be run independently across many cores of a GPU. Traditionally, GPUs could compile shaders, which were able to calculate the colour of a single pixel on the screen. This shader would be run for every pixel to colour a final image. Kernels are just like these old shaders, but perform generic computation, instead of a graphics processing routine. Instead

of iterating over each unit of work, we simply describe how to do a single piece of the work, given some identification of what work one should be doing.

Let's take the example of writing a basic shader which performs vector addition. Luckily, even though CUDA is designed to be a C-like language, we are able to write all of our CUDA code in native Julia (with the help of *CUDA.jl* for compilation). As a starting point, let's write a basic `for` loop that would be executed on the GPU:

```
function basic_vec_add!(c, a, b)
    for i in eachindex(c, a, b)
        @inbounds c[i] = a[i] + b[i]
    end
    return nothing
end
```

This code will add the vectors `a` and `b` together and store the result in the vector `c`. Our CUDA kernel should make each core perform one of these inner loop additions. We are able to specify which core should do the work through the use of some special variables which are provided to each core when the kernel is scheduled for execution. Take the following example:

```
function basic_vec_add_cuda!(c, a, b)
    i = threadIdx().x
    if i <= length(c)
        @inbounds c[i] = a[i] + b[i]
    end
    return nothing
end
```

Here, we have used the `threadIdx` function which gives us a named tuple with `x`, `y` and `z` components. This is an identifier to the core as to which thread of work it is assigned. CUDA uses a thread based model for execution; each thread executes the code in the kernel independently of one another.

We can compile and launch this kernel on the GPU with the following syntax:

```
julia> a = CUDA.rand(128); b = CUDA.rand(128); c = similar(a);
julia> @cuda threads=length(c) basic_vec_add_cuda!(c, a, b);
julia> isapprox(c, a.+b)
true
```



We can see that our kernel worked as expected, despite the use of scalar indexing. The `@cuda` macro compiles and launches the function specified. We additionally have to specify the number of threads used as the length of the vector input. The CUDA programming model uses a **geometric partitioning** system. The system can be explained as:

- Each GPU has a 3-dimensional **grid** of **blocks**.
- Each **block** contains a 3-dimensional group of **threads**.
- Each thread is executed independently.

In our case, we restricted the execution to a single dimension  $x$ , but this just makes the number of elements in the  $y$  and  $z$  dimensions equal to 1. One important restriction in the CUDA programming model is making sure that a block has a **maximum** number of possible threads. This limit is usually 1024 threads. As we are often dealing with larger arrays, we need to make use of multiple blocks to allow execution across more threads than this limit.

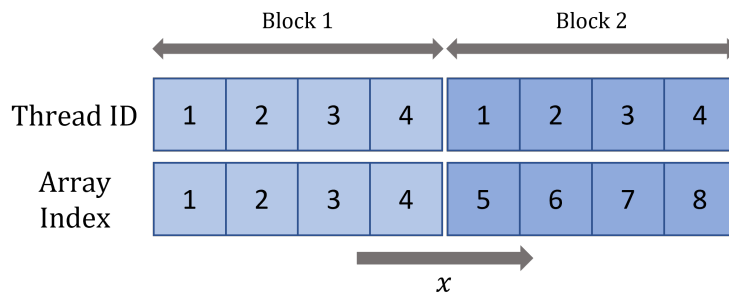


Figure 10.6. A kernel launched with a grid dimension of  $(2, 1, 1)$  and a block dimension of  $(4, 1, 1)$ . This will directly map onto the 8 elements of the input array.

Looking at Figure 10.6, we can visualise how the indexing scheme works in a single dimension. Each block has 4 assigned threads whose indices begin at 1 and end at 4. Each block also has an index which goes from 1 to 2. This is enough to map to each element in the example 8 element array. We can adjust our kernel to take into account block sizes:

```
function basic_vec_add_cuda!(c, a, b)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i <= length(c)
        @inbounds c[i] = a[i] + b[i]
    end
end
```

```

    return nothing
end

```

Now, we write a wrapper around the add function which works on CUDA arrays, which will automatically calculate the number of blocks and threads needed.

```

function basic_vec_add!(c::CuArray, a::CuArray, b::CuArray)
    n = length(c)
    @assert n == length(a) == length(b)

    num_threads = min(1024, n)
    num_blocks = cld(n, 1024)
    @cuda blocks=num_blocks threads=num_threads basic_vec_add_cuda!(c, a, b)

    return nothing
end

```

Notice that for an input of length 1029, a total of 2048 threads will be launched. It is now up to you as the programmer to decide how many threads to launch.

A common pattern seen throughout custom kernel launches is to grab the number of threads from a configuration provided by CUDA for a compiled kernel. This is then used to assign blocks.

```

function basic_vec_add!(c::CuArray, a::CuArray, b::CuArray)
    n = length(c)
    if n == 0
        return nothing
    end

    @assert n == length(a) == length(b)

    kernel = @cuda name="basic_vec_add" launch=false basic_vec_add_cuda!(c, a, b)
    config = launch_configuration(kernel.fun)
    threads = min(n, config.threads)
    blocks = cld(n, threads)
    kernel(c, a, b; threads=threads, blocks=blocks)

    return nothing
end

```

One should note that CUDA also has maximum sizes for blocks and grids. However, when getting to the grid level, one is very unlikely to have an array

large enough to cause an issue. For example, the maximum dimensions on this graphics card are:

```
julia> CUDA.max_block_size
(x = 1024, y = 1024, z = 64)
julia> CUDA.max_grid_size
(x = 2147483647, y = 65535, z = 65535)
```

The maximum size of a block is the product of the dimensions:

```
julia> reduce(*, CUDA.max_block_size)
67108864
```

As long as the dimensions you choose to multiply to a number less than the above, you do not need to use the grid.

Now there are some things that you may have noticed that seemed to reduce performance. For example, we are including an `if` statement in our kernel. This is necessary as the length of the array may not be easily split up into blocks. We are manually performing our bound checking here to ensure that we can safely read and write from memory. This is true in the case of an array of length 1029, which would have 1023 threads running which would index outside the array. To ensure this does not happen, we must guard with the manual bounds checking.

Additionally, CUDA imposes on us the restriction of returning `nothing` from the kernels. All memory allocations must be done **outside** the kernel. This usually means implementing your kernel as an in-place operation on pre-allocated memory. If you want to provide a better API for users and abstract away the allocations, one can wrap the call to the kernel:

```
function basic_vec_add(a::CuArray, b::CuArray)
    c = similar(a)
    basic_vec_add!(c, a, b)
    return c
end
```

Additionally, if the memory is only needed temporarily, you can create it outside the function and release it later. If you are writing the critical loop section of your code, it is often better to write the main algorithm with pre-allocated caches and then create a wrapper (like the one above) to optionally use if you do not want the users of the code to manually manage their cache. This gives the option for re-using a cache in a hot-loop.

## 10.5 *CUDA Libraries*

As writing efficient GPU kernels is extremely difficult, CUDA provides many libraries for solving common problems. The CUBLAS library has many routines for highly optimised BLAS<sup>10</sup> which run on the GPU. Here is an overview of libraries which CUDA provides:

<sup>10</sup> Basic Linear Algebra  
Subprograms- BLAS

- **CUBLAS**: Optimised BLAS routines, which involves matrix multiplications and vector calculations.
- **CURAND**: Library for random number generation on the GPU. This is useful for Monte-Carlo simulations.
- **CUFFT**: Library for calculating Fast Fourier Transforms.
- **CUSOLVER**: Library for solving linear algebra problems, such as diagonalising a matrix (eigenvalue and eigenvector decomposition).
- **CUSPARSE**: Library with support for using sparse matrices.
- **CUDNN**: Optimised routines for deep neural networks.
- **CUTENSOR**: Accelerated tensor linear algebra library providing tensor contraction, reduction and element-wise operation.

Many of the BLAS routines already have a higher level interface, for example, the matrix multiplication. One can multiply two matrices very easily using:

```
julia> n = 2;
julia> a = CUDA.rand(Float32, n, n);
julia> b = CUDA.rand(Float32, n, n);
julia> CUDA.@sync c = a*b
2×2 CUDA.CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 0.298787  0.213741
 0.109655  0.313506
```

Or the in-place version:

```
julia> using LinearAlgebra
julia> mul!(c, a, b)
2×2 CUDA.CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 0.298787  0.213741
 0.109655  0.313506
```

Since each CUDA array has its own type - the `CuArray` - Julia can dispatch to the most optimised routines. In this case, `CUDA.jl` creates a dispatch for the `mul!` function in `LinearAlgebra.jl` which calls the underlying `CUDA.CUBLAS.gemm!` which stands for “Generic Matrix Multiply”<sup>11</sup>.

Naming conventions in CUDA are incredibly old-fashioned, and it is not very clear what each function does. One will have to check the documentation<sup>12</sup> to see which function you need for your code. Even then, it is often difficult to decipher exact what algorithms are available. Creating high quality, well-documented, high-level front-end APIs for `CUDA.jl` is an ongoing process. Fortunately, many packages provide GPU acceleration, using these underlying libraries. You will find many GPU implementations in packages like `NNLibCUDA.jl`, which is a dependency of `Flux.jl` - one of the largest machine learning libraries in Julia.

Many of the high-level BLAS operations already extend the `LinearAlgebra.jl` functions to work with types from `CUDA.jl`. Try just changing the input types to your existing code. If you find errors, track down the functions which are not implemented and find a suitable GPU compatible replacement. This may mean having to write your own CUDA kernel.

<sup>11</sup> One can track down the actual implementation by using the `@which` and `@edit` macros, which allows one to find the source code for the method. This can also be done with the use of debugging, but this is very slow in Julia.

<sup>12</sup> NVIDIA, *CUDA Toolkit Documentation*, <https://docs.nvidia.com/cuda/index.html>, Accessed on 6th July 2022, 2022.

## 10.6 Benchmarking & Profiling

Benchmarking GPU code can be a bit tricky. When you are launching a CUDA kernel, this will simply *send* the program to the GPU, and not wait for it to finish executing. Let’s take the following example:

```

julia> a = CUDA.rand(256,256); b = similar(a);
julia> @benchmark begin $b .= $a .* $a end
BenchmarkTools.Trial: 10000 samples with 8 evaluations.
Range (min ... max):  3.675 μs ... 1.339 ms | GC (min ... max): 0.00% ... 99.02%
Time (median):       4.130 μs | GC (median): 0.00%
Time (mean ± σ):     4.278 μs ± 13.346 μs | GC (mean ± σ): 3.10% ± 0.99%

Histogram: frequency by time
3.67 μs |-----| 4.81 μs <
Memory estimate: 1.84 KiB, allocs estimate: 25.

```

This only measures the time taken to launch the kernel, not the time taken to execute the kernel. We can re-run this using the `CUDA.@sync` macro:

```

julia> @benchmark CUDA.@sync begin $b .= $a .* $a end
BenchmarkTools.Trial: 10000 samples with 3 evaluations.
Range (min ... max): 9.091 μs ... 23.074 μs      GC (min ... max): 0.00% ... 0.00%
Time (median):      9.795 μs                    GC (median):      0.00%
Time (mean ± σ):    9.833 μs ± 498.364 ns       GC (mean ± σ):    0.00% ± 0.00%

  9.09 μs      Histogram: frequency by time      11.4 μs <
Memory estimate: 1.84 KiB, allocs estimate: 25.

```

Which shows us that the kernel actually took a lot longer to run. *CUDA.jl* also provides the `CUDA.@elapsed` and `CUDA.@time` functions which can be very useful for measuring the performance of your code.

Remember that when you are benchmarking the GPU code, you should be sure to call `CUDA.@sync` to ensure that you are measuring the total time of a CUDA kernel.

### 10.6.1 Application Profiling

When you have a larger application, with multiple calls to custom CUDA kernels, it is sometimes better to profile an entire application. We can use the NVIDIA Nsight system to profile an application. The *CUDA.jl* documentation<sup>13</sup> suggests installing Nsight Systems directly from the NVIDIA website<sup>14</sup>.

<sup>13</sup> <https://cuda.juliagpu.org/table/development/profiling/>

<sup>14</sup> <https://developer.nvidia.com/nsight-systems/get-started>

Ensure that your Nsight Systems installation is working correctly by typing the following into the terminal:

```
nsys --version
```

On Windows, you will need to run the terminal as an Administrator and also ensure that the folder with `nsys.exe` is in your system PATH environment variable.

Nsight Systems can be used to view the bottlenecks in your system and diagnose whether kernels are being too frequently and hurting performance.

## 10.7 Tips

In order to fully utilise the additional resources provided by using a GPU, we must keep in mind some causes of poor performance.

### 10.7.1 Copying Data

Remember that a GPU has its own local memory, and any data processed by the GPU, must first be transferred onto the device memory. To make the terms clear, we refer to the main memory that the CPU has access to as the **host** memory. Often, you will see the CPU (and the rest of the machine) referred to as the **host**. The GPU only has access to its own memory, called **device** memory. It can take a long time for the host to copy memory to the GPU, and vice versa. This is why it is strongly recommended that all possible processing is done on the GPU until it is needed back on the CPU.

If you have a function that performs much better on the GPU, you must consider whether it is worth transferring the data to the GPU, processing it there, and then transferring it back to the CPU. Sometimes this will be beneficial, as the speed-up is so significant that the copy times are usually worth the cost. However, one can consider streamlining the processing of the data, so that it all occurs on the GPU. Even if one part of the process is actually faster on the CPU, it may be worth implementing a GPU version in order to keep the memory on the device and avoid copying.

### 10.7.2 Launching Kernels

An individual kernel launch can have quite a high latency, especially when compared with launching a CPU thread. While the CPU is constructing the kernel and sending it to the GPU for scheduling, the GPU may be quite idle. Launching many smaller kernels can incur a huge overhead cost, and cause the GPU to be idle for a large amount of time. If profiling with the NVIDIA tools, this is the performance bug to look out for.

To remedy this, one should attempt to fuse as many kernel calls together, and group the work into larger items. For example, if one is using array programming, make full use of fused broadcasting:

```

y .= sin.(x)
y .*= x
y .+= x .^ 3
y .= exp.(y)
# use a single statement instead
y .= exp.(x.*sin.(x) .+ x .^ 3)

```

While it may look better in the source code to split up the operations into multiple lines, it is often better to put these calls on a single line. The first section calls 4 kernels, whereas the single line only calls one kernel. If the call is extremely long, write a function for a single element and broadcast it across the entire array:

```

_f(x) = exp(x*sin(x) + x * x * x)
y .= _f.(x)

```

We will give an example of this optimisation in Section 10.8, by loading our kernel with work.

### 10.8 Case Study: Monte-Carlo Simulations

Here, we will provide an example of porting a Monte-Carlo random walk simulation to the GPU, such as the one shown in Figure 10.7. As with the earlier advice in our book, we will write a function that performs a single step in the random walk, shown in Algorithm 10.1.

```

function mc_random_walk_step(x, sigma)
    return x + randn(typeof(x)) * sigma
end

```

Now imagine that we want to study some population level statistics, by running many of these walkers in parallel. Despite us choosing a very simple example, Monte-Carlo simulations are an extremely useful tool for computational science. Our aim for this exercise is to perform some number of steps of this Monte-Carlo update for many independent walkers and obtain an array with their final positions in.

We can implement a non-allocating array version of our desired algorithm:

We can run our algorithm on the CPU easily:

```

julia> n=2048; x = zeros(Float32, n); y = similar(x);
julia> sigma = 1.0f0; steps=100;
julia> mc_random_walk!(y, x, sigma, steps);

```

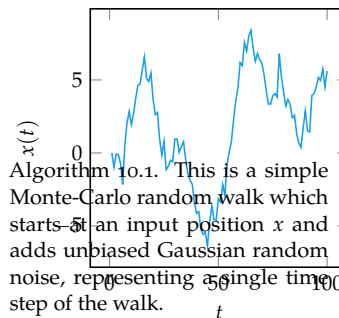


Figure 10.7. Shows an example of a continuous space, discrete time random walk, starting at the origin.



```

function mc_random_walk!(y, x, sigma, steps)
    # Copy the initial values from x into y
    y .= x
    for t in 1:steps
        y .= mc_random_walk_step.(x, sigma)
    end
    return nothing
end
end

```

Algorithm 10.2. This is one way of writing our random walk for an entire array of walkers, performing the array operations in step.

We can extend this to run on the GPU just by changing the types:

```

julia> x_gpu = cu(x); y_gpu = similar(x_gpu);
julia> mc_random_walk!(y_gpu, x_gpu, sigma, steps);


```

Let's benchmark the CPU version:

```

julia> @benchmark mc_random_walk!($y, $x, $sigma, $steps)
BenchmarkTools.Trial: 6429 samples with 1 evaluation.
Range (min ... max): 759.138 μs ... 1.039 ms      GC (min ... max): 0.00% ... 0.00%
Time (median):       775.499 μs                  GC (median):      0.00%
Time (mean ± σ):     776.457 μs ± 6.970 μs       GC (mean ± σ):   0.00% ± 0.00%

```



759 μs                      Histogram: frequency by time                      793 μs <


Memory estimate: 0 bytes, allocs estimate: 0.

And the GPU version:

```

julia> @benchmark CUDA.@sync mc_random_walk!($y_gpu, $x_gpu, $sigma, $steps)
BenchmarkTools.Trial: 4602 samples with 1 evaluation.
Range (min ... max): 919.045 μs ... 13.540 ms     GC (min ... max): 0.00% ... 88.97%
Time (median):       1.063 ms                    GC (median):      0.00%
Time (mean ± σ):     1.083 ms ± 363.878 μs       GC (mean ± σ):   0.97% ± 2.67%

```



919 μs                      Histogram: frequency by time                      1.21 ms <

Memory estimate: 142.00 KiB, allocs estimate: 2525.

We see that our GPU version is actually faster (for this number of walkers), however, there is actually a bit of a performance bug under hood that should be addressed. Upon each of our `for` loops, we are calling a new kernel. Instead, we should try to fuse these kernels together:

Now we can try and benchmark this again:

```

function mc_random_walk(x, sigma, steps)
    for t in 1:steps
        x = mc_random_walk_step(x, sigma)
    end
    return x
end
function mc_random_walk_fused!(y, x, sigma, steps)
    y .= mc_random_walk.(x, sigma, steps)
    return nothing
end

```

Algorithm 10.3. Writing out the loop for an input argument and increasing the work done on each cycle of the loop.

```

julia> @benchmark CUDA.@sync mc_random_walk_fused!($y_gpu, $x_gpu, $sigma, $steps)

```

BenchmarkTools.Trial: 8412 samples with 1 evaluation.

Range (min ... max):	487.146 μs ... 922.482 μs	GC (min ... max):	0.00% ... 0.00%
Time (median):	580.525 μs	GC (median):	0.00%
Time (mean ± σ):	588.867 μs ± 30.854 μs	GC (mean ± σ):	0.00% ± 0.00%

487 μs                      Histogram: frequency by time                      702 μs <  
Memory estimate: 4.47 KiB, allocs estimate: 77.

We have fused all the kernel calls together. This minimises the amount of overhead for scheduling, and allows the GPU cores to stay busy for the duration of the computation. This small change allowed us to dramatically improve the performance of our GPU code.

If the kernel calls are very large (e.g. a large matrix multiply), the overhead in calling multiple kernels is only a very small portion of the total time.

### 10.8.1 Custom Kernel

It is a good exercise to write a custom kernel and compare the execution to the broadcasted notation. Our kernel will be straightforward:

We can now benchmark on the same data:

```

julia> @benchmark CUDA.@sync mc_random_walk_gpu!($y_gpu, $x_gpu, $sigma, $steps)

```

BenchmarkTools.Trial: 7613 samples with 1 evaluation.

Range (min ... max):	585.835 μs ... 879.780 μs	GC (min ... max):	0.00% ... 0.00%
Time (median):	642.655 μs	GC (median):	0.00%
Time (mean ± σ):	651.092 μs ± 30.676 μs	GC (mean ± σ):	0.00% ± 0.00%

Memory estimate: 4.47 KiB, allocs estimate: 77.

```

function _mc_random_walk_cuda!(y, x, sigma, steps)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i > length(y)
        return nothing
    end
    @inbounds pos = x[i]
    for t in 1:steps
        pos += randn(typeof(pos))
    end
    @inbounds y[i] = pos
    return nothing
end
function mc_random_walk_gpu!(y, x, sigma, steps)
    n = length(y)
    @assert n == length(x)

    kernel = @cuda name="mc_walk" launch=false _mc_random_walk_cuda!(y, x, sigma, steps)
    config = launch_configuration(kernel.fun)
    threads = min(n, config.threads)
    blocks = cld(n, threads)
    kernel(y, x, sigma, steps; threads=threads, blocks=blocks)
    nothing
end
end

```

Algorithm 10.4. A custom written kernel for running a random walk simulation within a CUDA kernel, together with a wrapper which decides on how many blocks and threads to use.

```
586 μs      Histogram: frequency by time      767 μs <
Memory estimate: 4.11 KiB, allocs estimate: 76.
```

We can see that our custom kernel did not perform as well as our simpler approach. It is clear that writing a custom kernel is not necessary to achieve performance gains, as we can rely on the compiler to generate fast code for the GPU, using the array notation. It is entirely possible to rewrite our kernel to be of a similar performance to our previous implementation.

## 10.9 Exercises

**Exercise 10.1.** Implement your own custom random walk CUDA kernel which performs similarly to the original fused implementation in Algorithm 10.3. See how close you can get to the similar performance. (**Hints**): Try to use different random number generation (from CURAND).

**Exercise 10.2.** Take some of your existing code and port it to the GPU using the inbuilt functions, and try to write a custom kernel for the more complicated operations.

**PART III:**

**PROFESSIONAL SCIENTIFIC CODE**



## 11 *Version Control*

Version control is an essential part of modern software development. It is an indispensable skill for a modern software engineer, and a skill that is essential for any modern software development job. This skill is less common in academia, but researchers are slowly coming to terms with these new systems, and the huge benefit that comes with them. Many researchers will not have heard of the term “version control”, but this is because it is the more general term given to techniques and skills for versioning your software. Often times it is also called “source control”. Like how “Google” has become synonymous with search engines, “Git” is the equivalent for version control. “Git” will likely ring more bells for many researchers, but few use it. The reason that Git is so synonymous with version or source control, it has a huge, dominant, market share. From now on, I will use Git as the example when I talk about source control, however, one should know that there are other alternatives that do similar things, but they are often servicing niche corners of the market<sup>1</sup>.

Before we go any further, we should answer what version control actually is:

Version control is a system or set of systems responsible for managing changes to computer programs or documents, or other collections of information.

Many researchers are familiar with version control, when they write academic papers. It is not uncommon, to have many versions of the same file labelled “*mainv1.tex*”, “*mainv2.tex*”, “*mainv2final.tex*” or “*mainv2final2.tex*”. However, this problem becomes very unwieldy, very quickly. This is hard enough for a single person to manage, it becomes a living nightmare when one has to share one’s work with others. Papers are often emailed around with version numbers, only allowing a single person to write at a time. This is still common and widespread to this day, despite the solution having been around for multiple decades. As

<sup>1</sup> As an example, Subversion is often used in games design, as it has much better support for keep track of large assets such as high quality textures or complex 3D models.

of writing this in 2022, there is not a single reason for keeping this old system of working, as there are far more professional ways to work. In the past, using systems such as Git, have required a large learning curve, however, now it is easier than ever for people to work with, to help them be more productive.

### 11.1 *What is Git?*

If labelling your files manually with a version has so many problems, and is not a scalable solution when working with other people, what is the alternative? Here, we will discuss the way that Git handles these issues, as it is the most common software tool, and the industry standard.

#### 11.1.1 *Repositories*

We will first introduce the idea of a repository. A repository is just a fancy name for a folder. That's it. Nothing special to a repository, other than it is a defined folder, for which you want to keep track of changes to the software. What is special about a git repository, is that there is a special, hidden, folder inside your repository, which keeps track of versions for you behind the scenes. One never actually has to interact with this folder, but it is where the changes and history of the files in the folder are stored. The way that one interacts with this tool is usually through the Git software itself, which commonly takes the form of a command line interface. From the command line, you can tell Git to take a snapshot of your code, which will look at your files and see what has changed since the last snapshot, and keep track of these. Git is smart, in the sense that it will only store the differences between your files, instead of another copy of the entire file. Not only is this efficient, but it will have more benefits that we will talk about later.

#### 11.1.2 *Commits*

A snapshot of your software is called a *commit*. This is just the current state of your repository and all the files within. Each file does not exist separately to every other file, and one tracks the version of the entire repository, not just the version of one single file. The reason for this is that many files will often interact with one another, meaning that they are tightly coupled. One cannot have an accurate version of one file, without also knowing the version of the files that it interacts



with. Think of changing a variable name in one file, it should also be changed in another, or else the software will not run. This is why we have the idea of a repository, instead of tracking just individual files.

A commit is just a single snapshot of your code. Your repository begins in an empty state, and usually you make an initial commit with the first files added. Since Git just works with files in a folder, one can turn an existing project into a Git repository, either by initialising a repository in the desired folder, or copying existing files into a new repository. Git will keep track of your entire history of commits. This is the “log” of what has happened. A commit is not just a snapshot of the current state of your code. It also does not happen automatically every day, like a backup system would. A commit is a purposeful act by the developer. As such, one also accompanies a commit with a small message, describing what changes were made in the current commit. These small messages help navigate the history, if you ever need to look back at old versions.

As an example, imagine that you have written some code to provide some functionality. After a while, you decide that you don’t need it and delete the code. You commit this change with a message such as “*removed some\_func() from the code base*”. Later, even weeks down the line, you realise that that old function would have really helped now. If you were not using Git, you would be out of luck, and would have had to write it again from scratch. Worse, is that you would have feared losing any old code, so you keep it around, cluttering your folders and files. However, in Git, removing code is never a concern, as Git tracks the changes, including the deletion itself. All one has to do, is navigate back to the commit where the function was removed and one now has access to the old code. Nothing has been lost. You can keep your files and folders clean and free of different versions of your code, without fear of losing any information down the line as it is all kept in the history.

### 11.1.3 Branches

So far, we have talked about a repository and commits, which are only two parts of what Git provides. The third ingredient which is very important to understand is the concept of branches. A branch is just a collection of commits arranged in a timeline. When we were talking about the history of commits before, we would implicitly be calling that history the “*main*” branch, as it contains the main version of our software, and the history. The main branch was also historically called

the “*master*” branch as it served as the “*master*” copy of your code. The terms “*master*” and “*main*” branch are interchangeable, but modern projects tend to use “*main*”. This branch is the most important one to know about, since one can reap the majority of the benefits from Git, just by utilising the “*main*” branch in the previously described manner.

Imagine being able to track every single change made to a project, right back to the first line of code. This gives you a huge amount of safety when it comes to versioning, as it becomes very hard to lose progress. However, if one does not commit one’s code regularly, the tool becomes pretty useless. Commits should be small and frequent. A common meme amongst programmers is to commit changes across tens of files with thousands of lines of code changed, with the simple message “*minor changes*”. This is very bad practice, as it removes many of the benefits of using Git in your projects. This also does not scale well to larger software projects where one has to work with many other people. Even if you are just working by yourself, it is often good to practice the skill of committing a smaller number of changes, more frequently, with descriptive commit messages.

Git was designed from the ground up to be useless for many people working in a team, on the same piece of software at the same time, often in completely different locations. It is mostly designed without thinking about a shared folder/repository, so how does it manage keeping track of software versions between multiple people? In the past, the way of dealing with this is to freeze the code of a particular file or even the entire project while one person makes changes. Only when that person has finished, can another person work on it. This is exactly how academics write papers, by freezing the version of the paper and sending via email, only continuing to work on it once a colleague has completed their changes and sent back an updated version. Git takes a much smarter approach to this.

Before we abstract away to the idea of multiple people working on the same project, let us look at a similar problem with just a single developer. Imagine you are working on a feature that will take quite a long time to produce. For example, you are optimising a core part of the software and the software currently does not run. What if you needed to quickly use the previous, slower, version of the software to get a figure or a graph for a presentation? Do you completely undo the changes you have been making and revert back to the previous state while you use that version of the software, losing the progress you have made? The answer is no. In Git, one uses another branch when developing a feature that potentially can be breaking until the feature is complete. This leaves the main branch to be in

a useable state at any time. Once a feature is complete, one can merge the changes on the separate branch back into main, upgrading the functionality. This ensures that main is in a useable state at every point in time, and at no point has a feature that is “in development”, and breaks the code.

Before we said that a branch is just a collection of commits. In particular, it is useful to logically group commits together based on their purpose. Commonly, people use the idea of a branch to be a collection of commits that provides a certain feature, or fixes a specific bug. The scope of a branch should be somewhat limited to the feature, only containing commits and changes pertaining to that purpose. One usually names a branch according to this purpose, such as “*add-unit-tests-for-function-x*”, which specifies the scope of the changes to be made in that branch. The reason that it is called a branch is that one can view the main branch as the trunk of a tree. This is the perfect<sup>2</sup>, master copy of your code. One can then branch off the trunk (branching off main), which starts a new, parallel timeline to main, containing the entire history of main at that point. Creating a new branch starts at the version of the current state of main. Any commits made from this point will be on your new branch, and not exist in main.

<sup>2</sup> or as close as it can get

Branching off, also provides the same stability as a code freeze, as the developer is now unaffected by any changes in main. This means one can focus on developing a certain feature, without worrying about the other changes in the program. When the feature is finished, one can merge these changes back into main.

#### 11.1.4 Merges

One reason why academics usually work on a paper at the same time, is that the process of combining changes from two people into a single file is very labour intensive and error-prone. For anyone that has done this, they know the pain of having to read over multiple sections and copy each individual small change, even if it is just a spelling change.

Git takes a much smarter approach. As the developers have been committing their code over time, only tracking the changes between files, Git can intelligently know if two versions of the same line of code in the same file have been independently changed, and flag this as a “merge conflict”. Git will not automatically resolve these conflicts as there is no way of knowing which change was correct. Often times, one has to manually resolve these conflicts by taking parts from both. However, if two developers have added code in different parts of the same

file, but not overlapped at all, then Git will know to just put both sets of changes together. If one person wrote the introduction, and another the conclusion, then it can just be merged automatically.

When we want to merge changes into main, if there have been no other changes in main, since the branch was started, then there will be no merge conflicts, and all the new changes will be put into main. However, if we were to switch branches to main and create a new branch and merge that in before, then Git will automatically check for conflicts. It is the duty of the developer to handle any merge conflicts before a new merge commit is made. The majority of the time, especially when working in small teams, merge conflicts are very rare. On the occasions when they do occur, there are tools to make managing them painless, which will be covered later in this chapter.

The typical workflow of a developer, is to decide on a feature to add to the code, or a collection of changes to make. They then create a branch from main, giving it an appropriate name for the task at hand. They then begin to develop this feature. If the developer has to work on a different feature, they can switch back to main and create a different branch to work on that feature, being completely isolated from any of the changes previously made in the first branch. If this branch is completed first, then they can merge back into main. This is commonly called a “pull request”, as one is asking to pull the new changes into the main branch. This is often given a special name as in larger software projects, there are normally steps in place to ensure the quality of the code being added to the main branch. This usually involves the review of other members of the team, and the code needing to pass all of the unit tests<sup>3</sup>. When the developer switches back to their previous branch, they will be unaffected by these changes. If the developer needs these new changes to continue working, they can merge the new changes from the main branch into their current branch. This is often called “updating from main”. This gives the developer complete control of which parts of the code to include and work on at the same time. It gives them the complete flexibility and freedom to experiment with the code, not being afraid of breaking anything, as they can always revert back to a previous time, or work on a different branch if the experimentation interferes with other tasks.

<sup>3</sup> This is discussed in more detail in a later chapter (needs ref)

## 11.2 Working with Teams

In the previous section, we described a workflow which can be used by a single developer, who can now work on multiple features independently of one another, without fear of one piece of work breaking another. However, this functionality can be immediately scaled up to an entire team of people working on the same repository. Let us now imagine that a team of people are working on the same code base, and see what the workflow is like.

For starters, every contributor to a project has their own local copy of the repository. Remember, this is just a folder on your local machine, with a special hidden folder to keep track of the history of the repository. Each person can commit to their own repository, independently of one another. This means that changes can be made by people at the same time? How do these people share their changes with one another? This is where the idea of a centralised or shared repository comes into play. In Git, this is often called the “origin”. Like the “main” branch is the master copy of the software, the “origin” is the master copy of the repository itself. It acts as a centralised node, being able to communicate with all collaborators in the project. In the simplest case, we have a single main branch in the origin repository. This is often stored on a different computer in the cloud<sup>4</sup>. Instead of everyone working directly on this origin copy, they instead make a local copy on their local machine. Again, this is just a folder on your computer. In Git, we usually call this copy a “clone”, and you will here the phrase “clone this repository”, which just means make a copy of it on your local machine. The major difference between your local copy and the origin, is that they are separated and isolated from one another, except for the local repository pointing to the origin as the “master” version. The developer has to manually sync changes between the origin and the local machine. Changes are brought to your local machine by merging the origins copy of a branch into your local version of that branch.

At the very least, the local machine and the origin both have a copy of the main branch. As stated previous, no one should commit directly to main, especially if working with other people. For this reason, developers will work in their own branch. However, one can create a branch on one’s local machine, and sync this with the origin, to create the same branch there. When the developers commit their code to their local branch, they can choose to also “push” the changes to the origin, which copies the new commits from your local branch to the origin. If people are working on the same branch at the same time, sometimes, one

<sup>4</sup> Remember, the cloud is just a fancy abstraction for “someone else’s computer on the internet”.

will need to also pull changes down from the master copy into the local branch. Remember, a “push” is sending changes to the origin and a “pull” takes changes from the origin and applies them locally. We can simplify this process by just calling it a synchronisation, which performs both a push and pull (usually the pull comes first to see if there are any changes). Remember that the origin can have a different version to your local branch, so pulling in any changes is actually a merge, and may result in merge conflicts. However, if one developer works on their own branches, then this becomes unlikely. The main distinction is that merges must happen locally on a developers machine. Once a merge is completed and there are no merge conflicts, only then can the changes be pushed up the origin.

When one creates a pull request into main, one does not locally merge into master. Instead, one must first pull from origin’s main into the local feature branch and solve all merge conflicts. Once this is done, then one can complete the pull request, and merge one’s changes into the origin’s copy of main.

Notice that in this entire description, there can be multiple different people all interacting with the origin at the same time. We have used the idea of a local machine, but this local machine can belong to anyone. Having the centralised origin, removes the need for any two developers to interact with one another, but instead they can share code via the origin. No need to email code to someone! Just add them to your repository.

This model of working is very successful, and is used on some of the largest open source projects out there, such as the Linux operating system, which has tens of thousands of contributors. This system is very robust if used correctly and formalises not only the version control problem, but also the code sharing problem.

### 11.3 *Keeping a Clean Repository*

Before starting on the basics, one should know that not every file should be tracked and saved by Git. Especially during programming, there are many files which are generated from your code whose version you need not care about, since they can be generated from the source files themselves. Instead of having to wade through these useless files, one instead uses a *.gitignore* file. This file is simply a special Git file which one can list certain specific files or general extensions or folders which Git will ignore. This means that changes are not tracked.

The types of files you ignore depend on the language used and there are often template *.gitignore* files available online for most languages. If you generate your repository using GitHub or GitHub Desktop, you will be given a dropdown list of templates. It is highly recommended you use this.

If there are any files in your source code that contain sensitive information, such as an API key, email address, password etc, you should not commit this file. Remember that Git tracks your changes and so if you commit this information by accident, even if you remove it, it will still be there in the history, requiring more advanced Git skills to remove it. To save this from happening, all files like this should be listed in the *.gitignore* file.

Additionally, any generated data or plots should also likely be added to the *.gitignore* file, as it is not a good idea to commit binary data to Git, as it works best with text based files. Furthermore, GitHub has a file size limit of 100mb, which means that if you accidentally commit a large file, GitHub will not let you sync with the origin, and again, one needs advanced Git skills to remove the offending large file from your history, so that you can sync again.

## 11.4 Learning Git

One first point of protest is that Git is that one traditionally interacts with Git through a command line interface (CLI). This is not very user-friendly, especially to beginners who normally do not need to interact with a terminal<sup>5</sup> at all. However, it is a complete misnomer that you need to learn how to use the command line interface, in order to productively use Git. Many tutorials online will teach you how to use the command line, however, this is only the traditional way of interacting with Git. This method has a high learning curve as one must memorise commands to type in, along with being highly error-prone. Instead, this book will focus on teaching how to use Git via a Graphical User Interface (GUI). This simply replaces the most common commands with buttons instead, and streamlines the use of Git, and makes it a much more intuitive and productive tool, accessible to people with any amount of experience.

On every platform there are many GUI tools for Git. However, in this book we encourage the use of Visual Studio Code, which has a Git GUI built-in. One needs to install the underlying Git software on one's computer, which is described in the start of this book. On Windows or Mac, one can use GitHub Desktop<sup>6</sup>, which is a free GUI which is one of the most simple and robust Git GUIs out there. One can

<sup>5</sup> Another name for a command line. Usually these are just other programs. On Mac and Linux, the most common terminals are bash and zsh, and on Windows one uses cmd or Powershell. These are just ways of interacting with programs through a text-only interface.

<sup>6</sup> This can be supported on Linux, but it is more difficult to set up. Linux users will likely be better off using GitKraken.

easily switch or create new branches, update a branch from main, and even see the line by line changes visually. There is a way to visually select which changes you want to commit, along with your commit message. If you have made changes that you do not want to be in the history, you do not have to commit them! Instead, just disselect them from the GUI menu.

There is a huge amount of depth to git, and there are ways to do a lot of different tasks. However, I think it is important that beginners stick to a simple workflow, and only use the basics, as this will be the least error-prone and effective way of using Git.

#### 11.4.1 *Recommend Git Workflow*

If you are starting a new repository, you should select the appropriate *.gitignore* template for your language or project. The templates provided by GitHub are very reliable and are the recommended ones to use. If you choose to create this repository using GitHub, you should then clone this repository to your local machine. There are instructions for this online. If created locally, one should then publish your repository to a place like GitHub, which provides free hosting for both public and private repositories. Using a provider like GitHub for your origin provides many nice features, all for free! Most importantly, it provides a backup for all of your source code. Additionally, if you work with a team, it provides the centralised origin from which you can share code with the rest of the team. Even if you are on your own, this means that you can share code across many different computers, and you need not stick to a single development machine; giving you unparalleled flexibility in how you work.

Once a repository is created, with an origin and a local copy, one should commit any initial files into main. This can be done little by little or all in one go. This is for when you have an existing project, which is not tracked by Git. If you are starting a new repository, make sure that the initial files are there, such as the *.gitignore* and the README<sup>7</sup>.

When you are ready to start developing, use your Git GUI software to create a new branch from main, and give it a name specific to the feature you wish to develop. When you have created this, publish your local copy of the branch to the origin, so everyone knows you are working on a separate branch. Now, start writing your code! Stop every so often to commit small bundles of changes, with

<sup>7</sup> A README is a very useful file. Inside you can document how to get setup with the project, such as the software needed, and the commands needing to run the code.



a descriptive commit message. Committing only affects your local repository, so you should also push your changes to the origin.

Once you have finished developing your feature, and it is ready to be in main, use either the GitHub website, or your local GUI to create a pull request. This should be done on GitHub. This request will give you a page which shows the all of the changes you have made since the branch was created. One should always update from main before opening a pull request, so that you can locally solve any merge conflicts. Once the pull request is created, notify other members of your team to give it a review. During this review you should check over code quality, such as good variable names etc. A good reviewer will check-out (switch to) your branch, that you are trying to merge in, locally, and run the code to make sure it works. At this point you should also run any automated tests. If you are using GitHub, it is possible to setup an automatic system for GitHub to run the automated tests before merging in your code. Once everyone involved is satisfied with the code changes, someone (likely you) will approve and accept the merge into main. If there were any comments during the review, you can update changes and simply push them to the origin to update the pull request. If the review takes a long time, and the main branch has changed, one should also do another update from main before merging in.

Once development has finished, switch back to the main branch locally, and synchronise with the origin, pulling the new changes. The main branch should now have all of your changes! From this point, you repeat the process, with a new branch for each new piece of work/feature continually.

When working alone, it is tempting to just commit everything directly to main, however, I urge you to use branches instead, as it offers much more flexibility and gives you vital practice for when you eventually work with others. However, committing to main is better than not using any version control at all.

#### 11.4.2 *Advanced Topics*

#### 11.4.3 *Solving Merge Conflicts*

When you are solving merge conflicts, it is best to open them in Visual Studio Code. This will give you a side by side comparison between the incoming changes and your local changes. One can use the GUI to select which version you want to keep from the two. One can select individual lines, or entire blocks and use the

context menu (right click) to “stage” or choose those changes. The window at the bottom shows the final merged version. It is also possible to manually edit the conflicted file until there are no more conflicts. Git will add some text to the files to show a conflict, the GUI will remove this detail for you, but know that it can happen.

Merge conflicts using a GUI are much easier to deal with. All of the conflicts must be solved before finalising the merge. It is a good idea to run the software and test whether it works before continuing.

## 12 *Reproducibility*

When we are developing scientific applications, reproducibility is a core value of our code. It should be possible to run the same code and expect the same results. We should be able to send our code to others to verify our results. This chapter investigates techniques for improving reproducibility across our code base.

### 12.1 *Controlling for Randomness*

A large area of research requires the use of Monte-Carlo simulations, which are stochastic in nature. These techniques have randomness at their core. We can look to the example in Figure 8.1, which used a Monte-Carlo technique for estimating  $\pi$ . Each time we ran our code, the estimate for  $\pi$  changed, but why? To answer this question, we first need to take a look at how the random numbers were generated in the first place.

#### 12.1.1 *Random Number Generators*

There are numerous algorithms for generating random numbers. In some fields, such as computer graphics, this process is referred to as **noise**. These algorithms are commonly referred to as **Random Number Generators** (RNGs). The set of algorithms forms a core part of many fields from academic simulations to computer graphics to cryptography. These algorithms and their properties have a long history, but are still open areas of research. Let's take a look at how some of these algorithms work.

As a starting point, we should mention that many of the algorithms used today are known as **pseudorandom number generators** (PRNGs). One would normally expect a random number generator to be just that - *random*. However, usually

most such generators are in fact *pseudorandom*, which means they act like a truly random number generator, but are in fact entirely deterministic and predictable. PRNGs are often used in place of truly random generators, due to the fact that the latter is incredibly hard to reliably reproduce at the scale and speed required in modern applications. The bright side to this, is that most applications that use random numbers will suffice with only pseudo-random numbers.

A PRNG works by evolving an internal state (a collection of bits - usually a number) via the repeated application of a function. One very simple example of a PRNG function is the following:

$$X_{n+1} = (aX_n + b) \bmod m, \quad (12.1)$$

where  $a$ ,  $b$  and  $m$  are large integers.  $X_n$  represents the internal state. One can use this internal state to generate numbers of a distribution, for example, the  $n^{\text{th}}$  floating point number between 0 and 1 generated could be  $Y_n = \frac{X_n}{m}$ . We can code up this random number generator in Julia and look at the properties. In

```
mutable struct SimplePRNG{T<:Integer}
    X::T
    a::T
    b::T
    m::T
end
function Base.rand(rng::SimplePRNG)
    rng.X = (rng.a * rng.X + rng.b) % rng.m
    return rng.X / rng.m
end
```

Algorithm 12.1. A function to generate pseudorandom numbers using an internal state, using Equation (12.1).

Algorithm 12.1, we use a **struct** to keep track of the internal state of the PRNG, along with the choices of variables that make it up. We can choose some values for this simple random number generator:  $a = 3697$ ,  $b = 4397$  and  $m = 65521$ . We can initialise our PRNG with the state of  $X_0 = 42$ :

```
julia> rng = SimplePRNG{UInt32}(42, 3697, 4397, 65521);
```

We can now collect some values using the **rand** function defined earlier:

```

julia> [rand(rng) for _ in 1:5]
5-element Vector{Float64}:
 0.4369438805879031
 0.4486347888463241
 0.669922620228629
 0.7710352406098808
 0.5843927900978312

```

We can reset the internal state of our PRNG and see the same numbers being generated:

```

julia> rng.X = 42;
julia> [rand(rng) for _ in 1:5]
5-element Vector{Float64}:
 0.4369438805879031
 0.4486347888463241
 0.669922620228629
 0.7710352406098808
 0.5843927900978312

```

If the process of an RNG is ultimately deterministic, how can we evaluate what makes a good PRNG? The quality of a PRNG can be measured via the following statistical tests:

1. The periodicity of the process. As PRNGs rely on an internal state, chosen from a finite number of states, eventually all PRNGs will cycle to a previously visited state and being repeating. This gives some periodicity to the PRNG, which can make the process predictable and unsuitable for certain use cases.
2. Bias of the process. A PRNG generates numbers from a given distribution. For example, a method that produces floating-point random numbers between 0 and 1 will not output each possible number with equal probability. Certain numbers within this distribution will be biased towards.
3. Speed of the process. Certain applications such as computer graphics require fast generation of random numbers to render effects in real time. This means that the algorithms that produce them must be optimised to run quickly, while still produces high enough quality random numbers.

4. Predictability. If one has access to one, or a sequence of, random number(s) generated from a PRNG, sometimes it is possible to predict the internal state of the PRNG and therefore one can know all future predictions. Poor results of the previous traits can lead to a predictable PRNG, which is unsuitable for tasks such as cryptography.

We can run these statistical tests on our chosen example PRNG, choosing the initial internal state to be 42. Let's write an algorithm to determine the cycle period of our PRNG:

```
@inline get_state(rng::SimplePRNG) = rng.X
function measure_cycle(rng)
    visited_states = Set{get_state(rng)};
    n = 0;
    while true
        n += 1
        rand(rng)
        next_state = get_state(rng)

        if next_state in visited_states
            break
        end

        push!(visited_states, next_state)
    end

    return n
end
```

Algorithm 12.2. An algorithm to test the period of a random number generator which has already been seeded. One can test other random number generators by defining `get_state` for the specific type.

The cycle length of our PRNG is:

```
julia> rng.X = 42;
julia> measure_cycle(rng)
65520
```

As our PRNG is only capable of generating  $m - 1$  different numbers, it is good that our generator produces all of them before cycling.

For the next statistical test, we can generate a histogram of the values produced between 0 and 1 for  $10^4$  samples to visually inspect the output for any bias.

Looking at Figure 12.1, one can see that this random number generator is very poor at generating a uniform random number between 0 and 1, and would be

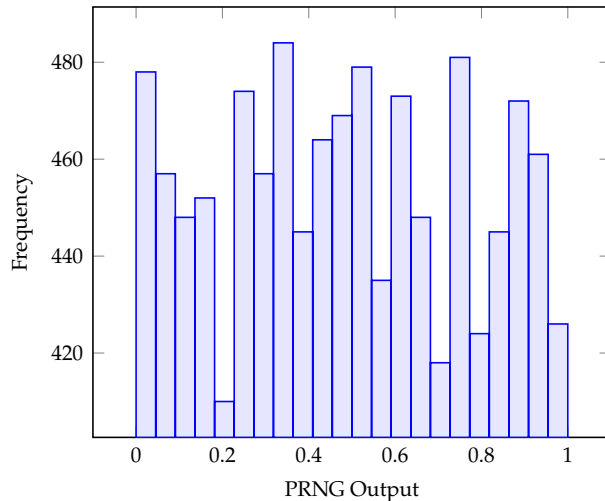


Figure 12.1. A simple distribution of  $10^4$  random numbers generated using the PRNG in Algorithm 12.1. The parameters of the PRNG were:  $X_0 = 42$ ,  $a = 3697$ ,  $b = 4397$  and  $m = 65521$ . This was generated starting with a seed of 42. One can see that this is a poor random number generator for numbers between 0 and 1.

unsuitable in many scientific applications which require the random number generation to be uniformly random.

We can apply the same test of the histogram using a standard default random number generator - the Xoshiro algorithm, which can be generated with the following code:

```
julia> using Random;
julia> mersenne_rng = Random.Xoshiro(42);
```

Many pseudo-random number generators use the Mersenne Twister RNG, but in Julia, the Xoshiro implementation is much faster, and is now the default.

The number supplied as an argument is called the **seed** of the PRNG. This is used to generate the initial state of the PRNG. In our previous example, our seed directly was the initial internal state, however, usually a function is used to convert the initial seed into a more suitable state.

## 12.2 Managing Software Versions

Another common problem when other people run your code, is that it does not work the same way on their machine as it does on yours. One of the key causes of this problem is software versions, particularly the versions of packages used

within your code. Julia makes this process relatively simple and should be a practice on every single project you use.

A new project, or even an existing project, should be self-contained within a single folder, preferably in a source controlled repository. Within this folder, one should create an environment for developing and running the code within. Environments have existed in various forms across multiple programming languages for many years. They serve as a record of required packages, along with their versions, that are used within the project. In Python, one can use Anaconda to manage these environments, and also to install the packages. Julia has a built-in package manager, which also provides a mechanism for specifying an environment that can be checked into source control.

The *environment* in Julia is just a simple *Project.toml* file. One should never need to manually edit this file, but instead, interact with it through the package manager. Open up the Julia REPL, inside your current repository, and run the following command to create that project file:

```
using Pkg;
Pkg.activate(".");
```

Of course, one can replace the `"."` with a fully specified path as well. One can also go into the package manager mode in the REPL by pressing the `]` key. You may need to add a package to see the *Project.toml* file appear. To do this, you must know the name of a package. After activation, type the following to add *BenchmarkTools.jl* to the current environment:

```
Pkg.add("BenchmarkTools")
```

This will ensure the creation of the environment file, which should be checked into source control. If you are using code from another person, when opening up the folder in the REPL, you should activate the environment of the current folder, and then run the *instantiate* command, which will download and install all the required dependencies in the project:

```
Pkg.instantiate();
```

Julia will usually download these packages into the current user's home directory. If the same package is used by multiple environments in different projects, then Julia will only install the package one and reuse it between each environment. This approach makes the environment system very lightweight and portable.



Alongside the environment file (*Project.toml*), one will also see a *Manifest.toml*, which should not be checked into source control as it contains all the specifics of packages currently installed on a system. This can vary from user to user and may conflict between different operating systems and users. Additionally, most of the information in *Manifest.toml* is superfluous when one already has *Project.toml*. When a user instantiates an environment, it will create the *Manifest.toml* file. The main difference is that the manifest contains all of the specific details of the packages currently installed, down to the minor version. While the environment file may specify a package to use, most of the time, any version of the package, beyond a version number will suffice. One can introduce restrictions on a package version by using an equality operator:

```
Pkg.add("BenchmarkTools>1.0")
```

This will allow any versions of the specified package above a major version of 1. When instantiating, Julia will fetch the latest package which fits the restrictions in the environment.

All development for the package should be done whilst the current environment is activated, so that any packages used will have to be added to the environment file. All of these changes will then be uploaded to source control. When another person is using your code, they can quickly activate and instantiate the environment and run any code they need straight away, with no further setup.



## 13 *Documentation*

Documentation is often overlooked in scientific software projects, especially research oriented projects. However, the lifetime of a software project can often be underestimated, with the same piece of code being passed down through multiple generations of PhD students. It is common for developers to forget what even their own code explicitly does, when it is written multiple months ago. This section will cover some best practices that will help make the code easy to understand and use and last longer. Documentation is a process that happens *while* writing the code in the first place, it is not strictly a process that happens once the code is completed.

It is important to cover what is meant when we talk about documentation. Asking many people, especially scientists that code, they would say that commenting one's code is the way to document it. This means that people that come after can read the comments to understand the function of the code. The main point one should take away from this chapter is that this view of documentation is completely **false**. This method is a bandage applied to poor programming practices in order to salvage some future readability and usability of the code.

### 13.1 *Commenting is **not** Documentation*

This will be probably the most surprising section of this chapter. This is not the narrative that is taught in many programming classes, especially in the sciences. However, this is a practice that has been followed by the mainstream software engineering industry for many decades. In his book, *Clean Code*<sup>1</sup>, Robert Martin dedicates an entire chapter to comments and what makes a good or bad comment. The one quote from that chapter to take away is summarised in just one short sentence:

<sup>1</sup> R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

The proper use of comments is to compensate for our *failure* to express ourselves in code. - Robert C. Martin

Note the use of the word *failure*. Each comment used is a capitulation to making the code itself understandable. This is the first principle to understand about documentation:

1. Code should be self-documenting wherever possible.

Let's look at an example to see this concretely:

```

"""
    d(a, b)

Calculates the distance between a and b,
where a and b are vectors.
"""
function d(a, b)
    # Check the make sure the vectors are the same length
    @assert length(a)==length(b)

    # Calculate the vector difference of a and b
    dlta = b .- a
    # Calculate sum of the squares of the difference
    l2 = sum(dlta.*dlta)
    # Calculate sqrt of the square sum to find dist
    l = sqrt(l2)

    return l
end

```

The above function is readable, but there are many comments. The first is known as a *docstring*, which is recognised by the Julia runtime, so that one can access the documentation for a function without having to visit the source code directly. This will be covered in more detail later. The docstring describes the purpose of the function and how it is meant to be called. Notice that here, this docstring stands in for the failure of the name and signature of the function itself to describe its purpose. The method signature is `d(a, b)`, which is utterly non-descriptive. Instead, we could avoid the docstring completely by using a comprehensible function name, such as `distance(a, b)`, which can be read as calculating the distance between variables `a` and `b`. There is still a lot to be desired here, as the distance is a metric which is somewhat ambiguous. Does one mean

the Euclidean distance, or the Manhattan distance? This disambiguation is what should be included in the docstring.

Next, moving on to the body of the function, we notice that there are checks to make sure that `a` and `b` are the same length. Notice that we are also expressing the need for `a` and `b` to be containers with the `length` function defined. Later on, we also impose the need for broadcasting to be defined on these objects, with the `-` function defined on the elements. We can continue with this analysis, but in Julia, these conditions are all met by any type derived from `AbstractVector`. We can document this need by explicitly defining the arguments of this function for only types derived from `AbstractVector`.

Finally, notice that the comments exactly describes what the following line of code does. Why doesn't the code itself tell you what is happening? The name `delta` is non-descriptive, a better name may be `a_to_b`, which describes the vector going from `a` to `b`. However, I believe that it is actually better to combine the methods together, as Pythagoras' theorem is well known. The resulting changes to the function become:

```
"""
    distance(a, b)

Calculates the Euclidean distance between a and b.
"""
distance(a<:AbstractVector, b<:AbstractVector) = sqrt(sum((a.-b).^2))
```

For more complex functions, it is usually better to break the function down into multiple lines. But this function is a simple one, which has a well known formula in the real world. The name of the function describes what the function does. The description in the docstring points to a "Euclidean distance", which can be looked up online to find out more, which will make the formula used obvious. Finally, the type signature makes it clear that any vector like type can be used, making it clear what sort of arguments can be used. Importantly, all of our changes allowed us to write a single line function definition. Short functions are easier to understand. This does not mean that one should write a huge statement on a single line, but the function easily fits into the reader's field of view. One should notice that we also removed the check to make sure that the vectors are the same length, since the function will already error on the broadcast. This is acceptable here, as the function is short enough to see the source of the error. However, on longer functions it is often better to error as early as possible.

However, one may take a look at this example, and argue what's the real harm in commenting a function in this way? For this example, the use is innocent enough, however, let's take a look at the following example:

```

"""
    fit(x, y)

Fits a linear function  $y=mx+c$  using least squares method.
Returns (m, c).
"""
function linear_fit(x, y)
    n = length(x)
    @assert n==length(y)
    sum_xx = sum(x.*x)
    sum_xy = sum(x.*y)
    sum_x = sum(x)
    sum_y = sum(y)
    denominator = n * sum_xx - sum_x*sum_x
    m = (n*sum_xy - sum_x * sum_y) / denominator
    c = (sum_y*sum_xx - sum_x * sum_xy) / denominator

    return (m, c)
end

```

Other than a few, perhaps poor, variable names, this function is perfectly okay. Despite the use of single letter variable names, this is only acceptable due to the mathematical context of this function. Commonly functions are expressed as  $x$  being an input to  $y$ . Additionally,  $n$  is commonly used as the length of a vector, or number of points in a list. This variable could be replaced with a more descriptive name such as `num_points`, but this would decrease the readability of the equations below.

However, let us imagine that we are creating a package where one can have a model which maps from  $x$  to  $y$ , which we represent with a struct. Let's take a look at the type definition:

```

abstract type AbstractModel end
struct LinearModel{T}
    slope::T
    intercept::T
end
predict(model::LinearModel, x) = model.slope * x + model.intercept

```

Now in our fit function, we can modify the code to return one of these models:

```

"""
    fit(x, y)

Fits a linear function  $y=mx+c$  using least squares method.
Returns (m, c).
"""
function linear_fit(x, y)
    n = length(x)
    @assert n==length(y)
    sum_xx = sum(x.*x)
    sum_xy = sum(x.*y)
    sum_x = sum(x)
    sum_y = sum(y)
    denominator = n * sum_xx - sum_x*sum_x
    m = (n*sum_xy - sum_x * sum_y) / denominator
    c = (sum_y*sum_xx - sum_x * sum_xy) / denominator

    return LinearModel(m, c)
end

```

This modification was easy enough, however, we have just introduced an error without thinking! Now the docstring for this function *lies*. This function does not return a tuple of (m, c), but instead returns a `LinearModel` struct. Not only this, but we have introduced breaking changes across our code base, and any other code base that uses this code. When refactoring and improving code, comments (in this case the docstring) often gets left behind, reflecting the state of how the code *used* to be. The fewer the comments one has, the fewer of these insidious errors one will encounter.

Fortunately, Julia specifically has a mechanism for making docstrings robust, since they provide a description of the developer facing API of our functions. We can introduce a test for this docstring, on the original function:

```

"""
    fit(x, y)

Fits a linear function  $y=mx+c$  using least squares method.
Returns (m, c).

# Examples
```jldoctest
julia> x = [1, 5, 9];
julia> y = 3 .* x .- 7;

```

```

julia> linear_fit(x, y)
(3.0, -7.0)
```


```

"""
function linear_fit(x, y)
    n = length(x)
    @assert n==length(y)
    sum_xx = sum(x.*x)
    sum_xy = sum(x.*y)
    sum_x = sum(x)
    sum_y = sum(y)
    denominator = n * sum_xx - sum_x*sum_x
    m = (n*sum_xy - sum_x * sum_y) / denominator
    c = (sum_y*sum_xx - sum_x * sum_xy) / denominator

    return (m, c)
end

```


```

This will now add a *unit-test*<sup>2</sup> to this function to ensure that the documentation stays up to date with the function. This is a special type of unit testing which makes sure that examples are up-to-date with the current code. The functionality for running these tests is in *Documenter.jl*, which is documented in the base Julia docs<sup>3</sup>.

Let us summarise the points seen in this section.

- Function names should be descriptive as to what the function actually does. If the function does many things, it is often better to split the function into multiple, shorter, functions.
- Argument names should be descriptive as to their purpose, or their purpose can be easily inferred from the function name and the type descriptions.
- Comments describing *what* the code does usually indicate that the code is written poorly. This is a failure of the code to express what is happening. This can usually be fixed by better variable names, or splitting complex sections into smaller functions.
- Comments can be used as a last resort to disambiguate a function, in order to keep function/variable names short and concise.

<sup>2</sup> We cover *unit-testing* in more detail in Chapter 14.

<sup>3</sup> JuliaLang, *Julia Docs - Documentation*, <https://docs.julialang.org/en/v1/manual/documentation/>, Accessed on 22nd July 2022.



- Using many superfluous comments vertically expand a function, increasing the chance that it will expand beyond the typical field of view of the reader, making the function harder to understand in its entirety.
- Type restrictions can stand in for comments, as this documents what type of objects should be used on these methods. However, this is the weakest form of documentation in Julia, since this can restrict the reusability of one's code and one should avoid typing most variables.
- Code changes over time, and not only the code, but the documentation, must be updated to reflect these changes. However, this is an additional responsibility for the developer. Using automated techniques for testing documentation is recommended when possible. Otherwise, reducing comments in favour of readable and understand code will reduce the maintenance burden of comments.
- Comments can be used to reflect the *why* of design choices, as long as they are short and concise. This avoids other programmers repeating the same mistakes as the original designers. However, it must be recognised that this often reflects a failure in the code design, and is only a bandage. It is often better to spend time fixing the problem that caused the requirement of the comment in the first place.

### 13.2 *Better Variable and Function Names*

We have already touched on this topic throughout the book, but it is a topic that needs special mention among a scientific audience. We, as scientists, are often fluent in the language of maths, feeling perfectly comfortable with symbols representing concepts. It is common to project this onto programming, using a single letter to represent a variable or a function, but this is a habit worth shaking. In this section, we'll look at the reasons for avoiding poor variable names, and when short, non-descriptive, variable names can be acceptable.

Linking back to the previous section, poor naming can introduce the need for comments to explain the code. As we have already seen, commenting code is usually superfluous, and a more descriptive name for a variable or function is usually not much more effort.

As scientists, the type of code we write, usually involves translating equations into code. Used within a method, shorter variable names and lead to cleaner equations that are easier to see. Let's take a look at a function which calculates the force between two objects:

```
struct NBody{Q, T}
    r::Q
    m::T
end
raw"""
    force(a, b)
```

Calculates the force felt by body a, due to body b.  
Uses Newton's Law of Gravitation:

```
```math
\overline{F}_{ab} = \frac{G m_a m_b}{|\overline{r}_b - \overline{r}_a|^3}
(\overline{r}_b - \overline{r}_a)
```
```

Result is a vector, measured in SI units (Newtons).  
"""

```
function force(a::NBody, b::NBody)
    G = 6.674e-11 # m^3·/ kg / (s^2) (SI Units)
    Δr = b.r-a.r
    r² = sum(Δr.^2)
    F = (G * a.m * b.m) / (r²)^(3/2) * Δr
    return F
end
```

This function is not actually that bad, despite the single letter variable names. This even makes use of Julia's support for unicode characters. This is all down to context. The function is small enough that the entire context, including the docstring, is accessible to the reader. Additionally, the equation used is directly put in  $\LaTeX$  format in the docstring itself, which is accessible to anyone using the function. Anyone familiar with some basic physics will be able to recognise the symbols for what they are, because of the familiar notation.

But one can still think, is this notation standard across all educational backgrounds? The answer to this thought is usually no for most topics. Notation is never consistent across a large array of people. For this reason, there are choices we can make to make our code more understandable. First of all, when using **structs** to represent data, we should aim to provide descriptive variable names

to each of the fields, since these are the most commonly used, and usually used far from the context of the definition. A better choice for the struct would be as follows:

```
struct NBody{Q, T}
    position::Q
    mass::T
end
```

This now improves the readability, wherever this code is used. The only remaining issue is with the notation. The function uses Unicode symbols to better represent the underlying mathematical symbols. However, one should not expect that everyone using or reading your code will have access to a Unicode supporting editor. This might make the code illegible on those systems. A common example for this is on terminal based editors, such as *Vi*, which one may use while editing code on a HPC. These symbols will not show up properly there, which should be kept in mind.

The variables within a function having proper names is not as important as the naming of the function itself, including the arguments. The line that defines a function and its arguments is called the *method signature*. In Julia, one can search the help via the name of a function, so it is important that the function have the most useful and obvious name for the functionality being encapsulated. Not only this, but this method name will be used in other functions as well, which reduces the need for commenting the code as this code becomes more readable too. Just as important, is the naming of the arguments. One should know what one can pass into a function. In Julia, it is often discouraged to specifically type the variables, and so one cannot always rely on a type definition to inform one's understanding. One can always use docstrings to help, but if a better variable name will suffice, then the docstring is entirely superfluous.

### 13.3 Automatic Documentation

As seen in previous parts of this chapter, we have covered the use of docstrings to document the proper use of our functions. We were even able to inject testing into the docstrings to make sure that the functions do what they were meant to do. This section will cover the basics of writing a good docstring, but before we dive into the details we must first ask the question - what is the purpose of a docstring?

Docstrings are designed as a way for a program to crawl through your program and create a complete set of documentation from your source code. Many packages you will see online, including the Julia documentation itself, is usually automatically generated using the information contained within docstrings. As this is usually the purpose, we can say that a docstring provides a means by which one can document the public API<sup>4</sup> of a project. Note the use of the word *public*. These are the functions that you intend to be used externally, outside the library. Additionally, one can use it to document functionality inside the library for use by other developers. As we are focused on Julia, which has no mechanism for hiding code, the public API of a library is usually defined as those functions which are exported from the library. Many code bases will have numerous internal functions that are not meant to be accessed by outside parties. Many of these functions will not need docstrings, and they can often be superfluous and have all the negative downsides discussed in the earlier section on comments.

Having a public facing API, which is well-documented, is a huge boost to usability of your code. If you are writing a library which is used by others, it is the first port of call when learning to use your code. If your documentation is well written, one need not dive into the source code itself. A public facing API is a very good thing to have as it creates an implicit contract between the library and the users. It defines how the code should be used, and what should not be used. This gives the library developers the freedom to improve and modify the underlying code, without changing the API. This can be as simple as bug fix or performance improvement or a drastic architecture design change, as long as the public API continues to stay the same, the downstream consumers will still be able to use your library. Changing this public facing API will usually involve breaking changes downstream, which can seldom be avoided, but good documentation, including being able to show changes from one version to another can ensure that any breaking changes are known about and consumers can resolve these breaks in their own code.

Maintaining documentation has famously been a difficult task. In the past, many libraries had very poor documentation, and it was rare to find a project which was well-documented that didn't have a lot of funding behind it. Developers became developers because they enjoy writing code, not necessarily because they enjoy writing documentation. But when there is a boring, monotonous task, you can count on a programmer to try and automate the process. In almost every modern language, there are tools for scraping the source code and automatically

<sup>4</sup> Application Programming Interface. In this context, it means the set of modules, structs and functions exposed to a consumer of the code.

producing beautiful and useable documentation. In Julia, this is provided by *Documenter.jl*<sup>5</sup>. While this book will not provide a full tutorial for using this package, just know that one can generate most of the necessary documentation directly from docstrings, which will be discussed here. Details on setting up *Documenter.jl* in your own project will not be covered here.

<sup>5</sup> JuliaDocs, *Documenter.jl*, <https://juliadocs.github.io/Documenter.jl/stable/>, Accessed on 22nd July 2022.

We have already seen the basics of documenting a function here, it starts with using the `"""` string notation that is copied from Python:

```
"""
    double(x)

Returns twice the input.
"""
double(x) = 2x
```

This allows us to write a string across multiple lines. It is entirely possible to write a docstring using a single `"`, but this is discouraged, as a docstring has a format.

The first rule of a docstring is that the first line should be the method signature (or a simplified version of it), indented by exactly 4 spaces to indicate that it should be formatted as code. One can also write optional arguments with square brackets as in the following example:

```
"""
    ntimes(x[, n=2])

Returns n times the input.
"""
ntimes(x, n=2) = n*x
```

One can also separate out the arguments onto multiple lines under a heading:

```
"""
...
# Arguments
- `x`: The number to be multiplied.
- `n::Integer`: The number to multiply by.
...
"""
```

In this case each argument has its own line and begins with a `-`, with the name of the argument surrounded by backticks<sup>6</sup>.

If one has some related functions which may be of interest, one can include a “See Also” section, as follows:

<sup>6</sup> The backticks usually represent a block of code within the docstring.

```

"""
...
See also [double!](@ref), [triple!](@ref), [quadruple!](@ref).
...
"""

```

Which will reference the functions `double!`, `triple!` and `quadruple!`, using a special notation for a link.

One of the most important sections for creating high quality documentation is the use of an examples section. We have already seen this before in a previous section, as we were able to use this to write a doctest. Go back to the previous section to see an example.

Finally, one can also format LaTeX formulae within a function definition, which will certainly be useful for documenting scientific packages. Revisit the previous example to see this in action. Copy the example into your REPL and search for the help on the function and see what is displayed.

This section is not a comprehensive overview of how one can use *Documenter.jl*, but be sure to consider it when sharing your code with others, and read their documentation for further guidance.

### 13.4 Summary

If there is anything to take away from this section, it is that no choices are perfect. It can be quite difficult to think of a good name for a variable, or hard to choose a variable name that does not impede the readability of a formula. For this reason, it will always come down to a judgement decision based on who will be using your code, and what is its purpose. Becoming better at writing readable code, that is well-documented, is a skill which takes practice. It often does not take too much more time, and can even save you time in the long term when you come back to try and use the code in the future.

One should remember that there are many tools out there which make the job of documenting your code much easier. IDEs will use the comments you write to give you reminders and context when you are using these functions, making for a more pleasant and error-free development experience. It is not always a tedious and difficult task.

As a test of your understanding, go back to a piece of code you have written and see how it can be refactored for readability. See how long it takes you to

make the code easier to use and better for a user that has never seen the code. The resulting code is likely something you will be proud of and want to show others.





## 14 Unit Testing

How often do you find yourself running some code that worked yesterday, but is now broken? This happens very frequently when you are quickly iterating on your code, especially in a dynamic language like Julia or Python, where a misspelled variable during compilation and instead crashes your program as it runs. If this error is behind checks that happen infrequently, these bugs may go unnoticed for a long time. Unit Testing is a practice that has been thoroughly developed in the software engineering industry which, when implemented correctly, can help avoid these bugs, and ensure that any new functionality does not break existing functionality.

A unit test, put simply, is just a way of checking that a piece of code functions as expected. To come up with an example, let us implement a two-dimensional vector, and provide some functionality:

```
using Base
import Base: +, -, *, /, ==
struct Vector2D{T}
    x::T
    y::T
end
(+) (a::Vector2D, b::Vector2D) = Vector2D(a.x+b.x, a.y+b.y)
(-) (a::Vector2D, b::Vector2D) = Vector2D(a.x-b.x, a.y-b.y)
(*) (a::Real, b::Vector2D) = Vector2D(a*b.x, a*b.y)
(/) (a::Vector2D, b::Real) = Vector2D(a.x / b, a.y / b)
(==) (a::Vector2D, b::Vector2D) = (a.x==b.x && a.y==b.y)
```

Algorithm 14.1. A simple implementation of a 2D vector type in Julia.

This was relatively easy to do, but how do we ensure that it works the way we expect? We can manually test this of course:

```

julia> a = Vector2D(2, 2);
julia> c = 1.5;
julia> c*a == Vector2D(3, 3)
true

```

However, this code is just what we wrote in the REPL. This code is not reproducible anywhere else, and usually does not cover all of our code. What if we suddenly change the implementation? Then these checks are not up-to-date. Instead, we put this code into a different file which runs these checks for us. In-built to Julia is the *Test.jl* package, which provides some macros for writing these tests. Let's take a look at what one of these files looks like:

```

using Test;
# Make sure you have access to the right files
include("vector2d.jl")

a = Vector2D(2,5)
b = Vector2D(10, 20)
@testset "Test vector addition" begin
    @test a + b == Vector2D(12,25)
    @test b + a == Vector2D(12,25)
end
@testset "Test vector subtraction" begin
    @test a - b == Vector2D(-8,-15)
    @test b - a == Vector2D(8,15)
end
@testset "Test scalar multiplication" begin
    @test 5 * a == Vector2D(10,25)
    @test a * 5 == Vector2D(10,25)
end
@testset "Test scalar division" begin
    @test a / 5 == Vector2D(0.4,1)
end

```

Notice that we have covered all the different methods that we wrote. Each invocation of the `@testset` macro, allowed us to write a block of code that calls the functions we wrote. These tests are broken up into different units to allow us to identify the precise part of the program that broke, if the tests should fail. These tests, along with the others for our code, can all be run together after making any changes, to ensure that the programs runs as expected. There are a few ways to run these tests, but the simplest is to just include the file, which I have saved in *code/vector2dtest.jl*:

```

julia> include("code/vector2dtest.jl")
Test Summary:           | Pass  Total
Test vector addition |     2     2
Test Summary:           | Pass  Total
Test vector subtraction |     2     2
Test scalar multiplication: Error During Test at code\vector2dtest.jl:16
  Test threw exception
  Expression: a * 5 == Vector2D(10, 25)
  MethodError: no method matching *(::Vector2D{Int64}, ::Int64)
  Closest candidates are:
    *(::Any, ::Any, ::Any, ::Any...) at julia\base\operators.jl:655
    *(::T, ::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16, UInt32, UInt64, UInt8} at julia\base\operators.jl:655
    *(::StridedArray{P}, ::Real) where P<:Dates.Period at julia\stdlib\v1.7\Dates\src\deprecated.jl:15
  ...
  Stacktrace:
  ...
Test Summary:           | Pass  Error  Total
Test scalar multiplication |     1     1     2
ERROR: LoadError: Some tests did not pass: 1 passed, 0 failed, 1 errored, 0 broken.
in expression starting at D:\Development\Julia\fictional-computing-machine\code\vector2dtest.jl:15

```

Notice that the macros show which of the tests have passed, with a name labelling the purpose of each unit test. Notice that this test failed the scalar multiplication test. This gives us the line at which this failed, and the reason why. We did not define a method for multiplying a vector by a number, only a number multiplying a vector. We cannot assume associativity, we have to implicitly define this. In our example we simply add the line:

```
(*)(a::Vector2D, b::Real) = b*a
```

This will fix that specific unit test. However, we notice that this exception stopped all unit tests from running. The final division unit test also fails with the error:

```

Test threw exception
Expression: a / 5 == Vector2D(0.4, 1)
MethodError: no method matching Vector2D(::Float64, ::Int64)
Closest candidates are:
  Vector2D(::T, ::T) where T<:Real

```

This is because we specified the types had to match. Again, this is easy to fix by adding a variant of the constructor:

```
Vector2D(x, y) = Vector2D(promote(x, y)...)

```

A final run will tell you that all tests pass. There are a lot more ways in which to test this data structure, which we will not go into now. Most importantly, we have some code which automates the verification process of our code. The rest of this chapter will give you an overview of why this helpful and provide a framework for high quality unit tests.

### 14.1 *Why test your code?*

As with most things, the answer to this section has a lot of nuances. Here, I will try to focus on the potential reasons for an academic to care about unit testing, as this is an area that is almost completely neglected in research.

#### 14.1.1 *Ensuring correctness*

The biggest reason for testing your code is to make sure that it works. This means that the algorithms are correct, and produce the results that you expect. When programs become more complex, knowing if the program is producing the correct answers can be quite difficult, and testing these cases usually feels circular as you need the result to tell you if it is correct. However, one should be able to break down a complex algorithm into smaller chunks, sometimes just a single function with a few lines of code, and thoroughly test the individual chunks. This can give the developer some assurance that if the smaller chunks are correct, then assembling them into a more complex structure is less likely to have errors. One can also write simpler tests to make sure the complex aggregate functions also function correctly with a few examples, knowing that the lower level routines are also robust through their own tests.

In research, having small bugs in your software which affect your final output can completely invalidate an experiment. Researchers should always be certain that their code is doing exactly what it is meant to be doing. What better way of doing this, than checking to see if the functions themselves give the answers you expect. There are many occasions when you write a general algorithm to calculate numerical answers to problems, which only have analytic solutions in specific cases. These specific cases can be used to verify that the algorithm is producing the correct results, as there is a known answer calculated by different means. If your algorithm does not match the known case, this can be the point at which

you discover the error so that you do not waste a lot of time experimenting and producing results with a flawed algorithm.

Ensuring correctness of your code is the central tenant of unit testing, all subsequent discussions of why to unit test, are just finer points which all lead back to the question of correctness.

### 14.1.2 *Optimisation*

This book, after all, is designed around the idea of writing high performance code. Often times, when people are optimising, they make many assumptions about what they can and cannot do, which may introduce bugs into the program. Often times, the safest way to optimise a function is to make sure that this function has unit tests that cover a wide variety of use cases for that function. Once these unit tests are written and known to pass with an algorithm, one can spend time focusing on optimising it, knowing that if one introduces a bug at any point, it is highly likely that this will be caught by the unit test. If this practice is accompanied by the use of regular commits with source control, a bug can be traced down to a specific commit by using the unit tests, and seeing which change introduced the new bug.

Instead of rewriting the same function for a more optimised version, one can start writing a parallel “fast” implementation of an algorithm alongside the original “slow” version. The unit tests become very simple, for the same inputs, both algorithms should produce the same answer. This makes it very easy to ensure that any new algorithm is at least as correct as an existing algorithm. Once the fast version is verified to be correct, it can take the place of the slower one, and the slower one can be kept in the unit tests to ensure that the faster one will always be correct, even if future modifications are made.

### 14.1.3 *Reproducibility*

A large part of science is making sure that results can be repeated and reproduced by independent teams of researchers. Numerical research, unlike experimental research, has an incredibly high advantage when it comes to fulfilling this expectation of good scientific practice. A well written computer program, can be sent to another team which can run the code with the same inputs and achieve the same outputs. Open-science is only just becoming popular, following in the

footsteps of the open-source movement, but it has such potential to accelerate scientific discovery and cut down on research based on incorrect and spurious results as new results can be easily verified by other researchers.

Making your code easily unit tested forces the code to be written in a way that it can be run by anyone. Unit tests should be simple and self-contained, making sure that results gained were not just one-time flukes, but can be reproduced at will. Unit testing defines a quick and simple procedure to run code on different machines to ensure that each machine is capable of reaching the same result.

One caveat that needs more discussion is how to ensure reproducibility when the underlying methods are inherently stochastic? This question is answered later in the chapter.

#### *14.1.4 Documentation*

One often over-looked use of unit testing is the ability to document the use of your code. This is not a replacement for well documented software, but can provide a base of examples for how different functions and parts of your code are expected to interact. Someone else using your code may not be entirely sure of the purpose of a function, but can see how it is used in the context of a unit test and better understand why it is there. Often, unit tests provide a perfect minimum working example (MWE) for a certain problem, which can be easily understood by readers of the code.

## *14.2 Writing Good Unit Tests*

This section could be a chapter in its own right, and many people have written extensively on the topic. This section does not aim to be a complete overview to the topic, but only an introduction. Here, we will discuss some key points in which to consider when writing a unit test.

### *14.2.1 Identifying Purpose*

Before writing a test for a function, make sure you know what you are testing for. A unit test should be small and specific. A good rule of thumb is to make the title of the unit test as specific as it needs to be to capture the entirety of the test. In this case, we should use an example:

```

@testset "Ensures that the database has created the customer, product, supplier, inventory and staff
    db = create_db(in_memory=true);
    @test has_table(db, "customer")
    @test has_table(db, "product")
    @test has_table(db, "supplier")
    @test has_table(db, "inventory")
    @test has_table(db, "staff")
end

```

This example is not actually that bad, as it is testing database creation and table population. However, when one writes out what is actually happening in this test, it becomes clear that a lot is going on. Instead, a better design would be splitting up this set of tests into smaller parts:

```

@testset "Database creation" begin
    db = create_db(in_memory=true);
    @testset "Customer table creation" begin
        @test has_table(db, "customer")
    end
    @testset "Product table creation" begin
        @test has_table(db, "product")
    end
    @testset "Supplier table creation" begin
        @test has_table(db, "supplier")
    end
    @testset "Inventory table creation" begin
        @test has_table(db, "inventory")
    end
    @testset "Staff table creation" begin
        @test has_table(db, "staff")
    end
end

```

These two examples are practically the same, except the second example names specifically what is being conducted inside. In most cases, it is best practice to make sure that the number of `@test` macros is limited to as few as possible. It can be tempting to write all of these tests together, as having to run a function multiple times might result in slow testing, however, Julia's unit testing design makes it easy for us to do this while still being efficient; in this case, the database is only created once for all of these tests.

A single unit test should have a specific purpose and test only that one thing if possible. If a function performs multiple functions, such as the `create_db` method, one can still test individual items on their own, leaving the others unaffected.

### 14.2.2 Black Box vs White Box Testing

A “black box” usually refers to an item that performs a task which is completely opaque to us. We are often ignorant of the inner workings of such an item, and can only observe what goes in and what comes out. While in our code, we can easily inspect what goes on inside a function, it is beneficial to take a more withdrawn approach and simply test if the outputs match our expectations. This is the type of testing we conducted earlier in the chapter with our 2D vector implementation. For many research cases, this is clearly the way to go.

A “white box” test, other the other hand, can be used to make sure the internals of the functions are doing what is expected of them. These tests are less common than black box tests as they are usually heavily reliant on the current implementation of the function as opposed to the behaviour. If one knows what functions will be called within a method, one can use a technique known as “mocking” to force a certain type of temporary behaviour.

One case in research that can come up a lot is the use of random numbers, which alter what an algorithm does or its results. These functions can often be very difficult to test, and instead one can mock these functions, to replace their outputs with known results. Let us take the case of a simulated annealing method, whose pseudo-code can be written as:

```
function step!(state, perturbation, loss_fn, s)
    current_loss = loss_fn(state)
    state .+= perturbation
    new_loss = loss_fn(state)
    if rand() < exp(-s * (new_loss - current_loss))
        # Accept the perturbation
        return true
    else
        # Reject the perturbation
        state .-= perturbation
        return false
    end
end
```

This method accepts or rejects a perturbation applied to a given state, stochastically, with a probability defined by the parameter  $s$  and the difference of some quality function comparing the current state, to the new perturbed state. This function however has some stochastic behaviour that is very hard to control. In this case, we can mock the `rand` function so that it always returns a certain value.



During research for this book, the only library available for this technique in Julia was *Mocking.jl*<sup>1</sup>. This library makes the process of mocking relatively simple:

<sup>1</sup> <https://github.com/invenia/Mocking.jl>

```
using Random
using Mocking
using Test
# ... define generate_problem() function
Mocking.activate()
rand_patch @patch Random.rand() = 1.0
@testset "Test simulated annealing step is true" begin
    apply(rand_patch) do
        state, perturbation, loss_fn, s = generate_problem()
        @test step!(state, perturbation, loss_fn, s) == true
    end
end
```

However, we need to make a few changes to our original algorithm to get this to work. Hopefully in the future, there will be packages that make this much easier in the future.

```
using Random
using Mocking
function acceptance(s, delta_loss)
    @mock r = Random.rand()
    return r < exp(-s * delta_loss)
end
function step!(state, perturbation, loss_fn, s)
    current_loss = loss_fn(state)
    state .+= perturbation
    new_loss = loss_fn(state)
    if acceptance(s, (new_loss - current_loss))
        # Accept the perturbation
        return true
    else
        # Reject the perturbation
        state .-= perturbation
        return false
    end
end
```

One can now have control over the value of *r* in the acceptance function to make sure that the correct outputs are known. Fortunately, when the `Mocking.activate()` call is not invoked, the `@mock` macro will compile away to nothing so that it has no impact on the code outside of testing.

This technique is most useful when facing hard-to-test corner cases which usually rely on disk access, networking or stochasticity. As these three things are usually written in underlying, reliable, libraries, one often need not specifically test these functions and can safely mock them.

### 14.2.3 Easily Testable Code

Often times, one can only write good unit tests if the code which is being tested is also well written. This means that functions are broken down into small chunks that can be tested in isolation. Additionally, this workflow also discourages the programmer to write functions with side effects that can be difficult to test. The process of writing the tests usually encourages the programmer to refactor the code that is being tested into something which is of a much higher quality.

The first step of writing easily testable code is to break down complex functions into smaller functions<sup>2</sup>. Ideally, a function should only be a few lines of code at most. Obviously complex algorithms often take up more screen real estate, but one should always strive to make a function as small as possible. This makes it much easier for a different person to understand what a function is doing as there are fewer lines of code to overwhelm them with unnecessary details.

One caveat in this form of testing, is when one is using stochastics in your code. However, when refactoring your code into many small functions, try to isolate where random numbers are being generated. Sometimes, it is best to write a function which takes in the random numbers as arguments. This way, the function becomes deterministic to the inputs, and moves the random effects to another function. One can test the deterministic part of the code thoroughly and robustly. This is the recommended alternative to using mocking as seen in the previous section. We can use the same example as an option:

```
function step!(state, perturbation, loss_fn, s; random_number=rand())
    current_loss = loss_fn(state)
    state .+= perturbation
    new_loss = loss_fn(state)
    if random_number < exp(-s * (new_loss - current_loss))
        # Accept the perturbation
        return true
    else
        # Reject the perturbation
        state .-= perturbation
        return false
    end
end
```

<sup>2</sup> In Julia, this can also provide a performance boost as type information is available on a function call and so more time is spent in type safe code.

```

    end
end

```

Here, we simply exposed the ability to choose the random number. It is also given a default choice for the random number so that anyone using the code will not have to generate the random number themselves.

#### 14.2.4 Isolation

The point about isolation harkens back to the previous section on writing reproducible and robust code. A collection of unit tests, also called a test suite, should be able to run in isolation from the main code and from each other. Unfortunately, Julia doesn't have the most robust system for ensuring that multiple unit tests do not conflict with one another, and as such, one must rely on an external package to make sure one isolates the unit tests from one another. The core principle behind this, is that the outcome of one test should not affect the outcome of another, especially if they are testing different things.

A package called *SafeTestsets.jl*<sup>3</sup> provides the `@safetestset` macro, which works similarly to `@testset`, but moves the code inside a new module scope. This works well when combined with separate files for testing different parts of your software, as one can have a single file to call each subsequent file, wrapped in a `@safetestset` call, ensuring that the effects from one file do not pass onto the other files.

<sup>3</sup> <https://github.com/YingboMa/SafeTestsets.jl>

The way one uses this package is very simple:

```

using SafeTestsets

@safetestset "Benchmark Tests" begin
    include("benchmark_tests.jl")
end

@safetestset "Correctness Tests" begin
    include("correctness_tests.jl")
end

@safetestset "Reproducibility Tests" begin
    include("reproducibility_tests.jl")
end

```

This one file will run all of your tests in isolation. This means that one can load conflicting packages in each section without them interfering with one another, and any imports from one file will not be available in another. It is strange that Julia does not have a way to do this by default, as isolation of unit tests are the norm amongst all other modern languages, but luckily there are packages that enable this behaviour.

### 14.3 *Bad Unit Testing*

For the most part, the more tests one has of a project, the better. Tests are very useful in providing evidence of the software working as expected. In the majority of cases, some lower quality unit tests are often better than none. Before we start talking about what makes a bad unit test, we should not discourage developers from writing tests, as practice is the only way of improving. Sometimes a bad unit test, will only be superficially bad, and will only be very benign. However, in Julia it is easy to write unit tests which have knock-on effects on other parts of the project. As these are the most serious issues, we will address them first.

#### 14.3.1 *Side Effects*

A bad unit test is one that causes side effects, or relies on side effects to work. The most common case in which this happens is using the file system as storage to test affects. Using the filesystem should be minimised as much as possible when unit testing as it introduces some abstract notion of “state” into the mix, which can often make your tests less robust.

If a unit test relies on a file existing, this means that the developer has to manage this file. If testing the code on a new machine, it might not work since the new machine did not have the file in the place where it was expected to be. In general, it is possible to write most unit tests without having to save any external state to disk. Testing should be as pure as possible, so that anyone can come and run a test, and it will produce the same result, regardless of the state of the disk of a particular machine.

Other side effects can be more subtle, for example introducing new variables into the global namespace. In general, using global namespace is often discouraged and is given the term “polluting the global namespace”. This can be as simple as importing a package, since a `using` statement will populate the global

namespace with variables and functions from the loaded module. This is often why we use *SafeTestsets.jl*.

### 14.3.2 *Slow Tests*

Unit testing can help provide a rapid iteration and development time as the developers do not need to spend as much time checking changes to their code, as the unit tests can ensure the code is correct. Many open source packages use unit testing to provide Continuous Integration/Continuous Development (CI/CD), which run unit tests every time someone makes a pull request into the main branch. This quality check is combined with other metrics, such as code coverage<sup>4</sup>, to ensure that incoming changes do not break any existing functionality. With this in mind, it is easy to see that introducing a non-performant test into the mix can potentially interrupt workflows in the future, causing people to have to wait longer to merge changes, or simply encourage people not to run the unit tests as frequently.

<sup>4</sup> A number which reflects the proportion of code in the code base that has been covered by a unit test.

Unit tests should be small and compact, and usually, should be very fast to run. The cases which a unit test covers, should be broad enough to cover a range of expected results, relying on the edge cases to ensure correctness.

### 14.3.3 *Leaving Out Edge Cases*

Often times, runtime bugs only manifest when presented with edge cases. Testing your code with only the normal expected inputs will mean that these edge cases will not be touched. One should ensure that your tests are covering the hardest edge cases so that they are robust to that. A unit test is not helpful if it does not cover the range of possible use-cases of a function.



## 15 *Code Organisation*

It is the tendency of software projects to keep growing and growing over time. Eventually, one has to seriously manage the growing complexity of the project, or else be left with an unintelligible file with many thousands of lines of code which all interact with one another. These types of project become difficult, even for the main developer, to understand. However, most pieces of software can be logically separated out into different files to make the developers job a lot easier to find what they need, and understand the software as a whole.

### 15.1 *Folder Structure*

As a starting point, one should discuss the folder structure. While for initial projects, having a single source file called *main.jl* is perfectly reasonable. Once any advanced level of complexity is reached, this convenient method begins to hinder and slow down development, as one has to continuously jump all over the file and context switch between different parts of the program. But before one separates out the code into different files, it is important to think about the folder structure.

The most common way to organise the files is to have three main folders, directly in the root of the repository. These three folders are *docs*, *src* and *test* for documentation, source code and unit tests respectively. One does not need to have all three from the start, but it can be useful to make space for them from the beginning of a project. In the root of the repository, one should also have the current environment file (*Project.toml*), which contains the information about all dependencies and project information.

A very helpful tool for setting up a repository folder structure is called *PkgTemplates.jl*, which has an interactive way of generating a folder, with the appropriate folders, along with many extras, such as documentation and unit testing.

## 15.2 Modules

In Julia, one uses the concept of a *module* to separate out code into logical chunks. Instead of every function being available in the global namespace, one can compartmentalise them into separate logical chunks, and only use the ones that are needed at any one time. A *module*, in Julia, is simply a namespace for a collection of type and function definitions, along with any global variables. The syntax for a module is very simple:

```
module MyModule
foo() = println("Hello, World!")
end
```

This will lock the `foo` function inside the `MyModule` module. This removes a lot of global namespace pollution, and allows names to be reused for different concepts without any conflicts<sup>1</sup>. In order to access this function, one must import it from the module:

```
import MyModule: foo
foo()
```

One can imagine that if one needed to import every function from a module to use it, that this way of segmenting code would fall out of favour very quickly. Fortunately, Julia provides the `export` keyword, which can be used to specify which functions should automatically be pulled into the current namespace when a user imports or uses a specified module. For example, if we modify our module to have two functions, but only want to expose the user to one of these functions we can be explicit:

```
module MyModule
bar(x) = println("Called bar with $x")
foo() = bar("foo")
export foo
end
```

<sup>1</sup>This is usually a bad practice, especially in Julia, since one can reuse the name of a function with one's own data type to add functionality.



Here, we have explicitly defined `foo` as the external API for this module. Julia does not disallow users to import a specific function from the module, but requires that it be explicit, like in the previous example. Now, the usage of this function is much more developer friendly:

```
using MyModule
foo()
```

Notice that the `foo` function can still call all the methods inside `MyModule` without a problem.

### 15.2.1 Modules in a Package

A package is defined inside the `Project.toml` file. While this can be defined manually, one should use `PkgTemplates.jl` to create this for you. Inside the generated template, one will find a single file inside of the `src` directory with the chosen name of your package. This will define a module, again with the name of your package. Inside here, one should include any other files that define modules. One can create a nested hierarchy of functionality, just as in other languages. It is important that all files with module definitions be loaded inside the main project module. This module should then use the loaded modules and re-export the relevant functions.



# References

1. T. Besard, C. Foket, and B. De Sutter, “Effective Extensible Programming: Unleashing Julia on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2019 (cit. on p. 187).
2. JuliaDocs, *Documenter.jl*, <https://juliadocs.github.io/Documenter.jl/stable/>, Accessed on 22nd July 2022 (cit. on p. 239).
3. JuliaLang, *Julia Docs - Documentation*, <https://docs.julialang.org/en/v1/manual/documentation/>, Accessed on 22nd July 2022 (cit. on p. 234).
4. P. Kharya, “TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x,” 2020 (cit. on p. 186).
5. R. Krashinsky, *NVIDIA Ampere Architecture In-Depth*, <https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/>, Accessed on 15th January 2023, 2020 (cit. on p. 185).
6. R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008 (cit. on p. 229).
7. NVIDIA, *CUDA Toolkit Documentation*, <https://docs.nvidia.com/cuda/index.html>, Accessed on 6th July 2022, 2022 (cit. on p. 199).

