Warwick Research Software Engineering

Introduction to Software Development

H. Ratcliffe and C.S. Brady
Senior Research Software Engineers

"The Angry Penguin", used under creative commons licence
from Swantje Hess and Jannis Pohlmann.

December 10, 2017

Contents

Preface

0.1 About these Notes

These notes were written by H Ratcliffe and C S Brady, both Senior Research Software Engineers in the Scientific Computing Research Technology Platform at the University of Warwick for a series of Workshops first run in December 2017 at the University of Warwick. These workshops are intended as an introduction to software development for those with some programming knowledge, taking somebody who can write simple code for themselves and introducing principles and tools to allow them to write code they can share with others and use for their research. In other words, all of the things that often get forgotten in favour of syntax and language constructs.

The notes were typeset in LaTeXby H Ratcliffe.
Errors can be reported to rse@warwick.ac.uk

0.2 Disclaimer

These talks are going to take a very practical stance and will not always agree with the beginner textbook theories you may have encountered. We're working from experience of actually writing scientific code, and we'll always try to note where something is a practical shortcut or sidestep. However perhaps the crucial thing to remember is rules are only useful until they're not, and often advice is given with an understanding that you will come to understand exactly why it was so formed, at which point you have the necessary discretion to know when to ignore it.

If you have ever read any of Terry Pratchett's Science of Discworld, for example, you may have encountered "lies to children",

> ... a statement that is false, but which nevertheless leads the child's mind towards a more accurate explanation, one that the child will only be able to appreciate if it has been primed with the lie

When statements fall into this category, they are currently useful, but one day will cease to be so. This is important to remember, and doubly important to remember if you find yourself passing on your new wisdom.

In general, we will only say that you **mustn't** do something, if we think that in all likelihood it will never be the correct thing to do. It may break standards or we see no circumstance where it would win in the cost-benefit stakes. (Although we may occasionally be wrong on that.) **Shouldn't** encompasses all of those currently, or generally, useful generalisations. If you find yourself going against a recommendation like that, think carefully before proceeding. **Must** and **should** are used similarly. And finally, if something you're trying to do seems painfully difficult or awkward, consider the possibility that there is a better way to do it.

After reading these notes, or attending our workshops, you should be in a good position to understand more detailed material, and to know what terms and phrases to search for to solve problems you encounter. Software development is a wide field with many schools of thought, often with incompatible constraints and priorities. **As researchers, your priority is research. You code needs to work, and you need to know it works. Anything that does not enhance that goal is decoration.**

0.3 Example Programs

Several sections of these notes benefit from hands-on practise with the concepts and tools involved. Test code and guided examples are available either as a tarball from wherever you got this .pdf or on Github at
https://github.com/WarwickRSE/SoftwareDevDec2017

0.4 Code Snippets

Throughout these notes, we present snippets of code and pseudocode. Our pseudocode is informal, and based roughly on Fortran. For example

Pseudocode

```
INTEGER i //An integer
REAL r //A real number (precision unspecified)
ARRAY a //An array (usually this would also have real or integer type)
FLAG l //A logical flag, with possible values TRUE or FALSE
FOR i = 0, 100 DO
  PRINT, i //Print numbers from 0 to 100
END
IF (i < 10) THEN

END
```

We also show examples in Fortran, C/C++[1] and Python[2]:

Fortran

```fortran
IMPLICIT NONE
INTEGER :: i
REAL :: r
REAL, DIMENSION(100) :: arr
LOGICAL :: l
DO i = 0, 100
  PRINT*, i  !Print numbers 0 to 100
ENDDO
IF (i < 10) THEN

ENDIF
```

C

```c
int i;
double r;
float[10] arr;
bool l; // C99 or C++ only
for(i = 0; i < 10; i++){
  printf('%d\n', i);   //Print numbers 0 to 100
 }
 if(i < 10){
 }
```

Python

```python
for i in range(0, 100):
    print i #Print numbers 0 to 100
if i < 10:
    pass
```

Sometimes we also show snippets of commands for shell, make, or git. These often contain parts which you should substitute with the relevant text you want to use. These are marked with {}, such as

```
git branch {name}
```

where you should replace "name" with the relevant text.

0.5 Glossaries and Links

We also include a series of glossaries, one per chapter, as well as a general one at the end. Words throughout the text that look like this: source (code) are links to these. Numbers[3] are links to footnotes placed throughout the text.

[1] We don't distinguish these because our examples are only simple, and we rarely use C++ specific features

[2] Mostly Py 2 or 3 agnostic except where stated

[3] Like this

Chapter 1

Introduction to Software Development

1.1 Basic Software Design

> "I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies." – C. A. R. Hoare

Before sitting down at your editor and actually writing code, you should always spend at least a little time on actually designing your program, no matter how simple. In time you will develop a preferred method of doing this, but how doesn't matter nearly as much as having some kind of plan. How detailed your plan needs to be depends on a few simple factors, and a lot of complicated ones. **If your code involves any sensitive data, you really must read up on relevant regulations and protocol**. This includes:

- Personal data
- Credit card or other monetary data
- Protected information (e.g. medical data)
- Safety critical interactions

Otherwise, your design can save you time, heartache and hair, but it needn't be carefully documented.

1.1.1 Structuring Source Code

You should already know how to write code. You know about control structures and about dividing your code into functions so you can reuse chunks. You may have encountered Object-Oriented programming, modules (in Fortran) or namespaces (in C)

and various other aids to structuring your code. At this stage, there are a few general rules you **should** follow, many more that may be useful, and a very few things you **must** do.

Essential Rules

- **Create a working code** - it sounds silly, but the absolute key thing is that you end up with some code that works, does what was intended, and is reasonably dependable, i.e. wont break unexpectedly or in hard to detect ways.
- **Follow your chosen language standard** - we will discuss this in more detail below, but the standards dictate what is valid code and a valid program. For example, in a valid Fortran program, all variable definitions must go at the top of a function. In a valid C program, all functions must have been defined before they are first used (either completely or via a "prototype" giving function arguments and return type)
- **Use some sort of versioning** - people joke about code iterations labelled "my_function_fixed_mike_January.c" and that is clearly silly, but one can do silly things with version control too. The core idea is to have some sort of record of when changes were made (and by whom). If you choose to do daily dated back-ups, and have the discipline to do it, that is fine. On the other hand, systems like git have built in abilities to compare, search and assign blame. See Sec 4.3 for more.
- **Validate your code enough** - checking and testing is vital, but what "enough" means varies. For a one-off small script, you might just run an example case; for large projects you will want to consider formal testing and validation

Useful Guidelines

- **Divide into reasonably sized functions** - in general, functions should do one thing, but in practice that one thing can become complicated. For example, writing data to a file is probably a single action, but need not be short. You may hear absolute rules like *functions should be 10 lines or less*, but you will likely notice that this is rubbish, and can add tangle. There is no line limit, but it is easier to write, and far easier to debug a function if you are not having to think about twenty things at once.
- Lay out equations etc to help a human - all formatting is gone once your code is compiled, or hits the interpreter, so use it to your advantage. You can use line breaks to split terms of a calculation into logical groups, align groups horizontally, use spacing to group and divide terms. You may also wish to keep equations as they appear in books or papers for now, as they can be rearranged later. Alternatively, you can put human-readable equations in comments.
- Document as you go - we'll discuss documenting a bit more, but in general do it as you go, but never before a function's purpose has been worked out. Orphaned comments can be very confusing.

Finally, there are many things that are good guidelines in general programming that are far more subjective for scientific code. A few of these are below:

- Global variables - One often encounters statements like "global variables are evil". There is some truth in the reasoning - functions which modify a global have "side effects" elsewhere in your code. However, consider a program calculating (some physical field quantity). Something that you seem to be passing to "nearly every" function is probably an actual global state, and if it is large you will not want to copy it around. A carefully named and documented global variable is not a bad solution here.

- You Aren't Going to Need It (YAGNI) - In general, create the simplest program to solve your problem. BUT do consider slightly what you'll need from it next week or you'll find yourself writing non-extensible code and doing a lot of work. Don't add every bell and whistle though.

- Do not optimise (at this stage) - similarly, until your code works, don't bother with (most) optimisation. BUT do avoid making choices that can *never* be fast enough.

- Don't repeat yourself (DRY) - this is a good general principle, BUT do not take it to extremes. In general it is better not to copy-and-paste code and then make small edits, but it is easy to find examples where things are subtly different and trying to write one piece to do them all makes it more complex and/or badly performing.

- "There's only one way to do it" from the Zen of Python - this idea sounds nice, but we think it breaks down here, and is something no other language even tries to do. There are many ways to solve most problems, and for the sort of code you'll be writing, your concerns may be slightly unusual and different to the norm. Getting things to work, and to perform adequately is more important than any points of philosophy.

1.1.2 Planning your Code

> I have always found that plans are useless, but planning is indispensable.
> – Dwight D. Eisenhower

Before you start coding in earnest, you will want to have some sort of plan. This need not be written down, nor be in any special form. For example, for a very small script you may need nothing more than your idea of what it should do, and a first guess as to how. For a large piece of code that will take you weeks or months, you probably do want some sort of formal plan.

You may find it helpful to write a flow-chart for your program, usually focusing on the movement of data. Plan out the general sections, for example user interactions, file I/O, your core algorithm. If you are planning complicated programs or want a

standard format to share with others, you may want to consider Unified Modeling Language (UML). This is rather overkill for most purposes, but can be useful for formal diagrams.

Pay particular attention to limitations and ordering, such as

1. Are any program sections dependent on others?
2. Where do you need user input?
3. Which sections are likely to be the crucial ones for testing and optimising?
4. Where does your code link to other code such as libraries?
5. Do any parts require research, study or unfamiliar libraries?

The section quote above is worth keeping in mind. When creating your plan, the final result is not the only goal. For starters, once you actually try to implement your plan there is a fair chance you will have to change it. Your goal is more to survey the field, find out the limitations and problems, work out how much work this is going to be etc.

1.1.3 Language and Implementation Standards

Many programming languages, C and Fortran included, have detailed language standards managed by ISO (the International Organization for Standardization). These dictate what valid code is, and what guarantees a compiler or interpreter must make about how it implements functionality. For example, both Fortran and C (modern variants)/C++ allow one to specify variables as "constants" or "parameters" and these must not be allowed to change. A compiler which allows them to be changed is violating the standard. Python has the PEP guidelines, which cover some language features, but watch out for where they merge into coding style. Similarly, MPI (used for parallel coding), and things like JSON (used for data sharing) define standards which must be obeyed by valid implementations.

It is usually not necessary to actually read these standards, as many textbooks and internet resources will give you a easier to understand version, but that does not mean you don't need to consider them. Code which does not conform to the strict standards is invalid, and may not work on a different compiler. Extensions to the standards exist, but these are subject to change.

As an example, consider this pseudo-code:

```
1
2 FUNCTION increment(INTEGER x)
3       x = x + 1
4       RETURN x
5 END FUNCTION
6
7 FUNCTION add(INTEGER x, INTEGER y)
8       RETURN x + y
```

```
9  END FUNCTION
10
11 BEGIN MAIN
12
13    INTEGER x = 2
14
15    PRINT add(increment(x), x)
16
17 END MAIN
```

and try to work out the value printed. When the add function is called, the compiler packs up the arguments and passes them (by value). If the function's arguments are worked out from left to right (and pass(ed) by reference), then you expect a value of 6 (incremented x plus incremented x), whereas if they go from right to left or are pass(ed) by value, you expect 5 (x plus incremented x). In C/C++ for example the order is not guaranteed, even though your compiler may always give you the same result.[1]

One of the few rules in this guide is **do not write code that violates standards or has undefined behaviour**. Even if it seems harmless now, it can backfire badly. In C the joke goes that undefined behaviour can crash (the best result, as at least you'll know), delete your hard drive contents (unlikely), or cause the implosion of the entire universe (very unlikely[2]). The likely worst case scenario is code which doesn't quite work right. Perhaps it works fine except when the same function is called 3 times in succession. Perhaps a particular value just gives the wrong answer. Perhaps it crashes one time in 10000. Bugs which you can't reliably reproduce, sometimes called Heisenbugs[3], are a nightmare

1.1.4 Selecting Algorithms

Imagine you have a series of objects you need to sort into order. Doing this by hand, for a small number of items, you are likely to proceed by examining each item in turn until you find first the smallest element, then putting that in its place. Now, find the new smallest, and put that in place, and so on. For ten items, this will take up to 10+9+8+... i.e. $n(n+1)/2$ examinations, and 10 moves of items. For a very large number of things, this is approximately equal to $n^2/2$ and so we say the algorithm scales as n^2.[4]

Imagine instead you have a large number of essays that need to be sorted into alphabetical order. What you might do is to run through the pile once, creating 26 piles (assuming English names), one for names starting with each letter of the Alphabet. Then, you take each pile in turn, and subdivide it by second letter, and so on, until

[1] For real C/C++ code to show this problem, increment would need to take a pointer/reference to x.

[2] probably

[3] Named for the Heisenberg uncertainty principle - as you nail down some aspect of the bug, for example where it occurs, some other aspect changes, leaving you no closer to a solution

[4] In formal language, "is big-O of n-squared" or $O(n^2)$. This means that n^2 with some unspecified prefactor is an upper bound on the time required

each pile has only one item (assuming no duplicate names).[5] It is not easy to see, but this has a time requirement proportional to only $n \ln n$ and for large n this is rather quicker.

As a second example, imagine you have your alphabetized essays and now wish to locate a particular student. You might start at the beginning and look until you find them, which in general will require you look at $n/2$ names. A common alternative is to pick the middle of the list and decide if your target name is before or after that name. You then take the relevant half of the pile, and take the middle element and again check for before or after. This requires only $\ln n$ examinations, which grows very slowly as n gets bigger. This is an example of an approach called bisection. **Binary bisection has uses in all sorts of areas, so make sure you understand how it works.**

Searching and sorting are prime examples of well developed, widely available, best practice solutions to a problem, where failing to research can make your code much slower than it needs to be. Slowness is sometimes acceptable, but it limits your code to smaller problems than you may one day wish to work on.

A more serious example involves money calculations. You might initially try storing floating-point, or real, numbers for currency amounts. However, you can quickly run into rounding issues because of how floats work. An easy first "fix" is to instead store integers for number of pennies. As long as your total amounts are never too large this is OK, until you attempt to deal with percentage calculations. An item costing 4.99 with a 10% discount could reasonably cost either 4.50 (10% giving 49.9p, truncated to 49 p before subtraction) or 4.49 (49.9p, rounded). Several films have people collecting all those fractions of pennies for themselves. This problem is particularly obvious with compound interest, where any rounding errors can grow. Banks have strict protocols on how these calculations must be done, and **if protocol matters you must find and follow the proper guidelines**.

There are many very good books on algorithms in general, and on algorithms and techniques for all sorts of fields. Here we just summarise some points to keep in mind when selecting your methods, in the form of inspirational quotes:

- As simple as possible, but no simpler. Don't choose complex methods just for fun, as they do tend to be trickier to understand, implement and test. But don't oversimplify either.

- Don't re-invent the wheel. Take advantage of prior art. However, sometimes the wheel is for a bicycle and you're designing a jumbo-jet, so it won't fit. The converse isn't much better - jumbo-jet wheels are far more complex and costly to maintain than a suitable bicycle wheel.

- Don't pour good money after bad. Sometimes you will make a mis-step and spend time on unproductive routes. Be willing to put that work aside and try again. Don't throw it away though - it may be useful another time.

[5] This is basically a radix or bucket sort although in practise you might do it more like a merge sort.

- Better the devil you know. If you know a technique that will work, but may, for example, be a bit more computationally demanding, it may still be a better choice than trying something completely unfamiliar.

1.1.5 When Not to Create Software

> If you wish to make an apple pie from scratch, you must first invent the universe. – Carl Sagan

An often overlooked consideration is that of when you simply should NOT write code. This is fairly unusual, although it should probably be considered more often that it is in practice. Note first the trivial example - when you need only a simple, single use script for a task. It is probably a waste of effort to generalise beyond your immediate needs. Otherwise, consider not creating something yourself when:

- A tool exists that can do your task well enough. For example, data conversion where a simple Excel import will suffice, or web-page creation where you might be better using a WSIWYG editor rather than writing HTML.
- When a code exists that solves your problem, unless, of course,

  It costs money and you don't want to, or can't pay for it

  It almost solves your problem but you can't extend or alter it to be perfect

  It is proprietary and not licensed for your use
- When the effort would not be balanced by the reward. Sometimes you could create something, BUT it would take time or effort that would not be rewarded, and you would be better taking a different research tack.
- When you lack the expertise. Again, no matter how tempting, if your code will require research-level computer science or algorithm knowledge, you may be better off adapting your research problem to the tools available, as you may not even realise the errors you are making.

1.1.6 Libraries and their Myths

The other time you should not "reinvent the wheel" by writing your own code is when there is a library which can solve your problem. A library written by several people with relevant expertise, that is used by and tested by many users and developers, can often be better than anything you might create yourself. This is especially true of large programs - it is very very unusual to need a custom operating system for example.

Libraries can be very powerful, save you a lot of work and save you from errors. Their benefits are obvious enough that we're instead going to focus here on when actually to use them.

Way back in March 2016 a developer decided to remove his contributions from a large package-management system, NPM. One of these functions, called "left-pad" was

11 lines of code designed to pad a string to a given length. Thousands of projects fell over, because this crucial dependency was now missing. The package was subsequently restored; a few simple commands got projects working again, and the system was changed so packages can never be removed. Using libraries for such "trivial" function can be dangerous, as you increase complexity for little gain.

Remember that every library you use is another thing your code depends on, another thing your users (or you) have to install, and another source of variations outside your control. In one code we work with, there are several released versions of a particular library (MPI) which simply do not work. We are able to check the version and inform the user before failing gracefully in these cases, but as the number of libraries grows this becomes impossible. Imagine you have a code that uses 2 libraries, each of which has 3 common versions. You now have 9 different possible environments that your code may use, and this can get difficult to debug if a user reports a problem. Some versions of some of your libraries may not even be mutually compatible. **The moral here is NOT that you should not use libraries, but to be selective:**

> Absorb what is useful, discard what is useless and add what is specifically your own – Bruce Lee

1.1.7 Scoping a Project

> Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law. – Douglas Hofstadter

Often you may find that your projects grow to fit the time available for them. This is not uncommon, and is the origin of the section quote. Sometimes, though, you will need to get a good estimate of how long you will need long before you start coding. This is one of the trickiest parts of development, and can take years to perfect. Formal systems based on Function Point analysis[6] abound. These use a proxies to try and estimate the size and complexity of a project based on known factors and previous experience. A very simplified application, aimed particularly towards academic code, is given in our estimators at https://warwick.ac.uk/research/rtp/sc/rse/project_estimator.pdf and https://warwick.ac.uk/research/rtp/sc/rse/project_estimatorwexample.pdf (including a worked example). This tries to quantify size and complexity based on inputs, outputs and libraries. If you have a recent project handy, try the sheet out to calibrate your estimates.

1.2 Aside - Programming Paradigms

For our purposes, there are two unrelated ideas we can refer to as "paradigms" or models. Firstly, there are those for source code itself. You may have encountered "object oriented" programming, where one creates model objects representing some physical or conceptual object and gives them data and functions to allow them to act.

[6] e.g. https://en.wikipedia.org/wiki/Function_point

Contrast this with "procedural" programming, where your code is a series of functions which are called directly. Finally, there are the more specialised "functional" style, where one abstracts the idea of the program's "state" and favours "recursion"; and "declarative" programming, where one states what should be done, and the engine (often a database system) decides how to do it.

The other meaning for programming model relates to the design principles. The first formal software design methodology was what is now called waterfall development. Here design proceeds in stages, with each stage being fully and completely determined before the next step is started. As we have mentioned at least once, some types of program have to work correctly because they handle people's data, money or safety. These often benefit from waterfall design: detailed program specifications are written up and confirmed before any code is created; code is then tested against these specifications; and in theory nothing is released which does not meet the standards set. The general alternative school of methods is based around agile development. These come from industry, where the customer tends to change the specifications at will, request new features close to deadlines, and refuse to clarify their needs. In response to this, agile methods only pin-down what will be done in short chunks, often a week at a time, and rely on regularly releasing working but incomplete code. This can work well in academic practice, since it minimises the time spent on formal design, but be careful to pin down enough details to make your code useful.

1.3 How to Create a Program From a Blank Editor

Once you have planned your code, you will find yourself sitting at a blank editor wondering where to start. For very simple programs you may simply want to just start typing, using some combination of the methods in Sec 1.3.3. If you are using a new language, a new technology, or working on a larger piece of code however, you will want to first consider prototypes and trial codes.

1.3.1 Prototypes and the Minimum Useful Unit (MUU)

When designing a physical object, one often starts by building some kind of prototype. The prototype may be smaller than the real object, may use cheaper or simpler to work materials, and may only pretend to deliver certain functions. For example, wooden car body concepts, protoboard or breadboard rats-nest circuit designs. Testing a new paint colour by painting squares on your wall is a very simple prototype.

In software design, there are similarly several varieties of prototype. Many web pages are designed by a designer using an illustration program as layered images without any code at all. The commonest sort you will encounter is probably the functional prototype, where you create a small program to do just some part of your idea, to check how it works, trial your methods (next subsection) etc.

The MUU, for us, means simply the smallest unit of program which is useful (surprise!), removing everything not totally essential. For example, your program may ideally take an input file as an argument, but it is probably useful if it has merely a

hard coded name. You can extend the program later to take a filename. Later again you may add an actual file browser to choose the file. Put another way, the MUU is the target to hit for your program or addition to be a success. That file-browser is no use if it doesn't actually work, and you'd mostly never want to share code which is broken.[7]

For scientific code, the MUU is often first the simplest code that will let you produce papers, and then it is any increment which allows you to produce a new paper. From another perspective it is whatever will satisfy your funder. Note that MUU is not meant as a perjorative, unlike the related Minimum Publishable Unit. Incremental development is powerful and lets you get work done even while you develop your code further.

1.3.2 Trialling Algorithms and Libraries

> You have to learn to walk before you can run. (And sometimes learn to crawl before you can walk).

It is always a good idea to write a simple program with a new tool, before trying to integrate it into your actual code. Even experienced programmers find it difficult to understand complex tools within complex programs. If one learns to play an instrument, the first steps are simply learning to handle it and to play single notes or chords, before combining them in sequences.

Trialling libraries usually means just creating a small program using the basic features you'll need. For example, you may need a high-quality random number generator for your code, but you want it to be repeatable (give the same numbers every time). You may try creating a small program to seed the generator and then print a few random numbers. You run this a few times and make sure that you get the same numbers, and if you have any problems, you can diagnose and fix them in a small piece of code rather than a larger one. This is sometimes called a minimum working example.

Trialling an algorithm is more work, but if you think ahead you can simply insert the prototype code into your main code.[8] You write up your implementation, and test it on some simple data, and check the result. This is also a good time to check the performance of both the algorithm and your version of it, if that is going to be critical. **Do not optimise the algorithm at this early stage, but do not be afraid to throw it away if it will clearly not serve your purpose.**

1.3.3 Getting Down to it

Once you have your plan, you're ready to start writing in your editor. There are many ways to turn plan into code, just as there are many ways to write papers. Eventually, you'll find a favourite, and it is likely to contain elements of the following:

[7] Of course, sometimes you do: even gcc, the C compiler, has shipped with broken parts. Often simply warning your user is enough.

[8] This can be a bad idea: some methods specifically mandate throwing away any prototype and starting again, but that is not necessary in general.

1. Comment first - start writing, in plain language comments, what each piece of code should do. Go around a few times, adding more detail, until you know "exactly" what to code. "Exactly" is in quotes because how exact you need to be depends on what you're writing and your personal preference. You may want each line fully described, or you may be happy with comments such as "Write the data to the file".

2. Files and functions - start by creating the files you'll need, breaking your code into chunks with a shared purpose. For example, you may want a "user interaction" module, or an object representing a physical object. Then fill in the functions signatures you know you need. Then gradually code your "main" routine, creating functions as you need them, but not filling them in. Then start filling in layers of functions.

3. Just do it - for simple scripts, problems you already know how to solve, or perhaps counter intuitively, problems you have no idea how to solve, you can just sit down and start writing, looking up how to do things as required. This has obvious advantages for the first two sorts of problem: for the third attempting to plan in detail is impossible without a lot of reading, and you may feel it is more productive to learn as you go. Remember that you may eventually need to refactor, or even rewrite your code in this approach, so allow time for this.

1.4 Patterns and Red Flags

Coding patterns are "language-independent models of robust, extensible solutions to common problems"[9] Similarly to our **should** designation, patterns describe a common, effective general solution to a problem. Anti-patterns are examples of code which matches our **should not** designation. They tend to be "commonly reinvented but generally bad solutions". While they are not strictly disallowed, e.g. do not violate any standards, they are usually sub optimal. **Do not assume every supposed anti-pattern is bad code** - global variables are often considered an anti-pattern, but sometimes they are a convenient, and obvious solution. **But don't jump to assuming your problem is special** - often anti-patterns really are useful only in odd niche cases.

Some of the red flags to watch out for, in your code and that of others:

- Magic numbers - numeric constants embedded in the source without comment.

  What is 86400?[10] Unexplained magic numbers can get forgotten.

  How many digits of pi will you need? Obvious constants can still need changing and now you have to edit every place it occurs.

[9]The term was coined by the original "bible" of patterns, Design Patterns: Elements of Reusable Object-Oriented Software, often referred to as the Gang of Four book, by software pioneers Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. This focusses on Object-Oriented code and few of its models are useful in scientific code, but you may encounter the book.

[10]The number of seconds in one day

Even 1 can be magic, if it occurs because you have 0 and 1 based arrays

- Global variables - too many, or the wrong ones.

  Actual global state can be a global, but avoid it for local state.

  What if a function goes wrong? Is the global in any sort of valid state? Was it changed? Could it have been only partly written (see also atomicity) and therefore be nonsense?

- The god object/class/function - one thing that tries to do everything.

  Functions should do their one thing well

  But sometimes dividing into functions too harshly only complicates your code or kills performance

- The golden hammer - when all you have is a hammer, everything looks like a nail

  Don't force your favourite language/tool/technique to do everything

  But don't spend time learning a new tool if yours is adequate

- Premature optimisation - optimising your code before it works, or at cost of readability/modifyability/correctness

  Don't optimise before your code works - doing the wrong thing faster is no use

  But don't make things harder for yourself - don't make decisions that make your code hard to make fast enough in the end.

- Multiple return types - in interpreted languages like Python, functions can return any type of value, different types in different circumstances

  Returning a dictionary with varying keys is OK unless you *need* particular keys

  Returning an array (if many results) or a single value (if one result) is OK with caution but it may be more trouble than it's worth

  In general don't return strings in some cases and numbers in others, such as a sqrt function which gives either a number or the string "Bad input". Instead consider returning a struct or class containing both items. You may still make mistakes, but they'll be less weird.

- Reflection - allowing your code to read, modify and write itself[11]

  This is very powerful in the right places, but can also be dangerous

  For example, constructing a string you then execute as code. Are you certain that the string is safe? In particular beware of ever executing user-input (see also Section 1.7.2)

[11] C and Fortran cannot really do this. Python can, as can many scripting languages

Constructing an object "on-the-fly" can be useful, but you must then be careful to only use it cooperatively, as you don't know what members it has

1.5 Practical Design

No plan survives contact with the enemy. - A well known paraphrase of a quote by H. von Moltke

This chapter so far has given a rapid summary of some rules and guidelines and general ideas, and also some of the red flags to beware of when designing and creating code. Most of this is based on our experiences with scientific software. As the section quote notes, no detailed plan survives sitting down and actually writing, which is a major motivation for agile development and its brothers. Even worse, in real situations you often don't have the luxury of using all of the best practices. This section gives a few examples of practical scenarios where some part of our ideals has to be relaxed, and one has to choose which parts to retain.

One useful model for doing this is to label everything with its MoSCoW method specification. This means deciding for each item whether it **must** be included, i.e. without it the project is deemed a failure; **should** be included, i.e. it adds real benefit and is strongly desired; **could** be included, i.e. it would be nice to have; and **wont** be included, i.e. it is not required at this time.

1.5.1 The Full Grant not-a-Problem

The easiest software project to start planning is the one where you have a large grant which will pay for you to create the program itself, and the program is the primary deliverable. All of the principles we have sped through in this chapter can be applied. You have time to formally design the piece, to trial different algorithms to find the optimum, and to document your software too.

1.5.2 The Create a Paper Problem

Sometimes the brief is as simple and general as to write some code in a general area in order to produce some good papers. This makes it hard to formally plan, and is ideally suited to an agile development approach where you plan only in one-week or one-month blocks. This is not the same as not planning at all! You may also be more informal about selecting algorithms, and you may set out to create code intending later to throw it away and write it afresh (refactoring and perhaps also rewriting). This means you don't care much about style: you may patch together your own code with snippets from manuals such as Numerical Recipes[12] **Never relax on standards - undefined behaviour is always bad**

[12] Available in C or Fortran (77 and 90), this is the handbook of numerical techniques for everything from linear equations to Fourier transforms.

1.5.3 The Rapid Development Problem

Code must be written now, either for a conference, a paper, or some other urgent endeavour. Shares elements with Create-a-Paper and Legacy-Necromancy. Find some time to plan regardless, but don't worry too much about style. Allowing time for checking and testing.

1.5.4 The Legacy Necromancy Problem

You have some old code that used to work but needs updating or fixing and are tasked with just making it work. You don't have time for a proper rewrite or refactor, you aren't able to carefully plan the structure and design but are forced to work with what exists. Don't touch the parts that work, and consider starting with writing tests so you can validate as you go.

1.5.5 The I in Team Problem

You are part of some larger team. Somehow you must collaborate to produce a piece of software. You'll need a good plan, focussing on synchronisation, i.e. who delivers what. Consistency of style can be very helpful, so consider a style guide.

1.5.6 The Coy Collaborator Problem

Like the I-in-Team you are part of a larger team, but here some members are unable or unwilling to share their source code and designs. Again you'll need to synchronise your work. Your main problem is going to be integrating your sections with theirs. Focus on guarantees made by closed source sections you may not edit. Document your interfaces.

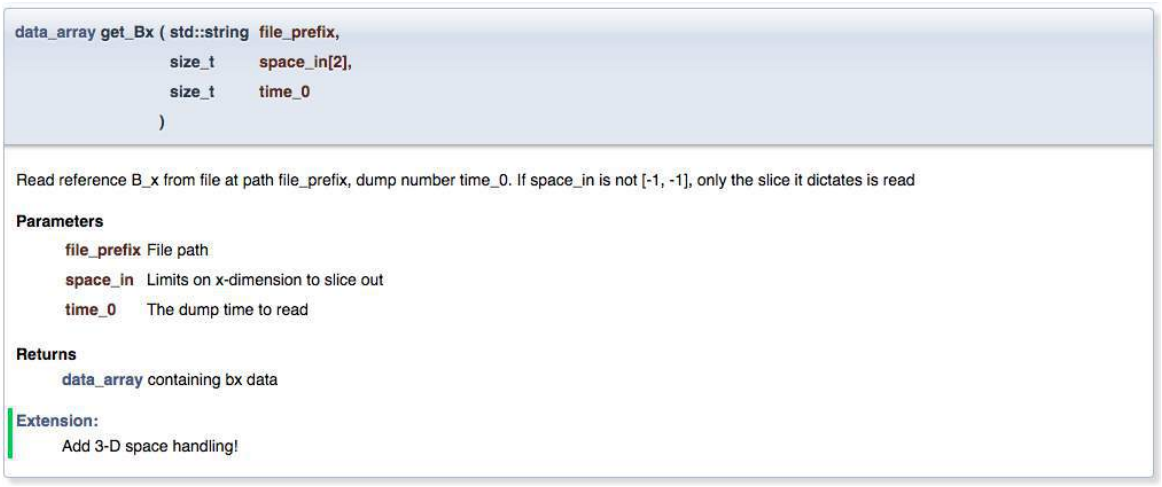
1.6 Documentation Strategies

1.6.1 Tools

Documentation tools exist that can read your source code and produce files that you can display to users or developers, like this (Fig 1.1) example from one of my[13] codes. This is the description of a function called get_Bx which takes 3 parameters. The documentation describes what the function does, what the parameters are, and what is returned. This also includes a note that I would like to extend this function to cover 3-D space.

Most tools have some awareness of language syntax and can identify things like function parameters. They then rely on you to provide a description of what these do etc. They can produce hierarchy graphs if you use classes, and call graphs (what

[13] HR's



data_array get_Bx (std::string file_prefix,
size_t space_in[2],
size_t time_0
)

Read reference B_x from file at path file_prefix, dump number time_0. If space_in is not [-1, -1], only the slice it dictates is read

Parameters

file_prefix File path

space_in Limits on x-dimension to slice out

time_0 The dump time to read

Returns

data_array containing bx data

Extension:

Add 3-D space handling!

Figure 1.1: Documentation produced by an automatic tool (Doxygen 1.8.9.1)

functions call which others). Many can also generate todo lists, like the Extensions list in the image.

Some common tools are listed in A.0.5 for various languages.

1.6.2 Self-documenting code

As you write code, you can take some simple steps to make it far easier and less onerous to document. In particular, you can make it "self-documenting" by naming functions, modules and variables to indicate their purpose.

Consider these pseudo-code snippets (some function signatures omitted for clarity) in order of increasing verbosity, paying particular attention to the comments:

```
MODULE num
FUNCTION m(INTEGER n_v, ARRAY val)
FUNCTION av
FUNCTION gav
//These names are very short and neither descriptive nor memorable. It is hard to tell what things are and what they do
MODULE io
FUNCTION prt(STRING str)
FUNCTION prtf(STRING str, STRING f, FLAG nl)
//Very very short names again. We know 'str' is a string, but what is it for?
//Often function names here differ only by a single character
```

```
MODULE numerical_functions
FUNCTION max( INTEGER n_values, ARRAY values)
FUNCTION average
FUNCTION geometric_average
//These names give some basic information about their functions
MODULE io
FUNCTION print(STRING text)
```

```
8 FUNCTION print_to_file(STRING text, STRING filename, FLAG new_line)
9 //Names are longer and more descriptive. The second string is clearly the
      filename to print into.
10 //The final flag variable is still quite terse, but we know this is some
     indicator about a new line
11 //Note we have no problem with naming the module io. Common acronyms save
      typing
```

```
1 MODULE functions_for_finding_basic_numerical_results_by_jim
2 FUNCTION maximum_value_of_an_array(INTEGER
     int_number_of_values_in_the_array, ARRAY arr_values)
3 FUNCTION average_of_two_numbers
4 FUNCTION geometric_average_of_two_numbers
5 //These names are long and irritating to type. The variable names repeat
     the type information unnecessarily
6 MODULE input_and_output
7 FUNCTION print_text(STRING text_to_print)
8 FUNCTION print_text_to_named_file(STRING text_to_print, STRING
     filename_to_print_to, FLAG print_new_line_at_end_of_text_or_not)
9 //The names are very long again and much information is redundant (both
     print_text_to_named_file and filename_to_print_to tell us the same
     thing)
10 //The flag name has become very long, and tells us what we (should) know
     from the type, that this is a true-or-false flag whether to do X or
     not
```

Self-documenting style does not mean you do not have to document your code. The examples I show above as "good" do not completely describe the functions, and as the last snippet shows, if you try to do this you quickly get too verbose and complicated.

- Think Principle of Least Surprise (PLS) and try to name things to save everybody time and effort, especially youself
- Don't repeat information already given by variable types, such as the logical flag, or the fact that a parameter is passed by reference
- Using common acronyms is a good idea; abbreviations and omitting vowels to save typing is a personal preference or element of your coding standard
- **Avoid ambiguous letters: 0 and O are easily confused, as are I, l and 1**

1.6.3 Hungarian Warts

You may hear references to Hungarian notation, commonly known as Hungarian "warts" on variables. This idea was suggested in the 70s and consists of pre-pending type information to a variable.[14] In its original form this was very useful, as it was intended to supply *additional* information above the given type, in particular its purpose. For

[14]Hence the naming - in Hungarian Family name precedes Given name

example, a string variable might be flagged as *s_name* or as *us_name*, standing for "string" or for "unsafe" string; the latter can be used for e.g. storing unvetted user input and should be handled with caution (see 1.7.2).

Unfortunately, the original Hungarian notation decayed into a tendency to repeat type information at the start of variables, such as *string s_name* or *int i_index*. This is not terribly useful in "explicitly" typed languages where you must specify the type of a variable, although it can be useful in languages like Python where the type is inferred for you. Of course, you must then be careful that your tag matches your type, or confusion and trouble will ensue.

1.6.4 What to Document

Documentation for code serves several very different purposes. Firstly, we distinguish between developer documentation and user documentation. The former is intended for those editing, extending or for library code using, your code. The latter is intended for normal users, who want to provide inputs and get outputs. Orthogonal to these distinctions, we distinguish between interface documentation, i.e. what parameters your functions take, and what they return, and implementation documentation, i.e. descriptions of the internal assumptions, limitations, performance etc of your code. In many cases you do not need to formally distinguish these, and may simply think in terms of the level of detail given. For example you may provide a quick-start guide and a more detailed use guide.
The following snippets are examples of typical combined docs:

Typical Combined Docs

```
int fimbriate ( class nurney overrun, flag decimate)
Routine to fimbriate the excess nurneys. This is used to reduce the
    errors produced by critically low entropy values.

Parameters
   overrun The nurney to process
   decimate Reduce the nurney amplitude by a factor ten before processing
Returns
   Calculated nurney amplitude
-------------------------------------------------------------------------
const double minimum_entropy = 0.707
   Critical entropy below which results become unreliable and fimbriation
        is indicated. This parameter can be tuned to reduce the
       calculation overhead. For best results use a value between -1.2 and
        23
```

These show formal user docs versus developer docs:

Typical User Docs

```
This program calculates a thingy based on the model of Jones and
    Williamsonsson. To compile the code, type
#make
To run the code type
```

```
#hex calculate

'Out of cheese' errors imply you have insufficient ants to run at this time. Try
#hex inititalise BRL
If problems persist, please reinstall Universe.
```

Typical Developer Docs

```
To add your own splines to the code, use
#add_splines(new_spline)
new_spline must implement the
#reticulate()
method and must be guaranteed positive definite. Several sample splines are provided for up to 5th order.
```

And these show formal interface versus implementation:

Typical Interface Docs

```
int fimbriate ( class nurney overrun, flag decimate)
Nurney fimbriation routine

Routine to fimbriate the excess nurneys. Called by reticulate, spline_advance

Parameters
   overrun The nurney to process
   decimate Reduce the nurney amplitude by a factor ten before processing
Returns
   Calculated nurney amplitude
-----------------------------------------------------------------------
const double minimum_entropy = 0.707
   Critical entropy below which results become unreliable and fimbriation is indicated.
```

Typical Implementation Docs

```
int main ( int argc, char * argv[] )
Calculates a thingy based on the model of Jones and Williamsonsson, using reticulated splines to smooth the greebles. If the total entropy is below zero, nurneys may appear.
-----------------------------------------------------------------------
const double minimum_entropy
   Used to avoid underflow when calculating minimum entropy. Must not be below -100 or infinite loops can result
```

1.7 Getting Data in and out of your Code

1.7.1 Input Strategies

Depending on the purpose, environment your code runs in, and flexibility of control you wish to give to users, you can need anything from a single user input to several distinct input paths. For a large code it would not be unusual to have half-a-dozen different means of control. From least to most flexible, with plenty of case dependency:

1. **Hard coded values** - e.g. a log file name
2. **User prompts** - e.g. for an output directory
3. **File-per-control** - e.g. reading a single string filename from a specific filename, or aborting if a particular file is present
4. **Environment variables and/or compiler flags** - e.g. to enable debugging output or control working precision
5. **Command line options** - e.g. specifying problem size or working directory
6. **Simple config files** - e.g. ini files, json files, Fortran name lists or files of key-value pairs
7. **Input control systems** - e.g. ability to specify maths expressions, automatic setup of a problem with given geometry, full GUI (graphical user interface) controls

One code we work on uses 5 different of these for various purposes, and 2 or 3 is not at all uncommon.

Hard coded values have plenty of use for parameters you change only rarely. For example you may write log information into a file called run.log in the specified working directory. Prompting the user should be used sparingly and consider offering an alternative for somebody running your code via a script.[15] Control by file existence is especially useful for causing a long-running code to abort. You simply check every-so-often for the presence of a file, e.g. STOP.

Environment variables share some perils with global variables and should be changed only with caution, but can be very useful for simple global config information. Compiler flags are invaluable for including or excluding debugging code, or selecting a code-path at compile time for performance. They can also be used to select a variable type at compilation (such as float versus double). They do introduce more code paths to test though so should not be overused.

Command line options allow a lot of flexibility and are perfect for programs invoked using a script. For example, when you compile your code, you probably supply the -o argument to specify the output file name. Problem sizes, input-file names and behaviour can all be controlled this way.

[15] They can pipe input to your code, but it is not always easy, especially in Python

A significant downside to all options discussed so far is reproducibility: **you should help your users to preserve the information to reproduce their work in future**. You can, of course, write code to output this information, or write it directly into your normal output files. However, your user still has to extract this and supply it to the new run. Config files are a good way to streamline this process. Many formats exist, from very simple, such as a file containing a series of named values, through block-wise files such as the old Windows .ini format

ini example

```
[control]
  use_float =1

[output]
  rolling_output = true
```

to complex nested structures such as XML or JSON files. JSON libraries exist in many languages and are a very good option in general. If you want to, you can embed either the entire config file, or some signature[16] of the file into the output.

Finally, writing good, robust GUIs is hard and relatively uncommon for scientific codes. In particular, direct control through a GUI is usually a poor choice for a code running on a cluster, as you may have to wait for your job to schedule. A GUI to create an input file can be useful, but is usually a late addition. For very complicated codes you may even consider a scripting interface, for example using Python, Ruby or Lua to setup and run your code. However, these options are out of scope for these notes. Note also that a live interface brings back the issue of parameter preservation unless you carefully output the configuration before running.

1.7.2 Validate your inputs!

> Do it! - Arnold Schwarzenegger

Always validate your inputs. What this means in practice can vary, but in general, **do not trust anything supplied by a user, even if that is you. This is not because users cannot be trusted, but mistakes and ambiguities can happen.** For example, imagine your code starts by deleting any output files in its working directory. You may write something like

```
FUNCTION cleanup_data(STRING relative_wkdir)
  STRING full_path = "./" + relative_wkdir //Assume working directory is
      subdirectory of current directory

  system.execute('rm -rf' + full_path) //Use system delete to remove all
      files in working directory
END FUNCTION
```

[16] e.g. an MD5 hash of the config file, but note that insignificant changes, such as whitespace, will change this

Now imagine your user forgets or fails to supply a directory name to your code. This function will happily delete any and all .dat files in their current directory. Worse, imagine what happens if they give you a directory with a space at the start! If you forget to trim this out, this code will endeavour to delete everything in their current directory and then everything in the working directory! This may sound silly and contrived, but worse things have happened.[17]

As another example, suppose your code is intended to convert from one file format to another, and you ask for two file names from the user, in and out. What happens if they give the same name for both? Will your code cope? What about if you ask for a number dictating an iteration count and the user gives you a negative value? Will you get an infinite loop? What if you ask for a size and the user specifies a very large number? Will your code eat all their memory and crash? Finally what about the classic "SQL injection", as in Fig 1.2 where names have been input and then inserted into a database without input sanitation. A crafty user can instruct your database to delete all its data. The last example is an entire course by itself, but in general: be nice to your users and yourself and check inputs for sanity before proceeding, and always be extra cautious when deleting or overwriting files and data.

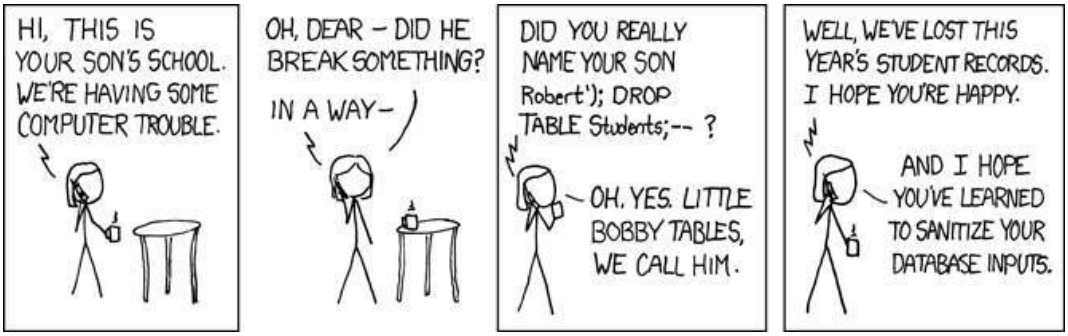


Figure 1.2: SQL injection in practice. Always check your inputs!! Permalink: https://imgs.xkcd.com/comics/exploits_of_a_mom.png Creative Commons Attribution-NonCommercial 2.5 License.

1.7.3 Aside: Input in Parallel Codes

If you are writing code for use in parallel, for example using MPI, it is important to bear in mind file-system locking. On many filesystems only one process at a time may access a file[18] and this means you can produce unexpected bad performance by having every process attempt to. Consider reading the file on one processor only and broadcasting to all others.

[17] If you know any bash scripting you may have realised that the command above is plain silly. It makes no attempt to check what it is deleting, instead relying on just a directory name, and user input validation can only help so much. However, see for example http://store.steampowered.com/news/15512/ 'Fixed a rare bug where Steam could delete user files when failing to start' Unexpectedly empty parameters hurt!

[18] Not true on all systems, and particularly not if the file is opened read-only

If you are reading large chunks of data, for example reading in a restart file (1.7.6) you will want to consider using MPI IO to allow collective reads. Alternatives are reading the entire data set in on one processor and broadcasting it (needs enough memory for the entire set on every processor), or reading in chunks on one processor and sending each chunk in turn to the relevant process (inefficient, one send per processor). MPI-IO allows multiple processes to read files at once and ensures they do not interfere with each other.

1.7.4 Output Data Formats

For data output, you have many choices depending on the size, purpose and lifetime of data files, such as:

1. One file per variable - for example Temperature.dat, Density.dat

2. Fixed format files - for example

```
10/11/2017
10, 20, 13
```

3. Named block files - for example

```
date
10/11/2017
temperatures
10, 20, 13
```

4. JSON, XML or other flexible formats

5. Full data file formats - for example fits files (in Astronomy), HDF5

Other considerations include

1. Do files need to be human readable? Binary files can be half the size of the equivalent text file, but you need to either know their content or use a specific binary format

2. Will the files be kept for long periods? Beware of choosing custom formats that may change or be deprecated

3. Do the files need to be split or combined?

4. Will you share data files with others?

1.7.5 Aside: Output in Parallel Code

In parallel programs you have several options for output. Like with input, but more commonly, only one process can write to a file at once, and you may not want to have each process write in turn. The simplest alternative is called file-per-process and, as it sounds, you write one file on each processor and recombine them in your analysis step. For many processors this becomes unwieldy and for very many processors you can exceed the number of open files allowed by your filesystem.[19] MPI-IO (covered in our Advanced MPI sessions) can be used to perform writes collectively, or you can use one of the parallel aware file IO libraries such as HDF5.

1.7.6 Continuation or Restart Files

Finally, if you are running large programs, or wish to run for very long durations, consider writing code to freeze the state of your program so you can stop and restart a job. This means outputting the current state of all essential variables, probably in double precision, and also of any other state you want to re-initialise. For example, if your program generates random numbers, and you want a stopped-and-restarted job to give identical results to one run continually, you need to output the state of the RNG and re-seed it when you begin again.

1.8 Sharing Code

Note: version control, or preserving your work as you create and edit code is covered in the next chapter. This section talks about preserving working code and sharing code you have used for published work. Chapter 4 talks more about making your software installable and managing packages and libraries.

1.8.1 Preserving Your Work

Once you have some code, and have tested and proven it (see Chapter 2) you will certainly want to use it for your work, and may want to share it with others too. Your primary concern may simply be preservation and reproducibility. Your research council probably has some rules about what you must do (see section 1.8.5), as may any journals you publish in, so first consider this.

If you are simply sharing code with fellow researchers, you may not need any sort of licensing beyond putting your name at the start of your files. This may be the case even if you use other people's packages or libraries, as long as you do not distribute them yourself. Once you start distributing your code online, for example using Github, you will want to consider choosing a proper license.

[19] If you do manage this, the sysadmin will not be happy either!

1.8.2 Basics of Licensing

Once you have decided you need a license, there are many online resources to help you choose, such as https://choosealicense.com/. In general, you want to consider points such as:

- Does my software use other software? What are its terms? Do I include their code in mine?
- Do I want attribution when others use my code? What about if they share it?
- What about if they take it and edit it, perhaps producing their own programs they also share?
- What if they profit from my work, directly or indirectly?
- What if my code affects somebody's work or computer?
- Does my funder require that I share my outputs? Or do they own them?

1.8.3 Resharing Code

Many packages and libraries require some form of integration with your code. For example, if you use a testing library you may need to include some code from it *even if you are not running the tests*. Or you may be using pieces of somebody else's source code. In these cases, you must check whether they put any limitations on resharing, and make sure your license is sufficient. Alternately, you can omit the shared code and give instructions for obtaining and inserting it.

1.8.4 University Resources

Many Universities may provide hosting services for source code, such as a git server or personal FTP filespace.

Some also have contacts to help you choose licenses etc making sure you obey funder regulations.

Finally if your code has commercial value, your University may have resources to help you benefit from this. Warwick offer https://www2.warwick.ac.uk/services/ventures/softwareincubator/

1.8.5 Research Council Expectations

If your work is funded by a research council you must read and obey any rules they have as to sharing your code and its results. For example, many funders require code be made available on request. Many journals require source-code used to create published data be shared. Be careful that you do not find yourself with incompatible restrictions in these cases.

Glossary - Software Development

agile development A (family of) software development method(s) where design is adapted to requirements regularly and no planning is done further ahead than a chosen time, often as little as one week. 9, 13

anti-pattern Patterns for the dark-side: models of code which are inflexible, restrictive, or unreliable common approaches to a problem. *C.f.* pattern, 11

bisection (AKA Binary bisection) Searching for something in an ordered collection (an item in a list, the version of code where something breaks) by repeatedly splitting into two halves, the one containing the target, and the one not, and then repeating the process on the containing half. For example, you have code that adds one to a particular variable in say ten places, you know that it starts at zero, but ends at 9, and you want to know which step is being missed.

0, 1, 2, 3, 4, **5**, 6, 6, 7, 8, 9	Target is above 6th element, value 5
6, 6, **7**, 8, 9	Target is at or below 3rd element, value 7
6, **6**, 7	Target is at or below 2nd element, value 6
6, 6	Target is above 1st element, value 6
6	Length is one, target found

Note that we select "at or below" and "above", and when the length is even, we choose the lower element as the "middle". These are not the only choices, but it is vital to be consistent or you will sometimes get the wrong answer. 6, 39

coding standard A set of rules dictating anything from techniques (some companies forbid pointers), naming conventions, source code layout (2 spaces or 4? Do braces go on a new line?) and every other element of code style. The coding standard may select a language standard but shouldn't cover matters of source code validity. 16

DRY Don't Repeat Yourself, the idea that you should try and reuse code as functions, modules, libraries etc rather than copy paste and edit. 3

input sanitation Removing active code or invalid characters from input. For example, suppose you were to (please don't ever do this without extreme caution!!) take some user input and execute it, as all sorts of online bots do. Now suppose you feed this directly to the system and your user enters "firefox http://my_super_virus_downloader.co.uk". 21

MoSCoW method A prioritization method where requirements are grouped into things that Must be, Should be, Could be and Wont be done, and then treated as such, often in a given iteration of a piece of code. 13

MUU Minimum Useful Unit; the smallest functioning, useful, releasable version of a program or feature. 9, 10

pattern Models of code which are flexible, general purpose, reliable common approaches to a problem. *C.f.* anti-pattern, 11

PLS Principle of Least Surprise; do the least surprising thing in case of ambiguity. 16, 36

prototype A partial product, giving useful information. For example, a look-and-feel prototype for a webpage may be a simple image showing how the page will be arranged. A prototype code may work on only a restricted set of inputs, or produce no output, or use a slower but simpler algorithm. 9

refactor . *see* refactoring, 11

refactoring Changing the content of code without changing its results, c.f. rewriting. For example moving some chunk of code into a function; placing several named variables into a single structure; renaming things for better consistency etc. 13, 26

rewrite . *see* rewriting, 11

rewriting Rewriting differs from refactoring as you may actually change results. Changing to a different inexact algorithm would generally be a rewrite, as would changing from plain-text output files to a special file format. 13, 26

UML Unified Modeling Language; a standard way of creating program description diagrams. 3

waterfall development A (family of) software development method(s) where design and creation steps are fully completed before the next stage can begin, so information flows only downwards. 9

WSIWYG What You See Is What You Get, editors like Word where you see formatting and arrangement as you go, as opposed to e.g. Latex where you must compile your document to see it laid out. 7

YAGNI You Aren't Going to Need It, a feature which may seem cool but isn't actually required right now, and probably never will be. If it ever is, your current version of it probably wont work anyway. 3

Chapter 2

Principles of Testing and Debugging

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. - Brian Kernigham

2.1 What is a bug?

Testing shows the presence, not the absence of bugs - Edsger Dijkstra

The definition of "bug" is "an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways." [Wikipedia (Software bug)] Note that not every example of a program "misbehaving" is a bug, depending on how one defines "unexpected". Sometimes there may be several valid ways of doing something, or the programmer may have chosen a compromise. **Compromises should be covered in code documentation. Make sure to consider this in your own code, as you may not remember your reasons in a few months time.**

As we mentioned earlier, undefined behaviour in C, "system dependent" behaviour in Fortran and in Python the absence of a standard and subsequent platform-dependence, are very important to watch for. What they mean is that your program may work correctly on your machine, but on another system may fail, and not necessarily in any way you might notice. Bugs which you can reproduce are troublesome - bugs which you can't are awful. Not only are you working in the dark to diagnose it, you can never be sure it has gone!

2.2 Bug Catalogue

This section is a rough categorisation of the common sorts of bug you might encounter, with examples and discussion. Not all of these can arise in all languages, and sometimes your compiler may catch the problem for you, especially if you turn on warnings (see Sec 2.5). Languages like Python tend to throw exceptions for many of these errors, so it is important that you do not squash or ignore those.

2.2.1 Logic or algorithm bugs

Logic bugs are a catch-all for when your program does what it ought to, but not what you wanted; effectively you have written the wrong correct program. They can be very tricky to find, because they often arise from some misunderstanding of what you are trying to achieve. You may find it helpful to look over your plan (**you did make a plan, right?**)

Symptoms:

- incorrect answers
- poor performance

Examples:

- Finding the minimum of a list the wrong way: if your data is sorted, the minimum value must be the first in the list but sorting the data to just find the minimum item is silly.[1]
- Missing parts of a range:

```
1 INTEGER a, c
2 a = GET_NEXT_INTEGER()
3 IF ( a < 0 ) THEN
4    c = 1
5 ELSE IF (a > 0 AND a < 5) THEN
6    c = 0
7 ELSE IF ( a >= 5) THEN
8    c = 2
9 END
```

  Notice that the case of a == 0 has been missed, and in this case c is undefined.

- Single branch of sqrt: calculating a from something like $a^2 = 9$, and forgetting that a can be either 3 or -3, introducing a sign error in further calculations
- Most typos: mistyped names, function assignment rather than call (Python), missing semicolon (C). Mis-use of operators, for example using & and && (C).
- The lampposts problem: for n slots, there are n+1 lampposts

2.2.2 Numerics bugs

Floating Point Basics

Computers store numbers in a fixed number of bits, using a binary representation. Usually, floating point numbers are stored as a number and a power of 2 to multiply it by, analogous to scientific notation.[2] Most languages define several types of float, currently

[1] This exact issue showed up in actual proprietary code. Company must remain anonymous

[2] E.g. $1.2 \times 10^3 == 1200$

32 bit and 64 bit being common. For 32-bit floats, 8 bits are used for the exponent, 23 for the significand[3] and one for the sign. This introduces several limitations. Firstly, for large numbers it can be the case that $X + 1 == X$ to the computer[4], because with only 23 bits, some of the number has been truncated. Secondly, they can't exactly represent all decimals, even those which terminate in normal, base-10 representation.[5] And finally, sufficiently large numbers cannot be stored at all, and will overflow.

Rounding Errors and Truncation

You may recall from school that for most calculations it is recommended to do all intermediate steps to as many significant figures as possible, before rounding the final result. In code, any time a number is stored into a variable it is converted to the correct type and any additional information is thrown away. For example in this snippet

```
INTEGER a = 3.0/2.0
PRINT a // Prints 1
```

the value 3.0/2.0 is calculated 1.5 and then stored into the integer a by truncating the non-integer part. Most programming languages truncate rather than round, i.e. they throw away the decimal part. This can lead to odd results in combination with floating point errors, for example the example below where 0.5/0.01 can be slightly less than 50, and so when truncated becomes 49.

Rounding errors are one of the reasons it is a bad idea to use floating point numbers as loop indexes, even if your language allows it. This snippet

```
REAL i
FOR i = 0.0, 1.0, 0.01 DO //Loop stride is 0.01
  PRINT i
END
```

can end up with i slightly less than 1 at the expected end point, and you may then sometimes get an entire extra iteration.

Combining Big and Small Numbers

When you add two numbers, you must first line up the places (like in long-hand arithmetic, matching units to units, tens to tens, hundreds to hundreds and so on), so because of the limited precision, if you add a small number to a large one, you lose precision in the smaller. In most cases, this wont matter much, because it will make only small differences to the final answer. There are two common issues however. Firstly, the order of the terms in the sum may make subtle changes to the final answer. In the worst cases this breaks associativity of addition,[6] because of the rounding at each stage. If this matters, there exist special algorithms for dealing with such sums, to

[3] Number being multiplied, 1.2 in previous note

[4] See machine epsilon

[5] Any number with denominator a power of 2 is exact, all others aren't

[6] Associativity is the property that $a + b + c = (a + b) + c = a + (b + c)$

reduce these rounding and truncation errors. Secondly, if you take two such sums and subtract them, you can get strange cancellation results and unexpected zero or non-zero answers, see the example below.

Overflow and Underflow Bugs

All numeric types have an upper limit on the number they can store. Some languages, such as Python, may hide this but it is important to be aware of. You can look up the limits for your platform and data type. If you try and store a larger number (positive or negative) you will get an overflow error.

For integers, most languages define two kinds, signed and unsigned. Unsigned integers cannot be less than zero, and can therefore be roughly twice as large. In Python, integers will automatically become large-ints if they get too big, which can lead to significant slow-down. In C-like languages, signed integer overflow is well defined and should wrap around to negative values. Unsigned overflow is undefined behaviour although it often seems to work.

According to the IEEE (I triple-E) standards, there are strict standards for floating point operations, in particular which combinations are infinity or **NaN**(see also below). Fig 2.1 shows these. Note that there is a "signed zero" which has to result from some operations, such as $1/-Inf$

Underflow means that a number gets too small (not too-large-and-negative). There is a wrinkle here, which is explored a little in the accompanying exercise, called "denormal" numbers. Usually the base part of the number always has leading bit set, so is a number like 1.*abcde* as in normal scientific notation but base 2.[7] In denormal numbers the exponent is already as small as possible, so the significant is made a pure decimal. This allows the representation of smaller numbers, but at lower precision, because there is a finite number of bits available.[8]

Mixed Mode arithmetic

Mixed-mode arithmetic means using different types of number in an expression. For example in C, Fortran or Python 2[9]

```
1 INTEGER a = 1, b = 2
2 REAL d = 1, e = 2
3 REAL c = a / b
4 PRINT a / b, c, d / e // Prints 0, 0, 0.5
```

a/b is 0.5 exactly, as is d/e. However, in the first case, a and b are integers so the result of their division is converted to an integer, here by truncating (throwing away the fractional part) to give 0. This is the case even when we ask it be stored into a decimal, c. d and e are floating point numbers, so behave as we expect. In complicated

[7] In normalised scientific notation 1.00×10^x and 9.99×10^x are both valid but e.g. 0.10×10^x is not.

[8] Suppose 5 "slots" are available: 1.2345×10^x has more sig.fig than $0.0123 \times 10^{x+3}$

[9] Python 3 promotes all divisions to be float, and uses the special operator // for integer division.

expressions it can be difficult to work out which parts might be integer divisions, so it is a good idea to explicitly convert to floating point to be sure. Note that similar conversions occur if you mix 32 and 64 bit (or any other sizes) of number. The rules can be complicated, so again it is a good idea to keep everything the same or you may get unexpected loss of precision.

What is NaN?

NaN(nan in Python) is a special value indicating that something is Not a Number. **NaN**can result from e.g taking square-root of a negative number, as there is no real valued solution. **NaN**is contagious - any calculation involving a **NaN**will have **NaN**result. However **NaN**has one special property - it is not equal to any other number, including itself, and does not compare to any of them either. Beware though: this is true for any other comparison too, so **NaN**is not less than **NaN**nor greater than it. Figure 2.1 shows which arithmetic operations can result in **NaN**.

Symptoms:

- subtly wrong answers
- severe changes to answers on small edits
- underflow and overflow exceptions/warnings/signals (see Sec 2.5)
- **NaN**, Inf or other special values in results

Examples:

- Non exact numbers: 1.0/33.0 = 0.030303030303030304 Where did that 4 come from? In C, if we ask for 1/33 to 45 dp we get 0.030303030303030303871381079261482227593660355
- The commonest non-bug in gcc: https://gcc.gnu.org/bugs/#nonbugs

```
#include <iostream>
int main()
{
  double a = 0.5;
  double b = 0.01;
  std::cout << (int)(a / b) << std::endl; //Prints either 49 or 50
      depending on optimisation etc
  return 0;
}
```

- In really bad cases this even breaks associativity: 1.0 - (1/33.0 + 2/33.0 + 29/33.0) - 1/33.0 may not give the same answer as 1.0 - (1/33.0 + 2/33.0 + 30/33.0)
- Divide by zero: in Python this is an exception. In gfortran you can set floating-point exception traps using -ffpe-trap etc

Left operand is in the column, right along the top

+	-inf	-1.0	-0.0	0.0	1.0	inf	nan
-inf	-inf	-inf	-inf	-inf	-inf	nan	nan
-1.0	-inf	-2.0	-1.0	-1.0	0.0	inf	nan
-0.0	-inf	-1.0	-0.0	0.0	1.0	inf	nan
0.0	-inf	-1.0	0.0	0.0	1.0	inf	nan
1.0	-inf	0.0	1.0	1.0	2.0	inf	nan
inf	nan	inf	inf	inf	inf	inf	nan
nan	nan	nan	nan	nan	nan	nan	nan

*	-inf	-1.0	-0.0	0.0	1.0	inf	nan
-inf	inf	inf	nan	nan	-inf	-inf	nan
-1.0	inf	1.0	0.0	-0.0	-1.0	-inf	nan
-0.0	nan	0.0	0.0	-0.0	-0.0	nan	nan
0.0	nan	-0.0	-0.0	0.0	0.0	nan	nan
1.0	-inf	-1.0	-0.0	0.0	1.0	inf	nan
inf	-inf	-inf	nan	nan	inf	inf	nan
nan	nan	nan	nan	nan	nan	nan	nan

-	-inf	-1.0	-0.0	0.0	1.0	inf	nan
-inf	nan	-inf	-inf	-inf	-inf	-inf	nan
-1.0	inf	0.0	-1.0	-1.0	-2.0	-inf	nan
-0.0	inf	1.0	0.0	-0.0	-1.0	-inf	nan
0.0	inf	1.0	0.0	0.0	-1.0	-inf	nan
1.0	inf	2.0	1.0	1.0	0.0	-inf	nan
inf	inf	inf	inf	inf	inf	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan

/	-inf	-1.0	-0.0	0.0	1.0	inf	nan
-inf	nan	inf	inf	-inf	-inf	nan	nan
-1.0	0.0	1.0	inf	-inf	-1.0	-0.0	nan
-0.0	0.0	0.0	nan	nan	-0.0	-0.0	nan
0.0	-0.0	-0.0	nan	nan	0.0	0.0	nan
1.0	-0.0	-1.0	-inf	inf	1.0	0.0	nan
inf	nan	-inf	-inf	inf	inf	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan

Figure 2.1: IEEE requirements for floating point number operations

- Sums: Adding a small thing to a large thing
- Small differences of large numbers: (1.0 + 1e-15 + 3e-15) - (1.0 + 4e-15) is not zero in e.g. Python2, but (1.0 + 1e-15 + 3e-15) - (1.0 + 4e-15 + 2e-16) is.
- Integer division: discussed under Mixed-Mode above. Result of 1/2 is different to 1.0/2.0
- In Python all real numbers are "double precision" so run up to about 1.8e308. Integers will automatically grow to hold their content BUT they will get potentially a lot slower when they do. In C or Fortran you specify type, float/double and int/long. Try this:

```
int j=2147483647;
printf("%d\n", j);
printf("%d\n", j+1);
```

  The value j is set to is the maximum possible 64 bit integer, so the +1 "overflows". The reason it becomes negative is because of how integers are represented. Note there are also "unsigned" integers, which overflow to 0.

- Instability: some algorithms work analytically but when numerical effects like rounding are included can become unstable and give a wrong or no answer

2.2.3 Initialisation Bugs

These types of bugs occur when you create a variable and forget or fail to set it (such as in the range-error example in Sec 2.2.1), or when you accidentally create a local variable and think you're working with a global (or vice versa). In C-type languages where you manage memory they can occur when you forget to allocate memory, or when you forget to nullify a pointer and then try and work with it. These latter are more serious as you can accidentally overwrite other parts of your program's data.
Symptoms:

- segmentation (seg) faults or crashing
- different answers run-to-run
- severe changes to behaviour on small edits
- rogue connections between supposedly independent bits of code
- changes with optimisation level

Examples:

- Undefined variable: variable gets whatever value that bit of memory happened to have. Code may give the wrong answer, may vary run to run, may be changed if unrelated code is edited

- Undefined pointer: a pointer gets whatever value the memory had. Usually leads to a segmentation (seg) fault if you try to access it. Always nullify your pointers.
- Unallocated memory: memory isn't allocated, but the pointer or reference to it "seems" OK and doesn't crash. Can overwrite other parts of your program's data etc and lead to anything from an infinite loop (if you accidentally clobber your loop index) to wrong answers, crashing at seemingly unrelated times, etc

2.2.4 Memory Bugs

Note: overlaps with Sec 2.2.3

These types of bugs occur when try and create very large objects, or very many objects, when you try to access beyond the range of an object such as an array, when you try to use an unitialized or already freed pointer in languages where you manage your own memory or lifetimes, e.g. C or Fortran, or when you rapidly create and free objects in a language with a garbage collector. A similar sort of error exists where you use resources like file identifiers.

Symptoms:

- hanging (if you try to obtain too much of a resource)
- changes to wrong data
- crashes (usually only if you write to completely invalid memory, or exceed available memory)

Examples:

- Off-by-1 error in array access:

```
1 ARRAY, REAL(100) arr
2 arr[0:100] = 1
```

  Remember that different languages use 0 or 1 for the first element of an array.

- Buffer overrun errors: (C strings, C++ if you use C-style strings etc)

```
1   char name[10] = "Abcdefghij"; // No space for null terminator,
        string is now invalid
2   printf("%s", name); //See string content and then some junk
```

- Using a freed pointer: When objects are freed, their memory is marked free, but is not changed until it is allocated to something else.[10] If you are lucky, use of a freed pointer will segfault or show up fast. If you are unlucky, it will work perfectly as the "ghost" of the data is still there, and show up only as subtle bugs. **Always nullify pointers after freeing the memory they point to**

[10] This may be familiar if you have ever accidentally deleted a file - often the file is still there on the disk and can be restored, but any new files written may overwrite it.

- Severely bad writes (out of program arena): common error in C, possible in Fortran but uncommon, can occur when you attempt to allocate a very very large memory block. For extra fun these can crash tools like Valgrind (later) when you try to diagnose.

- Mis-allocation of memory:

```
int * start = malloc(1024*1024*1024*3); //Exceeds MAX_INT, actually
    attempts to allocate a negative number
```

  Note that your compiler is unlikely to warn you about this, even if it detects the overflow

2.2.5 Speed Bugs

Slowness is not a bug in itself, but it can be symptom of various logic-type bugs. We include these here as they have many of the same characteristics. The commonest cause is repeating work unnecessarily, for instance inside a loop.
Examples:

- File IO

```
INTEGER i
ARRAY data[100] = GET_NAMES()
FILE output_file
FOR i = 0, 99 DO
  OPENW output_file //Open file for writing
   data[i] = TO_UPPER_CASE(data[i])
  WRITE output_file, data[i]
  CLOSE output_file
END
```

- Multiple loops where one would do, and/or loops where array ops would do

```
INTEGER i
ARRAY data[100] = GET_INTEGERS()
FOR i = 0, 99 DO
   data[i] = data[i] + 1
END
FOR i = 0, 99 DO
   data[i] = data[i] *2
END
```

  Usually the overhead of the loop is not much, but in these cases with very little work to do, it can be significant. These examples could also be replaced with a single array operation (in Fortran, Python, C++) which can be much more efficient.

- Wrong algorithms (see also Sec 2.2.1)

- Not breaking early:

```
1 INTEGER i
2 ARRAY data[100] = GET_NAMES()
3 FLAG found = FALSE
4 FOR i = 0, 99 DO
5   IF (data[i] == "Bob") THEN
6     found = TRUE
7     //Bob found
8   END
9 END
```

  Since the only purpose of this loop is to identify whether the given value exists in the data, we may as well BREAK on line 7, as continuing the loop is needless work.[11]

2.3 Non Bugs or "Why doesn't it just..."

We mentioned earlier the Principle of Least Surprise (PLS). Ideally, code and tools should work in the way that most people expect. This is a great ideal, but you still may find yourself saying something like "why doesn't it just..." or "Why doesn't it know that..." There is also a classic programmer joke phrase, "it's not a bug, it's a feature". Sometimes this is true, and something that looks like a bug is intentional, often as a result of a choice of compromises. Occasionally, a bug turns into a feature, such as Post-It notes, which were an experiment in strong glue, but produced instead a valuable removable glue. Another example of a wildly successful accident is the original ARM processor, which wasn't designed to be low power, but during testing it was found that the power circuits weren't working, and the chip was running on a fraction of a watt leaking from other circuits.[12]

Whenever you encounter the "why doesn't it" it is worth thinking about the answer, as this can teach you a lot. As an example, you might think an over-zealous virus scanner is "malfunctioning" when it detects a program you wrote as potential malware, but how can you identify malicious code? In particular, is there a way to know what answer a program will give for arbitrary input without running it? When we discuss testing this is worth keeping in mind.

There's also several syntactic examples, where you wonder why the computer can't tell what you mean. Some of these are truly unambiguous but just incompatible with how most languages work. For example, it is tempting to wonder why $2 < a < 20$ is not a valid comparison, because surely the computer "should be able" to work out what is meant. Similarly why doesn't $a, b = 10$ set both a and b to 10. In C the "comma

[11] In some security critical code, early breaking introduces the potential for a "timing attack" where one gets information based on how long code takes to execute. For instance, in the example here, we might be able to guess whether "Bob" is definitely in the list, because those cases would stop slightly quicker.

[12] e.g. https://www.epo.org/learning-events/european-inventor/finalists/2013/wilson/feature.html

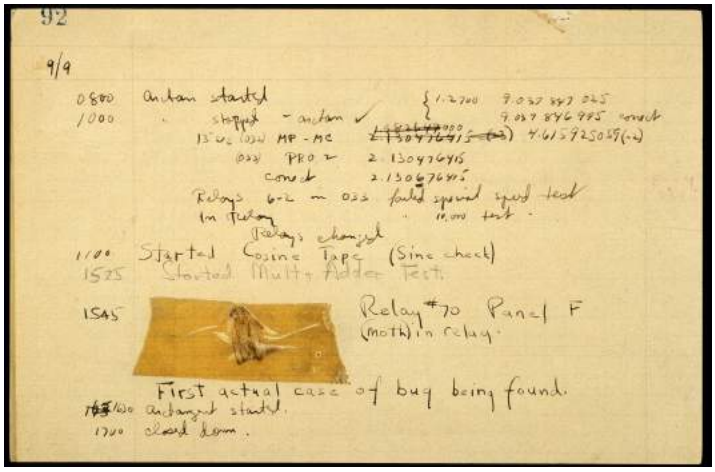


Figure 2.2: The first actual computer bug? http://americanhistory.si.edu/collections/search/object/nmah_334663

operator" in effect returns its second argument, so you get b=10 and a is untouched. In Python, a,b is interpreted as a tuple and the interpreter looks for two values on the right and doesn't find them, giving an error. In Fortran, a,b is simply invalid.

Ultimately, most languages are trying to be consistent, to avoid context where it is not needed[13], and to be able to do very complicated things. Sometimes this does make simple things hard. **But if you find yourself wondering "but why" very often, consider whether you have got the right tool for the job.**

2.4 Aside - History of the Bug and Debugging

There are several theories about the origin of the term *bug* for unintended behaviour in programs, but it certainly dates back to the 19th Century or before. It had entered common computer parlance by 1947, when "engineers working on the Mark II computer at Harvard University found a moth stuck in one of the components. They taped the insect in their logbook and labeled it "first actual case of bug being found."[14] Figure 2.2 is an image of this entry.

Even further back is this quote from Babbage:

> On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question. - Charles Babbage

This sounds silly, but you may be surprised how often a "bug" turns out to be almost exactly this but in reverse. You are *certain* that the code is correct, and the input data is correct, and yet the wrong answer comes out, again and again. To paraphrase

[13] I.e. wherever possible a chunk of code has the same meaning regardless of the surrounding code.
[14] Quoted from http://americanhistory.si.edu/collections/search/object/nmah_334663

Sherlock Holmes **when you have eliminated the impossible and still not found the answer, perhaps it wasn't as impossible as all that.** Keep this in mind when debugging. Computers are very complex, but they are also stupid. They do as they are instructed. Sometimes this is not what you *thought* you had instructed them to do, and very, very occasionally it is truly incorrect, but **your aim is usually to find where what you *thought* would happen diverges from what *actually* happens**.

Remember, when debugging, you are in good company. The founders of computing wrote buggy code, and some of the key developments in early days was learning how to test, validate and fix it, as for example

> As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs. – Maurice Wilkes 1949

or the wonderfully pithy

> There are two ways to write error-free programs; only the third one works. – Alan Perlis, "Epigrams on Programming"

2.5 Your Compiler (or Interpreter) Wants to Help You

A lot of work goes into writing a compiler or interpreter to be correct and efficient. Yet more work goes into making them user friendly. By default, most compilers are unintrusive, telling you only about actual errors, i.e. code which they cannot understand. Many of these may feel little better than "something went wrong", but as you gain experience they tell you more and more. Even better, most compilers allow you to turn on many layers of warnings about code which has been understood, but is a common source of errors.

For example, an old trick in C is to do the following:

```
1 int x = 10;
2 if ( 10 == x ){
3   x++;
4 }
```

Can you guess why? It is not uncommon to mistakenly type only one "=" sign inside the if. If this happens, then x = 10 is perfectly valid and will compile and surprise you by always executing the body of the if. Swapping the order makes this invalid. Note that this trick is no longer needed, as **compilers will warn you about bugs like accidental assignment instead of comparison if you let them**.

2.6 Basic Debugging

> The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.– Brian Kernighan (1979)

Sometimes referred to as Tracing, Old-School debugging or a host of mildly derogatory names, print debugging is the method of placing output statements at various places in your code to work out where an error occurs and what it is. Many people consider this technique antiquated. We disagree. The quote above is old, but still holds. **The fanciest debugger in the world will not help you if you are not *thinking***.

Further, if you are running code on a machine you do not control, so cannot install software onto, or on a cluster on many processors, or non-interactively, it can be the only feasible method you have available, so it is worth knowing even if you come to favour the symbolic debugger.

Finally, print statement debugging can be effective and fast when you have either no idea where a problem arises and wish to narrow it down, or when you know roughly where and what the problem must be and just need to confirm your suspicions.

The method is simple:

1. Turn on all compiler errors and warnings, and recompile/run your code. If this stops the error occurring, see Sec 2.6.3. Otherwise continue these steps.

2. Turn on additional error trapping if your compiler/interpreter offers it. gfortran for example has -ffpe-trap to find divide-by-zero etc. These are not on by default because they can be slow. Fix any errors this turns up.

3. Place some output statements roughly throughout the troublesome code, or your main code if you have no idea where to start

   Simple prints allow you to check whether code is reached or executed

   Printing values of variables helps you check for undefined or wrong data

   If your code exists with an error you may get a backtrace. This usually tells you the line where the failure occurs and gives you a starting point for the problem.

4. Begin to drill down to the source of the problem, placing output statements more tightly around the problem and running again

   The fastest way to find the problem is probably bisection again.[15] There's no rule for deciding what "half" means though: sometimes it is a number of lines of code, sometimes it is some kind of "functional unit"

5. **Stop and think**

   What do you know?

[15] Remember bisection works on ordered items: here the order is the order the lines of your program run in

What would you benefit from knowing?

What are you assuming, and are you certain it is correct?

6. If no problem can be found, return to step 3. but this time be very careful about what *impossible* means. Remember that some problems can cause errors in apparently unrelated code.

7. If the problem is not evident, repeat from step 4.

2.6.1 Remember to Flush!

Programs generally don't write to file every time you call something like PRINT, printf or cout<<. Instead they write to a *file buffer*, stored in memory. System libraries take care of filling up the buffer, and when enough data is there to be "worth" spinning up a hard-disk, they *flush* to disk. This means if your program crashes, you may not get all your expected prints because they never got flushed.

Flushes usually occur when the file is closed and when the program exits "properly". For debugging, you probably want to insert suitable flushes yourself, either after each print or each block of them. In C you can use `fflush`, in Fortran `FLUSH` and in C++ `std::endl`.

2.6.2 The Rubber Duck

Docendo discimus, (Latin "by teaching, we learn")

Everybody has, at some point, struggled with a problem for hours getting nowhere; giving up and asking for help, you find that as soon as you explain the problem the solution becomes obvious. The importance of the act of explaining is well know, and for tricky programming problems, there is no substitute for talking the thing through with another coder. Often, especially as researchers, there may not be anybody you can do this with. The commonest suggestion is to use a rubber duck.[16] Anything will do, but the key is to put the problem into words, preferably aloud. Explain what is happening and what should be happening. Often you will find that your have the answer before you even finish the question.

2.6.3 Debug Environment

Turning on debugging flags in your compiler (-g) for example, and some of the other debug options your compiler may offer does change the environment your code runs in. For example, some compilers disable optimisation in debugging mode, which can change the layout of your data and code. Some of the errors you may have can be affected by this, particularly memory errors, or undefined behaviour.

There are no easy ways to get around this behaviour. Your simplest option is to compile in normal mode and try and find the error with print statements. This is

[16] https://en.wikipedia.org/wiki/Rubber_duck_debugging

another reason why print-debugging should be in your repertoire, as some sorts of bug just do not occur inside the debugger.

2.6.4 Leaving Logging in Place

Sometimes, for especially troublesome code sections, it can be useful to leave your debugging statements in place for when you need them, but disabling them for normal use. In a compiled language, you can do this with the pre-processor. If you use C you will be familiar with the #include statement and may have seen #ifdef used in include-guards. You can put your debug code inside an #ifdef DEBUG block in C or Fortran,[17] and then they will be used only if you give the definition -DDEBUG to the compiler at compile-time.

In interpreted languages, the best option is to use a custom print statement. For example, you might do this:

```
global _debug = False

def debug_print(text):
    if _debug:
        print text #Python 2, use print(text) in Py3
```

Then to enable debugging you set the global to be True and rerun your code. This can also be done in compiled code, but remember that it does have some cost.

A more flexible option is to use a library designed for logging purposes. The Python logging module is good for this. You designate statements as Errors, Warnings, Debug etc, and then set the logging level. These modules benefit from letting you print to screen or to a file or not at all, from adding dates and source code line info, and from keeping log info separate to information you want to display to a user.

2.7 Assertions and Preconditions

> It's not a bug, it's an undocumented feature

While the quote above is a joke, there is some truth in it: once a shortcoming of code is documented, it is no longer strictly a bug. You may encounter bugs in bug trackers that are labelled "wont fix" or similar. This usually means there is a known bug, but that it will not be fixed. Often these were the result of a design decision which could have been made differently, but cannot now be amended, either at all or for a reasonable time, effort or complexity cost. On the other hand it is very annoying as a user to find that you have wasted time on something that will not work.

In your code, you are likely to make assumptions and approximations, and to have to choose between simpler, easier to implement algorithms and more complex ones. For example, suppose you had to implement code to calculate x^y. The simplest way to do this is by repeated multiplication: $x^n = x * x * ..x$ where n is an integer. If your

[17] Name your file .F90 (capital F) to indicate the preprocessor should run

code only ever needs to calculate positive integer powers, it is quite reasonable, and in fact recommended, to use this simple version. It is faster, simpler, and fit for purpose. However, **you must document that a limitation exists** and **you should consider adding code to check for violation of your limitations**.

Asserts are one way to ensure that you are dealing with types and values that you are able to. This is especially useful in languages like Python where you do not specify types directly. For example:

```
FUNCTION power(REAL x, INTEGER n)
  ASSERT( n > 0 ) //n must be positive
  power = 1
  FOR i = 1, n DO power = power*x
  RETURN power
END

INTEGER n = 3
REAL x = 2.0

PRINT power(x, n) //Prints 4.0
PRINT power(x, -1) //Assertion error
```

In languages that do not have an ASSERT, you can raise or throw an error if your condition is not met, or return with an error value.

```
def power(x, n):
  assert( type(n) is int)
  assert( n > 0 )
  power = 1
  for i in range(1, n): power = power*x
  return power
print power(2.0, 3) //4.0
print power(2.0, 3.0) //Assertion error line 2, n not an int
print power(2.0, -1) //Assertion error line 3, n not +ve
```

Formally, those things which must be true when a function is called are named "preconditions", and those things the function guarantees will be true when it completes are called "postconditions". Together these form the "contract" between the function and the calling code.

As a second example, consider accessing an array. In C simple arrays do not know their size; C++ provides the vector class, a 1-D array that can use checked or unchecked access; In Fortran, array size can be checked, but is not by default; while in Python out-of-bounds access is an error. The following snippet has several possible results, depending on language and compiler options.

```
ARRAY, REAL(100) data = GET_DATA()
PRINT data[101] //May print junk, throw an error, cause seg-fault
```

Remember that each check adds overhead, so performance focused languages tend to have them only as an option, while more general purpose languages make them always apply. Checks like these are the foundations of code testing. You want to know how things behave when given good and bad inputs, and be sure about what is going on.

2.8 Testing Principles

2.8.1 What Does Testing Tell You?

You may hear people say things like "any code that's not tested is wrong". Clearly this is not meant to be read literally, as the code may well be entirely correct, and adding a single test doesn't tell you it is always correct. Like most other pithy phrases, this is quite a simplification.

Consider the power function we used earlier, i.e.

```
1 FUNCTION power(REAL x, INTEGER n)
2   ASSERT( n > 0 ) //n must be positive
3   power = 1
4   FOR i = 1, n DO power = power*x
5   RETURN power
6 END
```

We want to pick some input values that will let us be sure the function works. Clearly we thought about the negative n case, so we should test that. 0 is often special in maths, so we probably want to check x of 0, and check that $n = 0$ gives the expected assertion error. If we look at line 4 we notice that our loop is from 1 to n, which suggests we should also check $n = 1$ (to make sure the loop runs once, not 0 times) and $n = 2$. This is the principle of checking the boundaries. This seems to be a comprehensive set of values, but we do need to check them all.

There is something crucial about what we have done here. We have carefully crafted our test values based on how we know the function works. This is sometimes called "white box" testing (in contrast to a "black box" where the code internals are unseen). "Aha" we now think. We *checked* everything that can go wrong! We know this function works all the time. In this simple example, we were able to do this. In more complex examples it is not so easy to spot all the possible things that can go wrong.

A classic example of fatal software bugs is the Therac-25[18] Therac was a radiation therapy machine, delivering electron-beam and X-ray treatments in the 80s. Previous versions had complex systems of hardware interlocks to ensure the proper shielding plates were in place, and that no unsafe doses could be delivered. Therac-25 removed these in favour of a computer system. Between 1985 and 1987 7 people were exposed to massive radiation overdoses. 2 or 3 died[19] and the others suffered serious and life-changing injuries.

Primarily, the Therac-25 disaster is an example of a terribly wrong solution. The removal of hardware interlocks on such a critical system was risky. Even your kitchen microwave has a hardware switch to turn it off when the door is open. In some senses it is an example of a lack of proper testing, for example, requesting one mode, and then requesting another within 8 seconds, while the system was still moving pieces into

[18](e.g. (*Contain some medical details*) https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/, http://courses.cs.vt.edu/professionalism/Therac_25/Therac_1.html)

[19]One died of the cancer being treated, but would have at least needed major surgery otherwise

place, left the system in an indeterminate state. This could, and should, have been caught. Another bug turned out to be a simple integer overflow.

Finally though, Therac-25 is an example of the real pitfall - only testing for the errors that were thought of, not being able to catch all possible inputs, and assuming that passing the tests meant correctness. To quote "In both of these situations, operator action within very narrow time-frame windows was necessary for the accidents to occur. It is unlikely that software testing will discover all possible errors that involve operator intervention at precise time frames during software operation."[20] The machine was tested, and was used for thousands of hours, and only a handful of incidents occurred. For a not-completely-inaccurate version of the problem, imagine a simple system which can swing a lead plate into position in front of the beam, and also activate the beam on high or on low. The operator selects low mode and no shield plate and your software begins to set up. The operator realises they selected the wrong mode, and changes inputs. The software switches the beam mode, but fails to note that the plate position was also edited. Nothing checks the settings are all consistent. The beam activates on high, with no shield in place. Hardware interlocks activate after a small delay and the system shuts off, but the machine readout calculates dose based on the settings it now has stored and reports insufficient dosage, allowing the operator to press a single key to fire the beam again...

Testing is incredibly powerful, and can catch all sorts of bugs. There are many well developed strategies and theories of how best to do it. **Some form of testing is essential in your software.** But there is also a golden rule you may not see written down often: **running tests tells you that the tests pass, not that the code is correct. You must work hard to make this the case, not suppose it.** with immediate corollary **The tests you really need to run are probably the ones you haven't thought of, because you never realised that could be a problem!**

2.8.2 Testing Terminology

Formal testing of code has a lot of associated jargon. In this section we give some brief summaries of the phrases you're likely to encounter. As usual, this is neither exhaustive nor comprehensive.

Unit Testing

Unit testing is testing each building block of your code in isolation, for example each function. You provide inputs and check the outputs. You may focus on testing the **domain** of functions, i.e. that they behave properly for all parameters, or you may focus on testing their correctness for common parameters, or accuracy across a range. Formally you should provide "mock" versions of all inputs, but in practice there are likely to be certain inputs you want to focus on.

[20] http://courses.cs.vt.edu/professionalism/Therac_25/Therac_4.html quoting Miller (1987)

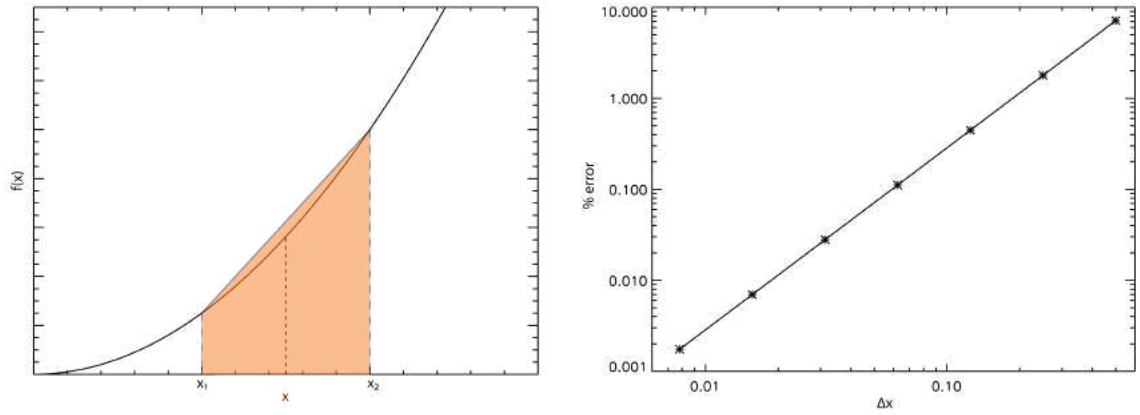


Figure 2.3: A simple integral using the trapezium rule, and the rough scaling of the error with the step size for $x_1 = 0.5, x_2 = 1.0$

Continuous Integration

Continuous Integration in its pure form means merging work from multiple developers regularly, running automatic code building and testing steps, and rejecting code that fails. For large projects with many developers, this ensures that people do not make incompatible changes, and that there is always a working latest version of the code ready to go. For our purposes we're interested only in the automatic-build-and-test element.

Regression Testing

Regression testing is perhaps the easiest sort of test to implement in scientific code. At the basic level you create a few test cases with known answers, either analytically or based on a first working version of your code. Each time you make significant changes, you run the code on these problems and compare the final answers. If the answers differ by *too much* then the test fails. Note that *too much* depends on circumstances. For example, consider a simple integral like this:

$$\int_{x_1}^{x_2} x^2 dx \simeq \sum_n 0.5(x_n^2 + x_{n+1}^2)\Delta x$$

as illustrated in Fig 2.3. We know the analytic solution, so we can work out the percentage error for different step sizes on the interval $[0.5, 1.0]$. You may recall from numerical analysis courses that the error scales as $(\Delta x)^2$. Now suppose your code performs an integral like this. Changes to Δx will change the exact solution you find, but there will be some range of acceptable answers. **Be careful when setting error intervals: absolute values are useful for things like floating point errors, but in most we probably want to use an error percentage** otherwise we have much more stringent constraints on the function $10x^2$ than we do $0.1x^2$**Unless your requirement**

Purpose	Tolerance[22]
64 bit machine precision[23]	$2^{-52} \simeq 2.22 \times 10^{-16}$
32 bit machine precision	$2^{-23} \simeq 1.19 \times 10^{-07}$
Exact calculation equality (64 bit)[24]	10^{-12}
Variant calculation equality (64 bit)[25]	10^{-10}
Numerical calculation equality (64 bit)[26]	10^{-6} to 10^{-8}
Approximation equality [27]	10^{-3} to 10^{-5}

Table 2.1: Typical tolerances for numeric results. The most important rule is to allow tolerances at least 10x smaller than the effect size you are seeking.

is bit-wise exact reproducibility, (page 46) do not compare floats for exact equality in your tests. There are circumstances where code like the following can print FALSE even though all the numbers are exact[21]

```
REAL a = 10.0, b = 5.0, c = 2.0
PRINT b*c == a
```

This can cause trouble if you change to another compiler or machine which behaves subtly differently.

Bitwise Exact Reproducibility

Sometimes, usually for mission-critical code, there is a requirement that changes not only do not change the answer a meaningful amount, but not not change it at all. I.e. the answer is the same in every bit, for every quantity, for all time. This is a very very stringent requirement. It usually means you must specify the compiler in use, down to an exact version, specify which optimisations it is allowed to make, even perhaps specify the processor you run on. Even reordering calculations can break bit-wise exactness. **Avoid bit-wise exactness testing where ever possible. In most cases it is excessive. This means no exact comparisons of floats.**

Test Coverage

Test coverage, or code coverage, is the fraction of code which is included in tests. Ideally all of your code would be tested, but in practise there are often rare combinations of errors that you never expect will occur, and some functions which are difficult or unproductive to test. Note that coverage refers to code, not inputs. 100% coverage

[21] This is quite rare, but can happen when one side of the expression is stored in "extended precision", currently usually 80 bits. The "extra" bits are irrelevant and will be truncated when the result is put back into a normal 64 or 32 bit variable, but at the point of comparison they are not the same.

[22] These rules of thumb are the ones HR uses in general code.

[23] For numbers approximately 1, see machine epsilon

[24] I.e. the same calculation performed with different code

[25] I.e. slightly different analytical calculations

[26] I.e. a good to excellent numerical result

[27] I.e. two different approximate answers

doesn't mean all possible inputs are considered: that might not even be possible. 100% coverage still doesn't guarantee correctness!

Performance Testing

Once your code works, and not before, you may want to consider profiling. Many tools are available to help you find which parts of your code are dominating its run time so you can improve performance. Usually they give you a graph of the percentage of time spent in each function, often in the form of a tree.

2.8.3 Testing As you Develop

Test-driven development refers to a method where you first write tests for a function, and then create the function. Once it satisfies the tests, it is complete. This has several advantages, including that you write the tests without knowing how the function works so are less likely to accidentally write a test that avoids its pathologies.

2.9 Testing for Research

> If you don't know where you are going, you might wind up someplace else. - multiple attributions, probably inspired by Lewis Carroll
>
> Man must shape his tools lest they shape him. - Arthur Miller

It is easy to get sidetracked by testing. There is a wealth of theory, dozens of libraries and frameworks, pretty dashboards and endless statistics. You can devote hours to getting code-coverage up, unit testing every function and building a comprehensive CI workflow. This is a mistake. **Testing is a tool, not a goal in itself. Your goal is writing *correct* software to do research.** You may at some point realise that we are going slightly against the tide here, and not stressing the vitalness of testing as much as many people do. This is true. We think it is most important that your code works, and we are not terribly concerned with how you strive to attain this.

But, testing is an incredibly useful tool for verifying your code against targets such as

1. Correctness
2. Reliability
3. Well-specified domains
4. Reasonable performance

There is nothing as demoralising to a researcher as examining your results and realising that the terribly interesting discovery you thought you had found was an error. This gets even worse if the result is already published. Equally, it is awful to spend hours writing code only to run it and realise it will take weeks to get a result, or to wait days for a job to schedule on a computing cluster only to have it abort with an error.

2.9.1 A Basic Testing Strategy

In our opinion, the optimal approach to testing for research codes is a mix of all the useful elements from various strategies, from the smallest elements of your code to the largest. Roughly then:

1. Unit test the basic functions of your code where practical. For example, test your numerical solvers in isolation. Test your data structures.

   Errors deep in your code are the hardest to find, so aim to be confident in the building-blocks before testing further up.

2. Test the domains of all your basic function too. Make sure you get a reasonable response if you provide bad input, noting that there may not be much you can do except fail. At least try to provide an error a user can understand.

3. Test larger chunks of code in the same way.

4. Run any available analytical test problems and check for a reasonable answer.

   Try to cover all of the basic functionality for a few likely parameters.

5. Test for regressions as you modify your code. Make sure your answers are not changing unduly.

   In particular it is a good idea to keep test problems around that previously broke the code or gave wrong answers.

6. Test on each new target problem against previous published results.

   For real questions there is probably no test problem of use, so you'll have to check against other people's work. Look for general agreement and remember that you are unlikely to reproduce their answers exactly unless you are using the same code and inputs.

As an example, consider the classic introductory programming task of writing a simple scientific calculator from scratch. Suppose we want to type in simple expressions, including basic mathematical functions and get an answer. Assume that we're not allowed to use functions like sin directly, and instead have to calculate the series expansions. Our final testing remit using this strategy may look something like the following. Note the overlaps between tasks and code areas.

- Unit tests, analytical results. We test our maths functions in isolation, making sure we get the right answers for classic known values, such as $\sin \pi$ or $\exp 1$.

- Domain testing. At the same time we check that invalid inputs gives us back a sensible error rather than crashing the program.

- Unit tests. We want to be sure we're taking user input correctly and printing it out correctly too. If we misread input expressions we'll never give the right answers.

- Chunk testing, analytical results. While we could write unit tests for all the bits in our expression parsing code, we might decide to simply test the final result against known correct results.

- Analytical results. Once we've got mostly working code, we try using the program on simple test problems, like the $\sin \pi$ example, but now using the user-interface and parsing code.

- Regression testing. We keep some example inputs around, along with their outputs. We also keep some inputs that caused errors previously to check we don't reintroduce those bugs.

- Real-world results. We might wonder if our calculator can calculate an expression like $\exp(\sin \theta)$ for which we don't know the answer, and have to turn to another calculator for confirmation.

2.9.2 Convergence Testing and Uncertainty Quantification

For research code, in particular numerical calculations, testing does not finish when you have released code. Convergence testing is not part of the traditional software set, but is an absolutely vital process which is often overlooked. Consider back to the integral example in Fig 2.3. How do we know when we have "enough" points to get a good answer? We increase the number of points and at some point the answer *stops changing* when rounded to the precision we want. At this point, we say the integral has *converged*.

For the integral, we know exactly what the *correct* answer is, so we can say absolutely when we are within a given accuracy. We also know absolutely that this integral does converge. If you took any higher maths courses, you may have met the harmonic series https://en.wikipedia.org/wiki/Harmonic_series_(mathematics),

$$\sum_{1}^{n} 1/n = 1 + \frac{1}{2} + \frac{1}{3} + ...$$

Each term is smaller than the last, so the sum looks like it is converging, but it actually grows without limit as n gets large. Thankfully this situation rarely arises in practice, but you must be cautious with convergence if you do not know your algorithms are stable and convergent.[28]

Exactly the same process can be carried out with global parameters of an entire code, such as a grid spacing. **Remember that any two points can be joined by a straight line so you need at least 3 points for convergence**, preferably more. If your quantity of interest seems to be converging, as in Fig 2.4, you can estimate the "true" value and your error.

[28] Further discussion is beyond the scope of these notes, but see any good text on numerical techniques.

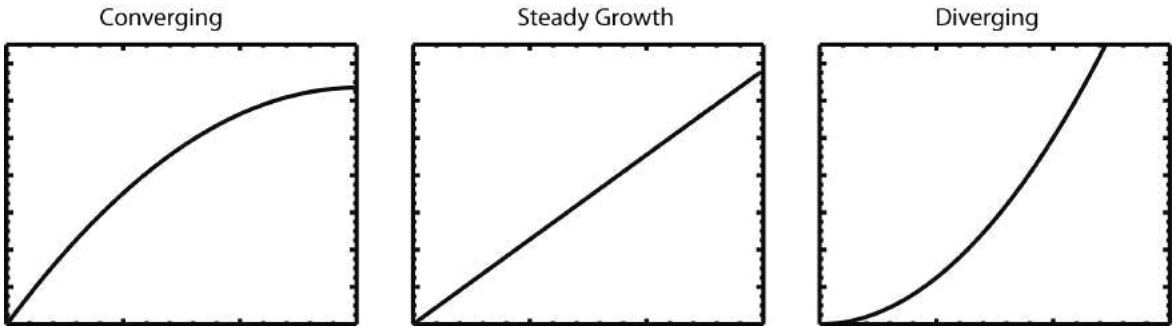


Figure 2.4: Shapes of curves: converging, steady and diverging

Formal correctness of software can be handled by Uncertainty Quantification[29] techniques. This involves rerunning calculations with multiple input values and using this to work out statistically what the expected range of outputs is. For a practical example, when you view weather forecasts predicted by model, it is likely the model was run several times with varying parameters to give an estimate of the likelihood of each outcome.

2.10 Responsibilities

> Extraordinary claims require extraordinary evidence – Carl Sagan, restating many others

As researchers, publishing a paper is effectively saying "these results are true and correct to the best of my knowledge". So if you publish using a code that you wrote, you need to be reasonably confident that it is "correct". How much testing that requires varies. In particular if you're claiming some sort of wonderful new discovery, or presenting results which might have real impact, you really must have a decent set of tests, and check extensively against the current state-of-the-art results. For more ordinary results, you'd test as thoroughly as you would some algebra or calculation, to avoid an embarrassing correction or retraction.

If, or when, you start sharing your code with other researchers, the responsibilities increase somewhat. You'll need to be quite sure your code can't cause chaos (like the clause in nearly every EULA about damage to computer or data, or penalties resulting from incorrect use of software). You also want to be as sure as feasible that the answers are correct. And finally, you now need to consider the possibility of somebody using your code "wrongly". We've noted before that limitations and assumptions ought to be clearly stated, and this is part of why. You'll definitely need some sort of user documentation at this point, covering all of this. You don't want to stumble across a paper and realise its wrong because your algorithm doesn't work in that regime.

Since you have (right?) got a set of tests that validate your code, you can also consider releasing those along with the code as a test-suite. Anybody who has your

[29] See e.g. https://en.wikipedia.org/wiki/Uncertainty_quantification and links therein

code can then run these to confirm there's nothing odd about their setup. Chapter 3 talks a little about testing frameworks which can help you here. The set you release probably isn't the entire set though, as you can set aside those which you only need during development. End-to-end and regression tests are probably the most useful here.

Finally, any published result should be reproducible. This means capturing the state of the code, it's inputs, and any other essential information for every paper. For the first item, version-control systems are ideal, and are discussed in Chapter 4. For the others, see the section on Input and Output in Chapter 1.

To summarise:

- Do "enough" testing, and release useful tests with code
- Document carefully and thoroughly, especially limitations
- Preserve your code to make results reproducible

Glossary - Testing and Debugging

code path Aka control flow path. The path taken through your source code when your program runs. For example, each "if" statement causes a branch into two paths, true and false. The complexity grows exponentially: a second if following the first gives up to 4 paths and a 3rd gives up to 8. The use of "up-to" is because in general many paths will be indistinguishable because the branches are independent. Ideally all code paths should be tested. 19

correct A program that uses all language features that it uses correctly. It does not necessarily give the answer that you expect. 27

floating point numbers Decimals, where the position of the decimal point is allowed to vary. This gives them the same relative precision over a very large range of values. Scientific notation is technically a floating point notation: although you always write a fixed number of decimals, the scaling factor 10^x means the absolute accuracy changes. 28

garbage collector When variables go out of scope the memory they occupy should be made available for reuse. In languages that allow references or pointers to variables, there is a problem with this, as you only want this to happen after the last reference to a given item is lost. This is done by the "garbage collector" in languages which have one. Usually, this runs every so often, or when available memory is getting tight, and cleans up all the now dead values. Note that languages like C don't have this, you must free memory when the last pointer or reference goes. In Fortran you must manually clean up Pointers, but Allocatables are automatically deallocated (since F95) and cleaned up like regular variables. 34, 58

invariant A condition which is always fulfilled. For example, within any for loop (FOR i =0, 10) you know that i is between 0 and 10. Often it is very useful to know that a number is positive, non-zero etc as this allows you use code that relies on this (e.g. if you're passing the number to a sqrt function, or dividing by it respectively).

machine epsilon Floating point numbers are stored in the exponential format $a \times 2^b$. With a limited number of bits, there is a finite step from one number to the next that can be represented. For example, in base-10 with a 5 digit significand (a) we have 1.0000×10^b and the next number we can show is 1.0001×10^b. For $b = 0$ the difference between these is 0.0001. For $b = 4$ the difference is 1. This step-size is called the machine epsilon, and is usually quoted for numbers close to 1. For very large numbers, the step size can be much greater than 1. This is one reason why you should not using floating-point numbers for large integers. 28, 46

minimum working example A small program that demonstrates something with as little extra code as possible. They are very useful when trying out a new library or code, or when reporting a problem or bug. If somebody is trying to help, they wont appreciate wading through reams of code to find the problem, so producing a small program that demonstrates your issue is very useful. 10

no-op Code that does nothing. Stands for no-operation, and can exist for many reasons. Can include incomplete statements, like `a;` in C or conditionals like `if(false)`. 55, 104

overflow A numerical error caused by exceeding the largest number (positive or negative) a type can store. 30

precondition and postcondition Guarantees about the inputs (precondition) or the outputs (postcondition) of a function. E.g. for a sqrt function, you may have to be sure that the input is positive, and the function may promise to return only the positive branch ($\sqrt{}(9)$ is plus OR minus 3).

regression Going backwards in code fitness, e.g. reintroducing a bug which was already fixed, making answer quality worse, breaking a working feature etc. 45

segmentation (seg) fault A severe error in code causing it to attempt to read or write invalid memory. 33, 55

syntax error An error in the sequence of characters in a piece of code. For example, a misspelled name, or a missing bracket, making the piece invalid and unreadable to the computer.

underflow A numerical error caused by attempting to store a number that is too small, often causing it to be rounded to 0. 30

unreachable code Code that never runs. This can be a function that is never called, or a condition that is always true or false, so its body is unreachable. 55, 104

Chapter 3

Tools for Testing and Debugging

Sec 2.2 went through the many types of bugs, with examples and their typical symptoms. Sec 2.6 discussed a basic strategy for debugging using print statements, while Sec 2.8 discussed basic testing strategies. In this Chapter, we focus on some tools to make debugging, profiling and testing quicker, easier, and more reliable.

3.1 ProtoTools

3.1.1 Your Brain

Several of the sorts of bug in the Catalogue in Chapter 2 will not be caught by any tool because they are logical errors. Testing goes some way towards catching these, but can't generally catch everything. **The most vital debugging and testing tool is your own brain.**

In particular, these tools will only give you information about errors. In simple cases this may be enough to find them for you, but not always. Sometimes you get an enormous cascade of errors, just like you do when you forgot to close a bracket early in your program. **Always start at the first error! Often the rest are the same error popping up later on.**

3.1.2 Your Compiler

The next debugging tool you should consider is the compiler (or interpreter) itself. Most offer a host of options for warning you about potential bugs, and flagging up things which may be ambiguous, erroneous, or just redundant. Note that not every warning is *accurate* as just because something is commonly a mistake doesn't mean it always is. On the other hand, a lot of warnings makes it hard to see the relevant ones, so it is often worth fixing them anyway.

Optimisation

Compilers have all sorts of competing demands to consider. Sometimes the main consideration is the time taken for the compilation step. Sometimes it is the size of the resulting program[1] or its speed. Early compilers were comparatively simple, transforming source into executable with some minor adjustments. Computer time was expensive enough that complex compilation was not worth it.

Modern compilers are incredibly complex systems, because they strive to simplify and optimise their output. Since this takes time, most give the option to set the level of optimisation to use. For instance, the default is usually level 0, which makes only minimal changes. Higher levels, commonly 1-3 can do things like inlining functions, move parts of calculation out of a loop, or remove loops entirely, replacing them with copies of the relevant code (loop unrolling).

There is one universal: **the compiler will not reorder *dependent* pieces of code; it can omit unreachable code and no-ops and can remove certain other things.**[2] It is not absolutely the case that compiler optimisations can't change the behaviour of your code, but it is unusual. More commonly you have invoked undefined behaviour and so there is no rule.

Usually you will want to debug at optimisation level 0, which does very little, but sooner or later you will encounter a bug which disappears when you do. This makes it particularly likely that you have forgotten to initialise something, or are doing something ambiguous. This is another reason to avoid undefined behaviour rigorously, because the bug often *goes away when you run the debugger*. Note that some compilers, such as Cray, turn off all optimisation in debug mode, which makes finding this sort of thing very tricky.

3.1.3 Program Output

If your bug occurs during an actual code run, which can happen even with good solid testing and development, your only source of diagnostics may be the program output itself, and whatever your computer or scheduling system gives you. Sometimes this can give you a lot of information, so don't ignore it. Also consider judiciously flushing output to disk. This will reduce performance a little, but used carefully is very useful.

3.2 Symbolic Debuggers

The symbolic debugger runs your code, but allows you to pause and interact with things as it goes. It lets you link the names you gave to variables to their internal representation, and it lets you add breakpoints into the source where it should stop and wait for input.

A lot of what you do with a debugger like this is the same as you would with print statements, with one major advantage for a particular class of errors, segmentation (seg)

[1] For example, programs to run on micro-controllers are very size constrained

[2] For example, tail-recursion, or C++ return-value optimisation

faults. In these cases, the debugger is an easy way to get a backtrace. The debugger also doesn't require you to recompile your code for each new "print", which can be a major advantage. Finally, in complicated cases the debugger lets you print details of memory, and where variables are stored, which can be useful. You can also set variable values to test bug fixes inside the debugger.

3.2.1 GDB for Compiled Code

For compiled codes, the commonly available debugger on Linux-like systems is called gdb. Basic usage is simple: you compile your code, but use the `-g` flag to include debug symbols, and usually turn off optimisation. Then, rather than running your code, you run it inside gdb, using `gdb {my_code}` That starts gdb and leaves you at the gdb prompt, `(gdb)` If you now type `run` your code will run[3], and you will either see it stop and print a backtrace telling you where things went wrong, or you will see a message like `[Inferior 1 (process 35607) exited normally]` Note that most GDB commands have a long form and also a short form (usually the first letter). You can use either or both.

Print-like Debugging

You can use the debugger just like you used print statements earlier on. Have a copy of your source code handy, and compile without optimisation (otherwise the line numbers may not match the code) and with debug symbols. Start the debugger.

Work out where you want to see variable values. Set breakpoints on each line you identify, using `b {linenum}` (short form) or `break {linenum}` (long) Now run the code. It will continue to the first breakpoint and then stop, printing the line to be executed next. You can now print variables by name using `print` or `p` and the variable name. Alternately you can show all local variables with the command `info locals`. Continue to the next breakpoint with `continue`. When the program reaches its end, you can start again by using `run`. In simple cases, this may be all you need to find a problem.

In more complicated cases, like code that loops, or a fault only for some input values, you can use *conditional breakpoints*. These can contain (almost) any valid expression using named variables from your code. For example, you can check for a function argument being > 0 with a line like

```
(gdb) break {linenum} if {argname} > 0
```

You can also set breakpoints to fire on entry to a function by name rather than line-number, such as

```
(gdb) break {my_function} if {condition}
```

You can remove breakpoints with `clear` and the linenumber or function name you used to set them. Alternately, each breakpoint is numbered, so you see output like

[3] On some systems, including OSX, you may be asked for your password to allow gdb to "attach" to a process. This is because you can attach gdb to running processes and use it to inspect their memory, so it does need some privileges.

```
Breakpoint 1, main (argc=1, argv=0x7fff5fbff8c0) at file.c:17
```

The

```
delete {breakpoint_num}
```

command lets you delete by number, which is useful if you have two on one line, for example. Finally, you exit the debugger with `quit`. If the program is still running, you may be prompted whether you really want to exit.

Frames and Traces

Each time your program calls a function, a new "stack-frame" is created in the call stack[4] which contains information on the function, its parameters and in particular where the computer should go to when the function ends. In the debugger you can view the backtrace showing each level of the call stack. For example

```
(gdb) backtrace
#0  push_particles () at src/particles.F90:135
#1  0x00000001000a19be in pic () at src/epoch2d.F90:190
#2  0x00000001000a1caa in main (argc=1, argv=0x7fff5fbffb18) at src/
    epoch2d.F90:35
```

Level 0 is our current level, and in this case we are 2 function calls down from "main" in a function called "push_particles". Note that this shows the line-numbers for each call.

In the debugger we can step back up the frames to see how we got here. For example we may want to examine some variables in the calling function to work out how we got a **NaN**, or a negative value. This uses the `frame {frame_num}` command or the `up {increment}` command which moves us *increment* levels up. `down` moves us back down (towards 0).
Note that if a function was inlined, no stack frame is created, and the function isn't shown in the backtrace. This is one more reason to debug at optimisation O0.

A Typical Session

A typical debugging session might look like this:

```
>>gdb ./eg1
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
15 printf("%d\n", *ptr);
(gdb) break 15
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

[4] A stack is a particular data structure which is "Last In First Out" so elements are taken off in reverse of the order they were put on, like a stack of real objects. Compare a "queue" which, like a real queue is FIFO: the first person to queue up is the first to be served.

```
Breakpoint 1, main (argc=1, argv=0x7fff5fbff8c0) at eg1.c:15
15    printf("%d\n", *ptr);
(gdb) print ptr
$1 = (int *) 0xfffff
```

This value is clearly bad. We might be lucky and the value was set in only one place so we know the error already. If we're less lucky we'll now have to try and trace the error.

Advanced: Attaching to a Running Process

If you work with parallel codes or on a system with a queueing system you may need to be able to "attach" GDB or similar tools to a process that is already running. This is not difficult, but can be fiddly. First, you need to get the process-id or PID (using `top`, `ps` or similar). Then you "attach" to this using either `gdb -p PID` or `gdb attach PID` (see e.g. ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_22.html). Note that GDB needs to be able to find the program executable to work. Then you debug as normal.

3.2.2 PDB for Python

For Python code you can use the standard library debugger which is designed to mimic tools like GDB. Here you invoke debugger and program from inside Python. Breakpoints, printing, stack frames etc are all the same as above. Obviously you don't need to worry about segfaults and such, but you will need to think about exceptions.

3.3 Memory Checking - Valgrind

For languages without a garbage collector you will sometimes need to check for, find or diagnose memory problems.[5] Several tools exist to help with this. Here we're going to discuss valgrind.[6]

Valgrind has several "tools" but the default is memcheck. This runs your program, but not in the normal way. Instead it runs "inside" valgrind which provides its own memory handling libraries and replaces the normal ones with these. It can then track memory allocation, access and deallocation. This does mean programs may run slowly, in some cases quite slowly, and require more memory than normal.[7]

[5] In languages with a garbage collector the problems are not gone but are different

[6] http://valgrind.org/docs/manual/manual.html

[7] From the previous link: "Your program will run much slower (eg. 20 to 30 times) than normal, and use a lot more memory."

3.3.1 Basic Running

The valgrind QuickStart guide[8] gives the basic steps to running your program. Compile as usual, but include the `-g` flag and stick to lower optimisation levels[9] (O0 or O1). Run the program using `valgrind --leak-check=yes {program_name}`. Fix the errors. **Fix earlier errors first. Often later errors are a direct result of earlier ones, so it pays to consider them in order.**

3.3.2 Reading the Final Report

Valgrind always gives a report at the end of the program, and often gives errors and warnings along the way. Typically the final report on a slightly buggy program looks like (the `==PID==` text starting each line is omitted for clarity as is valgrind's advice):

```
1  HEAP SUMMARY:
2      in use at exit: 39,377 bytes in 425 blocks
3    total heap usage: 521 allocs, 96 frees, 46,257 bytes allocated
4
5  200 bytes in 5 blocks are definitely lost in loss record 51 of 80
6     at 0x10000859B: malloc (...)
7     by 0x100000E3B: fill_array (***.c:33)
8     by 0x100000DEC: main (***.c:23)
9
10 LEAK SUMMARY:
11    definitely lost: 200 bytes in 5 blocks
12    indirectly lost: 0 bytes in 0 blocks
13      possibly lost: 0 bytes in 0 blocks
14    still reachable: 4,244 bytes in 4 blocks
15         suppressed: 35,081 bytes in 419 blocks
16
17 ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 15 from 15)
```

The heap is where any dynamically created variables are kept, that can be allocated and freed as your program runs. The first part, lines 1-3 summarises everything this program did with this memory. Next is a report of lost memory, that where a pointer or reference to it no longer exists. This is C code using `malloc` (see line 6) in which I created an array of length 10 (4 bytes per integer, 40 bytes total) 5 times and then lost the pointers. Line 5 tells me this; lines 6-8 tell me (roughly) where it happened.

Lines 10-15 are a report of all possible memory leaks. First, the definitely lost memory, to which we no longer have any pointer. "Indirectly lost" and "possibly lost" are likely to go away once you fix all "definitely lost" memory, so focus on that first. For details see http://valgrind.org/docs/manual/faq.html#faq.deflost. "Still reachable" is all memory in use when your program ends; the program still has references to it, so it could be explicitly freed, but wasn't. It's generally good practise to clean

[8] http://valgrind.org/docs/manual/QuickStart.html

[9] You don't need the program to crash to diagnose it, and optimisation can cause reported line numbers to be wrong. Also, high optimisation can lead to false positive results.

up: everything is freed when your program finishes, but one day you may make changes that turn these into actual leaks. "Suppressed" leaks are described in Sec 3.3.4.

Finally we have the summary of how many errors. Here we have `1 errors from 1 contexts`. Contexts is roughly the number of different errors (type, code location etc) and the number is the total occurrences.

3.3.3 Reading Error Messages

The commonest valgrind messages, and what they mean, are summarised here. *NOTE: valgrind **only** reports errors when a value is used. This is deliberate, to reduce false positives, but can make some errors tricky to track down.* Note there is a `track-origins=yes` which attempts to show you where the problem was created.

An uninitialised value was used

```
1 Use of uninitialised value of size 8
2 ...
3    by 0x1001D3827: printf (...)
4    by 0x100000D97: main (***.c:31)
```

Since we misused this value in a "print" we also get (in C code at least)

```
1 Syscall param write(count) contains uninitialised byte(s)
2 Syscall param write(buf) points to uninitialised byte(s)
```

which tells us the call to the system printing library has a bad count and a bad buffer. This isn't inside our code, but is due to it. In general, you can get a lot of opaque errors if you send uninitialised values into libraries, especially ones like print, so it can be a good idea to alter the code to do something (remember if you're not using the value nothing will be reported) less complex. In this case, the print gives us `715 errors from 84 contexts` as opposed to `8 errors from 2 contexts` with a simpler use.

An if statement, case/switch statement etc depends on something uninitialised

```
1 Conditional jump or move depends on uninitialised value(s)
2    at 0x100000DC1: main (***.c:34)
```

Writing beyond the bounds of an array

```
1 Invalid write of size 4
2    at 0x100000E2C: main (***.c:18)
3  Address 0x1007ffa68 is 0 bytes after a block of size 40 alloc'd
4    at 0x10000859B: malloc (....)
5    by 0x100000DF3: main (***.c:14)
```

Here we wrote an integer (4 bytes) to the wrong memory. The second part tells us where: 0 bytes past the end of a block of size 40 bytes we have allocated. I.e. we have

an array of size 10, and we tried to write to the 11th element.

Reading from memory we've already freed

```
Invalid read of size 4
   at 0x100000EC0: main (***.c:27)
 Address 0x1007ffa54 is 20 bytes inside a block of size 40 free'd
   at 0x1000089DF: free (...)
   by 0x100000E62: main (***.c:22)
```

Here we read from our array after we had called `free` (`DEALLOCATE` in Fortran). The second message tells us this was element 5 (5 * 4 bytes = 20 bytes) of our 10 element array.

Finally there's the one we saw above which tells us that we lost memory

```
200 bytes in 5 blocks are definitely lost in loss record 51 of 80
   at 0x10000859B: malloc (...)
   by 0x100000E3B: fill_array (***.c:33)
   by 0x100000DEC: main (***.c:23)
```

3.3.4 Suppressions

Valgrind, particularly on OSX or when using system libraries or libraries like MPI can find a lot of *false positive* errors. String buffers and padding are a common source of these. Valgrind comes with a default set of errors it should *suppress* because they're not caused by your code, and can't be fixed. You can read more about these at http://valgrind.org/docs/manual/manual-core.html#manual-core.suppress. You can safely ignore the line about suppressed errors in the output. You may sometimes want to hunt for or create a suitable suppression file for a library you're using to separate your important errors from the others.

3.4 Profiling and Profiling Tools

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.** Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. – Donald Knuth

When your code works, and not before, you can consider optimising. Before doing this, you need to know which parts are slow. Often a piece of

code will have only a few bottlenecks, or rate-determining steps, where it spends the majority of calculation time. The quote above is worth reading carefully, as it tells us both critical points. Also keep in mind the 80-20 rule: you often get 80% of the result from 20% of the work, but it will take a lot longer to get the final 20% benefit. Often this is not worth it. Finally though, **make sure to distinguish good design from premature optimisation: don't choose approaches that wont ever be fast enough to be useful.**

3.4.1 Callgrind

Valgrind also provides a basic profiling tool, called callgrind. This tells you how many times each function in your program was called, but note it doesn't tell you about time spent. Example output from a simple test program[10] is something like:

```
214,195,714  Prof_eg.c:solve_array [./prof]
110,000,000  Prof_eg.c:add_to_element [./prof]
  6,568,122  Prof_eg.c:divide_element [./prof]
    325,691  ???:__vfprintf [/usr/lib/system/libsystem_c.dylib]
    281,049  Prof_eg.c:fill_array [./prof]
    210,000  ???:rand [/usr/lib/system/libsystem_c.dylib]
```

This program creates and fills an array with random numbers, then calls a function called `solve_array`. This does a lot of add calls and a smaller number of divide calls. It also does some printing.

In this case there is nothing pathological in our code, although it strongly implies any optimisation effort would start with `add_to_element` as this is called by far the most. However that doesn't always mean it will take longest.

Using callgrind is quite simple. Once again, compile with -g and no optimisation and then run the program with `valgrind --tool=callgrind ./{program_name}` This creates a file called `callgrind.out.{pid}` The pid will be printed when valgrind finishes running, so make a note of it. Now, you run `callgrind_annotate {filename}` on the generated file to get output like above.

You can also have callgrind show you which function called which using `--tree=caller` to get more information. Full details are at http://valgrind.org/docs/manual/cl-manual.html#cl-manual.callgrind_annotate-options

3.4.2 System Tools

While call numbers can be useful, often you want to know the actual runtime used by each function. *NOTE: You also probably want the timings in "release" mode, i.e. whatever optimisation level you run at*

On Linux systems, the gcc toolchain provides a profiler called `gprof` which can do this. Compile your program with `gcc -pg`, run as normal, and then call `gprof {program_name}` We get two useful outputs, first the "flat profile" showing the total time in each function:

[10] DebugTools/C/Prof_eg.c and see 0.3 about obtaining the example code

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 67.87      3.28     3.28    10000   327.82   485.70  solve_array
 29.04      4.68     1.40 1000000000     0.00     0.00  add_to_element
  3.64      4.86     0.18 58908370     0.00     0.00  divide_element
  0.00      4.86     0.00        1     0.00     0.00  fill_array
```

and second the call graph

```
index % time    self  children    called     name
                3.28    1.58   10000/10000       main [2]
[1]     100.0   3.28    1.58   10000         solve_array [1]
                1.40    0.00 1000000000/1000000000     add_to_element [3]
                0.18    0.00 58908370/58908370     divide_element [4]
-----------------------------------------------
                                              <spontaneous>
[2]     100.0   0.00    4.86                  main [2]
                3.28    1.58   10000/10000       solve_array [1]
                0.00    0.00       1/1           fill_array [5]
-----------------------------------------------
                1.40    0.00 1000000000/1000000000     solve_array [1]
[3]      28.9   1.40    0.00 1000000000          add_to_element [3]
-----------------------------------------------
                0.18    0.00 58908370/58908370     solve_array [1]
[4]       3.6   0.18    0.00 58908370          divide_element [4]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[5]       0.0   0.00    0.00       1          fill_array [5]
-----------------------------------------------
```

Comparing index 3 and 4 in this, we see that our addition is called here 1000000000 times for a total of 1.4 s ($\simeq$ 1.4 ns per call) while our division is called only 58908370 times for a total of 0.18 s ($\simeq$ 3.1 ns per call). Note this is without optimisation: in this simple program optimisation implies inlining our add and divide functions for a roughly 50% speedup without us doing anything more.

On OSX the easiest option is often to use Activity Monitor, or rather the OSX sampler. Have your program run in a loop so that you have time to catch it.
Either open Activity Monitor, find your process by name, click the Gear icon 3rd across in the top left of the window, click "Sample Process" and wait.
Or type `sample {process_name}` and wait.
Example output on the same code as above (hex sample ids omitted):

```
Call graph:
    2212 Thread_3743037   DispatchQueue_1: com.apple.main-thread  (serial
       )
      2212 start  (in libdyld.dylib) + 1  [0x7fff8c3b25c9]
        2192 main  (in prof)
        + 1504 solve_array  (in prof)
```

```
6         + 561 solve_array  (in prof)
7         + ! 561 add_to_element  (in prof)
8         + 127 solve_array  (in prof)
9         +   127 divide_element  (in prof)
10        20 main  (in prof)
11          18 printf  (in libsystem_c.dylib)
12          ! 17 vfprintf_l  (in libsystem_c.dylib)
13      .... (more printf system calls here)
```

Times are in ms. Here we see that `add_to_element` takes 0.561 s whereas `divide_element` takes 0.127 s Since we know the former is called a *lot* more often than the latter, we can see that divide is far more costly, so perhaps we have more to gain by improving that.

3.4.3 Overheads

The other major consideration to keep in mind for optimisation is the overheads of various elements of a program. Here we limit to just a few items in cursory form.

Function Calls

As the example above showed quite dramatically, function inlining can give significant speedups. Sometimes however, especially in interpreted languages, this can't be done, and you can't force the computer to do it. Usually the speed cost is irrelevant, but if you divide your program into very many functions, it might start to cost you. Once again, when the program works and not before, consider inlining the worst functions yourself.

Branches and Misses

Particularly in compiled languages, the computer can try to predict what the program will do, and "get ready"[11] to do it. This can speed things up, but when the guess is wrong the benefit is lost. You can try various tools to see the misprediction rates, such as the valgrind cachegrind tool (with `--branchsim=yes`) or the Linux perf tool. In general however, avoid branches inside your critical loops where possible.

Disk, RAM and Cache

Reading and writing to disk is also a lot slower than reading from memory, and as well as RAM, computers have several levels of fast memory cache. You can read about Caching at e.g. https://en.wikipedia.org/wiki/Cache_(computing). For our purposes we just want to note that files are buffered, stored in memory until enough writes have occurred to be worth flushing to disk, so you may notice odd jumps in run time with problem size, and similar issues.

[11] E.g. https://en.wikipedia.org/wiki/Branch_predictor for more details

Useful Numbers

Some numbers to keep in mind, and a fascinating history of them since 1990 are available at https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

3.5 Testing Frameworks

Testing frameworks are libraries to help you write tests for your code, giving some degree of automation to the process. Many are written entirely in the target language, some are external; regardless, your tests must be written in the target language.

3.5.1 Why Frameworks

While you don't need to use a framework, it can save you time and effort, and in particular means no extra code that you yourself have to be responsible for developing and testing. In most cases you'll be using "data-driven" sorts of testing: you have a calculation to run, and a known answer, and you wish to confirm that they match.

Many frameworks have built-in support for comparing floating point numbers (recall Sec2.8.2) Another advantage of a proper framework is that you can often reuse tests you write on different programs, rather than baking them directly into the source. Finally, frameworks deal with the nitty-gritty of reporting results cleanly and usefully, including information about the source code (linenumbers where errors occurred etc).

3.5.2 Which to Choose

There are a plethora of options available in most languages, and a handful in Fortran (see e.g. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks). It is not really practical to go through one in detail here, since they are so different. Instead, we recommend a few options and suggest you look for online tutorials on whichever you choose.

3.5.3 Fortran

In Fortran, the best option is the Nasa created PFUnit. Originally created in 2005, this is still seeing some development. See:
https://en.wikipedia.org/wiki/PFUnit
Docs and link to code: http://pfunit.sourceforge.net/
Tutorial: https://sea.ucar.edu/sites/default/files/pFUnitTutorial.pdf

3.5.4 C

In plain C there are several options, many listed in the wikipedia link above.

We suggest Check

Docs and link to code: https://libcheck.github.io/check/
Tutorial: https://libcheck.github.io/check/doc/check_html/check_3.html

Alternatively you can use CppUnit (see C++ section) with plain C.

3.5.5 C++

In C++ you'll want a framework that can understand classes and inheritance.

CppUnit is a good option and also works with C:
About: https://en.wikipedia.org/wiki/CppUnit
Docs and code: https://freedesktop.org/wiki/Software/cppunit/

Boost.Test might be useful if you're already using Boost
Docs: http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/index.html

3.5.6 Python

Python has several frameworks built into it, including
PyUnit: https://wiki.python.org/moin/PyUnit
Nose: http://nose.readthedocs.io/en/latest/
Doctest: https://docs.python.org/2/library/doctest.html

3.5.7 How to Use Them

Once you've chosen your library, the best approach is to find a smallish piece of code (yours, or something open source) and have a go. Ideally you'll want something with a few functions, perhaps in more than one file. If you don't have anything handy, you can try resources like http://rosettacode.org/wiki/Rosetta_Code for snippets, or pick something small from a site like Github that interests you.

Write tests for the functions, until you think you have covered the likely bugs. Then make some changes to the results, or introduce some bugs, and make sure your tests would have caught them. If they didn't, write a test that does and try again. Make a note of the bugs you missed for future reference, and aim to catch them next time.

Once you're familiar with the library, you can use it in your own code. It can be daunting working out where to start if you have a lot of code to test, but in general:

- Start with at least one end-to-end test, for example a simple test case where you know the answer. This helps you know if you ever break your core code
- Add tests to any new code you produce as you produce it. You are most likely to spot the tricky bits when you write the code as that's when you know best how it works, so that's the time to write the best tests
- Focus on "fragile" areas: those that have broken in the past, those are are a bit tricky or the ones that are regularly modified

- Aim for broad coverage at first rather than exhaustive testing of a few functions
- Keep tests simple. You can't test the test code, so try to make it obviously correct

Remember, you almost never have as much time to test things as you'd like, so focus on the parts that are likely to break!

3.6 Fully Automatic Testing

If you are sharing code that people are using, you don't want to give them something that doesn't work. For example, if somebody downloads your code from a git repository, you want it to at least compile. Chapter 4 discusses workflows that keep a separate "release" and "development" track of code, but that doesn't solve the core problem: once something hits the release track, how can you be sure it actually works.

The solution to this is what's known as continuous integration.[12] The origin is in teams where it is useful to continually integrate work from multiple developers. For our purposes, we're interested only in the automatic testing part of the idea. Here you set up a system which will run tests every time you release code. This can be either every time you push to your repository, or only when you make code ready for release.

Either way, you will need both some tests to run, and an automated runner. If you are using a service like Github, there are various options for CI, detailed at `https://github.com/marketplace/category/continuous-integration`. We suggest looking into `https://travis-ci.org/` which offers free running of public code on Github.

If you're using your own git server, you will have to set up a system yourself. In particular, you will need something to provide test runners and to set up the system so that if the tests fail, the deployment will not go ahead. Check the docs: your system may have information on how to do it.

Glossary - Testing and Debugging 2

backtrace (Aka stacktrace) A readout of the call stack of your program. The call stack, loosely, is the path from where you are in the code back up to the main level. 39, 55, 56, 57

breakpoint A command to the debugger to stop and wait for your input (i.e. to "break" execution). 55, 68

call graph A tree showing which functions call, and are called by, others. 63

call stack The chain of function calls made by your program. A stack is a particular data structure which is "Last In First Out" so elements are taken off in reverse of the order they were put on, like a stack of real objects. Compare a "queue" which, like a real queue is FIFO: the first person to queue up is the first to be served. Each level of the call stack is called a stack frame. 57, 67, 68

[12] Strictly we describe only part of this philosophy here

continuous integration The process of continually combining code from multiple developers, usually with some automatic testing process which disallows contributing code that doesn't work. 67

debug symbols During compilation, the variable and function names you used are replaced with internal symbols to allow the compiler to link together all of their occurrences. For debugging, you can attach extra information, such as the name used in the source, the filename and line number where the definition was made etc. This information is used by symbolic debugger to map between your source code and the actual running executable. 56, 68

entry point (function) The start of a function. In theory, and in some old code, you could jump into your function somewhere other than the start. This is generally accepted to have been a Bad Idea. Note that you can generally return from a function in several places, but be sensible.

inlining To call a function, a program will construct a stack frame, copy (where necessary) variables into place in it, jump to the function code, run the content and then jump back to the calling point, copying the return value into place if needed. To avoid this, the compiler may *inline* a function instead, replacing the function call with the content of the function. 55, 63, 64

profiling Analysing where code spends its time, to identify performance bottlenecks and hotspots for optimization. 47, 54, 103

stack frame A single level of the call stack of your program, each frame contains information on the function called, its parameters and in particular where the computer should go to when the function ends.. 67, 68

symbolic debugger A debugger relying on debug symbols to allow you to run your code and to link it to the source you wrote. For example, you can print the value of a variable by name, even though that name may have been altered (mangled) in the executable. Also, you can set things like breakpoints at specific points in the source code, or you can ask the debugger to continue until the next function call etc. 39, 55, 68

test runner Processes (threads) that can run tests etc. Usually these are idling until they're called upon. Automated testing systems need runners to perform tests. 67

Chapter 4

Workflow and Distribution Tools

4.1 Build Systems

4.1.1 Why Use a System

For small programs, containing only a few files, it is straightforward to simply type the compile command when you need it. For example, you may have a command like

```
gcc test1.c test2.c -o test
gfortran test1.f90 -o test
gcc -pedantic test1.c -o test -lm
```

For a few files, a few libraries and a few compiler flags this is fine. But as the list grows it becomes easy to forget or misspell files, or forget to link a crucial library. It is also irritating to have to recompile everything every time, even though only a few files have changed, and it is impossible to run the build in parallel.

Aside - For Python Programmers

For Python or other interpreted languages there is no build step required. Tools like Make can be used to control your workflow, but there are better options for most purposes. However, you may encounter Make so it is a useful tool to know about.

4.1.2 Build Scripts

The simplest option for automating your code compilation is a script, typically a shell script. This allows you to build with a simple command instead of typing out a long line. You can of course do all sorts of clever things like parameterising the script, so that you can compile and link separately etc, but you can't solve the problems of recompiling everything, nor can you parallelise building.[1] There's also no easy way to tell if you *need* to recompile, since even files which have not changed themselves need re-doing when files they depend on change.

[1] Or rather, this would entail recreating Make yourself

4.1.3 Using Make

In 1976 somebody got fed up of wasting time debugging a program where it turned out the bug had been fixed hours ago but the recompilation was failing. His solution was to create a system to do things automagically, called Make, which is what we're going to discuss here.

Make is the most widely used build tool for compiled languages. The core Make standard is supported by all of the variants, but more advanced features and extensions are specific to a given implementation. Here we are going to restrict to Gnu Make although much of what we introduce is universal.

What Make Does and Doesn't Do

If you have ever built a large code from source, you may be familiar with the classic sequence

```
./configure
make
make install
```

This first configures the build for your particular machine and operating system, checking, for example, the size of an integer, and the availabliity of certain libraries etc, and creates a suitable "Makefile". The next line compiles the code, and the last line installs it, usually by copying the executable into the right place and telling the system where it is.

Make is responsible for the compile step here. A different tool, Gnu Autotools provides the configure step, checking for the name of the compiler and such things. Finally the last line uses Make, but probably only to call some shell or other commands that do the actual installation tasks.

Makefiles

To build codes using Make, you first create a Makefile.[2] In the previous subsection, Autotools was used to do this, but you will probably want to create your files by hand.

The basic idea of Make is to build *targets* using *recipes*, based on the modification state of their *prerequisites*(see also dependency). A Make rule contains several sections:

- target - something to be made. For example a file, such as a .o file produced from a .f90 or .c file, an action, such as the install action above.
- prerequisites - things that need to be made before this target can be. Can be other targets, or can be files.
- recipe - the command to build the target.

[2]The name can be Makefile or makefile, or any other name you wish, although you then have to specify the filename to make.

To tell Make to build a particular target, you type "make [target name]". Make checks all of the target's prerequisites. If any of these need to be rebuilt, it builds them. When this is done, it runs the recipes for the requested target.

Note the obvious property of this: if you were to make a target its own prerequisite you have an infinite loop. **This pattern, circular dependency, must be avoided.** While Make will warn you so nothing bad happens, you definitely wont get the intended result.

Make decides whether a target needs rebuilding based on modification times. For targets and prerequisites which are files, it checks when they last changed on disk, and if this is newer than the target's last modification, the target is rebuilt. For non-file targets there are some subtleties, discussed in Sec 4.1.3.

Rules

Rules are the blocks in the file which look like

```
Target : prerequisites
   recipe
```

The first rule is special and is called the default, and is invoked if you type simply `make`. Otherwise you can invoke a specific rule using `make {target name}`

Recipes

Make recipe lines must be indented using a TAB character. Spaces will not do. If you forget a tab, or use an editor which swaps them to spaces, you will see something like

```
Makefile:6: *** missing separator.  Stop.
```

Note that the "6" is the line number where the error occured.

Intermediate Files and Linking

For more complicated programs that aren't built in a single line, there are distinct "compilation" and "linking" steps. First, each source code file is built into an "object" file and then all the object files, and any libraries you need, are "linked" into the final executable program. Object files usually end in .o, sometimes .obj (especially on windows). Fortran module files are similar (although distinct from Fortran object files). If you use C maths libraries, for example, you may have to use "-lm" in your compile step to link them.

A First Useful Makefile

For a simple C program, a basic makefile is:

```
code : test.o
    cc -o code test.o
test.o : test.c
    cc -c test.c
```

The first, default rule, has one prerequisite, "test.o" and builds the final program. The second says how to build "test.o", in this case from a single file, "test.c".

Even this simple file unpacks to quite a complicated process. When we type `make`, the first rule is invoked by default. This finds that it must first build "test.o". Make jumps to the rule for "test.o". This depends on "test.c", which is not a target, just a file. So Make checks whether "test.o" needs rebuilding, and does so if necessary. The it goes back up to the code rule, and rebuilds this if necessary.

If we run "make" once, and then run it again, we see things like

```
make: 'target' is up to date.
make: Nothing to be done for 'target'.
```

This is great. We know our code has recompiled, and for large codes we don't waste any time rebuilding unnecessarily.

Variables

The simple compilation rules above are useful, but for more interesting tasks you'll want to use variables in your makefile. For example, suppose all your source files are in a directory "src": you could type this in every rule but you risk spelling mistakes, and it becomes a lot of work to rename the directory.

Variables in Make are set and used like

```
variable = value
$(variable)  #Use variable.
```

The dollar indicates this is a variable, while the brackets allow the name to be more than a single character. If you forget the brackets you wont get a helpful error. Instead the line will be interpreted as a single-letter variable followed by the rest of the variable name. **Remember the brackets around Makefile variables.**

Because of the nature of makefile commands, the file is read multiple times, and then any rules are executed, so some things can appear to happen out of order. This allows the system to be very powerful. For the current purposes all we need to worry about is variable assignments. *Makefile variable assignments have one major difference to those in other programming languages.* The normal "=" operator sets two variables to be the same[3], like this

```
var1 = test
var2 = $(var1)
var1 = test2
$(info $(var2)) #Print the variable.  -> test2
```

[3] i.e. var2 becomes a reference to var1

The ":=" operator does the assignment using the value right now[4]

```
var1 = test
var2 := $(var1)
var1 = test2
$(info $(var2)) #Print the variable. -> test
```

Note also that we don't use quotes on the strings here. Make treats quote marks like any other characters, so they would just becomes part of the string.

Make also provides a selection of *automatic variables* for use in rules. These give you access to things like the target being built and the prerequisites. In particular we have

```
    $@ # target of the current rule
    $< #first prerequisite of currrent rule
    $^ # space separated list of all prerequisites
```

For example this snippet shows the use of these by setting the recipe to write to the shell. The "@" character here tells make not to print the command it is about to run.

```
other : final
    @echo $@
final :
    echo $@
test: other final
    @echo $@ '|' $< '|' $^
```

When we run this with "make test" we get

```
echo final
final
other
test | other | other final
```

Notice a few things here. Make looks at the "test" target and sees it must first build other and final. It jumps to the "other" rule. This depends on "final", so it jumps to that rule. Final has no prereqs, so it can start building. It builds final (notice we see both the echo command and its result here). Now it returns to "other": this can now be built. After that, it returns to test. It knows "final" is already built so does not do that a second time. Now it builds test, and we see the automatic variable contents separated by pipes (|).

Implicit Rules

Make has one final internal variable that is very useful, which is the rule wildcard. The "%" stands for any character sequence, but anywhere it appears in the rule it is the same sequence. This is used in what are called implicit rules, for "implicit" targets, i.e. those that aren't specifically mentioned in the makefile. These are also called pattern rules, because they apply to targets matching a pattern.

[4] i.e. var2 becomes a copy of var1

The simplest sort of pattern rule looks like

```
%.o : %.c
    cc -c $<
%.o: %.f90
    gfortran -c $<
```

These rules match any target that looks like "[name].o" and build them from the corresponding "[name].c" or "[name].f90".

A full discussion of pattern rules is at https://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html. Note that Make has a lot of built-in rules for the normal procedure for a given language, summarised at https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html#Catalogue-of-Rules.

Multiple Rules

You may notice that the pattern rules above are perfect for generating a .o from a single .c or .f90, but don't allow you to specify more dependencies for your code. For example, in c %.o would usually depend on %.c and %.h. Or you may have some helper functions in a file which all your other files depend on. Make allows you to define multiple rules for the same target. **Only the last recipe is run, but prerequisites from all rules are considered**. This allows us to use the pattern rules like above, and also, elsewhere in the file, specify our file dependencies. See the example in Sec 4.1.3.[5]

Other Bits

Make has many other features to aid you in processing filenames, putting things in the right directories, and controlling your compilation by passing variables to make itself. We are not going to go into the details here, as there are many good tutorials out there, and you are best served learning the details when you actually need them. However, one last feature of Make is very common and quite important, which is using Make for a target which is not a file.

As we saw above, Make can run any command you wish, and targets need not correspond to a filename. These targets are always rebuilt, as Make has no way of tracking their last modification. A common use of this is to have a "clean" target, which cleans up intermediate and output files. This is fine, unless a file ever exists which is called "clean", when make will think this target is up-to-date and do nothing. To avoid this, and to make things clearer, make has a special target, `.PHONY`. Any prereqs of this special target are assumed to be phony, i.e. not to correspond to any file.

Full Example Makefile

A basic, but useful, Makefile is along the following lines:

[5] You can also write a rule with multiple targets to add some prereqs to them all. This can be useful for things like helper functions which are prereqs of all your code.

```
# Set compiler name
CC = cc

#Set phony targets
.PHONY : clean

#Default rule
code : test.o
    $(CC) -ocode test.o
#Pattern for compiling .c files
%.o : %.c
    $(CC) -c $<

#List the dependencies of test.o here
#It will be built with the rule above
test.o : test.c
#Clean rule
clean:
    @rm -rf code
    @rm -rf *.o
```

Parallel Building

The simple build scripts tend to use a single line to compile everything and then link in a single step. With Make, you instead give it dependencies, so it knows the order in which it needs to build files. This means it can work out which rules can be built simultaneously, and parallelise your building. Using `make -j {n_procs}` Make will use up to n_procs, but only as many as it can. This can speed things up a lot.

Pitfalls

Make is very powerful but it does have some traps that are easy to fall into.

- Forgotten dependencies

  Target wont rebuild when it needs to

  Can be very hard to diagnose

  Your compiler can often generate dependencies (see gcc -M)

- Circular dependencies

  Make will ignore these

  Might create a forgotten dependency

- Permanent rebuilding

  Non-file targets always rebuild

  Any rule that doesn't actually build the target file can too

- Overcomplication

  You can do all sorts of conditional compilation with Make

  See Sec 4.1.4 for tools beyond make

4.1.4 Cmake and Other Tools

As well as Make which we focused on, there are many other tools for building code based on dependencies and rules. Several of these, such as Gnu Autotools and qmake generate Makefiles for you; some use Make as a backend but also support other options, such as cmake, and some have their own system, such as meson. These have their own strengths and weaknesses. You may encounter them when using libraries, for example qmake is made by the developers of the QT GUI libraries, and is almost essential for building QT projects.

Once your makefiles need to account for things like multiple platforms (OSX, Linux flavours etc) or installing their own copies of needed libraries, you will want to look into these tools. We discuss them a little in Section 4.2. However, Make can serve most of your needs for quite a long time.

4.2 Distribution Systems

Sooner or later, you will want to move your code off the computer where it was developed. You may just be moving to a new machine, or onto a cluster to run code, or you may be sharing it with other people. You will want this to be as painless as possible. When you install programs, you probably download a binary executable specifically made for your machine and operating system. This is almost never the case for scientific code, apart from some commercial packages where they wish to keep the source code private.

You may have used a package manager (see also Sec 4.2.2), such as apt, or Homebrew on Mac. These sometimes obtain pre-built executables, but also can obtain and build code from source for you. It is again rare for scientific code to be available through OS packages, although in e.g. Python it is not unusual code to be part of larger repositories.

This means that when you distribute your source code, whether you do this by offering a tarball or via a version control system (see Sec 4.3), you need to consider making it work with multiple operating systems, compilers or interpreter flavours[6] and available libraries. You can restrict many of these things, for example you may flag your code for "Only UNIX like" or "Only Windows" systems, or you may require an installation such as SciPy for it to work.

[6] "Python" generally refers to CPython, a particular variant of Python. Others exist, such as IronPython and PyPy, and not all features work exactly the same in all of them.

4.2.1 Portable Python

Unlike compiled languages, where the language standard is fixed and the compiler takes care of (most) implementation details[7] in Python you may want to actually change your code for portability.

If you are not already, you should consider using Python modules, rather than simple scripts. Details are at https://docs.python.org/2/tutorial/modules.html and https://docs.python.org/3/tutorial/modules.html. Basically you have to put your script(s) in a named directory and then include a "__init__.py" file. This then allows you to use import to access your functions etc.

For more complex programs, you can use a proper packaging system. This packs up your scripts, and allows you to include a listing of libraries you depend on and which must be installed first. Some also provide web-based distribution systems so somebody can obtain your code directly. Full details are beyond scope of these notes, but roughly, the simplest option is to use Python's "distutils", where you create a "setup.py" file which can be used to install your package into the correct place for Python on the user's machine. Details are at https://docs.python.org/3.6/distutils/introduction.html. "setuptools" ("easy_install") adds facilities to include dependencies, and also adds the web download options. "pip" uses distutils/setuptools scripts via a web distribution system.

The *best* option is rather vexed. The page at https://packaging.python.org/discussions/pip-vs-easy-install/ lists a comparison of pip and easy_install. Either will work, although one thing to bear in mind is that both host packages publicly, so if you want to keep your code private, neither will suit you.

4.2.2 Compiled Codes

Package Systems

Although you are unlikely to distribute your own code via the official packaging systems of the various OSs, you may have to support users using them to install dependencies for your code. Various flavours exist, e.g.

- DEB (Debian based packages, for systems include Ubuntu, Mint) https://wiki.debian.org/Packaging/Intro
- RPM (Many non-Debian Linux distros) https://fedoraproject.org/wiki/How_to_create_a_GNU_Hello_RPM_package
- Ports (BSD, OSX) https://www.freebsd.org/doc/en/books/porters-handbook/why-port.html
- Homebrew (OSX) https://brew.sh/

[7] Of course, this is not really the case. For example, Windows and Linux use nearly always requires you write two versions of at least some code. Usually this will be for graphical use though, so if you stick with numerics at the command line you can avoid most hassle.

Simple Cases

For fairly small projects you can simply distribute your source code, using tarballs (zips) or a public repository, and provide a list of dependencies (or a simple script to obtain them) and a simple makefile with a few conditions. For example, you can write a makefile to accomodate different compilers, which expect different flags, by providing arguments in your make file like "make COMPILER=intel". As things get a little more complex you may turn to cmake, or qmake rather than hand-create a complex makefile. A user downloads your code, installs the needed libraries, and builds the code.

Intermediate Cases

In more complex cases, where you may wish to accomodate a lot of details of compilers, or choose between different libraries or even make them optional (for example, many programs that can work with PS files will use GhostScript if available, but have fallbacks otherwise), handwritten makefiles get too complicated. In this case you can turn to something like Autotools (http://inti.sourceforge.net/tutorial/libinti/autotoolsproject.html) to create a Makefile for a particular installation. This can also create an include file for your code giving it access to information like Integer sizes. With Autotools you have the (possibly familiar) sequence

```
1 ./configure --options=xyz
2 make
3 make install
```

Difficult Cases

For more systems even more complex than this, you will have to look at containerisation systems. *Note that the previous setup is enough even for major endeavours such as BLAS or SageMath.* These distribute an entire operating system and the installed software, which requires setting up disk access in and out of the container, network drivers and interconnects in HPC. Options include

- Virtual Machines (many options, Virtual Box etc)
- Docker (https://www.docker.com)
- Singularity (http://singularity.lbl.gov)
- Shifter (https://github.com/NERSC/shifter)

the last two of which are designed with HPC in mind.

4.3 Introduction to Version Control

This section will cover the basics of Version Control Systems (VCSs), including why to use them, and then gives a quick walkthrough of git, and some information about tools such as Github.

4.3.1 What is Version Control?

Version control is also known as source code management (SCM) in context of software, although note that most systems can deal with assets[8] too. Version control systems are designed to record changes you make to code. They track when, and by whom the changes were made, and usually allow you to add some explanation. They allow you to go back to an old version of the code, or of just some files. They also include tools to help you to merge incompatible changes.

4.3.2 Why Use It?

Many reasons:

- "I didn't mean to do that"

  You can go back to before a breaking change

- "What did this code look like when I wrote that?"

  You can go back as far as you want to the version you used for a particular talk or paper

- "How can I work on these different things without them interfering?"

  Branchs let you work on two features independently and only merge them at the end

- "I want a secure copy of my code"

  Most VCSs have some concept of a client and a server, so make it easy to store offsite backups[9]

  Many free services exist online, and you can easily set up your own too

- "How do I work with other people collaboratively?"

  Most modern version control systems include specific tools for working with other people

  Also powerful (often paid for) tools to make it even easier

  Dor collaborative editing of a single file (e.g. papers), there are better options

- "My funder demands it"

  More and more funding bodies expect code to be managed and made available

  Online version control is one way to do this

[8] Such as images and data files

[9] Proper backups should account for the moderately likely failure of your hard drive (i.e. use an external drive) and, for important things, the quite unlikely destruction of your office (i.e. use fully mirrored system like RTPSC desktop home, files.warwick, or a cloud service)

While the basic functions are quite similar in all VCSs the more complex features often differ quite a lot. The terminology often differs too. The most likely system you'll be using is "git"[10], so that is the one we are going to talk about here. Note that it is not the only good option. You're also likely to use some sort of online service, likely "github"[11]. Alternately, the Warwick SCRTP has an online system.[12]

Why NOT Use It?

The most important thing about version control is to do it. It doesn't matter how, as long as it works. If you're working alone, on one thing at a time, and are very conscientious, there is nothing actually wrong with simply keeping a dated copy of your files. In particular, freeze a copy every time you write a paper or run new simulations, and make sure to keep careful offsite backups (see footnote 9). This does now require more effort than using a VCS, although it will suffice for small or one-off projects.

4.3.3 A Brief History

Version control is as old as computers. The US National Archives Records Service kept copies of code on punched cards back in 1959, which managed a data density of about 100MB per forklift pallet. Important programs would be kept in the archives, and if changed a complete new card deck would be created. The birth of UNIX in the 70s gave rise to the first file-system based version control, storing file changes and allowing permissions to be set for different users (read and/or write etc).

Since then, there have been at least six major version control systems, roughly one every ten years. Several of these are currently in wide use. Those you are likely to meet at some point are

- Git: the topic of these notes
- Mercurial: `https://www.mercurial-scm.org`
- Bitkeeper: originally paid-for, now open source `http://www.bitkeeper.org/`
- Subversion: still around, needs a server running, but that can be on the local machine `https://subversion.apache.org/`
- Visual Studio Team Services: Microsoft only, but quite good

4.3.4 Features of Git

Git was created by (and named after) Linus Torvalds (of the Linux Operating System) in 2005, because the system they were using, bitkeeper, removed its free community

[10] `https://git-scm.com`
[11] `https://github.com`
[12] `https://wiki.csc.warwick.ac.uk/twiki/bin/view/Main/GitServer`

edition. Git shares many of the useful features developed by earlier version-control systems. In particular:

- Moved/renamed/copied/deleted files retain version history
- Commits are atomic (either succeed completely or fail to do anything)
- Sophisticated branching and merging system (see Secs 4.4.3 and 4.4.4 for details)
- Used a distributed storage system, where each developer has as much of the repository as wanted in a local copy and merges onto central server when ready

Note that **Git is not Github, and Github is not Git**. Github is one popular online host of git repositorys but it has its own model for how to work and adds features like issue-trackers.

4.4 Basic Version Control with Git

4.4.1 Setting up a Repository

Once you have installed git, you first want to set up some basic information. We noted that git stores the author of every change, and this means you have to provide your identity. If you try the steps below before doing this, git will insist you do. Usually, it is enough to set a single identity globally, for all your git use. You do this using[13]

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

However, you can use several different email addresses, for example for work and for personal projects. In this case, after "git init" but before anything else, you should

```
git config user.name "John Doe"
git config user.email johndoe@example.com
```

without the global flag.

Now before you can do anything else, you have to set up a git repository. You can do this in an empty directory or one already containing files. *Be careful if this directory isn't at the bottom of your directory tree as any subdirectories will also be included.* Simply type

```
git init
```

Now you can add files to the repo. You usually do this in two steps. First you add, or stage the change, that is get things ready, and then you commit. You can add multiple files, or parts of files, before carrying on.

```
git add src/
git commit
```

[13] If you copy and paste these, note that before "global" should be two hyphens

The second line results in a text editor opening to allow you to specify the "commit message" to explain what and why you are adding. The editor can be changed[14] and often defaults to vim or nano.

Figure 4.1: Messages Matter. Don't do this! Permalink: https://imgs.xkcd.com/comics/git_commit.png Creative Commons Attribution-NonCommercial 2.5 License.

Git commit messages should follow a particular format, which originates from its use controlling the code of the Linux Kernel.[15] A typical message looks like

```
First check in of wave.f90

wave.f90 will be a demo of using a ''wave'' type MPI cyclic transfer 0->1->2 etc.  in order.
```

The first line is the subject, and should generally be less than 50 characters. The second line must be blank. Any text here is ignored. The subsequent lines are the message body, and should generally be less than 72 characters. You can use as many lines as you like, but be concise.

You now save and exit the editor, and git gives a short summary of what was committed. If you quit without saving the commit is aborted. The state of the files we committed has now been saved. Now we can make some changes to the files, and commit those. If we just try

```
1 git commit
```

we get a message like

```
On branch master
Changes not staged for commit:
...  no changes added to commit
```

which tells us we didn't stage the new changes with `git add`. We can do many `add` steps before we finally commit. We can also see what changes have been made at any point using

[14] e.g. https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

[15] These details are surprisingly hard to find written down, and you will probably meet many people who don't know them. Be considerate and share!

```
git status
```

which tells us the current state of the working directory: which files have changes that have been added, which have unstaged changes, and which files are not included in the repository. If the last list is very long, you may want to use a `.gitignore` file to tell git to ignore some file types. See e.g. https://git-scm.com/docs/gitignore

There are two useful shortcuts: for a few files that have been previously added so are known to git, we can explicitly commit them, without an add step like

```
git commit file1.txt file2.txt
```

or we can commit everything which is changed using

```
git commit -a
```

In all cases, we get the editor, we write a useful commit message and then we get some report like `1 file changed, 2 insertions, 3 deletions`

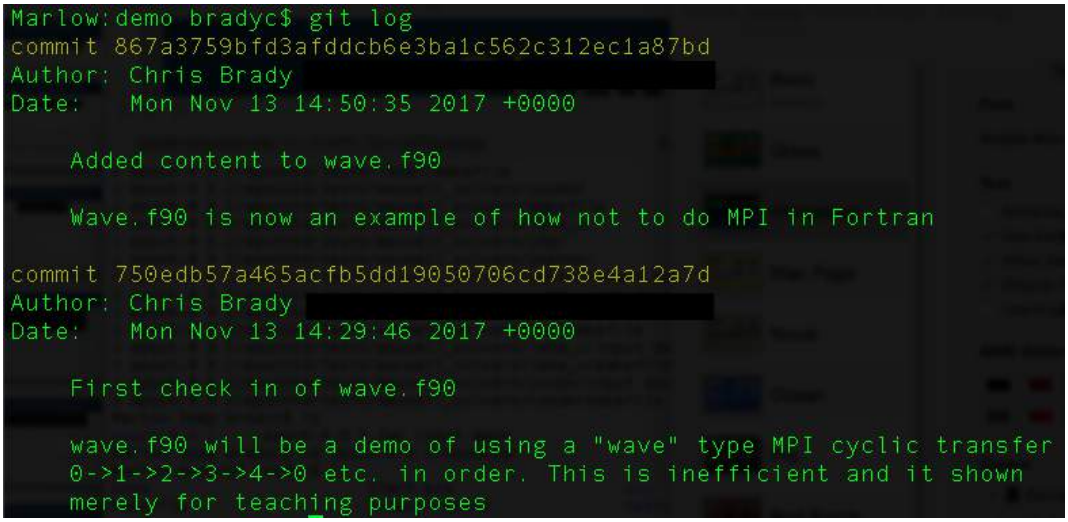


Figure 4.2: Typical "git log" output

We can see all of the commits we have made using the log.

```
git log
```

gives us output like Fig 4.2 Note the string after the word "commit". This is the "commit id" which uniquely identifies the commit. Git also accepts a shorter form of this, usually the first 8 characters.[16]

```
diff --git a/src/wave.f90 b/src/wave.f90
index ffaa053..c2e694e 100644
--- a/src/wave.f90
+++ b/src/wave.f90
@@ -3,7 +3,7 @@ PROGRAM wave
   USE mpi
   IMPLICIT NONE

-  INTEGER, PARAMETER :: tag = 100
+  INTEGER :: dummy_int

   INTEGER :: rank, recv_rank
   INTEGER :: nproc
```

Figure 4.3: Typical "git diff" output. A line referring to "tag" has been removed, a line defining "dummy_int" has been added.

4.4.2 Viewing and Undoing Changes

Git can show you a list of differences between two commits, or a list of differences between a given commit and the current state using the command "git diff", as e.g.

```
git diff abc123xyz                #All changes since abc...
git diff abc123xyz efg456uvw      #Changes between abc... and efg...
git diff abc123xyz file1.py file2.py   #Changes since abc... in file1 and
    file2 only
```

The output is in a "git-diff" format:
Lines with a "+" in the left-hand gutter have been added
Lines with a "-" have been removed.
Changed lines are shown as a removed line and then an added line.
The other lines are there to give context.
You will also see some sections starting with "@@" which give the line-number and column where the changes begin. The first pair is in the original, the second in the final, version. Example output is in Fig 4.3.
In vim these are coloured (usually green for adds, red for removes, blue for line numbers and your default colour (here bright-green) for everything else).

Undoing changes can become quite messy. Git is a distributed system, so if the code has ever left your control, you can't simply remove changes by changing the history, or everybody else's state will be broken. "reverts" are new commits which remove old changes, to put things back to how they were. They leave both the original commit and the new revert commit in the log. **If you accidentally commit something protected, like a password or personal data, a git revert will not remove it.**

[16] If you know about hashes, you may know about hash collisions, where different data gives the same output. Git needs the hashes to be unambiguous. For very large projects, the first 12 characters may be needed to ensure this, as e.g. https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection#Short-SHA-1

Take care, because fixing it will not be fun![17]

To revert one or more commits, use

```
git revert {lower_bound} {upper_bound}
```

where the lower bound is exclusive (last commit you want to leave unchanged) and the upper bound is inclusive (last commit you want to undo). When you do this, you will get the commit message editor for each reverted commit, saying `Revert ?original commit message?`. You rarely want to change these.

4.4.3 Branching

If you are working on several things at once, you may find branches useful. These are versions of code that git keeps separate for you, so that changes to one branch do not affect another. Whenever you create a repository, a default "master" branch is created. Adds and commits are always on the current branch. The command

```
git branch
```

will show the name of the branch you are on.

You can create a new branch using

```
git branch {name}
```

The branch is based on the last commit (on whatever branch you are on when running the command)[18]

The branch command doesn't move you to the new branch. You do this using

```
git checkout {name}
```

You will get a message, usually `Switched to branch 'name'`, or an error message. To create a branch and change to it in a single step, use

```
git checkout -b {new_branch_name} {existing_branch_name}
```

where the existing branch name is optional. This is very useful when working with a branch from a remote server, for example.

Checkout also lets you go back to some previous version of the code, and create a branch from there using

```
git checkout -b {new_branch_name} {commit ID}
```

You can checkout old versions without changing branches too, but this puts your repository into an odd state, so is best avoided for now.

Note that if you have uncommitted changes when you run git branch, those changes will come with you, and can be committed. If you try and change branches when

[17] E.g. https://stackoverflow.com/questions/31057892/i-accidentally-committed-a-sensitive-password-into-source-control

[18] You can branch from branches, and create very complex trees, but for now you will mostly want to create branches based on master.

you have uncommitted changes, you may get an error, saying `error: Your local changes to the following files would be overwritten by checkout:`. You can either commit those changes, or consider using "git stash" to preserve them to use later. See e.g. https://git-scm.com/docs/git-stash for the latter.

4.4.4 Merging

When you use branches to develop features, you usually eventually want to bring them back into the main version, when they're ready to be shared with users, or are fully complete. This is a merge, and uses the command

```
git merge {other_branch_name}
```

which brings changes from the other branch into the current one.

```
  ! Subroutine to initialise a thermal particle distribution
  SUBROUTINE setup_particle_temperature(part_species)

<<<<<<< HEAD
    REAL(num), DIMENSION(1-ng:,1-ng:), INTENT(IN) :: temperature
    INTEGER, INTENT(IN) :: direction
    TYPE(particle_species), POINTER :: part_species
    REAL(num), DIMENSION(1-ng:,1-ng:), INTENT(IN) :: drift
=======
    TYPE(particle_species), POINTER :: part_species
>>>>>>> non-thermal
    TYPE(particle_list), POINTER :: partlist
    REAL(num) :: mass, temp_local
    REAL(num), DIMENSION(3) :: drift_local
    TYPE(particle), POINTER :: current
    INTEGER(i8) :: ipart
    INTEGER :: idir
    TYPE(parameter_pack) :: parameters
#include "particle_head.inc"
```

Figure 4.4: A conflicted "git merge". non-thermal contains a change incompatible with our current branch (labelled HEAD as we're currently on it)

If you're lucky, the merge will be automatic and you will see a message about `Fast-forward` and are done. Otherwise, you will end up with files containing markers using the git diff format. Figure 4.4 shows an example. You will have to go through each file and "resolve the conflicts" (fix what git didn't know how to merge) before git lets you commit them. When you are done, finish using

```
git commit      #As normal
git merge --continue #Alternative in newer git versions
```

There are tools to help with merges, but they can get quite complicated, and while git tries to understand the language, it is a difficult problem in general. For example, if you have changed the indentation of a whole block of code, you may see the entire thing being removed and added again, and showing as a merge conflict.

Fig 4.5 shows a typical flow of branching and merging. When feature 1 is complete, it is merged back to master, and the feature 2 branch pulls in those changes to stay up-to-date, before continuing work. When feature 2 is finished, it is merged too.

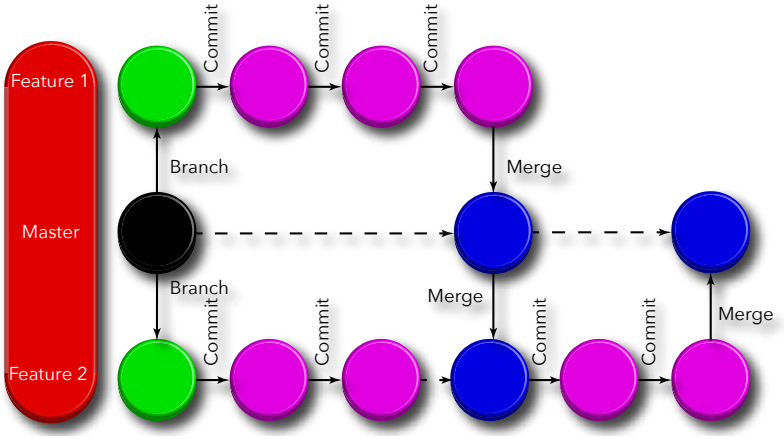


Figure 4.5: A schematic of typical git workflow with two feature branches and one master.

4.4.5 Remote Git Servers

```
Marlow:demo2 bradyc$ git clone https://github.com/LMFDB/lmfdb.git
Cloning into 'lmfdb'...
remote: Counting objects: 46154, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 46154 (delta 0), reused 2 (delta 0), pack-reused 46148
Receiving objects: 100% (46154/46154), 19.45 MiB | 1.52 MiB/s, done.
Resolving deltas: 100% (34294/34294), done.
Checking connectivity... done.
Marlow:demo2 bradyc$
```

Figure 4.6: Typical “git clone” command, for a Github repo.

Git is a distributed, networked version control system, which is the core of its real power. You can link between a local repository and a remote one, on a server, or on e.g. Github, and git remembers that. You can clone code from a remote repository and git will remember the origin. To clone code, you need the url of a remote server, then use the command

```
1 git clone {url}
```

Fig 4.6 shows an example using github - here you can get the url showing the green “clone or download” button on the repo’s page. This completely sets up the repo, and stores the “remote tracking” information (mostly the url you used). Note this will be a subdirectory of where you ran the command.

```
1 git branch -a
```

will tell you about all branches, including those on the remote you now have references to. Your master will now be linked to a remote master branch. The other branches are not downloaded by default, so if you check them out you will see similar text to Fig 4.6 about counting and receiving.

4.4.6 Pull and Push

When the copy of the code on the remote is updated, you will need to pull in those changes, with

```
1 git pull
```

This happens on a per-branch basis. Note that there is a related command, fetch, which just updates branch information and downloads changes, but doesn't merge them into yours. If your local copy has also changed, you will have to deal with merging changes from other developers with your own.

To upload your changes to the remote, you can push them, using

```
1 git push
```

If you are working with somebody else's repository, check whether they allow you to push directly. On e.g. Github, a different model is used, see Sec 4.4.7 Git tries to merge your changes with the remote copy, so make sure to pull first, or it will fail.

4.4.7 Github Flow

Once again, github is not git. However, it is one of the most popular public remote systems, and is quite easy to use, and also adds nice features like issue trackers.[19] Once you sign up for a Github account, you can push a local repository to github's server. Instructions are at https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/ or are given when you create a new repository via your github profile. A quick walkthrough of basic git for Github is at https://guides.github.com/activities/hello-world/

The other common task is to work on somebody else's code, and share your modifications with them and their users. They may give you push access to their github repository, but usually do not. Instead, you create a fork (basically a copy, but which knows where it was copied from) of their repository, push your work to it, and then make a pull request asking the owner of the main repository to pull (as in "git pull") the changes from your version of the repository. Figure 4.7 shows a typical pattern, and more details are at https://guides.github.com/introduction/flow/.

[19] To allow your users to tell you about bugs, requests etc.

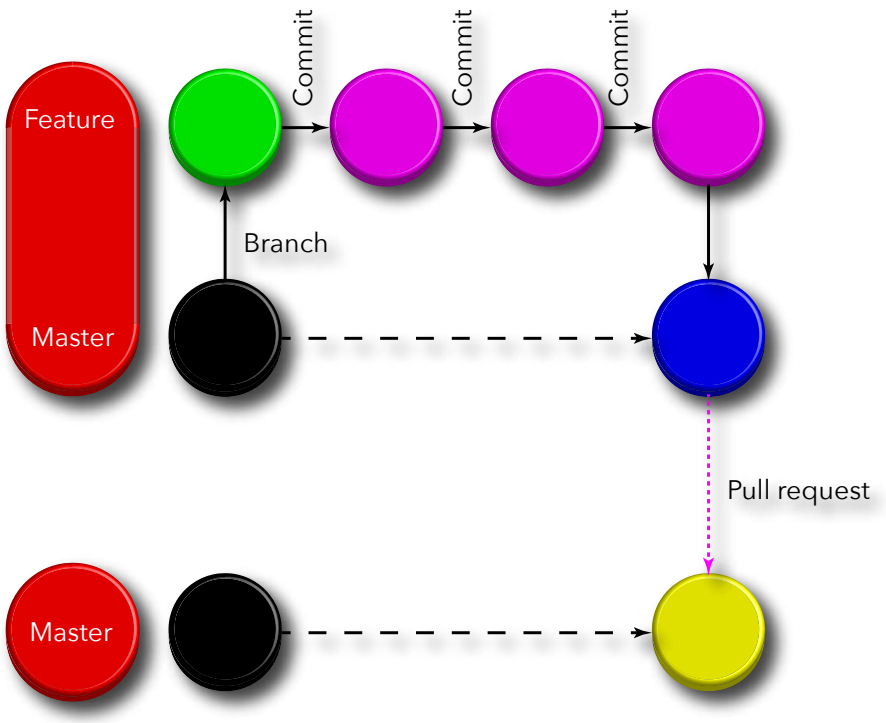


Figure 4.7: A schematic of typical Github workflow. You develop a feature on your fork, and them submit a pull request to have it included in the main repository.

4.5 Releases and Versioning

4.5.1 Versions of Code

If your code produces output files, you will at some point find yourself wanting or needing to know which version of your code a particular file was created with. You could check the dates, and try to work it out, but much better is to give your code version numbers whenever you make a change, and to write this version number into your files.

One very common, and very useful, versioning system is to use 3 numbers combined into a string like "1.1.4". The exact purpose of the three numbers varies, but in most cases they reflect levels of change:

- 2.x.y is a major version. This usually introduces significant new features beyond those of version 1.p.q. Sometimes it means old behaviour has been removed, changed, or otherwise broken.
- x.2.y is a minor version. This usually adds smaller new features, adds new behaviour without breaking the old, etc.
- x.y.2 has many meanings. It may be called patch, build, release or otherwise, and usually starts with a number. Sometimes this is followed by a dash and an

alphanumeric sequence. Usually this version represents small changes, fixes to bugs, etc.

In other words, you should "bump" the major version when results change substantially, for example you may change algorithm, or when you add some major new feature. Major version changes can break forwards compatibility. **Avoid breaking backwards compatibility where practical. That is, older data files should still work with newer code.**[20] Bump the minor version when you have small changes, perhaps adding a new feature but preserving all the old. Finally you can include a third identifier which gets incremented for every change.

4.5.2 Tagging your Output Files

The first thing to do, once you've decided on a version scheme such as the 3-part one above, is to make sure it is embedded into your output files (see also Sec 1.7.4). This means you can know which version created the files, and that is the first step to making them reproducible. The easiest way to do this is to put a (constant) string in your source code with the number, and to have the code print it into your files. Make sure to update the string when you make changes though!

4.5.3 Git Tags

Because it is easy to forget to change the version number when you commit changed code, you can take advantage of git's way to connect a version number to a particular code state, which is to use tags. Other VCSs also have methods to do this. When you wish to set or to change version number, you use one of the commands

```
git tag {tag_name}
git tag -a {tag_name} -m {tag_message}
```

The latter stores your name as creator and has various other advantages, such as ability to specify a message, so is generally recommended. Using the 3-part system we may do something like

```
git tag -a v0.0.1 -m "Prototype version"
```

We can then find out about the tag and the commit is is attached to using

```
git show v0.0.1
```

Tags aren't included in a push by default, so you have to do the special

```
git push {tag_name}
```

to share them.

The major advantage of this is that you can then include some recipe in your makefile which can extract the version information and pass it on to your code. Because git tags

[20] Note that often backwards compatibility is achieved by keeping the old code, and having the system use it when given an old file, but this does often mean code duplication.

are separate to your source code, you can go back and do this after you have committed changes. The code to extract the information is a bit horrible, but for example, if you are sure your code will only be obtained via git (and not e.g. downloaded, in which case you'll need something more complex) you add the following to the preamble part of your makefile

```
GIT_VERSION := $(shell git describe --abbrev=4 --dirty --always --tags)
CFLAGS += -DVERSION=\"$(GIT_VERSION)\"
```

as described at https://stackoverflow.com/a/12368262, and then use the variable VERSION inside your code.

More info on using tags is at https://git-scm.com/book/en/v2/Git-Basics-Tagging

Glossary - Workflow and Distribution

backwards compatibility A guarantee that anything possible or valid in an older version remains possible, so that e.g. input or output files from an older version can still be used. Sometimes this means that you can make the code behave exactly as it used to, sometimes it means only that you can use the files as a base for new files. For example Excel can read any Excel file, from any version correctly. *See also* forwards compatibility & sideways compatibility, 90, 105

branch Repositories can have more than one branch, an independent set of changes, often for some purpose such as a specific feature or fix. Branches can be merged together to combine their changes. 79, 92

commit (A commit) A chunk of changes, usually along with a message and an author etc. (To commit) To record files or changes in the version-control system, i.e. to add its existence and state to the history and store files and content however the system decides (often as incremental diffs). 81, 92, 93

dependency In general terms dependencies are the libraries, tools or other code which something uses (depends on). In build tools specifically, they are also known as prerequisites and are the things which must be built or done before a given item can be built or done. For example I may have a file-io module which I have to build before I can build my main code. 70

fetch To download changes to a repository. In git, this includes information on new branches but does not update your local copy of the code. *See also* pull, 88

fork To split off a copy of code to work on independently. Often this implies some sort of difference of opinion as to how things should be done. In the Github model, forks are encouraged in normal development: one forks code, adds features and then may or may not make a pull request back to the original repository. This

way only core developers can commit to the main repository, but anybody can easily modify the code. 88, 92

forwards compatibility A guarantee that files etc from a newer code version will still work with the older version, although some features may be missing. E.g. a file format designed to be extended: extended behaviour will be missing, but will simply be ignored by old versions. *See also* backwards compatibility & sideways compatibility, 90

library Code to solve a problem, intended to be used by other programmers in their programs. For example, in C there is the Standard Library which contains things like mathematical functions, string handling and other core function which is not part of the language itself. *See also* module & package, 92

master A common name for the primary branch of a code. Master should always be working and ready to release, as it is the default branch in git. Some consider it a bad idea to commit changes directly to master, preferring to work on branches and then merge or rebase.

merge Combining one set of changes with another, usually by merging two branchs The combined version usually shares its name with the branch that has been merged "onto". *See also* rebase, 86, 88, 91, 92

module A piece of a program, something like chapters in a book. In python modules are the things you import (possibly from a larger package). Fortran has modules explicitly, that are created using MODULE [name] and made available elsewhere with the USING statement. In C modules are closest to namespaces. *See also* package & library, 92

package A piece of software, usually stand-alone. In contrast a library is usually code only used by other programs, but there is a lot of overlap. This may contain multiple smaller modules. In context of OSs, packages are software that can be installed, whether they're available as compiled binaries or source code. *See also* module & library, 92

pull To download changes to a repository. In git this then integrates them with your local copy. If you have local changes, the remote changes are merged into yours. 88

pull request A request to pull-in code from elsewhere. In the Github model, one forks code, makes changes, and then raises a pull request for them to be integrated back to the original repo. 88, 91

push To upload your local state to a remote repository (see repository). You can push commits, whole branches, git tags etc. 88, 90, 93

rebase Replay changes as though they had been applied to a different starting point. Because systems like git work in terms of changes to code, they can go back through history and redo changes from a different base. For example, one can "rebase" a branch onto another. The changes in the first branch are taken one by one, and applied to the second branch. This differs from merging mainly in how changes are interleaved in the history. *See also* merge,

repository (Aka repo) A single project or piece of software under version control. In general a local repository is a working (in the sense of "to be worked on") copy of the code, whereas a remote repository is a copy everybody shares, pushing their work and combining changes. The remote copy can be sitting on somebody's machine - remote is a designation not a requirement. Note that git does not require a remote repo (or server), but some systems like subversion do. 81, 92

sideways compatibility A guarantee that code remains compatible with other code. For example you may create files for another program to read, and you want to make sure that your output remains compatible with their input requirements, even when these may change. *See also* forwards compatibility & backwards compatibility,

stage Staging means preparing changes to be added (commited) and comes from the similar concept in transport, `https://en.wikipedia.org/wiki/Staging_area`. 81, 82

VCS Version Control System; a tool for preserving versions of code and details about who, why and when they were created. 78, 79, 80, 90

Chapter 5

Wrap Up

> Only a Sith deals in absolutes - Star Wars

These notes have gone rapidly through a wide range of ideas, tools, and principles to give you a basic toolkit for developing software. Many things were left out and glossed over, and even the topics we have covered were necessarily fairly cursory, to make sure you have a grasp of the breadth of the subject. We promised at the start that we would only ever describe something as a **must** if it really was always the case. We gave far more **should** statements. These are broadly true, but not absolutely.

All of these statements are reproduced in Appendix B. Question them. When you can explain why we say them, and why they can break down, you know you understand. You may have noticed a few themes running through these statements too, mainly

- Make things that work, and will keep working
- Test and debug so you can be sure things work
- Think things through and understand why you're doing them
- Use the tools available or you'll waste time

5.1 Warning: The Expert Beginner

It is terribly tempting to take all your new tools and principles and throw them at everything. Even the restricted set of things we have covered here can make a real difference to your work. But we have not taught you everything you will need to know. There is a trap you must now be careful to avoid, sometimes referred to as "the expert beginner": stopping learning because of a mistaken belief that one is now an expert. This is the subject of an interesting article at https://www.daedtech.com/how-developers-stop-learning-rise-of-the-expert-beginner/ Use resources, ask questions, and don't be afraid to be wrong:

> It's only those who do nothing that make no mistakes. – Joseph Conrad

5.2 Where to go From Here

Everywhere! There are plenty of links to get you started in Appendix A and there are endless resources in the form of books and online guides to help you. The glossaries throughout this text have mentioned all sorts of concepts you might want to read further on. A few quotes in closing:

> First learn computer science and all the theory. Next develop a programming style. Then forget all that and just hack. – George Carrette

> Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter. – Eric S. Raymond

Glossary - General Programming

algorithm A sequence of steps to go from some input to some specified output.

atomicity Most code operations map into several instructions to the computer. Atomicity is the property that either the entire action will be performed, or none of it. This is commonly encountered in databases: for example if you overwrite a record with a new one you want to be sure not to end up with a mashup of the original and new record if something goes wrong. In some cases writing to a variable is not atomic: you could end up with one byte in memory from the old value and 3 bytes from the new, giving garbage. This is mainly a concern with multi-threaded programming or interrupts. 12

automagically (humorous) Automatically, as if by magic. Used mostly for systems with nice properties of doing what is actually needed rather than following a simple recipe, or when the details of how it works are tedious and boring, but the outcome very useful. 69

compiler flag (Aka directive) Command line arguments passed to the compiler to control compilation. For example in C you can define a value (for use with #ifdef etc) using -D[arg_name][= value]. Optimisation levels (how hard the compiler works to speed up or reduce memory use of your program) are usually set with a directive like -O[level number]. 69

heap Program memory that can be used dynamically (determined as the program runs), for example anything used with malloc in C, ALLOCATABLEs in Fortran etc. *C.f.* stack, 59

interface The functions available, including their signatures. The bare minimum somebody would need to use a chunk of code. 14

interpreter The interpreter, sometimes called a REPL (read-evaluate-print loop) is the program which runs your code in interpreted languages. Usually you can get a prompt at which you can type code directly, or you can invoke the interpreter with a script, and it will run.

language standard Rules specifying what valid code is in a given language, and what must be guaranteed by a compiler or interpreter about how this is implemented. 2, 4, 25, 97, 102

mutability In languages like Python, some data types are fixed when created, and cannot be changed later. These are called immutable. In practise you will mainly notice this with tuples. You can create a new tuple from some values, but you can't change a single element. Similarly, with strings you cannot change a single character, you have to create a new string with the change included.

pass(ed) by reference Different languages pass parameters into functions differently. When passed by reference, a reference to the variable is given, so any changes will affect the named variable in the calling code. For example a function
FUNCTION inc(x)
x = x+1
END FUNCTION
y=1
inc(y)
PRINT y
would give 2. *See also* pass(ed) by value, 5

pass(ed) by value Different languages pass parameters into functions differently. When passed by value, the current value (at call time) of the variable is copied to a dummy variable inside the function. For example a function
FUNCTION inc(x)
x = x+1
END FUNCTION
y=1
inc(y)
PRINT y
would give 1 as y is not changed by the call to inc. *See also* pass(ed) by reference, 5

scope Scope of a variable is the region of the program in which it exists and can be used. Most languages have "function scope" so variables you create inside a function can't be used outside it. C-like languages add "block scope" so a variable defined, for example, inside an if-block is lost when the block ends. 51

source (code) Your program text. This is distinct from the runnable executable, or the bytecode or tokenised code produced by e.g. Python. iii

stack Program memory used for static variables (where the memory needed is known at compile time and can't change) such as numbers, strings etc. *C.f.* heap,

subroutine (c.f. function) In languages like Fortran, subroutines are sections of code which can be used like a function but have no return value.

undefined behaviour Things which are not specified by a language standard can do whatever they want - their behaviour is undefined. Beware that this can mean doing exactly what you expect. Until it doesn't. 5, 27, 30, 40, 55, 102

Appendix A

Links and Resources

Warwick RSE lists some useful general programming resources at https://www2.warwick.ac.uk/research/rtp/sc/rse/training/ You'll probably find courses on almost any aspect of programming on your favourite online service.

A.0.1 Tools Used

All of the tools and resources we used in these notes are available online, along with some very good guides.

- Doxygen http://www.stack.nl/~dimitri/doxygen/
- GDB https://www.gnu.org/software/gdb/
- PDB https://docs.python.org/2/library/pdb.html or https://docs.python.org/3/library/pdb.html
- Valgrind http://valgrind.org/docs/manual/manual.html
- gprof https://sourceware.org/binutils/docs/gprof/
- perf https://perf.wiki.kernel.org/index.php/Main_Page
- Make https://www.gnu.org/software/make/
- git https://git-scm.com/ and https://git-scm.com/book/en/v2

A.0.2 Online Sharing and Editing

You may often want to share a snippet of code with others, for example when asking for help with a bug or providing it. Sometimes you may want to test a small piece of code while away from your normal computer, or you want to share a snippet and its results. Below are the most popular online tools, although many others exist. Note that most will keep the code you enter, especially if you're not signed up. Some make

it public, showing lists of recent items; several allow you to make your item unlisted but sharable via link.
Where possible below I have created an example using the IEEE floating point rules C code and included that Sharing only:

- Pastebin https://pastebin.com/ With a free account you can control and delete your content. Example at https://pastebin.com/Vdqmm4rE
- Github Gists https://gist.github.com/ Requires Github account. Example at https://gist.github.com/hratcliffe/2dd02726bb18323c78d78acbd9d2c1f1

Allow running and showing results:

- CodePad http://codepad.org/ (Allows private pastes) Example at http://codepad.org/k0q8k0ef
- IDEOne https://ideone.com Example at https://ideone.com/0ryEtS Includes Fortran support
- https://aws.amazon.com/cloud9/ Requires AWS signup, may not be free.
- For SQL database code http://sqlfiddle.com/

A.0.3 Code Style

Code style is very subjective, and the crucial thing is to be consistent. Adopt your own style, but when working in teams etc be sure to follow any guidelines. The following are not recommendations, just examples of the wealth of options available online.

- Mozilla's general guidelines for code https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style
- NASA's C style guide. From 1994 but ANSI C is from 1989 so still useful http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf
- Google's C++ guide https://google.github.io/styleguide/cppguide.html
- Stroustrup and Sutter's C++ guide. Covers style and much more in Modern C++ https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
- Fortran Style http://www.fortran90.org/src/best-practices.html
- Met Office Fortran style http://research.metoffice.gov.uk/research/nwp/numerical/fortran90/f90_standards.html
- Python PEP-8 style guide https://www.python.org/dev/peps/pep-0008/
- Google's Python Style https://google.github.io/styleguide/pyguide.html

A.0.4 Validation Tools

In interpreted languages in particular, there is no compiler to check for syntax errors etc. A useful alternative is the "linter". The origins of the name are lost to history, but these tools read your code and analyse correctness and style. Note that correctness and style can begin to merge together. For example, some regard Python's PEP-8 as guidelines, some consider them to define correct Python code.
Note: these are often called static validation tools, as they read your code as it is on the page, rather than work out the full context of a statement.

- PyFlakes https://pypi.python.org/pypi/pyflakes
- PyLint https://www.pylint.org/
- JSHint (For Javascript code, but very useful if you ever deal with JS) http://jshint.com/about/
- For shell scripts https://www.shellcheck.net/
- Dozens of online regex checkers if you use those e.g. https://regex101.com/, https://www.regextester.com/ (make sure to get the right flavour http://www.regular-expressions.info/tutorial.html)

A.0.5 Code Documentation Tools

Many tools exist that allow you to embed special comments in your source code and then extract them as some nicely formatted HTML, LaTeXand/or Markdown code. These are great for generating developer documentation and docs for libraries etc. Several options also exist for hosting the docs online. A few of these are listed below. See also Sec 1.6.1 Generators:

- Wikipedia link to list of options https://en.wikipedia.org/wiki/Comparison_of_documentation_generators
- Doxygen (great for C++, good for C/Fortran, Python parsing experimental; output as Latex and HTML) http://www.stack.nl/~dimitri/doxygen/
- Python docstrings and pydoc https://docs.python.org/2/library/pydoc.html
- Sphinx (designed for Python, can handle a range of languages) http://www.sphinx-doc.org/en/stable/
- Pandoc (doesn't generate docs, invaluable if you need to convert formats) http://pandoc.org/

Hosting:

- Github.io https://pages.github.com/ which allows you to create a webpage in Github repo form
- Read The Docs https://readthedocs.org/

A.0.6 Resources and Help

Everybody will need assistance with something at least once, and there are many resources for doing so. Often finding a good tutorial or guide is enough. If you do need to ask for direct help, our guidelines for making a good bug or issue report are at https://warwick.ac.uk/research/rtp/sc/rse/faq#bugreport and apply to most asking-for-help scenarios as well as actual bugs.

- Tutorials etc on the Python wiki https://wiki.python.org/moin/BeginnersGuide/Programmers
- Tutorials etc on the Fortran wiki http://fortranwiki.org/fortran/show/Tutorials
- https://stackoverflow.com/ Sometimes described as "beginner unfriendly" but as long as you check if your question was already tackled and then write a good post, usually helpful
- Places like reddit often have useful fora such as https://www.reddit.com/r/learnprogramming/ or https://www.reddit.com/r/programming/
- Stackexchange is similar to stackoverflow, e.g. https://softwareengineering.stackexchange.com/

Appendix B

Must and Shoulds

B.1 Must

- As researchers, your priority is research. You code needs to work, and you need to know it works. Anything that does not enhance that goal is decoration.
- If your code involves any sensitive data, you really must read up on relevant regulations and protocol
- Create a working code
- Follow your chosen language standard
- Use some sort of versioning
- Validate your code enough
- Before you start coding in earnest, you will want to have some sort of plan
- do not write code that violates standards or has undefined behaviour
- if protocol matters you must find and follow the proper guidelines
- Do not optimise the algorithm at this early stage, but do not be afraid to throw it away if it will clearly not serve your purpose.
- Do not assume every supposed anti-pattern is bad code
- But don't jump to assuming your problem is special
- Never relax on standards - undefined behaviour is always bad
- Self-documenting style does not mean you do not have to document your code
- Always validate your inputs.

- If your work is funded by a research council you must read and obey any rules they have as to sharing your code and its results
- you did make a plan, right?
- Always nullify pointers after freeing the memory they point to
- when you have eliminated the impossible and still not found the answer, perhaps it wasn't as impossible as all that.
- compilers will warn you about bugs like accidental assignment instead of comparison if you let them
- The fanciest debugger in the world will not help you if you are not *thinking*
- Stop and think
- you must document that a limitation exists
- Some form of testing is essential in your software.
- running tests tells you that the tests pass, not that the code is correct. You must work hard to make this the case, not suppose it.
- Unless your requirement is bit-wise exact reproducibility, (page 46) do not compare floats for exact equality in your tests.
- Once your code works, and not before, you may want to consider profiling
- Testing is a tool, not a goal in itself. Your goal is writing *correct* software to do research.
- Remember that any two points can be joined by a straight line so you need at least 3 points for convergence
- The most vital debugging and testing tool is your own brain.
- Always start at the first error! Often the rest are the same error popping up later on.
- When your code works, and not before, you can consider optimising. Before doing this, you need to know which parts are slow.
- This pattern, circular dependency, must be avoided.
- Make recipe lines must be indented using a TAB character. Spaces will not do.
- Only the last recipe is run, but prerequisites from all rules are considered
- The most important thing about version control is to do it. It doesn't matter how, as long as it works.

- Git is not Github, and Github is not Git
- If you accidentally commit something protected, like a password or personal data, a git revert will not remove it. Take care, because fixing it will not be fun!
- If you are working with somebody else's repository, check whether they allow you to push directly. On e.g. Github, a different model is used, see Sec 4.4.7

B.2 Should

- Divide into reasonably sized functions
- Binary bisection has uses in all sorts of areas, so make sure you understand how it works.
- The moral here is NOT that you should not use libraries, but to be selective:
- It is always a good idea to write a simple program with a new tool, before trying to integrate it into your actual code.
- Avoid ambiguous letters: 0 and O are easily confused, as are I, l and 1
- you should help your users to preserve the information to reproduce their work in future
- do not trust anything supplied by a user, even if that is you. This is not because users cannot be trusted, but mistakes and ambiguities can happen.
- Compromises should be covered in code documentation. Make sure to consider this in your own code, as you may not remember your reasons in a few months time.
- But if you find yourself wondering "but why" very often, consider whether you have got the right tool for the job.
- your aim is usually to find where what you *thought* would happen diverges from what *actually* happens
- you should consider adding code to check for violation of your limitations
- The tests you really need to run are probably the ones you haven't thought of, because you never realised that could be a problem!
- Be careful when setting error intervals: absolute values are useful for things like floating point errors, but in most we probably want to use an error percentage
- Avoid bit-wise exactness testing where ever possible. In most cases it is excessive. This means no exact comparisons of floats.

- the compiler will not reorder *dependent* pieces of code; it can omit unreachable code and no-ops and can remove certain other things.
- Fix earlier errors first. Often later errors are a direct result of earlier ones, so it pays to consider them in order.
- make sure to distinguish good design from premature optimisation: don't choose approaches that wont ever be fast enough to be useful.
- Remember the brackets around Makefile variables.
- If you are not already, you should consider using Python modules, rather than simple scripts.
- Avoid breaking backwards compatibility where practical. That is, older data files should still work with newer code
- The first thing to do, once you've decided on a version scheme such as the 3-part one above, is to make sure it is embedded into your output files (see also Sec 1.7.4).