

# Scientific computing: An introduction to tools and programming languages

“what you need to learn now to decide  
what you need to learn next”

Bob Dowling  
rjd4@cam.ac.uk  
University Information Services

I should also explain why this course exists. A few years ago a member of the UIS conducted some usability interviews with people in the University about the usability of some high-end e-science software. The response he got back was not what we were expecting; it was “why are you wasting our time with this high level nonsense? Help us with the basics!”

So we put together a set of courses and advice on the basics. And this is where it all starts. I'm going to talk about the elementary material, what you need to know and the courses available to help you learn it. You do not need to attend all the courses. Part of the purpose of *this* course is to help you decide which ones you need to take and which ones you don't.

This course is the first in that set of courses designed to assist members of the University who need to program computers to help with their science. You do not need to attend all these courses. The purpose of this course is to teach you the minimum you need for any of them and to teach you which courses you need and which you don't.

The final slide of the talk gives a URL through which all the relevant UIS courses can be found.

# Course outline

Basic concepts

Good practice

Specialist applications

Programming languages

# Course outline

**Basic concepts**

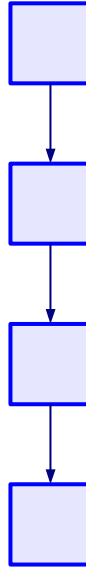
Good practice

Specialist applications

Programming languages

# Serial computing

Single CPU



Let's start with the most basic of concepts: the program.

A *simple* computer program is a linear series of instructions which the computer does one after the other. On completing one it moves on to the next and so on.

These instructions need not all be of the same sort. Some may take much longer than others. Some instructions may be called repeatedly. Some may trigger activity elsewhere and not complete until that remote activity is over. But the principle is that a program is a series of commands which are run one after the other.

In the simplest case all the instructions take place in the same CPU (more precisely on a single core in a single CPU).

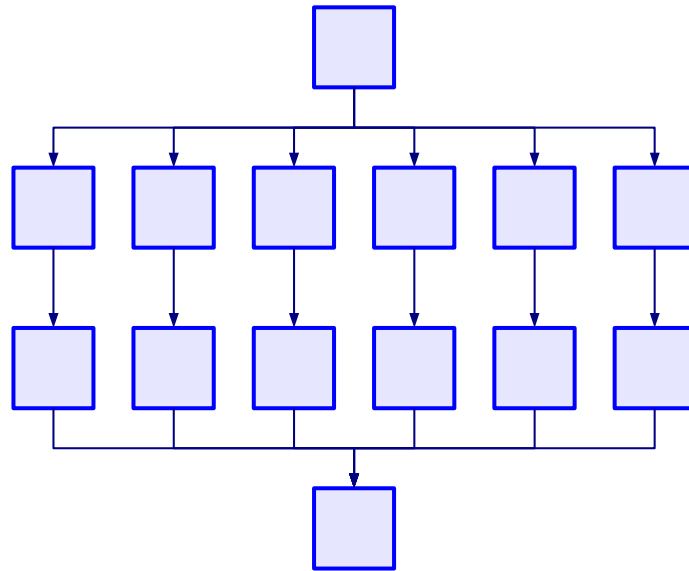
# Parallel computing

Multiple CPUs

**Single  
Instruction  
Multiple  
Data**

MPI

OpenMP



But programs need not be linear. In the past decade or so scientific computing has moved into “parallel programming” where computers with multiple processors can run the program through many of them simultaneously. It's also possible to run multiple “threads” of control on a single processor.

The most common form is called SIMD (pronounced “simm-dee”) which stands for “single instruction, multiple data”. In this example the program takes the memory allocated to the program’s data and allocates a different CPU (in the slide, six CPUs) to run the same code over the different components of the data.

This is typically much harder to program correctly and requires specialist skills. Two common frameworks for developing parallel code are MPI (Message Passing Interface) and OpenMP (Open Message Passing).

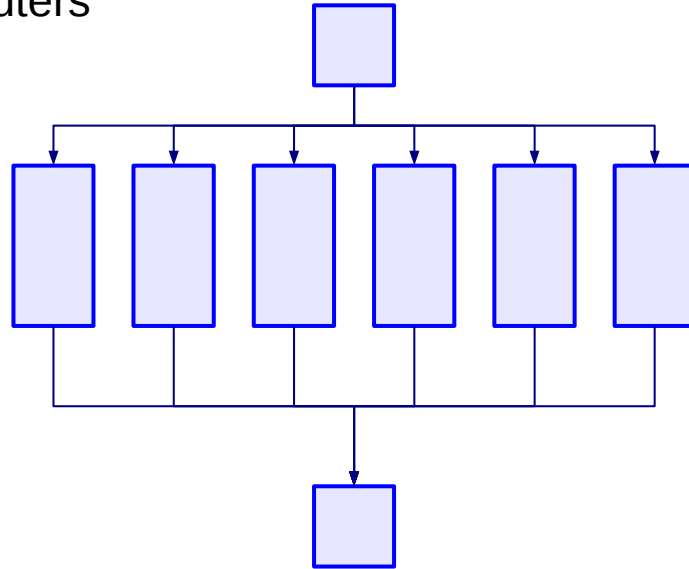
# Parallel computing courses

Parallel Programming:  
Options and Design

Parallel Programming:  
Introduction to MPI

# Distributed computing

Multiple computers



In the extreme case of parallel programming, we split the calculation not only between CPUs but between computers. In this case we have to consider the time taken to copy the data between computers as well.

# Distributed computing courses

HTCondor and CamGrid



# High Performance Computing course

High Performance Computing:  
An Introduction

# Floating point numbers

e.g. numerical simulations

Universal principles:

0.1 → 0.10000000000001  
and worse...

```
>>> 0.1 + 0.1  
0.2
```

```
>>> 0.1 + 0.1 + 0.1  
0.30000000000000004
```

The base hardware is what manipulates the data in your programs. As a result it can have an effect on that data. For example, computers typically work in base 2 (“binary”) so they can record real numbers like  $\frac{1}{2}$  exactly. Decimal fractions like 0.1, however, have to be approximated.

The vast majority of numerical simulations require real numbers, approximated as “floating point” numbers in computers. Under certain circumstances this can become significant. Inaccuracies in the floating point representation can accumulate and give erroneous results. Note that the inaccuracies often start hidden and then reveal themselves unexpectedly.

# Floating point courses

Program Design:  
How Computers Handle Numbers

# Text processing

e.g. sequence comparison  
text searching

`^f.*x$`

“Regular expressions”

fabliaux  
factrix  
falx  
faulx  
faux  
fax  
feedbox

...  
fornix  
forty-six  
fourplex  
fowlpox  
fox  
fricandeaux  
frutex  
fundatrix

But there's more to life than real numbers. Under the covers computers work with integers just as much as, and sometimes much more than, floating point numbers. These tend not to be used to represent numbers directly but to refer into other sorts of data. Good examples of these situations involve searching either in databases or in texts.

# Regular expression courses

Programming Concepts:  
Pattern Matching Using Regular Expressions

Python 3: (includes a regular  
Advanced Topics expressions unit)  
(Self-paced)

# Course outline

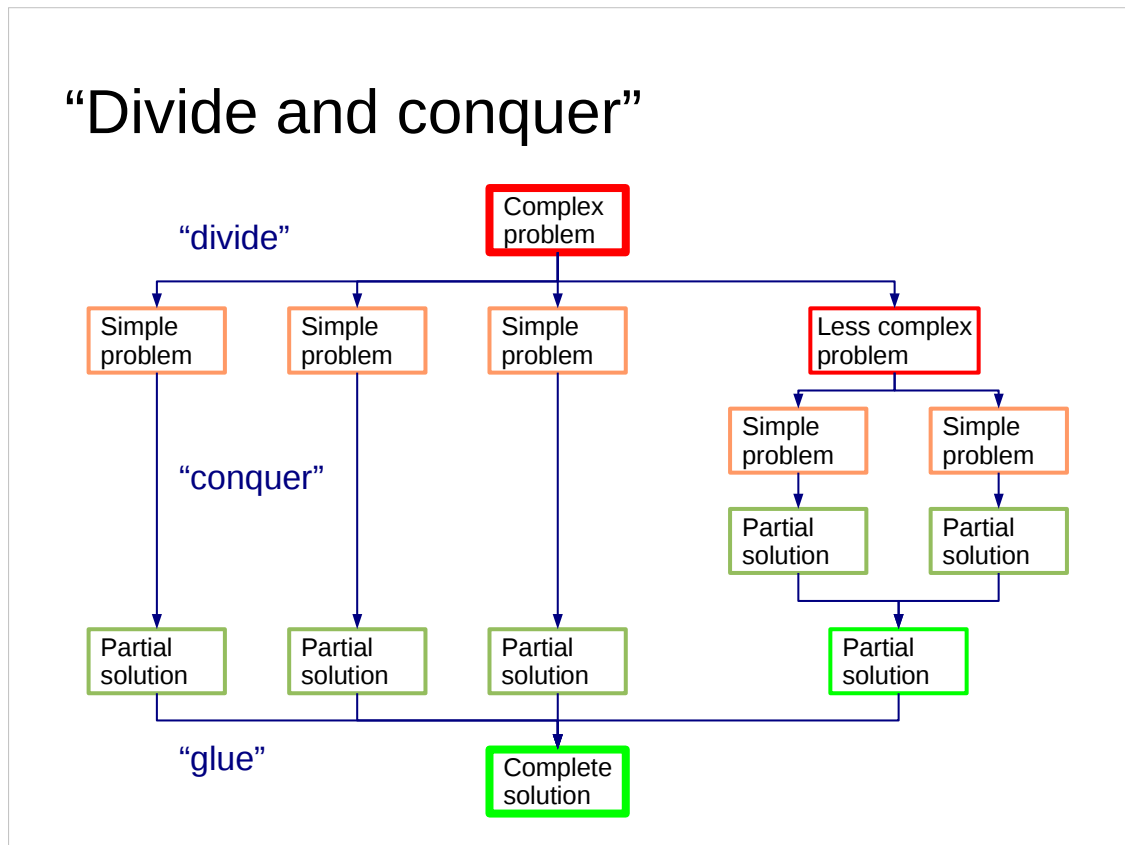
Basic concepts

**Good practice**

Specialist applications

Programming languages

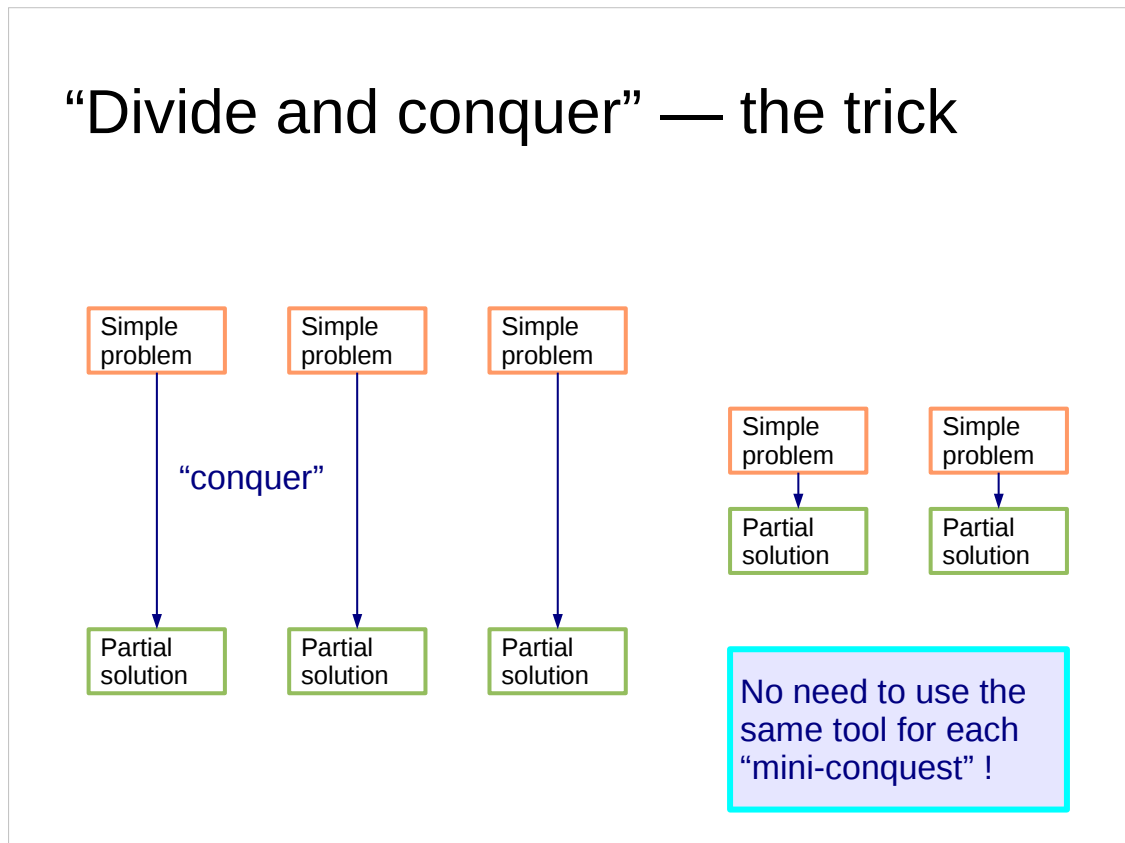
# “Divide and conquer”



Suppose you have a task you need the computer to perform. The key to succeeding is to split your task up into a sequence of simpler tasks. You may repeat this trick several times, producing ever simpler sub-tasks. Eventually you get tasks simple enough that you can code them up.

That sounds trivial. But that trick, repeated often, is how programming works. **“Divide and conquer.”**

## “Divide and conquer” — the trick



The reason that this works so well is that you don't have to use the same tool for all the subtasks. Different tools are suitable for different bits of your task. So long as you can glue the parts together again there is no need to use one tool for everything. While it may sound harder to use lots of different tools rather than just one the simplification gained by the splitting and the specificity of the tools more than makes up for it.



## Example

“ Read 256 lines of data represented in a CSV format. Each line should have 256 numbers on it, but some are split into two lines of 128 numbers each. Run Aardvark’s algorithm on each 256×256 set of data. Write out the output as text in the same CSV format (exactly 256 numbers per line, every line) and plot a heat graph of the output to a separate file. Keep reading 256-line lumps like this until they’re all done. ”

In practice, the instructions you are given to write code look scary at first glance. The trick is to divide and conquer.

# Example

Read 256 lines of data represented in a CSV format.

Each line should have 256 numbers on it, but some are split into two lines of 128 numbers each.

Run Aardvark's algorithm on each  $256 \times 256$  set of data.

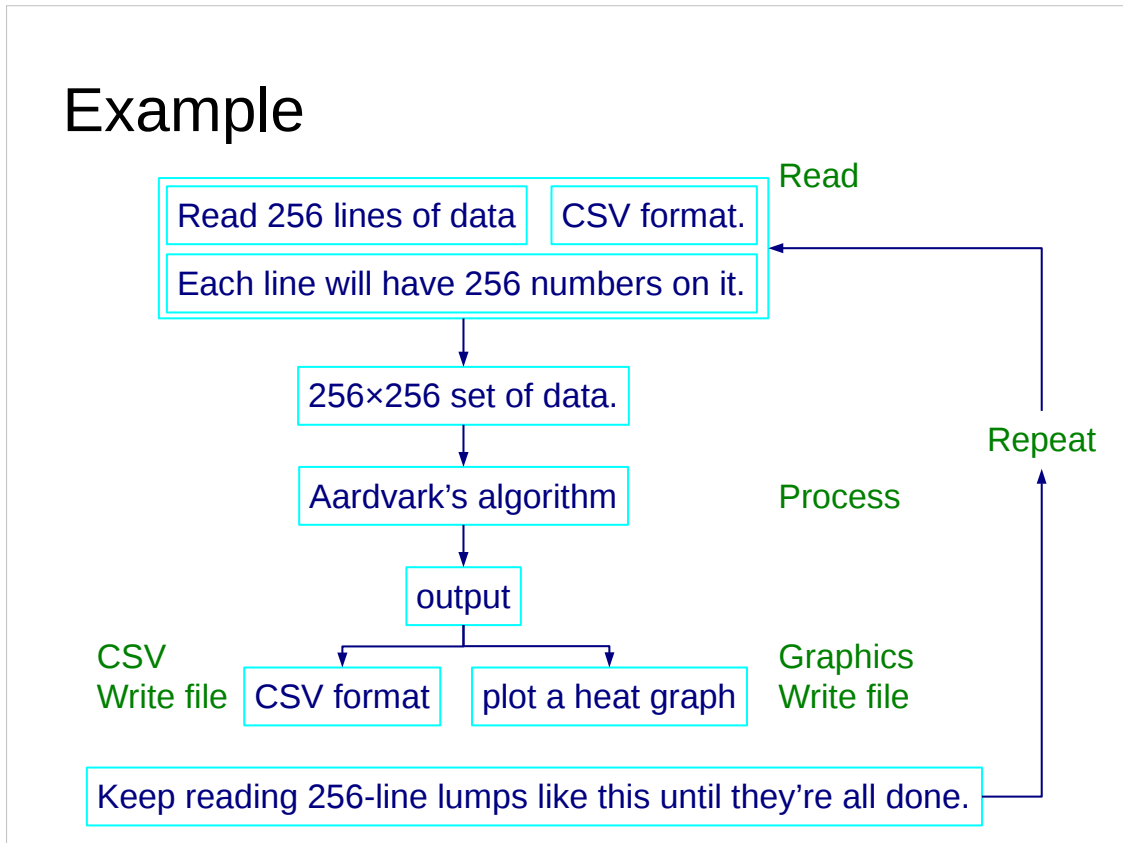
Write out the output as text in the same CSV format (exactly 256 numbers per line, every line)

and plot a heat graph of the output to a separate file.

Keep reading 256-line lumps like this until they're all done.

So we start by dividing. What are the bits we need to worry about?

# Example



If we can conquer each bit we win.

# “Structured programming”

Split program into “lumps”

Use lumps methodically

Do not repeat code

“**Lumps**” ?



Programs

Functions

Modules

Units

Now we will look at the bits themselves. They go by various names, such as “objects”, “functions”, “modules” and “units”. I prefer “lumps”. It's a simple, no-nonsense word that rather deflates the pompous claims made by some computing people.

If you split your program up into sets of these lumps, and reuse lumps when you need the same functionality twice or more, then you stand a good chance of success. If you don't, and write chaotic, unstructured code then you will have to work much harder to get a program that works and harder still to get one that works correctly.

So why am I talking about this before we've even looked at any programming languages? It's because this rule about splitting your program up into a structured collection of parts is common over every single programming language. It's an absolute rule — and they're rare in this business!

## Example: unstructured code

```
a_norm = 0.0
for i in range(0,100):
    a_norm += a[i]*a[i]

...

b_norm = 0.0
for i in range(0,100):
    b_norm += b[i]*b[i]

...

c_norm = 0.0
for i in range(0,100):
    c_norm += c[i]*c[i]
```

Repetition !

So what do I mean? Let's look at an example of bad code. You don't need to worry about the language (it's a language called Python that we will talk about later) because I hope the general principle is clear. We're calculating

$$\sum_{i=0}^{i=99} x_i^2$$

for three different sets of 100 values in three different parts of a program.

This commits the cardinal programming sin of repetition. If we wanted to improve the way we calculated this sum we would have to do it three times. (And if we want accurate sums we do need to improve it.)

We might also want to speed up our program. But does our program spend the majority of its time doing these sums or a tiny fraction of its time? If I spend an hour speeding it up is that more time than I will ever save running the slightly faster program? I can't tell until I isolate this operation into one part of my program where I can time it.

## Example: structured code

```
def norm2(v):  
    v_norm = 0.0  
    for i in range(0,100):  
        v_norm += v[i]*v[i]  
    return v_norm
```

Single instance  
of the code.

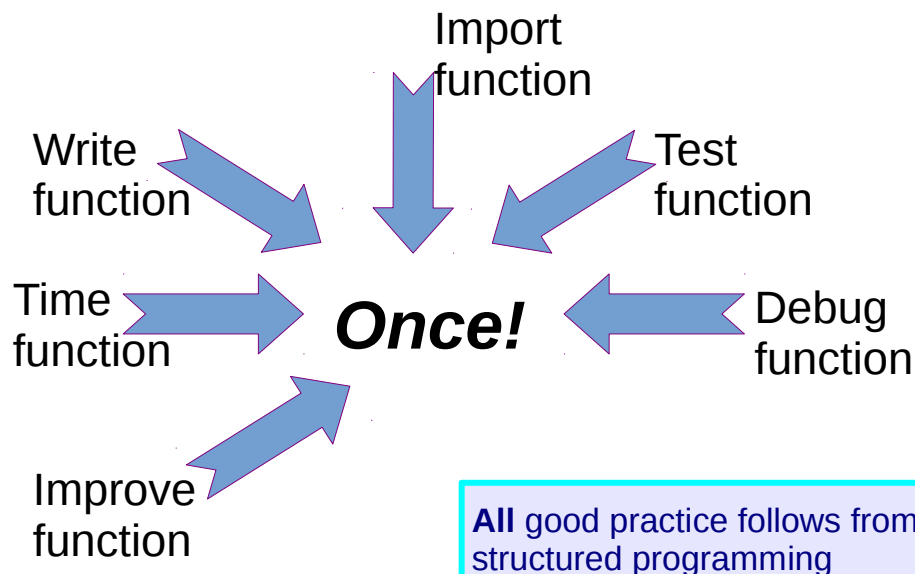
```
a_norm = norm2(a)  
...  
b_norm = norm2(b)  
...  
c_norm = norm2(c)
```

Calling the function  
three times

So let's improve it. We take the repeated operation and move it to a single place in the code wrapped in a function. Then in the three parts of the program where we calculate our sum, we simply make use of this function.

(What I am illustrating here is written in the Python but the principle is universal and hopefully it's simple enough to read that you don't need Python fluency to follow along.)

# Structured programming



So our code is only written *once*.

We can make any improvements or speed ups we want in just one place.

Furthermore we can find out how much time is spent running that function in total.

Also, now that we have a separate function, we can test it in isolation from the rest of the program to check it gives the right answer! If we do find mistakes we only have to fix the code in one place and that place is typically easier to find.

## Example: improved code

```
def norm2(v):  
    w = []  
    for i in range(0,100):  
        w.append(v[i]*v[i])  
    w.sort()  
    v_norm = 0.0  
    for i in range(0,100):  
        v_norm += w[i]  
    return v_norm
```

Improved code

```
a_norm = norm2(a)  
...  
b_norm = norm2(b)  
...  
c_norm = norm2(c)
```

No change to  
calling function

So let's make an improvement.

We don't need to understand the details of the improvement but in a nutshell, if you are adding up lots of numbers always add together the smallest numbers first to get a more accurate answer.

(For example, in the C programming language on one particular computer if I add up  $1/n$  starting with  $n=1$  up to  $n=10,000,000$  I get approximately 15.404. If I add them up starting with  $n=10,000,000$  and counting down to  $n=1$  I get 16.686!)

What is important is that I have made the improvement only once and it has immediately affected all three calculations in the program. This would have been much harder (three times the typing plus the work finding the cases in the program) if I hadn't split out the calculation into a function.

All good programming follows from good structuring.



## Example: improved again code

```
def norm2(v):  
    w = [item*item for item in v]  
    w.sort()  
  
    v_norm = 0.0  
    for w_item in w:  
        v_norm += w_item  
    return v_norm
```

More flexible,  
"pythonic" code

```
a_norm = norm2(a)  
...  
b_norm = norm2(b)  
...  
c_norm = norm2(c)
```

Still no change to  
calling function

So let's improve it again!

This version makes the function cope with sequences of numbers of any length. This is just good house-keeping but might be useful if all of a sudden my sequences had 1,000 entries in them rather than 100.

It has also introduced a native Python idiom which makes it (fractionally) faster.

Again, I have only had to make the change once and there is no change to the main flow of the code; I just call the function, same as I did before.

## Example: best code

```
from library import norm2
```

Somebody  
else's code!

```
a_norm = norm2(a)  
...  
b_norm = norm2(b)  
...  
c_norm = norm2(c)
```

No change to  
calling function

And now we come to the ultimate improvement:

Get someone else to do it for you!

There are people who make their living writing routines to do this donkey work extremely quickly in ways adapted specifically for your sort of computer. They take all their functions and wrap them together in so-called "libraries". What you need to do is to call on one of these libraries and to use a function from it. You don't need to know how it does it, and the details may vary from machine to machine, but you just need to know that it does it. These functions are written by experts. Almost certainly they have done a better job than you ever will. So don't compete with them; exploit them. Do not try to do for yourself what someone else has done for you already.

**Never re-invent the wheel.**

# Structured programming courses

Programming Concepts:  
Introduction for Absolute Beginners

# Libraries

Written by experts

In every area

**Learn what  
libraries exist  
in your area**

Use them

Save your effort  
for your research



These libraries of functions are your salvation. All you need know is what libraries exist and how to call them. Much of our programming courses consists of telling you this and giving you an introduction to the library to get a feel for its shape.

Your time is better spent on original research than inferior duplication of work that already exists. Save your effort for your research!

## Example libraries

**Numerical Algorithms Group**



**Scientific Python**

**Numerical Python**



So what can we get from libraries? The question “what can't we get” probably has a shorter answer. You name it; the libraries have got it.

For example the NAG libraries and the Scientific Python libraries have functions for at least the following topics:

- Roots of equations
- Differential and Integral Equations
- Interpolation, Fitting & Optimisation
- Linear Algebra
- Special Functions

and many, many more.

# Hard to improve on library functions

```
for(int i=0; i<N, i++)
{
  for(int j=0; j<P, j++)
  {
    for(int k=0; k<Q, k++)
    {
      a[i][j] += b[i][k]*c[k][j]
    }
  }
}
```

```
for(int k=0; k<Q, k++)
{
  for(int j=0; j<P, j++)
  {
    a[i][j] += b[i][k]*c[k][j]
  }
}
```

This "trick" may save you 1% on each matrix multiplication.

It is a complete waste of time!

Here's a trivial example of why you should rely on external libraries.

These two ways to multiply matrices differ in only one regard: the order of the operations. One is faster than the other because of the way the system manipulates the memory that the values are stored in.

You don't need to know this. You just need to know that the person who wrote the matrix multiplication function in the matrix library you should be using did know it and picked the right algorithm.

The one on the right is slightly faster in some languages. Neither is as accurate as it could be because we ought to order the added terms smallest to largest before adding them if we were concerned about precision.

However, this is the level of change that you might make to give your program a 5% speed up.

But for large matrices the difference between these two techniques is utterly irrelevant.

## Hard to improve on library functions

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

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = M_1 + M_2 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

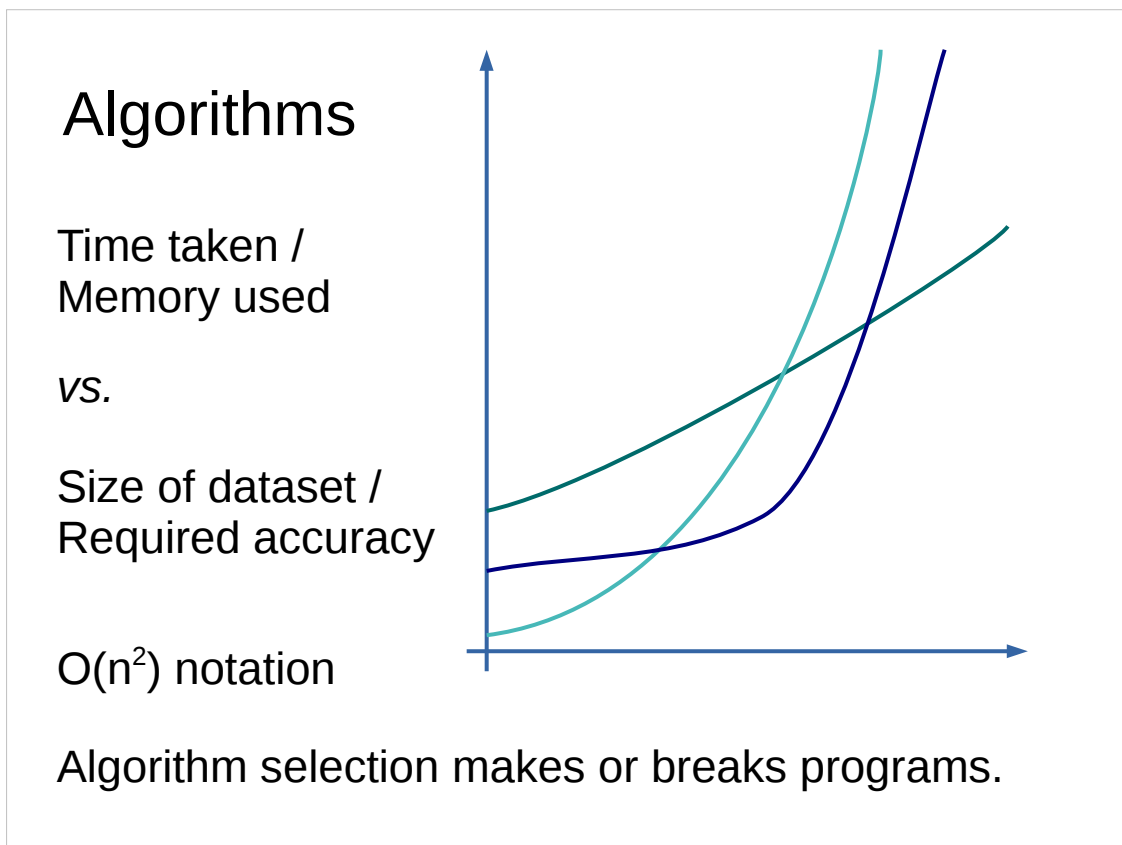
$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Applied recursively: *much* faster

Volker Strassen's algorithm, as shown in the slide, is far more efficient for large matrices but I doubt you would ever have thought of it. I also doubt you would want to code it, either.

**So use a library!**



Every function implements a recipe. It generates output from input somehow. The posh name for that “somehow” is an algorithm. There are efficient ways to do some tasks and inefficient ways to do the same thing. No optimizer is going to save you from choosing the wrong way to do something.

For example, there are two ways to sort a list of numbers. (Actually there are very many more than two.) Suppose they both take 10 seconds to sort 10 values. One, called “bubble sort” would then take roughly 40 seconds to sort 20 and 1,000 seconds to sort 100. The other, called “quick sort”, would take 26 seconds to sort 20 values and 200 seconds to sort 100. The rate at which different algorithms scale as the size of their inputs go up is of critical importance and has grown its own notation called “Big O” notation and we say that bubble sort is “order of n squared” because the amount of time it takes increases like the square of the number of elements it has to sort. We write this as “ $O(n^2)$ ”, hence the name “Big O notation”.

The importance of picking a good algorithm cannot be overstated. This is why you should exploit external libraries written by dedicated people. They have found the good algorithms.



## Unit testing

Test each function individually

Test each function's common use

“edge cases”

bad data handling

Catch your bugs *early* !

Extreme version: “**Test Driven Development**”

Now we move on to another aspect of generic good practice, but another one that comes from splitting a program into a structured collection of lumps.

If we have split our common actions into functions we can test those functions individually or in small groups. This is called “unit testing”.

If you write a function to sum  $x^2$  over 100 values you ought to write a test that feeds it 1 a hundred times. Do you get 100? Or do you get 99 because you have the end condition slightly wrong?

If you get into the discipline of testing functions as you go along, then you will save yourself an enormous amount of debugging time towards the end of the program writing. But it does require discipline in the short term.

# Revision control

Code “checked in” and “checked out”

Branches for trying things out

Communal working

**Reversing out errors.**

Source code for most programs (but not all, e.g. spreadsheets) consists of plain text.

Revision control lets you say that “this is version X of this file” (or this set of files). You can then say “go back to version 5” or “show me version 2”. Some systems support locking where you can say “I’m working on this file; nobody touch it.” Other don’t let you lock but merge back the changes made by people in different parts of the same file.

Revision control systems work over multiple computers too. This lets lots of people all work on the same project.

Another standard facility is to split off a “branch”. If you want to experiment with some changes you start with version 5 say and then create version 5.1, 5.2 etc. rather than 6 and 7. At the end of the experiment, if you like what you have got, you merge your final version 5.x back, or you simply drop it and return to the original version 5.0 and start again.

Perhaps most importantly if you do build a version 6 and realise you have it completely wrong you can abandon it and return to version 5!

## Revision control

Two main programs: `subversion`



`git`



Starting from scratch? `git`

Something in use already? Use it!

**GitHub**

`github.com` free repository (for open source)

`try.github.io` free online training

There are two current revision control systems: `subversion` and `git`.

The more recent program, `git`, was designed by Linus Torvalds when the `subversion` program could no longer cope with his software project (the Linux kernel).

















If you have an existing revision control system, use it, whatever it is.

If you are starting from scratch, use `git`.

`git` has an additional advantage. If you are prepared to work in an open environment (anyone can join in) on open source software there is a company, GitHub, that will give you a `git` repository free. (And if not, then they're not too expensive.) Better still, they will even teach you how to use `git`!

# Integrated Development Environment

“All in one” systems: necessarily quite complex

	Eclipse	Most languages	  
	Visual Studio	C++, C#, VB, F#, ...	
	XCode	Most languages	
	Qt Creator	C++, JavaScript	  
	NetBeans	Java	  

Often sitting in front of a revision control system modern programmers often (typically?) use an Integrated Development Environment (“IDE”).

These combine editors that help you with specific languages, guides to other parts of your programs or system libraries, revision control systems, build systems to see if your program actually compiles, and test systems to run your unit tests. (You did write unit tests, didn’t you?)

By far the most commonly used IDE at the moment is Eclipse. In the pure Microsoft world Visual Studio is a close second and the pure MacOS world tends to use XCode.

# make — the original build system

Command line tool `$ make target`

Dependencies `target ← target.c`

Build rules `cc target.c -o target`

`Makefile`

Used behind the scenes by many IDEs

But the grand-daddy of all build systems outstrips the IDEs thousands to one. (In fact most IDEs use it behind the scenes.)

The make program records how one file depends on another, so that if you update one file make knows which other files now need to be rebuilt. It also stores information about how to rebuild most types of file.

The configuration files for make, known as `Makefiles` are a little strange.

The strangeness harks back to the creation of the first make in 1977 by Stuart Feldman of Bell Labs. The (potentially apocryphal) story is this: Feldman knocked together a quick version of make which used lots of short cuts and dirty tricks letting him write a prototype quickly. He let some people use it and went home for the night. It proved so popular that when he returned to work the following morning it was in use in so many projects that any change of specification was vetoed by his colleagues.

# Building software courses

Unix:  
Building, Installing and Running Software

# Course outline

Basic concepts

Good practice

**Specialist applications**

Programming languages

# Specialist applications

Often no need to program

Or only to program simple snippets

All have **pros** and **cons**

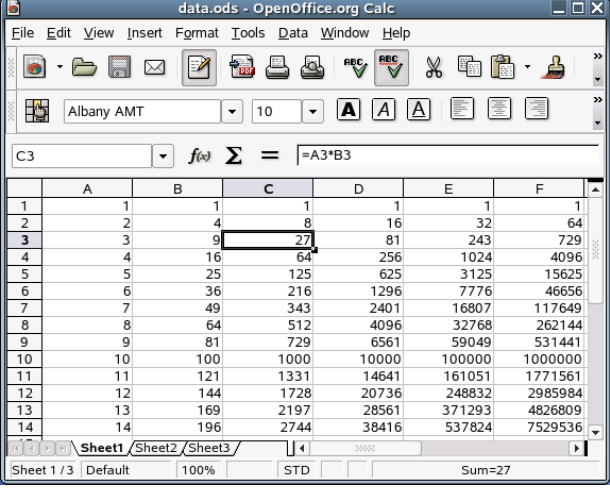


# Spreadsheets

Microsoft Excel

LibreOffice Calc

Apple Numbers



The screenshot shows the OpenOffice.org Calc application window titled "data.ods - OpenOffice.org Calc". The menu bar includes File, Edit, View, Insert, Format, Tools, Data, Window, and Help. The toolbar contains various icons for file operations and editing. The spreadsheet area displays a table with columns A through F and rows 1 through 14. The formula bar shows the active cell C3 containing the formula  $=A3*B3$ . The table contains the following data:

	A	B	C	D	E	F
1	1	1	1	1	1	1
2	2	4	8	16	32	64
3	3	9	27	81	243	729
4	4	16	64	256	1024	4096
5	5	25	125	625	3125	15625
6	6	36	216	1296	7776	46656
7	7	49	343	2401	16807	117649
8	8	64	512	4096	32768	262144
9	9	81	729	6561	59049	531441
10	10	100	1000	10000	100000	1000000
11	11	121	1331	14641	161051	1771561
12	12	144	1728	20736	248832	2985984
13	13	169	2197	28561	371293	4826809
14	14	196	2744	38416	537824	7529536

You might not think of spreadsheets as “programming” but spreadsheets embed single commands, which the program interprets, in amongst the data. They are a legitimate interpreted programming system. They simply hide it well.

There is only one spreadsheet most people are familiar with: Microsoft Excel, part of the Microsoft Office suite of programs. There is a free equivalent, Calc, which is part of the LibreOffice.org suite of free Office tools. They are equivalent enough for most purposes so long as the functions don't invoke too complex a set of macros. Apple's Numbers component of iWork also provides a spreadsheet but is not particularly similar to Excel.

# Spreadsheets

Taught at school

Easy to tinker

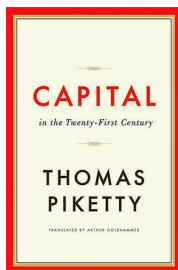
Easy to get started

Taught *badly* at school!

Easy to corrupt data

Hard to be systematic

*Very hard to debug*



Example:  
Best selling book,  
buggy spreadsheets!

This format does not lend itself well to revision control and shared working as everything is stored in a single, binary format file.

It is also very easy to corrupt data in a spreadsheet and notoriously hard to debug problems.

A recent, high-profile example of this came from the book “Capital in the Twenty-First Century”. This was a politically charged book that made some very significant claims regarding the concentration of money in the hands of “the 1%”. Unfortunately, the author based his calculations in Excel and made mistakes. When the mistakes were corrected many of his conclusions vanished.

There is, incidentally, a moral in this tale beyond “don’t use spreadsheets for anything important”. When you are debugging a program it is not enough to stop looking for bugs when you get answers you agree with.

# Excel courses

Excel 2010/2013:

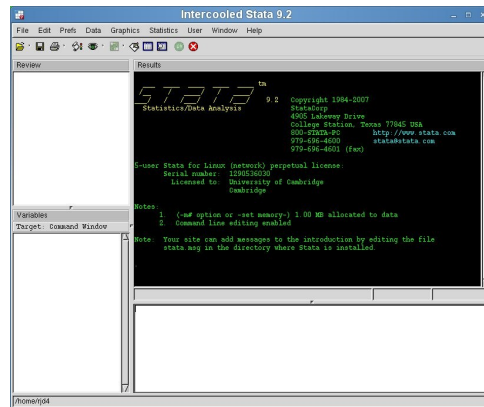
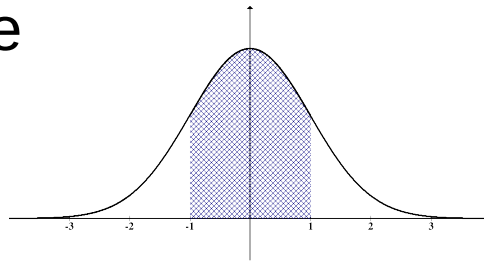
Introduction

Analysing and Summarising Data

Functions and Macros

Managing Data & Lists

# Statistical software



There are three big players in statistics packages in Cambridge: Stata, SPSS and R. Stata is the big commercial player (with SPSS a close second.) R is the free package.

# Statistical software

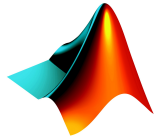
Stata: Introduction

R: Introduction for Beginners

SPSS: Introduction for Beginners

SPSS: Beyond the Basics

# Mathematical manipulation



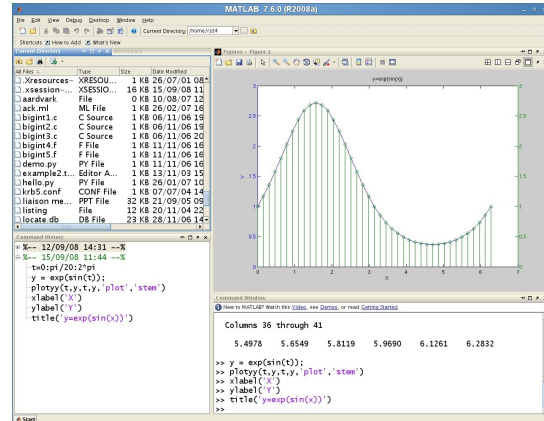
Matlab



Octave



Mathematica



There are packages for helping with mathematical manipulation and casual graphing of interim results. The two big players in the Cambridge environment are MATLAB and Mathematica with MATLAB tending to dominate over Mathematica. There is also a free product called Octave which seeks to be a clone of Matlab.

We tend to recommend people avoid Mathematica. While it seems deceptively easy to use at first there is no consistency to it. With Matlab and Octave, once you know the way to do a few tasks the rest seem intuitive.

# Mathamtical software courses

Matlab:

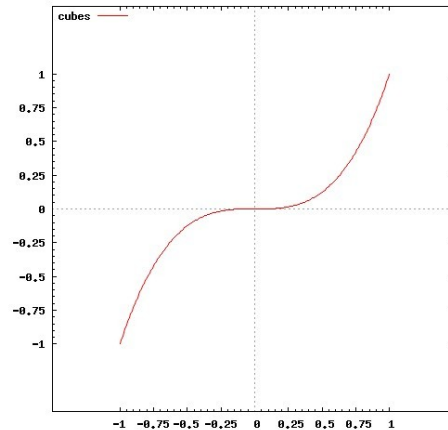
Introduction for Absolute Beginners

Linear Algebra

Graphics (Self-paced)

# Drawing graphs

Manual or automatic?



If you want to want to plot graphs based on the output of your programs you will need some sort of plotting package. The “bad way” to do this is to take a graphics library (in Fortran or C, both exist) and to bolt some graphics code into your numerical program. The right way is to have your numerical program produce its results and then write a distinct graphics program in a graphics-specific language or package. Alternatively you can import the data into a purely manual graphics package and fiddle to your heart's content. I assume you will have better things to do with your time and just want a program to create a graph glued to the other bits of your project. This is what I mean by “automatic” rather than “manual”.

A very common approach is to export as comma separated value files (CSV) and to then create graphs in Excel or another spreadsheet.

There are two dedicated graphics languages: gnuplot and ploticus. Both are available on the MCS. In addition there is a graphics module for Python called matplotlib.

Note that even if your main program is written in Python and you want to use the Python graphical module we still advise that you split the two tasks — creating your data and graphing your data — into two separate programs.



# Courses for drawing graphs

Python 3:  
Advanced Topics  
(Self-paced)

(includes a  
matplotlib unit)

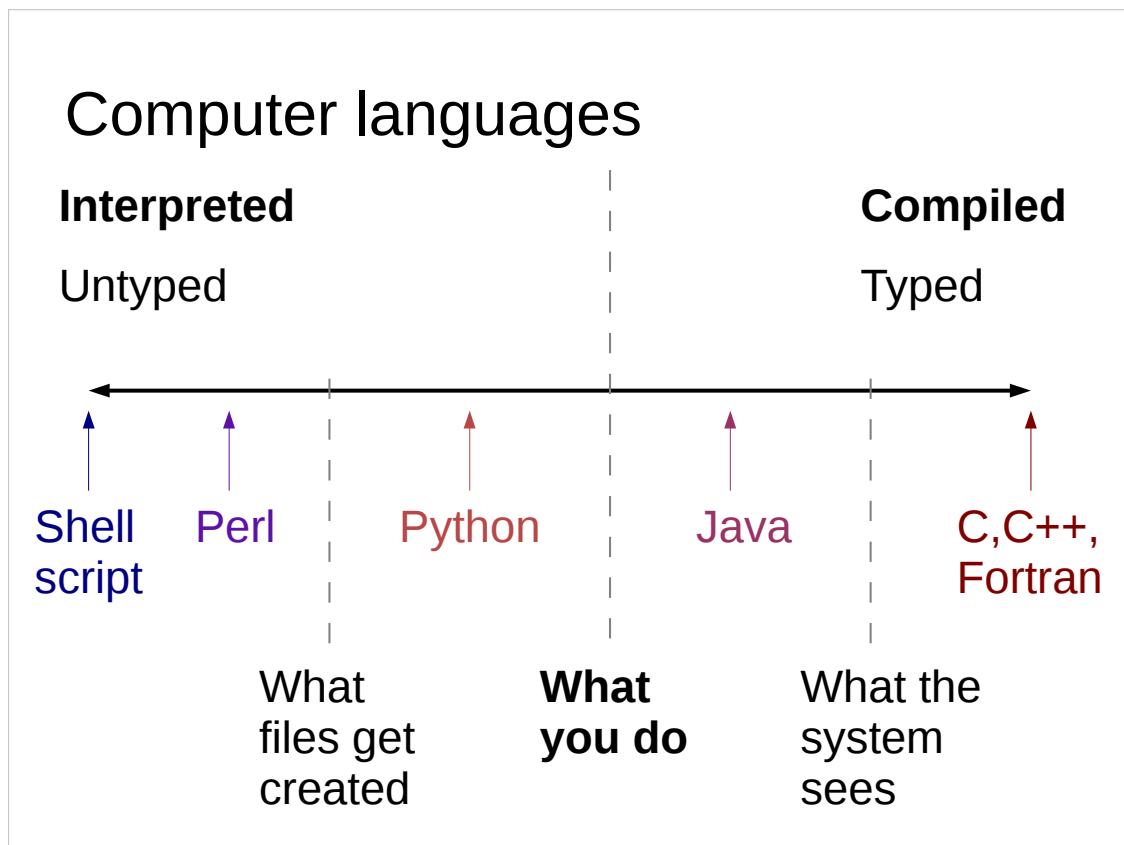
# Course outline

Basic concepts

Good practice

Specialist applications

**Programming languages**



Programming languages are traditionally split into two camps: “compiled languages” which are converted from plain text to machine code which is then run by the computer, and “interpreted languages” where the plain text is read, line by line, by an “interpreter” which then issues machine-level commands on the script's behalf. But reality is less simple than that and they actually form a spectrum.

At one end we have shell scripts which are pure interpreted languages. Perl is essentially the same, though internally the Perl text is converted into a condensed “byte code” which is then interpreted. Python goes a stage further and writes out the byte code for later re-use, creating a file thing.pyc for each text file thing.py. (The “c”, confusingly, stands for “compiled”.) However from the user's perspective this all happens automatically and there is no need to be aware of the conversion.

With Java there is an explicit user step where the user compiles the Java source code (thing.java, say) into a pseudo-machine code file (thing.class). This contains code for a fictitious CPU emulated by the Java run-time system which essentially interprets the Java byte codes. From the user's perspective there is a compilation phase, even though what is produced is not native machine code.

Finally there are the true compiled languages like C, C++ and Fortran. With these languages the compilation phase generates native CPU instructions, true machine code, which can be passed directly to the computer.

# Shell script

## Suitable for...

**gluing programs together**

“wrapping” programs

small tasks

Easy to learn

Very widely used

## Unsuitable for...

performance-critical jobs

floating point

GUIs

complex tasks

The shell is the fundamental interpreted language. The commands you type at the command line are interpreted by the shell and acted on. Similarly we can put those commands in a file and have the shell interpret them from that.

Shell scripts are the classic “glue” for holding together a set of programs. If you have a set of programs which can be run from the command line and which have to interoperate then a shell script is what you want to use.

They can also be used for “wrapping” programs. This lets you run programs with your default parameters, or in a certain environment, without having to manually set each parameters manually or change your environment manually each time you run it.

Shell scripts can also be used to run certain small tasks themselves. So long as the task is very simple, and stays very simple then this is OK. Small scripts like this have a habit of growing with time, though, and very soon you end up in a situation where you should be using one of the more powerful scripting languages we will meet later.

Shell scripts are not suitable for computationally intensive work (though they can call other programs that are, of course) and they are not suitable for writing GUIs in (though people have tried).

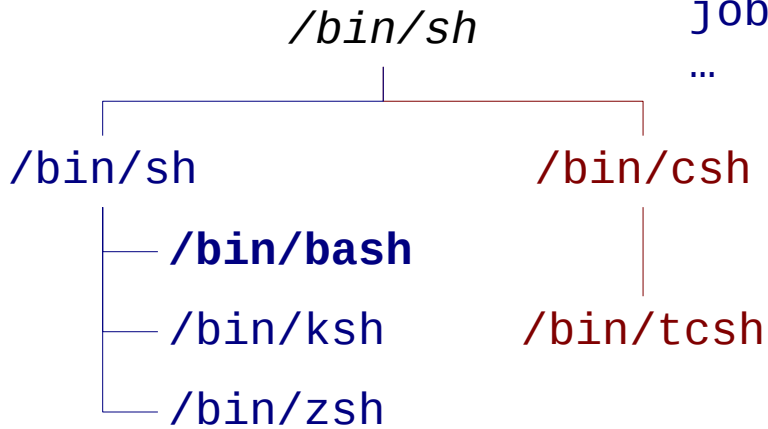
# Shell script

Several “shell” languages:

```
#!/bin/bash
```

```
job="${1}"
```

```
...
```



There are many shells. The only rational one to choose is **bash**, the **B**ourne **a**gain **s**hell, which is a play on the name of Simon Bourne who wrote one of the very early shells.

The most important schism is between the “C-shell” and the “Bourne shell” shells. Avoid C-shell; it’s dying.

# Shell scripting courses

Unix:

Introduction to the Command Line Interface  
(Self-paced)

Simple Shell Scripting for Scientists

Simple Shell Scripting for Scientists  
— Further Use

“Further ~~shell~~ scripting”?

**Python!** 

A word of caution is advisable here. We teach quite a bit of shell scripting in the UCS course, but not all of it. If you ever find yourself looking for an advanced shell scripting course then our advice is that you have left the arena where shell script is the right tool for the job. We would recommend Python as a better alternative.

Just because the shell *can* do a bit more doesn't mean that you should use it for that. So this leads us on to the more powerful scripting languages...

# High power scripting languages

Python

```
#!/usr/bin/python
```

```
import library
```

```
...
```

Perl

```
#!/usr/bin/perl
```

```
use library;
```

```
...
```

Both have extensive  
libraries of utility functions.

Both can call out to libraries  
written in other languages.

The shell, which we saw in the previous slides, was designed for launching other programs rather than being a programming language in its own right. We will now turn to the two primary scripting languages that were designed for that purpose: Python and Perl.

Again, neither is directly appropriate to computationally very intensive work but both can make use of external libraries that have been written in other languages that are. Python, in particular, has developed a major following in the scientific community and is no slacker for medium scale problems.



# Perl

The “Swiss army knife” language

Suitable for...

Bad first language

**text processing**

data pre-/post-processing

Very easy to write  
unreadable code

small tasks

CPAN: **C**omprehensive  
**P**erl **A**rchive **N**etwork

“There's more than  
one way to do it.”

Widely used

Beware Perl geeks

Perl was written to be a replacement for the text manipulation programs `sed`, `awk` and `grep`. These were simple tools designed for specific sorts of text manipulation “in line”. They would typically sit in a pipe line of commands and filter the data as it flowed past, a line at a time. Perl can be used for all that but a whole lot more besides.

Perl has a very extensive support library supplied by the Comprehensive Perl Archive Network (CPAN). Most of it does not come installed by default but has to be added as and when you need the components. The problem is that there are a lot of interdependencies between the elements of the CPAN library and if you try to add one you find yourself importing a whole stack of them. There is a utility called `cpan` to assist with this but it is still far from adequate. (You can find the archive at <http://www.cpan.org/>.)

Perl is suitable for simple text processing but is not suitable to learn as your first serious programming language. It is infamous for “write once read never” code that is quite illegible to anybody other than the person who first wrote it and is hard work even for him or her after six months. Perl takes pride in its slogan that “there's more than one way to do it”; any task can be tackled by Perl in many different ways. Unfortunately, if the author of a Perl script knew one way and the reader of the script knows another the reader will have problems understanding just what the script does. Perhaps because of its hostility to the casual reader, Perl has attracted the worst sort of geeks who take a perverse pride in writing dense, wholly impenetrable Perl code. At least it keeps them off the streets.

<b>Python</b>	<b>“Batteries included”</b>
Suitable for... text processing	Excellent first language
data pre-/post-processing small & large tasks	Easy to write maintainable code
Built-in comprehensive library of functions	The “Python way”
Scientific Python library	Very widely used
	Code nesting style is “unique”

The other powerful scripting language we will discuss is Python. Python was written after Perl became widely used and has the benefit that its author learned from Perl's mistakes. Despite being more recent it has caught up and is now very common in Cambridge and the scientific community worldwide, overtaking Perl.

Python is also very easy to learn and we recommend it as a first programming language.

It comes with its own fairly extensive libraries which give it the slogan “batteries included”. Most of what you need for general computing comes with the language.

In addition the scientific community has built the “Scientific Python” (SciPy) libraries which are in turn built on top of the “Numerical Python” (NumPy) libraries which provide very efficient array-handling routines (written in a language other than Python).

You can learn all about SciPy at <http://www.scipy.org/>.

Python lends itself very naturally to writing well structured and manageable code. It has a style of code that is unique and which puts off some people but it's easily dealt with in the editor. The issue is that where most languages use open and closed brackets to clump instructions together, Python uses levels of indentation.

# Python courses

Python 3:  
Introduction for Absolute Beginners

Python 3:  
Introduction for Those with  
Programming Experience

Python 3:  
Further Topics  
(self paced)

# Compiled languages

No specialist system and scripts are not fast enough

Library requirement with no script interface

Use only as a last resort

Compiled language

C

C++

Fortran

Java

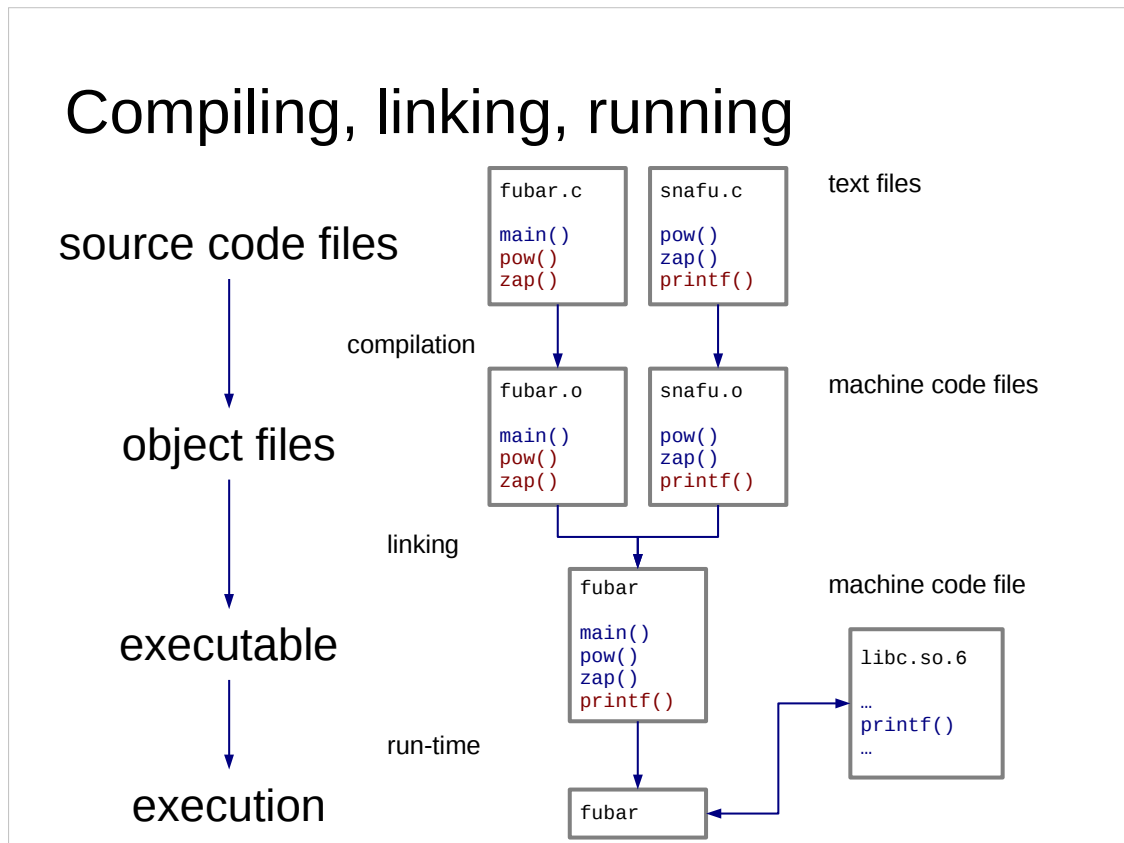
Then there are the compiled languages.

It's perhaps slightly unfair to categorise them as "last resorts" but they do require more effort to write in and are more trouble learning than the others.

So if there is no appropriate specialist system and the Perls and Pythons of this world aren't fast enough, or if you need to use a library written in a compiled language that cannot be accessed through a simpler scripting language, then you may have to use a compiled language.

I will cover three "true" compiled languages here and also Java because from the point of what you have to do there is an explicit compilation stage.

# Compiling, linking, running



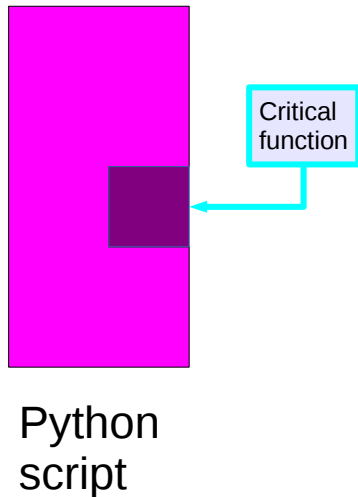
I'll use C as the example in these slides, but the same applies for C++ and Fortran.

We start with the source code (typically multiple files).

Compilation proper consists of taking the individual plain text source files and turning them into machine code for the computer. Each source file, `fubar.c` say, is individually converted into a machine code (or "object code") file called an object file, `fubar.o`, which implements exactly the same functionality as the source code file. Any function calls in the source code are translated to function calls in the machine code. If the function's content isn't defined in the source code then it's not defined in the machine code. And so it goes on. This is a pure "translation" process; source code is translated, file for file, into machine code.

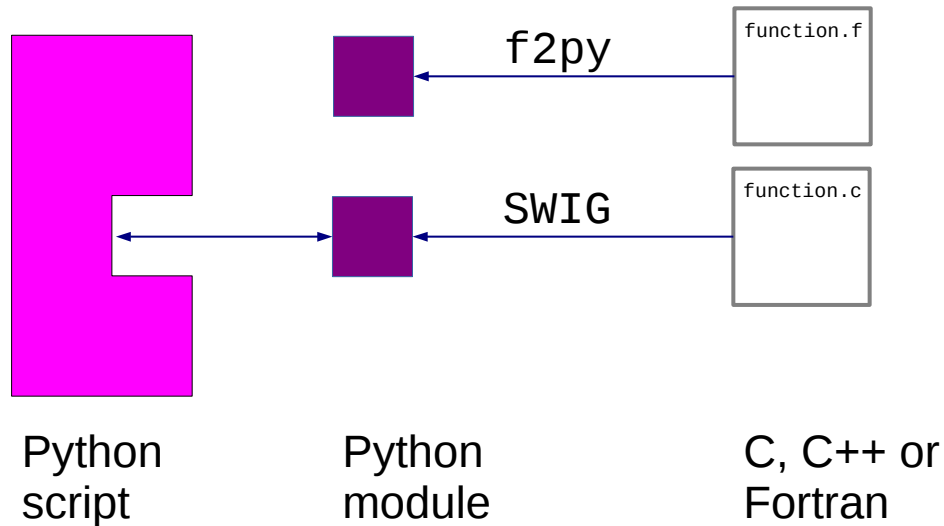
The next stage is called "linking". This is the combination of the various machine code files into a single executable file. The function definitions defined in the various object files are tied together with their uses in other object files. Calls to functions in external libraries are tied to the file containing the library so that, at run-time, the operating system can hook those function definitions in too.

## No need to compile whole program



Also note that if you don't need to write your entire program in a compiled language just because you need to write part of it that way. For example there are hooks to call Fortran routines from Python and Python objects which can be manipulated by Fortran. Many of the support libraries for Python are written in languages other than Python.

## No need to write the whole program in a compiled language



If there is a numerically intensive section in your program by all means write it in C or Fortran. But don't drag the rest of the program with it. There are tools which take C, C++ or Fortran code and create machine code libraries that act as Python libraries (called “modules” in Python-speak).

# Fortran

The best for numerical work

Excellent numerical libraries

Unsuitable for everything else

Very different versions:

77, 90, 95, 2003

For numerical work there's Fortran. There still is no comparison; if you are doing numerical work you are best off using Fortran. The best numerical libraries are written in Fortran too.

However, it is probably the wrong choice for more or less anything else.

You also need to be careful about the various different versions of Fortran. For a long time Fortran 77 was the standard. Now we tend to use a mix of Fortran 90 and Fortran 95. Fortran 2003 has yet to make a serious impact.



# Fortran course

Fortran:  
Introduction to Modern Fortran

Three full days

# C

The best for Unix (operating system) work

Excellent libraries

Superseded by C++ for applications

Memory management

The C programming language made its name by being the language used to write the Unix operating system. As a result it is the best of the compiled languages for interfacing with the operating system. Because it is the language for an operating system used by developers a very large number of libraries and programs have been written in it.

Arguably it has been superseded for application programming by C++ but it is still very widely used.

The most important problem with C is the issue of “memory management”. In C you are required to explicitly “free” objects that you no longer need to return their memory space allocation to general use. Programs that don't do this suffer from “memory leaks” and tend to grow with time. Once they get too big for the system running them they become slow as the system has to compensate for the amount of memory they claim to need. Finally they collapse. Alternatively, programmers can accidentally free memory that the program actually does require. These programs tend to die suddenly. It's also possible to point accidentally to the wrong part of memory and get nonsense results back.

All these memory management issues can be handled with careful programming, but the language offers no assistance of its own.

# C++

Extension of C

Object oriented

Standard template library

General purpose language

*Very hard to learn well*

Strictly speaking C++ is an extension of C. However, it should be approached as an entirely different language. “Writing C in C++” is a classic mistake.

C++'s extension over C is that it implements “object oriented” programming. Think of objects as particularly powerful “lumps” of your program. However, using objects is a whole extra skill that has to be learnt.

C++ also comes with a particularly useful library called the “standard template library” which allow these objects to be manipulated in various ways. Because this library has been written by experts it typically forms a very useful resource to avoid you having to code the methods yourself.

All told, C++ is a decent general purpose language.

The downside, however, is the C++ is a *huge* language. It also has a serious number of gotchas including its own style of memory management problems.

C++ is easy to learn the basics of but very hard to learn well. To quote Bjarne Stroustrup, the creator of C++, from the introduction to his book: “How long will [learning C++ from scratch using this book] take? ... maybe 15 hours a week for 14 weeks.” (Stroustrup, Bjarne (2008). Programming: principles and practice using C++.) That's an hour a day for every working day in 42 weeks!

## C++ books

### **“Thinking in C++, 2nd ed.”**

Eckel, Bruce (2003)  
(two volumes: 800 and 500 pages!)

### **“Programming: principles and practice using C++”**

Stroustrup, Bjarne (2008)  
harder but better for scientific computing

## From the intro to Stroustrup's book

“How long will [learning C++ from scratch using this book] take? ...  
maybe 15 hours a week for 14 weeks.”

# C++ course

C++:

Programming in Modern C++

12 lectures, 3 terms,  
significant homework

Uses Stroustrup's book

# Java

Object oriented

General purpose language

Much easier to learn and use than C++

Some poorly thought out libraries

Multiple versions:      Use  $\geq 1.6$   
1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7

Finally I'll talk about Java. This, like C++, is a good general purpose language and is much easier to learn and to use. It implements automatic memory management so those difficulties are gone too.

Because it is implemented as a byte-code interpreter, interpreting the code generated by the supposed compiler, its compiled files work across all platforms with at least the particular version of the Java runtime system.

Some of its libraries aren't particularly well thought out, however, and there is a good deal of difference between the various versions of the language, though the Java maintainers do guarantee back-compatibility. If you stick to versions 1.6 or later you should do OK.

# Java courses

Object oriented programming    CL lectures

(also classes,  
ask at the CL)



# Scientific Computing

[training.cam.ac.uk/ucs/theme/scientific-comp](http://training.cam.ac.uk/ucs/theme/scientific-comp)

[scientific-computing@ucs.cam.ac.uk](mailto:scientific-computing@ucs.cam.ac.uk)

[www.ucs.cam.ac.uk/docs/course-notes/unix-courses](http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses)