

Introduction to Scientific Computing

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

Bob Dowling
University Computing Service
rjd4@cam.ac.uk

www-uxsup.csx.cam.ac.uk/courses/SciProg



This course is the first in a 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.

1. Why this course exists

2. Common concepts and general good practice

Coffee break

3. Selecting programming languages

UCS

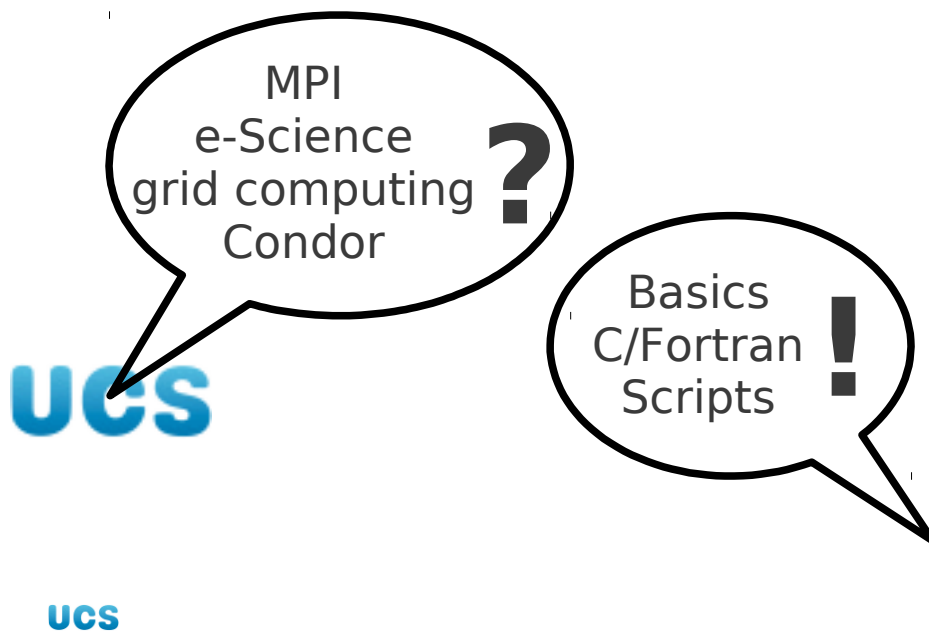
This afternoon is split into two one hour blocks separated by a coffee break.

The first block will talk a little bit about computer programs in general and what constitutes “good practice” while writing them. It also describes some of the jargon and the other problems you will face.

The second will talk about the types of programming language that there are and help you select the appropriate ones for your work.

Throughout, the course will point to further sources of information.

Why does this course exist?



I should explain why this course exists. A couple of years ago a member of the UCS 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.

Common misconceptions

Any program can be copied from one machine to another.

Every line in a program takes the same time to run.

Programs with source code have to be built every time they're run.

Everything has to fit in the same program.

UCS

Here are some of the misconceptions we encountered while we were asking our questions.

Perhaps the hardest misconception to address was that everything needs to fit into a single program.

Common concepts and good practice

UCS

So let's get started.

I can't split “common concepts” and “good practice” into two cleanly separated units. Concepts lead to good practice and vice versa.

A simple program

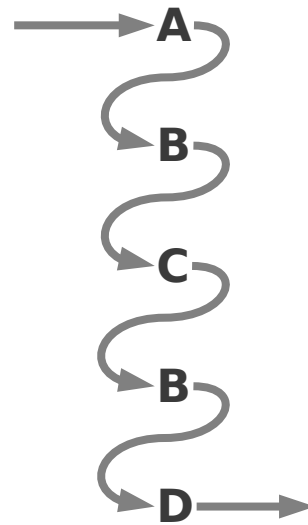
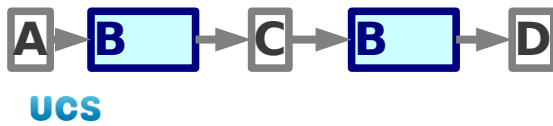
Linear execution



Not constant speed



Repeated elements



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.

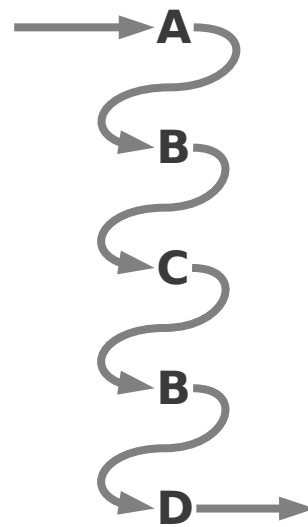
Lines in a script

Chunks of machine code

Calls to external libraries

Calls to other programs

UCS



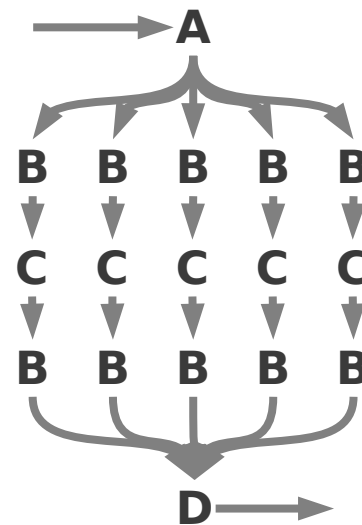
So what are these instructions? Well, it depends.

They might be individual Unix commands typed into a text file (a “shell script”). They might be individual machine code instructions working at the very lowest levels in the computer. We will see everything in the entire spectrum over the course of this afternoon.

“Parallel” program

Much harder
to program

Single
Instruction
Multiple
Data

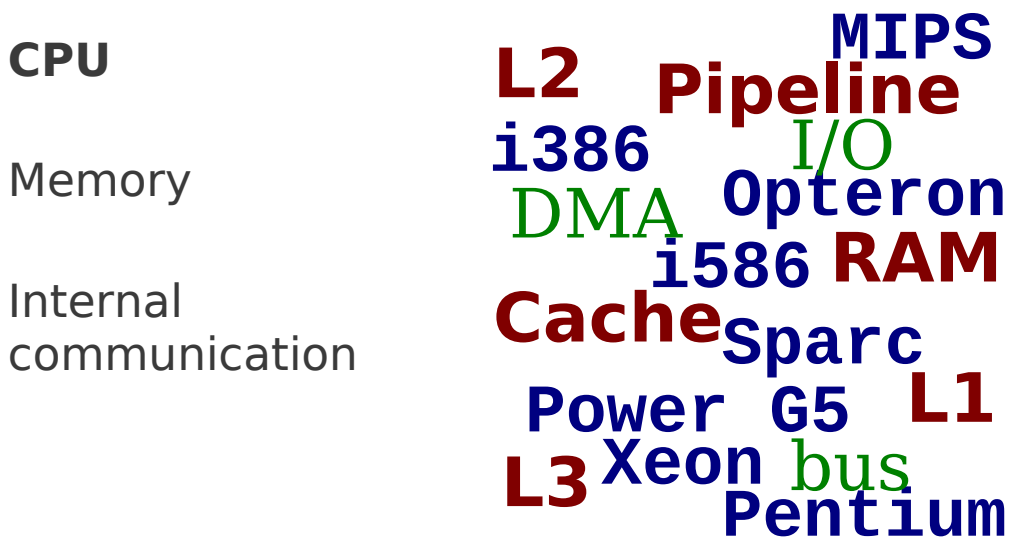


UCS

But programs need not be linear. In the past ten years or so scientific computing has moved into “parallel programming” where computers with multiple processors can run the program through many of them simultaneously or multiple computers can be made to work in parallel. It's also possible to run multiple “threads” of control on a single processor. This is typically much harder to program correctly and requires specialist skills.

The most common form is called SIMD (pronounced “simm-dee”) which stands for “single instruction, multiple data”. In this example the block of code “A” needs the same work done on five different pieces of data. These are run in parallel and then the “D” block of code glues the five steps of results together.

Machine architectures

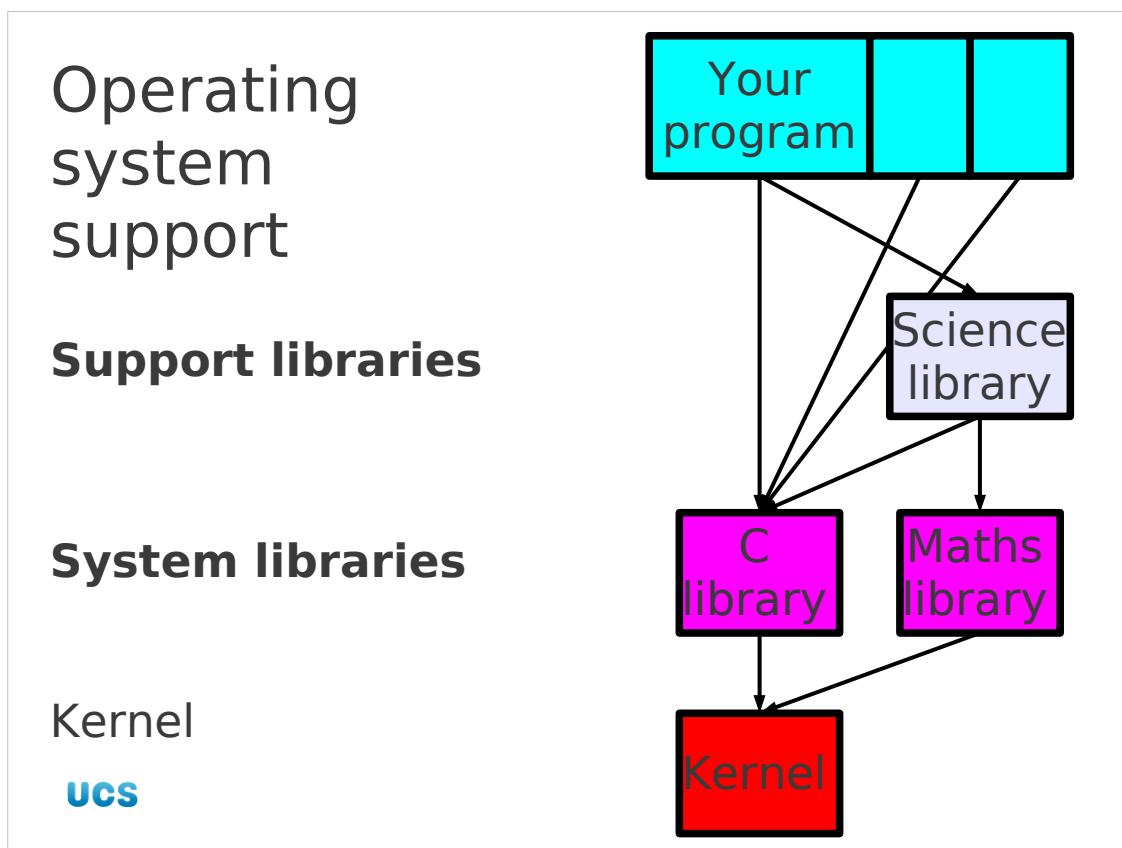


UCS

There is one issue that we must be aware of. Not all computers are the same. They can differ in terms of the core central processing unit (CPU) and so speak different machine code languages, they may have different memory systems so the mechanisms to access bits of data will vary from machine to machine. They may differ in the nature of the support systems that they provide for programs to use.

A program that worked on one system may not work on the other. More subtly, a program that worked well on one machine may perform better or worse on another.

Some sorts of programs (typically the ones compiled into machine code) are inherently non-portable; the machine code that is to be run needs to be rebuilt for each different type of computer. The CPU is the most important component of a machine architecture for this reason. Other sorts of programs more such as shell scripts work perfectly well as they are transferred from one system to another.



On top of the hardware lies the operating system. At its heart there is the “kernel”, the ultimately privileged program which controls all access to the system.

Typically user programs use routines stored in software libraries to talk to the kernel more directly. The “operating system” can be thought of as the collection of kernel and low-level support libraries. Your applications may bring in libraries of their own which don't talk directly to the kernel.

(Note that this is a deliberately restrictive definition of “operating system”.)

“Floating point” problems

e.g. numerical simulations

Universal principles:

$0.1 \rightarrow 0.1000000000001$

and worse...

**“Program Design:
How computers handle numbers”**

UCS

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. If this matters to you the UCS offers a course on this subject.

<http://www.training.cam.ac.uk/ucs/course/ucs-numbers>

<http://www-uxsup.csx.cam.ac.uk/courses/Arithmetic/>

Other problems

e.g. text searching
sequence comparison

$\text{^f.*x\$}$ → firebox
fix
fusebox

**“Pattern matching using
Regular Expressions”**

“Python: Regular Expressions”

UCS

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 expressions aren't just toys for cheating at crosswords; chemists and biologists have found them very useful.

We run two courses in searching texts using systems called “regular expressions”. The general course goes further into the details of more powerful regular expressions, the Python-specific course burrows deeper into how Python can use them and the Python-specific extensions.

<http://www.training.cam.ac.uk/ucs/course/ucs-regex>

<http://www-uxsup.csx.cam.ac.uk/courses/REs/>

<http://training.csx.cam.ac.uk/ucs/course/ucs-pythonregex>

<http://www-uxsup.csx.cam.ac.uk/courses/PythonRE/>

Split up the job

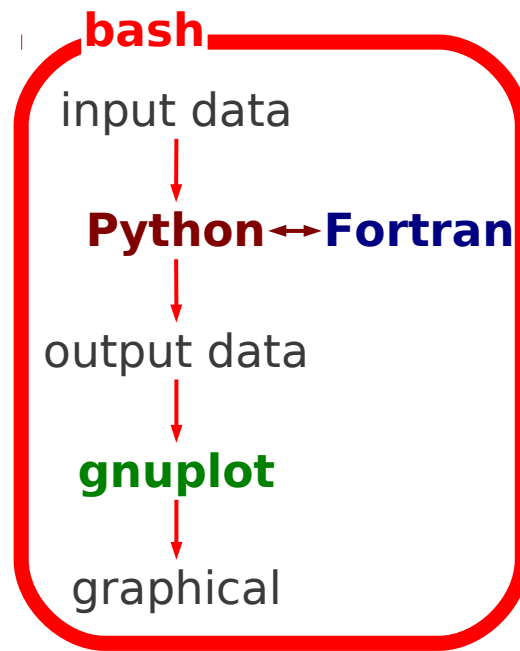
Bits of the job



Different tools
&
“Glue”

**“Divide and
conquer”**

UCS



So let's look at the first, and most important, “good practice”. This corresponds to the misconception that everything needs to be run from a single program.

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.

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.

For example you might split your task into one part that reads in the input data and which calls a set of functions written in a different programming language to process them and then spits out some output data. That program might be written in a scripting language for most of its logic but call a function written in a different language for its numerical work. Another program then takes the output data and turns it into a graph. Wrapped around all this might be a simple shell script that says “run the first program from the input data” followed by “create the graph from the output data”.

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

Choose a suitable tool for each bit

What tools?

How to pick the right one?

Pros & Cons



FORTRAN
MATLAB
Perl
Java
bash
gnuplot
Python
C++
Excel
SPSS

UCS

The hard bit is to know what is the suitable tool for each bit of the task. The function of this course, and particularly its second half, is to take you through the set of tools, explaining what they are suitable for.

“Glue”

Splitting up
↑ ↓
Gluing together

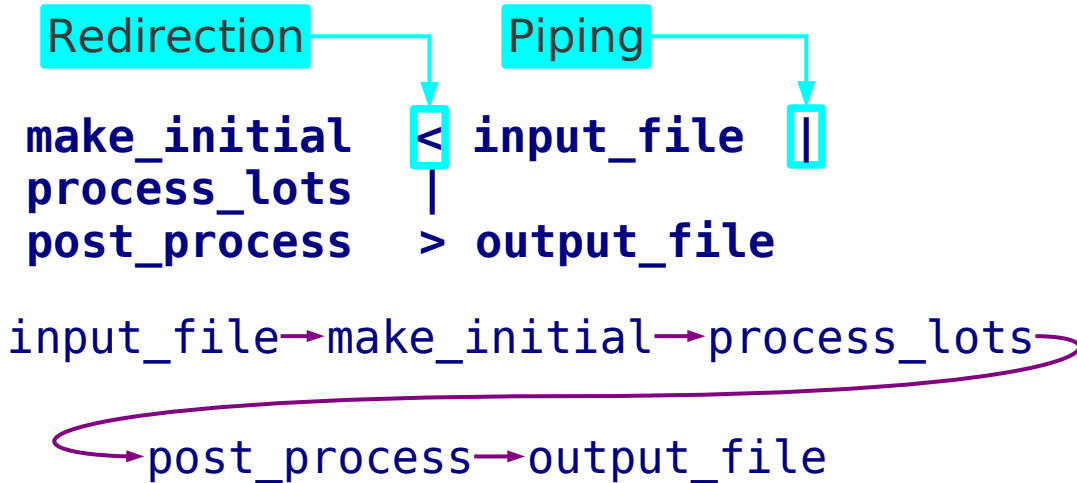
1. Pipeline
2. Shell script
3. GUI

UCS

We mustn't forget the glue. While we split a task up we still need to take the sub-tasks and recreate the original job.

There are three common ways to do this. We will quickly review them.

Pipeline



Course: **“Unix: Introduction to the
UCS Command Line Interface”**

If your task is easily split into a simple chain of tasks, each of which needs to pass its results on to the next program, then the simple Unix command line may be all you need. The Unix concept of “piping” lets you specify a list of commands, the first of which reads from a file or the keyboard, the last of which writes to a file or the screen and every one in between simply passes its results on to the next command in the list.

For example, if I was doing an N-body model:

- the `user_input` file might be a three line file containing the number of particles, their mean masses and their mean velocity,
- the `make_initial` program might be a program that reads in those three lines and spits out a file carrying the parameters for one randomly generated particle on each line,
- the `process_lots` program might evolve this system forwards for a time interval, and then spit out their final equivalent parameters, and
- the `post_process` program might determine the mean velocity of the particles passed to it.

If this is sufficient for you then the Unix introductory course will teach you everything you need to know about how to string commands together.

<http://www.training.cam.ac.uk/ucs/course/ucs-unixintro1>

<http://www-uxsup.csx.cam.ac.uk/courses/UnixCLI/>

(Multiple instances over the year. Some are self-paced, others not.)

Shell script

```
#!/bin/bash -e

job="${1}"

if [ ! -f "${job}.dat" ]
then
    make_initial < "${job}.in" > "${job}.out"
fi

while work_to_do "$
do    {job}.dat"
    process < "${job}.dat" > "$
{job}.new"
done && "${job}.new" "${job}.dat"

post_process < "${job}.dat"
```

Course: **“Simple shell scripting
for scientists”**

ucs

The next level up on the simple command line is the shell script. This is basically a file containing the instructions we were previously typing at the command line. However now that we have a file we can add a level of sophistication that is typically sufficient for all reasonable tasks.

We want to be able to make choices depending on the situation and to repeat activities until some condition is satisfied.

For example if my processing was going to take a very long time then I might want to checkpoint:

- Create an “initial state” file from an input file of parameters.
- Take a state file, process it a bit (but not necessarily to completion) and then write out the new state.
- Repeat this until it reaches the final state.
- Post process the final state to generate the desired format of output.

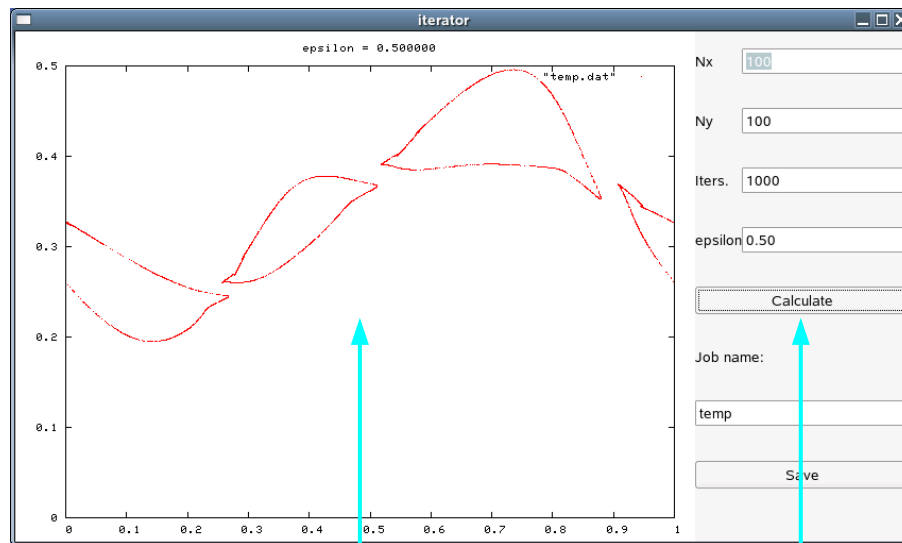
Why is that useful? Because things might go wrong. If my job takes seven days to run and the machine crashes on average every other day then I want to “checkpoint” several times a day so that when the machine crashes I can pick up where I left off and carry on rather than having to start from scratch each time.

We offer a course on shell scripting:

<http://www.training.cam.ac.uk/ucs/course/ucs-scriptsci>

<http://www-uxsup.csx.cam.ac.uk/courses/ShellScriptingSci/>

Graphical User Interface



Finally there is the graphical user interface (GUI). This can be excellent for either end of the pipeline: getting the user input or displaying the program output to the user.

Do not try to use the GUI for the intermediate processing. It should be used to fire off the heavy work in a separate process. If that process doesn't take too long to complete then the same GUI can show the output as well as the input. However, there's no rule saying that you need to use the same tool for input and output!

Writing a GUI is considerably harder than writing a shell script, but if you need to then we offer a course on a relatively simple tool for building simple interfaces called "Glade". This course isn't given any more but the notes are available on-line.

<http://www-uxsup.csx.cam.ac.uk/courses/Glade/>

“Lumps”

Splitting up
↑ ↓
Gluing together

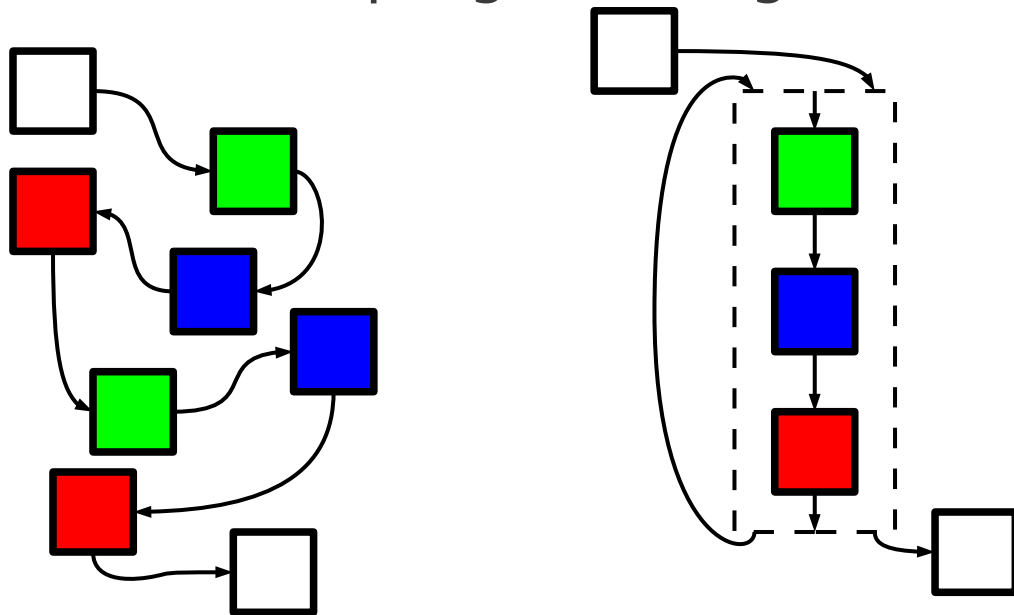
“Objects”
“Modules”
“Functions”
“Units”

UCS

We've discussed splitting up a problem into component tasks and looked quickly at how to glue the bits together.

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.

Structured programming



UCS

“Don't repeat yourself”

So we need to start looking at these “lumps”.

As with the case of splitting a task into sub-tasks and sub-sub-tasks, each of which is ultimately a program, the trick to getting a program to be tractable is to split it down further into its component “lumps”.

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 (that's the second half of the afternoon)? 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!

Never repeat 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



UCS

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.

Structured code

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

Define a function
called "norm()"

Single instance
of the code

...

```
a_norm = norm(a)
```

...

```
b_norm = norm(b)
```

...

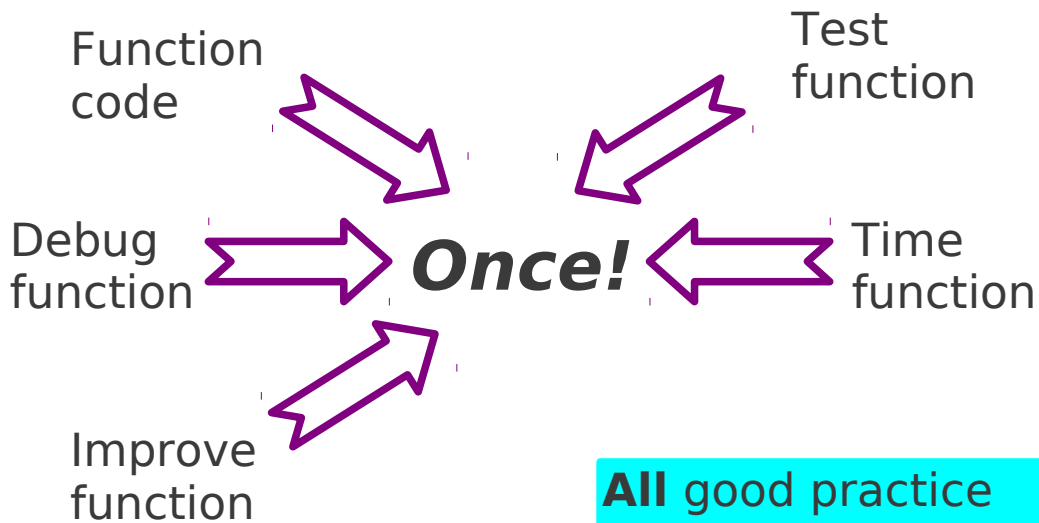
```
c_norm = norm(c)
```

Calling the
function

UCS

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 code



All good practice follows structuring.

UCS

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.

Improved code

```
def norm(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
```

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

UCS

Improve
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.

More flexible code

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

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

UCS

Improve
function



So let's improve it again!

All I've done now is to make 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.

Again, I have only had to make the change once.

More “Pythonic” code

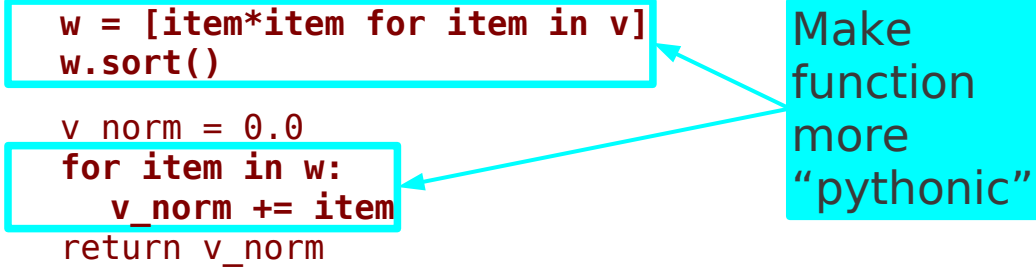
```
def norm(v):
```

```
    w = [item*item for item in v]  
    w.sort()
```

```
    v_norm = 0.0
```

```
    for item in w:  
        v_norm += item  
    return v_norm
```

Make
function
more
“pythonic”



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

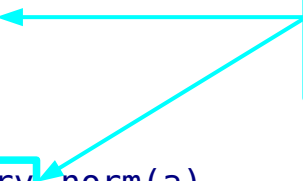
UCS

I can go further and work in the more “Pythonic” style. We're now leaving the realms of language-independent modifications but note that, regardless of the language you are writing in, if you want to make your code more “idiomatic” for that language you should structure it first and only then introduce the idioms.

Best sort of code

```
import library  
  
...  
a_norm = library.norm(a)  
...  
b_norm = library.norm(b)  
...  
c_norm = library.norm(c)
```

Get someone else to do all the work!



UCS

And now we come to the ultimate code 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.

Libraries

Written by experts

In every area

**Learn what
libraries exist
in your area**

Use them

Save your effort
for your research

UCS



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!

An example library — 1

Numerical Algorithms Group: “**NAG**”

- Roots of equations
- Differential equations
- Interpolation
- Linear algebra
- Statistics
- Sorting
- Special functions

UCS

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 for C and Fortran have functions for at least the following topics (listed according to their “chapter”):

C Roots of equations

D Differential and Integral Equations

E Interpolation, Fitting & Optimisation

F Linear Algebra

G Statistics

H Operations Research

M Sorting

P Error Trapping

S Approximations of Special Functions

(See <http://www.nag.co.uk/numeric/FN/FNdescription.asp> for details of the library.)

An example library — 2

Numerical Python: “**NumPy**”
Scientific Python: “**SciPy**”

- Large data arrays
- Vectorization
- Plotting
- Optimization
- Non-linear equations
- Partial differential equations

UCS

Alternatively, for users of scripting languages there are two sets of Python modules called “NumPy” (providing facilities for numerical work on arrays of data) and “SciPy” built on top of it to provide useful scientific functions. See http://www.scipy.org/Topical_Software for the complete set of scientific modules.

Unit testing

Program split
into “units”.

Test each unit
individually.

“Programming is like sex:
one mistake and you're
providing support for a
lifetime.”

— Michael Sinz

Catch bugs earlier.

Saves time in the long term.

UCS

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.

Debuggers

Step through
running code.

Examine state
as you go.

“Each new user of
a new system uncovers
a new class of bugs.”
— Brian Kernighan

**Better to write
debugging code
in your program!**

UCS

Another tool you may encounter is the debugger. This is a utility that lets you step through your program one bit at a time (one line at a time, one function at a time, stopping each time you get to a particular function and running line by line through just that function) and check up on your values as you go to see where it goes wrong.

A better practice to get into is to write debugging instructions into your program so that you can tell it to print out this information on its own if you want it.

The optimiser

Finds short cuts
in your code.

Leave it to the
system!

Structured code
optimises better.

“Premature optimisation
is the root of all evil
(or at least most of it)
in programming.”
— Donald Knuth

Optimisation \neq Magic wand

UCS

Another tool — one that you rarely see directly — is the optimizer. This takes your code and adapts it to perform the same operations but a bit faster, finding short cuts through the way that your code would be run on a particular system.

A common error is that people try to write hideously complex code to find short cuts for themselves. Don't. Write simple, clean code and let the optimizer do its work.

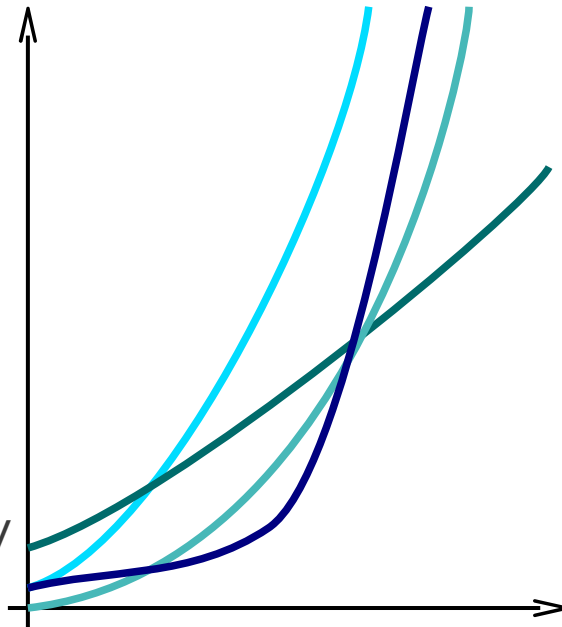
Please note that having an optimiser does not make bad code good. There's no magic wand.

Algorithms

Time taken /
Memory used

vs.

Size of input /
Required accuracy



Algorithms selected make or break programs.

UCS

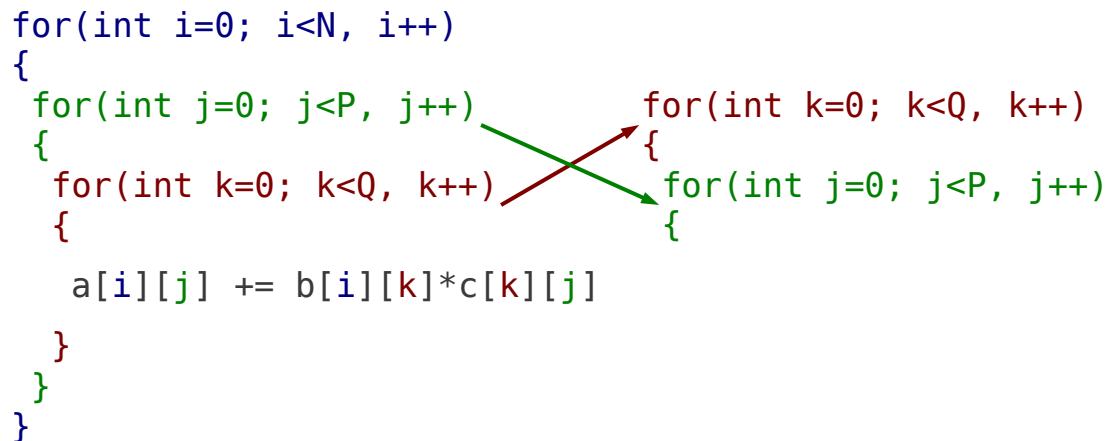
Every function implements a recipe. It generates output from input somehow. The posh name for that “somehow” is an algorithm. There are efficient ways to do some tasks and inefficient ways to do the same thing. No optimizer is going to save you from choosing the wrong way to do something.

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.

Example: Matrix multiplication

```
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]  
    }  
  }  
}
```



The diagram illustrates a transformation of a matrix multiplication loop structure. On the left, the loops are nested as `for(int j=0; j<P, j++)` followed by `for(int k=0; k<Q, k++)`. On the right, they are swapped to `for(int k=0; k<Q, k++)` followed by `for(int j=0; j<P, j++)`. A green arrow points from the `j` loop of the left snippet to the `j` loop of the right snippet, and a red arrow points from the `k` loop of the left snippet to the `k` loop of the right snippet, indicating the swap.

UCS

Here's a trivial example.

These two ways to multiply matrices differ in only one regard: I have changed 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.

Example: Matrix multiplication

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

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

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

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

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

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

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

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

ucs

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

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

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

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

Of course, neither is the most efficient. Volker Strassen's algorithm, as shown in the slide, is far more efficient but I doubt you would ever have thought of it.

So use a library!

UCS advice

escience-support@ucs.cam.ac.uk

Advice on...

libraries

techniques

algorithms

UCS

If you ever want advice on algorithms or library selection the University Computing Service does have some experts able to help you. Please feel free to contact them at the email address

escience-support@ucs.cam.ac.uk

I'll be repeating this address at the end with the rest of the contact details.

Coffee break

15 minutes

Wrists
Spine
Eyes

Caffeine addiction

UCS



Let's take a break.

We'll stop for roughly 15 minutes to let you get some air.

There is a serious point that should be made about breaks and using computers. Take short breaks often. (One minute every ten minutes is better than five or six minutes every hour.)

This will let your wrists relax which will help avoid repetitive strain injury (RSI). This can cripple you for life, so it's well worth avoiding.

Standing up and walking about also straightens your spine which helps avoid back ache.

Finally, working at a screen (visual display unit or VDU) also lowers your blink rate so your eyes dry up. Your focus is also pulled on to a screen roughly a metre from your nose. Getting away from your screen lets your focus relax and your tear glands do their work.

However, most importantly for this course, a short break lets the lecturer get his caffeine fix. See you all in 15 minutes.

Welcome back. Any questions from the first half of the course?

Choosing a language

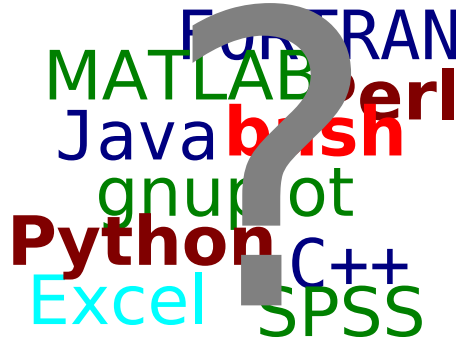
What's ...

best?

available?

used already?

suitable?



UCS

The second half of the course builds on the generalities of the first half and concentrates on identifying what language or tool is most appropriate for each bit of your computational problem.

The answer, unfortunately, is often not to the question “what's best?”.

Typically the first question is “what's available?” I can wax lyrical about various computational tools until I'm blue in the face (or you are) but if you haven't got them they're no use to you. Fortunately, Cambridge is well stocked.

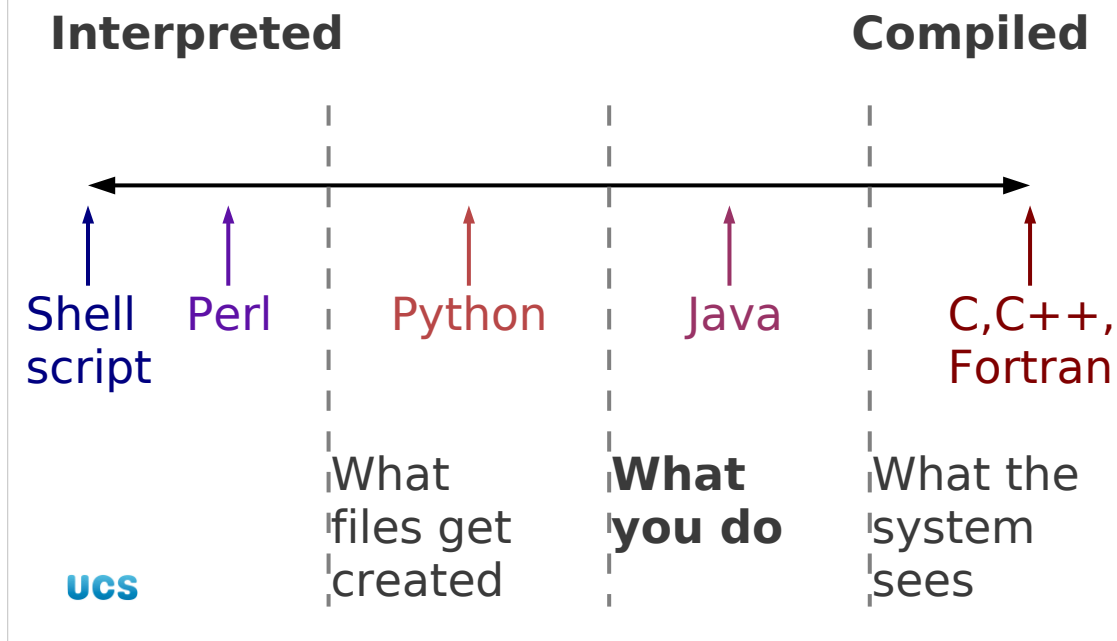
The second question is typically where the (non-reinvented) wheels come off our idealistic quest for the right tool for the job. In many research groups and many entire fields of study there is a body of programming work already in existence that you and your colleagues have to use. If this is all written in one specific programming language, say, then there is good reason for you to use that language too.

If you have the opportunity then break free. You can always call their old programs from your new ones. There is often more flexibility than you think.

Finally, there is the question of what's best. The answer to that depends on what you need from it. Do you need really high performance numerical work? Or are your programs going to be over so quickly that what you really need is something you can write quickly? Do you need to do numerical work or are you manipulating long lists of letters?

Decide what you need and then choose a language. Resist the temptation to stick with what you currently have and to fake a justification.

Classes of language



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 “**c**ompiled”.) 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.

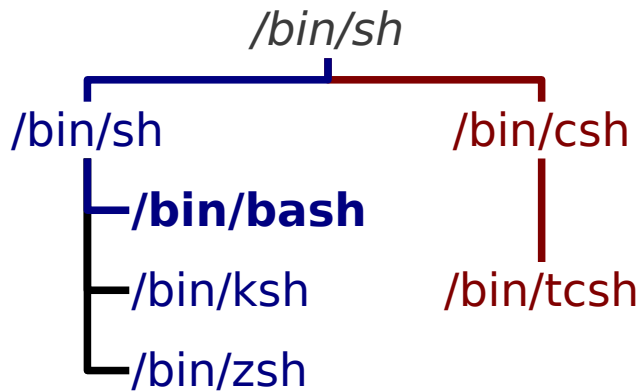
Shell scripting languages

Several scripting languages:

`#!/bin/bash`

`job="${1}"`

...



UCS

We touched on the shell earlier. Now we will revisit it to consider it as a language.

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.

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 script

Suitable for...

gluing programs together

“wrapping” programs

small tasks

Easy to learn

Very widely used

UCS

Unsuitable for...

performance-critical jobs

floating point

GUIs

complex tasks

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).

UCS courses

**“Unix: Introduction to the
Command Line Interface”**

“Simple shell scripting for scientists”

UCS

If you have decided that a shell script is what you need then the UCS does two courses that might be what you need.

The first course teaches the interactive use of the shell: typing commands at the command line, creating pipelines of commands and so on.

<http://training.csx.cam.ac.uk/ucs/course/ucs-unixintro1>

<http://www-uxsup.csx.cam.ac.uk/courses/UnixCLI/>

The next course teaches the construction of basic shell scripts for simple tasks.

<http://training.csx.cam.ac.uk/ucs/course/ucs-scriptsci>

<http://www-uxsup.csx.cam.ac.uk/courses/ShellScriptingSci/>

A note for the curious: “shell scripting” is so named because a file of instructions to follow is called a “script”. The thing that reads the instructions is called a “shell” because it's the layer on the outside. The core operating system is called the “kernel”.

“Further she~~X~~scripting”?

Python! 

UCS

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
```

Perl

```
import library
```

```
...
```

Call out to libraries
in other languages

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

```
use library;
```

```
...
```

UCS

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.

Neither was initially designed for numerically intensive work but Python, in particular, has developed a major following in the scientific community with its NumPy and SciPy libraries.

Perl

The “Swiss army knife” language

Suitable for...

text processing

data pre-/post-processing
small tasks

CPAN: **C**omprehensive
Perl **A**rchive **N**etwork

Widely used

UCS

Bad first language

Easy to write
unreadable code

“There's more than
one way to do it.”

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 **C**omprehensive **P**erl **A**rchive **N**etwork (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.

Python “Batteries included”

Suitable for...

text processing

data pre-/post-processing

small & large tasks

Built-in comprehensive
library of functions

Scientific Python library

UCS

Excellent first
language

Easy to write
maintainable code

The “Python way”

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. It is now very common in Cambridge and the scientific community worldwide.

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.

UCS courses

“Python: Introduction for absolute beginners”

“Python: Introduction for programmers”

“Python: Further Topics”

UCS

The UCS offers a wide range of Python courses and if you perceive a bias against Perl and towards Python for scientific programming then you perceive correctly.

The Python for Absolute Beginners course is designed for people who have never programmed before. If you have, but still want to learn Python, you should attend the “Python for Programmers” single day course which is designed to bring you up to the same level of Python knowledge as the three afternoon absolute beginners course.

<http://training.csx.cam.ac.uk/course/ucs-python>

<http://www-uxsup.csx.cam.ac.uk/courses/PythonAB/>

<http://training.csx.cam.ac.uk/course/ucs-python4progs>

<http://www-uxsup.csx.cam.ac.uk/courses/PythonProgIntro/>

The “Further Python” course follows on from either of these courses and builds up your knowledge of Python to the point where you can tackle serious programming tasks.

<http://training.csx.cam.ac.uk/course/ucs-pythonfurther>

<http://www-uxsup.csx.cam.ac.uk/courses/PythonFurther/>

Because we use Python to introduce the various concepts in programming, we recommend the courses prior all our advanced programming courses (such as the Fortran course). However, if you learn Python and its support libraries you may discover that you do not need those advanced languages any way.

UCS courses

“Python: Numerical programming

“Python: Unit testing”

“Python: Regular expressions”

“Python: Interoperation with Fortran”

“Python: Operating system access”

UCS

We also drill down into the details of various aspects of Python use by scientists with a wide selection of useful courses.

Not all the courses are given every term but all their notes are available on-line from <http://www-uxsup.csx.cam.ac.uk/courses/>:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonnumprog>

<http://www.training.cam.ac.uk/ucs/course/ucs-progtest>
<http://training.csx.cam.ac.uk/course/progtest>

<http://training.csx.cam.ac.uk/course/ucs-pythonregexp>
<http://www-uxsup.csx.cam.ac.uk/courses/PythonRE/>

<http://training.csx.cam.ac.uk/course/ucs-pythonfort>
<http://www-uxsup.csx.cam.ac.uk/courses/PythonFortran/>

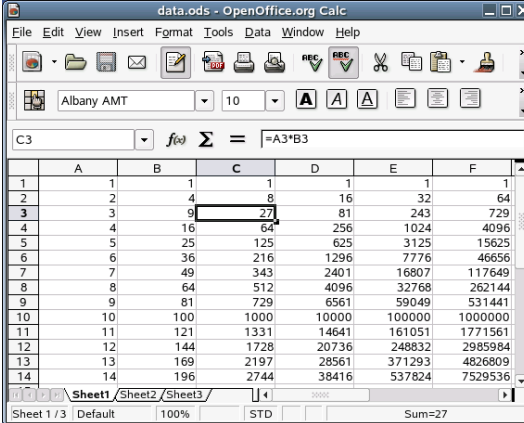
<http://training.csx.cam.ac.uk/course/ucs-pythonopsys>
<http://www-uxsup.csx.cam.ac.uk/courses/PythonOS/>

Spreadsheets

Microsoft Excel

OpenOffice.org 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 has a grid with columns A through F and rows 1 through 14. The formula bar at the top shows 'C3' and the formula '=A3*B3'. The spreadsheet 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

The status bar at the bottom shows 'Sheet 1 / 3', 'Default', '100%', 'STD', and 'Sum=27'.

UCS

Spreadsheets are another example of an interpreted language and one of the ones where the model of a simple linear list of commands starts to falter. Spreadsheets embed single commands, which the program interprets, in amongst the data.

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 OpenOffice.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

UCS

Excel has the advantage that it is taught in school. Having seen what is taught in school, however, we have to say that we're not impressed.

Spreadsheets lend themselves to chaotic, meandering sets of interlinked cells. In the computing world this is called “spaghetti code” and is impossible to maintain without immense effort. Writing good spreadsheets with some claim to be “structured code”, requires discipline and a very methodical style. It is also distressingly easy (without disciplined use of cell locking) to accidentally corrupt a spreadsheet.

It is also harder to write unit tests than it is in more structured languages (though it is not impossible).

The spreadsheets do have the property that they incorporate a good deal more than just calculations. Basic graphics are also available to represent the embedded data but they are *not* suitable for complex graphical representation and, worse, try to lead you into using inappropriate graphical representations of data.

UCS courses

“Excel 2007: Beginners”

“Excel 2007: Introduction”

“Excel 2007: Functions and Macros”

“Excel 2007: Managing Data & Lists”

**“Excel 2007: Analysing &
Summarizing Data”**

UCS

The UCS offers various courses on Microsoft Excel.

To see all our Excel courses use this search link:

<http://training.csx.cam.ac.uk/ucs/search?type=courses&query=Excel+2007>

Specialist systems

Database	→ PostgreSQL, Access, ...
Graphs	→ gnuplot, ploticus, ...
GUIs	→ Glade
Mathematics	→ Mathematica, MATLAB, Maple
Statistics	→ SPSS, Stata

UCS

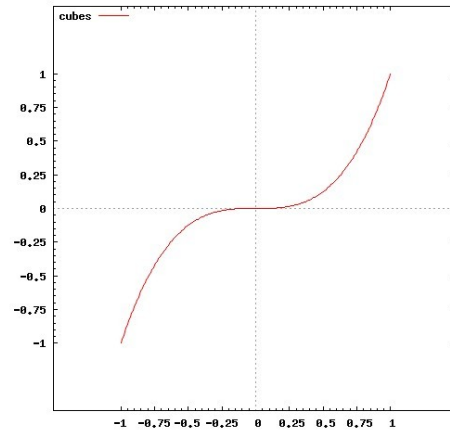
Most of the specialist systems are based around interpreters. Databases interpret SQL (structured query language) instructions, a specialized graphing system like gnuplot reads its instructions from a text file (or the keyboard) and interprets them and so on.

The next few slides will cover the most useful of these.

Drawing graphs

Manual vs. automatic

gnuplot
ploticus
matplotlib



“Gnuplot for simple graphs”

UCS

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”.

There are two dedicated graphics languages: gnuplot and ploticus. Both are available on the PWF and a course is available for download introducing gnuplot.

<http://www.training.cam.ac.uk/ucs/course/ucs-gnuplot>

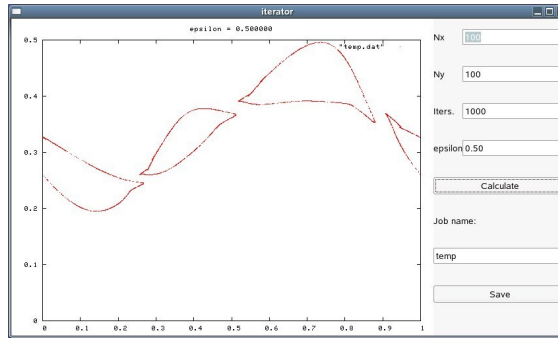
<http://www-uxsup.csx.cam.ac.uk/courses/Gnuplot/>

In addition there is a graphics module for Python called matplotlib. It also has modules for talking directly to gnuplot and ploticus.

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.

GUIs

Glade



Back end: C, C++, Perl, **Python**

“Glade: Introduction to building GUIs”

UCS

Glade is a tool for building GUIs. It is a GUI itself and the application author's “drag, drop and type a bit” instructions then get converted into a Glade configuration file which is then interpreted by Glade to build the actual GUI.

This GUI needs some back end to direct and that can be written in a variety of languages, one of which is Python. This is the back end language that we use in the Glade course.

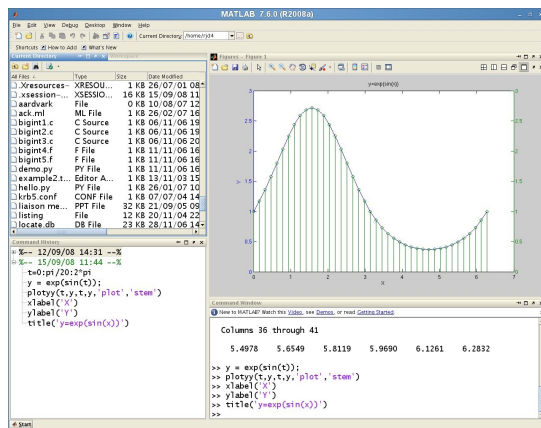
The GUI builder constrains you to use a specific set of GUI elements (called the Gnome Toolkit) but it's not a terrible constraint.

We used to offer a course on Glade but now we just make the notes available for download to get you started.

<http://www.training.cam.ac.uk/ucs/course/ucs-glade>

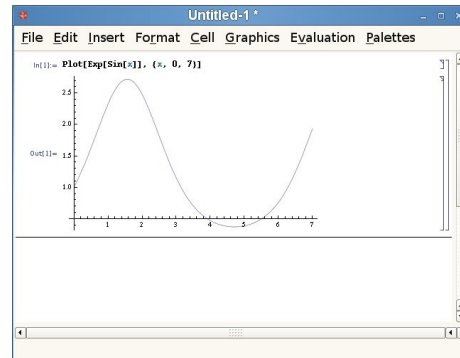
<http://www-uxsup.csx.cam.ac.uk/courses/Glade/>

Mathematical manipulation



MATLAB

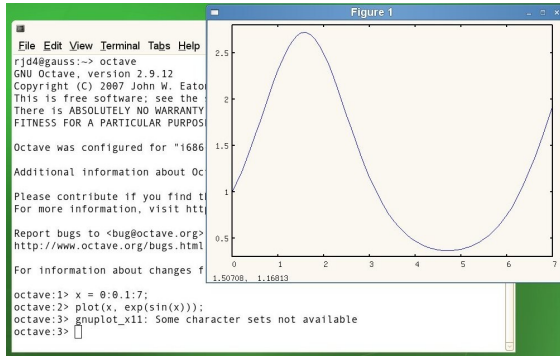
UCS



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.

Mathematical manipulation



Octave

UCS

There is also Octave which is a free analogue, but does lack quite a bit of the functionality of the commercial products.

Mathematical manipulation

Suitable for...

“fiddling”

Small and medium
numerical work

Graphical subsystem

Problems...

additional
“modules”
often needed

UCS

Generally speaking the mathematical manipulation packages are good for “fiddling” and small to medium size numerical work. This is almost a circular definition as a good description of “large” numerical problems is that they are “too big for MATLAB” or “too big for Mathematica”.

They both have graphical subsystems which allow for graphs to be generated and manipulated manually. This contrasts with the gnuplot or matplotlib style of graphing which is more suitable for automated graph production.

Note that MATLAB farms out much of its specialist functionality to “toolboxes” (its name for modules) which require separate licensing. What works on one MATLAB instance may not work on another if it lacks the toolboxes used. Mathematica takes a more “kitchen sink” approach.

UCS courses

“MATLAB: Basics”

“MATLAB: Graphics”

“MATLAB: Linear algebra”

“Mathematica: Basics”

“Mathematica: Graphics”

“Mathematica: Linear algebra”

UCS

The UCS offers sets of courses in both packages.

MATLAB: Introduction for absolute beginners

<http://www.training.cam.ac.uk/ucs/course/ucs-matlab>

MATLAB: Graphics

<http://www.training.cam.ac.uk/ucs/course/ucs-matlabfu>

MATLAB: Linear algebra

<http://www.training.cam.ac.uk/ucs/course/ucs-matlablinaleg>

Mathematica: Basics

<http://www.training.cam.ac.uk/ucs/course/ucs-mathematica>

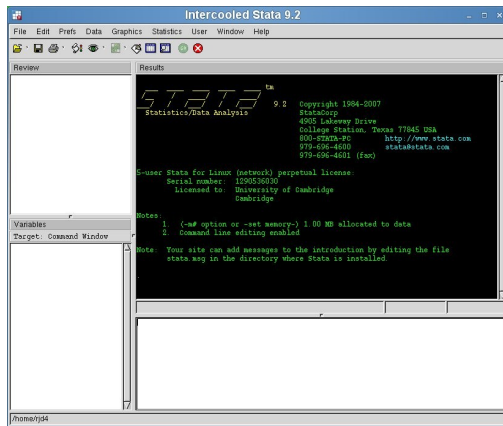
Mathematica: Graphics

<http://www.training.cam.ac.uk/ucs/course/ucs-mathgraphics>

Mathematica: Linear algebra

<http://www.training.cam.ac.uk/ucs/course/ucs-mathlinear>

Statistics



Stata

UCS

There are two big players in statistics packages in Cambridge: Stata and SPSS with the free product R making rapid in-roads. (SPSS is not available under PWF Linux.)

UCS courses

NB: we do not teach statistics!

“SPSS: Introduction for beginners”

“SPSS: Beyond the basics”

“Stata: Introduction”

“Stata for regression analysis”

“Regression analysis in R”

UCS

The UCS offers courses on both SPSS, Stata and R.

Please note that we teach how to do statistics with these packages. We assume you know the statistics already. We do not teach the principles of statistics in our SPSS, Stata and R courses any more than we teach calculus in our MATLAB or Mathematica courses.

SPSS: Introduction for beginners

<http://www.training.cam.ac.uk/ucs/course/ucs-spss1>

SPSS: Beyond the basics

<http://www.training.cam.ac.uk/ucs/course/ucs-spssbb>

Stata: Introduction

<http://www.training.cam.ac.uk/ucs/course/ucs-stata1>

Stata for regression analysis

<http://www.training.cam.ac.uk/ucs/course/ucs-stataregress>

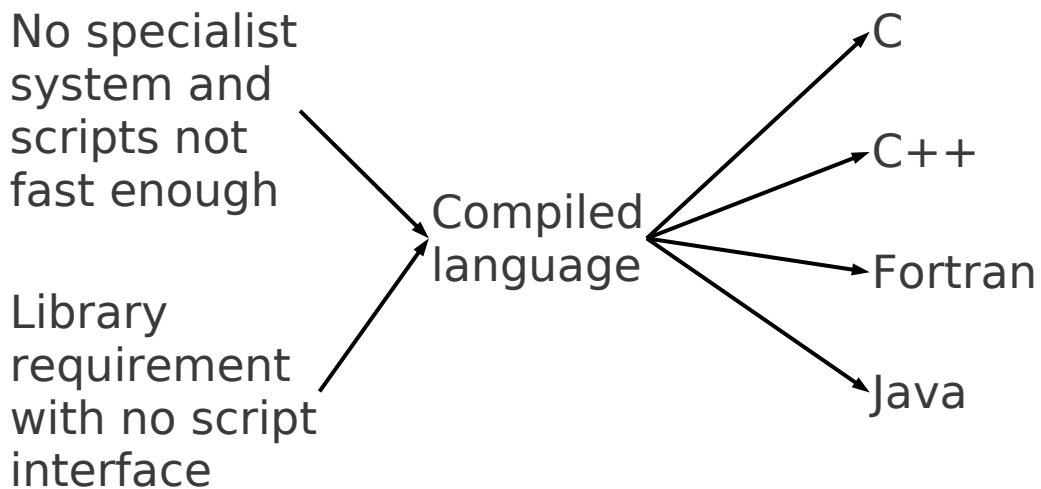
R: Introduction for Beginners

<http://www.training.cam.ac.uk/ucs/course/ucs-r>

Regression analysis in R

<http://www.training.cam.ac.uk/ucs/course/ucs-rregress>

Compiled languages



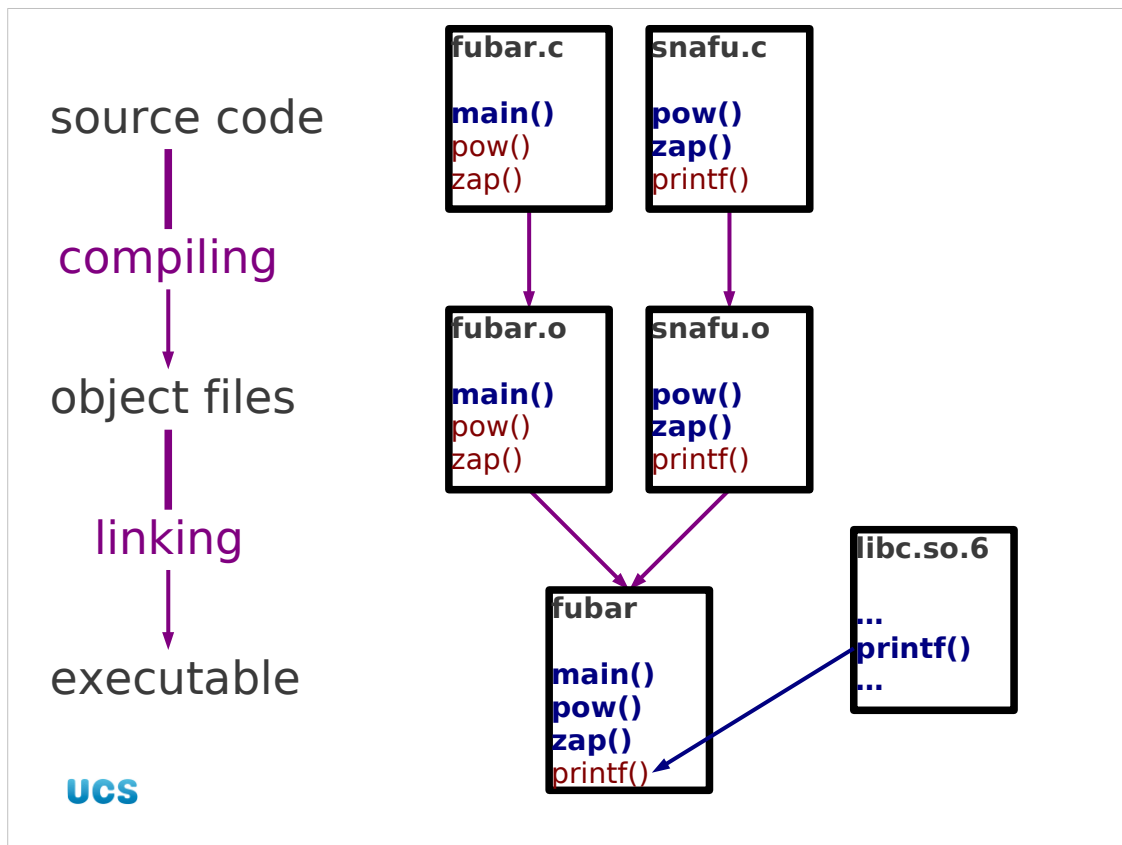
UCS

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 specialist systems. However the skill, once acquired, can be more portable precisely because it is not specialist.

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 then talk about Java because from the point of what you have to do there is an explicit compilation stage.



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.

“Somebody else's code”

“Unix: Building, installing and running software”

UCS

Before you all go thinking that you need to learn C or Fortran because you have code already written that uses it, pause for a moment. If all you need to do is build the code and run it then, unless you end up having to modify the code, you do not need to know the language it's written in.

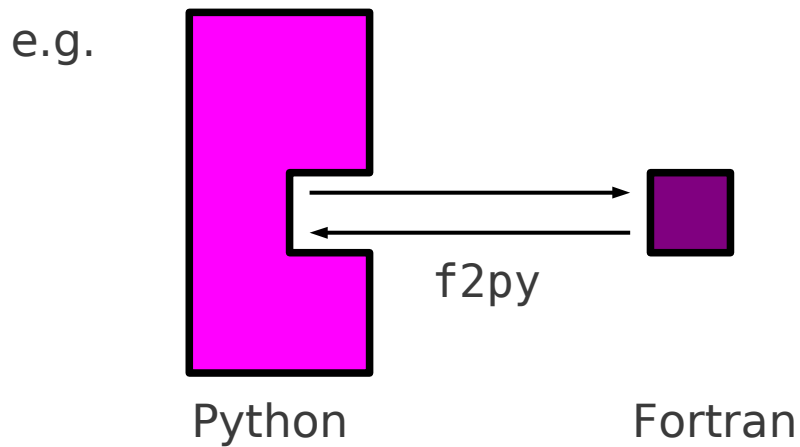
The UCS offers a course with the catchy title of “Building, installing and running software” that teaches the most common techniques for building code in compiled languages. Knowledge of the programming language — while obviously an advantage for when things go wrong — is not necessary.

Incidentally the “installing and running” bit of the course is all about how you don't need system privileges on a Unix box to use your own software suite.

<http://www.training.cam.ac.uk/ucs/course/ucs-unixware>

<http://www-uxsup.csx.cam.ac.uk/courses/Building/>

You don't need to write the whole program in a compiled language!

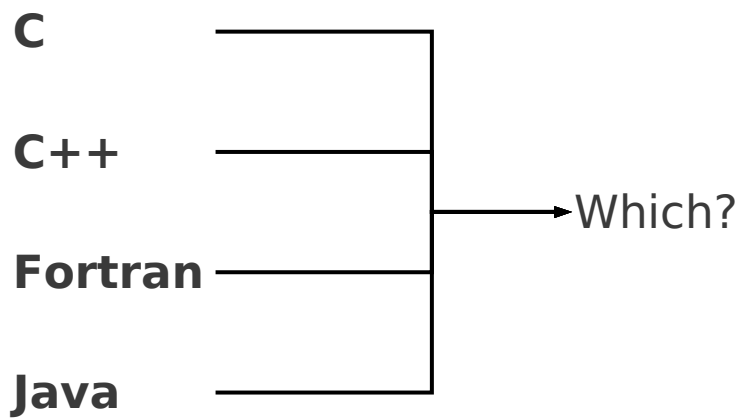


UCS

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.

If there is a numerically intensive section in your program by all means write it in Fortran. But don't drag the rest of the program with it.

Compiled languages



UCS

So which compiled language should you use?

Fortran

The best for numerical work

Excellent numerical libraries

Unsuitable for everything else

Very different versions:
77, 90, 95, 2003

UCS

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 courses

“Fortran: Introduction to modern Fortran”

“Python: Interoperation with Fortran”

UCS

We offer two Fortran courses.

The “Introduction to modern Fortran” takes three full days and the “Interoperation with Fortran” only takes one afternoon but assumes you already know both Fortran and Python.

Fortran:

<http://www.training.cam.ac.uk/ucs/course/ucs-fortran>

<http://www-uxsup.csx.cam.ac.uk/courses/Fortran/>

Python and Fortran:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonfort>

<http://www-uxsup.csx.cam.ac.uk/courses/PythonFortran/>

C

The best for Unix work

Excellent libraries

Superceded by C++

Memory management

UCS

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 superceded 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.

Very basic C course

“C: Introduction for Those New to Programming”

UCS

We offer one course in C, which aims to to get everyone to the stage of being able to write small utility programs in C for carrying out simple calculations and data manipulation.

<http://www.training.cam.ac.uk/ucs/course/ucs-c>

C++

Extension of C

Object oriented

Standard template library

Good general purpose language

Very hard to learn *well*

UCS

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 good 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!

Learning C++

“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

UCS

The best we can offer you for C++ is a couple of book recommendations. These are not light reading!

Bruce Eckel has written a number of good books introducing programming in various languages. His Python book is a classic. Bjarne Stroustrup is the creator of C++ and still the world's definitive expert. He knows what he's talking about. He is the author who reckons that learning the language properly takes 15 hours a week for 14 weeks.

Parallel programming

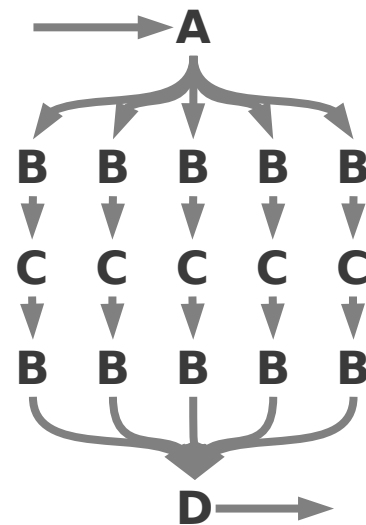
**“Parallel programming:
Introduction to MPI”**

**“Parallel programming:
Options and design”**

MPI library

Fortran, C, C++

UCS



We offer one course in parallel programming, using the MPI library from three languages: C, C++ or Fortran. Obviously, you need to know at least one of these languages before taking this course.

<http://www.training.cam.ac.uk/ucs/course/ucs-pprogmpi>

<http://www-uxsup.csx.cam.ac.uk/courses/MPI/>

We also offer a course on the theory of building parallel programs.

<http://training.cam.ac.uk/ucs/course/ucs-pprogopt>

<http://www-uxsup.csx.cam.ac.uk/courses/Parallel/>

Java

Object oriented

Good general purpose language

Much easier to learn and use than C++

Cross-platform

Some poorly thought out libraries

Multiple versions: Use ≥ 1.5

ucs 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6

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 1.5 or later you should do OK.

Java courses

Computer Lab Java courses

IA: **“Object Oriented Programming”**

IB: **“Further Java”**

UCS

We don't teach a Java course, either. The Computer Laboratory (the academic department) does teach Java as a standard lectured course and remember that as members of the University you are entitled to attend any lecture course you want outside the medical and veterinary sciences. You aren't entitled to the supervisions, course work marking etc. The Lab's introductory course is called “Object-Oriented Programming” and there is a second called “Further Java”.

Other UCS courses of interest

“vi editor: introduction”

“emacs editor: introduction”

(But there's always “gedit”)

“Programming concepts: introduction for absolute beginners”

“Program design: organising and structuring programming tasks”

UCS

Finally, I must mention some other courses of general use to people handling computers or even just large amounts of data in text files. We teach how to use the two primary editors of the Unix world, emacs and vi. For much text editing, however, there's the GUI text editor “gedit” which is often easily good enough for most uses.

emacs:

<http://training.cam.ac.uk/ucs/course/ucs-emacs>

<http://www-uxsup.csx.cam.ac.uk/courses/Emacs/>

vi:

<http://training.cam.ac.uk/ucs/course/ucs-vi>

<http://www-uxsup.csx.cam.ac.uk/courses/Vi/>

We also have a couple of theoretical courses (as opposed to hands on courses) which go into greater detail on the structuring of programs.

Programming concepts:

<http://training.cam.ac.uk/ucs/course/ucs-progbasics>

Program design:

<http://training.cam.ac.uk/ucs/course/ucs-progdesign>

Courses

All UCS courses

<http://training.csx.cam.ac.uk/ucs/theme>

Scientific computing “theme”

<http://www-uxsup.cam.ac.uk/courses/>

<http://training.csx.cam.ac.uk/ucs/theme/scientific-comp>

UCS

If you want a list of all the UCS courses, please see our training web site. The Scientific Computing courses form a coherent subset of those and many have a web page of their own on the other server quoted. Some of the newer courses' notes may not be on-line until the first time the course is given for real.

We particularly encourage you to take a look at the detailed notes for the courses to decide whether they are right for you. We don't object to people using our notes instead of going to courses, either. Our function is to get you better at using computers for your research, not to put bums on seats in our classrooms. That's a means, not an end.

Contacts

UCS service desk

service-desk@ucs.cam.ac.uk

Scientific computing support

scientific-computing@ucs.cam.ac.uk

UCS

Finally, if you have any difficulties with any UCS facility you can contact our Service Desk for assistance. Perhaps more relevant to this course, however, is the scientific-computing@ucs.cam.ac.uk email address which can be used for any manner of scientific computing support query.