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
Course outline

Basic concepts

Good practice

Specialist applications

Programming languages
Course outline

Basic concepts

Good practice

Specialist applications

Programming languages
Serial computing

Single CPU
Parallel computing

Multiple CPUs

Single Instruction
Multiple Data

MPI
OpenMP
Parallel computing courses

Parallel Programming: Options and Design

Parallel Programming: Introduction to MPI
Distributed computing

Multiple computers
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 → 0.1000000000000001
and worse...

>>> 0.1 + 0.1
0.2

>>> 0.1 + 0.1 + 0.1
0.30000000000000004
Floating point courses

Program Design:
How Computers Handle Numbers
Text processing

e.g. sequence comparison
text searching

```
^f.*x$
```

“Regular expressions”
Regular expression courses

Programming Concepts:
Pattern Matching Using Regular Expressions

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

Basic concepts

Good practice

Specialist applications

Programming languages
“Divide and conquer”

“divide”

Simple problem

“conquer”

Partial solution

Partial solution

Partial solution

“glue”

Complete solution
“Divide and conquer” — the trick

No need to use the same tool for each “mini-conquest”!
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.
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.
Example

Read 256 lines of data
Each line will have 256 numbers on it.

256×256 set of data.

Aardvark’s algorithm

output

CSV format

CSV format

plot a heat graph

Keep reading 256-line lumps like this until they’re all done.
“Structured programming”

Split program into “lumps”
Use lumps methodically
Do not repeat code

“Lumps”?

Programs
Functions
Modules
Units
Example: unstructured code

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

...  # Repetition!

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

...  # Repetition!

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

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

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

Single instance of the code.

Calling the function three times
Structured programming

Once!

Write function

Import function

Test function

Time function

Debug function

Improve function

All good practice follows from structured programming
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

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

a_norm = norm2(a)
...

b_norm = norm2(b)
...

c_norm = norm2(c)
**Example: best code**

```python
from library import norm2

a_norm = norm2(a)
...

b_norm = norm2(b)
...

c_norm = norm2(c)
```

Somebody else’s code!

No change to calling function
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
Example libraries

- Numerical Algorithms Group
- Scientific Python
- Numerical Python
Hard to improve on library functions

```c
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]
        }
    }
}
```

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

It is a complete waste of time!
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
Algorithms

Time taken / Memory used

vs.

Size of dataset / Required accuracy

$O(n^2)$ notation

Algorithm selection makes or breaks programs.
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”
Revision control

Code “checked in” and “checked out”

Branches for trying things out

Communal working

Reversing out errors.
Revision control

Two main programs: subversion
git

Starting from scratch? git
Something in use already? Use it!

github.com free repository (for open source)
try.github.io free online training
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
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
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
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!
Excel courses

Excel 2010/2013:

Introduction
Analysing and Summarising Data
Functions and Macros
Managing Data & Lists
Statistical software
Statistical software

Stata: Introduction

R: Introduction for Beginners

SPSS: Introduction for Beginners

SPSS: Beyond the Basics
Mathematical manipulation

Matlab

Octave

Mathematica
Mathamtical software courses

Matlab:

- Introduction for Absolute Beginners
- Linear Algebra
- Graphics (Self-paced)
Drawing graphs

Manual or automatic?
Courses for drawing graphs

Python 3: Advanced Topics (Self-paced) (includes a matplotlib unit)
Course outline

Basic concepts

Good practice

Specialist applications

Programming languages
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
Shell script

Several “shell” languages:

```bash
#!/bin/bash

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

/bin/sh

- /bin/bash
- /bin/ksh
- /bin/zsh

- /bin/sh
- /bin/csh
- /bin/tcsh
- /bin/ksh
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! ✔
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.
Perl

The “Swiss army knife” language

Suitable for...

- text processing
- data pre-/post-processing
- small tasks

Bad first language

CPAN: Comprehensive Perl Archive Network

Very easy to write unreadable code

“There's more than one way to do it.”

Widely used

Beware Perl geeks
Python

Suitable for…

- text processing
- data pre-/post-processing
- small & large tasks

Built-in comprehensive library of functions

Scientific Python library

“Batteries included”

- Excellent first language
- Easy to write maintainable code
- The “Python way”
- Very widely used
- Code nesting style is “unique”
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
Compiling, linking, running

source code files

object files

executable

execution

compilation

linking

run-time

text files

machine code files

machine code file

fubar.c
  main()
  pow()
  zap()

snafu.c
  pow()
  zap()
  printf()

compilation

fubar.o
  main()
  pow()
  zap()

snafu.o
  pow()
  zap()
  printf()

linking

fubar
  main()
  pow()
  zap()
  printf()

run-time

libc.so.6
  ...
  printf()
  ...

machine code file
No need to compile whole program

Python script

Critical function
No need to write the whole program in a compiled language.
Fortran

The best for numerical work

Excellent numerical libraries

Unsuitable for everything else

Very different versions:
  77, 90, 95, 2003
Fortran course

Fortran:
Introduction to Modern Fortran

Three full days
C

The best for Unix (operating system) work

Excellent libraries

Superceded by C++ for applications

Memory management
C++

Extension of C

Object oriented

Standard template library

General purpose language

Very hard to learn well
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 [leaning 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
Java courses

Object oriented programming

CL lectures
(Also classes, ask at the CL)
Scientific Computing

training.cam.ac.uk/ucs/theme/scientific-comp

scientific-computing@ucs.cam.ac.uk

www.ucs.cam.ac.uk/docs/course-notes/unix-courses