Modelling and data analysis ‘Winter School’

Nick Golledge[1],
Liz Keller[1,2],
Alex Gossart[1],
Alena Malyarenko[3],
Angela Bahamondes-Dominguez[3],
Mario Krapp[2],
Dan Lowry[2],
Stefan Jendersie[1]

[1] Antarctic Research Centre, Victoria University of Wellington, New Zealand
[2] GNS Science, Lower Hutt, New Zealand
[3] NIWA, Wellington, New Zealand

1 | Welcome & Introduction

Day 1

10:00	Arrival & welcome	Nick
10:15	Introduction to programming Navigating the command line environment, scripting vs programming, pros & cons of various languages	Nick
11:30	Introduction to models Climate model basics: components, types of models, internal variability. CMIP overview, climate sensitivity	Liz & Dan
13:00	Lunch	
14:00	Time-series data – lecture Principal component / empirical orthogonal function analysis, calculation of correlations, anomalies, detrending	Mario
15:30	Afternoon tea	
15:45	Time-series data – tutorial	Mario
17:00	Wrap-up	

Day 2

09:00	Spatial data – lecture Understanding gridded data, map projections, data analysis and manipulations, masking, extracting vertical / horizontal sections	Alex & Alena
10:30	Coffee	
10:45	Spatial data – tutorial	Alex & Alena
12:15	Lunch	
13:15	Document preparation in LaTeX Learn the basics, write equations, insert figures, create your own tables, insert references	Angela
14:45	Afternoon tea	
15:00	Work Structure & Version control Defining a workflow, handling 'big data', version control for scripts/documents, best practice guidelines	Stefan

1 | Aims, Methods, & Scope

- The **aim** of the Winter School is that, by the end of the two days, participants will be able to find and download (climate model) data of interest, use simple scripts to process, analyse, and plot those data, integrate these outputs into a typeset document, and use version control software to keep track of changes.

- We will use ***Python*** for the majority of the work but will incoporate examples from other languages if necessary. We'll introduce you to packages like LaTeX and tools such as *github*.

- This workshop is only intended to provide an **introduction** to working in a command-line environment, and exposure to some of the functionality available in this realm. It is not intended to be a complete course on programming, modelling, or data analysis ;-)

2 | Command-line basics (*nix)

Basic commands

command	example	description
ls	`ls -ltrh`	list directory contents (in long format, newest last)
cd	`cd ../mydir/mysubdir`	change directory (up one level, down two)
rm	`rm delete-this.txt and-all-these.*`	remove file(s)
mv	`mv rename-this.txt to-this.txt`	move (rename) file(s)
mkdir	`mkdir ./new-directory`	make a new (empty) directory
cp	`cp this.txt ./new-dir/to-this.txt`	copy file (possibly to new location)

Linux c-line tools

tool	example	description
pwd	`pwd`	Find out what your current personal working directory is
sed	`sed -e 's/a/b/g'`	stream editor, swap 'a' for 'b'
awk	`awk '{print $2, $3}'`	print fields 2 & 3 from file/stream

Other packages & utilities

package	example	description
pdflatex	`pdflatex myfile.tex`	compile LaTeX document
git	`git clone golledni/WinterSchool`	Make a local copy of a github repository

2 | Simple (bash) shell scripting

- We can combine many simple commands, tools, and utilities to achieve more complex tasks

```
pwd
/home/golledni/MEGA/Work/AntSciPlat/WinterSchool

pwd | sed -e 's/\// /g' | awk '{print "Today my",$1,"is the",$NF}'
Today my home is the WinterSchool
```

2 | Simple (bash) shell scripting

- But to do anything more complex than simple pipes we probably want to write a script file to contain our sequence of commands:

```
#!/bin/sh

lastupdated=`head -n 1 new_papers.txt`
echo "Last updated " $lastupdated
now=$(date +%F)

echo $now > new_papers.txt
# get list of directories to loop through
list=`ls -l | grep ^d | awk '{print $9}'`
# find NEW papers in each of those directories
echo "\nNEW PAPERS:\n" >> new_papers.txt
for dir in $list ; do
        echo "\n********** "$dir" **********\n" >> new_papers.txt
        find ./$dir -type f -newermt $lastupdated -print | awk -F"/" '{print $3}' | sed -e 's/.pdf/}
 /g' -e 's/_/ /g' -e 's/^/\\subsubsection{/' >> new_papers.txt
done
```

2 | Control structures

- Often we want to apply a test, or series of tests, to the data we're processing, and do different things with the data depending of the results of those tests
- Control structures are what help us achieve this, and are fundamental to all languages
- The two most common generic structures are
  - `if` statements, and
  - `for` or `do` loops

`if` statement:

```
i=0
if [ $i -ge 1 ]
then
   echo "i = $i"
else
   echo "i < 1"
fi
```

`do` loop:

```
i=0
imax=10
while [ $i -le imax ] ; do
   echo "i = $i"
   i=`expr $i + 1`
done
```

2 | “Hello, World!”

Bash:

```
#!/bin/sh

echo "Hello, World!"
```

Python:

```
#!/usr/bin/env Python

print "Hello, World!"
```

Julia:

```
#!/usr/bin/env Julia

print("Hello, World!")
```

Fortran 90:

```
PROGRAM HELLOWORLD

   IMPLICIT NONE
   print *, 'Hello, World!'

END
```

C++:

```
#include <iostream>

int main() {
   std::cout « "Hello, World!";
   return 0;
}
```

2 | Interpreted vs. compiled languages

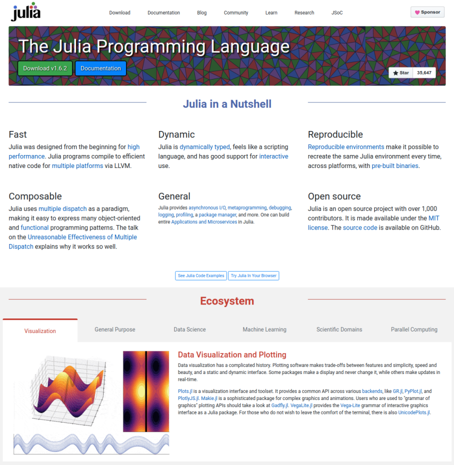
- Compiled languages require a *compiler* to convert user code into machine code
- Typically they create a platform-dependent binary (executable) file
- If the code doesn't change, the binary can be run again and again
- Once compiled, programs using these languages are typically very fast to run

- Interpreted languages read and execute user code line-by-line
- No separate compilation step is required, so programs are platform-*in*dependent
- But, interpretation has to happen every time the code is run
- As a result, this kind of code is typically slow to run

2 | Just-in-time compilers

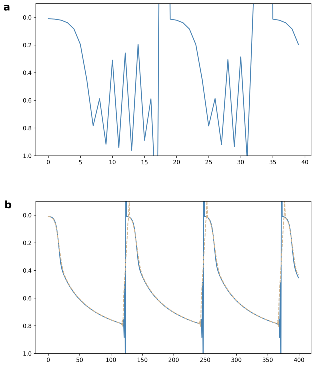
Just-in-time (JIT) compilation:

- Some modern languages like **Julia** use the JIT (or dynamic compilation) approach
- With JIT, compilation of relevant code occurs <u>at runtime</u>
- If same packages / modules are called in subsequent runs, no re-compilation is necessary
- This approach combines the best aspects of interpreted and compiled languages



2 | Fundamentals of numerical modelling

- Usually, a numerical model consists of a set of calculations that are repeated
- Typically, each repetition of the solution involves a step forward in time
- A spatially *discretized* model may use an *implicit* or *explicit* time step
- Numerical solutions are prone to error (compared to an analytical solution)
- Accumulated error tends to produce instability & model crash
- Usual culprits are fluxes getting too great, or time steps being too long



2 | Fundamentals of numerical modelling

- Often the equations we are trying to solve are differential equations, i.e. they describe a quantity that changes through time

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$