Introduction to computing at Bullard

Laurence Cowton, Matthew Falder, Rob Green, Karen Lythgoe, Matt Davis, Ross Hartley, Tim Craig, Alistair Crosby & Jeff Winterbourne

October 20, 2015

Contents

1 Introduction

This guide is primarily a companion to our introductory talk, but we hope it should also function as a stand-alone introduction to some of the Linux tools most commonly used at Bullard. We presume that you are already familiar with the command line, but that's it. There is a lot to Linux and, unfortunately, much of it is less than user-friendly. As always, your best sources of help are `man` pages, the Bullard web pages[1], Google, Dave Lyness[2], Ian Frame[3] and – of course – your fellow students in person. Never be afraid to ask!

1.1 Notes about this document

1. A ◇ at the beginning of a line means you should type the following text at the command line and then press Enter.
2. The " ‘ " means a *backquote* on the keyboard, not an apostrophe (" ’ ").
3. Files for sections 2–4 can be copied from `/home/mwd27/For_Students/comp_talk/` on any computer connected to `sirius`.

1.2 The computing system at Bullard

The computing system at Bullard consists of the main Linux server (called `sirius`) and local machines assigned to each user. There are also shared facilities such as A4 printers, the A0 poster plotter, RAID backup machines, the Bullard FTP server, and the Cluster, all of which can be accessed from any machine on the network. You can also access these machines remotely if you need to work from home, as this document will explain.

1.3 Some commonly used text editors

All files are, ultimately, coded sequences of 1's and 0's on computer discs, which can be interpreted by some programs and not others. However, *textfiles* deserve a special mention because they are so widely-understood and easy to work with. Text (or ASCII) files, as the name suggests, can encode of any of the characters on your keyboard, plus a few more, and can be read by text editors, word processors, spreadsheets, Linux utilities and indeed your own programs. You will likely use them to store most of your data, reports and programs. There are several text editors which you will find on your machine.

`gedit` - has a very simple and easy to use graphical user interface, with language specific colour highlighting (great for coding!). It also has a very convenient search and replace tool.

`kate` - increasingly widely used, similar to gedit, but with a "smarter" interface/highlighting, including a terminal, easy switching between files, and "folding" of code - whole loops can be minimised, in a way that often makes longer codes easier to understand.

[1] `http://sirius/`
[2] `dgl11@cam.ac.uk`
[3] `if201@cam.ac.uk`

`vi/vim` - fairly powerful but the commands are almost entirely keyboard based. A graphical version called `gvim` exists which combines the two. The command `:q!` gets you out of `vi` without altering anything.

`emacs/xemacs` - similar to `gvim` as it allows both keyboard and mouse commands, but is possibly easier to get used to.

2 Bash commands

When you open up a terminal, you are starting a program called a *shell*. A shell is both an interactive command language and the language used to script the system. In other words, in addition to simply executing a program by typing its name, you can also write simple programs of your own to automate sequences of instructions. The most widely-used shell in Bullard (there are several) is called `bash`, which we will use throughout the rest of the guide[4]. Many terminals are set to use `c-shell` by default. Type

◇ `echo $shell`

to find out which one your are using.

To execute a program, just type its name, and press enter. For instance

◇ `firefox`

opens a new firefox browser. Every command has its own *man page* listing all possible options. Man pages can be accessed by typing `man` followed by the command, for example:

◇ `man firefox`

(Gives you more options than you could ever care to use!). Press `q` to quit.

To open a program in the background (i.e. so you can continue to use the terminal) include an ampersand (`&`) at the end of the command, for instance

◇ `firefox &`

2.1 Files and directories

Now let's establish which directory you are in. Type

◇ `pwd`

The shell should automatically start you off in your *home* directory, for example `/home/mwd27`. To see what files that directory contains, type

◇ `ls`

The long versions, `ls -l` and `ls -lht`, give you extended information about how big each file is, sorted by when it was last modified. To make a new directory, say one called `thesis` (you have to start somewhere...), type

[4] In addition to this guide, there is a good online tutorial at `http://www.freeos.com/guides/lsst/`

◇ `mkdir thesis`

To remove a directory, say one called `junk`, type

◇ `rmdir junk`

To move a file, say one called `todo.txt`, into your new thesis directory, type

◇ `mv todo.txt thesis/`

The copy command `cp` does the same but without destroying the original file. To move into your new directory, type

◇ `cd thesis/`

and to move back up to the parent directory, type

◇ `cd ..`

The `..` can be repeated, so, for instance, `cd ../../../` moves three levels up the directory tree.

There are two ways to move to your home directory

◇ `cd ~`

and

◇ `cd`

This is because ~ is a short cut for your home directory (see below) and the default directory for the command `cd` is also your home directory.

◇ `cd -`

will take you back to the previous directory you were in which is very useful when working in two directories at the same time.

It is possible to access files and directories there from any other directory. For example

◇ `gedit ~/file.dat &`
◇ `cd ~/thesis`

Finally, to delete a file, say one called `old_plan.txt`, type

◇ `rm old_plan.txt`

Deleted files cannot be recovered unless backed-up, so be careful when using this command!

2.2 Predictive text and whatnot

The philosophy of linux commands is that they should involve as little typing as possible (that is, for instance, why you type `cp` rather than `copy`). It is therefore not surprising that bash includes a number of useful shortcuts. In particular:

1. Pressing the up arrow repeatedly cycles through all the previous commands.

2. Pressing TAB mid-way through a command is the equivalent of turning predictive text on, i.e. the system will try to guess what you are typing

3. Pressing CTRL-R before typing a command or part of a command will bring back the last instance when you typed that command, which is useful when the command has a long list of options, and you haven't typed it in a while. Pressing CTRL-R again will cycle further and further back through the histoy of that command.

4. To save you typing out frequently used commands, you can set up a shortcut, or *alias*. For instance, if I type

   ◇ `alias ss='ssh -X tjc52@sirius'`

   then every time I want to log into Sirius (we'll come to these commands later), all I need to do is type `ss`. Aliases can be added to the `.bashrc` file[5] in your home directory and will be remembered every time you log in.

5. Pressing CTRL-Z will pause a running process. To start it again type either `fg` to run in the foreground, or `bg` to run in the background. This second option is very useful if you forget to include an `&` first time around.

2.3 Simple text utilities, directions and pipes

If you want to see quickly the contents of a text file without the hassle of opening up a text editor, you can use the commands `cat`, `more`, `less` and `head`. `cat` splurges the entire contents of the file into the terminal; `more` and `less` do the same, but only one screen at a time; and `head` gives you only the first 10 lines (use the `-n` flag to change the number). The command `wc -l` tells you how many lines a file has. Finally, the `sort` command – as the name suggests – sorts a list of lines in alphabetical order. To sort a list of lines in *numerical* order, use `sort -n`, and in *reverse* order, use `sort -r`. As with pretty much all options, these can be combined, so *reverse numerical* order would be achieved with `sort -rn`.

To write a stream of text to a file, use the `>` symbol. So, let's say we have a file `words.txt`,

◇ `sort words.txt > sorted_words.txt`

makes a new file, `sorted_words.txt`, which is the contents of `words.txt` in alphabetical order. If you want to *append* an existing file with your output, use "`>>`" instead.

To send the output of one program directly into another program we make use of a *pipe*, denoted by the `|` symbol. So, for example, to display only the first 20 lines of the file `words`, first sorted into alphabetical order, type

◇ `sort words.txt | head -n20`

Pipes can, of course, be used in series. For example

◇ `sort words.txt | head -n20 | sort -r`

sorts the first 20 items of the sorted list in reverse alphabetical order.

To join two text files together, head to tail, use the `cat` command with two arguments. To join two text files in parallel, use the `paste` command. Let's say we have a file `data1.dat`, which contains the following

[5] Or `.mycshrc` file if you are using the c-shell

```
1.0 5.6
2.0 6.9
3.0 5.4
```

and a second file, `data2.dat`, which contains the following

```
4.0 3.9
5.0 3.5
6.0 2.0
```

then

◇ `cat data1.dat data2.dat`

gives

```
1.0 5.6
2.0 6.9
3.0 5.4
4.0 3.9
5.0 3.5
6.0 2.0
```

and

◇ `paste data1.dat data2.dat`

gives

```
1.0 5.6 4.0 3.9
2.0 6.9 5.0 3.5
3.0 5.4 6.0 2.0
```

2.4 Regular expressions

Two *regular expressions* that are commonly used in the shell are the wildcards `*` which matches any string of characters of any length (including zero) and `?` which matches exactly one character. For instance, typing

◇ `ls *.sh`

will list files in the present working directory with names ending with `.sh`. Typing

◇ `mv ??????  junk/`

will move any file (or directory) that has a name exactly six characters in length to a directory called `junk`. Regular expressions are used in many programs (such as `vi`, `sed`, `awk` and `grep`) and can get much, much more sophisticated. There is plenty of documentation out there to help you once you start to get the hang of things.

2.5 Variables and arithmetic

In linux, and in programming in general, a *variable* is a piece of text which stands for something else, which might be a number or some more text. Variables are very useful in scripts, both as a way of saving typing when you want to change a fragment of code that is repeated lots of times, and because they allow you to change parts of a script automatically as it runs. You can assign a value to a variable using the `=` sign. For instance, to assign the text "green" to the variable "apples", type

◇ `apples=green`

When you use a variable, you must preface it with the `$` sign. So to see the value of our variable, type

◇ `echo $apples`

If you haven't yet defined a variable (for instance, had you typed `echo $pears`), then nothing is displayed. `echo`, like its name suggests, simply returns whatever you gave it, e.g.

◇ `echo hello`

```
hello
```

which is more useful than you might think.

Maths is not bash's strong point, but it is possible to perform simple *integer arithmetic* on numerical variables. Say you have a variable `num` set equal to some number, the following line will increment that value by 1:

◇ `num=‘echo $[$num+1]‘`

The rather inelegant construction `echo $[$num+1]` returns the value of `$num` plus one, and the backquotes mean "substitute the output of whatever is in the quotes here", e.g. if `num = 1`, then `num=‘echo $[$num+1]‘` is the same as saying `num=2`.

If you ever need to do non integer math on the command line (because bash thinks $3/2 = 1$), you will need to use another program, for example `bc`.

◇ `num=‘echo "$num/3" | bc -l‘`

2.6 Scripts

Conveniently, you can run a batch of commands at once by opening a text editor, writing them out in a *script*, saving it, and then running it at the command line. A script is a text file, which must start with the following incantation:

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

which tells the system which shell to use. Once you have saved the script, you need to give yourself *permission* to use it. Say the script is called `script.sh`, simply type

◇ `chmod u+x script.sh`

at the command line (see `man chmod` to set or unset other permissions). You are then ready to run it, by simply typing in the script's name. [6] Now let's look at some commands.

2.7 Simple tests and loops

Two simple, and very useful, features of bash are its ability to only run a sequence of commands if some particular condition is met (a test), or to run a series of commands repeatedly until some condition is met (a loop). Tests usually start with the word `if`, and loops usually start with the words `for` or `while`. Let's look at the following script:

```
#!/bin/bash

for i in *.dat
   do
   echo $i
   wc -l $i
done
```

This script does the following: for every file in my current directory with the suffix `.dat` (the name of which is assigned, on the fly, to the variable `$i`), print its name to the terminal (`echo`), and print how many lines it contains (`wc -l`). Let's say you have 1000 files in your directory. To check the length of each `.dat` file manually would take ages; to write a script to do it for you takes 6 lines and as many seconds. Let's look at another. Imagine you have a series of files `data1.dat`, `data2.dat`, `data3.dat`, etc., the script

```
#!/bin/bash

num=1
while [ $num -le 10 ]
   do
   echo $num
   head data$num.dat
   num=‘echo $[$num+1]‘
done
```

will print the first 10 lines of the first 10 files. How? First, a counter variable `$num` is set to one. Then a `while` loop is set up, which will run for as long as `$num` is less than or equal to (`-le`) 10. At the end of each iteration, the value of `$num` is incremented by one, so there are ten iterations overall.

Finally, let's look at a test:

```
#!/bin/bash

num=1
```

[6] Depending on your path, you may need to type `./script.sh`

```
while [ $num -le 1000 ]
   do
   if [ ! -e data$num.dat ]
      then
      echo File $num does not exist
   fi
   num=‘echo $[$num+1]‘
done
```

This script runs through the first 1000 `.dat` files, in the same manner as before, and prints a message if the file does not exist (the `!` bit means "not", and the `-e` bit means "exists"). It is a far better a way to determine that you're missing `data872.dat` than squinting at the screen for hours with a pen and paper...

It is also possible to use an `until` loop instead of a `while` loop.

2.8 sed

`sed` (literally, stream editor) provides a quick way to search for and replace text in a file. For example, in the phrase "I like fish", we could use `sed` to change 'like' to 'hate', as below.

◇ `echo I like fish | sed s/like/hate/`

gives

```
I hate fish
```

This replaces only the first instance on each line; if you want to replace all instances append `g`.

◇ `echo I like fish because fish is tasty | sed s/fish/soup/`

gives

```
I like soup because fish is tasty
```

but

◇ `echo I like fish because fish is tasty | sed s/fish/soup/g`

gives

```
I like soup because soup is tasty
```

Now let's say you are using a program `program` which reads parameters and data file names from some input text file `input.dat` and writes output to a text file `output.dat`. This arrangement is common with many scientific programs, especially those written before modern shells. We have at our disposal an example input file `input_template.dat` which currently specifies a fictitious data file `fish.dat`. You want to run the program multiple times in a loop, each time changing `input.dat` to specify a different data file name. The script

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

```
for (( num = 1 ; num <= 100 ; num ++ ))
   do
   sed s/fish/data$num/ input_template.dat > input.dat
   ./program
   mv output.dat output$num.dat
done
```

will run the program 100 times, each time copying the contents of the example input file to the actual input file, but replacing the word `fish` with `data$num`, such that the program first reads in `data1.dat`, then `data2.dat`, etc., and produces corresponding output files `output1.dat`, `output2.dat`, and so on until `output100.dat`. Note how the loop construction differs from our last example: `num = 1` sets `$num` initially to 1, `num ++` increments `$num` by one each time, and `num <= 100` tells the loop to stop iterating once `$num` is greater than or equal to 100.

2.9 awk

We now come to probably the most useful of the quick Unix processing utilities. At its most basic, `awk` (or `gawk`) reads through a text file, line by line, manipulates what it finds, and then outputs the answer to either the terminal or a file. Let's go back to our tables of numbers `data1` and `data2` listed earlier. Imagine we want to produce a file with three columns, one of which is column 2 of `data1`, one of which is column 2 of `data2`, and the other is the difference between them. The command

◇ `paste data1 data2 | awk '{print $2,$4,$4-$2}'`

does the job nicely, and gives

```
5.6 3.9 -1.7
6.9 3.5 -3.4
5.4 2.0 -3.4
```

`paste` produces a 4-column file, and then `awk` extracts only columns 2 and 4, corresponding to column 2 of each of the original files (the column number is denoted by the `$` sign). You will find yourself using the combination of `paste` and `awk` a lot. You can also use `awk` to do more complicated calculations. For instance:

◇ `awk 'BEGIN {sum=0} {sum=sum+$1} END {print "Average is", sum/NR}' pacific.dat`

prints the average value of all the numbers in column 1 of the file `pacific.dat`. How? First, a variable `sum` is set to zero at the beginning (heralded by the `BEGIN` statement), then awk reads each line in turn and increments `sum` by the value of column 1 (`$1`) and then, at the end, it divides the sum by the number of records read (`NR`), and writes the answer to the terminal. `awk` is especially useful for such *one-liners*, and you can find many such examples on the web[7].

awk can be used to fill-in many of bash's shortcomings when it comes to floating point arithmetic. For instance

◇ `i=‘echo $i | awk '{print $1+0.5}'‘`

[7]For example, `http://www.ce.berkeley.edu/~kayyum/unix_tips/awktips.html`

increments the value of the variable `$i` by 0.5.

Lastly, awk is useful for simple if statements. For example if I want to find the information for a particular event from a long data file, I would write:

```
◇ event=‘ awk ’{if ($1   /’event_name’/) print $0 }’ information_for_all_events.dat
‘
```

So if the first column in the file 'information_for_all_events.dat' contains the string 'event_name', the whole line is printed to the variable $event.

2.10 grep

Finally, `grep` is a useful program which picks out lines which fulfil a particular criterion, and writes them to the terminal. So, for example

```
◇ grep fish words.txt
```

prints out all the lines of `words.txt` which contain the word `fish`, and

```
◇ grep -v fish words.txt
```

prints out all the lines which *don't* contain the word `fish`. This command is very useful for excising lines of a numerical data file which are listed as `NaN`, or "no value".

Suppose you had a file `event_info.dat` which listed, in columns, earthquake event data such as: time, latitude, longitude, depth and magnitude. You could add the following two lines to a script to extract the latitude and longitude for each event if you already know the time (`$time`):

```
lat=‘grep $time event_info.dat | awk ’{print $2}’‘
long=‘grep $time event_info.dat | awk ’{print $3}’‘
```

3 Gnuplot

Gnuplot is a useful (and rather under-used) way to make quick plots of data, and it also doubles-up as a powerful calculator and parameter-fitting utility in its own right. Let's imagine we have a table of data with two columns, `pacific.dat`. Column 1 is depth to the sea floor (d), and column 2 is the gravity anomaly at the same point (g), and we would like to see how well they are correlated.

◇ `gnuplot`

```
gnuplot> plot "pacific.dat"
```

should give us a nice picture straight away, and we can see that they are indeed correlated rather well.

Gnuplot does a least squares inversion to fit a function to the data. For example, to find and plot the function of the form $g = ad + b$ which best fits the data , where a and b are free parameters, we type the following

```
gnuplot> f(x)=a*x+b
gnuplot> fit f(x) 'pacific.dat' via a,b
gnuplot> plot "pacific.dat", f(x)
```

Gnuplot has many more features, but a slightly frustrating manual (which you call up by typing `help`). As ever, the internet is a better source of advice.[8]

Finally, we show an example of using gnuplot as a simple calculator to work out $e^{-0.7} + 5$.

```
gnuplot> print exp(-0.7)+5
5.49658530379141
```

For more information talk to Laurence Cowton, or look online where there are some excellent resources available.

[8] A good place to start is `http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`

4 GMT

Although gnuplot is the preferred tool for quick plots, for maps and presentation-quality images the choice of most people at Bullard is the suite of programs that go under the heading GMT (Generic Mapping Tools). As well as producing postscript images, GMT also includes a number of useful data processing utilities. It is very well documented, and we recommend that you familiarise yourself with the official *GMT Cookbook*[9], which includes numerous example scripts which can be adapted to your own needs. The purpose of this section of the guide is to start you off by taking you through three easy examples. One point before we start: the online cookbook refers to GMT version 4.5, but Bullard has various versions, often GMT version 3.4. Certain commands mentioned in the manual (such as those concerning time series and legends) may therefore not work when you try them, so check the local man pages. Beware!

4.1 A simple Cartesian data plot

`2006.11.15_z.dat` is a text file of seismometer displacement during an earthquake off northern Japan, which we are going to make a picture of. All the following commands should be typed into a script, which can then be executed when finished.

First, define some frequently-used options as variables, which will save us typing if we want to change them.

```
infile=2006.11.15_z.dat
outfile=2006319_z.PS
rgn=-R0/3600/-7/7
proj="-JX6.5c/3c -P"
```

`$infile` is the name of the input ASCII data table (coluns time and displacement), `$outfile` the name of the output postscript graphic, `$rgn` is the plotting region (the syntax is `-R[min x]/[max x]/[min y]/[max y]`), and `$proj` is the projection (the syntax is `-J[projection type][size in x-direction]/[size in y-direction]`, where `X` means "Cartesian" and `6.5c` means 6.5 cm wide). Finally, `-P` means the paper orientation should be portrait (the default is landscape).

Then set the various text labels to sensible sizes (there are many other defaults e.g. HEADER_FONT, which you can find by typing `more .gmtdefaults4` once you have run the script):

```
gmtset ANOT_FONT_SIZE 11p LABEL_FONT_SIZE 11p HEADER_FONT_SIZE 14p
```

Then type

```
psxy $infile $rgn $proj -W2/0/0/255 -Ba500f250:"Time, s":/a2f1:"z displacement,
machine units"::."Event 2006319 recorded at HLMB":WSne -K > $outfile
```

[9] `http://gmt.soest.hawaii.edu/`

(all as one line). This command looks complicated, but isn't so bad once you get your eye in. `psxy` is the name of the program which produces postscript plots of points and lines, and `$rgn`, `$proj`, and `$infile` are the variables discussed above. The `-B` option is a bit of a mouthful, but follows a standard formula. It means, literally, "annotate the x-axis every 500 units, tick every 250 units, and label `Time, s`; then annotate the y-axis every 2 units, tick every 1 units, and label `z displacement, machine units`; then give the plot the title `Event 2006319 recorded at HLMB`". The text labels are only added to the left hand (`W`) and bottom (`S`) axes. The right-hand (`e`) and top (`n`) axes are left unadorned and so are in small-case. The string `-W2/0/0/255` means plot the data as a continuous blue line of thickness 2 units with a blue colour. In GMT, colours are expressed as red/green/blue, where each index can vary from 0-255. There are plenty of charts on the internet which allow you to find the appropriate values for your hue of choice[10]. Finally, the `-K` means that we're going to add more information to the image.

We then conclude by adding a couple of text labels using the command `pstext`

```
echo 825 2 12 0 4 CM "P" | pstext $rgn $proj -O -K >> $outfile
echo 1500 2 12 0 4 CM "S" | pstext $rgn $proj -O -K >> $outfile
echo 2500 6 12 0 4 LM "Surface waves" | pstext $rgn $proj -O >> $outfile
```

where the piped text string has the format x-position, y-position, font size, text angle, font number, justification code, text. Note that you have to add `-O` if you're plotting on top of existing information.

In a GMT script with more than one plotting command, the first line must only contain `-K`, the last line must only contain `-O`, and lines in between must contain both `-O` and `-K`. **A quick mnemonic is** K, OK, O.

Finally, you can ask the script to display your handiwork using

◇ `ggv $outfile &`

or, on some machines

◇ `gv $outfile &`

4.2 A simple map

Now let's look at how to plot a simple map showing seismometer locations, including the station at which the seismogram we plotted in the last section was recorded. Locations and names of nine stations in the British Isles are contained within a text file called `stations.loc` with the following formatting

```
-4.625 54.177 IOM
```

where column 1 is longitude East, column 2 is latitude North, and column 3 is the station code. Again, start by defining commonly-used text strings:

[10] e.g. `http://www.visibone.com/colorlab/big.html`

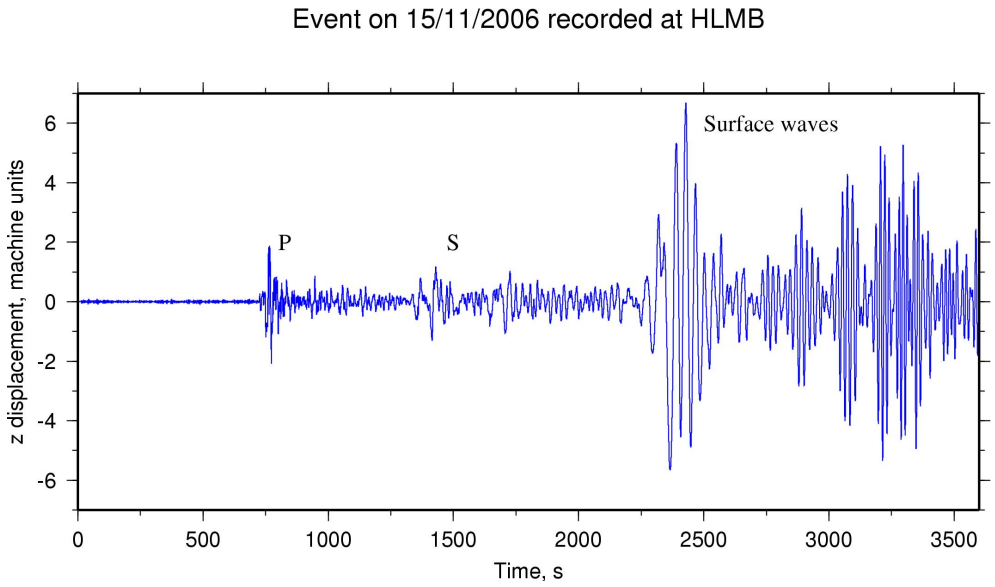

Figure 1: Plot of the seismogram in the previous exercise

```
infile=stations.loc
outfile=ukseis.ps
rgn=-R-11/3/49/60
proj="-JS-4/90/6.5c -P"
```

This time the `-R` bounds are geographical co-ordinates, and the projection (`-JS`) is a polar stereographic projection (see the Cookbook for details).

Then type

```
pscoast $rgn $proj -Di -Ba2f2 -K -G0/255/0 -S0/0/255 > $outfile
```

`pscoast` uses GMT's inbuilt coastline archive to construct a basemap of Great Britain, the `-G` and `-S` options set the colouring of the land and the sea, and the `-Di` option sets the coastline quality to intermediate to save memory. Remember to include the `-K` (we're adding more later).

Now we add the station location points as text strings on a white background

```
awk '{print $1,$2,11,0,4,"CM",$3}' $infile | pstext $rgn $proj
-W255 -O >> $outfile
```

We need the `awk` bit beforehand because `pstext` requires text strings to be in the format lon, lat, font size (11), text angle (0 degrees), font (4), justification (lon and lat refer to the **C**entre **M**iddle of the box), and the text itself. Note as this is the last command, there's no `-K`. Your final plot should then appear as illustrated overleaf.

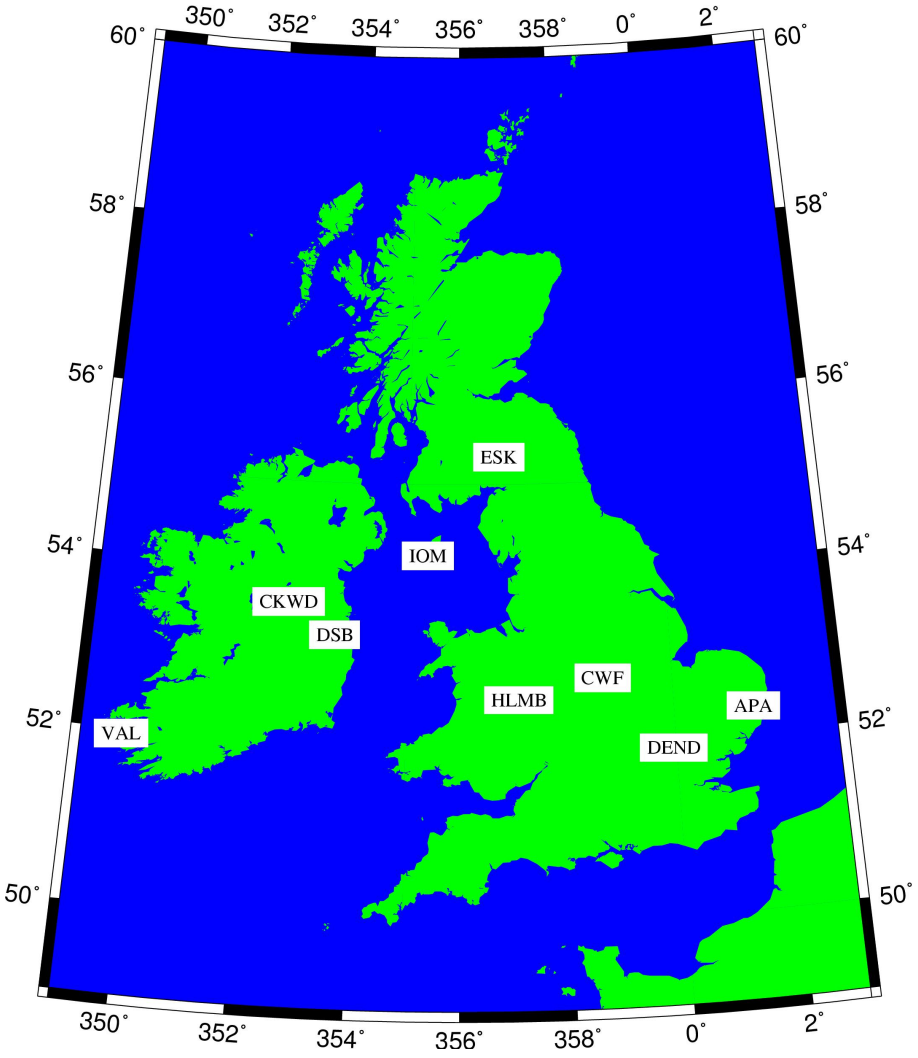

Figure 2: A simple map produced using GMT

4.3 A fancier map

One of GMT's strengths is plotting *gridded* data such as topography. In this section, we show how to plot the focal mechanism of the earthquake in the first plot on top of a shaded relief map of topography. This plot will be of sufficient quality to use directly in a thesis, presentation or paper. In GMT, the term "grid" means, technically, a netCDF file, which is a widely-used, non-ASCII, way of encoding grids of numbers (text files are easy to manipulate, but not very efficient in terms of storage). The GMT utilities `xyz2grd`, `nearneighbor`, and `surface` all convert ASCII tables of data into netCDF format, but differ in terms of how they interpolate in the gaps. We will talk later about how you can obtain this kind of data yourself.

We start as usual by defining variables (the topography grid file is called `kuril_topo.grd`).

```
infile=kuril_topo.grd
outfile=quake_map.ps
rgn=-R133/173/36/56
```

```
proj="-JM6.5c -P"
```

Note that `-JM` is a Mercator projection. We now have to define a colour scheme using the program `makecpt`. The one we're going to use is based on the *globe* palette[11], and is defined from -6000 m to +6000 m. Deeper than 6000 m below sea level, we set the colour to be pure blue.

```
makecpt -Cglobe -T-6000/6000/50 -Z > topo.cpt
echo B 0 0 255 >> topo.cpt
```

We then use `grdgradient` to calculate the shading required, illuminated in the `-A` option from the West and the North.

```
grdgradient $infile -A270/0 -Gintensity.grd -Ne0.2
```

We're now ready to start plotting. The program `grdimage` produces pictures of gridded data. The `-I` option prefaces the shading file (`intensity2.grd`) and the `-C` option the colour palette file (`topog.cpt`). We have raised the plot 3 cm (`-Y3c`) on the page in preparation for the colour scale to be added later.

```
grdimage -Y3c -Iintensity.grd $infile -Ctopo.cpt -Ba5WsNe:."Event 2006319": $rgn
$proj -K > $outfile
```

We now use the program `psmeca` to add the focal mechanism. The appropriate parameters for this event have been taken from the Harvard CMT website[12]. Type

```
echo 154.33 46.71 14 1.74 -0.56 -1.18 1.64 2.58 -0.77 28 0 0
200611151114A | psmeca $rgn $proj -Sm0.6c -O -K >> $outfile
```

again, all on one line. Now add coastlines as a thick black line using the program `pscoast`.

```
pscoast -W2/0/0/0 $rgn $proj -Di -O -K >> $outfile
```

Finally, we shift the pen down a bit and add the colour scale using the program `psscale`

```
psscale -Y-1 -D7/0/10/0.35h -E -Ba1500/:"Topography, m": -Ctopo.cpt -O >> $outfile
```

The final plot is as follows. We can see the earthquake occurred on a thrust fault in the Kuril trench.

[11] See `http://solition.vm.bytemark.co.uk/pub/cpt-city/` for a complete list

[12] `http://www.globalcmt.org/CMTsearch.html`

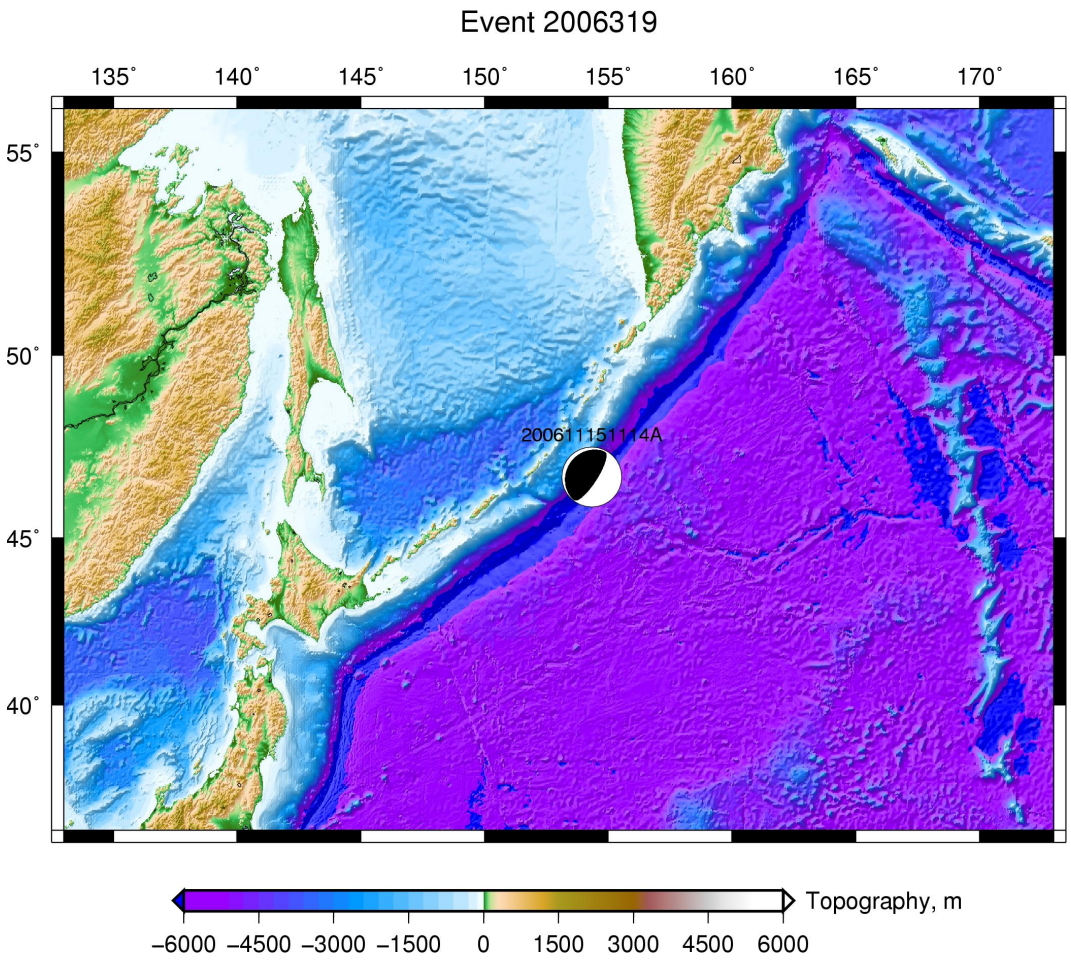

Figure 3: A fancy map showing the event in Figure 1

4.4 Other GMT programs

Below is a useful (but not exhaustive) selection.

minmax Find minimum and maximum values of numerical text files

grdfilter Apply a smoothing filter to GMT gridded data

grdtrack Extract the value of a GMT grid at lon,lat (or x,y) points listed in a text file

blockmean Spatially average irregularly-sampled lon, lat (or x, y) points listed in a text file

sample1d Re-sample a 1-D text file at user-defined points using interpolation

grdmath Add, subtract, multiply and divide pairs of GMT grids

grdinfo Get information about the size, range and resolution of a GMT grid

project Find lon, lat co-ordinates along a great circle between two points. Use in conjunction with `grdtrack` to extract topographic profiles.

pshistogram Make histograms of single-column numerical text files

5 LaTeX

LaTeXis the program which most people at Bullard use to write scientific reports (including this one). It is a markup language not unlike HTML; in other words, you write a text file with a series of formatting instructions, and the `latex` program translates that into a pretty and well-ordered document. Unlike Microsoft Word, it is not WYSIWYG (What You See Is What You Get), which can be annoying, but the reward is a superior level of typesetting, plus the automatic compilation of correct figure numbers, alphabetical reference lists and tables of contents. LaTeX is particularly good at typesetting equations, although its handling of graphics can be exasperating at times.

Say you have a text file `report.tex`,

◇ `latex report.tex`

will produce a formatted file `report.dvi`, which you can view on some computers using

◇ `xdvi report.dvi &`

To turn that into a printable pdf, use

◇ `dvipdf report.dvi`

A particularly useful add-on is BIBTeX, which allows you to link-in databases of references. It is a good idea to build up such databases as you go along. To get the citations right, you need to run `bibtex` once and `latex` three times.

◇ `latex report.tex`

◇ `bibtex report`

◇ `latex report.tex`

◇ `latex report.tex`

The best strategy with LaTeX and BIBTeX is to work with a *template*, a fine example of which has been made available by Jessica Irving[13]. Note that EndNote users can also export their reference databases in BIBTeX style. Also particularly recommended is the *LaTeXuser's guide and reference manual* by Leslie Lamport, which exists in various forms around Bullard. Just to get you started, below is a listing for a minimal article, which you are encouraged to play with.

```
\documentclass[12pt]{article}
\bibliographystyle{plain}
\usepackage{graphicx}

\begin{document}

\title{My first \LaTeX~document}
\maketitle

\section{Introduction}
```

[13]Copy using `cp` from `/home/jcei2/For_Students/latex`

```
\LaTeX is commonly used by people in Bullard, whereas Microsoft Word is commonly
used by people downtown, and by the rest of the world.  \LaTeX is excellent for
equations and for documents with large numbers of lists, subsections and references.
It is less good for Desktop Publishing and anything with lots of figures.  Trying
to change the basic page format involves entering a world of pain, which is why all
the theses from Bullard look the same.  You would not want to try and introduce
\LaTeX into an advertising agency full of Creative types.

\subsection{Equations and references}

Here is an equation.

\begin{equation}
\frac{\partial T}{\partial t} = \kappa\frac{\partial^2 T}{\partial z^2}
\label{diffusion}
\end{equation}

You can reference that as Equation \ref{diffusion}.  It is given in many papers,
including \cite{example}, where the text inside the {\tt cite} command is a key
in your BibTeX database.  To finish off, here is a figure.

\begin{figure}[h]
\begin{center}
\includegraphics[width=14cm]{quake_map.ps}
\caption{An example of a postscript figure}
\end{center}
\end{figure}

\clearpage

\bibliography{test}

\end{document}
```

In this example, the contents of the file `test.bib` are

```
@article{example,
AUTHOR = "D. McKenzie",
TITLE = "Some remarks on the development of sedimentary basins",
JOURNAL = "Earth Planet. Sci. Lett.",
VOLUME = "40",
PAGES = "25--32",
YEAR = "1978"
}
```

LaTeX is also available for Windows, and has various linux / Mac GUIs (e.g. `kile` or `Texworks`, which are very good.

6 Programming with fortran

There will come a time when you encounter a task that is simply too much for bash, `awk` and GMT to handle by themselves, and at that point you have three options. The first is to give up, wave your hands about and do some serious colouring-in; the second is to get someone else to do it for you; and the third is to write a program yourself. Sadly the first will only work until you get the reviews back, and the second you can only really do if you are a supervisor. Luckily the third is not nearly as scary as it sounds.

There are a huge number of programming languages out there, and most of them bear alarmingly little resemblance to English. Luckily, fortran, the most commonly-used by the scientific community, is not too bad in that respect. In this introductory session, we will go through a very simple program to illustrate the basics of the language. It turns out that you can do this task in GMT, but it is a good example of the kind of thing you may find yourself writing, and it is easy to see how the code could be adapted to handle more complex manipulations.

The example below was originally written for fortran 77, which although very widely used within Bullard, is now somewhat creaky. A newer version, eg. fortran 90/95/2003/2008 may well be easier to use, as some anachronistic limitations (line lengths that would fit onto a punch-card, for example) have been removed. The code below can also be compiled using gfortran, which is now the preferred compiler.

To convert geographical co-ordinates to Eastings and Northings in km using a polar stereographic map projection (`-JS` in GMT), you need to use the following transformation

$$x = 2R \tan\left(\frac{\pi}{4} - \frac{\phi}{2}\right) \sin\lambda \tag{1}$$

$$y = -2R \tan\left(\frac{\pi}{4} - \frac{\phi}{2}\right) \cos\lambda \tag{2}$$

where λ is longitude and ϕ is latitude, R is the radius of the Earth, and all angles are in radians. We wish to read in a text file of geographical co-ordinates and transform them into x and y.

The first thing to say about fortran is that it is a *compiled* language, unlike bash, which is an *interpreted* language, i.e. one where your commands are executed by the system as you go along. Compilation is the process by which your (near-) English is translated all at once into a block of machine code. The most commonly used compiler for fortran under linux is called `gfortran`. Say our program is called `polar.f`, we would compile it using

◇ `gfortran -o polar polar.f`

and then run it by typing

◇ `./polar`

The code for our program `polar.f` is as follows.

```
      PROGRAM POLAR
```

```
      IMPLICIT NONE

C     Declare variables

      REAL*8 LON, LAT, DAT, X, Y, PI, RAD, RADIANS, SF

c     Constants

      PI = 3.14159265
      RAD = 6371.0e3

C     Open input file of data

      OPEN( 1, FILE='input.dat', STATUS='OLD' )

C     Open file of data which output will be written to

      OPEN( 2, FILE='output.dat', STATUS='UNKNOWN' )

C     Prompt user for output data scale factor

      WRITE(*,*) 'Please enter scale factor (1 for m, 1000 for km)'
      READ(*,*) SF

C     Read-in the file line by line until the end

      DO 20

            READ( 1, *, END=30 ) LON, LAT, DAT

C           Convert to radians

            LON = RADIANS( LON, PI )
            LAT = RADIANS( LAT, PI )

C           Calculate X and Y

            X = SIN( LON )
            X = X * TAN( PI/4.0 - LAT/2.0 )
            X = X * 2.0 * RAD / SF

            Y = COS( LON )
            Y = Y * TAN( PI/4.0 - LAT/2.0 )
            Y = -Y * 2.0 * RAD / SF

C           Write to file

            WRITE(2, 10 ) X, Y, DAT
```

```
10      FORMAT(3(G14.6,1X))
20      CONTINUE
30      CONTINUE

        CLOSE(1)
        CLOSE(2)


        END

        REAL FUNCTION RADIANS( DEGREE, PI )
        REAL*8 DEGREE, PI
        RADIANS = DEGREE * PI/180.0
        RETURN
        END
```

Let's go through the different parts of the program.

1. Some preliminaries first. Lines beginning with `C` are *comments*, and are purely for the benefit of the reader. A charming hangover of the days when programs were written on punch cards is that fortran 77 code must not occupy the first six columns of the page, and must never overrun column 72 (although you can put continuation statements in column six which spread long statements over several lines).

2. The first set of statements *initialise* the variables. `LON` is longitude, `LAT` is latitude, `DAT` is whatever the geographical data is in the third column, `X` and `Y` are the transformed X and Y co-ordinates which we wish to find, `PI` is π, and `RAD` is the radius of the Earth, both of which are defined in the next line. Finally, `SF` is a scale factor. All these variables are of type `REAL*8`, which means that they are floating point numbers stored to double precision.

3. We then open the file `input.dat` in which our geographical data is stored, and the file `output.dat` to which we will write our transformed data. The status `OLD` means that the file must already exist, whereas the status `UNKNOWN` makes no such presumption.

4. The next bit of the code prompts the user to enter the scale factor (i.e. whether we want our distances in m or km). `WRITE(*,*)` writes the subsequent text to the terminal, and `READ(*,*) SF` means "read whatever the user types and assign it to the variable `SF`". A more complete program would make sure the answer was a number rather than a word.

5. We then read the input file line by line and do the transformation. The construction `DO 20 ... 20 CONTINUE` means "do whatever is represented by the ... forever" and is called an *infinite loop*. The stuff in-between can broken down as follows

   (a) `READ(1,*,END=30) LON, LAT, DAT` means read the next line of the file labelled 1 (`input.dat`) and assign column 1 to the variable `LON`, etc. The instruction `END=30` means jump to the line labelled 30 when the end of the file is reached. Line 30 is after the end of the loop so the iteration will continue only as long as there is data to read.

(b) The conversion to radians is handled by the *function* `RADIANS` which is defined at the end of the program. Functions are a useful way of avoiding repeating the same bit of code over and over again. You can also use *subroutines* to this effect.

(c) The calculation of `X` and `Y` is then performed using fortran's built-in maths functions. Note how quite a long formula is broken down into several steps to make the code cleaner. Note that in programming the apparently illogical `X = X + something` means "add something to X".

(d) We then write our new values for `X` and `Y` to file 2. The formatting instruction is given in line 10, and says "make each field 14 digits wide and separate by 1 space, with each number having no more than 6 decimal places". If you don't need to have your output beautifully lined up in columns, just use `*` instead, which is *free format*.

6. We then close the two files and stop the program.

There is a lot more to fortran, of course, but not *that* much more, and there are some excellent online resources to help you along.[14]

For any sort of serious maths, there are numerous pre-written subroutines which you can slot into your code without the need to try and write them yourself. Particularly recommended is *Numerical Recipes* by Press et al., which comes as a textbook, a comprehensive set of subroutines, and a corresponding set of example codes. There are numerous copies floating around Bullard. There are also excellent tutorials available from such sources as the Cambridge Physics Department. (www.mrao.cam.ac.uk/ rachael/compphys/SelfStudyF95.pdf)

Finally, fortran is not the only programming language. `perl`, `python`, `C` and `C++` have all been used by people in recent times at Bullard. Google for tutorials to start you off.

7 Printing

To print a postscript file, use the command:

◇ `lpr -P[printer name] [document name]`

If the file is an ascii text file, it needs to be converted to postscript before the printer can print it. Use the following command to print:

◇ `a2ps -P[printer name] [text file]`

A list of printer names can be found on the Bullard internal website[15]. Use `acroread` or `evince` to display and print pdfs, and `gimp` to display and print bitmaps (JPEGs etc.).

If you have trouble printing, the commands

```
lpq -P[printer name]
lprm [job number]
```

[14] The best being `http://www.star.le.ac.uk/~cgp/fortran.html`

[15] `http://sirius:631`

will list the current jobs on the print queue with their status and remove a print job from the print queue, respectively (the job numbers are listed with `lpq`'s output).

To see all available printers type

◇ `lpstat -p`

Always print in black and white if you can, since colour printing is more expensive and uses a lot more energy. We have two new Photocopier printers, which are the preferred printers to use since we effectively rent them, so need to print over a certain number of pages to make them worthwhile.

Below is a list of printers at Bullard:

Photocopier Colour and mono printer, plus copies and scans, printer room at the back of the Old House (the best option for printing)

Drum-colour Colour/mono printer, plus copies and scans, Drum Building

Drum-mono Mono printer, plus copies and scans, Drum Building (same phyiscal object as above, different defaults)

wolfson-mono Mono printer, Wolfson Building

wolfson-colour Colour printer, Wolfson Building

Plotter-2 Colour A0 plotter for printing posters, behind the part III room, Wolfson Building (this is the new plotter which is better and quicker than the one below)

plotter Colour A0 plotter, behind the part III room, Wolfson Building.

8 Doing things remotely

8.1 Accessing another machine

To access other computers and run programs on them remotely use a *secure-shell* (`ssh`). The syntax is

◇ `ssh -X [your login name]@[name of machine you want to log into]`

e.g.

◇ `ssh -X jrw65@sirius.esc.cam.ac.uk`

To access computers within the network, you can omit the user and domain names:

◇ `ssh -X sirius`

To log out, simply type `logout`, `exit` or CTRL-D.

To access your computer from outside the network you'll need to first `ssh` into `sirius` and then `ssh` into your computer. The `-X` option allows X-windows forwarding, which means you'll be able to view graphical interfaces too. If on Windows, either install the cygwin emulator[16], or download the excellent PuTTY[17]. PuTTY can also be used with XMing to enable graphics capabilities.

[16] `http://www.cygwin.com/`

[17] `http://www.chiark.greenend.org.uk/~sgtatham/putty/`

8.2 Transferring files between computers

To transfer files from one computer to another, use `scp`. For instance

◇ `scp send_this.text jrw65@sagitta:/space/jrw65`

transfers the file `send_this.tex` from wherever I am now to the directory `/space/jrw65` on the workstation 'sagitta'. It can also work in reverse, so

◇ `scp jrw65@sagitta:/space/jrw65/fetch_this.text .`

retrieves the file `fetch_this.tex` and puts it in my present working directory.

Another option is `sftp` which opens an interactive connection to the remote machine, allowing you to navigate, list files, make directories and print your current directory on both at the local and remote machine (with `lcd, lls, lmkdir, lpwd, cd, ls, mkdir` and `pwd` respectively.)

◇ `sftp maf49@sirius:/home/maf49`

◇ `cd ./folder_on_sirius/`

◇ `lcd ./folder_on_local/`

then

◇ `get file_of_interest.txt`

or using wildcards

◇ `mget file_of_*.txt`

You can also move files the other way, using

◇ `put transfer_this.tex`

and

◇ `mput transfer_this_*.tex`

Finally, you can use WinSCP[18] to transfer files between Windows and linux machines.

8.3 Password-less `ssh` and `scp`

To `ssh` into, and `scp` to/from, another machine at Bullard without entering your password (necessary for running backup scripts to other machines every night) you'll need to set up a private/public key pair, using `ssh-keygen` [19]. Type

◇ `ssh-keygen -t rsa`

and then copy the contents of ∼`/.ssh/id_rsa.pub` to ∼`/.ssh/authorized_keys`

Alternatively, this website explains all the steps: `http://linuxproblem.org/art_9.html`

As your home directory is identical on every machine in Bullard, you'll not be prompted for your password again with `ssh` or `scp`.

[18] `http://winscp.net`

[19] see `http://rcsg-gsir.imsb-dsgi.nrc-cnrc.gc.ca/documents/internet/node31.html`

8.4 Extracting files from an FTP site or website

To access files on an FTP site or website, use either `wget` or `ftp`. For instance:

◇ `wget http://www.lotsofdatahere.com/data/fish*.txt`

retrieves all `.txt` files starting with `fish` on the aforementioned website. `ftp` is a more involved interactive file transfer utility, i.e. you log in and change directories on the site itself. If using *anonymous ftp*, you use `anonymous` as your login name and your email as the password. For example:

◇ `ftp ftp.lotsmoredatahere.com`

```
ftp> anonymous
ftp> jrw65@cam.ac.uk
ftp> cd /pub/data/
ftp> mget soup*.txt
ftp> quit
```

8.5 Compressing and archiving

Compressing and *archiving* are very useful when transferring data as they reduce both the size of files and the number of files. They can be done with the `tar` and `gzip` commands. For instance

◇ `tar -cvfz compressed_archive.tgz directory1/ directory2/`

produces a compressed file `compressed_archive.tgz`, which contains the contents and file structure of `directory1/` and `directory2/`. To uncompress, use

◇ `tar -xvf compressed_archive.tgz`

If you want to compress a single file, use

◇ `gzip file.dat`

and

◇ `gunzip file.dat.gz`

to uncompress the file (which was renamed by `gzip`). These functions are often easily performed using the file browser in linux.

9 Wireless access

Okay, so you've just bought a shiny new laptop. You'd like internet access anywhere at Bullard without having to carry an extra-long network cable around with you. You'd also prefer not to have to enter a password *every* time you connect. Actually, come to think of it, you'd like the same access in college ... and maybe even other departments when collaborating with friendly folks there. Hey, wouldn't it be cool if you could automatically connect at any UK university, or (let's keep dreaming) universities across the globe?

Well, let us introduce you to eduroam. eduroam (educational roaming) is the secure, world-wide roaming access service developed for the international research and education community.[20] With it, you can automatically connect to the internet at an ever increasing number of institutions around the world.

All you need to do is to set up your laptop (or smartphone) according to the instructions at:

`http://www.cam.ac.uk/cs/wireless/eduroam/localusers.html`

Your "eduroam identifier" is your Cambridge email address (including the `@cam.ac.uk`) and your "network access token" (password) can be accessed from:

`https://tokens.csx.cam.ac.uk/`

for which you'll need your Raven login details.

10 Backing up

At Bullard we back up our personal machines to high-capacity RAID storage systems, but it is **y**our own responsibility to back up your work. Backup as often as possible. Hard discs do fail, and lost work is not an excuse for missing deadlines. There are several RAID machines, generally each group has one or two that are probably in another building, so ask your group for details.

Home directories are automatically backed up, but only up to a 10Gb limit. For the majority of your work, you will need to set up your own backup script.

When backing up, we recommend you use `rsync`. It takes a while to run the first time you use it (as it has to back-up the entire contents of the desired directory) but the next time you come to back-up that directory, only the files you have altered since the last back-up are sent. `rsync` thus saves you lots of time needlessly backing-up files that you haven't altered since the last time you backed up.

`rsync` can be used with several option, defined as a variable (for use in a script)below:

```
stdoptions=" --stats --compress --archive --exclude='*~' "
```

This says that you want the backup to compress data when sending it, to provide statistics on the number of files backed up (among other things), to avoid backing up files ending in ~, and to use archive mode.

`--archive` is equivalent to:

```
--recursive --perms --links --times
```

This means that archive mode backs up recursively (it includes subdirectories), preserving permissions, modification times and links between files.

rsync is then run in the script as follows:

[20] `http://www.eduroam.org/`

```
rsync $stdoptions /space/arh79/ arh79@lapis:/raid1/arh79/procyon_backup/ > $logfile
```

where lapis is a RAID machine and `logfile=/space/arh79/backup/backups.log` . If you are running rsync at a busy time you might want to preface your command with (for example) `nice -n+5` to give the job low priority on the system.

The example provided will write standard output (e.g. stats) to the logfile, and will spit any errors out in the terminal. It is highly recommended to automate your backup script to regularly run at times of your choosing. This can be done with `cron`. You first need password-less login, see the section above or [21].

To edit your crontab type

```
crontab -e
```

and crontab will open in your default text editor.

My crontab looks like:

```
23 3 * * * /space/arh79/backup/procyon_backup.bash
```

The command is in the form `m h dom mon dow command`. The line runs a back-up to lapis on Monday to Sunday at 3.23 am (23 min, 3 hour, * for every month and year and * for every weekday). The back-up scripts essentially contain the rsync command above, but mine also includes the line:

```
mail -s "Backup to Lapis run" arh79@cam.ac.uk
```

This sends me a blank email with the above subject to tell me that the script has run. Please note that this doesn't tell you it is working properly. Check the backup or the logfile to check that!

`Cron` will want to email the output of `rsync` (any errors in the example script) and so you need to specify an email address to send this too, otherwise it will send it to root (and Ian does not want to see your back-ups every day...). At the start of your `crontab` add a `MAILTO` command, e.g.

```
MAILTO=arh79@cam.ac.uk
```

If you are backing up two computers to each other, make sure you use `--exclude` for the directories where backups are stored, or you will create a loop and fill up all the machine's hard drive with backups of backups of backups.

11 The cluster

Currently the cluster has 256 cores for general use and another 256 cores that belong specifically to Arwen Deuss (so you will need to ask to use those cores). The cluster runs with two heads, atlas

[21] `http://linuxproblem.org/art_9.html`

and maia, and jobs can be submitted on either. Refer to the website [22] for a good user guide. If you want to use the cluster, you will need to ask Dave Lyness for a login.

The principle uses are to run the same program lots of times with different inputs (i.e. do a parameter search) and to write individual programs which split the control flow into parallel processing streams, and thus run much faster than they would on a single machine. Each core has 8GB of memory, so can handle large matrices but can't be used for long term storage. Use the RAID machine of your group to store large volumes of data.

Login to the cluster with `ssh atlas` or `ssh maia`. Scripts are submitted from your home directory and are sent to a node. To submit a script `job.sh` to the cluster, type

◇ `qsub -l walltime=10:00:00 job.sh`

This will submit the script with a running time of 10 hours (the default walltime is 24 hours, but it is good to specify the walltime anyway). The script will work from your home directory unless otherwise stated, and should look something like:

```
#!/bin/bash
RUNDIR=$HOME/data1
mkdir -p $RUNDIR
cd $RUNDIR
$HOME/program
```

This example script will create a data directory `data1` and will run the program `program`, which is stored in your home directory, on one node. The – probably copious if it's an inverse program – output will be dumped into `data1`. You can submit multiple scripts to use multiple nodes, although it is poor form to use very large numbers of nodes without consulting other users first.

12 Where to get common types of data

12.1 Global topography and gravity

12.1.1 Matt F's new fave topo

Google "ERDDAP" - follow the obvious link (a subdomain of coastwatch.pfeg.noaa.gov). Then you can search the database of a few hundred datasets, many of which are oceanography related, but many are generally useful.

This website includes a very simple way of downloading data from specific regions, with lots of alternative topo datasets available. The very best feature of this website though is the dizzying array of different formats the data is available in.

Check this website out, even if you don't need topo data...

[22] `http://maia`

12.1.2 Gravity, and other topo

For medium-resolution topography and marine gravity anomalies, the satellite geodesy section of the Scripps Institution of Oceanography is your best bet.

`http://topex.ucsd.edu/`

You can retrieve text files of topography and gravity within a box (in this case, 10-20 E, 10 S - 10 N) directly using the following commands (all on one line)

◇ `wget -O topography.file "http://topex.ucsd.edu/cgi-bin/get_data.cgi?north=10&south=-10&west=10&east=20&mag=1"`

◇ `wget -O gravity.file "http://topex.ucsd.edu/cgi-bin/get_data.cgi?north=10&south=-10&west=10&east=20&mag=0.1"`

For onland gravity anomalies, coverage is more patchy than at sea. The GRACE satellite mission provides globally-uniform data down to a wavelength of $\sim$ 400 km, which can be downloaded from

`http://www.csr.utexas.edu/grace/gravity/`

Higher-resolution data for North America, Australia and Great Britain is available at Bullard (see Mark Hoggard). Other data usually has various commercial strings attached (talk to Dan McKenzie if seriously interested).

You can convert your text tables into GMT grids using variations of the following command

```
xyz2grd -R10/20/-10/10 -I0.02 -Gtopography.grd -V
```

where the `-I` command is the grid cell spacing in degrees.

12.2 Ultra-high resolution (SRTM) topography

Talk to Simon Stephenson.

`http://SRTM.csi.cgiar.org/` – for ArcGIS ArcAscii files.

`http://seamless.usgs.gov/` – for GMT plotting

12.3 Seismograms and earthquake locations

IRIS is the best starting point for global seismic data. Data can be manually downloaded from Wilbur

`http://www.iris.washington.edu/wilber`

or automatically requested via breqfast

`http://www.iris.edu/SeismiQuery/breq_fast.htm`.

Event data catalogues can be accessed through the USGS

http://neic.usgs.gov/neis/epic/epic_global.html

or use the EHB catalogue

http://www.isc.ac.uk/ehbbulletin/

Focal mechanisms and psmeca inputs are provided by the Harvard CMT catalogue

http://www.globalcmt.org/CMTsearch.html

12.4 Marine sediment thickness

http://www.ngdc.noaa.gov/mgg/sedthick/sedthick.html

12.5 Ocean crustal ages

http://www.earthbyte.org/Resources/agegrid2008.html

If you have problems reading the 2008 grids in GMT, consider using the 1997 grids instead (they are similar enough).

12.6 Global seismic tomography

A commonly-used model is Jeroen Ritsema's S40RTS, which can be downloaded from the following website together with a clever set of plotting scripts

http://www.geo.lsa.umich.edu/∼jritsema/Research.html

However, it has a rather low resolution and is not necessarily cutting edge. If you're seriously interested in making tomographic interpretations, talk to Keith Priestley or one of his students or postdocs first.

12.7 A global crustal thickness and sediment model

http://mahi.ucsd.edu/Gabi/rem.dir/crust/crust2.html

12.8 Plate boundaries and graphical plate reconstructions

http://www.scotese.com/

http://element.ess.ucla.edu/publications/2003_PB2002/2003_PB2002.htm

http://www.ig.utexas.edu/research/projects/plates/

12.9 Global heat flux measurements

http://www.heatflow.und.edu/index2.html

13 FAQs

13.1 How do I kill a running process?

If you have a script or program running in the terminal that needs to be stopped prematurely, press CTRL-C. If your system is runing slowly, run `top`. To kill a command from this window press `k` then the process ID. If your system has frozen, try logging into a different terminal by pressing `ctrl atl F1` , then running top. If this fails log in from elsewhere and use the `kill` command. You will need to know the process ID, which you can find by typing `ps aux`.

13.2 How do I make a poster in linux?

There are several ways to make a poster: Open-Office, which is probably the simplest and easiest to manipulate; `inkscape`, a bit like Corel Draw; or, if you're feeling brave, LaTeX. Print it using **Plotter-2**, upstairs in the Wolfson building.

13.3 How do I turn a postscript into a bitmap?

Use `convert`[23], e.g.

◇ `convert -density 300 image.ps image.jpg`

`convert` is an incredibly powerful image manipulation tool, and can easily be called from scripts to rotate, crop, extract, resize and convert images between a mind-blowing array of formats. Consult Laurence Cowton

You can also load the postscript using `gimp` or `inkscape` (make sure you set the resolution to at least 200 dpi, otherwise the image will appear fuzzy).

13.4 How do I turn a postscript into a pdf?

Easy. Use `ps2pdf`.

13.5 How do I digitise a figure in a paper?

Use the brilliant `g3data`. You will first need the figure in a bitmap format; if you can't get one directly from the journal website, either scan a paper copy, or take a screen-grab and crop using `gimp`. Consult Laurence Cowton or Mark Hoggard

[23] See `http://www.imagemagick.org/script/convert.php` for manual

13.6 How do I make an animation?

There are numerous utilities (e.g. `gifmerge`, `gifsicle`) that produce *animated gifs* out of a sequence of individual frames, and which are viewable in most web browsers and in PowerPoint. You can convert GMT postscripts to gifs using `convert`.

For longer or high resolution animations, it may be better to make a .avi video. This can be done with a program called `mencoder` but the options can be hard to figure out. Consult google or Mark Hoggard

13.7 How do I transform data between (lon, lat) and (x,y) in different projections?

Use the program `cs2cs` or GMT's `mapproject` . Ask Mark Hoggard for details.

14 Other stuff

Although the primary purpose of this guide is to introduce you to tools used universally around Bullard, there are numerous other programs and languages which are used frequently by particular groups. Following is a brief, but my no means comprehensive, list, including whom you should ask for help.

14.1 SAC

Seismic Analysis Code: used for displaying, picking and filtering seismograms. Check out the manual and tutorial at `http://www.llnl.gov/sac`. Consult Charlie Schoonman, Rob Green or Andy Howell.

14.2 Seismic Unix

Used for displaying, picking, filtering and modelling seismic data. Fabulous for use in seismic processing scripts that are automated. Consult Matt Falder.

14.3 Perl

A sort of awk-on-steroids; its disciples swear by it. Recommended reading is *Learning Perl*, the llama book, published by O'Reilly.

14.4 Python

A powerful scripting language, that is more user-friendly than fortran. The Computing Service run useful courses (1 day courses, several a term) about Python. Recommended reading is *Learning Python*, the wood rat book, published by O'Reilly. There are also several applications which make

it easier to mess around in python, including the very useful ipython and ipython notebook (soon to be called Jupyter), which have autocompletion features and much easier access to man pages. Useful modules in earth sciences include Obspy (`http://www.obspy.org/`) and the basemap toolkit for matplotlib. Consult Ian Frame, Sanne Cottaar, Chris Richardson, Andy Howell or John Rudge.

14.5 Obspy

A python toolbox for seismology. Versatile and widely used by Bullard seismologists (and tectonics, whatever they count as). There is an excellent tutorial at `http://www.obspy.org/`.

If you are using it for a task formerly done using (for example) SAC, be careful, because some default options are different to their equivalent functions in other software. One example is that the anti-alias filter in `tr.decimate()` can introduce a phase shift unless you choose to disable it. Consult Sanne Cottaar, Rob Green, Jenny Jenkins or Andy Howell.

14.6 C/C++

Very versatile languages, with many preexisting libraries. Not used widely at Bullard, for no particular reason.

14.7 R

A superior statistical package, though underused. Consult John Rudge.

14.8 Matlab

An interpreted environment for doing numerical computations with matrices and vectors. It is also good for signal processing, plotting, algebra and much more. Consult Tim Greenfield.

One large disadvantage is that a licence for Matlab will cost you [24] 76 for a licence. An excellent (free) alternative is Octave.

14.9 Octave

An almost perfect clone of matlab. It will run almost any "matlab" script, and you can have it on your own computer for free.

Octave is based on python, and it would be sensible to consider whether doing your task directly in python may be a better alternative, as python is, if done right, just as fast, and far more powerful than either octave or matlab.

[24] or your supervisor if you are persuasive

14.10 ArcGIS

Powerful proprietary program for modelling, displaying and overlaying data on landscapes, widely used in industry and by the active tectonics community. Consult Simon Stephenson, Fred Richards or Veronica Rodriguez-Tribaldos.

14.11 Google Earth

The quickest way to get pretty 3-D landscape plots. Consult Simon Stephenson.

14.12 Petrel

Proprietary program owned by Schlumberger and installed on a number of machines on site. Displays industry SEG-Y seismic data and allows you to pick out horizons and tie in wells. Consult Laurence Cowton.

14.13 Omega2

Large package for the processing of industry seismic reflection data, also owned by Schlumberger and available on selected machines. Consult Alex Dickinson or Kathy Gunn.

14.14 Git

Version control software, used to store backups of files as development progresses. Very useful for working out what you (or someone else) were thinking when you made a particular set of changes. Also very useful for collaborative editing. Possible to make online backups using github. Consult Ian Frame or Andy Howell, or the tutorial at `https://try.github.io`.