Introduction to unix/linux I

A familiarity with unix or linux and particularly the command-line interface is important in computational science. Even if you use graphical tools in your day-to-day work, at some point it is likely that you will have to access a compute server remotely or find that you need the flexibility provided by the command-line. The first set of sessions in the Mathematics: Computational Classes are to introduce you to working on the command-line and the C++ language.

There are many programs which do the same thing. You are allowed to use whatever ones you like best though command-line tools are *strongly* encouraged. The notes provided do not cover all possibilities. In particular, I have chosen to use bash (Bourne Again Shell) as the unix shell as it is widely used as the default shell. Other possibilities include csh, tcsh and zsh: feel free to try them. Some commands are the same between shells, some are different.

Similarly there is a variety of terminals you can use to run the shell in: they have more similarities than differences. OSX has its own terminal and xterm (supplied by X11), which is the same as that found in linux. Linux distributions tend to have several: the different desktops all have their own versions as well as the ubiquitous xterm and others such as aterm. I will switch between different ones.

The choice of text editor (more on this later) is particularly personal and is one we leave to you.

These sessions are aimed at introducing you to a command-line interface and so most of the time will be spent trying things out. If you wish to do this elsewhere (e.g. at a CMTH workstation), that is fine. I will be here to answer any questions. Experimentation is encouraged!

Contact

James Spencer

email: j.spencer@imperial.ac.uk

website: http://www.cmth.ph.ic.ac.uk/people/j.spencer

Unix philosophy

A key idea behind all unix systems is to have many small programs, each of which perform a specific task, rather than large "jack-of-all-trades" programs. To accomplish complicated tasks, tools can be connected together. This flexibility makes it possible to combine utilities to achieve what we want without having to write the utilities ourselves; for example data can be sorted without having to write a sorting program ourselves.

Additional resources

GNU/Linux Command-Line Tools Summary by Gareth Anderson
A little dated but an excellent and comprehensive introduction to the Linux command line. http://tldp.org/LDP/GNU-Linux-Tools-Summary/html/GNU-Linux-Tools-Summary.html

Advanced Bash-Scripting Guide by Mendel Cooper
A thorough exploration of scripting and unix utilities but starts without assuming any previous knowledge and gives lots of examples. Invaluable. http://tldp.org/LDP/abs/html/

I have borrowed from both of these in places.

Bash reference manual
The ultimate reference guide. http://www.gnu.org/software/bash/manual/bashref.html

Launching the terminal

The Mathematics: Computational Classes require command line tools and a C/C++ compiler: the gcc suite (http://gnu.gcc.org) is free and includes what is widely regarded as the standard C/C++ compiler.

Linux

A terminal can be launched in most Linux distributions from the main menu: it's typically in the Accessories sub-menu. Some distributions can launch the terminal from the desktop menu (right click on the desktop). Some other (free) programs will be needed: these will be mentioned as needed.

Mac OSX

Enter terminal into the spotlight search box and select the Terminal application. I suggest you then add it to the dock permanently for convenience. The iTerm application (a free download) is a much nicer terminal to use.

Windows

Cygwin (http://www.cygwin.com) provides a Linux-like environment for Windows, including gcc.

Command format

Commands are typically in the format:

```
command [options] compulsory_arguments
```

Throughout the notes the above format will be used to indicate shell commands. The output from the commands will not (in general) be given: try out the commands to see what happens! Commands referred to in the main text are formatted like *this* (at least the first time they're mentioned).

Options are usually signified with either a single dash ('-') or a double dash ('--'). A single dash is used fort short options (i.e. a single letter) whereas the double dash is used for options which are words. Short options can usually be grouped together:

```
ls -l -h
ls -lh
```

Short options often have more verbose equivalents. Both long and short options can take arguments (which can stop short options being grouped together).

Help

Most commands come with help which briefly explain their common options and how to run the command:

```
open -h
open --help
```

The *open* command is only really useful under OSX, where it is used to open files and directories. In fact, the Linux *open* command is completely different (and doesn't even have a help option) and is used only in C/C++ programming. Try also *ls --help*.

More information can be obtained from the *man page* associated with the command:

```
man ls
man man
```

The man page can be scrolled forwards and back and searched by typing / followed by the search string.

But what if you don't know what the relevant command is in the first place? You can search for it either using google or on the command line using *man*:

```
man -k "copy file"
```

man -k searches a short description of the command for the string provided (in the above example, "copy file"). *man -K* searches the entire man page but this is much slower. Sometimes *man -k* lists many commands (try *man -k copy*!) but at least it provides a starting point.

There are quite a lot of differences in the options for BSD commands (used in OSX) and those in Linux. For this reason I will rarely give options to commands: please look at the help or man pages for these. Some commands are used in examples but not explained: these should be tried and their functions looked up using the man pages.

Files and directories

The *ls* command lists files and directories. With no path specified it lists the contents of the current directory:

```
ls
ls /tmp
```

The string after *ls* is used to list only select files or directories. You can use the ? and * wildcards, which match a single letter and any (or no) letters respectively:

```
ls /t?p
ls D*
ls Documents Downloads
```

Directories are separated using a forward slash. The filesystem starts from the top-level or "root" directory, denoted by /. A path to a directory or file can be specified either as an absolute path (i.e. specified starting from the root) or as a relative path (specified relative to the current working directory). A single dot, '.', represents the current directory and a double dot, '..', represents the parent directory. *pwd* prints the current directory you're in.

You can move around the filesystem using the *cd* command:

```
cd /tmp
pwd
cd -
pwd
```

cd - is special and returns you to the previous directory you were in. Running *cd* without specifying a directory returns you to your *home* directory. The home directory is the directory which contains your files, directories and settings. Generally you only have the necessary permission to create, delete and modify files and directories within your home directory unless another user has granted you permission to do so with their files.

mkdir and *rmdir* create and delete (empty) directories:

```
mkdir my_dir
rmdir my_dir
```

cp copies files and directories and *rm* deletes files and directories:

```
echo "test" > test_file
cp test_file test_file_2
rm test_file*
```

mv is used to move or rename files and directories:

```
mv file1 file2       # Renames file1 to file2
mv file1 dir1        # Moves file1 to directory dir1
mv file1 file2 dir1  # Moves file1 and file2 to directory dir1
```

cp, *mv* and *rm* have many options governing their behaviour (especially for working with directories). Refer to the man pages for more details.

It is quite useful to create new files (especially for experimenting with) or to update the last modification time of an existing file. The *touch* command does this:

```
touch new_file
ls -l
sleep 5
touch new_file
ls -l
```

The contents of a file (or files) can be viewed using *cat*:

```
echo "test" > test_file
cat test_file
```

cat dumps the whole file to screen which is unhelpful for large files. The *more* command can be used to print the file to the terminal one screen-full at a time. Space moves forwards in the file and search is the same as with man pages. *more* is quite limited: you can only move forwards. *less* (which is actually more than more in this case) allows files to be scrolled through and searched in both directions.

Because bash separates commands on spaces, it is inconvenient to use spaces in the names of files and directories despite the fact that this is common in other operating systems.

Input, output and redirection

Input, output and error information are treated separately in a unix system.

standard input
Standard input is the input from the user. Normally refers to the keyboard.

standard output
Standard output is the output from a program and is printed (by default) to the screen (or more specifically, to the terminal from which the program is run).

standard error
Standard output contains any error messages from a program. This is also printed by default to then screen and so output and error messages appear to be mixed. The key difference between standard error and standard output is that standard output can be buffered and standard error is not. This means that error messages can appear earlier than output if the output buffer is not full. (We shall discuss this distinction a little more in the C++ sessions.)

All three of these can be redirected:

>
Send standard output elsewhere.

2>
Send standard error elsewhere. For example:

```
echo hello world > hi
cat hi hi2 2> err
```

```
more err
cat hi hi2 > out 2> err
more out err
```

&>
Sends standard output and standard error to the same place.

<
Takes information from somewhere else (normally a text file) and sends it to standard input as if you had typed it in yourself. For example:

```
tr '[a-z]' '[A-Z]'  < hi
```

Note that using > to redirect output or error to a file will cause that file to be overwritten. Use >>, 2>> and &>> to append to files if they exist.

Sometimes you don't want to see any output and/or error messages. There's a "black hole" which things can be redirected to in this case called /dev/null:

```
cat hi hi2 2> /dev/null
```

Text editors

Input to and output from command line programs is commonly in plain (ASCII) text. For this reason we must use an editor which does not add formatting information or saves files a proprietary format. This also allows unix tools to interact easily. A text editor (rather than word processor) is used to edit files. Most text editors also have options such as syntax highlighting and automatic indentation which are useful when writing programs.

Some text editors are:

vi* and *vim
A standard unix editor (also available for windows). *vim* is an improved editor based upon *vi*: many systems now alias *vi* to *vim*.

emacs
emacs requires much more resources than vim but is far more than just an editor. Is almost an operating in its own right. See emacs and xemacs under Linux and AquaMacs under OSX.

gedit* and *kate
GUI text editors included with the GNOME and KDE environments respectively. *gedit* is also available for OSX.

TextEdit
OSX. Launch from spotlight.

notepad
Windows only. Very basic.

XCode
OSX. An integrated development environment rather than just a text editor. Launch from spotlight.

The choice of editor is very personal and it's worth spending some time looking at the options and finding one which suits you as they can make a huge difference to your productivity. However some familiarity with vi/vim is recommended as it is available on almost all unix-based systems.

In this course I will use *vim*, but you are welcome to use an editor of your choice. The default behaviour of *vim* can be controlled by options given in the ~/.vimrc file.

Tasks

1. Running

   ```
   ls ~/
   ```

   doesn't show any files beginning with a dot. Why not? How can you see them?
2. Create a directory and some file(s) in the directory. Try deleting the directory using *rmdir*. What happens? How can you delete it? (Hint: look at the *rm* man page.)
3. Run the command

   ```
   mkdir ~/dir1/dir2
   ```

   How can the error be fixed?
4. Familiarise yourself with the *vim* text editor (or an editor of your choice). The vim tutor has its own command:

   ```
   vimtutor
   ```

   Ensure that you go through a tutorial before the next session. Many other text editors also have a tutorial.
5. The operators <, > and >> were described above. There's also (unsurprisingly) a << operator. Find out what it does.