A beginner's guide to Linux

Ian Rutt

1 Introduction

The Linux operating system is probably the best environment for serious scientific computing. The way it functions will seem unfamiliar if you have not encountered it before, but it is not hard to learn, and can be quickly harnessed in powerful ways.

Although Linux sports a graphical (Windows-like) user interface, we will be focusing on the *Command-Line Interface*, where the computer is controlled by issuing commands at a text prompt. We will cover basic commands for handling files and directories, as well as commands that are useful for data-processing. This document is a tutorial on using Linux, and a basic reference for the techniques we'll be covering; the practical sessions will provide an opportunity to practice them.

UNIX and/or Linux?

You'll often hear people use the terms *UNIX* and *Linux* interchangeably. UNIX is a type of proprietary/commercial operating system (OS) which has enjoyed widespread use since the mid-1980s. Linux is a UNIX-like OS which is built of free software components: the Linux kernel originally developed by Linus Torvalds in 1991, and the software of the GNU project. There are differences between UNIX and Linux, but from the user's point of view, they are slight.

---

2 The File system

Before we start looking at issuing commands, it's worth getting an overview of the way the Linux system is organised. As with any computer system, *files* are an important part of it. Just as in Windows, a file can contain text, data, a program or whatever. These can be organised by collecting them into *directories* (called *folders* in Windows), which can exist inside one another, in a tree-like structure. Figure 1 shows a sample file system: the *root* directory, denoted '/' contains four directories, `dir1` to `dir4`, one of which contains other directories. The location of a file or directory is written as the sequence of directories that contain it, separated by forward-slashes, so that the location of directory `bob` in Figure 1 would be written:

`/dir2/jim/bob`

This expression is known as the *absolute path* of `bob`. By contrast, a path expressed without the leading slash is a *relative path*, i.e. it is taken in relation to the current directory.
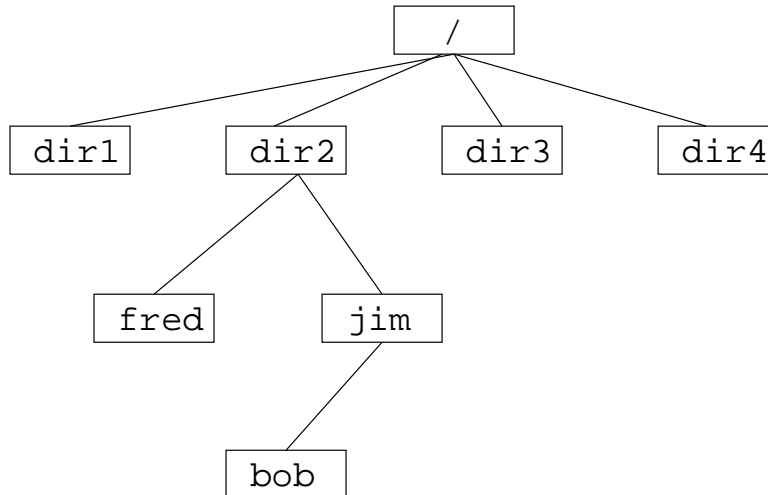
Figure 1: Linux file system

Three other symbols can be used in paths. The current directory is denoted by a single dot (.), the next directory up by two dots (..), and the user's home directory by a tilde (~). For example, if your current directory is `fred` in Figure 1, you can refer to `/dir2/jim/bob` as `../jim/bob`.

3   The Command-Line Interface (*a.k.a.* the *Shell*)

At the heart of the Linux system is the *kernel*, which runs constantly in the background, managing programs which are running, hardware, input and output, and all those things that most people don't want to have to worry about when using a computer. To allow the user to interact with the system, a command-line interface is provided, which is called the *shell* because it acts like a layer between the user and the kernel.

There are various shells available for Linux, but the most commonly-used one is called *bash*[1]. The shell is actually a sophisticated programming language, and can be effectively used to automate processes and control other programs.

3.1   Using *bash*

When you open a Linux terminal or console, you'll typically be presented with a prompt that looks something like this (the exact form depends on the system configuration — like almost everything else in Linux, the prompt can be customized):

```
[user@machine ~]$
```

This is where your commands appear when you type them. Try it now: the Linux command `ls` *lists* the contents of the current directory (this output is just an example - yours will look different!).

---

[1]The name is a pun, the details of which are not important...

```
[user@machine ~]$ ls
AdminDocs  bin  bookmarks.html  Desktop
```

Additional options can be added to a command, usually by using a minus sign; the `-l` option produces a long listing, with details about files sizes, etc:

```
[user@machine ~]$ ls -l
total 504
drwx------  2 irutt staff   4096 Aug  9 10:40 AdminDocs
drwx------  3 irutt staff   4096 Jun 18 12:39 bin
-rw-------  1 irutt staff 489658 Mar  2  2007 bookmarks.html
drwx------  2 irutt staff   4096 Sep 13 16:20 Desktop
```

Various extra pieces of information are shown, in addition to the filenames, including the owner of the file, its size, and who is allowed to access it.

Commands may also take *arguments*, such as the names of files that to be operated on. Arguments usually go at the end of the command, after the options. The `ls` command can take the name of a file or directory as its argument:

```
[user@machine ~]$ ls -l bin
```

This generates a long listing of the `bin` directory.

3.2 Navigating the file system

You can find out your current path using the command `pwd`:

```
[user@machine ~]$ pwd
/geog/home/user
```

You can change your location in the file system using the `cd` command (= *change directory*):

```
[user@machine ~]$ cd my_directory
[user@machine my_directory]$ pwd
/geog/home/user/my_directory
```

Now, the working directory is `~/my_directory`, and the prompt has changed to reflect this.

The final set of basic commands needed for navigating the file system are for copying, moving and deleting files, and for creating and destroying directories. The basic forms of these commands are:

cp *file1 file2*	Copies *file1* to *file2*. If *file2* already exists, it is overwritten.
mv *file1 file2*	Moves *file1* to *file2*. If *file2* already exists, it is overwritten.
rm *file*	Removes *file*.
mkdir *dir*	Creates directory *dir*.
rmdir *dir*	Removes directory *dir*, which must be empty.

3.3 Wildcards and operations on multiple files

When using some of the file commands given above, it is useful to specify multiple files easily. For instance, we might want to move all files ending `.txt` to their own directory:

```
[user@machine ~]$ mkdir text_files
[user@machine ~]$ mv *.txt text_files
```

What we have here is a form of the `mv` command that takes multiple files and moves them to a target directory. The first argument is a *pattern* that *matches* multiple filenames; the '*' is a *wildcard* that matches any text, so that `*.txt` matches filenames that consist of any text, followed by `.txt` — i.e. filenames that end in `.txt`.

Other wildcards are available — this is the full list:

Wildcard	Matches
`*`	zero or more characters
`?`	exactly one character
`[abcde]`	exactly one character listed
`[a-e]`	exactly one character in the given range
`[!abcde]`	any character that is not listed
`[!a-e]`	any character that is not in the given range
`{debian,linux}`	exactly one entire word in the options given

As you can see, it is possible to construct very sophisticated expressions for multiple filenames.

Take care with wildcards!

Obviously, careless use of wildcards can result in the unintended loss of important data. For instance, an accidental additional space can transform a command to delete files ending in `.txt` (`rm *.txt`) into one that deletes *all* files in the current directory (`rm * .txt`). One way of guarding against this is to use the `-i` option with `rm`, `cp` and `mv`: it makes these commands prompt for confirmation for each file that is to be deleted or overwritten. On some systems, these commands are configured to behave like this by default.

3.4 Finding out about commands

Most Linux commands have a large number of options that can be used to change their behaviour. For instance, the `ls` command has dozens of options that control the content and format of the output. But how do you find out what the options are? Obviously, a good reference book is very useful, but it can be as quick to use Linux's built-in manual pages. These can be accessed using the `man` command:

```
[user@machine ~]$ man ls
```

This will open a reference page for the command in your terminal window. Press *space* to go down the page, *b* to go back up, *h* for help, and *q* to quit and return to the command-line.

You'll notice from the manual pages for `ls` that some options have *long forms*, beginning with two dashes (*e.g.* `--recursive` is equivalent to `-R`). Also, it is possible to combine short

options together in a single option. For instance,

```
[user@machine ~]$ ls -laF ~
```

displays the contents of the user's home directory, in long format, including files and directories whose names begin with a dot, and adds additional characters at the end of filenames to distinguish directories ('/') and executables ('*'). These options are all described on the `ls` manual pages.

3.5 Pipes, redirects and command quoting

One of the most useful aspects of the Linux command-line is the ability to take the output of a command and use it as the input to another command. There are three ways of doing this: pipes, redirects and command quoting.

3.5.1 Pipes

A *pipe* is a way of allowing the output of one command to be used directly as the input to another command. Some commands take input from the keyboard; using a pipe, we can make them take input from another command instead.

Here's an example. The `grep` command is a very useful tool for searching through text to find parts that match a desired pattern; the syntax is `grep pattern`. Try it at the command line now, with a simple search pattern:

```
[user@machine ~]$ grep at
```

Here, '*at*' is the search pattern. Having pressed enter, you're left (rather perplexingly) with a flashing cursor, and not much else. Type some text — it doesn't matter what — and press enter at the end of each line (press *ctrl*-C to return to the prompt when you get bored). What you should notice is that when the line you typed contained the letters 'at', the computer repeated them back to you.

This may not seem much use, but when the input is from another command (perhaps one that produces a lot of output), `grep` is very powerful. Here's how it's done:

```
[user@machine ~]$ ls -a ~ | grep "^\."
```

The first part is a command (`ls -a ~`) lists the names of all files and directories in your home directory. The vertical line ('|') is the pipe, which feeds that list into the command that follows (`grep "^\."`). The search pattern given here, contained in quotation marks, matches lines that begin with a dot (*why* it matches that isn't important to us at the moment). As a consequence, a list of just those files and directories whose names begin with a dot are output to the screen. And if the list is too long, you can always pipe the output to `less`, which allows you to move through it a page at a time (*q* to quit):

```
[user@machine ~]$ ls -a ~ | grep "^\." | less
```

One thing that's important to understand about pipes is that feeding the output of one command into the input of another is *not* the same as using the output from the command as the *arguments* to another. What this means is that this command

5

```
[user@machine ~]$ ls | rm
```

will not delete all files listed by `ls` (obviously, you would use `rm *` to accomplish this task in reality). The mechanism needed for doing this is explained below.

3.5.2 Redirects

Often, it is desirable to write the output of a command to a file, or to supply the input of a command from a file. This can be achieved using a *redirect*. The simplest redirect uses the '>' operator:

```
[user@machine ~]$ ls > my_files
```

The result is a file that contains the output from `ls`. Use `less` to look at it:

```
[user@machine ~]$ less my_files
```

If you want to write the output to the end of a file that already exists (to *append* to it), this is done with the '>>' operator:

```
[user@machine ~]$ ls >> my_files
```

Now, if you look at the contents of `my_files`, you'll see the output from `ls` twice over.

Now, in reality, the output from a command comes in two *streams*, known as *standard output* and *standard error*. Up until this point, we've been concentrating on the standard output — the output of a command that has completed successfully. However, commands can also produce error messages, and these are handled separately. Try this:

```
[user@machine ~]$ rm > test
```

What happened? Does your file `test` contain anything? What happened here was that an error message was produced, and was sent via standard error to the screen. The redirect operator '>' only redirects the standard output, leaving error messages to be passed to the screen. Sometimes this is what you want, but it might be advantageous to send error messages to the same file, or a different one. To do this, *file descriptor notation* must be used. You can do a lot with this, but I'll just mention two possibilities. To redirect standard error to a different file from standard output, use '2>':

```
[user@machine ~]$ rm > test 2> errors
```

And to redirect the standard error to the same place as the standard output, use '2>&1':

```
[user@machine ~]$ rm > test 2>&1
```

The other type of redirect that is of interest to us allows the supply of run-time input from a file (this is called *standard input*), and uses the '<' operator:

```
[user@machine ~]$ grep too < test
```

Here, the contents of file `test` are used as the input for `grep`, and lines that match 'too' are output[2]. This technique is particularly useful if you have some model code that prompts the

---

[2]This is a somewhat contrived example, as `grep` will also take a filename as an argument: `grep too test` accomplishes the same thing

user for input: instead of having to type in the parameters each time the model is run, you can put them in a text file and use redirection.

3.5.3 Command quoting

In the section on pipes, it was noted that they cannot be used to allow one command to supply the arguments to another command. The way to make this happen is to use *command quoting*. Bash understands three types of quotation marks: ' (single-quotes), " (double-quotes), and ` (backward-quotes). Each has a different use, and it is the last of the three that is used for command quoting[3]. To replace a command with its output, all you need to do is enclose it in backwards quotes:

```
[user@machine ~]$ rm `ls *.txt`
```

The result of this command is the deletion of all files ending in '.txt' (OK, this is a trivial example, but it illustrates the principle — the command `ls *.txt` is replaced here by its output. You'll be surprised how useful this is!).

4 Environment variables

I'm sure you're familiar with the concept of a *variable* — an entity that holds a value, which can be changed as time goes by. An *environment variable* is simply a variable that exists in the Linux command-line, and can be read and written by programs that are run from that command-line. There are several reasons why you might need to set or examine the contents of an environment variable; some commands use them to determine where to look for particular files, for instance. Most environment variables are set automatically when you log in, but sometimes it is necessary to adjust them yourself.

Environment variables names are almost invariably composed entirely of uppercase letters. Accessing their contents is straightforward:

```
[user@machine ~]$ echo $PWD
/geog/home/user
```

The dollar ($) indicates that what follows is a variable name, rather than plain text. We use the `echo` command to 'echo' the contents of the variable to the screen. As you can see, the environment variable `$PWD` contains the current path, as returned by the command `pwd`.

To set an environment variable, we use the *bash* command `export`:

```
[user@machine ~]$ export MY_VARIABLE=a_value_of_some_kind
```

Note that we *do not* put a dollar before the variable name. If we did, the shell would try and substitute the name with the contents of the variable, which isn't what we want[4]. Of course,

---

[3]The backward quote is usually found to the left of the number one on a standard keyboard.

[4]You should also be aware that this is one thing that other UNIX/Linux shells do differently. The *C shell* (`csh`) uses the `setenv` command, with a space between the variable name and its contents rather than an equals sign.

you can't just set any old environment variable; in particular, strange and unpredictable things will happen if you set a variable like $PWD which is set by the system.

Probably the most useful environment variable to know about is $PATH, which is a list of directories that the system searches when looking for commands. You can see what's in your PATH using echo, as above.

You'll notice that the directories are separated by colons. This means that if you need to add a new directory to the list, you can just tag it on the end:

```
[user@machine ~]$ export PATH=$PATH:/my_directory
```

The presence of the dollar before the second mention of PATH is because in this case we *do* want the shell to substitute the current contents of the variable. A word of caution: it's generally considered to be bad practice to add the current directory (.) to the PATH, for security reasons. If you need to execute a command located in your working directory, but not in your path, then prefix the filename with './'.

5 Processes

Linux is *multitasking* operating system; that is, it's capable of doing more than one thing at a time. To keep track of everything that's going on, the system organises the tasks it's doing into units of activity called *processes*. You can think of a process as a running program. What might come as a surprise is the number of processes that are running even when you're not doing anything. The ps command outputs a list of processes:

```
[user@machine ~]$ ps -A
```

This delivers a list of all running processes, each with its process identification number (PID) on the left-hand side, and with the name of the process on the right. Like other commands, ps can take a long list of different options to control the processes listed and the format of the output. Without any options, only the processes started from within your current terminal window are listed.

5.1 Running processes in the background

When you run a simple command like ls, it's fine to wait while it runs, and get the prompt back when it finishes. However, often you'll want to set a program running, and carry on using the terminal while it runs. This is particularly true of programs that spawn their own windows, such as file editors or viewers. Here's how to run the popular text editor emacs in the background:

```
[user@machine ~]$ emacs &
```

That's all there is to it: just add an ampersand (&) at the end of the line. (Close emacs's window to quit).

Sometimes you'll have started to run a program in the foreground, and then want to move it to the background. This is easy too: type *ctrl*-Z to stop the program temporarily, followed by the command bg (= background). The program will then continue to run in the background:

```
[user@machine ~]$ emacs
```

(press *ctrl*-Z)

```
[3]+  Stopped                  emacs
[user@machine ~]$ bg
[3]+ emacs &
```

You'll notice that after you press *ctrl*-Z, `emacs` becomes completely inert, until you restart it in the background.

5.2   Leaving processes running when you log out

Often, a numerical model or data-processing routine will take a long time to run — hours or perhaps days. It can be inconvenient (not to say risky) to just launch these commands from the prompt and leave them running. If you log out, or even just close the window where the command is running (even one that's running in the background), it will terminate. And of course, if you're logged into the machine from elsewhere, the connection could fail for some reason, and you'd have to start again.

To avoid these problems, it's good to start long-running processes using the `nohup` command (= *no hang-up*). Since you won't be able to look at all the output from the command on the screen, it's useful to use a redirect:

```
[user@machine ~]$ nohup my_model arg1 arg2 > my_output 2>&1 &
```

Here, `my_model` is the command you're running; `arg1` and `arg2` are the arguments taken by the command, and `my_output` is where the standard output and standard error are being redirected.

5.3   Managing processes

OK, so you've started your model running using `nohup`, but then you realize that you've made a mistake with the input parameters, which you need to rectify. But how do you stop the model? Pressing *ctrl*-C or even logging out won't work — what you need to do instead is to *kill* the relevant process.

The first step is to discover your model's process number. This is simple enough: we just use `ps`:

```
[user@machine ~]$ ps -u user
```

The `-u user` option selects processes belonging to the given user — obviously you would substitute your own username here. The command returns a long list of processes. We look for our model in the list of running processes:

```
9795 ?        00:00:17 my_model
```

And then we kill it (using the `kill` command: the use of `-9` ensures the process is well and truly dead):

```
[user@machine ~]$ kill -9 9795
```

Another useful tool for process management is `top`. This provides and interactive, continually updated view of the most processor-intensive running processes. You can use it to check whether your model is running properly, see whether other things are slowing your machine down, and discover how long a process has been running for (among many other things). The command doesn't need any arguments: it's simply `top`.

Here's some sample output from `top`:

```
top - 14:04:42 up 16 days,  3:27,  4 users,  load average: 0.11, 0.13, 0.09
Tasks: 132 total,   2 running, 130 sleeping,   0 stopped,   0 zombie
Cpu(s): 26.0% us,  0.3% sy,  0.0% ni, 71.9% id,  1.7% wa,  0.1% hi,  0.1% si
Mem:   3838400k total,  3703712k used,   134688k free,    57236k buffers
Swap:  2040212k total,      224k used,  2039988k free,  3049276k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
20383 irutt     25   0  210m  13m 1576 R  100  0.4   0:04.25 genie.exe
 9917 irutt     16   0  164m  20m 8724 S    2  0.5   0:05.60 gnome-terminal
 5641 root      16   0  184m 134m 8052 S    1  3.6  16:04.60 X
  562 root      15   0     0    0    0 S    0  0.0   0:09.94 kjournald
11488 irutt     16   0  295m  72m  21m S    0  1.9   0:29.86 firefox-bin
    1 root      16   0  4756  556  460 S    0  0.0   0:01.27 init
    2 root      RT   0     0    0    0 S    0  0.0   0:00.15 migration/0
    3 root      34  19     0    0    0 S    0  0.0   0:00.01 ksoftirqd/0
    4 root      RT   0     0    0    0 S    0  0.0   0:00.16 migration/1
    5 root      34  19     0    0    0 S    0  0.0   0:00.02 ksoftirqd/1
    6 root      RT   0     0    0    0 S    0  0.0   0:00.11 migration/2
```

From this, we can learn that I'm running something called `genie.exe`[5], which is using 100% of one CPU, and has so far consumed 4.25 seconds of CPU time. Type *q* to quit `top`.

6   Utility Command Reference

This section is intended to give you a manageable overview of a small number of the most useful commands available on the average Linux system. Neither the list nor the descriptions of the commands are at all comprehensive, but they should give you an idea where to start. For a complete description, you should turn to the `man` pages on your system, or to a reference book (see bibliography).

6.1   Looking at the contents of files

There are various ways to look at the contents of text files. The simplest are `cat`, `more`, `less` and `tail`:

---

[5]GENIE is an Earth System model

cat	`cat file`
	Outputs the contents of `file` to the screen.
more	`more file`
	Outputs the contents of `file` to the screen, a page at a time
less	`less file`
	A more sophisticated version of `more`. However, the advantage of `more` is that the text generally remains on the screen after `more` exits, whereas with `less` it does not.
tail	`tail file`
	Displays the last ten lines of `file` to the screen. Can be made to display more or less using the `-n` *number* option. There is also a corresponding command called `head` (no prizes for guessing...)

6.2   Comparing two text files

The key command here is the imaginatively-named `diff`, which takes the names of two files as arguments, and outputs any differences it finds. This is most useful when the two files can be expected to differ only slightly, or not at all.

6.3   Compressing and archiving files

Given the potential size of some data files, it can be useful to compress them, where possible. If you only have a single file you want to compress, you can use `gzip`:

```
[user@machine ~]$ gzip my_file
```

This will produce a compressed file called `my_file.gz`, and remove the original, uncompressed file. To restore the file to uncompressed state, simply use `gunzip`:

```
[user@machine ~]$ gunzip my_file.gz
```

With that, your original file is restored. Using the `-v` option with either command results in information being output to the screen about the compression/decompression process.

If you have more than one file that needs archiving/compressing, the `tar` command is your friend. The syntax is a bit obscure (`tar` is a venerable command that dates back to the days when data was routinely archived on tape), but entirely manageable when you get used to it. The first argument after the command is a series of letters that specify the operation to be performed:

List of `tar` options

x	extract files
c	create an archive
t	list files in an archive
v	be verbose (i.e. print lots of output)
z	use file compression
f	the following argument is the name of the archive file

Obviously, it doesn't make sense to specify more than one of **x**, **c** and **t** at a time. So, to archive the contents of `my_dir` to a compressed archive, this is the command you would typically use:

```
[user@machine ~]$ tar czvf my_dir.tar.gz my_dir
```

The suffix `.tar.gz` is conventional for the resulting archive files. To uncompress the directory, the procedure is very similar:

```
[user@machine ~]$ tar xzvf my_dir.tar.gz
```

Care must be taken to make sure this operation doesn't overwrite any pre-existing files located target directory

6.4   Editing files

As mentioned previously, `emacs` is a popular and very versatile editor, available on almost all Linux systems. One of its great advantages is that it has many built-in highlighting and indentation modes for use with different programming languages. For example, Fortran code is automatically coloured to reflect its syntax, and the `TAB` key can be used to auto-complete `do`-loops and `if`-blocks.

Emacs is a sophisticated program, and has a built-in version of the LISP programming language, in addition to many useful text-editing tools. If you want to learn how to use it effectively, consulting one of the many books or websites devoted to the subject is essential.

For the more adventurous, and especially when in a situation where emacs is unavailable, the standard Linux editor is called `vi`. Explaining how to use `vi` is definitely beyond the scope of this document!

6.5   Printing

Printing is one of those things that varies somewhat between Linux systems; over the years, there have been many ways of handling printing in UNIX and Linux, and so there is perhaps more variation between systems than in other matters. So, here we will cover one of the most common printing frameworks, the *Common Unix Print System* (CUPS). Most of these commands should work on most Linux systems.

First, you need to find out what printers are attached to your system, and what they're called. The main command for discovering information about printers is `lpstat`:

```
[user@machine ~]$ lpstat -p -d
```

This combination of options returns a list of currently available printers and also gives the name of the default printer. On my system, I get this output:

```
printer GlaciologyMono is idle.   enabled since Jan 01 00:00
system default destination: GlaciologyMono
```

So, I have only one printer available (called `GlaciologyMono`), which is the default.

The basic file format for printing in the UNIX/Linux is *Postscript* (files generally end in `.ps`). Postscript was devised by Adobe, and is in some respects similar to their popular *Portable Document Format* (PDF). Postscript files may be printed directly to a printer using the `lpr` command:

```
[user@machine ~]$ lpr -P GlaciologyMono my_file.ps
```

The `-P` option is used to specify the destination printer: if it is omitted, the default printer is used.

Without any options, `lpstat` can be used to discover information about what print jobs are being processed:

```
[user@machine ~]$ lpstat
GlaciologyMono-270    irutt    20480    Thu 04 Oct 2007 12:00:26 BST
```

Only the user's own print jobs are displayed. To cancel a job, the `cancel` command is used, along with the job ID:

```
[user@machine ~]$ cancel GlaciologyMono-270
```

One very useful command that can be used to print plain text files is called `a2ps` (the name is shorthand for *ASCII-to-Postscript*). This command will format text in a sensible way, and convert to Postscript format. The number of columns, page orientation, font size, etc, can be specified using options. Usually, a2ps is configured to send its output direct to the default printer, but this can also be changed: see the relevant manual pages for details.

6.6   Accessing remote systems

It is often useful to be able to access other computer systems from the Linux command-line. Two basic methods exist: `ssh` allows you to log onto another computer and give it commands through the shell as though you were physically sitting in front of it, while `sftp` allows you to transfer files to and from another computer. In both cases, the initial 's' stands for 'secure' — the connections are encrypted.

Using `ssh` (*Secure Shell*) to access another computer is simple:

```
[user@machine ~]$ ssh -X username@remotemachine.somewhere.ac.uk
```

Here, `remotemachine.somewhere.ac.uk` is the computer you want to log into, and `username` is your username on that machine. The `-X` option allows software running on the remote machine to display windows on your display. You'll be prompted to enter your password, of course.

Transferring files is done using `sftp` (*Secure File Transfer Protocol*), and the syntax is very similar:

```
[user@machine ~]$ sftp username@remotemachine.somewhere.ac.uk
```

After entering your password, you're presented with a prompt that accepts a number of simple commands:

List of selected `sftp` commands	
put	copy a file from local to remote machine
get	copy a file from remote to local machine
cd	change current directory on remote machine
lcd	change current directory on local machine
ls	list files in current directory on remote machine
lls	list files in current directory on local machine
rm	delete file on remote machine
rename	rename a file on the remote machine
help	display help text
exit	quit `sftp`

One thing to note is that the `put` and `get` commands only act on *one file at a time*, so if you want to transfer large numbers of files, it is more efficient to use `tar` to make an archive of the files and transfer that.

Another useful command that can be used for file transfer is `scp` (*Secure Copy*). See its manual pages for more information.

7   Customizing Linux

One thing almost all regular users of Linux do is to customize the way Linux, and especially *bash*, behaves when they log in. There are several hidden files located in your home directory (they're hidden because they begin with a dot, so don't show up unless you do `ls -a`). The most useful is probably `.bashrc`, which consists of a list of commands to be executed when *bash* is started.

Among the tasks you can perform by adding lines to `.bashrc` are these:

- Customize the prompt — for instance, to display the whole path, of just part of it, or not at all.

- Set extra search directories in the `PATH`.

- Set `rm`, `cp` and `mv` to prompt for deletions by default.

The last of these is accomplished by adding the following three lines to `.bashrc`:

```
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
```

Selected bibliography[6]

Barrett DJ (2004) *Linux Pocket Guide*, O'Reilly.

Cameron D, Elliot J and Loy M (2004) *Learning GNU Emacs*, O'Reilly.

Siever E, Weber A, Figgins S, Love R and Robbins A (2005) *Linux in a Nutshell*, 5th Ed., O'Reilly.

---

[6]Despite appearances, I wasn't in the pay of O'Reilly when I compiled this list.