When it comes to working with large numbers of files, large data sets, doing many calculations for simulations, or to research computing generally, computational scientists turn to UNIX as the host operating system of choice. The following material is meant as a very brief introduction to using UNIX.

1 Unix Commands and Concepts

One interface to a UNIX operating system is a terminal window with a command prompt. Commands may be typed following the prompt and the commands are interpreted after the return key is pressed. There are many commands and programs that are available and these vary among different types of UNIX and the options that are installed.

The fact that this type of interface has been around since the early days of computing (see Fig. 1) should not be taken to suggest that it is antiquated or that it lacks power. Instead, many tasks can be accomplished more rapidly and efficiently using this textual interface, rather than a graphical one. In the sections that follow, fundamental commands and concepts are introduced that should enable a user to accomplish many of the common interactions with the operating system.

Figure 1: The origin of the word “terminal” is that users once sat at one of several keyboard and monitor combinations (i.e., terminals) that were connected to a single computer. One common model was the Digital/DEC VT100, introduced in 1978 and pictured above. Terminal windows on contemporary computers still emulate the functionality of these devices.

1.1 Logins to a multi-user system

Modern operating systems are multi-user systems, meaning that they allow multiple users to login with username and password credentials. UNIX-based systems have the additional feature that multiple individuals may be and are expected to be logged in at once. The means by which the opportunity to login arises depends on whether you are sitting at a computer display that is directly connected to the UNIX computer or you are connecting to the computer remotely. We will address the first situation this week and discuss remote connections in a later class.

Assuming that you are sitting at a UNIX computer, you very likely have a graphical user interface that presents you with a dialog box for your username and password. If these are accepted as a valid pair, you are presented with a graphical interface with standard features (a Desktop, a menu bar, etc.). Within the graphical environment, an application that provides a command-prompt will be available. In the case of Mac OS, it is found in the directory `/Applications/Utilities` and is called `Terminal`. Every UNIX will have terminal applications, it is just a matter of finding them (a common one is ‘xterm’).

Upon logging in and opening a terminal window, the default directory in which one begins is the ‘Home’ directory. Your Home directory is a directory that is associated with your username and to which you have a full set of permissions for altering its contents. This is in contrast to higher-level directories, many of which may only be modified by the computer’s administrator (which may be you).

Once you have opened a terminal window, you will be presented with a UNIX command prompt where you may enter commands. The prompt is customizable to contain different information and may as simple as:

```
$
```

or a bit more complex like:

```
[labuser@hybrid ~]$
```

where `labuser` is the username, `hybrid` is the name of the host computer, and ~ indicates that the user is currently working in the home directory (~ is shorthand for the home directory, more about this below).

One of the nice features of a graphical user interface is that you can easily open and manage multiple terminal windows simultaneously. This was more difficult back in the day (Fig. 1). These windows are independent processes and you can use them for unrelated and parallel tasks. Think productivity.

1.2 Managing files and directories

One of the principal metaphors of interacting with a computer involves the storage and organization of data into what we have come to call directories (or folders, given the visual representation) and files. Creating, moving, and deleting files and directories are fundamental tasks that may accomplished through a graphical user interface (GUI) or at the command line. I assume you know about GUIs. Knowing how to accomplish these tasks from the command line is a requirement for remote connections to a UNIX computer and can speed your work considerably relative to interacting with the GUI through your pointing device ("mouse").

Assuming that you have a terminal window open, you will a command prompt, which may or may not tell you what directory you are in (see above). Upon logging in, you begin in your home directory, but to find out where you are, you can alway use the `pwd` command:

```
$ pwd
/Users/labuser
```

In this case, you are in the directory `labuser`, which is a subdirectory of the directory `Users`, which itself is a directory at the root (/) of the filesystem. The full specification of a directory or file is called 'the path'. In this case the path is from the root directory to the individual folder named `labuser`.

1.2.1 Listing the contents of directories

To list the context of a directory, you can use the `ls` command.

```
$ ls

 Desktop/    Movies/    Sites/  etc/
 Documents/  Music/     admin/  incoming/
 Downloads/  Pictures/  bin/    research/
 Library/    Public/    comp/   teaching/
```

In this case there are sixteen items in the directory `labuser` (all are subdirectories in this case). There are many options to `ls`, a few of which are so useful that it is worth making them the default behavior of `ls`. For example, I use `ls -F` to be able to distinguish among different types of items in a directory. / and nothing and star ...

You can also point `ls` at directories other than the one you are presently in, so you can try `ls /`, `ls /Users` and `ls /bin` to examine the context of a few directories (try these with `ls -F` as well). The contents of any directory can be listed in this way. The main difference between the simple `ls` and `ls -F` is that the latter will display a slash (`/`) immediately after each item that is a directory and an asterisk (`*`) after each that is executable (and other symbols for other special items), but nothing for extra for simple files. This is very helpful, as it is routine to want to know whether an entry like `research` is a simple file or a directory.

There are two other important options for `ls` to know about (there are 37 options in total on the system on my desk): `ls -a` and `ls -l`. Try them out on a directory of your choosing (e.g., `ls -a ~/`). `ls -a` displays information about all files, including ones that begin with a period and are typically hidden from `ls` and the GUI representation of a directory. `ls -l` results in a long format listing of the contents of a directory, including information about the time an item was modified and the permissions for access to an item (more about permissions later). As with many other commands, you can combine options together to get `ls -alF` (I use this combination often enough that I have a shorthand for it, which will be described in the next handout).

1.2.2 Moving around: changing directories

Theoretically, you could do all of your work in one directory and you could access any directory without moving from your home directory. But it is not very convenient or efficient way to work. Instead we organize things into folders, and moving into a particular folder makes access to its contents much easier than specifying the full path to an item.

The command to change directories is `cd` followed by the directory that you want to change to, so for example try:

```
$ cd /usr/local
```

Note that the command does not return or print anything to your terminal window (unless there is an error). Instead, your point of reference has changed to the named directory and we say that you are now "in" in this directory. It's a bit strange, but that's the metaphor. Try the `ls` to verify that you are indeed now somewhere else.

Issuing the `cd` command without an argument results in the user moving to the user's home directory. It's a short-cut and is used often to move out of the current directory to the home directory.

Above, you could have moved to `/usr/local` in two steps instead of one. That is, you could have first changed into the `/usr` directory and from there into the `local` directory. Here's how: `cd /usr` and then `cd local`. Note that in that second step, you are able to change into the `local` directory without specifying its full path, but instead you provide its "relative" path (the path relative to your current directory).

The relative path concept leads to the idea that rather than always starting at the top, at the root of the directory structure, you can move directly and relative to your current location. You have just seen that you can change into a subdirectory of the current directory easily, by simply naming it in the argument to `cd`. The parent directory of the current directory is specified with two periods, so `cd ..` takes you one directory up in the hierarchy of directories, and `cd ../..` takes you two levels up (if you like the family metaphor, into the "grandparent" directory). Similarly, you can go up one and directly into a neighboring directory with something like this: `cd ../neighbor`, assuming that there is a directory named `neighbor`.

A final concept is that the current directory is specified by a single period (`.`). This might not seem important now, but in some cases you want to make certain that you are referring to a file in the current directory and in this you can write `./myfile`. If you list the full contents of a directory (`ls -aF`), you'll note that there are entries for the current and parent directory there (`.` and `..`).

You will be happy to learn that your modern command-line interface is likely to feature tab-completion, meaning that you can begin to type a command or filename and hit tab to complete the name. If you have typed enough for the completion to be unambiguous, it will be completed; otherwise you can hit tab in quick succession and be presented with the possible matches. This makes moving around the directories pretty quick, since little typing is required.

1.2.3 Making changes

So far we have been looking around with an existing set of files and directories. Now we need to learn how to make changes.

To create a new directory, simply give the command `mkdir newdir`, where `newdir` is an argument to the command and is name of your choice. In naming things on a UNIX system, there are many special characters to avoid. It is simplest to stick to alphanumeric characters, as well as `_`, `.`, and `-`. You definitely want to avoid other punctuation marks and slashes, and even spaces are best avoided.

Copying files is easy and is accomplished with `cp` command. For example, to make a new copy of a file, with a new name, simply type `cp oldfile newfile.txt` (here you have give the file a new basename and have given the file the txt extension). As above, you can use wildcards, and you can use `cp` to make copies elsewhere. For example, `cp *aug08.txt ../data/` would make copies of all the files that end in `aug08.txt` in a neighboring directory, without renaming them. Note that by default `cp` does not work on directories; to copy them, check out the option `-R` in the manual page for `cp` (more about manual pages below).

A related command is `mv`, which is used to move files, without making a copy, and with or without renaming. So `mv oldfile newfile.txt` would result in all the data in `oldfile` now being in `newfile.txt`, and there being nothing referred to by the name `oldfile`. Similarly, `mv *aug08.txt ../data/` would result in these files being moved, not copied.

To remove files, there is the `rm` command (as in `rm file.txt` or `rm *aug08.txt`. Be careful with this one. Unlike the GUI interface for deleting files, you will not be asked for confirmation and the files do not go to "Trash" folder. They are removed. That's it. You can set things so that you will be asked for confirmation before deleting, but still, once the files are removed, they are gone.

Removing directories is simple; as long as they are empty you use `rmdir dirname`. If there are files in the directory and you are sure you want them all removed, you can use the command `rm -rf dirname` to remove the directory and all of its contents (`-r` is for 'recursive', meaning that `rm` will descend into subdirectories and remove everything, and `-f` is for 'force', as in do not bother to ask for confirmation and make it possible to uproot and remove directories).

As an illustration that you should NOT try, consider that with the appropriate permissions, you could erase your entire hard drive with `rm -rf /`. Obviously, do not try this one if you do happen to have administrator privileges to the computer you are using. Your computer will not be very useful afterwards until you reinstall the operating system and restore your files from your backups.

1.2.4 Permissions

If you have logged into a UNIX computer, you have a user account that gives you a certain level of access to read, write and execute files on the computer. The access control and permissions are controlled on a per-file, or per-directory basis. For example, you have permissions to write new files to your home directory, but probably not to do the same to the home directories of other users. Instead, you may have read access to other users' directories.

This may sound arcane, but file permissions are part of what makes UNIX a multi-user system. Permissions keep users from overwriting one another's files, keeps user from mucking up administrative settings, and allow access to certain files as programs. You will need to modify permissions if you want to give access to files to others on the same system or if you are writing programs yourself and want to let the operating system know that a file is a program.

There are three main categories of permissions that are associated with a file: *read*, *write* and *execute*. These are simple and intuitive. If you have read access, you can look at a file's contents. If you have write access, you can modify a file, including deleting it. If you have execute permissions, you can run the file as a program (this way the use of certain programs can be restricted to certain users or the administrator).

These permissions can be set separately for the owner of the file (*user*), the *group* to which the *user* belongs, the *others* (other users) who have accounts on the system. So there is quite a bit of control of access possible. The owner and group associated with a file can be modified too (the commands are `chgrp` and `chown`).

The permissions associated with files (and directories; it turns out that directories are really not a different thing) are evident when you list the contents of a directory in long form with `ls -lF`. An example, partial listing is below:

```
drwxr-xr-x  80  labuser  staff  2720  Jun  26  15:35  bin/
drwxr-xr-x  42  labuser  staff  1428  May  30  16:03  comp/
-rw-r--r--  1   labuser  staff   532  Aug  19  17:19  array.c
```

The first two entries begin with `d`, which indicates that they are directories, whereas the third begins with `-`, which indicates that it is a regular file. All three items are owned by `labuser`, who is a member of the group staff. This information is followed by the modification date and time, and the names for the items. At the far left, just after the `d` or `-` are the permissions, which can be *read* (`r`), *write* (`w`) or *execute* (`x`), and are listed in order for the *user* (owner), the *group* and *others*. In the example above, `array.c` can be read by everyone, but only written (modified) by the file's owner. Access to directories is controlled by modifying the *execute* (`x`) attribute.

Your system will have a default set of permissions for new files and directories you create, but these can be modified with `chmod`. For example, to remove access to the `comp` directory above and restrict it to only the directories owner, the command would be: `chmod og-xr comp` (this remove execute and reading privileges for others and members of the group). To make file executable for the user, use: `chmod u+x myprogram`

1.3 Inspecting files

Beyond listing files and directories, at the command line it is common to want to know something more about your files and to have a look inside files. Many of the files that you will use in scientific computing are text files and there many tools to work with these from the command-line.

As you may already know, the name of a file is not necessarily a good indication of its contents and type. For example, a file may or may not have a suffix to indicate the type of file, or it may have a wrong or unknown suffix. A simple tool to find out whether a file is of a known type is the command `file` with the file of interest as the argument. For example, `file array.c` correctly returns the following:

```
array.c:  ASCII c program text
```

It recognizes that the file contains ASCII text and that it contain source code in the C programming language. It determined this on the basis of the file contents, not on the basis of its name. Try this out on some of your files. `file` does not know about all filetypes, but it is able to recognize many common types.

Opening files in the appropriate application, from the command-line is simple if your UNIX system has the command `open`: simply provide the file you want to open as an argument to `open`, as is `open array.c`. This will work best if you are working with a graphical interface to the UNIX computer (more about this later).

Two simple programs to look inside text files are `more` and `less`. For example, `less array.c` opens the file in the terminal window and allows you to page through the file. The space-bar advances to the next window of text, `p` takes you to the top of the document and `q` quits the command and returns you to the command-line. There are many other functions, including the ability to search through the file. Find out whether you have `more` or `less` on your system (or both) and give them a try.

Two other commands that can be useful are `head` and `tail`, which print out a few lines from the head or the tail of a text file (the default is 10 lines on my system). This is an easy way to have a look at the beginning or end of a huge file without opening the whole file (e.g., to see the column headings at the beginning of a text file of data). These two simple programs have options too (see the manual page).

1.4 Launching programs and job control

Beyond working with files and directories, and simply storing data on a UNIX system, you will want to work with programs. For many tasks you will want to start programs from the command-line, although if you have a graphical interface to a system you will also be able to launch many applications from the GUI. In other words, you may launch Microsoft Excel in the GUI by double-clicking on an icon, but many other programs will be opened by invoking their name one the command line.

For example, if you want a simple editor for text, one option is the program called `pico` (some systems also have `nano`, which is an enhanced `pico` clone). To launch this program from the command-line, simply type its name. If your system has this program and it is in one of the directories that is searched for default for programs (the `PATH`), it will be launched. A menu of keystroke commands that are available in the program will appear at the bottom of the screen. The caret symbol (`^`) refers to the control key.

An enormous number of other programs exist and we will use some of them in this course. Programs that are installed by the computer's administrator typically reside in directories named `bin` and there are several default directories with this name. UNIX systems will have `/bin`, `/usr/bin`, and usually also `/usr/local/bin` and other directories where programs, or binaries, are stored. The set of directories to be searched for a program name when it is typed at the command-line is defined in a environment variable called `PATH`. You can see the contents of this variable by typing `echo $PATH`. Later we will see how you can modify this and other environment variables.

1.4.1 Job control

Scientific computing often involves running programs that run for a long period. One of the UNIX features that facilitates running jobs that take a long time to complete is that they can be put into the background, where they can continue to run without interaction from the user. In fact, if done properly, programs can be launched in the background and they can continue to run even after the user logs out.

As a simple example, on a Mac OS computer, there is a program called `jot` that prints out random numbers in the range of 1–100 if invoked with the `-r` option. So `jot -r 1000` prints out 1000 random numbers in the range of 1–100. This does not take very long. If you increase the number of replicates you can give the computer something to do. Suppose that you didn't want all of the output going to the screen, but instead wanted to store the results in a file. You would do this with redirection of the output (more about this below). Instead of having the output going to the screen (the standard output), you can redirect it to a file using >, as follows: `jot -r 10000 > randnumbers.txt`. If the numbers of replicates were large and you did not want to wait around for the command to complete, you could put the commands (also called jobs) in the background by putting an ampersand (`&`) at the end of your command: `jot -r 10000 > randnumbers.txt &`.

To see what jobs you are currently running in a terminal window, type `jobs`. You can have a large number of jobs running simultaneously from one window and still control them. It is not that common anymore to shuffle jobs between the foreground and background, but there are commands to do so (`fg` and `bg`)

Now suppose that instead of generating 10,000 random numbers you were running a simulation that took a few days to run. The benefit of being able to put jobs in the background should be evident, particularly if you would like to log out of the computer in the interim and return later to examine the results. The details of how to keep jobs running depends on the UNIX system that you are using. It may work by default without any extra steps required, but there is a cost to this. If users launch long-running programs unintentionally or forget about them, logging out will not terminate them and the system can get clogged up with unattended programs. So many systems require you to specify that you want things to run even after you log out. Two options to investigate for your system are the commands `nohup` and `screen`.

An important command to know is how to terminate a job that you no longer want to keep running. In UNIX, jobs that are in the foreground can be terminated by holding down the control key and hitting 'c' (I will typically write this as `CTRL-c`).

What about jobs that you launched yesterday and that are still running today, but that no longer are useful (e.g., it is clear that an optimality search is not converging on an answer fast enough to be useful). This is an opportunity to pull back the curtain and see what all the jobs (processes in this context) are that are running on a system. On Mac OS and many other system, you can get a detailed listing of all processes that are running with `ps aux` (`ps` stands for process status). You might need to increase the size of your terminal window to be able to read all or most of the output. Look for your username and you will see all of the processes that are associated with you. The second column in the output is the unique `PID` (process id) for each process that is running. You are in charge of your processes, so if you find one that you no longer want, and that is not associated with a terminal window (in which case you could use `CTRL-c`, see above), you can terminate it with `kill` *PID* (where *PID* is the numerical id for your process). Unlike the everyday usage of the word, there are different levels of `kill`, so that to really yank a process out by its roots you would type `kill -9` *PID*.

1.5 Manual pages

At this point you might be wondering where one finds information about all of these commands that make UNIX work. One source of information is right at the command-line, in a collection of manual pages that are included with the system. These can be very terse and are not necessarily friendly to the beginner, but they do contain a listing of all of the options associated with a command or program and will at least explain what a command does at some level. So, for example, to learn more about the `jot` program we used above, you can type `man jot` and the manual page for `jot` will be opened. You can page through the manual with the same commands as are used for `less` or `more` (see above and `man less` for how these work).

The manual pages are a good place to start. Alternatively, Google will also lead you to information about commands. Although often Google finds lots of copies of the UNIX manual pages that are already at your disposal.

1.6 Sending output somewhere: the screen, pipes and redirection

1.6.1 Redirection

In Section 1.4.1, we saw how output could be redirected to a file instead of to the screen (which is often the *standard output*). This is a fairly common practice and another instance of output redirection that arises commonly is for concatenating files (placing the contents of text files end-to-end). The command for concatenating and printing files is `cat` (see `man cat`). Simply typing `cat somefile.txt` will print the contents of somefile.txt to the screen. If the file is big, this is not very helpful, because the contents will fly by on the screen. More interesting perhaps is the ability to take a collection of text files and make them into one. For example, this application arises if you have many text files that contain data from instruments and you want to combine them all before reading them into an analysis program. This is often preferable to opening each file separately from within the program, particularly if the number of files is large. For this application the lines of data should have information (date and time, or other identifier) that makes it unnecessary to store the data in separate files. A simple means by which you could combine all the relevant files within a directory would be:

```
cat *data.txt > loggerdata_alldays.txt
```

Here the `*` is a wildcard character that matches anything and in this case grabs all files that end in `data.txt` but are preceded by anything else. The example above works for any number of files and is a nice illustration of the power of the command-line. Remember that you could omit the redirection of the output and have all the data directed to your terminal window. Find some text files of your own to experiment with and try `cat` with and without the redirection of output.

In some cases you do not want to overwrite the output file, but you want to append new data to the file. For appending output you use >>, whereas > means to write (or overwrite) a file with the specified name.

```
cat newdata.txt >> loggerdata_alldays.txt
```

1.6.2 Pipes

In Section 1.4.1 you saw that a lot of information is printed to the screen when you used `ps aux` to list all of the processes on the system. One of several ways to sift through this information to is to pipe the output to another program for displaying or manipulating it. One option would be to send the output, through a pipe (`|`) to the program `less`, where you could page and search through the output: `ps aux | less`. That is a little friendlier, but what if you know that you are looking for something specific, like the `PID` for a program named `login` (I chose this one because this or something similar should be running on your system). In this case you could use the program `grep` to search through the data coming down the pipe to look for the text "login" and to only print those lines that contain it: `ps aux | grep login`. Try this.

You can string together more commands like this with pipes to do successive operations on the data. At the end you can even redirect the output to a file, as above, rather than having it appear on the screen.

1.7 Additional features of the shell

Earlier you were introduced to command and filename completion, whereby typing the beginning of the name of a filename and then hitting tab will allow you save on typing. Similarly, modern shells (the environment in which you operating at the command-line) offer the ability to move through the history of commands that have already been executed. To step back to previous commands, use the up-arrow key (the down-arrow key moves forward in the history). To search for a particular text string in the history, you may be able to type `CTRL-r` and then begin typing the string you are searching for (hitting the right-arrow will escape this search and leave the text you found on the command-line, to be modified or executed). The history of commands is stored in a buffer of some size and persists across login sessions.

2 UNIX environment and organization

It is misleading to refer to *the* UNIX environment and its organization. Whereas there are conventions that have emerged over time, there are still differences between different implementations of UNIX and certain aspects of a system's organization are highly customizable by the user. However, it is possible to describe key features of UNIX environments, their organization and important aspects that can be modified by the individual user.

2.1 Root: the administrator

One new concept that pertains to this topic is that of a computer's administrator, as opposed to a typical user account. UNIX computers have many different user accounts that are associated with individual, real users. But there are also a number of special accounts that are for specific purposes and have restricted privileges (e.g., a program that provides web server functionality is usually run by the system using a special account, perhaps named 'www', 'http' or 'apache'). There is also a very special, special account named 'root'. This is the account with the highest level of authority on the computer. A real person who administers a UNIX system has access this account, either directly by logging in as *root* with a password, or through elevating their normal privileges with the command `su`.

Check out the several different 'users' who are running programs on your system with `ps aux | sort | less`. The user associated with a process is listed as the first item on each line.

Now look at the ownership of some of the important files that come with the system. For example, have a look at the ownership and permissions for files in `/bin` with `ls -l /usr/bin`. They should all be owned by root (listed in the third column). Contrast this with some of your own files by looking at their ownership.

Note that directories also have an owner associated with them and that important system directories prevent simple users from writing to them. For example, look at the top of the directory hierarchy (`/`) and the permissions for this directory (`ls -al /`; why do have to add the `a` as an option?). Note the permissions for the `.` directory. As a simple user you are in *other* category (not in the *user* or *group* category), with the permission listed in last triplet. This listing probably indicates that you do not have *write* privileges to this directory, which will explain why you can not write files or directories here (try it). It also means you can not accidently delete important files (like those in `/bin` or `/sbin`). These are good things. They keep regular users from spreading detritus around the system in places that are not theirs and from uprooting important system commands and directories. Users are free to muck things up as much as they want in their home directory, because it does not get in any one else's way.

2.2 The bash shell

When you log into a UNIX account you are presented with a command prompt. The interpreter for the commands that you type is referred to as the shell. A commonly used shell is *bash* (the GNU Bourne-Again SHell). Note that this shell provides important functionality to interact with the operating system, but that there are typically several different shells available on any given UNIX system. This is one of several examples of the modularity of a UNIX operating system.

Most of the time you will not think much about the shell. But when you get started on a new system, it can be worthwhile to configure the shell to operate in a way that you like. The settings

Table 1: An example `~/.bash_profile` file. This file is also available on the course website. To use the digital version, you will need to place the text file in your home directory with the `.bash_profile` name. Comments are preceded by `#` character and should give you an idea of what each commands is doing. There are many more variables that could be modified (see the manual page for *bash* and resources on the web about environment variables.)

```
# .bash_profile file for interactive bash shells.
# includes functions and aliases

## set the default permissions for new files
## for files (666 - 022 = 644) this will give:  -rw-r--r--
umask 022

## set the command prompt
PS1='[\u@\h \W]\$ '

## set the title bar for Terminal on MacOS
PROMPT_COMMAND='echo -ne "\033]0;${USER}@${HOSTNAME%%.*}:${PWD/#$HOME/~}\007"'

# aliases to enhance ls and to make file commands interactive
alias ll="ls -laF"
alias ls="ls -FG"
alias rm="rm -i"
alias mv="mv -i"
alias cp="cp -i"

# append ~/bin and current directory to PATH
PATH=$PATH:~/bin:.
export PATH
```

for features of the shell are controlled by variables called environment variables. These can be set for individual login sessions or set as defaults for all logins. To set environment variables for all logins, you can create or modify the file `~/.bash_profile`, which is read in each time the user creates a new instance of a bash shell, including when a new terminal window is opened (Table 1).

You may also want to think a bit more about bash to do certain commands several times, like renaming all files with a certain basename to some new name. This might require shell programming or scripting. You should be able to find good resources to help you with bash programming on the web. Alternatively, anything you might want to do with shell programming you will be able to accomplish with Perl.

2.3 Organization of the filesystem

Many parts of the organization of files into directories on a UNIX filesystem are not important to a typical user of a UNIX system. Instead, the user has full access privileges to a home directory and relies on the system to provide certain programs and libraries. As you have already learned, the `PATH` environment variable contains a list of directories that are searched by default for programs when you enter a program name on the command-line. By default, the `PATH` contains a set of directories that the system provides, but in Table 1 there are directions for adding the current directory (`.`) and a `~/bin` directory to the `PATH`.

The current directory can be useful to have in the PATH when you have small analysis programs in the directory with data files. It is common to have many instances of a program with the same name, but with slightly different contents that are appropriate for a particular dataset. To run a program like this on data, simply change into the directory of interest and run the program. To ensure that you are running the copy of the program from the current directory, rather than from a copy that is elsewhere in your PATH (e.g., from ~/bin), you can specify the current directory in the command: `./myprog datafile*.txt` (rather than leaving the leading `./` off).

Having a directory for binaries (programs) that belong to an individual user can be very useful. This is where a non-administrator user can install programs without affecting things for other users. The (~/bin directory will not exist by default and will need to be created. It is a handy place to put UNIX programs that you use regularly.

It is simple for the system administrator to set up shared folders where files may be shared among users. Some of these exist by default on some UNIX systems (e.g., Mac OS systems have /Users/Shared). For temporary files that do not need to be retained, UNIX systems typically have a /tmp directory (there may be several but /tmp is usually there) that is scrubbed of its contents on some interval that can be set by the sysadmin (system administrator). This is usually set to multiple days or weeks. Temporary directories can be very helpful on shared systems on which there is a quota system that limits storage in your home directory. Some research systems will have specific directories for usage of this type, to separate research files from temporary system files, and these commonly are called *scratch* directories.

2.4 Various UNIX utility commands and programs

In addition to the communications protocols and programs that we will cover in the future, UNIX comes with a large number of programs and commands that will be useful at some point in your work. Equivalent functionality often exists through programs available in the GUI, but it can be useful to have these available at the command-line.

2.4.1 Compression

Large files, particularly large text files containing data, can be compressed to smaller size for long-term storage or for transfer to other computers on the network. Many different compression systems exist and there are UNIX programs to compress and decompress files of all formats. Common formats on UNIX are tar (tape archive, but usually does not involve tapes) that deals with files that have a .tar suffix and gzip that deals with files that have a .gz suffix. Sometimes these are combined to make files with a .tgz suffix and tar can open these (`tar -xvzf myarchive.tgz`, where the x is for expand, v is for verbose reporting of progress, z indicates that the file is also *gzipped*, and f indicates that there is a filename following the command; note that these options may vary on different types of UNIX).

On Windows, people often create ZIP archives and these can opened with the command `unzip myarchive.zip`.

UNIX systems will often include the commands bunzip2 and compress for other types of compressed archives and additional programs are available on the web.

2.4.2 Searching the filesystem

The GUI typically provides a search interface to the filesystem, which can be quite powerful (in the case of OS X, even the contents of files are searched). But you may simply want to find a file within a constrained part of the filesystem, or within the UNIX system directories that are often not searched by the GUI tools. Two tools that are very useful are: `locate` and `find`.

The search tool `locate` depends on a database that is updated on a periodic basis by the system (weekly on my system). This may or may not be enabled on your system, but if it is, then finding a file by name is as simple as `locate myfile`. On a system on which the database has not been created you will get an error. Note that this tool will only find things in the database and that the longer it has been since the last update of the database, the more out of date the results will be. But it is fast and for searching for things that change infrequently it is the right tool.

The search tool `find` is for looking through the current directory contents. It is very powerful and has many options, most of which I have to relearn anytime I need to do something slightly different. But for finding all files that match a particular filename pattern, this is all you need:
`find . -name *data.txt -print`, which translates to: search for files that end in `data.txt` starting in the current directory and descending into subdirectories and print out the results. Useful options include searching for files based on their creation or modification dates, or any other file attribute.

On Mac OS X, there are also command-line tools to search the file metadata that are used by the GUI search tool called Spotlight (e.g., `mdutil`). So if your search needs grow to be significant, you may benefit from these tools.

2.4.3 Miscellaneous

A quick way to see how much space you have available on your harddrives and any attached disks is `df`. I usually use `df -h`, so that the output is human-readable units, rather than bytes. To learn how much space a certain directory use, use `du -h  /research/summer08`.

On a shared system it can be useful to know how busy the system is before you fire up a simulation. The command `w` will give a listing of who is online and how busy the systems processors are (see load averages, which are calculated over three different time windows). More detailed information is available with the command `top` (quit the program by typing 'q').

2.4.4 The Kitchen Sink

A lot things come with a UNIX system, but obviously there are useful, optional things missing. There are many open-source, free programs that run on UNIX, in addition to the well-known commercial applications (e.g., Adobe and Microsoft products).

Apple maintains a repository of open-source UNIX programs for download at:
`http://www.apple.com/downloads/macosx/unix_open_source/`

In addition, there is a community group called *FINK* that maintains UNIX programs for OS X and distributes these (as of Oct. 2008 there are 8982 packages at `http://www.finkproject.org/`). *FINK* includes different SQL databases, *latex*, image manipulation programs (e.g., imagemagick), scientific libraries and programs, etc.

3 UNIX over the network

A key component of research computing involves communications among computers. This can take various forms, including:

1. simple web queries to retrieve data from a public repository or database,
2. logging into a shared computer from a remote location,
3. using software to distribute many tasks across a collection of computers,
4. sharing files across a network to different computers, or moving files between computers over the network,
5. launching a graphical program on a remote computer, but having the user interface displayed on a local machine.

There are multiple options to accomplish each of these tasks. Here I have collected information about communication protocols and software to do research computing in a network environment. This is not a comprehensive treatment, but instead is meant as a starting point.

One common thread in my discussion of these issues is that when given a choice, it is worth considering protocols and software that provide modern security for communications. It is true that we are concerned with scientific computing and not bank transactions. But it is also true that computer security should be on your mind, particularly if you are using a computer that is connected to the internet (or sometimes is connected). Presumably that includes everyone's computer.

Each day there are programs running on remote computers that contact the firewall computer for my lab and search for openings to access the computer. This is roughly the digital equivalent of trying to break into houses by walking down the street and trying the doors on each house to see whether any are open. Similarly, it is straightforward to monitor internet traffic and search for usernames and passwords along with the client and host computers that are involved in communication (this is called sniffing passwords). This is why I will introduce programs that offer encryption of the communication over the network, rather than older programs (e.g., telnet and ftp) that send information unencrypted and bare over the network. This is also why administering your own networked UNIX computer is a task that should be approached with due diligence and caution. For example, it was only a few years ago when the FBI visited the Math department to shutdown a hacked UNIX cluster.

3.1 Remote logins

For many years people used `telnet` to connect to remote computers. This application still exists and some servers still provide telnet service, but `ssh` (Secure shell) is a preferable and widely available alternative. Many UNIX systems have disabled their `telnet` service. `ssh` should be available at the command-line on a UNIX-based system and ssh-clients are also available for Windows (e.g., PuTTY).

One of the advantages of `ssh` is that all communications are encrypted. A second advantage is that it has an option to use key-based authentication, rather having to enter a password for each usage of `ssh` and its companion `scp` (secure copy, see Section 3.2). This can be very useful in a cluster environment and anytime you are doing many operations with `ssh`. Thirdly, `ssh` makes it simple to set up forwarding of `X11` graphics between computers, meaning that you can use graphical programs over the network (see Section 3.4).

Using `ssh` can be as simple as: `ssh frontier.uwyo.edu`. This example assumes that you have the same username on the local computer as on `frontier`. If that is not the case, you can provide your username on the command-line: `ssh mike@frontier.uwyo.edu`. You will be prompted for your password on `frontier` or for your passphrase for key-based authentication, if you have that set up. Note that you may be able to simply give the short version of the server's name, depending on how your host computer is set up (`ssh mike@frontier`).

The details of how to use key-based authentication will depend on which implementation of the `ssh` client server you are using. Here I will outline the concepts and have placed links to additional information on the course website (see `ssh` under Useful Links). The key concept involves one-time generation of a key-pair. One of these is a private key and the other is a public key. The public key is the algorithmic result of the combination of your private key and a passphrase. You could think of the public key as equivalent to the sum of the private key and the passphrase, but summation is a pretty simple algorithm and `ssh` uses more sophisticated ones. The public key is placed on the server that you want to log into. At your client computer, you use software to load your private key and provide your passphrase, and the software calculates a key to be compared to your public key on the remote computer. Only by having both your private key and the passphrase are you be able to obtain a match with the remote public key. Thus you are authenticating your identity locally and `ssh` can then establish communications with the remote computer on the basis of that authentication. In addition, `ssh` provides other checks to verify that the computer you are contacting is in fact the host you requested.

`ssh` can do other things besides only logging you into a remote computer. For example, you can use it to run single jobs on a remote computer and then quit, without ever opening an interactive login session.

Any options for `ssh` that you want to store for individual servers (e.g., a specific username for a server), or for the overall behavior of the program, can be stored in `~/.ssh/config`.

3.2 Moving files

We would not get very far with computers if we could not copy files over the network. Many protocols exist for moving things across the network. A venerable option is `ftp`, but it has been superseded by more secure offspring (`sftp` and `scp`) and smarter and more capable offspring (`rsync`).

Using `scp` is analogous to the UNIX `cp`, except that the source and destination specification for the copy operation has been extended to include remote computers. So,
`scp myfile.txt frontier.uwyo.edu:newfilename.txt`, would copy and rename the file when it is placed on `frontier`. As with `ssh`, you will be prompted for a password for `frontier` or key-based authentication will be used if you have it set up. Similarly, the source of the file can be remote, as in `scp frontier.uwyo.edu:myfile.txt .`, which would copy the file to the local directory, without renaming it. And both source and destination can be remote, as in

```
scp frontier.uwyo.edu:myfile.txt einstein.unix.org:
```

which implies that `myfile.txt` would be copied into the user's home directory on the computer `einstein`.

If you are moving big files over a slow network, you should consider using compression in `scp` (with `scp -C`).

3.2.1 Remote synchronization of directories (rsync)

It is common to want to copy a whole directory between computers, between a computer and a USB drive, or to want to bring an older copy of a remote directory up-to-date with changes that you have made to a local copy. In the last case, ideally you would not copy all files again, but instead would only copy those that have changed, particularly if a lot of data are involved. One tool for these jobs is `rsync`. If this synchronization is over a network, it can be done securely through an `ssh` connection.

1. `rsync -av datadir /Volumes/USBDrive` – copies the directory `datadir` into the top directory of a removable USB device. If `datadir` already exists in that location on the USB device, only differences between the directories will be copied.
2. `rsync -av -e ssh datadir einstein.unix.org:` – copies the directory `datadir` into the users home directory on the `einstein` server.

In the examples above, the `-av` options refer to "archive" and "verbose" output. There are many other options, including the possibility to prune a remote tree and delete old files that have been deleted locally (`--delete`, see `man rsync`).

3.2.2 Hypertext transfer protocol (http)

You are very familiar with `http` because it is the protocol that computers use for communications that we refer to as the "web". We use "browsers" such as *Firefox*, *Explorer* and *Safari* to contact remote computers using `http`. Even though it is familiar, I mention it here because it is an example of file transfer and because it can be used to push and pull files on the network. When you click on a link in a browser, you are requesting a copy of a file from a server and your browser receives and interprets it. Similarly, you know that with web-based mail programs or with the class website, you can push files on to a server (as an "attachment"). There is nothing categorically different about these operations than the ones I described above, except that they are accomplished with a different protocol.

Also, it turns out that browsers are not the only programs that can use `http`. A very useful program for downloading large files with `http` is `curl`. For example, suppose you wanted to download a whole set of CD images or a DVD images that contains a distribution of the linux operating system or some other large thing. You can do this with a browser, but if the transfer were interrupted for any reason there is a good chance that you would have to start over from the beginning. I am thinking of files that take a substantial time to transfer over a fast connection (e.g., 30 minutes). `curl` can resume file transfers from the point that they were interrupted and has other nice features.

For example, to download a 359 megabyte CD image that has a small linux installation that can be run from the CD drive on a PC, you would do the following (note that `curl` can do `ftp` or `http`):

```
curl -O ftp://ftp.ussg.indiana.edu/pub/xlivecd/xlivecd-20041201.iso
```

3.3 Private networks

Private networks are physically separated from the world-wide-web by a firewall and other software that limits access to them. The computer network at the University of Wyoming is a private network. This means that trolls out on the web should not be able to contact individual computers and bang on them to see whether there are any open doors. There are exceptions, like the computer that is the firewall to my lab network, but this is intentional, so that collaborators and users from off-campus can get access to the lab computers.

It is possible to connect remotely to the university's private network. The software to do so establishes a *Virtual Private Network* (VPN). Once the VPN is established, to computers on the private network it appears that your computer is also on the private network and can therefore access private resources.

An important private resource on networks involves access to shared filesystems. You are probably familiar with "Network Drives", "Network Shares" and other terms for referring to interfaces to data stored on remote computers. There are a number of protocols for making drives accessible over the network (in addition to those above). Three important ones are: *NFS* (Network File System, UNIX servers), *AFP* (Apple File Protocol, Mac OS Servers), and *SMB* (Server Message Block, Mac OS and Windows Servers). UNIX machines can access, or mount, directories using any of these protocols. Windows computers can only access SMB shares.

In a computer cluster environment, users' home directories will typically live on one drive, but will be "shared out" or made available to individual computers using one of these protocols. Access to these resources is restricted to the local network, so private networks provide a layer of security and appropriately limit access.

When a remote drive is accessed, it is placed into the UNIX filesystem hierarchy, and it can be accessed from the command-line. On Mac OS X, remote drives show up in `/Volumes`, just like removable devices, if they are mounted manually by the user. If the remote drive is mounted by the system, it shows up in `/Network/Servers`.

3.4 X11

The graphics on many UNIX computers are drawn using software called `X11` along with a *windows manager*. This software has been around for a long time and is installed even on modern operating systems that also have a different display system (Mac OS X). In this case, `X11` has been modified to run alongside the main graphical software.

It is reasonable to wonder why `X11` hangs on despite its age and other graphical software. One reason is that it has features that the newer software lacks and another reason is that there is a lot of software out there that was written to operate under `X11`. In other words, it is still useful.

To use `X11`, one launches it on the local computer and this becomes the `X11` "server". Client applications may be launched on the local machine and they will be permitted to connect to the "server" and appear on the computer's display. Similarly, if properly set up, client applications can be launched remotely and can connect to the "server" and appear on the computer's display. This is where `ssh` comes in, because it makes forwarding `X11` clients easier. All you need to do is invoke `ssh` with the `-X` option to request "X11 Forwarding" (alternatively you can put `X11Forwarding true` into your `ssh` configuration file, `~/.ssh/config`). It may also be the case that `X11Forwarding` is on by default.

If `X11` is installed on a Mac OS X computer, it will be in `/Applications/Utilities`. It is an option in the installation, so it will not be on all Mac OS computers (but can be found on the install discs). `X11` is running on all linux computers.

For example, assuming that `X11` is running, try `ssh -X frontier.uwyo.edu`. Then launch the text editor `xemacs` with `xemacs &`, or open the statistics software SPSS with `spss &`, or Maple with `xmaple &`

Somewhat similar results can be obtained using *Virtual Network Computers* (VNC) connections to computers. These allow only a single user to connect to the machine and then send all graphics over the network, not just the graphics for the individual application.

4 Text files and editors

4.1 In the beginning there was text

It all seems very simple. We get and send text files in email or via http, we create text files with editors, and we look at and manipulate them with various programs. Much of our work does not require us to know much about this file format, but a few insights into the underlying technical aspects are useful. For one, this gives us a basis for understanding the differences between file types and their advantages and limitations. Text files are the common denominator of data storage, so it pays to understand what they entail.

On some level you know that data stored and utilized by a computer consists of binary data, `0`s and `1`s, electronic bits that are either set or not, and that storage of these bits occurs electronically in the processor, in RAM (random access memory), or in various other repositories of data (caches, solid state memory, hard disk memory, etc.). You may also know that for the telegraph (and for some radio communications) people encode(d) the alphabet and numbers using Morse Code. Today one of the simplest methods of digital encodings for the English Alphabet is the *American Standard Code for Information Interchange* (ASCII)[1]. People often refer to data that is encoded this way as plain text. This is true in many cases, particularly when referring to many of the files that contain scientific data, but computer text and its encoding has also evolved beyond ASCII and other methods of encoding exist (particularly for all those languages other than English that need characters not in the ASCII set). Encodings of text that go beyond ASCII are particularly important for documents that are exchanged internationally and need to be rendered on computers other than the one on which the document was created.[2][3]

For now, scientific data from electronic instruments, public databases and other sources are often encoded using ASCII, or a modern superset UTF-8. But what does it mean for data to be encoded? Literally, it refers to the translation from a character to binary storage and vise versa. We know that data are stored as bits. If a file is an ASCII text file, then those bits are read in sets of eight, an 8-bit byte[4]. An 8-bit byte has $2^8 = 256$ possible states (only 7 of the bits are used for actual encoding, so it is $2^7 = 128$), which is not a big number, but plenty to encode the English alphabet, numbers, some symbols, and funny, archaic things like "form feed" and "bell". ASCII is a translation table between binary values and letters of the alphabet, among other things, so that `1000001` is equivalent to `A` and `1100001` codes for `a`.

ASCII is useful because it is fairly compact storage.[5] With one 8-bit byte, we can store an alphanumeric character. Text files that use ASCII are the common currency for storage and you have already seen that configuration files for UNIX are simply text files. Similarly, files that contain computer code to be interpreted by R and Perl, by a web browser (html), or to be compiled as C code, are written as text files. Beyond this, text is the common currency for retrieval of data from databases and for processing in software pipelines.

[1] http://en.wikipedia.org/wiki/ASCII

[2] http://en.wikipedia.org/wiki/Unicode

[3] http://en.wikipedia.org/wiki/UTF-8

[4] You may be aware that computer architectures and operating systems today are 32-bit or increasingly 64-bit systems. That is a different issue and involves the amount of memory space they can address and keep track of. A 32-bit register is limited to being able to keep track of 4 GB of RAM.

[5] It is not terribly compact. Often considerable storage can be saved by "compressing" text files with compression software (`gzip`, `zip`, etc.). Loss-less compression involves achieving greater efficiency in the encoding of data without throwing any of it out.

In scientific computing you will not often need to know more than that you have a text file and you will be able to work with it.[6] But knowing a bit about what text files are gives us a basis to discuss files that are not text and how to create text files properly.

4.2 Other file types and encodings

All files contain binary data. Text files are a special class of file, with standards for their encodings and storage methods. But beyond simple ASCII text files, the computer world is full of files that are associated with particular programs and use their own special method of encoding.[7] This is to be expected, because even in tasks as simple as word-processing, not only do we want to encode the letter A, but we want to specify that the letter is to appear on paper of certain size, a certain distance from the paper edge, and in a given size and particular font. And where would we be if music could not be encoded in a file format?[8] You get the idea, there is more to encode than alphanumeric characters. Consequently different formats have been and are developed for encoding data and the formats are associated with corresponding software. Surely you have experience with files that are in a given format but you lack the software to read that format. For example, to read a file in Photoshop file format, you need Photoshop or something that has been made to read this file format, whereas a lot of the data in the file could have been encoded in an open image format like `jpeg` or `tiff`. Some file formats are proprietary and their encodings are secrets, and the companies that own these do not want users or competitors to know how to read the files without their software.

To get data out of a file format that is restricted to a particular program, often you have the option to export the data and save it to another format from within the program. It is best to do this before the program becomes obsolete. You will be very unhappy in ten years if you can no longer access important data. Text files will outlive us, but WriteNow and WordStar files will not (both are essentially dead, even though they were popular or dominant word processors in their day). No one knows what will happen to proprietary file formats associated with biotechnology applications.

Alternatively, often there are converters to read certain file formats from within a programming language, because the format of encoding has been released to the public. For example, the statistical software R can read from SAS files, ESRI shape files (for GIS), various file formats for microarray data (incl., Affymetrix, et al.), etc. Similarly, Perl has many add-ons to read file formats. The point is that these have to be programmed by someone and that this is a complex undertaking. It is not something that you can expect to do yourself with a little computer experience. Either you will find a way to export data to text from within the specialized software, or you will find an existing way to read the data into the next step of your analysis using conversion software.

A final issue to raise is the meaning of the three letter extensions that are part of filenames. You should know that these are simply a convenience for the operating system and you to know which programs to use to open which files. They should reflect the file type and the encoding that is used for the data, but they do not have to. That is, you can name files without an extension and still open them using the corresponding software (even if it sometimes will need prodding and convincing to trust you that the file really is in Word 2007/2008 format). You can also rename

[6] One exception to this is that Windows, UNIX and Mac OS have historically used different characters to end a line. Many programs have been written to recognize any of three options as a line ending, but in writing your own programs, particularly for reading data that you have received from other users, this can be a small obstacle.

[7] For fun, try examining the contents of a Word or Excel file using the UNIX program `less`.

[8] Answer: in the 1970s, prior to the introduction of the compact disc.

files with the wrong extension and not change the contents. It is true that once you rename a file from `myfile.pdf` to `myfile.xls` (after waving off a possible warning), double-clicking on the file will try to open the file in Excel and fail. But if you navigate to the file from within Adobe Reader (or another pdf program), you will be able to open it, despite the nonsensical and misleading file extension. File extensions are part of a file's name and are used as hints for the operating system for associating files with particular programs. The operating system allows you to choose which programs to associate with files with a particular extension. But it is the file contents themselves and their encoding that dictate which programs can read them.

4.3 Text Editors

You already know that there are many programs that can read text. That leaves the question of what programs should you consider using to write text, particularly to write code in a computer language. There are many to consider and the choice can come down to personal preference and habit. For simple things you can get by with TextEdit (OS X), NotePad or WordPad (Windows), but be sure that you save files as text, not as RTF files (Rich Text Format, not a simple text encoding).

It is unlikely that you will be satisfied working with these simple editors if you end up programming very much. Instead, you will want an editor that is aware of the syntax of the computer language that you are writing. With such an editor you will have the option of turning on syntax highlighting and many other options that will help you write syntactically correct code and work more quickly in finding errors or modifying existing code. Editors for programmers come with the ability to work with many programming languages by default and often can be enhanced by add-on modules for other languages.

Two featureful and free text editors to consider are Emacs (all OSes) and TextWrangler[9] (OS X only). A version of Emacs for the command-line and for use with X11 is pre-installed on most UNIX computers. Emacs is also available as a program compiled for OS X graphics[10] and for Windows.[11] Learning Emacs is worthwhile, because it is widely available. Otherwise, for simple tasks on UNIX, do not forget about `pico` and `nano`, which are simple editors that list their key commands in the terminal window. In a pinch, you can write text in Word and save the file as "Plain Text", but since Word knows nothing about programming languages you are just as well off using a very simple editor.

[9] http://www.barebones.com/products/textwrangler/

[10] http://www.apple.com/downloads/macosx/unix_open_source/carbonemacspackage.html

[11] Emacs is a bit more difficult to get running on Windows, so you may want to consider other editors on that platform.