Course Notes for Data Analysis in Geophysics Documentation

Release 1.0

Eric G. Daub

Nov 08, 2018

CONTENTS

CHAPTER ONE

INTRODUCTION TO THE COURSE

Why does this course exist? Doing research in Geophysics requires processing *lots* of data. As an example, Hi-net is a seismic network operating in Japan. It consists of 1000 stations, each recording 3 components, at a frequency of 100 Hz. This turns out to be about 9 TB of data *every year*.

We could not possibly process this amount of data by hand. Instead, we need to automate this task using a computer. This course is meant to help you learn how to do this.

But we are not here simply to automate data processing tasks – we are scientists, so it is important to keep in mind the particular needs of scientists when writing computer programs to analyze data.

The scientific method says that we need to do tasks

1. Accurately
2. Reproducibly
3. Efficiently

These are listed in order of importance, as there may be tradeoffs between these goals in a given task. For instance, an earthquake may only happen once, so we cannot expect reproducible results in that case. Note that efficiency is last – it is far more important that you write code that is correct, easy to understand and maintain, and produces reproducible results than code that runs fast.

1.1 What is data?

At the basic level, data is a bunch of numbers. However, that by itself is not enough, as we need to know units, as well as how the data was collected, where it was collected, etc. This is usually called *metadata*, which means data that describes other data. How do we handle data on a computer?

1. **Binary** (i.e. sequences of 0s and 1s) – A computer treats everything as a sequence of 0/1, so perhaps this is a good approach.

   Pros: This turns out to be very efficient, as we do not add anything extra above what the computer needs to understand the numbers

Cons: The data is not human readable, and there is no way to include metadata. Byte ordering is not standardized (artifact of history of computer hardware development)

2. **ASCII** (i.e. text file) Data is represented as a series of text characters.

   Pros: Human readable format, also allows for combination of data and metadata in the same file

   Cons: Text representation of numbers is less efficient than straight binary format, so datasets can be very large if there is a great quantity of data.

3. **Combination Format** Format combining both text and binary, for instance text metadata and binary for the numerical aspect of the data. Also includes proprietary formats like Microsoft Excel.

   Pros: Overcomes limitations of both file formats.

   Cons: Need to define a standard format for doing so, but there are often competing version for a particular type of data. Also need special software to read data formats, as special formats may not be built into all software packages. Proprietary formats may not be useful without the special software needed.

1.2 What does Data Analysis involve?

1. **Reformatting data:** You may get data in one format, but software that you have requires another. For instance, you might get a text file that contains all the correct information, but needs to be put into a different text format for use with another program. Or you might need to convert from ASCII to binary or vice versa.

   This is something you should always automate! Human typing errors are very common. Computers do not make mistakes or get tired.

2. **Processing data:** This is the actual number crunching part. You might use a program from someone else, or a program that you write yourself.

   This course uses *high level* tools for analysis. High level means that there are many built-in tools and capabilities in the software. This is a good approach for many reasons

   - Shorter code is easier to debug, so more likely to be correct.
   - Built-in tools are likely to have been more rigorously tested than something you write yourself. Also more likely to be more efficient than your implementation. It is a good idea to avoid re-inventing the wheel!

   Eventually, you may need to write your own code and for performance reasons you might need to use a lower level programming language. Fortunately, writing a computer program is highly transferrable across different programming languages, so if you get good at using a high level language, it will not be too difficult to translate your skills to another language as nearly all of the same concepts will still apply.

3. **Representing processed data:** In publications, you are most likely going to publish plots and maps, rather than the data itself. We will explore computer tools for making graphs and maps in this class, which are much more efficient than drawing the plots yourself in a drawing program.
4. **Automating everything:** To do your analysis in an accurate and reproducible fashion, you should automate as much as possible. This reduces errors, increases efficiency (computer programmers are notoriously lazy), and makes things easy to do again in case you realize you need to do something differently.

CHAPTER

TWO

UNIX BASICS 1

As discussed in the first lecture, analyzing data often requires several different tools that you must string together and run on a large number of data files. This would be tedious if done by hand, and you are very likely to make a mistake. The Unix shell is the tool that is most commonly used to tie together different tasks. I use a Unix shell every single day, much more often than any other tool that we will be covering in this class, and thus it is very important to develop your Unix skills.

We often have to interact with the operating system to get the computer to perform some task. In many cases, this task is not particularly straightforward without a large amount of work on your part. Let's say that in your research, you are detecting and locating earthquakes, and have some computer code that analyzes a large number of waveforms that you automatically download from the internet every day. To do this efficiently, you would need to figure out which files on your hard disk include new waveform data, and process only those data files. To do this manually, you would need to look through potentially hundreds to thousands of files to find the few newest ones, and then manually type all of them into some file to analyze those files. That would be extremely time consuming, and you are also likely to make some sort of typing error in the process, or miss some new piece of data.

Instead, what would be ideal is to be able to tell your computer to find all of the files in the folder where you keep your waveforms that were created in a certain time period (say, since you last performed your analysis), and then feed those into your code to do the analysis. Computers are very good at doing such rote tasks quickly and efficiently; we just need to know how to explain to the computer what we are trying to do. This is not easy to do on a Windows computer or within the operating system GUI on a Mac or a Linux machine, but it is very possible (notice that I did not say easy!) using the Unix command line.

In addition to my notes on Unix, I have included a link to a book on the course website that provides an introduction to using the Unix terminal on Mac OS X. You should read through both of these before coming to class, and be ready to spend the class time playing around with the shell to start getting comfortable.

2.1 Unix Philosophy

The Unix family of operating systems originated in the late 60's/early 70's at AT&T's Bell Labs. Since then, many different versions have been developed that are very similar (but not exactly alike), including BSD Unix (developed at Berkeley in the 70's), Sun Solaris (now owned by Oracle), and the various flavors of Linux. The Mac OS is a descendant of BSD Unix, while some of the other Unix-like operating systems that you are likely to encounter (Sun, Linux) broke from the original Unix system at different points. This means that there are usually slight differences between the operating systems in terms of the specific commands. However, the "Unix Philosophy" is generally the same across these operating systems and the differences usually involve only slight tweaks to the command syntax.

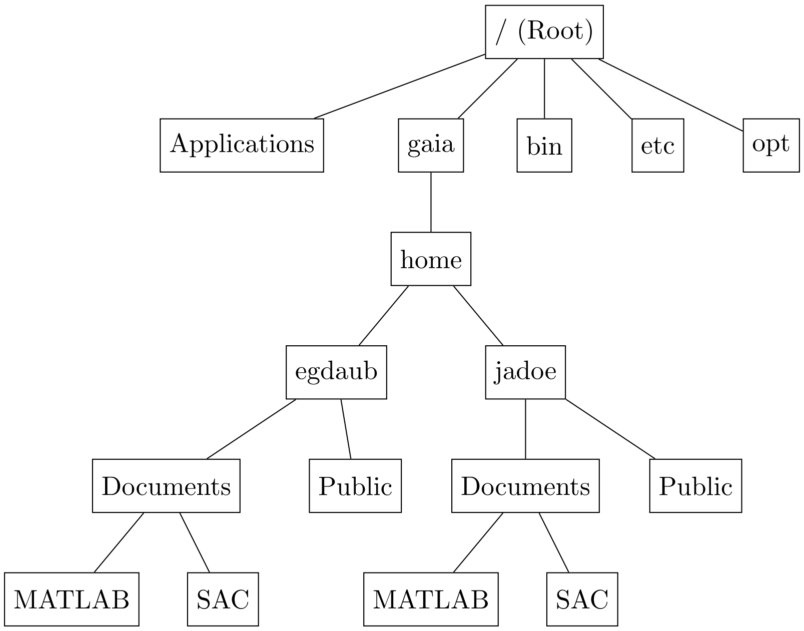
Put broadly, the "Unix Philosophy" is one of modular commands that are designed to do just one simple, well-defined thing and to do it well. If we need to do something different, rather than modify that original command, we start from scratch and make a new command, and then have some way for the two commands to interact so that we can string them together and perform a task. This is in contrast to Windows (and the GUI-side of Mac OS), where the operating system has been designed to help even the novice user perform simple tasks. Because the Unix commands are not simple and obvious, it has a much steeper learning curve than using a system like Windows – in Windows, the smarts are stored in the OS, while in Unix the smarts are stored in the user. It has also been said that "Windows makes easy tasks easy; Unix makes difficult tasks possible."

Unix systems are multitasking and multi-user (note that any user can use any computer in the Mac Lab, and the machines have no trouble performing more than one task at a time). Originally, the only way that the user could interact with the operating system is through the command terminal (either a teletype where you typed commands and then a printer printed out the response or a single terminal window on a monitor), but most Unix systems these days have at least a file system browser that shows you the filesystem graphically. However, when you connect to a computer remotely, you still need to do so using a terminal. You may occasionally have a problem with your computer and need to boot in what is known as "safe mode" where you only can access your machine through a terminal. This means that knowing how to do everything in a terminal is a must for any Unix user.

2.2 Browsing the Filesystem

Enough about philosophies and history: how do we use a Unix Terminal? Open the terminal and type `pwd` followed by a carriage return. The terminal should print out `/gaia/home/<username>/`. This is what is known as your home directory (or folder), and by default your new terminal sessions start in this directory. The command `pwd` is the Unix command that tells you the current working directory (`pwd` is short for Print Working Directory). Note how the command follows the Unix philosophy: it does only one thing.

Unix uses a hierarchical filesystem. The topmost directory is what is known as Root, and all other directories fall into a place within the hierarchy, illustrated below (note that this isn't complete, it just illustrates the big picture):



Note that in the directory `/gaia/home/` there are multiple directories, with each directory corresponding to one user. This highlights the multiuser nature of Unix: every user has their own space to store things. We can specify a location in the filesystem with either a full name (i.e. `/gaia/home/egdaub/Documents/MATLAB`), or we can specify a relative location from the current directory (if the current directory is `/gaia/home/egdaub/`, then we could simply specify `Documents/MATLAB`). Both are equally valid in Unix.

Because of how Unix commands work, it can be slightly annoying to have files and directories with spaces in their names (spaces usually designate the end of one argument and the start of the next). Many people choose to avoid having spaces in file and directory names, instead using the underscore character _ in place of the space (you may have noticed that I do this with most of my files). You can use spaces in file and directory names, but when entering the name into the Unix command line, you need to put a backslash (\) prior to the space to tell the terminal to interpret that as a space within the path rather than the end of the path. For instance, the path to the directory `Drop Box` in all of your `Public` folders must be entered into the terminal as `/gaia/home/<username>/Public/Drop\ Box` to avoid problems.

We can change our position within the hierarchy using the `cd` command (short for Current Directory or Change Directory). If we are in our home directory, and want to change to the `Documents` directory, we enter `cd Documents` (Unix is case sensitive, so you need to capitalize `Documents`). Try that, and then use `pwd` to verify that we are in the `Documents` folder. Note that we can use either the relative or the full location in the filesystem when using `cd`.

What if we need to descend a level in the hierarchy? We could type out the full location of the directory one level higher, but if we are very deep in the chain and just want to go up one level, that would be annoying. Instead, we can use `..` (two periods) to refer to the directory one level higher. Thus, to change from `Documents` back to our home directory, we can type `cd ..` to go up a level. Note that we can combine `..` with other directory locations: if I am in `Documents` and want to enter my `Public` folder, I can enter `cd ../Public` and make the change with a single command.

One other thing that is handy when navigating the file hierarchy is `~`, which is the same as the home directory. To navigate to your home directory from anywhere, you can type `cd ~`.

Once we are in the desired directory, how do we see what is in that folder? We use the `ls` (list) command, which lists all of the contents in the current directory. Try it out – it will print all of the files and directories in the current directory to the terminal, like this (obviously your filenames will be different):

```
MATLAB                         data_syllabus.synctex.gz
awk                            data_syllabus.tex
ceri7104_dataanalysis.docx     gmt
compexam                       homework
csh                            lectures
data_syllabus.aux              python
data_syllabus.log              sac
data_syllabus.pdf              studentwork
```

However, you may notice that unless you already know what these files are, the standard `ls` is not particularly useful. To see more information, enter `ls -l` to see more information about all of the files, which should print out something like this:

```
total 472
drwxr-xr-x   53 egdaub  staff    1802 Nov  4  2015 MATLAB
drwxr-xr-x    2 egdaub  staff      68 Aug 10  2015 awk
-rw-r--r--@   1 egdaub  staff  147712 Sep 25  2015 ceri7104_
↪dataanalysis.docx
drwxr-xr-x    3 egdaub  staff     102 Jan  7  2016 compexam
drwxr-xr-x    9 egdaub  staff     306 Dec 17  2015 csh
-rw-r--r--    1 egdaub  staff      84 Aug 25  2015 data_syllabus.aux
-rw-r--r--    1 egdaub  staff    5231 Aug 25  2015 data_syllabus.log
-rw-r--r--@   1 egdaub  staff   52581 Aug 25  2015 data_syllabus.pdf
-rw-r--r--    1 egdaub  staff   15882 Aug 25  2015 data_syllabus.
↪synctex.gz
-rw-r--r--    1 egdaub  staff    4942 Oct 19  2015 data_syllabus.tex
drwxr-xr-x   56 egdaub  staff    1904 Dec 17  2015 gmt
drwxr-xr-x   77 egdaub  staff    2618 Dec  7  2015 homework
drwxr-xr-x  148 egdaub  staff    5032 Nov 24  2015 lectures
drwxr-xr-x   12 egdaub  staff     408 Oct 19  2015 python
drwxr-xr-x   11 egdaub  staff     374 Nov 19  2015 sac
drwxr-xr-x   12 egdaub  staff     408 May  2 12:34 studentwork
```

This tells us much more about the files and directories in here. The first column tells us several things: the leading `d` tells us whether or not this item is a directory, and then the remaining `r`, `w`, and `x` items tell us the permissions of a file, which we will discuss in the next lab. The `@` is specific to Mac OS, and means that there are additional file attributes that we can view. The second column tells us the number of items in that particular directory or file, and the third and fourth column tells us who owns the file, and the group that that user belongs to (`staff`). Then we have the size of the item, followed by the date and time when it was last modified, and finally the name. This gives us much more information about the files which could be potentially useful to us.

Why did we get different output that was different in this case for the same command? The addition of the `-l` to the command, which is an example of a command line option that told the OS that you wanted a particular version of the command (in this case, `-l` means that you want the "long" version of `ls`). Many commands have numerous options that you can specify – `ls`, which is one of the simplest and most used commands in Unix, has almost 40 different options.

`ls` is not restricted to listing all of the contents in the current directory. If you give `ls` a file name, it will list that file, or give you a message telling you that file does not exist. While this may not seem useful right now, when we talk about wildcards you will see how this can be a useful tool. Also, you can give `ls` a directory, and it will list all files in that directory without having to make that your working directory.

As you navigate through the filesystem, you will find a couple of terminal shortcuts to be useful. You can access your command history using the up and down arrows, so if you want to re-enter a previous command, use the up arrow to bring it back into the command window. You can also tab-complete files; if you type part of a file name and then press the tab button, the terminal will complete the file name if there is a unique file that starts with the characters you have typed. If there is not a unique file, then the terminal will complete it as far as it can until it reaches an ambiguity. If you are already at a point where there are more than one possible completions, it will not complete, but if you press tab again, it will bring up all options that match what you have typed so far.

One other command that I use often is Control-u, which deletes everything that you have typed so far in a command. This usually happens to me when I have started a long command, and then realize that I forgot to enter some other command first. Control-u clears all of the text in the command line so that you don't have to delete the entire string of text manually.

2.3 Getting Help

What if you need a specific option among the 40 options within `ls` but you don't know what it is? You can read the manual for a Unix command by typing `man <command>`. For instance, to see all of the options for `ls`, enter `man ls` into the command line. You should see a long page describing `ls` and all of the options. Scroll down with the down arrow (one line at a time), the space bar or "f" key (one page at a time), or back up with the up arrow key (line) or the "b" key (page) to read the documentation. Type "q" to exit the manual page. If you ever do not know how a command works, the manual page is often helpful (though in the spirit of the Unix Philosophy, they can often be rather opaque).

2.4 Viewing Text Files

Using `cd` and `ls` we have found the directory and file that we are interested in. How do we view the contents of a text file? There are several commands that can be used in the command line to do this.

If you just need to see the first several lines of a file (say you want to double check that the file is what you think it is), you can use the `head` command. If you have a text file `textfile.txt`, you can view the first several lines with `head textfile.txt`. Similarly, if you just want to see the final lines of a file, you can use `tail`.

What if you want to look at more than just the beginning or end of a text file? You can print the entire file to the screen using `cat` (short for concatenate). To view the file `textfile.txt`, enter `cat textfile.txt` to see the whole file printed out to the terminal.

(One technical note: you may have wondered why a command to print a file's contents has the name concatenate. This is due to some of the peculiarities of Unix. More generally, `cat` can take several text files and concatenate them into a single file; here we just give it one file so it concatenates it with itself. `cat` takes the concatenated file and sends it to what is known as "standard output," which in this case means that is prints the file contents to the terminal. This is how we can use a command that was designed to concatenate files to simply display a file in the terminal.)

If our file is short, then it does not matter whether we use `head`, `tail`, or `cat` to display a file. However, if our file is very long, then `cat` will just dump the entire file to the screen, regardless of how long it is. For us, this is not a big deal (we can just scroll up in the terminal window to see the entire file), but for past users of Unix that had a terminal that could not scroll, this would pose a problem as we would only see the end of the file. (Users using the teletype would be able to read the entire printout, but for very long files the file would take a very long time to print, which would be rather annoying). To make looking at files more manageable, Unix has the `more` command, which prints a file one page at a time. However, `more` can only scroll forward, so the ironically named `less` command that allows for both forward and backward scrolling has superseded `more` (on a Mac if you type `more` you will actually run `less`). You can scroll forward page by page with `less` using the spacebar or the "f" key, or line by line with the arrow key. To go backward, use the up arrow to go line by line or the "b" key to go page by page. Type "q" to exit `less`.

(The differences between `more` and `less` lead to the often repeated Unix joke that `less > more`, which is supposed to make it easy to remember which command to use.)

One other useful command line tool for learning about files is `wc` (Word Count). `wc` can count the number of words, lines, characters, and bytes in a file. Usage is `wc <filename1> <filename2> <filename3> ...` where the arguments are the names of the files that you wish to evaluate. For example, to see the word count for the file `testfile.txt` enter `wc testfile.txt` into the terminal. You can use a number of command line options: `-l` only counts the number of lines, `-w` only counts the number of words, `-c` only counts the number of bytes, `-m` counts the number of characters, and `-L` determines the length of the longest line. I often use `wc -l` to determine the size of an ASCII dataset where each line contains a data item so that

I know what to expect when loading the data into MATLAB or some other program.

2.5 Editing Text Files

To edit a text file from the terminal, we can edit a text file using a number of different command line editors. While it is certainly possible to edit a text file on a Mac using the TextEdit program, if you are connecting to a Unix system remotely, or need to start your Unix machine in safe mode, the only way you will be able to edit a text file is through an editor. Knowing how to use a terminal-based text editor is an essential Unix skill, and most find that in many cases, a terminal-based text editor is faster and more efficient than an editor like TextEdit.

The Mac OS comes with several Terminal-based text editors: `nano` (same as `pico`), `vim` (same as `vi`), and `emacs`, plus possibly more that I do not know about. Most of these editors exist on other Unix systems, and you should be comfortable using one of them. You should also have a basic knowledge of how to use all of them, in case you find yourself on a situation where your favorite editor does not exist on an unfamiliar computer.

If you already know how to use one of these editors, great. You should feel free to use whatever editor you like in this class. (Personally, in my first Unix experiences when I was an undergraduate, I used `emacs`, and so that is still my editor of choice.) However, it is *always* a good idea to be able to use several command line text editors. If you have never used any of them, I recommend starting with `nano`, as it is the simplest and shows you keyboard shortcuts on the screen. If you already are comfortable using one of these editors, I would spend time during the class period practicing with a new editor. There is a built in tutorial for `vim` that you can access by typing `vimtutor` into the shell. You can also find tutorials for any text of the text editors using a web search.

2.6 Managing Files and Directories

Now that we know how to get around, view, create, and modify text files, we may want to rearrange our files in some way. There are a number of commands to do this.

First, to make a new directory in the present directory, use `mkdir` followed by the name of the new directory. For example, `mkdir ~/Documents/SAC` will create a directory name `SAC` in the `Documents` folder. You can also do this using the Finder, but it is usually faster to use `mkdir`.

To copy a file to a new directory, use the `cp` command (copy). The syntax is `cp <file> <newfile>`, so to copy the file `textfile.txt` to the file `otherfile.txt` type `cp textfile.txt otherfile.txt` to copy this file. If you want to copy a file to a different directory but keep the same name, you can replace the new file name with an existing directory. If I have a file `textfile.txt` and wand to place a copy in my `MATLAB` directory, I would use `cp textfile.txt MATLAB` (assuming that the `MATLAB` directory was in the current directory; otherwise I would need to specify the full location of the `MATLAB` directory). Be warned that if you specify a target file that already exists, the file will be overwritten (often without a warning!).

To move a file to a new location (or rename it in the same location), we use the `mv` command (move). `mv` behaves much like `cp` but the previous version of the file no longer exists after using `mv`. To rename `textfile.txt` to `samefile.txt` I would type `mv textfile.txt samefile.txt` into the terminal. Again, the same caveat of not overwriting an existing file that applies to `cp` applies here.

You can delete files with the `rm` command (remove). You should be *very* careful with this command, as when you delete a file from the command line it is gone forever (Windows and Mac users will be familiar with the Recycle Bin and Trash, respectively – `rm` does *not* send files to these locations). For this reason, it is often suggested to use `rm -i` instead of `rm`. `rm -i` will ask you if you really want to delete the file and require you to type `yes` or `y` into a prompt to confirm. This is in general a good idea, and many system administrators make this the default behavior (i.e. when you type `rm`, the terminal automatically interprets it as `rm -i`). However, when this is the default behavior, it encourages people to be lazy with their responses (automatically typing `y` for a large number of files) or lazy in using wildcards to select files to delete knowing that there will be a chance to use a prompt to prevent deletion. For this reason, I try to just avoid using `rm` as much as possible from the command line and delete things from the Finder so that they go in the Trash rather than being deleted forever. This same `-i` option is available for `mv` and `cp` in the event that you try to overwrite an existing file using those commands.

Finally, to remove a directory use `rmdir` followed by the directory name. The directory must be empty before you remove it. There is a way to remove a directory that is not empty using `rm`, though it can be dangerous if it is mistyped because if you accidentally specify a directory high in the hierarchy, you can delete your entire home directory (and if you have administrative privileges, you could potentially delete the whole file system!). My preference here is also to delete things from within the Finder so that I can retrieve things from the Trash if I make a mistake.

2.7 Summary

Here are a list of the commands that we introduced in this lab. They are among the most common, and you will likely find yourself using them very often:

- `pwd`
- `ls`
- `cd`
- `..` (directory one level higher)
- ~ (home directory)
- `man`
- `head`
- `tail`

- `cat`
- `less` (same as `more`)
- `wc`
- `nano`
- `vim`
- `emacs`
- `mkdir`
- `cp`
- `mv`
- `rm`
- `rmdir`

CHAPTER

THREE

UNIX BASICS 2

Hopefully you are becoming familiar with the terminal environment in Unix. Today we will examine more basics for working in Unix.

3.1 Wildcards

When entering file and directory names, Unix gives us a number of special ways to refer to filenames, knowing as "globbing" using "wildcard" characters. Wildcard characters that have a special meaning in the terminal include `*`, `\`, `?`, `^` and `[ ]` and can be used to designate various patterns. One example is the `*` character, which stands for any number of characters or any type. This is often used to find files with a certain extension, as `*.txt` will match any file with the extension `.txt`. You can also use multiple wildcards in the same expression; for instance if you want list all files containing the word "data" that ends in ".txt" you could use `ls *data*.txt` and the shell would show you all of the files matching that pattern.

Other useful wildcards include:

- `?`, which can be used to represent a single unknown character. For instance, `?at` matches `Bat`, `bat`, `Cat`, `cat`, and many others, but not `at` alone.
- `[<characters>]` can be used to represent any one of the characters included in the square brackets. As an example, `[CB]at` matches `Bat` and `Cat`, but not `bat` or `cat`. You can also specify a range of characters such as [A-Z] for any capital letter, [a-z] for any lowercase letter, and [0-9] for any numeric character, or [A-Za-z0-9] for any alphanumeric character.
- `^` negates whatever is entered. For example, `[^<characters>]` matches any single character *except* those included in the square brackets. `[^C]at` will match `Bat`, `bat`, `cat`, and many other strings, but not `Cat`.

What if your expression contains one of the special wildcard characters? Like we saw with having a space in the file name, you can enter an "escape character" by preceding the character with a backslash `\`. You will also need to escape wildcards in some other situations, like when using wildcard to denote a file name when using `find` (see below). Thus if you want to match a sequence that includes a question mark, use, for example, `hello\?.txt`.

There are more sophisticated pattern matching techniques called "regular expressions" which we will use more extensively when we talk about AWK. Getting used to using these wildcards in the terminal will make regular expressions a bit easier to understand, so be sure to spend some time experimenting with them.

3.2 Finding Files

To make queries of files in the filesystem, use the `find` command, which has a number of sophisticated options. The syntax for `find` is `find <locations> <criteria>`, where `<locations>` are the directories that you wish to search (including all subdirectories) and `<criteria>` are the specific options that you want to search for (these are specified by command options).

Command options for `find` that I use frequently include:

- `-name <pattern>` finds files whose name matches the given pattern. You are allowed to use wildcards in name patterns, but you must put a backslash in front of any special wildcard characters (otherwise, the shell expands the wildcards *before* searching the files, while we want the wildcards to remain until `find` does its search).
- `-iname <pattern>` same as `-name`, but is case insensitive
- `-type <type>`, target is of a specific type, common examples include `f` regular file, `d` directory
- `-atime <time>`, target has a most recent access date that differs from the present date by exactly `<time>` days, rounded to the next 24 hour period. You can also specify units other than days for `<time>` using any of `smhdw` (seconds, minutes, hours, days, weeks, respectively). For files that are newer than the time specified, use `-<time>`, and for files that are older than the time specified, use `+<time>`
- `-mtime <time>`, works like `-atime` but uses file modification date rather than access date

For example, to find files within my home directory with a name that match the name "test", I would enter `find ~ -name test -type f` into the terminal. To find all directories in my `Documents` directory that were modified in the past 24 hours, use `find ~/Documents -type d -mtime -1` to perform this search. Wildcards can be very useful for finding files of a certain type; to find all SAC files in my home directory I can use `find ~ -type f -iname \*.sac` to find all SAC files. Note that using `-iname` ensures that both files ending in `.SAC` and `.sac` are found, and that I put a backslash in front of the wildcard character to prevent expanding the character prior to invocation of `find`.

Exercise: Practice searching for different types of files. I use this command frequently, so it is a good idea to be comfortable with its use.

3.3 Finding Text Within Files

We can also search for text patterns within files. This is done using `grep` (short for **G**lobally search a **R**egular **E**xpression and **P**rint). The basic syntax is `grep <pattern> <files>` where you specify a single pattern and (potentially) multiple text files in which to look for that pattern. For example, if you have a list of fruits as a text file `fruit.txt`, then `grep apple fruit.txt` will print any lines in the file that contain the string "apple."

As the title alludes to, the pattern specification is a regular expression and can be used to match rather complex patterns. Practice using `grep`, using straight text strings as well as wildcards to make various patterns that appear in a text file. You can also use wildcards on the text files if you want to search a large number of files for the pattern. As previously mentioned, we will come back to regular expressions and will talk a bit more about `grep`.

Exercise: Using a text editor, create a text file that contains several lines of text. Use `grep` to find patterns, in particular to practice using various wildcards to match patterns.

3.4 Permissions

Back in the last lab, we noted that `ls -l` gave us a rather cryptic list of characters at the beginning of each entry:

```
total 472
drwxr-xr-x   53 egdaub  staff    1802 Nov  4  2015 MATLAB
drwxr-xr-x    2 egdaub  staff      68 Aug 10  2015 awk
-rw-r--r--@   1 egdaub  staff  147712 Sep 25  2015 ceri7104_
↪dataanalysis.docx
drwxr-xr-x    3 egdaub  staff     102 Jan  7  2016 compexam
drwxr-xr-x    9 egdaub  staff     306 Dec 17  2015 csh
-rw-r--r--    1 egdaub  staff      84 Aug 25  2015 data_syllabus.aux
-rw-r--r--    1 egdaub  staff    5231 Aug 25  2015 data_syllabus.log
-rw-r--r--@   1 egdaub  staff   52581 Aug 25  2015 data_syllabus.pdf
-rw-r--r--    1 egdaub  staff   15882 Aug 25  2015 data_syllabus.
↪synctex.gz
-rw-r--r--    1 egdaub  staff    4942 Oct 19  2015 data_syllabus.tex
drwxr-xr-x   56 egdaub  staff    1904 Dec 17  2015 gmt
drwxr-xr-x   77 egdaub  staff    2618 Dec  7  2015 homework
drwxr-xr-x  148 egdaub  staff    5032 Nov 24  2015 lectures
drwxr-xr-x   12 egdaub  staff     408 Oct 19  2015 python
drwxr-xr-x   11 egdaub  staff     374 Nov 19  2015 sac
drwxr-xr-x   12 egdaub  staff     408 May  2 12:34 studentwork
```

These characters signify what are known as "permissions." These characters tell us who is allowed to read (`r`), write (`w`), and execute (`x`) this file. There are three sets of letters signifying this that follow the first characters, which tells us whether or not the entry is a directory): the first set of three tells us what the owner is allowed to do (for the above list, the owner is `egdaub`), the second set of three tells us what the group is allowed to do (for the above list, the group is `staff`), and the third set tells us what any other user is allowed to do. Thus, the sequence `-rw-r--r--` signifies that the user can read and write the file, the group can read the file, and others can read the file. These are the default permissions for a newly created file. Note that for the directories, the listing is `drwxr-xr-x`. The leading `d` says that this is a directory, and that all can read and execute, but only the owner can write. Execute is a bit of a misnomer here because this is a directory (you cannot really "execute" a directory, it just means that said class of users can `cd` into that directory and search in that directory).

These permissions can be changed using the `chmod` command (Change Mode). The syntax is `chmod <mode> <files>` and there are several different ways to specify the mode. You can add, remove, or set specific permissions for specific groups using the letters `u`, `g`, `o`, and `a` (for user, group, others, and all respectively), the operators +, −, and = (add permission, remove permission, and set permission, respectively), and `r`, `w`, and `x` (read, write, and excecute, respectively). Thus, to add execute privileges for the user, enter `chmod u+x file.txt` into the terminal. To remove read and write privileges for others, enter `chmod o-rw file.txt` into the terminal. To give everyone read, write, and execute access, enter `chmod a=rwx file.txt` into the terminal. You can also specify more than one file in the command, and the specified changes will be made to all the files given in the command.

A shorthand way to specify permissions is to use an octal number to specify the binary representation of read, write, and execute as follows:

Permissions	`---`	`--x`	`-w-`	`-wx`	`r--`	`r-x`	`rw-`	`rwx`
Binary	000	001	010	011	100	101	110	111
Octal	0	1	2	3	4	5	6	7

To set owner permissions to `rwx`, group to `r-x`, and others to `r--`, you can enter `chmod 754 file.txt` (the first octal number is for the owner, the second is for the group, and the third is for others). This is shorter than specifying each one individually through multiple commands, so if you need to change many different permissions at once, this is a handy trick.

3.5 Aliases

You may have noticed that some of your Unix commands can get a bit long and difficult to type if you choose a number of options. If you have a command that is long and is one you use frequently, you may consider making an alias. An alias is a way to essentially define your own Unix command using some other command. For instance, if you want to always remove files using `rm -i`, then you could make an alias to do this. Enter `alias rm 'rm -i'` and then a carriage return. This sets the command `rm` to execute `rm -i` automatically. Try this, and see that it will ask you to confirm when deleting a file. To undo an alias, precede the command with a backslash (`\rm`). To remove an alias, use the `unalias` command (`unalias rm` to remove the example here). `unalias -a` will undo all aliases.

3.6 Other Shells

One thing to note here: the syntax for an alias in the shell that you are currently using (the Tenex C shell `tcsh`) is different from some other Unix shells. Each of these different shells represent a terminal with different available commands and shortcuts. While they are all relatively similar, they are not exactly the same. Other shells available in the Mac Lab include the Bourne-again shell `bash` (default on most GNU/Linux systems), the Korn shell `ksh`, the Tenex C Shell, and the Z Shell `zsh`. While the majority of the commands that we will use work in all shells, there are some specific ones like this that are specific to a certain type of shell.

To run a different shell, enter the appropriate command into your present terminal. It will start a new session within that shell (the current directory will remain the same). To exit, type `exit` and you will return to your first shell. You can set your default shell in Terminal to be whatever you like – go to Terminal > Preferences... and set the "Shells open with" command to whatever shell you want. You need to give the full path, and all of the shells are under `/bin`.

3.7 rc files

If you want to set up an alias for all of your terminal sessions, is there a way to avoid typing in the alias every time you start up a terminal? You can automatically set things like aliases using a startup file. For the default `tcsh` in the Mac lab, the startup file is a file in your home directory `~/.tcshrc` and is executed automatically every time you start up a `tcsh` session. Note that the file begins with a period – this means that this is a file that is not normally visible when you type `ls` in your home directory. To see files that begin with a period, use the `ls -a` option. When you `ls -a` in your home directory, if you still do not see the file, then it does not exist. You can create this file in a text editor like `nano` – enter `nano .tcshrc` when in your home directory, and then type in any commands you would like to be executed by default on startup. Common things to put in a startup file include setting aliases, changing the terminal prompt appearance, and setting variables known as environment variables (more on that later in the semester). Try putting an alias in your startup file, restarting the terminal, and seeing that the alias is in effect. The other shells have similar startup files, such as `~/.bashrc`, though some have more than one possible startup file and preferential execution of one versus the other, depending on whether the shell is the login shell or was launched from another shell. If you want to keep using `tcsh`, then do not worry about login versus non-login shells – regardless of how your shell is launched, it will always execute `~/.tcshrc` file when it starts.

3.8 Making Programs Interact

3.8.1 Standard Input/Output and Pipes

One thing you may be wondering now is if the Unix philosophy is to make simple, robust programs that do one thing and do it well, how do we use multiple programs to do something complex? This requires that we introduce the concept of “pipes,” “standard input,” and “standard output.”

As you may have noticed, all of the commands you have typed typically print out some result to the screen. This could be a list of files using `ls` or `find`, the contents of a text file using `cat`, or a list of lines matching a pattern using `grep`. This output is referred to as “standard output” and every Unix command that produces output prints it to the screen in roughly the same way.

Programs can also receive input via what is called “standard input,” which can come from the keyboard, from a file, or from the standard output of another program. This interaction where the standard output of one program becomes the standard input of another is referred to as “piping” and is the principal way that we can do more complicated Unix tasks using the basic commands.

As a very simple example, let’s say that we have a directory containing a large number of files and we want to be able to look at the long format list (`ls -l`) of the contents. However, we want to read this in the terminal with the `less` program, rather than having to manually scroll up and down. This can be accomplished using a “pipe:” enter `ls -l | less` into the terminal, and you should have the list contents open using `less`. The verical line `|` is what is known as a “pipe,” and its meaning here is to tell the terminal to use the standard output of the command `ls -l` as the standard input to the `less` command. We can then read the output from `ls -l` in `less` like it is any old text file.

While this may seem really simple (and you would be correct) we can do more complicated things. We can search within the output of some program using grep via a pipe – to find out which files in a directory some user can read, write, and execute, enter `ls -l | grep rwx` and the terminal will print the lines that contain the string “rwx.” Using `grep` in a pipe to find patterns in the output of some other command is a very common use of pipes, and you are likely to come across many examples where this is useful.

3.8.2 Input and Output Redirection

There are other ways to deal with command output, as you can save results of terminal operations as files. Saving output to files (as well as using files as standard input) is done with the operators <, >, >>, and the command `tee`. < signifies that the file following the < operator is to be used as input to the command. Many of the commands that we have already seen use files as standard input without needing to use < (for example, try entering `cat testfile.txt` and `cat < testfile.txt`; the output should be identical), but you will likely encounter cases where you need to specify the use of a file as input.

The operators >, >>, and the command `tee` can all be used to save command output to a file. >

saves the command line output in a new file, overwriting the old file in the event that file exists. Try entering `ls > temp.txt` into the terminal; the result should create a file "temp.txt" containing the result of the `ls` command. `>>` will append the results of a command to the file. Check the contents of "temp.txt," and then enter `ls -l >> temp.txt`. Use `less` to view "temp.txt" to verify that it contains the results of both the short and the long list command.

Sometimes if you are piping the result of one command to another, you have some intermediate output that you might want to save for another purpose. To save the standard output from a command while simultaneously sending the same output to standard output, the `tee` command can be used. Basically, `tee <filename>` saves its standard input to file, but also sends its standard input to standard output. Try using `tee` by entering `ls -l | tee temp.txt`. The `ls -l` results should both print to the terminal and save to the file "temp.txt."

3.8.3 `xargs`

One final trick for making programs interact is the following: what if you have some program that takes a number of individual file names as input, rather than standard input (i.e. a text stream)? For example, in the piping example above, we piped the output of `ls` into `grep` to search the output for a certain pattern. However, what if we didn't want to search the output for that pattern, but rather the contents of those particular files for a pattern? We cannot do that using a pipe. Similarly, if we want to find a bunch of files, and then copy them to some other directory, we cannot use standard output in the `cp` command, since `cp` takes a file and copies it to another directory.

You can perform the above operations by making commands interact with the `xargs` command. `xargs` takes standard input and reformats it as a list of arguments, then repeatedly executes the command that follows using each of the elements of the list as an argument. By default, `xargs` puts the argument at the end of the command, so if you have a command where the inputs need to be placed at a different point in the command, you should specify where `xargs` should put the arguments using the `-J` option and some other character as a placeholder (see an example of how to do this below).

As an example, let's say I want to search every shell script (files with a suffix ".sh") in my home directory for the pattern "gmt" to figure out which of my shell scripts invoke GMT commands. This is one of those cases where I need to use `xargs` to turn standard output into a list of arguments. I can do this with `find ~ -name \*.sh | xargs grep gmt` as a terminal entry. The first command is `find` to find all shell scripts (files ending in ".sh") in my home directory. The `find` command by itself sends a list of files to standard output. Since `grep` needs a list of arguments, rather than standard output, `xargs` is used to transform standard output into a list of arguments. What is actually happening is that `xargs` tells the terminal to execute the command `grep gmt <file>` repeatedly, where `<file>` is replaced on each successive execution with a line from the output from the `find` command. Note that in this case, because `xargs` puts the argument at the end of the command by default, and `grep` takes its target file as its last argument, we could use `xargs` without needing to specify any options.

As another example, here is a situation where we cannot use `xargs` without any options. Let's say that we want to copy every file that we modified in the past 24 hours to a single folder as a backup copy. Since `cp` takes a filename as an argument, rather than standard input, we need to use `xargs` to achieve this. Also note that the syntax for `cp` is `cp <file> <target>`, so the default usage of `xargs` will not work because we need to insert our filenames in the *middle* of the command. To successfully perform this operation, we use the `-J` option for `xargs`, where we designate a special character to serve as a placeholder, and then `xargs` replaces the placeholder with the argument derived from the input in each execution of the command. Assuming that the directory `~/backup` already exists, then we can copy all recently modified files to this directory by entering `find ~ -type f -mtime -1 | xargs -J % cp % ~/backup` into the terminal. The first part of the command is the `find` operation, which sends the files satisfying the critera to standard output. This is then piped into `xargs`, which takes the list of files and inserts each part of the list into the appropriate place in the `cp` command (using `%` as the placeholder, which is specified by the `-J` option). Note that other versions of Unix may have a different way of doing this command substitution, so check the manual page if you are on a system other than the Mac OS.

Do not worry if you do not totally understand how to use pipes, redirections of output, and `xargs`, and everything else here. Unix takes time to learn how to make it do what you want. Over the semester, as we work with Unix more and start using Shell Scripts, AWK, and GMT, you will gradually become more comfortable with this. However, nothing can take the place of practice: if you want to truly become comfortable with Unix, then you should use these tools on a daily basis in your research.

3.9 Summary

Here are a list of the commands that we introduced in this lab. They are among the most common, and you will likely find yourself using them very often:

- Wildcards `*`, `\`, `?`, `^` and `[ ]`
- `find`
- `grep`
- `chmod`
- `alias`
- `unalias`
- `tcsh` and many other shell types
- > (redirect to file)
- < (redirect from file)
- `|` (pipe to next command)
- `tee` (send to file and standard output)

- `xargs`

CHAPTER

FOUR

BASIC PROGRAMMING WITH PYTHON 1

Now that we have a sense of how to use the Unix shell, we will examine some programming basics. Today we will be mostly getting acquainted with Python, which is a common programming language used in many scientific fields. Python has several nice features that make it useful for beginning programmers, and the standard implementation of Python is open source and free. You are of course free to install Python on your own computer if you want to practice programming away from the computer lab. Python is my default programming language for general scientific computing. I use it almost as often as a Unix shell, as there are extensive tools for scientific number crunching, analysis, and plotting. As we will see, MATLAB is another option for many of these types of tasks, but I find the Python programming language to be more powerful and flexible and thus I have found that for me, Python has almost completely replaced MATLAB. I do more serious computing in a compiled language like C or Fortran, but if the task is simple and does not need to be optimized for performance, Python is typically my choice.

This lab session will be interactive, as I will demonstrate everything myself on the computer. You should type along with me when I am demonstrating something, and then I will give you a chance to practice on some exercises.

4.1 Python

Python is a high level programming language. "High Level" refers to the fact that the language comes with many built-in features that might not be present in other languages (and you would therefore need to write these yourself). The higher the level of a language, the shorter programs tend to be. This is generally a good thing – it is better to have a group of coordinated experts implement a feature to get it working correctly than to have every single user write their own version. It also saves you time, in that programs are faster to write and debug.

One reason Python is often used for introductory programming is that Python programs tend to be rather literal. They often read fairly clearly in English, making them easier to understand. This is not always the case, as we will see as the semester progresses (just wait until you are trying to wade through an incomprehensible GMT shell script, and you will appreciate many aspects of Python).

A second feature of Python is that it uses indentation to denote syntax. This means that Python takes the layout and spacing of a program into consideration when interpreting its meaning. The alternative is to use specific characters to denote the meaning. While one method is not better than any other, it is generally agreed that the type of indentation required by Python is good programming practice, as it makes programs more readable and easier for humans to interpret. For this reason, I consider the requirements imposed by Python a good thing for beginners, as it forces them to develop good programming habits. I will demonstrate how this works as we go today.

We will be using Python 2 in this class. It is important to note that code written for Python 2 and Python 3 are not interchangeable – there are some important issues with the language that have been cleaned up in Python 3. If you want to install Python on your own computer and use examples from this class, make sure you install Python 2 (or understand how you need to change your code for Python 3).

4.2 Interpreted vs. Compiled Languages

One issue with programming is the fact that the processor on your computer cannot directly interpret a program. Computer processors can only do a limited number of operations (think add, subtract, multiply, divide and logical operations), so every operation must be translated into a series of instructions of this type. Fortunately, we have computer programs that are able to do this for us. There are two types: interpreters and compilers.

- **Interpreters** turn a program into instructions one line at a time. The interpreter reads a line of code, checks if it is valid code, creates a set of instructions, and then executes the instructions. It then proceeds to the next line until the end of the code to be executed is reached.
- **Compilers** read the entire program and turn the whole thing into executable instructions, saving the instructions as an executable file. The instructions are not executed until the file containing the instructions is run.

These two models have pros and cons. Interpreters are easier to use in general, because the interpreter has direct access to the part of the code is currently being executed. If it encounters a problem, it can tell you exactly where it ran into that problem in your code. However, because it is constantly creating instructions and executing them on the fly, interpreters cannot perform optimizations as readily as compilers, so code performance can suffer as a result. Compilers tend to create code that runs faster, but if there is an error in your code, it can be much more difficult to track down since it is not known which part of your code produced the instructions that could not be executed.

All of the languages that we will use in this class are interpreted. This makes them easier to use, but not ideal if performance is necessary. However, computers these days tend to be fast enough that you are unlikely to need a compiled language unless you are doing some serious number crunching.

4.3 Interactive Sessions and Scripts

To start up the Python interpreter, open a Unix shell and type `python` and a carriage return. You will see some information about the version of Python and then a prompt `>>>`. This is an interactive session with the Python interpreter. You can enter one set of instructions at a time, and the interpreter will execute them as they are typed, printing out the results to the shell. This is a nice way to use Python for simple work, as you can check your work as you go along by directly seeing the output results. To exit, type `exit()` at the prompt.

For more complicated work, it is convenient to save all of the code to be executed into a single file, say `myscript.py` and then run the entire file through the Python interpreter. This is called running a script, in that the file is a set of commands to be executed (much like a script is a set of lines to be read in a performance), and can be done by typing `python myscript.py` followed by a carriage return into the Unix shell. This will run the entire script, with any output written to the shell.

Initially, you will probably do your programming interactively in Python until you get the hang of it. After that, as your programs get more complex, you will want to write scripts using a text editor (hopefully you will use this as a chance to practice with a command line editor) and then run them directly from the shell.

4.4 Basics

Start up the Python interpreter. I will provide some demonstrations of the following on the screen. You are encouraged to type along with me – playing around with an interpreter is usually the best way to learn how to do things. I will show you the following:

- **Basic data types:** integers, strings, and floating point numbers make up the most elementary objects in almost any programming language, including Python. **Integers** are signified by simply entering the integer into the interpreter: `5`, `-1` and `10000` are valid integers in Python. **Strings** are any set of characters enclosed by either single `'` or double quotes (`"`). If you want to include quotes in the string, then either use the opposite type of quotation marks to enclose the string, or put a backslash (\) before the quote inside of the string. Finally, you can denote a **floating point number** (i.e. a number with a decimal part) by typing a number with a trailing decimal point, such as `100.` or `0.001`, or writing the number in scientific notation like `1.e-3`.

  You can change between these data types in Python. To change to an integer, use `int( )`; to change to a float, use `float( )`; and to change to a string, use `str( )`. Some transformations are not possible: for instance, if you have a string that contains non-numeric characters, you will not be able to change it into a float or an integer. If you change a float into an integer, Python will drop the decimal part of the number. If you are intending to write a program to work with a certain data type, it is a good idea to convert any program input into the correct datatype to prevent errors further down the line.

- **Basic operations:** Python can do basic operations on these primitive data items. For instance, to add two integers, you can enter `2 + 3` and the interpreter gives you back the result. Built-in operations on integers include addition (+), subtraction (−), multiplication (`*`), division (`/`), exponentiation (`**`), and the modulus operation (the remainder after dividing the first number by the second, denoted by `%`). Note that when dividing two integers, you get an integer back!

  Operations follow the standard order of operations. To alter the order in which operations are carried out, use parentheses to group operations: `2 * 3 + 4` is not the same as `2 * (3 + 4)`.

  These same operations, except for modulo, can be applied to floating point numbers. The only difference is that you always get a floating point number back, rather than an integer. If you combine integers and floats, Python will automatically change the integer into a float and perform the floating point version of the operation.

  For strings, operations include addition (which concatenates two strings), and multiplication (which repeats the string a given number of times). Note that if you try to do something nonsensical like add a string and an integer, Python gives you an error message.

- **Print Statements:** To print something out, use the `print` statement. Often, beginning programming books and tutorials start with a program known as "Hello world!" In Python, this program is very succinct, taking up only a single line: `print 'Hello world!'`. You can also print the results of more complicated operations. Print statements are useful for debugging, as they let you see intermediate results on the screen, allowing you to check if they are correct.

- **Assigning variables:** While you can do basic calculations with Python, a programmable computer isn't particularly useful unless you can store intermediate results to be re-used later. Otherwise, you would have to repeatedly type in complex expressions, where you are likely to make a mistake, which defeats the purpose of having a computer do the calculations. Assignment is done through the assignment operator =, which binds a name on the left hand side to a value on the right hand side.

  Variable names can be anything you like, within a set of rules: (1) variable names must start with a non-numeric character (numbers are allowed, but not in the first position), (2) they cannot contain any special protected characters (like =, +, `.`, etc.) that have other meanings in Python, (3) they cannot be certain protected keywords (we have not encountered any yet, but we will see examples today such as `if` and `for`).

  Assignment is very handy, as it lets you (1) re-use previous results without repeating the calculation (or having to type it in again!), (2) break up complex statements that are difficult to type into multiple pieces, and (3) choose informative variable names can help make your code easier to read by a human (such as me when I am grading your homework, or most importantly yourself in 2 weeks when you are re-visiting a piece of code that you completely forgot about in the meantime).

- **Input:** To enter data from the keyboard, we can use `raw_input( )`. If you provide a string to `raw_input`, that string will be displayed on the screen when asking for input. So for example, I can use the following code to enter a string into a Python program:

```
a = raw_input('Enter a string: ')
```

The string entered will then be stored in variable `a`. Raw input is always a string; if you need to enter a numeric value, use `int(a)` or `float(a)` to convert the string to the appropriate type.

- **Comments:** All programming languages have characters that tell the program to ignore whatever is written afterwards. This lets you add documentation that explains why you are doing what you are doing (again, something that is very handy to you in 2 months when you re-visit some code you wrote and forgot about), explain details about how something works, and things like that. Documenting code is a *very* important habit to develop. I will mark you down on homework assignments if you turn in code that isn't documented sufficiently. Comments can be either full line comments, or occur inline after other Python code.

  Comments in Python are designated using the pound character (`#`). In general, it is a good idea to document *how* and *why* you are doing what you are doing, rather than *what* you are doing. Take the following comment examples to see which is more useful:

  ```
  # npoints is one less than a
  npoints = a - 1
  ```

  This is not particularly helpful, as it is obvious from the code itself what you are doing in this case. A more useful thing to comment is explain *why* you are doing something:

  ```
  # decrease a by one to get npoints, as first is double counted
  npoints = a - 1
  ```

  This is much more helpful – not only do we know that `npoints` isn't the same as `a`, but *why* they are not the same. If we need to modify the code, comments explaining why something was done would be more informative for us.

All of the above are basic features of any programming language, so these are not particular to Python (and we will see them again several times during the semester).

4.5 Conditional Statements (if)

Above, we saw a bit about the fundamental building blocks of any programming language. However, these pieces alone aren't enough to make computers particularly useful. This is because the Python interpreter executes every line we enter, one at a time. But what if some future calculation depends on the result of a previous one? We would need to do the intermediate calculation, then decide for ourselves what action to take, and only then could we enter the remaining set of instructions.

We can avoid this by using an `if` statement. The basic structure of an if statement is as follows:

```
if condition1:
    statement1
elif condition2:
    statement2
...
else:
    statement
```

Depending on the various conditions, this program will execute different statements. If boolean `condition1` is true, then the program executes `statement1`. If `condition1` is false, then the program checks if `statement2` is true; if so it executes `statement2`. You can add as many `elif` blocks to the program as you like, and the interpreter will continue checking them one at a time until it encounters one that is true. If none of the preceding statements are true, then when the `else` part is reached, `statement` will be executed.

In an if statement, the `elif` and `else` blocks are optional. You do not need an `else` block at the end, nor do you need any `elif` blocks in the middle. You can therefore tailor the if statement to do whatever specific steps you need for any type of condition.

The `condition` parts are what are known as **boolean** datatypes: they can only have two values, either `True` or `False`. This is another type of basic item that is found in all programming languages, and you can usually also use `1` for true and `0` for false. One way to get a boolean is to make a comparison. For instance, if you have an integer variable `a`, you could see if that variable is equal to 1 with the statement `a == 1` (note that comparisons in Python use the double equals sign, while assignment uses a single equals sign; this is true for most programming languages). Other comparisons include: > (greater than), < (less than), >= (greater than or equal to), >= (less than or equal to), and != or <> (not equal to, both are equivalent).

Booleans can be combined to form booleans using the operations `and` and `or`. `a == b and c == d` is true only if a and b are equal *and* c and d are equal. If either of the two comparisons are false, then the whole thing is false. `or` only requires that one of the conditions be true. Multiple booleans can be combined with parenthesis (if needed to specify precedence). If you want the opposite of a given comparison operation, use `not` before the comparison. So for instance, `not 1 == 2` is `True`, while `not 2 > 1` is `False`.

One thing you should have noticed above is that all of my statements were indented from the left by the same amount. In general, it is a good idea to indent blocks of code that belong together like this – it makes it clear to the reader what code is executed when each of the various conditions are true. You should always indent your code consistently in this class when turning in assignments, and if I find it hard to read, I will take off points.

However, most programming languages do not *require* you to indent your code this way. Some other characters indicate which statements belong to the various code blocks. Python, on the other hand, *requires* that you indent your code consistently and actually uses the spacing to interpret the meaning of your program. If you try to run a Python program that is not consistently indented, you will either (a) get an error message back, or worse (b) your program will run but not give you the right answer. Some people find this aspect of Python to be annoying, but to me this is a very positive feature about Python (particularly for beginners as it forces you to develop good habits).

Spend some time experimenting with if statements. Try some with just an `if`, others with `if` and `else`, and others yet with `elif` blocks. Print statements are good things to include in the various blocks, as they let you know visually that your code ran as you intended.

4.6 Loop Statements (while, for)

While conditional statements help us to automatically treat different cases in different ways within a program, the computer is not a powerful tool for automation unless we are able to do things repeatedly. Repetition requires looping capabilities. Without a loop, the only way to write a computer program that does more computations is to make the program longer, which requires more typing by you (and thus, more opportunities for you to make a mistake). Loops, on the other hand, allow you to execute something many times. This is what makes a computer a powerful tool for analyzing data: we can do the same thing over and over again on different data without doing extra work.

Programming languages have two basic types of loops (though some have additional variations on them): `while` loops, and `for` loops. Let's examine each one separately:

While Loops: These loops allow you to run a block of code repeatedly as long as a certain condition is true. This is best illustrated with an example:

```
a = 10
while a > 0:
    print a
    a = a - 1
```

When executed, this code will print out the integers from 10 to 1, counting down one at a time. (Try this for yourself.) How does this work?

First, we initialize the variable `a` to be 10. Then we enter the loop, which executes as long as `a` is greater than zero. At the start of the first loop, the condition is true, so the loop executes, printing 10 to the screen, and then setting the value of `a` to be `a -1`. Note that this equation does not really make sense mathematically – how can `a = a -1`? However, recall that = is not an equality in Python; it is an assignment operator. Here, we first calculate the value `a -1` and assign it to the variable `a` afterwards, overwriting the old value of `a` in the process.

From there, we have completed the loop, so the cycle starts over again. Python checks if `a > 0`, and since it is, it executes the loop again. This continues until we reach the case where `a` is zero. In that case, the condition is no longer true, so Python does not execute the loop, and we reach the end of our mini program.

There are a few things to be aware of here. First, note that like with the `if` statement, I have indented all of the code in the loop. This is something you should do in any program you write, even though it may not be required, as it gives a visual cue as to what code is executed each time through the loop. In Python such indentation is necessary. Try running the program again, but change the indentation: you will either get an error message, or the program may run forever, because there isn't appropriate spacing in the loop body.

Second, and most importantly, is that I made sure to write this loop so that `a` decreased every time through the loop. Otherwise, we could not guarantee that the loop would ever finish (something known as an infinite loop). Try running your program without the `a = a -1` line: you will see `10` printed out to the screen in perpetuity. This is an important thing to keep in mind when writing a program, as if you have an infinite loop in your code, it will never run to completion.

One thing to note about `while` loops is that we may not know easily in advance how many times we need to go through the loop. In this case, it was fairly obvious how many times it would run, but in other cases, `a` might decrease by a different amount each time through the loop, or it may not decrease every time. Because of this, most programming languages have another kind of loop that executes a fixed number of times, reserving while loops for the case where loops are executed an unknown number of times.

For Loops: To execute a loop a fixed number of times, use a `for` loop. Here is an equivalent program to the one above, using `for` instead of `while`:

```
for i in range(10):
    print 10 - i
```

To use a `for` loop in Python, you need to tell it what specific values to iterate over. One simple way to do this is to use `range`. By default, if you type `range(10)`, then it will repeat your loop for values ranging from 0 to 9, with increments of 1. If you want to start at a different value other than 0, then `range(1,11)` will start at 1 and end with 10 (note the last value is one less than the final number). To get regularly spaced numbers, use `range(<start>,<stop>,<step>)`. So to get all even numbers from 0 to 20 (inclusive), you could use `range(0,21,2)`. Within the loop, `i` then has whatever value is appropriate for that iteration through the loop.

You might notice that an explicit `for` loop is not really necessary for doing simple loops like this, as you can just as easily write this as a `while` loop, and `for` loops are really only a convenience. This is mostly true, though in Python in particular there are more complex data types where `for` is much more sensible for iterating through a bunch of values, as opposed to `while`. In general, I find that I use `for` loops much more frequently than `while` loops, as most of the cases where I need a loop involves a specified number of repetitions.

Sometimes, you may want to stop running a loop in the middle of a series of iterations. `while` loops provide one way to do this, but you can also change the behavior of either `for` or `while` loops with the `break` and `continue` statements. A `break` statement immediately terminates execution of a loop, while a `continue` statement proceeds immediately to the next loop iteration, skipping over any remaining statements. For instance, a version of our original while loop could alternatively be written as:

```
a = 10
while True:
    if a == 0:
        break
    print a
    a = a - 1
```

This would produce the same output as the original program (try it!). If we wanted to run the same skip over 5 for some reason, we could use a conditional and a `continue` statement as follows:

```
a = 11
while a > 1:
    a = a - 1
    if a == 5:
        continue
    print a
```

Looping constructs are central to efficient programming, as they are what make using a computer to repeat tasks so efficient. Any time you are doing something over and over, you should get into the habit of figuring out how to turn it into a loop – as you will see, pretty much every type of software we deal with in this class has a way to loop over items (be it individual numbers, individual lines, individual files, etc.).

4.7 Practice

Here are some problems to practice programming in Python. Work on writing code either directly in the interpreter, or as an external script.

- Write a program that calculates the average of three floating point numbers, printing the result to the screen.
- Write a program that prints out the square of each integer in a sequence of your choosing (use `range`).
- Modify your previous program to exclude any numbers whose square is between 10 and 50.
- Write a program to print out all integers that are perfect squares less than or equal to 100.
- Write a program to print all positive integers less than 300 that are divisible by 3 but not divisible by 2.
- Write a program to test if a positive integer is prime. (*Reminder:* An integer is prime if it cannot be divided evenly by all smaller positive integers beginning with 2.)

CHAPTER

FIVE

BASIC PROGRAMMING WITH PYTHON 2

Today we continue developing basic programming skills. We will focus on two very powerful concepts that you will see frequently in all programming languages: functions/procedures, and data types that are collections of primitive data types.

5.1 Functions and Procedures

Our programming so far has involved using loops and conditionals to manipulate data. While this works well, if we have a long and complicated program, the list of operations to perform can become rather long. This makes it hard for a human to read and comprehend all that is going on, particularly if there are many different variables that are floating around and being used. Instead, it would be better to break the program up into a series of sub-pieces, each of which is self-contained and can be dealt with separately. This increases comprehension of code, and is generally considered good programming practice.

Further, we may often have a set of operations that we commonly perform. It would be rather tedious to repeatedly write the same code (or to copy and paste it), and in doing so you are likely to make a mistake. Additionally, if you realize that you made a mistake, or find a better way to carry out a calculation, you would need to make changes in *every* occurrence of this set of code. A better practice would be to have a single set of code to carry out such operations. This saves you work, makes the code easier to read, and changing the code is simpler because you are certain that there is only one place where changes need to be made.

Programming languages make this possible by being able to define *functions* (sometimes referred to as *procedures* or *subroutines* if they are not truly functions). I will use function as a broad term covering all of these things here (as Python has a single syntax covering all of them), but some programming languages use a different syntax for functions versus procedures.

5.1.1 A Simple Example

We can define a function in Python using the `def` keyword. Here is a simple function that adds two numbers together:

```python
def add(a, b):
    'adds two numbers a and b together, returning the sum'
    return a + b
```

There are several things going on here:

- The first line tells Python that we are defining a function because of the `def` keyword. Following that, we have the name of the function (`add` in this case), and then the input arguments to the function, in this case `a` and `b`. The function name is subject to the same rules as variable names: it must start with a non-numeric character, and cannot contain protected characters or be any protected keywords.
- The next line contains a string. This is called a docstring, and it a short piece of documentation that describes the function. It is a good idea to create docstrings for all of your functions. The docstring should describe what the function does, what its inputs are, and what its outputs are. Here, a short string is sufficient to mention that `a` and `b` are numbers, and that the function gives you back their sum. However, if you want to provide a detailed, multi-line string, you can do so if you enclose the entire string in triple quotes (`''' ... '''`).
- Finally, we have the body of the function. Note that it is indented to indicate what set of commands belong in the body of the function. This function is just one line, which adds the two numbers and then uses the `return` statement to indicate that this is what you get back when you call the function.

To call the function, we type `add(3,4)` into the interpreter or our script, which will either print 7 on the interpreter output, or we can embed that into code just like any other integer. So if we wanted to use `add` to sum many different integers, I can enter `add( add(3,4),add(6,8) )` to get 21 back. Python calls `add` three different times here: once with 3 and 4, once with 6 and 8, and then a third time with 7 and 14 (the results from each previous call).

While this is a very simple example, if you needed to do a more complicated set of operations, this will allow you to define your function just in one place, and then concisely call it many different times. If you need to change the function, you only need to do so once. This is a very powerful concept: it allows for modularity in your code. You can check that small functions work independently from one another, making it easier to debug a large and complicated program. It also makes reading a program easier, as you can keep each piece of code shorter with several function calls replacing large sections of code. Functions are probably one of the most important concepts in software design, and you should always strive to think about breaking your problem into smaller pieces that can be written as functions.

5.1.2 Scope

What happens when you call a function? Python creates what is known as an *environment* that is separate from the rest of your program. All variables given to the function as arguments are available in this environment, as are any additional variables that you may define within the environment. You might have variables defined in your main program that have the same name as variables in your function, and the fact that Python creates a new environment means that any changes to variables that occur within a function cannot be seen by the main program, as they are local to the function. This is what programmers call **scope**: it tells you about where variables can be seen by other things. For example, the following program will not modify the value of a in the main program:

```
a = 1

def myfunc():
    a = 2

myfunc() # sets a = 2 in the other environment, but changes nothing in␣
↪the main program
print a  # will still print 1
```

When a is printed, it will still have the original value of 1, rather than the local value of a during the execution of myfunc.

However, Python does let you use variables defined in other parts of the program in a function, with a few specific rules. If you access the value of a variable of a function, Python first checks the environment created for the function for that variable name. If it finds one, it uses that, but if it does not find one, it looks to the environment that called the function (called the *parent* environment) and checks for that variable. If it finds one in a higher level environment, it uses that value. If it does not find one, it keeps looking to the parent environment of the parent environment, until it finds a value or runs out of environments to check. However, it is generally better programming practice to just use local variables when writing a function, as the intent is not always clear when you use variables that are not local to a function, *except* for global constants that should only be defined once (for example, you do not want to have to redefine π every time to write a function).

5.1.3 Default Arguments

Python lets you set default values for the arguments in a function. This is done as follows:

```
def exponent(x, y = 2.):
    'raises x to the power y, returning the result. default behavior␣
↪is to square x'
    return x**y
```

We can call exponent either as exponent(4.,3.) to cube 4, or exponent(4.) to automatically use 2 for the exponent. Any number of arguments can have default values; the only restriction is that the required arguments (i.e. parameters with no defaults) must all be listed before any optional arguments (i.e. parameters with defaults) when defining the function. You can specify parameters in any order that you like when calling a function if you give the parameter name when calling the function: exponent(4.), exponent(4.,2.), exponent(x = 4.), exponent(x = 4.,y = 2.), and exponent(y = 2.,x = 4.) are all valid and equivalent in Python.

5.1.4 Functions are Objects

One convenient feature of Python is the fact that functions can be treated like any other variable. For example, you can set a variable equal to a function, which lets you use that other variable to call the function. For example:

```
def square(x):
    'returns the square of input number x'
    return x*x
f = square   # note that I do not use ( ) to call the function, this␣
↪simply binds the name f to the function square
print f(4)   # prints 16
```

Function names are just like any other variable in Python, and so you can assign a function to any variable name that you like. You can also pass a function to a function just like any other variable: the function that is passed as a parameter will be assigned to the designated value within the new environment created for the function. Here is an example:

```
def square(x):
    'returns the square of input number x'
    return x*x

def doTwice(x, f):
    'applies a function f taking a single argument to input x twice'
    return f(f(x))

print doTwice(2, square)   # prints 16 = (2**2)**2
```

This is a very powerful tool, as you can create functions where the functions that are called do not need to be known when defining the function. Many other programming languages allow for this in addition to Python, so it is a handy trick to be aware of.

As you can see, functions are extremely powerful. Whenever you are writing code, you should get in the habit of thinking about how to break down what you want to do into smaller chunks that can be easily written as a function. Then, building up from the most granular level, write the base functions, then the functions that tie those functions together, and so on until you have your entire code written. I will try to suggest good ways to break up homework problems this way so you can get used to this mode of thinking.

5.2 Lists

In science, we frequently have to deal with data that is a collection of numbers, rather than just a single number. For instance, a seismogram contains values at many different times, as well as north/south, east/west, and vertical components. GPS data also has multiple components. Datasets like InSAR are collected over an extensive spatial area. It would not be very convenient to use computers to analyze these types of data if we needed to give a distinct variable name to every single number, as that obscures the underlying structure of how the data was collected. Rather, we would like to aggregate appropriate collections of values into single entity that we can deal with more concisely in our programs.

One tool for handling collections of data in Python is using the built-in **list** datatype. A list is just a collection of items: the items in a list can be anything: integers, floats, strings, booleans, or even other lists. For example, here is how you create a list of integers in Python:

```
mylist = [ 1, 2, 3, 4, 5 ]
```

One nice thing about lists in Python is that not all list items have to be of the same type. You can combine different types in the same list, though most of our uses of lists will involve situations where the items have the same type.

5.2.1 Indexing

We can access the entire list using the variable name `mylist`, or we can examine the individual items in the list using the indexing operator `[ ]`. To get the first item in this list, we would enter `mylist[0]` (try doing this in the interpreter; it should print out `1`). Note that the convention in Python is to use 0 to indicate the first position – programming languages that work in this way are called "zero-indexed," and the index in such a language indicates an offset from the beginning of the list (hence 0 means zero offset from the first position, 1 is one place away from the first position, etc.). Some other languages (most notably MATLAB, as we will see in a few classes) use 1 to indicate the first position, and such languages are called "one-indexed."

Python also lets you enter indices in other ways. You can get a subset of a list by specifying a range of indices. For example, to get a sublist containing positions 1 through 3 from our list above, we can enter `mylist[1:4]`. Note that the length of the sublist is the difference between the two indices: even though we entered 4 as the second number, the final entry in our sublist is at position 3. If you want to skip over every other entry, enter `mylist[1:4:2]`, which means "position 1 through 4, taking every 2 positions." You can omit the starting and/or ending index if you want to start at the beginning of the list and/or end at the end of the list: `mylist[::2]` gives every other position in the entire list, `mylist[1:]` gives a sublist that starts at the second position through the end of the list, `mylist[:4]` gives the first 4 items in the list, and `mylist[:]` gives the whole list.

You can also use negative numbers for the indices. Since the index refers to offsets in Python, a negative number just means that the offset it taken in the backwards direction: `mylist[-1]` is the last item in the list, `mylist[-2]` is the second from the end, etc. You can also take negative indices to be the indices for a sublist: `mylist[-3:-1]` gives you the final 2 places, and `mylist[::-1]` reverses the order of the entire list. Negative indices are useful if you want the last element in a list, but do not necessarily know how many total items are in the list (though you can easily get the length of a list using `len( )`).

What if you need to represent 2-dimensional data? You can do so with a list of other lists. For example, one way to represent a 3x3 matrix is as a list of length 3, where each list entry is a list of length three. We can define such a matrix like this:

```
matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

To access the matrix elements, we now need to use two indices. Since `matrix[0]` refers to the first list element (which is itself a list) then `matrix[0][0]` refers to the first element of the first element. You can also use the same range tricks with multiple indices. Try some of the following to be sure you know how they work: `matrix[1][-1]`, `matrix[::2][::2]`, `matrix[:][::-1]`, and feel free to try some other options on your own.

Getting used to indexing is tricky, so take some time now to practice. MATLAB has similar indexing tricks, thus understanding them will be handy in many different contexts in multiple programming languages. You might also hear people call this "slicing," as it lets you cut a particular piece out of a list.

Indexing can also be applied to strings in order to get a substring. If `a` is a string, then `a[0]` returns a string only containing the first character, `a[0:2]` returns the first two characters, and `a[-1]` return the final character. All slicing operations described above work in the same fashion on strings as they do on lists.

5.2.2 List Operations

There are many things you can do with lists:

- **Addition:** You can use the addition operator to concatenate two lists. If `a` and `b` are both lists, then `a + b` is a list containing all of the elements of `a`, followed by all of the elements of `b`.
- **Multiplication:** Multiplying a list by an integer n returns a list that concatenates n identical copies of the list. This is a nice trick for creating a long list where every entry is the same: if I want a list containing 100 zeros, then I can use `[0.]*100`.
- **Comparisons:** You can directly compare lists to use in conditional statements using any comparison operator (i.e. ==, >=, etc.). The result depends on the exact comparison being done, but in general there is an element-by-element comparison. If using the equality operator, you will get back `True` only if the lists are the same length and each element is the same. If using a comparison like >= where it is possible that you could get different answers at each place in the list, Python gives you back the result from comparing the first item in the two lists.
- **Checking if something is in a list:** You can check if a certain item is in a list using the form `<item> in <list>`. So for example, to check if 5 is in `mylist`, I would enter `5 in mylist`, which should give me back `True`. If you need to know the index where an item is, use the `index` command: `mylist.index(5)` will return the first index where 5 occurs.
- **Counting the number of occurrences in a list:** To count the number of times that something occurs in a list, use `count`. So if we want to count the number of times a list entry is

1, I would use `mylist.count(1)`.

5.2.3 Lists are Mutable

Lists can be changed on the fly in Python. You can add items, take them away, or shuffle them around. Some common ways to manipulate lists include:

- **Append:** You can add a new item onto the end of the list using `append`. To add the integer 6 onto the end of the list above, we would type `mylist.append(6)`. Try this, and then print out the entire list to verify that it has changed.
- **Insert:** To add a new item to a list in a specific location, use `insert`. You need to specify the index when you call insert, so to put item `a` at the beginning of list `mylist`, use `mylist.insert(0,a)`.
- **Pop:** To remove an item from a specific location in a list, use `pop`. If you do not specify an index, `pop` will remove the final item in the list. So to remove the final item from a list, use `mylist.pop()`, while to remove the first item, use `mylist.pop(0)`. Note that `pop` returns the item that is removed, so this is a convenient way to perform an operation on all items on a list one at a time.
- **Remove:** To remove a specified item from a list, use `remove`, which will remove the first instance of the given item in the list. For example, `mylist.remove(5)` will remove the first occurrence of 5 in the list.

5.2.4 Iterating Over Lists

We used `for` along with the `range` function in the previous lab to repeat computations. You can also use `for` along with a list to iterate over all items in a list. The syntax is

```
for <item> in <list>:
    # repeats the body of the loop with <item> changing
```

So for example, if I have a list and want to print all of the entries individually,

```
for x in mylist:
    print x
```

This is a handy way to loop over a list without modifying it (like you would need to do with `pop`). In fact, this is exactly what Python is doing when you use a loop with `range`: `range` actually creates a list containing the appropriate integers specified by the arguments, and then `for` iterates over the list items.

5.2.5 One Annoying Detail

One detail you need to be aware of when using Python lists concerns how Python stores a list in memory. If I have an integer variable `a = 5`, and then I set `b = a`, then there is no ambiguity about what what I mean when I set `b = a`: there is only one integer with a value of 5, and Python sets the value of `b` to be 5 as well. This is also true for floating point numbers and strings, as there is a single, unique representation of the item in the computer memory.

Lists are different, because lists can be altered and changed on a whim, which means that there is a more complicated way to store them in the computer's memory (since it is not a unique, unchangable number or string). When I define `a = [1,2,3]` and then set `b = a`, there is some ambiguity about what I am doing: did I mean to copy list `a` to a separate location in memory, or do I just want `b` to refer to whatever `a` happens to be, even if I change it?

The designers of Python have decided to use the second option, also known as **aliasing**. When you have a mutable object like a list and assign it to that variable, the variable points to a location in the computer's memory, saying that the list is stored there. When you assign another variable to that same variable, Python simply tells the second variable to point to the same location as the first variable. This is more memory efficient, as you do not need to copy a potentially long and complex list over to a new location in memory. However this can have unintended consequences. Try entering the following into Python:

```
a = [1, 2, 3]
b = a
b.append(4)
print a # will print a list with 4 items
```

This can be annoying if you did not mean to alter `a`, and wanted to make a copy in `b`. To avoid this, we have two options:

- For a simple list, we can make a copy of `a` in `b` using the notation `b = a[:]` or `b = list(a)`. Either of these will simply return the enire list as a new list, which can then be assigned to a new variable. However, this may not work in all circumstances, as this is what is known as a "shallow copy:" if your list contains other lists, Python will still use aliases for those items.
- If we want to create a full copy of another list, we need the `deepcopy` function, which will make a "deep copy" with full copies of all items in the list, including any lists contained in the list. This is not a normal part of Python, so we need to *import* it into Python. Type `import copy.deepcopy` to import the `deepcopy` function. Then you can make a full copy of your list using `b = copy.deepcopy(a)`.

Do not worry about the details of `import` yet; we will talk about this in more detail in the next class.

This is just to make you aware of things that can go wrong when changing lists. This is something to worry about when you send a list to a function and then modify the list in the function. Python simply makes an alias of the original list when it is sent to the function, so any modifications to the list will affect what is in the list in the environment from which the function was called. Be aware of this, and make copies of a list if you do not intend for a function to modify a list!

5.3 Practice

Here are some problems to practice concepts related to functions and lists. In each case, it is a good idea if you come up with several tests to check that you are doing things correctly.

- Write a function that calculates the average of three floating point numbers.
- Write a function that calculates the average of an arbitrary number of floating point numbers. Pass the numbers to the function as a list.
- Write a function that takes a list and reverses the order of the items in a list. The function should not modify the original list. Do this both using a for loop and indexing.
- Write a function that takes a list of numbers and a function that takes a single number as an argument, returning another number. Have the function apply the function to every item in the list, and return a list containing the results. Your function should not modify the original list.
- Write a function to test if a list of integers are prime, returning a list of boolean values. *Hint:* You should write a helper function `isprime` that tests if a number is prime, returning a boolean value (you can adapt your code for determining if a number is prime from the last lab if you like). Use your function above that applies a function to a list of values to help you do this.

CHAPTER

SIX

BASIC PROGRAMMING WITH PYTHON 3

This lab focuses mainly on one crucial skill for programming that you will use over and over as you write code for scientific purposes: **debugging**. No one writes perfect code the first time through, and you will want to develop the ability to test and find bugs in your code. Debugging requires more than anything else that you be *persistent* and *systematic* in looking at your code each step of the way. I will highlight some useful techniques for debugging in the following, but you should know that mostly you need to think carefully about what your code is doing. We will also talk a bit more about importing other modules, which gives you flexibility to reuse your own code, as well as external code packages written by others. I also briefly discuss simple ways to save data in a text file and how to read text data into Python from an external file. You will need to perform some basic text input for your homework/final project.

6.1 Types of Bugs

What types of bugs might you encounter when writing a program? There are three main types:

- **Crash:** The most basic type of bug is when your program crashes and does not run to completion. You have probably encountered this problem already when working with the Python interpreter: you get some sort of error message that describes the problem, and the line where the problem occurred.

  This is the easiest type of bug to fix, because (1) you know that there is a bug based on the fact that your program crashed, and (2) Python gave you some information about where it occurred in the program. Fixing crashes may still pose some difficulties, but compared to the other two types of bugs, this is in general much easier to fix.

- **Program Never Finishes:** Next up is a bug that leads to a program that never runs to completion. This is a bit harder to figure out, because it may just be that your (correct) program takes a *long* time to run, you just were not willing to wait long enough. This highlights an important issue in programming: it is *always* a good idea to have some idea of how long your program should take. Good ways to check this include (1) run your code on a very small subset of your data, and then progressively increase the size to see how long it might take to run on the full data, and (2) include a print statement every so often within your program to indicate progress through the code. I use (2) in almost every piece of code that I write: if I am solving a complicated differential equation with multiple time steps, I print out time step information, or if I am running something that does calculations for a large number of data files, I print out the number that I am on every so often.

If you do in fact have a code that never stops running, you may have an infinite loop in your code, or some other exit condition that is never met. These are easier to fix than the final type of bug, as you usually can tell rather easily if you are in a situation when your code will never finish, and thus take steps to fix the problem.

- **Wrong Answer:** The trickiest type of bug to fix, and the one you will no doubt spend most of your time worrying about how to correct, is when your code runs to completion but gives the wrong answer. These bugs can be notoriously difficult to find, as in many cases you might not know the right answer to the problem and thus cannot determine that there is a bug. Or you may know that there is a problem, but finding the culprit within a long and complicated code can be exceedingly difficult. In either case, this is more often than not the type of bug that will give you trouble.

The techniques I mention below are useful for finding all of these types of bugs, though you will find that for the most part they are useful for the second and third types (crashes are relatively easy to diagnose, particularly with an interpreted language like Python).

6.2 Print Statements

A classic technique for debugging is using print statements. While more sophisticated tools are available, we will not use them in this lab, and instead focus on ways to use print statements in debugging. The idea is to print out values in your code where you know what the answer should be, and if they do not agree, take steps to correct the values. Most of the time when I am debugging a piece of code, this is the strategy I use, and it is one I will try to employ when you ask me for help on something.

Let's look at a rather simple example. Let's say that we didn't know about the built-in method in Python for adding two lists, and instead decided to write our own function to do this. We will call this function `appendList`, and it should take two lists as inputs, and return another list as output. Additionally, we would like for `appendList` to not change the inputs to the function (i.e. it should have no side effects). Here is a first attempt at writing this as a function:

```
def appendList(list1, list2):
    'appends list2 to list1, returning the combined list. does not␣
↪modify either input list'

    outlist = list1
    i = 0

    while i < len(list2):
        outlist.append(list2[i])
    return outlist
```

If you try to call appendList, you will notice that the program never finishes. You probably can see my mistake, but if you cannot, how might we systematically figure out the problem? One good way to check is by adding a print statement in the function to see what is going on when the function is called. For instance, I might try something like this:

```
def appendList(list1, list2):
    'appends list2 to list1, returning the combined list. does not␣
↪modify either input list'

    outlist = list1
    i = 0

    print 'starting iteration over list2'

    while i < len(list2):
        print 'list index value:', i
        outlist.append(list2[i])
        print 'current value of outlist:', outlist

    return outlist
```

Doing this, we immediately see that we forgot to increment i each time through the loop, leading to the infinite loop. The function endlessly adds items onto the end of outlist. The print statements helped us pinpoint this problem. While I can fix this easily with an increment, I might realize that I could have avoided this in the first place with a for loop, so I could rewrite like this, keeping some debugging print statements in place:

```
def appendList(list1, list2):
    'appends list2 to list1, returning the combined list. does not␣
↪modify either input list'

    outlist = list1

    print 'starting iteration over list2'

    for item in list2:
        print 'item to be appended:', item
        outlist.append(item)
        print 'current value of outlist:', outlist

    return outlist
```

Note that when I inserted the print statements for debugging, I tried to include additional information about the meaning of the variables that are printed. This isn't totally necessary, but it can serve as a helpful as a reminder to you as the additional words make it explicit what the variables should be.

Doing this, we find that our code runs to completion (try it yourself, using two short lists), and we can see that the items are correctly appended onto `outlist`. Print out the resulting list in the interpreter to confirm that the function worked correctly. While it may seem that everything is okay, there is still one more thing to check: this function is *not* supposed to modify the input lists, but try printing out the first list that was passed to `appendList`. You should see that the first input list has been modified, which was not supposed to happen. How could we go about debugging this? I might add more print statements that tell me the values of `list1` and `list2` as I go through the loop:

```
def appendList(list1, list2):
    'appends list2 to list1, returning the combined list. does not␣
↪modify either input list'

    outlist = list1

    print 'starting value of list1:', list1
    print 'starting value of list2:', list2
    print 'starting iteration over list2'

    for item in list2:
        print 'item to be appended:', item
        outlist.append(item)
        print 'current value of outlist:', outlist
        print 'current value of list1:', list1
        print 'current value of list2:', list2

    return outlist
```

This should make it clear what is going on: because of how Python implements aliasing in lists, we appended the items onto `list1` when we append to `outlist` because the two variable names point to the same location in the computer's memory. The fix for this is to make a copy of `list1` in `outlist`, rather than an alias. Here is the corrected code, with the print statements still in place:

```
def appendList(list1, list2):
    'appends list2 to list1, returning the combined list. does not␣
↪modify either input list'

    outlist = list(list1)    # outlist = list1[:] is also acceptable

    print 'starting value of list1:', list1
    print 'starting value of list2:', list2
    print 'starting iteration over list2'
    for item in list2:
        print 'item to be appended:', item
        outlist.append(item)
        print 'current value of outlist:', outlist
        print 'current value of list1:', list1
        print 'current value of list2:', list2

    return outlist
```

You can verify that this version of the function does not modify the input, which is confirmed by the print statements.

In general, using print statements to debug your code is a good strategy. As your programs get more complicated, though, it may not be obvious how exactly one should use print statements. Here is where being systematic and persistent pay off: a good strategy is to pick a point about halfway through the program, and insert a print statement there to compare program values to expected values. If all of the relevant variables have their expected values, then the bug is most likely after this point of the code, while if there is a disagreement, then the bug is somewhere ahead in the code. Repeat this bisection method until you are able to hone in on where the bug occurs. Once you find and fix the bug, if you are still having trouble there is probably another bug in your program, so start over again in the middle of the program. This is where persistence comes in, as you need to repeatedly stick to a plan to look through the program in a systematic way.

6.3 Test Functions

A useful method for working out bugs that is related to print statements, but is slightly more sophisticated, is the use of test functions. The idea is that if designed correctly, each separate task in a program should be written as a function. We can exploit this modularity and make our debugging and testing more robust by examining each function individually with a **test function**. A test function is a separate piece of code that calls a function multiple times with particular values (often chosen to test certain cases that may give a code trouble), comparing the results to the expected values and printing out the results. Here is an example test function for `appendList` above:

```
def test_appendList():
    'tests appendList for several different cases, printing out results
↪'

    list1_vals = [[], [1], [1, 2]] # list1 values
    list2_vals = [[1], [], [3, 4]] # list2 values
    combined_vals = [[1], [1], [1, 2, 3, 4]] # expected results

    # loop over test cases, printing out test results
    for list1, list2, combined_expected in zip(list1_vals, list2_vals,␣
↪combined_vals):

        # copy lists before calling to check for modification

        list1_copy = list(list1)
        list2_copy = list(list2)

        combined_actual = appendList(list1, list2)

        if (combined_actual == combined_expected and list1 == list1_
↪copy and list2 == list2_copy):
            result = 'PASSED'
        else:
            result = 'FAILED'

        print 'Test:'
        print 'list1: expected = ', list1_copy, ', actual = ', list1
        print 'list2: expected = ', list2_copy, ', actual = ', list2
        print 'combined: expected = ', combined_expected, ', actual =
↪', combined_actual
        print 'Result: ', result
        print ''
```

This code runs three different test cases through the function `appendList`, and prints out the results. I introduced a new piece of syntax in writing this, `zip`, which is a convenient function for iterating over multiple lists at once. Basically, `zip` combines together a series of lists to let you use the convenient syntax of `for <item> in <list>`, but doing so over multiple lists at the same time without having to worry about indices. You can also do this using indices, but I find this approach to produce code that is easier to understand.

There are a few things to notice about this test function:

- First, note that I set up the problem using three lists and a for loop to iterate over the test cases and the expected results, rather than writing out the same code several times. I did this for two reasons: first, I want to make sure that each test is run in exactly the same manner, and I also want to maintain flexibility about the number of tests that can be run. To add an additional test, all I have to do is add another set of items to `list1_vals`, `list2_vals`, and `combined_vals`, and the code automatically does the additional tests. If it is difficult to run additional tests, you are not likely to do them!
- Second, note that I was careful about checking whether or not the original lists were modified by comparing the input lists to separate copies. As we saw above, it is possible to get the correct output, but incorrectly modify the input, and thus we should test the inputs as well as the output. If a function is not supposed to modify the input, then it is a good idea to build this into the test function.

- Finally, this test function could have be written independently of a working version of `appendList`. This means that I could have even written the test function *before* I wrote the actual function – once I have defined what the function `appendList` does, I can figure out how to test it without even writing it. You don't actually need to do this, but it is a good idea to think about how to test a piece of code during the entire process. The sooner you start doing rigorous testing of a piece of code, the better.

Writing test functions where you compare results with expected outputs for each piece of code is in general a good idea. Start small with the lowest level functions, and then work up to the highest level. This way, if you ever need to change a piece of the code, you can run all of the appropriate tests again to verify that you have not broken anything. In some of your homework questions, I will run a suite of test problems analogous to the test shown above to verify that your code does everything correctly, and doing so yourself is also a good idea.

6.4 Assertions

One final piece of advice for writing bug-free code in the first place is to use **assertions** in your code. Assertions are statements that cause your program to stop running if the conditions that they test are not met, and can be a good way to help catch bugs. Typically, when I write a function, the first few lines are assertions that check a few conditions that might trip up my code if the inputs are not correct. Therefore, I know right away that something is wrong, and *where* it went wrong, rather than having to deduce it from the final result and then work my way through the code systematically to find the problem. While this cannot help me debug `appendList` if it is not working correctly, if I was writing `appendList` as part of a larger project these could help me figure out if there is a bug in any code that calls `appendList`.

For instance, let's say that I am writing a piece of code where I need to combine two lists into a third list without modifying either list. This can be easily done with our `appendList` function. However, if one of the lists contains just a single number, and I somehow think that `appendList` works like the append method for a list, I might try something like `appendList(someList,a)`, where `a` is a float.

As it stands now, if I try to run the code, I get an error message regarding that `'float' object is not iterable`. While I might be able to figure out that this is the problem from the error message, I can use assertions to test this and make the failure more explicit. Basically, before doing anything inside the function, I would like to be certain that both of my input arguments are lists. If they are not, then instead of trying to do a set of operations that I know will fail, I can raise a very specific error message that explains the problem in plain English. Here is the code with assertions:

```
def appendList(list1, list2):
    'appends list2 to list1, returning the combined list. does not␣
↪modify either input list'

    assert type(list1) is list, "input argument list1 must be a list"
    assert type(list2) is list, "input argument list2 must be a list"
    outlist = list(list1)    # outlist = list1[:] is also acceptable

    for item in list2:
        outlist.append(item)

    return outlist
```

With the added assertions, now if I try `appendList(someList,a)`, I will get an `AssertionError`, along with the message `input argument list2 must be a list` (the optional string in an `assert` statement will be included in the error message). This is much more helpful in figuring out how to correctly call `appendList` than the previous error message. If I call the code correctly, both assert statements silently pass, and do not affect the execution of the code.

To check the type of a variable, I used the protected keyword `is` along with the `type` function. `type(<object>) is <type>` gives back a boolean value, and can be used to check the type of a variable. This is often something that you would want to check. If something is a counting index, assert that it is a nonnegative integer. If something needs to have a numeric value, then assert it is a float or integer. If two lists must have the same length, add an assertion to check that the lists have the same length. I try to check anything that may make my program go haywire or execute incorrectly with an assertion, as they typically give more useful information than other error messages.

Assertions can also be used as a handy trick to crash a program intentionally. If I am trying to debug something, and in doing so change something so that I *know* that the remainder of the program will not run correctly (or take a long time to run), I often short-circuit the program by adding `assert False`, which always stops the program dead in its tracks. Otherwise, to prevent the remainder of the code from running, I would have to comment it all out (which may take a long time to do and then undo when the problem is fixed). I can save myself time and typing in such a case by adding a simple assertion to the code.

6.5 Practice

Here are some problems to work on in lab today. Use print statements to help debug them as you work. You should use assertions where appropriate, and also write a small test function for each problem.

- Write a function `isPalindrome` to test if a list is a palindrome (i.e. the list is the same backwards as it is forwards: `[1,2,1]` is a palindrome, `[1,2,2,1]` is a palindrome, but `[1,2,3,1]` is not).
- One method for calculating the square root of a number s is as follows: (1) take a guess x_n, (2) check if $x_n^2 = s$ within some allowed amount of tolerance (if so, we have a good enough answer and we can exit), (3) if the guess isn't good enough, make a new guess

$x_{n+1} = (x_n + s/x_n)/2$ and return to step 1. Write a function `calcsqrt` to implement this method, taking s as an input and returning the square root.

- Add appropriate assertions and write a test function for some of the functions from Lab 5.

6.6 Importing External Code

By now, you have a basic understanding of how to program in Python (or any other language, for that matter – if you can program in Python, it is easy to learn to program a different language). Many of the things I have highlighted are aimed and making better written, more usable computer code, in particular how to use functions to accomplish this. But what if you are writing some code, and would like to use a function that your wrote at some previous time? From what we have seen so far, you can only do this by copying and pasting the function into your Python script. If we only could use a single file for all of our coding, this would get out of hand very quickly, and if you needed to make a change to the function, you would need to find it in every file and change it.

The solution to this problem is to use Python's `import` capabilities. Basically, the idea is that we can load functions and other code from a separate file using `import`, which are loaded into what is known as a `module`. Let's say we have a function `add` that we have defined in the file "addition.py." If we would like to use `add` in a separate piece of code, we can do the following:

```
import addition
print addition.add(1., 2.)
```

This code will import all functions in the "addition.py" file, and allow you to use them with the name `addition.<function>`. The use of `addition` is what is known as a **namespace**, which is just a fancy way of saying that Python groups functions together based the place from which they were imported. Namespaces help you keep track of where different functions come from. You can choose the namespace where external functions are imported with the following syntax:

```
import addition as add
print add.add(1., 2.)
```

If you were going to use the `add` function many times, choosing a shorter namespace can help cut down on typing.

If you have many functions in `addition`, but are only going to use `add`, you can also selectively import functions from a module with the following syntax:

```
from addition import add
print add(1., 2.)
```

This puts `add` in the current environment, so it if you want to minimize the amount of typing, this is the way to go. Finally, you can import all functions from a given module into the present environment using:

```
from addition import *
print add(1., 2.)
```

This is okay if there are only a couple of functions in the `addition` module, but if there were many of them, it can be confusing to not group them all together in a separate namespace and this is not considered good programming practice. This is particularly true if you import more than one module using `*`, and it is not made explicit which functions belong to which modules.

Where does Python look for modules? First, it looks in the current working directory from which python was called. If it finds a module in the present directory, it uses that module. If it does not find one in the current directory, it searches all the paths specified on what is called the "Python path," and the last place it looks is the standard installation directory for python packages. Do not worry about the last two places for right now; these are only needed if you want to create packages that can be imported from a different directory than the one you are working in (it is fairly easy to modify the Python path to find packages in other directories; I can show you how to do this if you ask).

In addition to your own modules, there are many externally available packages that are useful in many contexts. We will not cover these in this course, but I frequently use many of them, in particular `numpy` which defines numerical arrays that are basically a more powerful version of a list and useful for scientific computation, `scipy` which gives access to many standard methods for scientific computing, and `matplotlib` which is a powerful plotting library. These free and open source modules replace most of the MATLAB functionality using Python, and these are becoming common tools in science. You are of course welcome to experiment with them on your own to see if they will be useful in your research. Many of these tools are installed in the Python installation in the Mac Lab, but they are also easy to download and install on your own computer.

6.7 Reading From and Writing To Files

Over our previous three classes, we have seen how we can use Python to carry out complicated tasks on data that we enter ourselves. However, as discussed in the first class, we usually need to perform calculations on data that we obtain from *external* sources. This means we need to get the data into Python, and need a way to save results of calculations. Python has a number of capabilities for reading and writing data. We will not cover all of them, but for your homework and term project, you will need to use Python to perform calculations based on data input from a text file, so here I cover the basics of text file input and output.

6.7.1 Reading From a File

Python has built-in capabilities for dealing with text files through what is known as a **file object**. A file object is a fancy data type that represents a file on your system, and includes functions that let your read and write from the file. To open a file object, use the following syntax:

```
f = open('myfile.txt', 'r')
```

The basic command is the `open` function. `open` takes two arguments: the name of the file (`'myfile.txt'` in this case), and the mode under which that file will be opened (`'r'` in this case, which indicates that the file should be opened in read-only mode). You can specify either an absolute or a relative path for the name of the file, but you must give a string for the file name. You have several options for the mode: `'r'` indicates read-only mode, `'w'` indicates write-only mode, and `'rw'` indicates read-write mode. You do not need to provide the second argument as `'r'` is the default, though I try to always put this into my codes as I want to be explicit about how a file will be opened so that I do not accidentally open a file that I do not want to modify in write mode.

Note that if you want to read from a file, the file must exist before you try to open it. If it does not exist, you will get an error message. You also need to be careful when opening a file in write mode, as it will overwrite a file if it already exists.

Once you are finished with the file, you should close it using the `close` function of the file:

```
f.close()
```

This will close the file. The file will automatically close when the program finishes if you did not explicitly close it, but it is a good idea to close files when you are done with them because they can slow down your program if too many are open.

To read from a file that has been opened in read mode, use the `read` function of that particular file:

```
f = open('myfile.txt', 'r')
a = f.read()
print a
f.close()
```

The entire contents of the text file will then be stored in the variable `a` as a string, and will be printed to the screen. Note that everything read from a file is a string, so if you need to get numbers out of a file you will need to convert to the appropriate type.

Reading the entire contents of a file into a single string is fine for short files, but if you have a large data file it might be a bit tedious to handle the data in this way. More likely, you will have a file where a certain number of items are written on each line. To read in a single line from a file, use the `readline` function. So if you had a file with three lines, you might do the following:

```
f = open('myfile.txt', 'r')

for i in range(3):
    a = f.readline()
    print a

f.close()
```

As before, this will print the file to the screen, but each line will be separated from the next one. However, what if we don't know how many lines our file has? Fortunately, Python makes it easy by letting us loop over the file itself to implicitly read it one line at a time:

```
f = open('myfile.txt', 'r')

for line in f:
    print line

f.close()
```

Note that you can implicitly perform the `readline` function each time through the loop in this way. This is the way I recommend handling reading text from a multi-line file.

How do you deal with text once you read it in? A frequently used function for handling strings read from a file is `split`, which takes a string and divides it into words, giving you a list of strings. For example, if you have a text file with three numbers on each line separated by a space, you get those numbers as follows:

```
f = open('myfile.txt', 'r')

for line in f:
    stringlist = line.split()   # stringlist is a list of three strings
    numberlist = []
    for string in stringlist:
        numberlist.append(float(string))
    # numberlist is now a list of numbers
    # peform some analysis, etc.

f.close()
```

6.7.2 Writing To a File

Writing to a file is similar to reading from a file, you just need to open the file in write mode. Once you have the file open in write mode, you can write data to the file using the `write` command. As an example, imagine you have carried out a calculation on 100 different numbers, getting two different results. You have the results stored in a list `results`, with each list entry being a list of length two holding the two results. To write this to a text file, you might do the following:

```
f = open('calculation_results.txt', 'w')

for result in results:
    f.write(str(result[0])+' '+str(result[1])+'\n')

f.close()
```

Note that you can only write strings to a text file and that I had to explicitly convert them here before I could write to file. You might also notice that I had to put in my own dividing characters between the entries: there is a space between the two numbers, and a newline (i.e. a carriage return) character to separate each line. You can also separate the entries with a tab character using `'\t'`.

You can do more sophisticated things with python, including writing binary data and writing things in a Python-specific format that lets you save and reload complex objects. We will not worry about these things here, as for this class reading and writing simple text files is all that you will need to worry about. However, if you use Python in your research, you will no doubt want to take advantage of these capabilities.

CHAPTER

SEVEN

MATLAB 1

MATLAB is short for MATrix LABoratory, and is a common high-level tool for numerical analysis in science and engineering. A few things to know about MATLAB:

- Fundamental data type is matrix (double precision floating point numbers)
- Interactive and "interpreted" (there is a "compiler" that makes code run faster), runs `.m` files and stores data using `.mat` files
- Many, many built-in functions (we will only scratch the surface in this class)
- Built in 2D and 3D graphics capabilities
- Additional toolboxes add further capabilities for extending MATLAB beyond its basic functionality

MATLAB was originally a package for matrix math, designed to provide an interface to linear algebra libraries written in Fortran, but has grown over the years into a software package useful for a range of scientific applications. It is commonly used in academic, research, and industrial settings (which is why geophysicsts should know how to use it). MATLAB currently uses a proprietary version of the LAPACK linear algebra routines.

MATLAB is commercial software (we have a license for it on all of the computers in the Mac Lab). For those that have strong feelings about proprietary software, there are a number of free open source alternatives: GNU Octave, FreeMat, SciLab, R, a combination of Python packages, etc. Some of these are meant to be MATLAB clones – GNU Octave and FreeMat are supposed to run MATLAB code with the same syntax, while the others have a distinct syntax. I have used GNU Octave extensively in my research in addition to MATLAB, as it is fairly easy to install under various Linux distributions. I have found it mostly compatible with existing MATLAB code, though I often have to make a few modifications and sometimes must replace functions that do not exist in Octave (and some Octave code is not compatible with MATLAB). It also does not run as fast. I use Python along with a number of additional packages in my research in place of MATLAB, though that has been born out of the fact that I simply like programming in Python better (rather than the fact that MATLAB is commercial).

Regardless of all that, MATLAB is a fixture in science and engineering, and you will need to be familiar with it in any career path you choose. You will no doubt encounter software for geophysical research that has been written in MATLAB. In this course, we will be doing our lab exercises in the Mac Lab using MATLAB. The computers have been updated to the most recent release, though most of what we cover should work on other versions of MATLAB. You are also free to use one of the open source alternatives, as long as you use one that has the same syntax as MATLAB. If you decide to use something other than version of MATLAB in the Mac Lab, all I ask is that you be sure that MATLAB in the Lab can run all of your code, as I like to run your code myself – if I have to substantially tweak your code to get it to work, I will take points off.

7.1 Everything is a Matrix

The main philosophy in MATLAB is that everything is a matrix (specifically, a matrix of floating point numbers with double precision). This is very convenient for doing numerical analysis of data, because as we have seen with Python, we very often need to represent arrays of data in scientific settings. Even if you just want to consider a single floating point number in MATLAB, it is treated as a 1x1 matrix.

MATLAB stores matrices using the 1-indexing convention (i.e. if a matrix `A` contains 3 elements, then `A(1)` is the first element and `A(3)` is the last element), and does so using a convention known as “column major order.” This refers to how the data is arranged in the computer’s memory: if we want to create a 2x2 matrix, MATLAB sets aside enough memory to store 4 floating point numbers. However, there are two different ways we could store the 4 matrix entries in memory: we can either store the first row, then the second row, or we can store the first column, then the second column. MATLAB uses the second convention. This is important to know when you read data from a file into MATLAB, because numbers will be read into consecutive locations in memory from the file. We will look at some examples of how to do this in the coming labs.

Everything also is done using double precision floating point numbers in MATLAB. There are no separate types for integers or booleans, rather they are double precision numbers that are treated in a special way. This is convenient in that you do not worry about how to peform operations on different data types. However, the downside is that floating point numbers take up more memory than integers, and *much* more memory than booleans (booleans require just one bit of storage, while double precision numbers require 64!). If you are dealing with very large datasets that are intentionally stored in a different format to save space, you can sometimes crash MATLAB just loading the data into memory due to the huge increase in memory requirements when you convert to floating point numbers.

7.2 Basic Programming Tools

MATLAB contains all of the basic programming tools that we have seen so far: loops, conditional statements, functions, arrays, etc., so what we have studied so far will be useful in learning to program in MATLAB. We can manipulate matrices in MATLAB using many of the same tools we used in Python:

- **Basic operations:** MATLAB has many built-in operations, such as +, −, *, /, ^ (exponentiation). Additionally, many standard mathematical functions are included, such as `exp`, `sin`, `cos`, `mod`, etc. One nice trick about MATLAB is that if you can usually apply an operation to an entire matrix: for instance `sin(a)` will give you back a matrix with the sine of the original matrix for each matrix entry. You may also perform basic matrix operations such as addition, subtraction, and multiplication directly in MATLAB without having to loop over every entry in the matrix. This is convenient, as it lets you skip many loops in your code.

- **Loops:** You can perform calculations using loops with MATLAB. Here is a simple example that does the same thing as we have seen in Python:

```
for i=1:10
    i
end
```

  This will print out numbers 1 through 10, much like our simplest loop example in Python. You may have noticed in Python that if you type an expression into the interpreter interactively, Python prints out the result, but if you enter the same thing into a function or script, Python does not print out the result. MATLAB has a different convention: MATLAB always prints out the results of each statement entered (you will see `ans =`, followed by each number), regardless of whether it is done interactively or through a script (though results in a function are not printed out). This is why we did not need to include a print statement in this loop, simply entering `i` is sufficient to print out the result. Another way to print output is to use the `disp` function, which has the advantage of working even within a function. To suppress this output, put a semicolon at the end of each line. (This is one thing I find annoying about MATLAB, and few things irritate me more than forgetting a semicolon on some MATLAB code).

  Indentation is not important in MATLAB; the `end` statement tells MATLAB when the loop ends. You should still indent your loops the same way we did in Python when submitting code (I will always do so in my examples), but it is not necessary for MATLAB to understand what you are trying to do. The same goes for all other types of code blocks that are described below.

  To write the same thing as a while loop, we would enter

```
i = 1;
while i <= 10
    i
    i = i+1;
end
```

  As you can see, I can suppress output for the initial assignment for `i` with a semicolon, as well as when I increment `i` within the body of the loop. `break` and `continue` work in the same way as in Python when dealing with loops.

- **Conditionals:** The syntax for conditional statements is similar to that of loops: `end` tells MATLAB when the loop is finished.

```
i = 5;
if i == 1
    'i is one'
elseif i > 10
    'i is greater than 10'
else
    'i is less than 10'
end
```

This also shows how strings are defined in MATLAB. You must use single quotes to define a string in MATLAB. To include a quote character in a string, precede it with another single quote: `'Eric''s program'`. MATLAB does not support integer data types, but instead converts them to floating point numbers.

- **Functions:** You can define functions in MATLAB, however functions must be saved as a separate file, whose name is the function name followed by `.m`. So to create a simple function to add two numbers together `add`, I would create a file "add.m" that contains the following:

```
function c = add(a, b)
    % adds two numbers a and b together, returning the result

    c = a + b;
end
```

Like Python, MATLAB allows for documentation to be embedded into a function, though unlike Python it uses comments (denoted with `%`) to serve as the documentation rather than a docstring. However, it serves the same purpose, and you can view the documentation for any function (built-in or user defined) using the `help` function. Typing `help add` will display the commented line in the MATLAB command window. Rather than using a return statement like in Python, we define the variables that will be returned from a function in the function definition and simply use those in the function body.

You can define other functions within a function file, and you can even define functions within functions (though they can only be called from within that function). However, you cannot define functions from the interpreter the way you can in Python. This is something I find very annoying about MATLAB, because I often like to define short, one-time functions from the command line when I am doing interactive work (and a big reason why I use Python more frequently). Functions also do not benefit from MATLAB's "compiler" and I find that well-written functions in MATLAB can unfortunately run much more slowly than the same commands entered into the interpreter.

MATLAB supports default arguments, though it is more cumbersome to do so than in Python. To define default arguments, MATLAB has a variable `nargin` (short for *number of arguments in*) that represents the number of inputs given to a function. You can use this value to specify default inputs. For instance, if we want to write a function `exponent` that by default squares a number, we would create a file "exponent.m" that contains

```
function expval = exponent(x, y)
    % raises input x to the power y (default is 2)

    if nargin == 1
        y = 2;
    end

    expval = x^y;
end
```

Basically, you can use `nargin` to query if the function is called with fewer than the standard number of parameters, and depending on the number of inputs you can assign default values. However, this is less flexible than we saw in Python: you are only allowed to pick one case for each value of `nargin`, so if you have more than one optional argument you must decide in advance what defaults that corresponds to.

- **Indexing Arrays:** You can perform many of the same tricks in MATLAB as you can with Python. We will cover this in more detail in the next lab, but the syntax is quite similar. If `a` is a vector, `a(1:2)` returns a matrix containing only the first two elements of `a`, `a(end)` returns the last element, and `a(4:2:10)` takes every second entry starting with 4 and ending with 10 (inclusive). Note that this is slightly different from python, both in the order of the numbers and in the differences between zero- and one-indexed arrays.
- **Assertions:** You can (and should) use assertions in your MATLAB functions just like in Python. Use `assert(<condition>)`

These basic tools should get you up and running with MATLAB.

7.3 Importing External Data into MATLAB

So far, we have been developing programming skills. However, in order to analyze real data, we need to be able to read the data into our programs. With Python, we mostly used `raw_input` to do this, but eventually we will need to read data from files. There are several ways to do this in MATLAB:

- `load` can be used if your file is a text file that only contains your data (i.e. no metadata is allowed, and it must be consistently formatted). MATLAB figures out how many rows and columns are in the data, and creates a matrix from the data. This is the simplest case, and I have provided two example files to use for this: type `load <filename>` to store the matrix in the name of the file. To store the data in a different variable, use `a = load(<filename>)`, where you must put the filename in quotes so that MATLAB treats is like a string.

  The two provided files contain the same data (a 5 column x 20 row matrix), but stored in different row/column ordering: one is ordered by column (MATLAB's standard way of storing data), the other by row. Try reading both in to see how they are organized differently. If you want to get the "correct" ordering for the row-ordered data, take the transpose of the matrix once it is read in (you can do this in MATLAB using the apostrophe: if `A` is a matrix, then `A'` is the transpose of that matrix).

- `fscanf` can be used to read more complex data. You need to first open the file using `fopen`, then read the data with `fscanf`, and then close the file. Here is an example:

  ```
  fid = fopen('columnorder.dat','r');
  a = fscanf(fid, '%f', [5 20])';
  fclose(fid);
  ```

  First, we open the file, giving the file name and the `'r'` indicates that we are only reading from the file (not writing to the file). Once the file is open, we get what is called a file handle, which is just a number that tells MATLAB what file we want to read from (this lets us have more than one file open at a time).

  Next, we use `fscanf` to read from the file specified by the handle `fid`. The `'%f'` means that we are reading a floating point number, we could also use `'%i'` for an integer and `'%s'` for a string (this will read a single character). The last entry tells us the shape of the matrix we want to make: it will contain 5 rows and 20 columns, after which I need to take the transpose to get the right matrix back. This is because the `load` function actually works backwards relative to how the numbers are ordered in the file (which is how `fscanf` read the file). To read the row ordered data, open `roworder.dat` and enter `a = fscanf(fid,'%f',[20 5]);`

  Finally I close the file handle using `fclose`. As you can see, this is more time consuming than `load`, but if you have data that is formatted in a complex way, this is how you can tell MATLAB how to deal with it.

- `fread` is for reading binary data. I have provided the column order data as a binary file (saved with double precision format with a little endian byte ordering). To read this data, I can do the following:

  ```
  fid = fopen('columnorder_binary_double.dat','rb');
  a = fread(fid, Inf, 'double');
  fclose(fid);
  a = reshape(a, [20 5]);
  ```

  This will read the data into a single vector, which is then reformatted into a 5x20 matrix using `a = reshape(a,[20 5])`. If you wanted to have MATLAB do the reshaping for you, use `a = fread(fid,[20 5],'double')`, analogous to how `fscanf` works. If your data was not double precision, or had a byte-ordering that is different from the conventions on this computer, you will need to supply different commands (see the documentation for more details). If you want to try a file with a different byte ordering and format, I have included the same data with a big endian convention and a 32 bit integer format, which you can read using `fread(fid,[20 5],'int32','b')`

While binary data has the disadvantage of not being human readable, reading and writing binary data is usually significantly faster than the equivalent text version of the data. If you find yourself waiting frequently for large datasets to read or write, converting to binary can often speed things up.

These are the basic ways to load data, though MATLAB has some additional fuctions to handle text data with delimiters between each matrix entry (`dlmread`). See the documentation if you are having trouble reading a data file in a particular format; there is a good chance there may already be a built-in way to handle the data. Some of your homework will require reading data of various formats, so spend some time when doing the exercises learning how these commands work.

7.4 Practice

Here are some problems to work on in lab today with MATLAB.

- Practice reading the different data types into MATLAB with the files I have provided.
- Rewrite your square root function from the previous lab as a MATLAB function. As a reminder we can calculate the square root of a number s is as follows: (1) take a guess x_n, (2) check if $x_n^2 = s$ within some allowed amount of tolerance (if so, we have a good enough answer and we can exit), (3) if the guess isn't good enough, make a new guess $x_{n+1} = (x_n + s/x_n)/2$ and return to step 1. Write a function `calcsqrt` to implement this method, taking s as an input and returning the square root. Make sure you put an appropriate assertion statement into your code (think about what could cause your calculation to fail), and use print statements to help debug it. Write a test function `test_calcsqrt` to test your code. To confirm that your code works, you can use the built-in function `sqrt` (or use exponentiation with a power of 0.5) in MATLAB to check your results.
- Rewrite your function to test if the entries of a matrix are prime, putting the respective values in a matrix of the same shape. If the entry is prime, use a value of 1 to indicate true, otherwise use a value of 0 (false). As before, it is a good idea to define a helper function to determine if a single number is prime.

CHAPTER

EIGHT

MATLAB 2

One powerful capability in MATLAB is the way that you can manipulate and view matrices. This includes doing matrix math (addition, multiplication, etc.), and how we can view the data that is in a matrix. We have seen a taste of matrix manipulation in our experience with Python indexing for lists and strings, and MATLAB has similar capabilities. This lab will focus on a number of things in using matrix operations, including use of the colon (`:`) operator in MATLAB, which is one of its most powerful, yet often tricky, elements.

This set of notes can only start to describe all the things you can do with MATLAB. I have included a document “mtt.pdf” (“MATLAB Tips and Tricks”) that a group of MATLAB users on the internet put together a number of years ago describing various techniques to manipulate matrices and arrays in MATLAB in the fastest and most efficient way. Feel free to refer to it if you like for your personal knowledge as you use MATLAB in your research, but you won’t need to know all of the information contained therein for this class.

8.1 Matrix Math

MATLAB can perform complicated math on matrices with a succinct syntax (so that you do not need to include `for` loops to iterate over an entire matrix to do the same calculation on all of its elements. Doing math in this array fashion in MATLAB is generally more efficient than writing your own `for` loops, though this difference is much less than it used to be due to MATLAB’s “compiler.” However, the compiler does not work on MATLAB functions, so this remains an important skill when dealing with complex data analysis tasks. This efficiency comes at a cost to transparency: often, the syntax for such array operations can be rather opaque when compared to an explicit loop, and it may be tricky to (a) figure out how to express some desired computation as array math, and (b) decipher what someone else’s code is doing, particularly when that someone else is you and you wrote your code 2 months ago and cannot remember how it works. This highlights that you should always comment your code, and that commenting becomes especially important the more fancy your MATLAB skills become. If you hand in a homework assignment with complicated array operations that are not described with comments, you will lose points. We will discuss tips for writing MATLAB code this way in the next few labs.

Examples of ways to do array operations include, but are not limited to:

- Matrix math, such as addition, subtraction, multiplication/division by a constant, can be done directly on matrices.
- MATLAB is designed to do linear algebra, so it is often most efficient to write your operations as a series of matrix operations. You can use matrix multiplication and exponentiation on matrices, though the dimensions of the matrix must follow the rules for the corresponding operation.
- If your operation is not true matrix multiplication, but rather array multiplication, division, or exponentiation (i.e. perform the given operation element by element), use `.*`, `./`, or `.^` on two matrices with identical dimensions. For example, if A and B are each 2x2 matrices, then the following loop calculations

  ```
  for ii=1:2
      for jj=1:2
          C(ii,jj) = A(ii,jj)*B(ii,jj)
      end
  end
  ```

  can be done concisely using `C = A.*B`. This is very handy, and I find myself using this operation much more frequently than true matrix multiplication.
- Most built-in mathematical functions work on arrays without any need to do loops. Things like taking an exponential (`exp( )`), logarithm (`log( )`), trigonometric functions of all types (`sin( )`, `cos( )`, etc.)

8.2 Slicing, Viewing, and Broadcasting

The colon operator is very useful for looking at array slices (i.e. a particular range of a matrix) and viewing those particular contents. You can use it to refer to an entire range of an index, a particular range of an index, a particular spacing of an index (a nice and quick way to de-sample data). For example:

- `A(:,1)` shows the first column of `A`
- `A(1,:)` shows the first row of `A`
- `A(1:2,1)` shows the first two elements of the first column of `A`
- `A(2:5,1)` shows elements 2 through 5 of the first column of `A`
- `A(5:end,1)` shows all elements beyond position 5 in the first column of `A`
- `A(1:2:7,1)` shows the odd elements through 7 of the first column of `A`
- `A(2:2:end,2:2:end)` shows only the even-numbered elements of `A`

As you can see, you can pull many tricks using the colon operator. Practice using it on a matrix to get more comfortable with it. You can use such tricks to broadcast only a particular range of data to another function. Note that there are a few differences from Python: the step (when provided) goes in the middle of the colons, and you use `end` to represent the last entry. Negative indices are not allowed in MATLAB.

8.3 Evaluating a Matrix with a Matrix

Another powerful (and often opaque) trick is to use a matrix to evaluate a matrix. The matrix must have all positive values within the limits of the matrix being evaluated, or be a logical matrix (see the next section). For example, if `A` is a 5x5 matrix, and `b = [1 2 5]` then

- `A(b,1)` shows elements 1, 2, and 5 of the first column of `A`
- `A(2,b)` shows elements 1, 2, and 5 of the second row of `A`
- `A(b,b)` is a 3x3 matrix populated by the elements indexed by all possible pairs involving 1, 2, and 5

This is very handy trick for picking out certain pieces of a large dataset.

8.4 Logical Matrices

MATLAB also has a logical data type, which can have two values: `1` (true) and `0` (false). You can also create matrices of logicals. These are useful if you would like to find certain values of a matrix that satisfy particular conditions. Evaluating a matrix with a logical matrix requires that the logical matrix have the same length as the corresponding dimensionIn the following, `A` and `B` are vectors of the same length:

- `A > 1` returns a logical vector, where entry `k` is `1` if `A(k) > 1` and `0` if `A(k) <= 1`
- `A > B` returns a logical vector, where entry `k` is `1` if `A(k) > B(k)` and `0` if `A(k) <= B(k)`
- `A(A > 1)` returns a vector containing only the entries of `A` that are greater than 1

The final type of expression above is extremely useful if you need to find data that exceeds a certain threshhold. See the exercises for an example application.

8.5 Exercises

1. These problems use the datafile "newmadrid2015.txt" which is the earthquake catalog for the New Madrid Seismic Zone for 2015. The catalog is in ASCII format. Each line consists of four numbers: date (in decimal years), latitude, longitude, and magnitude. Read the data into MATLAB, saving the data as a large matrix.

2. Use the colon operator to display all four pieces of data for the first event.
3. Use the colon operator to display only the first 5 event dates in the command window. Then, use the colon operator to display the final 5 event magnitudes.
4. Use the colon operator on the large matrix to view all four pieces of data for every 5th entry (i.e. entries 5, 10, 15, ...).
5. Evaluate the large data matrix with another matrix to display all four pieces of information for events 20, 38, 98, 119, and 160.
6. Using a logical matrix, display the date of all events with a magnitude above 3.0.
7. Using a logical matrix, display the date of all events with a latitude smaller than $36°$.
8. Using a logical matrix, display the date and magnitude of all events in the rectangular region with latitudes between $35.5°$ and $36°$ and longitude between $-90.5°$ and $-90°$.
9. Using a logical matrix, display the date of all events in January. You will need to convert decimal years into the appropriate format – try to do this using array math.

CHAPTER

NINE

MATLAB 3

MATLAB has built-in 2D and 3D graphics capabilities that can generate production quality figures. Basic plotting is relatively straightforward, but MATLAB has more sophisticated features that give the user control over most of the plot details. The appearance of a plot can be tweaked manually using the MATLAB GUI, though the same things can be done at the command line level. I personally like to write m-files to produce all of my graphics and specify the appearance using the command line so that I have (a) precise control over the graph appearance (I'm probably a little more fussy than most people about figure appearance) and (b) the ability to easily reproduce the same graph format if I need to make any data changes. MATLAB has other tricks for reproducing figures, but I find that knowing a bit about how MATLAB handles graphics internally helps me make the figure look the way I want it to.

9.1 Basic 2D Plotting

MATLAB has many, many plot types. Probably more than 95% of my plotting is done using `plot`, `semilogx`, `semilogy`, `loglog`, `scatter`, and `hist` (though you can make `plot` do what the logarithmic scales and `scatter` do very easily, so those could all be considered the same), so it is likely that you do not need to master very many of them.

Make some example plots using the above commands. To quickly generate data to plot, use `linspace` to create x-axis data, and then evaluate the x-axis data using a built-in MATLAB function.

For 2D plots, you can tell MATLAB precisely how you want it to plot symbols and lines by using the following syntax:

- `plot(x,y,'o')` will plot a circle for every data point.
- You can also plot both lines and symbols using `plot(x,y,'-o')`
- You can control line styles by specifying `plot(x,y,':')` for dotted lines, for example.
- Color is specified by `plot(x,y,'r')` for a red line, with other colors available with a shortcut including `'b'` for blue, `'g'` for green, and `'k'` for black.

- Look at the documentation to get information on all of the plot styles. You can also change the style after the fact using the plot tools or a graphics handle (more on that later), which gives you even more control over the plot style than the options available through the shorthand notation.

To plot more than one curve on the same plot, you can either call the `plot` function with multiple pairs of arguments, such as `plot(x,y,x,y1)`, or you can use the `hold` command. Once you call `hold on`, all successive plotting commands will plot to the same axes (rather than creating a new one):

```
plot(x,y)
hold on
plot(x,y1)
hold off
```

Be sure to turn `hold off` when you are done issuing plotting commands for a given figure, or your next plot will show up on the same axes.

Spend some time familiarizing yourself with various MATLAB plotting functions. Try your hand at the following:

- Make a basic xy plot showing a single curve using `plot`. Do the same for `semilogx`, `semilogy`, and `loglog`.
- Make a basic xy plot showing a single curve, where the line is dotted and the symbols are upward pointing triangles.
- Make a basic xy plot showing four curves, where one has a solid line, one has a dotted line, one has a dashed line, and one has a dot-dashed line
- Make a scatter plot using the `scatter` command (the `rand` function is convenient for making datasets with a large scatter)
- Plot a histogram using `hist`. Again, the `rand` function is useful for generating data for making a histogram. Note that `hist` creates both a histogram plot and the bin edges/counts. Newer versions of MATLAB have replaced the `hist` function with separate functions for creating a histogram plot (`histogram`) and the bins/counts (`histcounts`), so be aware of this if you are using a more recent version.

9.2 Basic 3D Plotting

MATLAB can also plot data in 3D. Commonly used functions for doing this include `plot3`, `pcolor`, `contour`, `surf`, and `mesh`. You can easily create data for `pcolor`, `contour`, `surf`, and `mesh` using the `peaks` function. You can also calculate your own surface functions, for which `meshgrid` is handy to create 2D arrays for the x and y coordinates of a grid.

Most 3D plots use a colorscale to denote values. To display a colorbar, type `colorbar`. You can change the colormap when calling the plotting command. There are many different colormaps to choose from – the default `parula` colormap that has appeared in more recent versions of MATLAB is generally a good choice. This is because the brightest color in the colorbar (yellow) is at one extreme, and the darkest color (deep blue) is at the other. The intermediate values are a shade of green, which interpolates nicely between the two colors. This choice has two advantages: (1) black-and-white versions of plots on this scale accurately represent the values, as the yellow will show up as white, while the dark blue will show up as black, and the values that are in between will be gray, and (2) the most common form of colorblindness is being unable to distinguish red and green, so this map is good in that it does not use both of them. The `hot` colormap is also a fairly good choice, as it uses red to interpolate between the dark black and bright yellow/white at the extremes, yielding the same benefits as `parula`.

Rainbow colormaps should be avoided, such as the old MATLAB default of `jet`. They may look nice, but it is actually a poor way to represent your data. This is because the brightness of the plot does not vary in a way that is associated with the values represented by the colors (the brightest colors are the yellow and light blue, which are not at either end of the spectrum). This means that a black and white print-out will be practically unusable. Additionally, the most common type of colorblindness is an inability to distinguish green and red, so any colorbar that uses both green and red will be hard for a colorblind person to read. Thus, rainbow colorbars are a bad choice for these types of plots. (Note that many people do not recognize this problem, and continue to use rainbow colormaps!)

For 3D plots, you can click and drag on the figure window to change the viewpoint, in case the default viewpoint is one that obscures the important features of your graph.

- Make a 3D line plot using `plot3`. A good function to use for these plots is the following:

  ```
  t = 0:pi/50:10*pi;
  xt = sin(t);
  yt = cos(t);
  ```

  which will show an upward-going helix if you plot `t` on the vertical axis.

- Make some different 3D plots of the `peaks` function using `pcolor`, `contour`, `surf`, and `mesh` with a color bar. Use `colorbar hot` to change the colorbar to something better than the default. You can also try the 3D version of a contour plot with `contour3`.

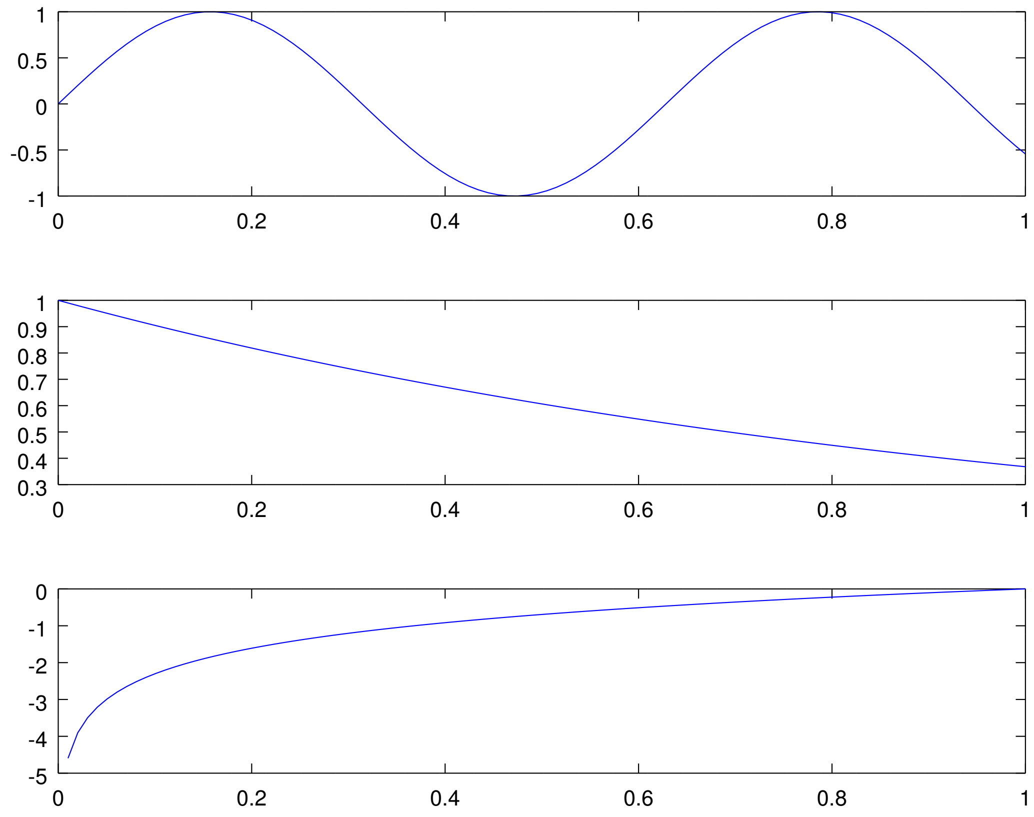
9.3 Multiple Plots

What if you want to show several plots in the same figure? The `subplot` command automatically sets the position of each set of axes depending on the number of plots and the desired arrangement. The syntax is:

```
subplot(3,1,1);
plot(x,y);
subplot(3,1,2);
```

```
plot(x,z);
subplot(3,1,3);
plot(x,f);
```

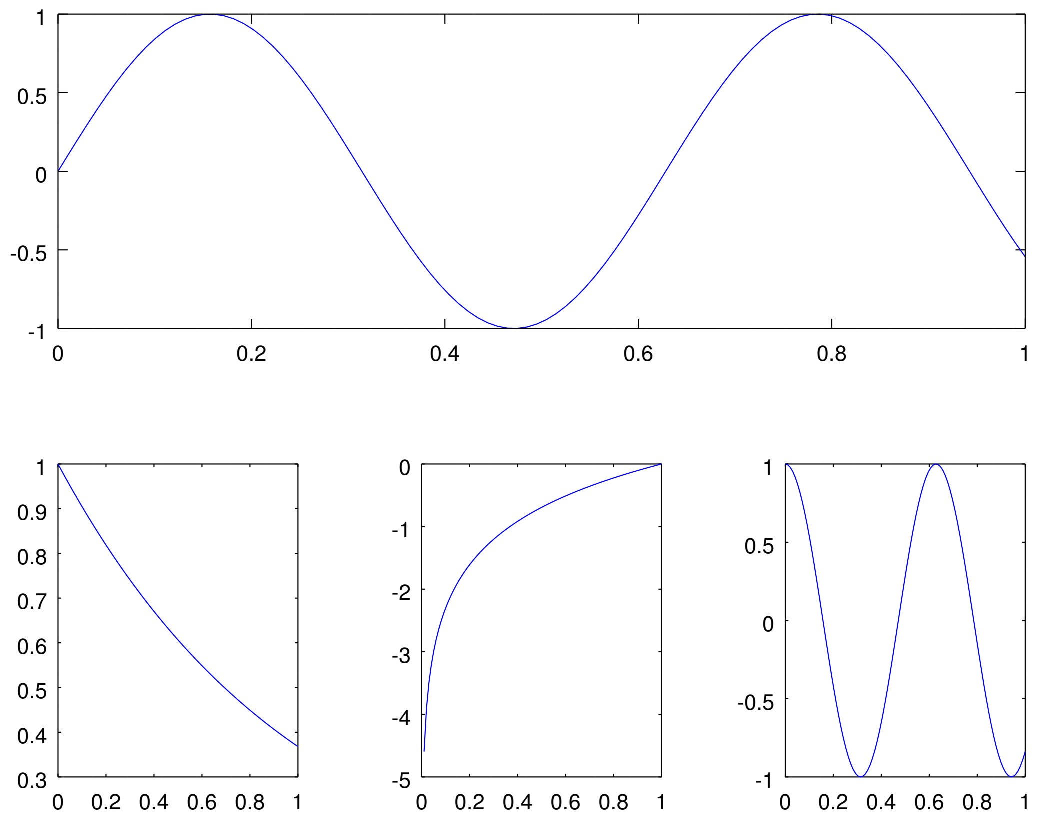
This produces a set of three plots on top of one another, like this:



Similarly, `subplot(1,3,1); ...` will produce three plots side by side. You do not need to call all instances of subplot, which lets you make complicated arrangements:

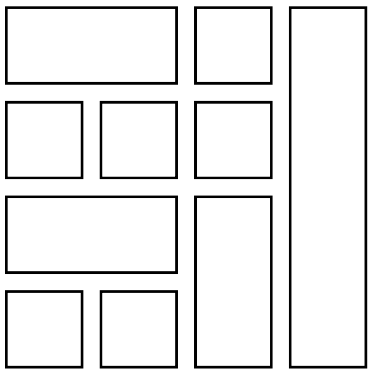
```
subplot(2,1,1);
plot(x,y);
subplot(2,3,4);
plot(x,z);
subplot(2,3,5);
plot(x,f);
subplot(2,3,6);
plot(x,g);
```

makes a large plot on top, and 3 plots side-by-side in the lower part of the figure:



Try making the following figures with multiple plots:

- A 3x3 grid of plots.
- The arrangement below:



9.4 Limits, Labels, Titles, and Legends

To change axis limits, use the command `axis`. `axis` takes one argument, which is a vector of length 2, 4, or 6. If only 2 arguments are given, it sets the minimum and maximum x limits to those values. Adding additional pairs of numbers specify y and z limits. You can also use `xlim`, `ylim`, and `zlim`, each of which takes a vector of length 2. There are additional options for the `axis` command, see `help axis`.

To label an axis, use the `xlabel`, `ylabel`, or `zlabel` command, which takes a string as its argument, and sets the appropriate axis label to this string. You can also specify details about the label (font size, font name, etc.), which we discuss further when we talk about graphics handles. Similarly, you can title your graph using the `title` function, which takes the same arguments as the label commands.

To add a legend to a graph, use the `legend` command. `legend` takes one string per line that appears on the plot, and labels that line in the legend using that string. You can also specify additional parameters in the `legend` call; one that is particularly helpful is the `'location'` parameter, which tells MATLAB where to draw the legend. Usually it tries to find the place it thinks is best (the default value for `'location'` is `'best'`), but you can specify `'north'`, `'south'`, `'east'`, `'west'`, and all combinations like `'northwest'`, which place the legend in various places in the figure. Experiment with `legend` to understand how it works.

Exercise: Take one of your basic plots from above, and add an x label, a y label, a z label (for one of the 3D plots), and a title. Add a legend to the plot with the four curves with different line styles.

9.5 Graphics Handles and Customization

You can customize your plot using the MATLAB user interface using the plot tools. From an open plot, click on the “plot tools” icon on the figure window, which will dock your figure window into the user interface. The plot tools interface lets you choose all kinds of options (font size, font name, tick locations, tick sizes, legend placement, whether to draw a grid or not, line thickness, etc.). There are many, many options to choose from (more than I can list here), and you can generate plots with a variety of styles.

Spend some time changing different parameters using the plot tools to understand how to customize a plot. In particular, make the following changes to some of your previous plots:

- Change the line thickness
- Change the frequency of the axis tick marks
- Add secondary tick marks
- Add grid lines
- Change the font

- Change the font size
- Change the marker edge and fill colors on your `scatter` plot
- Change the legend location
- Change the colorbar location on a 3D plot

In my opinion, the default MATLAB plots make it easy to make plots that are hard to read. The biggest problem is that the default font size is *always* too small, nearly independent of the intended destination of the plot (poster, journal article, presentation, web). I have no idea why they insist on making 10 pt font the default, especially when they make the default figure size so big and make it difficult to change the size of the figure. The plots in this document use the default values, and I think they are difficult to read when resized for a document like this. In my opinion, you should *always* increase the font size of a MATLAB plot, which you can see in some of my examples.

While the plot tools are great for getting a figure to look the right way, the downside to using them is that you have to repeat yourself. As I have tried to emphasize thus far, automating things saves you time and makes your results reproducible. To see why this would be useful, let's say that you have just spent 15 minutes making your plot look just "so" using the plot tools and you happily head to your advisor's office to show off your work. After discussing for a bit, you both decide that while the plot looks very nice, it would really be best if you made a small change to the plot. If you do not have a way to automate the tweaks to the figure, you will have to spend 15 minutes producing the figure again.

There are several ways to help make your results more reproducible:

- MATLAB includes a `.fig` format for saving figures, which lets you open a saved figure directly in MATLAB. This is useful if you just need to tweak the formatting of the plot, add a legend, etc., but won't let you directly change anything about the underlying data in the plot. This is more akin to saving your work than actually automating production of your figure, so while it is a good idea, I don't recommend this approach.
- If you save your figure as a `.fig` file, one way you can help make your figure reproducible is to use the "Generate code..." option under the "File" menu. This will create a MATLAB m-file based on the `.fig` file to re-do all of the figure formatting. Once you have the m-file, you can use that as a script to automatically generate plots with the same customizations. This won't necessarily help you automate generation of your original figure, but if you used a MATLAB function or script to generate the original (unformatted) plot, you can combine the original code and your tweaking code to automatically re-create your figure that you worked hard to make in the first place.

  As long as you don't mind spending the time using the plot tools to customize the plot the first time, this is a good practice to follow to avoid doing extra work every time you make the same figure.
- You might have some tweaks you like to put in every figure, such as increasing the font size by defauly. You would need to put those in using the plot tools every single time you save a plot before creating the m-file, then running that code again. It's a bunch of extra work, and such customizations are tailor made for using a function.

One way around this is to write a MATLAB function that you call when making your plots that does this customization for you. This is easy to do once you have created an m-file based on a .fig file, as an m-file can be easily turned into a function. Just have the lone argument to the MATLAB function be the figure number (use `gcf` if you aren't sure what the right number is), and have the function return the same figure number. This will allow you to repeadly apply common formatting commands without extra work.

- However, the best approach in the first place is to automate graphics customization when writing your original code. This requires knowing some basics about MATLAB's rather opaque graphics handle system. It takes a bit of work to become familiar with them, but if you want to be a real pro at producing nice plots with MATLAB, it is really the best method.

  I use the graphics handles whenever I make a plot in MATLAB, as it allows me to fully control how my figure looks and do it reproducibly and in an automated fashion. I will always include such commands in my versions of MATLAB code (both in-class and homework solutions), so that you can see examples as to how it can be used. While you don't need to understand any of this at all to be a proficient user, if you are curious what my plotting commands mean, knowing how graphics handles work will clarify what I am doing.

I have included a few documents that give useful tips on using graphics handles on the class website. One is an article explaining some basics of MATLAB graphics handles ("Matlab_Graphics.pdf"), and another is a MATLAB script ("graphics_handle.m") that gives several examples of customizations with comments explaining what I am doing. Read those more carefully to get an idea of what you can do with MATLAB.

The short version of graphics handles:

- Everything (figure, axes, line, markers, patches, text) has a handle, a unique number that MATLAB uses to identify that object. This is the same concept as the integer that identifies a file.
- Everything fits into a hierarchy of parents and children. All axes are children of some figure, and all lines are children of some axes.
- Once you have the handle, you can see what properties that object has by typing `get(h)`, where `h` is the handle (integer number) referring to that object.
- To see only the value of a specific property, type `get(h,'<property>')`.
- To set the value of a specific property, type `set(h,'<property>',<value>)`. If you don't know what values the property can take, type `set(h,'<property>')` without providing a new value for the property.
- If you don't know the handle of the current figure, axes or object, you can get it using `gcf` (figure; short for "get current figure"), `gca` (axes), or `gco` (object). This is always the most recent figure, axes, or object that was modified from the command line.

Feel free to play around with handles of various objects and see what their properties are and how you set them. If you don't mind using the plot tools, feel free to skip over this – I was a proficient MATLAB user for many years without knowing one bit about how to use graphics handles, though

I learned more as I started to automate more and more of the work that I was doing. Regardless of how you decide to do it, be sure that any plot that you create can be reproduced in some automatic fashion (it will save you loads of time and effort).

9.6 Saving and Printing

Finally, how do you save your graphics in a format that can be used for another purpose? I mentioned previously that you can save a figure from the figure window, and that you can choose any number of formats. If you want to save a file from the command line (to put it into a script so that you can automate and reproduce things), the `print` function is very useful. The basic syntax is `print '<driver>' '<name>'`, where `<driver>` specifies the type of file that you would like to save (or the type of printer that you desire to use for output). `print` can also be called as `print('<name>','<driver>')` or `print(<handle>,'<name>','<driver>')` to print a specific figure based on its handle. The most common formats will be `-dpdf` for PDF output, `-deps` for black-and-white Encapsulated PostScript (EPS), `-depsc` for color EPS, and `-dpng` for PNG files. Since PNG is a pixel-based format, you can give an optional `-r<number>` to specify resolution in pixels per inch (the default is 150). Any of these are fine for your homework – if you care about the technical differences between any of them, I am happy to explain.

So for example, to save the current figure as a PDF, you would type `print '-dpdf' 'figure.pdf'` or `print('figure.pdf','-dpdf')`. To save the current figure as a PNG with a resolution of 100 ppi, you would type `print '-dpng' '-r100' 'figure.png'` or `print('figure.png','-dpng','-r100')`. To save the figure with handle `h` as a color EPS file, use `print(h,'figure.eps','-depsc')`. Try saving the figure that you have made in this lab using this method.

Additionally, MATLAB has the `saveas` command, which works in a similar manner as `print`. The syntax is slightly different: `saveas('figure.pdf','pdf')`, `saveas('figure.png','png','r100')`, or `saveas(h,'figure.eps','epsc')`, but you should be albe to achieve the same results as using `print`.

I have provided some examples of using the `print` and `saveas` commands in the same m-file where I illustrate graphics handles.

CHAPTER

TEN

MATLAB 4

MATLAB is an interpreted language, in the sense that you can type a command and it executes immediately. This is in contrast to a compiled language, which converts your entire code into a format suitable for execution by your computer's processer and then executes it. While MATLAB has what is called a "Just In Time" compiler (JIT) known as the "accelerate" feature, the fact that MATLAB is interpreted sometimes slows it down. However, this has become less true in recent years, as the JIT compiler speeds up a lot of loops that MATLAB used to take a long time to execute. If you want to see the difference that the JIT compiler makes, type `feature accel off` into the command line (`feature accel on` turns it back on).

One way that MATLAB code can be sped up is by doing what is referred to as "vectorization," which essentially means that instead of using loops, we attempt to write computations as matrix or array operations on an entire vector/matrix that have been optimized for speed by the programmers that develop MATLAB. Figuring out ways to vectorize code is often more art than science; you either need to learn some of the non-obvious tricks that people have figured out over the years, or come up with them yourself. This lab looks at some ways that you can optimize MATLAB code.

My strategy for optimizing MATLAB code is really quite lazy. If there is a really obvious way to vectorize something, I do so from the beginning. If there isn't something obvious, I'll just write it as a loop and run it to see how long it takes. If it runs slowly, then I will think for a little bit about vectorizing it (like 15 minutes maximum). If I can't find an obvious way to vectorize it, I will just write a Fortran program to do it. Even vectorized MATLAB code usually runs slower than unoptimized C or Fortran code, and I can usually write looped Fortran code faster than I can think about how to vectorize something since it is a direct translation of my existing code. However, if you are not a good C or Fortran programmer, then you might find that it is handy to know more vectorization tricks.

I have included a document that describes many techniques for vectorizing MATLAB code ("mtt.pdf", also referred to by the MATLAB 2 lab) with the materials for today. It is worth looking through, and is a good source for tricks to optimize many common MATLAB operations. There are also many other places where you can find info on MATLAB vectorization on the web that you can find using your favorite search engine.

This lab may take you longer than the class period to complete, so feel free to spend more time on it during the next class (the next class covers more miscellaneous topics that are not as essential to using MATLAB as vectorization). The second homework will also focus on this skill, so this is definitely something to spend some time on as it will certainly help you write efficient MATLAB code.

10.1 Defining Arrays

There are several tricks for defining arrays quickly. Besides the method of evaluating an entire vector (i.e. `x = 0:0.01:1; y = sin(x);`), plus `linspace`, `logspace`, and other predefined matrices like `eye` (identity matrix), and the `reshape` command to transform one matrix shape into another. Additionally, there are a couple of other tricks:

- `repmat` ("repeat matrix") – if you have an existing matrix that is repeated multiple times, then `repmat(A,size)` concatenates multiple copies of `A`. Try it out and see how it works (for instance, try giving it a scalar, and then `[m n]` for its size argument).
- "Tony's trick" (named for the person who figured out how to do it and then spread the word on MATLAB message boards). This trick makes a constant matrix by taking advantage of the fact that you can evaluate a matrix with a logical matrix, and that MATLAB treats even a scalar value as a matrix: `A = b(ones(m,n))` creates an m x n matrix with the value of `b` everywhere.

  Basically, MATLAB is evaluating the scalar value at every location in the matrix where it finds a 1, so it assigns 2 to every matrix element. Note that this gives the same result as `2*ones(m,n)`, but is faster because it avoids repeatedly doing multiplication.
- `diag(A)` makes a square matrix with the entries of a vector `A` on the diagonal. There are additional functions to generate other kinds of sparse matrices that are useful for vectorized matrix generation.

10.2 Shifting Arrays

We may want to shift array positions to calculate something. One can shift the elements of a vector by one place using `A([end 1:end-1])`, or for k places by using `A([end-k+1:end 1:end-k])`. This can also be done for a matrix along the first axis using `A([size(A,1) 1:size(A,1)-1],:)`.

You can also do this using the `circshift` function, though this trick is faster. This trick will be useful on the homework assignment, both in vector and matrix form.

10.3 Array Math

You can do array math using compact notation in MATLAB. Matrices and vectors can be added and subtracted and multiplied by scalars. We have already mentioned the element-wise operators like `.*`, `./` and `.^`. There are additional tricks, but they are rather specialized so if you need to do some complex array math like multiplying certain slices of a matrix by different numbers, there may be a nice way to do it in a single line. See the file "mtt.pdf" for some examples.

It is also helpful to recall that MATLAB was originally designed as a linear algebra package, and consequently it is very fast when doing matrix math. Thus, if there is a way to write a computation as a linear algebra problem (i.e. matrix multiplication, solving a linear system, etc.), then that will most likely give you the most efficient execution. As an example, we will re-cast the sum of a Fourier series into a matrix multiplication problem to illustrate one way of "vectorizing" MATLAB computations.

10.4 Timing Code

MATLAB has a convenient method for timing code. To time a section of code, place `tic;` before the code, and `toc;` after the code. After running, you will see the elapsed time. However, note that if you include other things like plots and graphics inside of the `tic toc`, the numbers may not be meaningful because producing graphs can be much slower than the numerical computation. Be sure that you are aware of what you are timing when comparing code!

10.5 Practice Problem

We will look at an example of vectorizing MATLAB code for summing a Fourier series to obtain a "seismogram." This example is taken from Stein and Wysession, *Introduction to Seismology and Earth Structure*. The problem looks at pulses traveling on a string that are reflected at the boundaries, illustrated below. A string of length $L = 1$ is fixed at both ends, and waves travel with a speed $c = 1$. Because of the fixed ends, waves that reach the boundaries are reflected with the opposite sign. We are interested in determining the signal measured at $x = 0.7$ due to a source at $x_s = 0.2$ (though your code should allow for an arbitrary specification of x and x_s).

This problem can be solved through a normal mode summation. Each mode is given a weight that depends on the source position x_s, receiver position x, and source duration τ, and the wave amplitude is a function of time $u(t)$:

$$u(t) = \sum_{n=1}^{\infty} \sin\left(\frac{n\pi x}{L}\right) \sin\left(\frac{n\pi x_s}{L}\right) \exp\left(-\frac{(\omega_n \tau)^2}{4}\right) \cos\left(\omega_n t\right)$$

The frequencies are $\omega_n = n\omega_0 = n\pi c/L$. In practice, we will take the sum over a finite number of modes over a finite number of time steps (the two should be ideally be linked, which you will learn about in Signal Processing, though here we will allow for the number of modes and time resolution to be specified independently).

1. First, write a MATLAB function that loops over all modes and all time points to calculate $u(t)$. Run the program using a total time of $T = 1.4$, $M = 200$ time points, $N = 200$ modes,

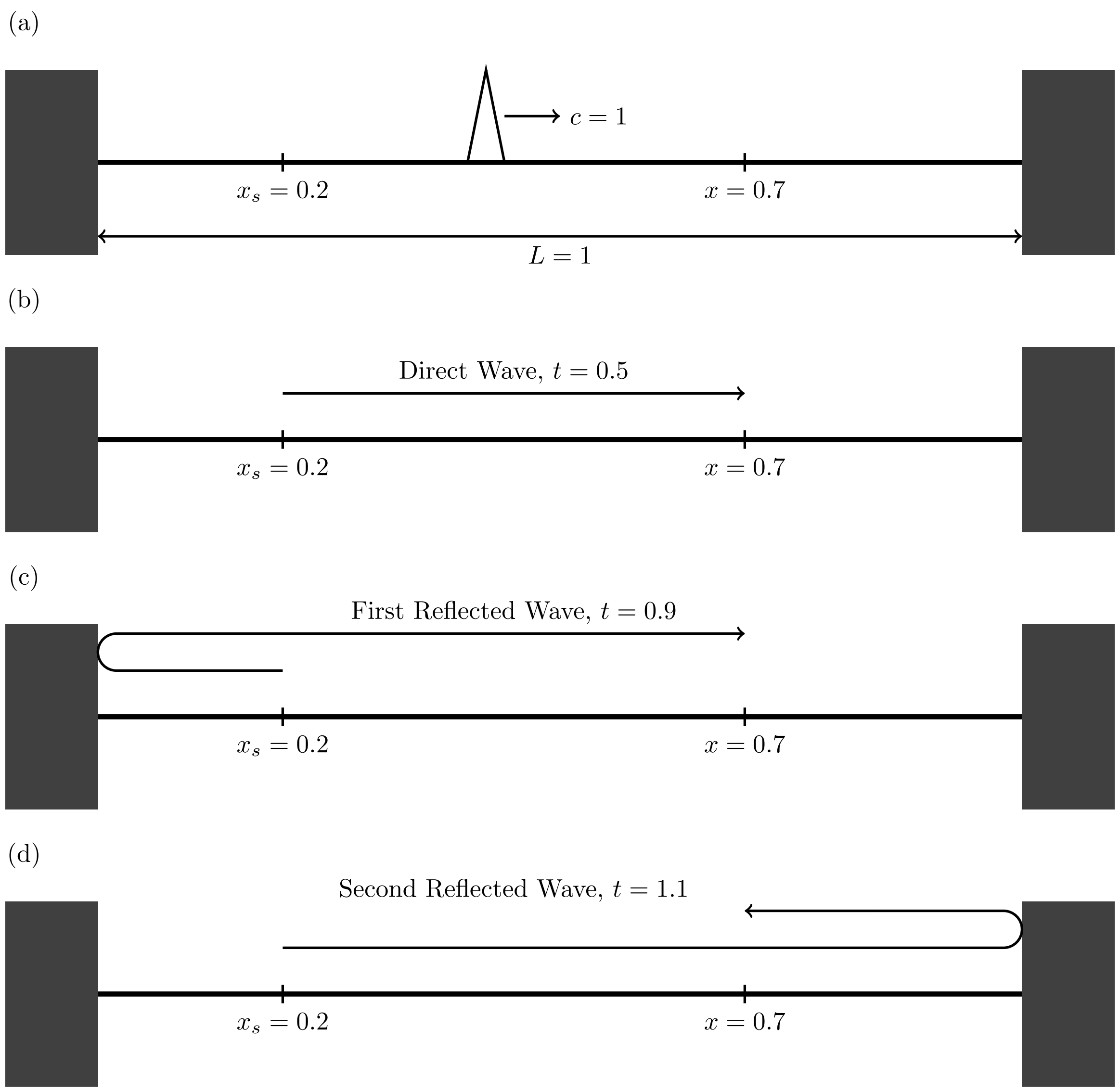


Fig. 10.1: Schematic for seismogram creation through a sum of normal modes. (a) A pulse is sent from the source point $x_s = 0.2$, and we would like to calculate the waves detected at a receiver at $x = 0.7$. The string has a length $L = 1$, and the signals travel at a speed $c = 1$. The ends of the string are fixed, so that a pulse arriving at the end is reflected with the opposite sign. (b) Direct arrival of the wave propagating directly from x_s to x. The signal should arrive at $t = 0.5$. (c) First reflected arrival, which reflects off the left boundary, and arrives at $t = 0.9$. (d) Second reflected arrive, where the direct arrival continues past x, reflects at the right boundary, and arrives at $t = 1.1$.

and $\tau = 0.02$. Time the execution using `tic` and `toc` as mentioned at the beginning of the lab. Note that it is best coding practice to do this as a function, which will make this code re-usable for different choices of the parameters. Think about what parameters you will need to pass to this function to make it as general as possible.

2. Now we will try to figure out how to "vectorize" this sum by writing it as a matrix multiplication problem. First, it is helpful to notice that when we compute the sum, the first three factors in the sum do not change as we change time points – they are only dependent on x, x_s, and τ, which are constants. Therefore, we can imagine collecting this term into a "time-independent" vector of length N. Define a vector that represents the x, x_s, and τ factors in a MATLAB function (if you want to compare the time for executing each of these, you will want to write a separate function), and set it to the appropriate values by using array evaluation (i.e. without looping over n).

3. The remaining issue is to break down the time dependence. Note that for a single time point t_0, we are summing $\cos(\omega_n t_0)$ weighted by the time-independent vector over N modes. This can be written as the product of a length N row vector representing the values of $\cos(\omega_n t_0)$ and the length N column vector for the time-independent factors discussed above. Write another MATLAB function, eliminating the loop over the different normal modes, so that you are doing a loop over all time points and doing a matrix product during each execution of the loop. Note: you will need to redefine your row vector representing $\cos(\omega_n t)$ every time step to accommodate the changing value of t.

4. Now that we have vectorized the normal mode sum, it should be clear that instead of redefining $\cos(\omega_n t)$ at every time step, we can make this into an $M \times N$ "time-dependent" matrix, where M varies over all t and N varies over all n. This will reduce the summation problem into a single instance of matrix multiplication, which MATLAB can carry out very quickly.

5. The one remaining issue: how do we define the $M \times N$ time-dependent matrix without looping over the whole matrix? Putting our linear algebra thinking caps on again, we realize that an $M \times N$ matrix is really just the product of a length M column vector and a length N row vector – the $\omega_n t$ factor can be written as (M time points as a column vector) $\times$ (N modes as a row vector). MATLAB can then easily take $\cos()$ of a matrix.

6. Finally, implement a function to make a vectorized sum to calculate $u(t)$ for the same parameters mentioned above. Time the execution, and compare with the looped computation. Your time series should look something like the plot below.

One important thing to note here: what we did here was to optimize how MATLAB computes this sum, but all that we did was eliminate the loops to give us code that does the same computations in a shorter time period. We still added up the contributions of N modes for M time points in the end, so we didn't save the computer any computations. While you should do this sort of optimization as much as possible, a better (and more clever) way to optimize this problem is to recognize that we are simply doing a Fourier transform of the source function, doing a simple multiplication, and then doing an inverse Fourier transform. Why is this important? Because some smart people in the 60's figured out that you can compute a Fourier transform on the computer faster than our matrix multiplication (though mathematics titan K. F. Gauss apparently knew about this algorithm in the early 1800's). Thus, one can do this calculation faster than our method here using Discrete Fourier

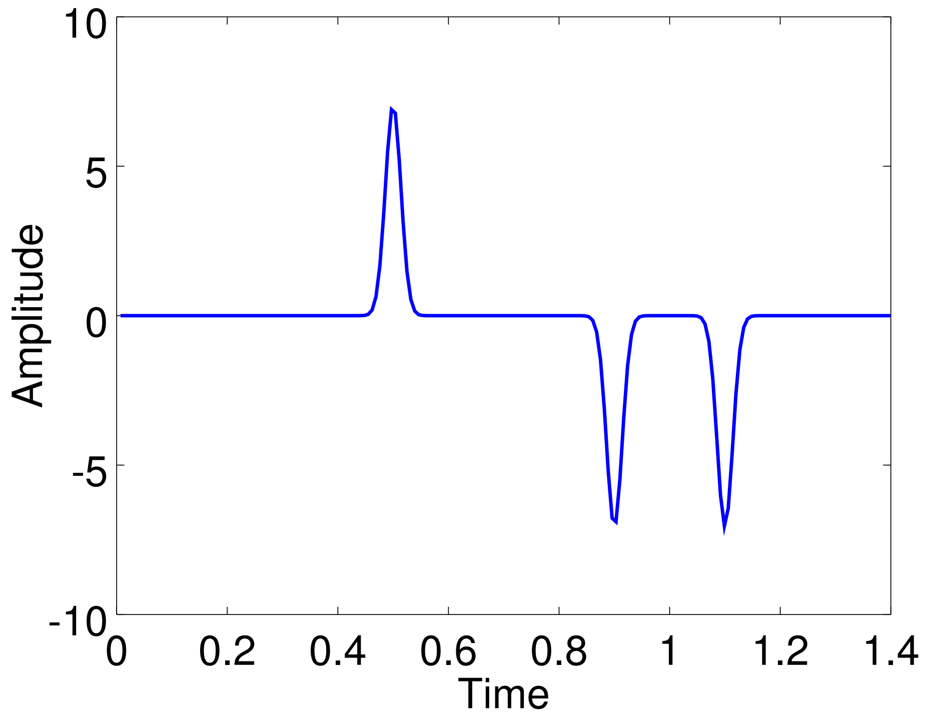


Fig. 10.2: Response calculated for a source at $x_s = 0.2$ of duration $\tau = 0.02$ at point $x = 0.7$ for $N = 200$ modes. The arrival times correspond to the expected values based on the diagram above.

Transforms. This is to serve as a reminder that while you can make your MATLAB code more efficient by vectorizing certain computations, you stand to gain even more by coming up with a way to do less computation in the first place.

CHAPTER

ELEVEN

MATLAB 5

This lab covers odds and ends regarding MATLAB.

11.1 Floating Point Numbers

We have been dealing with floating point numbers in both Python and MATLAB. These are computer approximations to real numbers, and most of the time they behave just like real numbers. However, there are some cases when treating floating point numbers exactly like real numbers can trip you up. This section talks a bit about the details of floating point numbers, and potential pitfalls that you may encounter if you treat them exactly like real numbers.

To represent a floating point number, we use a specified number of bits to represent the number. These bits are divided between the sign (1 bit), a fixed number of decimal places, and a fixed number of bits for the exponent, akin to scientific notation. Because this is on a computer, floating point numbers use base 2 instead of base 10. Thus we can think of a floating point number f as follows:

$$f = \left(1. + \sum_{n=1}^{p-1} \mathrm{bit}_n 2^{-n}\right) \times 2^m$$

To maximize precision, floating point numbers are normalized so that there are no leading zeros (by recalibrating the exponent), represented here by the 2^m here. Since in base-2 the leading digit of a normalized number will always be a `1`, we can drop this implicit bit to improve precision.

A "single precision" or 32-bit floating point number will be represented by the following: 1 sign bit, 8 exponent bits (wtih an implicit offset of 127 to represent negative exponents), and 24 mantissa bits (including the implicit bit). For example, π can be written as a single precision number as follows. First, write π to 32 digits in base-2:

$\pi =$ `11.001001000011111101101010100010`

First, we normalize by dividing by 2^1, and keep 24 digits in the mantissa:

$\pi =$ `1.00100100001111110110101` $\times 2^1$.

The exponent is expressed as 1 + 127 = 128, or in binary `1000 0000`. The sign bit is zero, and we drop the implicit bit, leaving us with the following (using SEEEEEEEEMMMMMMMMM-MMMMMMMMMMMMMM notation):

π = `01000000001001000011111101101`.

Single precision numbers give you about 7 significant figures represented in base-10, and an exponent range from -38 to 38. That is fine for many applications, but serious numerical work usually requires more, which is why double precision (64-bit) is standard in MATLAB. Double precision numbers give you about 16 significant figures, with exponents ranging from -308 to 308. One can also use quad precision (128-bit) floating point numbers, which have an additional number of bits dedicated to both the exponent and mantissa. The properties of floating point numbers in various precision are summarized in following table.

Type	Sign	Exponent	Mantissa	Total bits	Exponent bias	Bits precision	
Single	1	8 ($\pm$38)	23	32	127	24_2	7_{10}
Double	1	11 ($\pm$308)	52	64	1023	53_2	16_{10}
Quad	1	15 ($\pm$4965)	112	128	16383	113_2	34_{10}

11.1.1 Problems with Finite Precision

The fact that numbers are represented by a finite number of bits has a few important implications for arithmetic and other things that you need to do with numbers on a computer:

- Adding two numbers of very different magnitudes will often not yield the result you intended if the smaller number does not register within the precision of the larger number. To see this in MATLAB (which uses double precision, so we need to add something smaller by at least 10^{-16}), enter the following:

  ```
  a = 1.;
  b = a + 1.e-17;
  if a == b
      disp('a and b are the same');
  end
  ```

  You can mitigate this problem by increasing the precision of both numbers, as after doing so the smaller number will then register within the significant figures of the larger numbers.

- Subtracting numbers that are nearly the same will give a number that appears to be more precise than it actually is. In MATLAB, try the following:

  ```
  a = 1. + 1.e-15;
  b = a - 1.;
  disp(b);
  ```

  This will display mostly meaningless digits, because these digits are smaller than the smallest digit represented in `1. + 1.e-15`. Increasing the precision does *not* alleviate this problem, as increasing the precision adds on meaningless digits to both numbers.

- Testing for equality between floating point numbers often not give you the expected results because of rounding errors. When comparing floating point numbers `a` and `b`, it is best to compare `abs(a-b) < epsilon`, where `epsilon` is a small number set by the precision of the numbers. For an example of this, try the following in MATLAB:

  ```
  a = 0;
  delta = 0.01;
  for i=1:100
      a = a + delta;
  end
  if a == 1
      disp('a is one!');
  end
  ```

  This will not print anything out, because there is round-off error adding in up 0.01 (which cannot be exactly represented with a finite number of digits using double precision, despite its simple base-10 representation) 100 times. You should always use some small tolerance when comparing floating point numbers!

These are things to be aware of when doing math on computers. However, it is important to note that while round-off error is always something to worry about, it is not unpredictable. That is, if we do the same thing multiple times, we get the same result. For instance, when adding up a bunch of small numbers we may not get exactly the sum we expect, but if we subtract those same numbers off of the result, we *will* get the same number we started with:

```
a = 1.;
delta = 0.01;

for i=1:100
    a = a + delta;
end

for i=1:100
    a = a - delta;
end

if a == 1.
    disp('a is one!');
end
```

This will in fact print out something. There may be round-off errors owing to the fact that 0.01 does not have an exact representation as a double precision floating point number, but adding and subtracting the double precision representation of 0.01 does leave us with the same number we started with.

11.2 Saving Data to a File

We spent some time in the first MATLAB class covering how to get data into MATLAB. However, you are also likely going to need to get data out of MATLAB, too. Here are your options when it comes to writing data to file

Let's say you are using MATLAB and have the results of a particularly long calculation that you would like to write to disk for future use. There are a few ways that you can accomplish this, depending on the intended use of the data.

- **MATLAB native .mat files:** MATLAB's default method of saving data is in binary .mat files through the `save` command. You can save more than one variable to the same file – the default use of `save` is to save all workspace variables to the same file (done by invoking `save(<filename>)`. To save only one variable, you can call `save(<filename>,'<variable>')`. MATLAB can also save .mat files as ASCII files, with a number of options for formatting the output. Use the `help` command to see all of the different possibilities. Note that this is saved in a MATLAB-specific format, so you need to use MATLAB to read these files (use `load(<filename>)`).
- **ASCII formatted files:** There are two options for non-MATLAB-specific saving human-readable files. `fprintf` is the equivalent of `fscanf` and can be used to write formatted data to a text file. `fprintf` takes 3 arguments: a file id, formatting instructions, and the data to be written. For example, if I have a vector `A` that is 10 elements in length that I would like to write to file with each entry on its own line, I can use the following:

```
fid = fopen('outfile.dat','w');
for n = 1:10
    fprintf(fid,'%f\n',A(n));
end
fclose(fid);
```

  You probably recognize the `fopen` command from before; here it is important to include the `'w'` so that MATLAB will open the file in write mode. `fprintf` includes the file id, the formatting (in this particular case, we print as a floating point number, followed by a carriage return (`\n`), and the data to print. If you want to put a tab after printing a character, use `\t` instead of `\n`, or you can add spaces as well. You can specify the number of digits to write by using `'%m.nf'` for the formatting string, where `m` is the minimum number of characters to print and `n` specifies the number of figures to include after the decimal point. Thus, `'%5.3f'` will print a minimum of 5 total characters (it will be padded with blank spaces at the left if need be), three of which are after the decimal point.

  An alternative is to use `dlmwrite`, which writes character delimited formatted text to a file. `dlmread` is another option for reading data into MATLAB that I did not mention the first time around – it is used to read data that has a common character that separates every data point. A comma is the default delimiter (as CSV files are a fairly common format), but an alternative delimiter can be specified when invoking the command. An example use is dlmwrite(<filename>,A);. See the documentation to learn more about the specific options.

- **Binary files:** To write to a file in binary format, use fwrite, which is the writing equivalent of fread. To write the matrix A to disk as a binary file as a double precision float with a little endian byte ordering, use

```
fid = fopen('outfile.dat','wb');
fwrite(fid,A,'double','l'); % you can specify other formats and␣
↪byte-ordering values if necessary
fclose(fid);
```

  The only thing to note here is the use of the 'wb' tag to specify that the file should be opened in binary write mode. Everything else should be familiar from the use of fread.

If speed is a concern, writing binary files (using either save or fwrite, depending on the intended use of the data) is much faster than writing formatted files, and will take up less disk space. If you will be writing lots of data at full precision and the files do not need to be human readable, consider using one of the binary options.

11.3 Additional MATLAB Data Types

11.3.1 Data Structures

Sometimes in geophysics we have more complex data than can be simply represented by vectors or matrices. For instance, a seismogram often contains three components, each of which has the same length. There are multiple ways we might handle this in a MATLAB code:

- Three separate vectors, one for each component
- One matrix, with three columns (one for each component)

Either method will work, but they both have their drawbacks. If each component is its own vector, then we can use descriptive names to describe each one (i.e. station1_N, station1_E, station1_Z), which makes our code more understandable. However, this does not highlight the fact that all of these vectors represent different aspects of the same measurement, since each variable is separate. Further, if we want to write a function to process the components, we need to pass all three vectors to the function as separate arguments. Using a matrix solves our problems, but now it is less clear what each component is and our code is harder to understand.

We can get the best of both worlds by using a data structure, which is a single variable that holds multiple data objects, each of which has their own name. One way we might create a data structure for a seismogram is as follows:

```
station1.N = [ <array> ];
station1.E = [ <array> ];
station1.Z = [ <array> ];
```

You would obviously create the appropriate vectors for assigning each component (by reading the data from a file, for instance). This gives you a single data object `station1` that makes it clear that this is a single measurement and can be passed to functions in a simple fashion. You can also add additional data to the structure, like the time (a vector), and even other attributes that might not even be vectors, like station latitutde/longitude, sampling frequency, and the instrument type:

```
station1.t = [ <array> ];
station1.latitude = 36.;
station1.longitude = -90.;
station1.samplerate = 100.;
station1.type = 'broadband';
```

This is a handy trick for writing code that is clear and easy to understand, as you can wrap all relevant data into a single structure. If you want to initiate an empty structure, use `struct()`, to which you can then add entries as above. To initialize a structure with multiple entries at once, use `struct('<fieldname>',<fieldvalue>,...  )`. Because of how MATLAB stores the field names internally, you need to explicitly use a string for the field name when initializing in this manner.

11.3.2 Cell Arrays

MATLAB arrays work well for numeric data, and structures are handy for combining multiple data types into a single object. But what if you have data that is a collection of a large number of strings? A structure is not really useful for this, because we would need to store every string as its own unique entry, which obscures the structure of the data. This is also cumbersome for accessing the data, since you would need to use a string to access each entry, rather than a convenient numeric index.

We might think that we can simply use an array of strings to represent the data in a convenient way, using the same syntax for defining an array of numbers:

```
stringarray = ['abc' 'def'];
disp(stringarray(1));
```

However, this will simply print out `a`, which is not what we really wanted. This is because a string is already represented within MATLAB as an array of characters. When we defined `stringarray`, we were really just combining two arrays of characters, much in the way MATLAB lets you concatenate vectors or matrices using `vectorcat = [vector1 vector2]`.

To handle arrays of strings, MATLAB provides the cell array datatype. Cell arrays use curly braces `{}` for defining and indexing the arrays, rather than the normal square brackets (for defining) and parentheses (for indexing) arrays. Cell arrays can hold any data type in each entry, including strings, booleans, numbers, and even entire matrices, so you are not restricted to numeric values like in a matrix.

Here is an example cell array holding the data from `stringarray` above:

```
stringarray = {'abc' 'def'};
disp(stringarray{1});
```

This will give us the desired result. As mentioned above, we can put any type of data in each entry of a cell array, for example

```
mixedarray = {'abc' 123 [1 2]};
disp(mixedarray{1});    % displays 'abc'
disp(mixedarray{2});    % displays 123
disp(mixedarray{3});    % displays 1 2
disp(mixedarray{3}(1)); % displays 1
```

This allows us to combine different datatypes into a single array that can be indexed like a matrix.

11.4 Exercises

1. Practice saving data in various formats from your workspace. Check that you were successful by reading the data back into MATLAB using the equivalent read commands from our previous work with MATLAB.
2. Create a data structure representing an earthquake catalog (you can use the file "newmadrid2015.txt" file from the second class on MATLAB as input data). Include separate fields in the data structure for the date, latitude, longitude, and magnitude.
3. I have provided 1000 binary .mat files zipped together in the file "lab11files.tar.gz", on the course website. Use a cell array to read all of them into MATLAB. *Hint:* Use the shell to write all the file names into a text file, then read that file into MATLAB using `fscanf` to define your cell array, and then loop over the cell array and use the `load` command to read each one into MATLAB.

CHAPTER

TWELVE

SEISMIC ANALYSIS CODE 1

One tool that geophysicists use to process seismic data is Seismic Analysis Code (SAC). SAC is installed on all of the computers in the Mac Lab, and can be obtained for no charge from IRIS (Incorporated Research Institutions for Seismology, which is one of the main consortiums for archiving and distributing seismic data) if you are affiliated with an IRIS institution. SAC has many seismic processing functions, makes several kinds of plots, and can read and write ".sac" data files (more on this later). SAC has been around for a while, which means it has a few limitations based on its age, but it is still used enough that geophysicists should be familiar with it.

SAC capabilities include many tools commonly used when processing seismic time series data: arithmetic, Fourier transforms, integration/differentiation, spectral estimation, filtering, stacking, decimation and interpolation, correlation, phase picking, instrument correction, vector component rotation, envelope creation, frequency/wave number analysis, and plotting. While we will not be covering all of these in this class, if you have a working knowledge of how to use SAC, you can figure out how to use any of these tools without too much trouble.

There are many alternatives that replicate the functionality of SAC, so while you should know how to use SAC, you may want to explore some of these other options if your research involves processing seismic data. First, there is a MATLAB toolbox MatSeis that has many functions useful for seismic data analysis. There also are many individual MATLAB functions that people have written to perform many of the operations in SAC. You can of course also write your own MATLAB programs to process seismic data. As a Python fan, I like using the Python package ObsPy to do many of the same things that SAC does (I believe that ObsPy is installed on the computers in the Mac Lab). And for serious data processing, there are many C and Fortran codes available to perform various tasks. Talk to other graduate students to find out how they process data.

12.1 Running SAC

Unlike MATLAB, SAC does not have a nice GUI. You run SAC from the Unix shell by typing `sac`, followed by a carriage return. The Terminal should now look something like this:

```
SEISMIC ANALYSIS CODE [11/11/2013 (Version 101.6a)]
Copyright 1995 Regents of the University of California

SAC>
```

We now have a command prompt within SAC to enter commands. To quit SAC, type either `exit` or `quit`, followed by a carriage return. This will return you to the Terminal prompt.

We will be reading and writing files from within SAC. By default, SAC reads and writes files from the current working directory. You will probably want to create a separate folder for all of your SAC work in this class. One method is to create a directory `SAC` in the `Documents` folder, though you are welcome to arrange things however you like.

12.2 SAC Basics

SAC is used to analyze time series data. Like MATLAB, SAC is interactive and command-driven (either by entering commands directly in the terminal, or by executing commands saved in a script), so you need to give it commands for it to do anything. SAC commands can be typed either as upper case or lower case (commands are all converted to upper case before they are interpreted by SAC). However, if you are giving SAC file names to read from or write to, SAC interprets the case of those statements literally. There are also abbreviated names for many functions to save you time in typing them. I will generally use all uppercase and give the full function name when introducing a function, but lowercase when making subsequent references to it. I will mention the abbreviated names as well, but will stick to using the full names in the notes.

SAC commands are fall into three different categories: (1) parameter-setting, which changes the values of the parameters within SAC, (2) action-producing, which carry out actions based on these parameter values, and (3) data-related, which determine the data in memory that these actions will be performed on. Note that this means that commands that set parameters appear to do nothing until you execute the corresponding action-producing command.

Additionally, there is the `HELP` command, which lets us read the documentation on any of these commands. `help` without any additional arguments brings up a list of all commands. `help <command>` brings up the documentation page for that particular command. To exit the documentation and return to the SAC command line, type `q`.

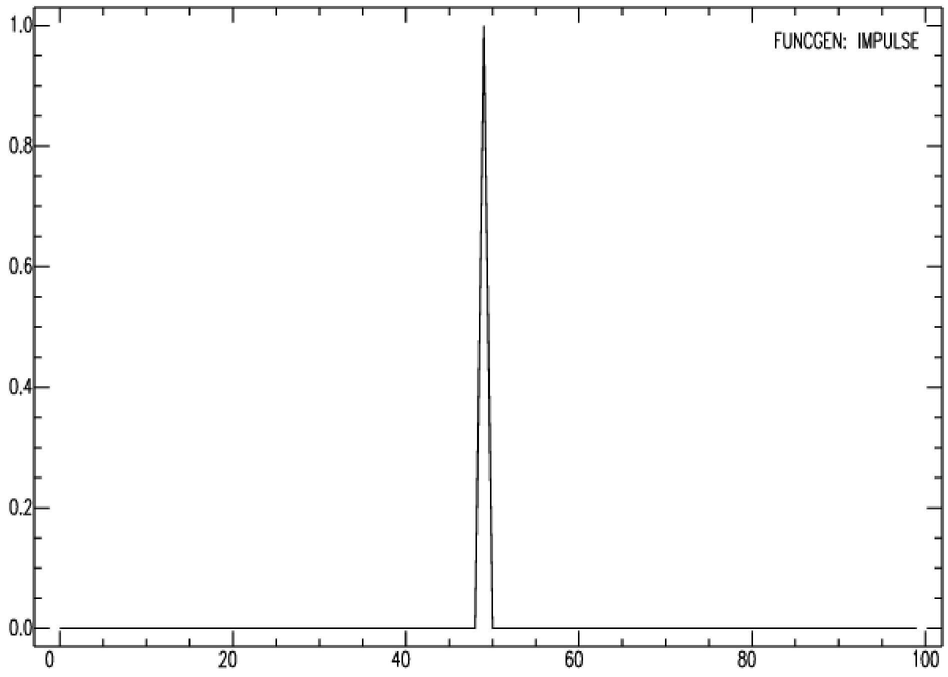
12.2.1 FUNCGEN

To put a basic time series into SAC, use the `FUNCGEN` command. This can be entered in all caps, in all lowercase, or with the shorthand `fg`. If you simply type `funcgen`, followed by a carriage return, SAC will create a default time series for you. There are additional options that can be specified (we will get to those in a second), but this is a basic way to create data within SAC.

After you type `funcgen`, it is not clear that anything has happened. To see the results of the `funcgen` command, we need to be able to plot the data and look at it.

12.2.2 PLOT

To plot the current data that is in memory, we use the `PLOT` command, or `p` for short. Try this with the function that you created using `funcgen` – the plot should show up in an external window, and look something like this:



To change the limits on a plot, use the `XLIM` and `YLIM` commands. The syntax is `xlim <min> <max>`, then issue the plot command (remember, SAC commands to set parameters do nothing until you enter a corresponding action-producing command). Also note that this will affect all future calls to `plot`, even if you generate new data in SAC. To turn off axis limits and return to the default behavior, enter the command `xlim off`.

12.2.3 Some Details

Now that we can create functions and plot data to visualize, here is a bit more about what is going on. When we invoke `funcgen` with the default parameters, SAC creates an impulse function (also known as a delta function) in memory. By default, time ranges from 0 to 99 with 100 data points, and the impulse occurs at $t = 50$. The function is one at the time of the impulse, and zero otherwise.

When we plot the data, SAC creates a plot in an external window. The plots are relatively simple, and reflect the legacy of SAC being originally used on Tektronix 401X graphics terminals:

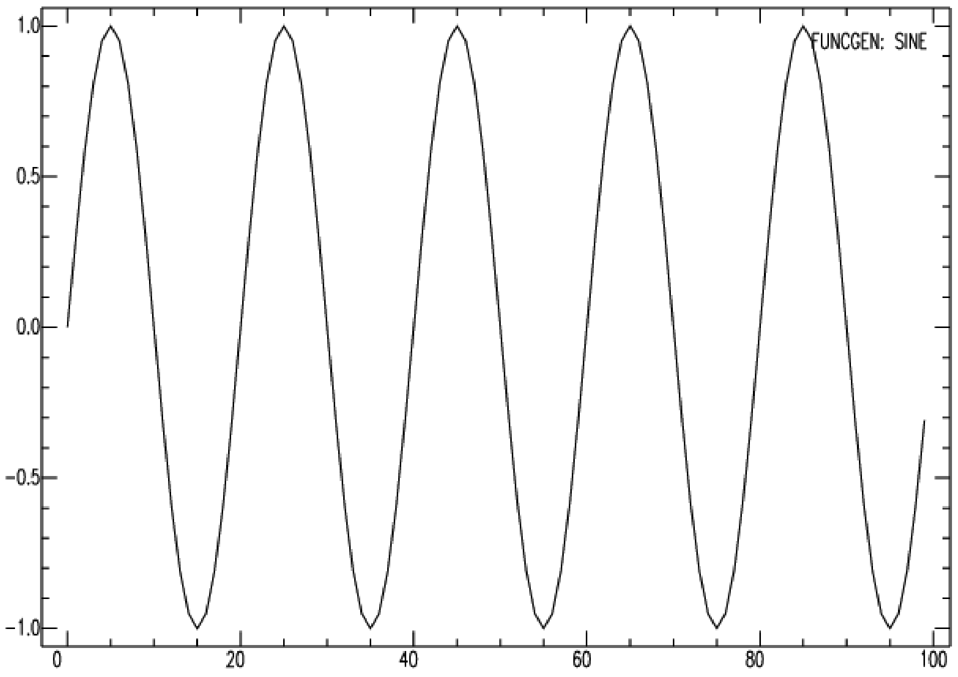
These 1980's era terminals were state-of-the-art when SAC was developed at Lawrence Livermore National Laboratory, and so some limitations of the graphics remain in the present version of SAC – in particular, SAC will often desample a time series to plot it more quickly, which was a necessity on the Tektronix system, but not a real concern on modern computers. However, we will see that many of these limitations have seen improvements in the modern version of SAC.

12.2.4 FUNCGEN (again)

Now that we know how to view our results, we can play a bit more with the `funcgen` command. Look at `help funcgen` to see some of the additional options. In addition to the default `impulse` function, we can also generate `step`, `boxcar`, `triangle`, `sine`, `line`, `quadratic`, `cubic`, `seismogram`, `random`, and `impstrin` (multiple impulses). Some of these have additional parameters; for example, for `sine` you specify two parameters (the frequency and phase), for example:

```
SAC> funcgen sine 0.05 0
SAC> plot
```

will produce the following graph:



Other parameters that you can specify when calling `funcgen` (regardless of the type of function you are creating) include the number of points `npts`, the time resolution `delta`, and the beginning time `begin`. The default values of these parameters are `npts = 100`, `delta = 1`, and `begin = 0`. To call `funcgen` with additional parameters, use `funcgen impulse npts 100 delta 1 begin 0`. There are also default values for all of the functions that take parameters. Once you set a parameter to a different value, that becomes the default value for subsequent calls to `funcgen`, unless you specify that parameter again.

Exercise: Try out some of the other functions available in SAC, in particular some of the functions that require additional parameters like `sine` and `random`. Also produce functions over differing time window using `npts`, `delta`, and `begin`. Make a plot of each function to verify that you know how `funcgen` works.

12.3 READ, WRITE, and ".sac" Files

One thing you may have noticed is that when you create a new function in memory, you lose all ability to access the previous function. However, modern seismology often requires processing multiple signals simultaneously. How can we do this in SAC?

In order to get multiple signals into memory in SAC, we need to read data from a file using the `READ` command (`r` in shorthand). SAC has its own data format, and is one of the many standard data formats that you will find for waveform data. Other formats include “ah” (Ad-Hoc, the data format of the IRIS/PASSCAL program), “SEED” (Standard for the Exchange of Earthquake Data, standard format of the IRIS Data Management Center), “SUDS” (Seismic Uniform Data System, developed by the USGS), “SEG-Y” (the standard for seismic reflection data), and others that continue to pop up from time to time. There are additional software packages for many of these additional formats that fulfill many of the same functions as SAC.

SAC files all have a header that displays information about the data in the file. Once you read in the data, you can access the header information using the `LISTHDR` command, which can be abbreviated with `lh`. The header for a seismic data file will usually include such information as time ranges, instrument details, and any seismic phase pick information that is present in the file. We can also look at the header of files that we create within SAC – these typically only include time information and the type of function, but are still useful if you don’t remember the details of how you created the data that you currently have loaded in memory.

However, what if we don’t have our data in a file, but want to generate it within SAC? Then we use the `WRITE` command (shortened to `w`). Once we have created our function, we call `write <filename>`, and SAC writes the function to a .sac file with the specified name. Then we can read the file back into memory using `read <filename>`. When you call `write` with the filename of an existing file, it overwrites that file without a warning, so you need to be careful when using `write`.

Exercise: Generate a function using `funcgen`, write it to disk using `write`, and then read it back into SAC using `read`, verifying that you have succeeded by plotting the data.

Why go to the trouble of writing the signal to disk? Now that we can read and write data, we can get multiple signals into SAC using the `read` command, followed by multiple filenames: `read file1.sac file2.sac file3.sac`. This will load all three signals into memory, allowing you to process and plot the signals. Alternatively, you can load files one at a time without overwriting the data presently in memory using `read more <filename>` (you can call `read more` with no files loaded in memory, so it is okay to call it without having already some files in memory).

Exercise: Create three functions using `funcgen`: an impulse, a boxcar, and a triangle. Save each one to disk, and then read all three into SAC. Plot the functions. Notice that when you use `plot` with multiple functions in memory, it plots them one at a time, waiting for you to hit return to see the next one.

12.4 Obtaining ”.sac” Files from Public Data Sources

While you can learn to use SAC with data generated within the program, it is far more interesting to use real data from real earthquakes. One such source if data is the IRIS Data Management Center, which has a nice web interface called “Wilber 3” for finding earthquakes and downloading waveforms. Go to the Wilber 3 website at http://ds.iris.edu/wilber3/find_event to take a look at recent earthquake recordings that are available to download.

Click on an event from the map or the list (by default, it shows events for the past 30 days, but there is a drop-down menu above the map that allows you to change what is displayed). From there, it will go to a map showing the epicenter and the location of stations. By default, it shows only broadband channels from the Global Seismic Network, but you can change the options at the right of the map. Below the map, you can select the stations that you want to download using the checkboxes at the left of the table. Click on "Request Data" to download the selected stations (you will need to select the desired time windows and data format; choose "SAC binary (little endian)" for use with SAC on the Mac Lab computers). You should receive an email link that allows you do download the data.

Exercise: Once you have some real waveforms, try loading them into SAC and plotting them as above. You may find that it is a pain to type in the full filename when reading the data, but SAC recognizes the same wildcards as the shell, so you can use wildcards to reduce the amount of typing that you do when loading multiple files into memory.

12.5 Plotting Multiple Functions

What if we want to plot several functions on the same graph? SAC has additional plotting commands that handle multiple types of data. `PLOT1` (`p1` when shortened) plots multiple signals on separate axes with a common time axis for all signals. If the three signals have different time ranges, SAC extends the time axis to be large enough to show all signals. Each graph has its own unique amplitude limits. However, if you would like to give each plot its own time axis, use the `relative` command (as opposed to the default `absolute`). If you have three data files in memory and type `plot1 relative` into the command terminal, then it will give each plot a separate time axis that corresponds to the limits of the data.

We can use `PLOT2` (`p2`) to plot multiple signals on the same axes, and the time and amplitude limits are chosen to be large enough to show all points. As with `plot1`, you can have the plot show relative values instead of absolute values.

With multiple plots (particularly `plot2`), we may want to have our plots displayed in color. To do this, we need to use the `color` command. `color` has several options. If we just want to change the color of the lines in the plot, use the `color <selectedcolor>` command (see the `help color` documentation to the available colors). This will make all future plot calls use this color for the lines on the graph. If you want different colors for each successive line that you plot, then use `color on increment on`, which cycles the colors for each line that is plotted.

Exercise: Use the seismograms that you just downloaded to make plots using `plot1` and `plot2`. First, read in all of the data files into SAC. Use the `listhdr` command to examine the header variables, which includes information on the instrument type, event, and the recorded signal. Plot the signals using both `plot1` and `plot2`. You will note that it is difficult to distinguish the traces on the overlay plot, so change your `plot2` to be a color plot to help you distinguish the waveforms.

Notice that in your data files, we have three components per station, which required several data files. This highlights one of the limitations of SAC – each file can only contain one trace, so multiplexed data needs to be stored in multiple files. This is one of the downsides of the .sac format, a particular problem given the amount of data available today. Fortunately, automating things with a computer makes it easier to handle many files, so with appropriate care we can handle this efficiently and reliably and continue to take advantage of the many functions available in SAC.

CHAPTER

THIRTEEN

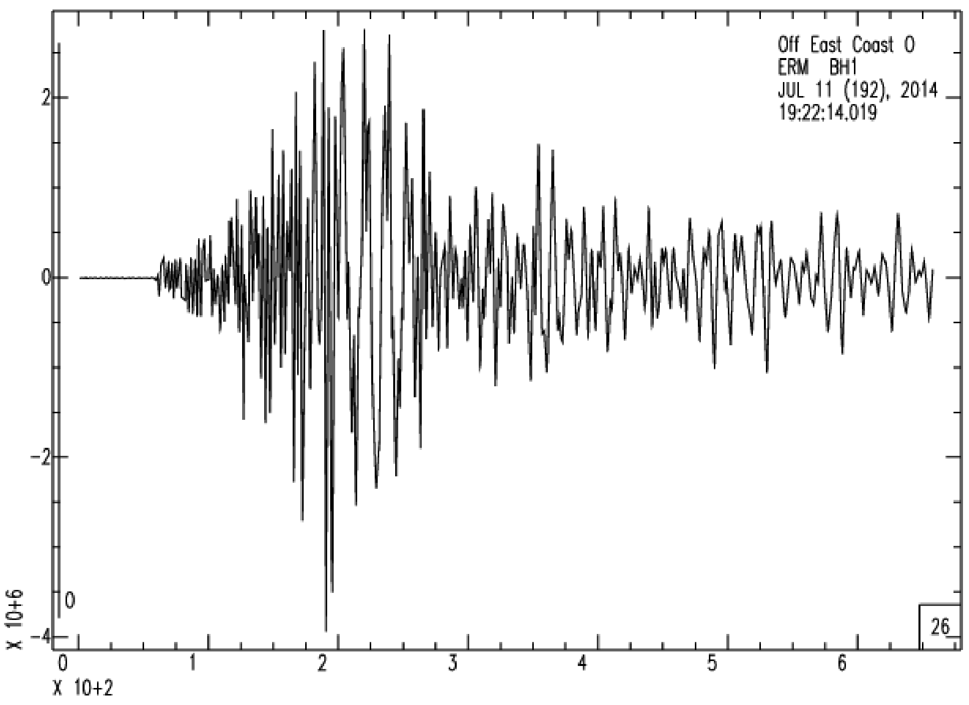
SEISMIC ANALYSIS CODE 2

This lab focuses on additional material relating to graphics in SAC, and how to write SAC macros (which are essentially scripts to carry out a series of SAC operations, allowing you to conduct more complex computations using SAC).

13.1 Details about SAC Plots

13.1.1 Quick and Dirty Plots

Run SAC on your computer, and load one of the seismograms that you downloaded for the previous class, and plot the function. Your plot should look something like this:

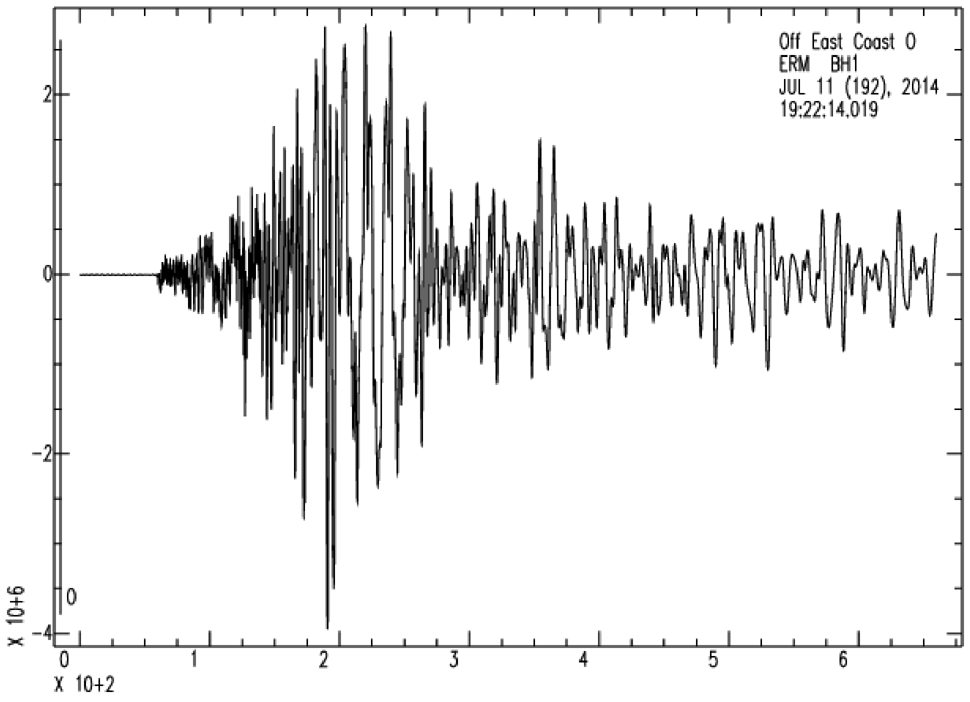


Notice the box in the lower right hand corner– for my plot, it says “26” but depending on the length of the signal that you plotted you will see a different number. This number signifies that SAC is only plotting every 26th point in this graph.

Why does SAC do this? On the original Tektronix system, the data transfer rate is painfully slow, to the point that displaying a plot of seismic data at a typical sampling frequency would take several hours. Thus, by default SAC desamples long time signals before plotting them in order to reduce the time needed to transfer the data to the display.

Unfortunately, this is not implemented very well. When SAC performs this operation, it simply desamples the data, taking very few points. This is unlikely to give accurate information about the amplitude of the signal (because the probability that the maximum value of a particular wiggle is displayed is only 1/26), and means that the plots are not particularly useful for analysis (particularly for very long signals with many data points, which would require even more aggressive desampling). A better way to do this would be to take the maximum value within every desampling window and plot that value, making the signal amplitudes useful (referred to as decimation). While it requires an additional processing step, the time required to do this extra processing step is minimal compared to the data transfer time on the original Tektronix display.

Fortunately for us, since we have modern computers with modern displays, we do not have to worry about long times to display signals. To prevent this behavior in SAC, we need to turn off the "quick and dirty plot" option (which is set to "on" by default. If you type `qdp off` into the command line and then plot your data again, you will see a plot with no number in the corner:



This plot has information that accurately reflects the signal amplitude (and notice that the display time was not noticeably different than before), though for us the amplitudes aren't markedly different. If you are going to use SAC plots for any real scientific purpose, turn off the quick and dirty plots!

13.1.2 Labeling Axes and Titles

To label axes in SAC, use the `XLABEL` or `YLABEL` command. The syntax is `xlabel on '<label>'` for the x-axis (similar for the y-axis. For example, if I wanted my x-axis to be labeled as 'Time (s)' I would type `xlabel on 'Time (s)'` into the terminal. As you may remember, this will have no effect on the current plot until you issue the `plot` command again, so you need to issue these commands before plotting for them to take effect.

To add a title, use the `TITLE` command. It is issued in the same way as the `xlabel` command, so to title a plot, type `title on 'Seismogram'` to add the title 'Seismogram' to your plot.

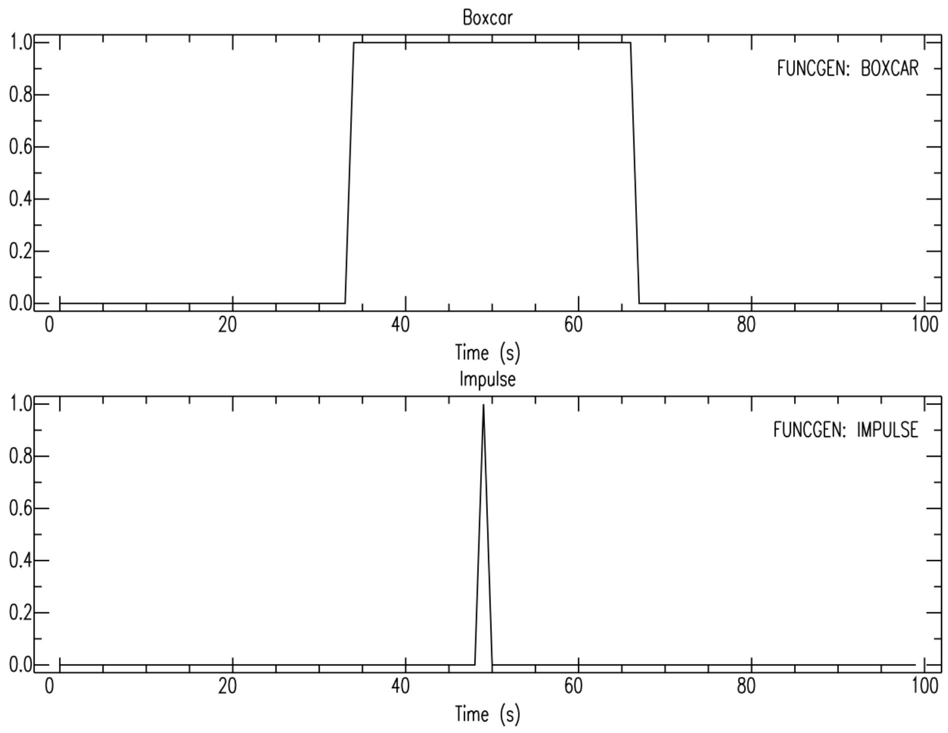
Again, you will not see any effect from the `title` command until you issue the plot command.

13.1.3 Changing the Plot Appearance

We have already discussed how to change the limits on a plot and how to make a color plot, but there are additional customizations that are possible. To change the spatial size of a plot, use the `XVPORT` and `YVPORT` commands. The syntax is `xvport <min> <max>`, where `<min>` and `<max>` are numbers between zero and one that specify the minimum and maximum values of the axes as a fraction of the entire plot window. The default values of `xvport` are 0.1 and 0.9, and the default values of `yvport` are 0.15 and 0.9. To make a smaller plot, I would issue the commands `xvport 0.1 0.5` and `yvport 0.15 0.55` and then the plot command.

To put multiple plots in the same figure, we use the `xvport` and `yvport` commands in conjunction with the `BEGINFRAME` command (shortened to `begfr`). When you enter the `beginframe` command, SAC will add a new axes to the existing figure when subsequent plot commands are issued, rather than making a new figure. Thus, once we enter the `beginframe` command, we can add plots in specific locations by setting `xvport` and `yvport` and then giving the plot commands. When the `ENDFRAME` command is issued, the default behavior resumes. An example to make a plot with two axes is:

```
beginframe
xvport 0.1 0.9
yvport 0.15 0.475
funcgen impulse
title on 'Impulse'
xlabel on 'Time (s)'
plot
xvport 0.1 0.9
yvport 0.575 0.9
funcgen boxcar
title on 'Boxcar'
xlabel on 'Time (s)'
plot
endframe
```



13.1.4 Saving Plots

So far, we have been making plots and displaying them on the screen. What if we want to save a plot for use in another document?

The simplest way to save figures from SAC is to use `SAVEIMG` to directly output a figure window to a standard graphics format. This command is flexible and will let you save the currently displayed figure window to PDF, PS, or PNG formats without doing any additional work. Once you have a figure up and displayed, simply enter `saveimg <filename>.<format>`, where `<filename>` is your desired name for the file and `<format>` denotes the extension of the graphics format that you wish to use. Thus for example, to plot an impulse function and save as PDF, you would do the following:

```
funcgen impulse
title on 'Impulse'
xlabel on 'Time (s)'
plot
saveimg impulse.pdf
```

This has the advantage over the more complicated procedure described below in that the labels are rendered as text rather than as lines. Because of this, I would generally advocate using `saveimg` to produce plots.

However, if you are interested in more details about producing graphics in SAC, there is another procedure that gives you a bit more control. SAC has its own graphics format, known as ".sgf" (SAC graphics format). This format is analogous to the MATLAB ".fig" files as they only readable by SAC, and is not useful if we want to include the figure in other documents like presentations, posters, or papers. To save a plot in the ".sgf" format, we need to follow a few steps.

SAC has two different output devices. A device is just a fancy name for the destination where the graphics will be displayed. So far we have used the screen as our output device, but we have the ability to tell SAC to send our plots to a different device. To use the screen as an output, we use the `XWINDOWS` device (can be abbreviated `x`). You may have noticed so far that when we issue a plot command, the plot window shows up in an application called "X11" which is the xwindows application for a Mac.

The other available output device is `SGF`, or **SAC Graphics Format**. If we choose this output device, plots will not be displayed on the screen, but instead will be written to file. To change the output device, we use the `BEGINDEVICES` command (`bd` in shorthand). By default, the device is set to `xwindows`, but by typing `begindevices sgf`, all future plots will be saved to file as an ".sgf" file instead of being displayed on the screen. To switch the device back to the screen, type `begindevices xwindows`.

When you use the sgf device, each plot command that you execute will save to a new sgf file. The first plot after you start SAC will be saved to "f001.sgf" in your current working directory. Subsequent commands will save as a file with the name "f002.sgf," "f003.sgf," etc. If any file already exists (say from a previous SAC session), then it will be overwritten without a warning, so if you want to save your sgf files, you will need to rename them after each SAC session.

To convert an sgf file into a graphics format that is useful outside of SAC, we use the `sgftops` utility (note that I called `sgftops` a utility, as it is not an official command in the same sense that `plot` and `read` are commands). This converts an sgf file to PostScript, which is a standard graphics format that can be opened with Applications such as Preview on the Mac (when Preview opens a PostScript file, it automatically converts it to PDF), Adobe Illustrator, or on the command line using ghostscript. To convert a file, use the syntax `sgftops f001.sgf sacplot.ps` which converts the sgf file "f001.sgf" to the file "sacplot.ps" which can then be viewed in another application.

There are two additional options when using `sgftops` that you can include in the command. The first specifies the line thickness in the resulting PostScript file (the default value is 1), and the second allows you to specify the page layout or add an id that specifies some details about the plot. To add an id, make the second option `i`, to change the page layout specify `s`, or `si` for both. The `s` option is useful if you want your plot in portrait rather than the default landscape mode – this option will prompt you to enter specific values for the x and y translations (relative to the lower left corner of the page, assumed to be US Letter size in portrait orientation), rotation (measured counterclockwise from horizontal), and scale. An example, to obtain a plot with thicker lines type `sgftops f001.sgf sacplot.ps 1.5` into the command line. To rotate and scale the plot into portrait mode, use `sgftops f001.sgf sacplot.ps 1 s` and then at the prompt, type the following four numbers, each followed by a carriage return: 0.5 0.5 0 0.75. This will result in a plot in portrait mode.

As you can see, this requires considerably more effort than using `saveimg` without a huge difference in the final outcome. Because of this, I do not use the SGF procedure much any more, and simply use `saveimg` to save figures in SAC.

Exercises: Try out some of the plot commands described above. First, plot a seismogram with the quick and dirty plot option off. Then, plot a seimsogram with labeled axes and a title, save it as a PDF file. Compare with the results using an SGF file converted to PostScript. Finally, use `beginframe` along with `xvport` and `yvport` to illustrate four of the different functions created by `funcgen` in a 2x2 grid of plots.

13.2 SAC Macros

Like with MATLAB .m files, we can write scripts to carry out a series of commands, called "macros" in SAC. SAC macros are text files containing a series of commands that can be called either from within SAC or directly from the command Terminal.

For example, here is a series of commands. Enter them into a text file (using an editor of your choice), and then launch SAC. You can name the file whatever you like, but I prefer to include `.macro` at the end so that I know that the file is a SAC macro. From inside SAC, type `macro test.macro` (using the name that you chose for your file) to execute the macro. For example,

```
# SAC macro to generate three functions, write to disk, and plot
funcgen impulse
write impulse.sac
funcgen boxcar
write boxcar.sac
funcgen triangle
write triangle.sac
read impulse.sac boxcar.sac triangle.sac
plot1
```

# and * serve as comment characters in SAC, so that line will not be executed. However, you cannot put inline comments into SAC, so comments must be on their own line. Alternatively, you can run the macro from the Terminal by typing `sac test.macro` directly from the command line.

13.2.1 Input Arguments

While this type of macro is easier than typing a series of commands into the command line (where you are likely to make a mistake if you are not careful), macros are more useful when they can take input arguments like a function in MATLAB or Python. SAC macros do not have an explicit way to write functions, but you can pass an arbitrary number of arguments to any SAC macro, and the parameters will be available using the variables `$1`, `$2`, etc. As a simple example, if we wanted to write a macro to read three SAC files and plot them, change your macro to the following:

```
# SAC macro to read three files from disk and plot
read $1 $2 $3
plot1
```

If you type `macro test.macro impulse.sac boxcar.sac triangle.sac` into the command line. You should get the same plot as before. If you call the macro without specifying any arguments, you will get a prompt asking you to input the arguments.

What if you want to read an arbitrary number of SAC files? You can make the macro more flexible by specifying keywords, which can take an arbitrary number of arguments. Change your macro to the following:

```
# SAC macro to read an aribtary number of files from disk and plot
# files to read are set using the "files" keyword
$keys files
read $files
plot1
```

By typing `macro test.macro files impulse.sac boxcar.sac` into the command line, you can run this macro, and you can change the number of input files to whatever you like without changing the macro. If you do not give any values for `files`, SAC will prompt you to enter the values (Note: I have found that this does not always work on SAC installed in the Mac Lab, so you may end up getting an error if you do not provide the input arguments).

You can also specify default parameters. If you want your macro to plot impulse.sac, boxcar.sac, and triangle.sac by default, then change the macro to:

```
# SAC macro to read files from disk and plot
# files to be read are set using the "files" keyword
# if no files provided, default files are impulse.sac boxcar.sac␣
↪triangle.sac
$keys files
$default files impulse.sac boxcar.sac triangle.sac
read $files
plot1
```

Now you can run your macro with any number of arguments, or with no arguments to plot those three functions by default. (Note that I have also found that setting default parameter values does not always work correctly in the Mac Lab for some reason, so you may sometimes get an error when doing this in the Mac Lab.)

13.2.2 Variables

One of the valuable aspects of any programming language is the ability to assign values to variables. This can be done in SAC using what is known as a "blackboard" variable. You can set a blackboard variable using the `SETBB` command, followed by the variable name and the value. For example, `setbb xmin 40` will define a variable `xmin` with the value 40. To access the variable, use `%xmin`.

You can also calculate other parameters from variables, which you can do using the `EVALUATE` command. Let's say that we want a macro that will plot an arbitrary number of functions from an arbitrary starting time for 100 seconds. We will have keywords for the entries for xmin and files, or

```
# SAC macro to read files from disk and plot with a time range of 100 s
# files to be read are set using the "files" keyword
# if no files provided, default files are impulse.sac boxcar.sac␣
↪triangle.sac
# also must provide argument xmin to set start time on plot
$keys xmin files
$default xmin 0
$default files impulse.sac boxcar.sac triangle.sac
setbb range 100
evaluate to xmax $xmin+%range
read $files
xlim $xmin %xmax
plot1
```

Run this by typing `macro testmacro.txt xmin 25 files impulse.sac boxcar.sac triangle.sac` to plot the three functions from 25 to 125. The first line defines that input arguments for `xmin` and `files` should be provided, then it determines `xmax` by adding 100 to `xmin`. Then, the macro reads data from the files, sets the x limits, and plots the signals.

Sometimes, you might want to use an input argument or a blackboard variable to represent a part of a file name. A common example might be if you are reading data froma a seismic network, and the file names are identical with the exception of one piece representing the station name or the component in question. To use a variable within another string, you need to append the variable name with the same character that is used to access its value. So for example, let's say you have two stations, S1 and S2, with components N, E, and Z, and the filename convention from the network operator is stationS1N.sac, stationS1E.sac, stationS2Z.sac, etc. The following macro would correctly read the file specified by the `stat` and `comp` inputs:

```
# SAC macro to read a station component from disk and plot
# must input the station name and component when calling
$keys stat comp
read station$stat$$comp$.sac
plot
```

This can be called using `macro test.macro stat S1 comp N`. Similarly, if `comp` was stored as a blackboard variable instead of a keyword, the equivalent read command would be `read station$stat$%comp%.sac`. This is very handy when complicated, but predictable, filenames are used that only differ in a couple of arguments.

13.2.3 If statements

Macros can use if statements, with a syntax very similar to that of MATLAB. Let's say that in the above macro, you only want to change the x limits if `xmin` is greater than zero. In that case, we could use an if statement:

```
# SAC macro to read files from disk and plot with a time range of 100 s
# files to be read are set using the "files" keyword
# if no files provided, default files are impulse.sac boxcar.sac␣
↪triangle.sac
# can also provide argument xmin to set start time on plot (default 0),
↪ only works if positive
$keys xmin files
$default xmin 0
$default files impulse.sac boxcar.sac triangle.sac
setbb range 100
if $xmin gt 0
    evaluate to xmax $xmin+%range
    xlim $xmin %xmax
else
    xlim 0 %range
endif
read $files
plot1
```

This will set the limits differently if xmin is negative (`gt` stands for greater than). Other comparisons available inlcude `lt` (less than), `le` (less than or equal to), `ge` (greater than or equal to), `ne` (not equal), and `eq` (equal). The `else` (or `elseif`) is optional.

13.2.4 Loops

There are several ways to execute a loop in SAC. The first option is to loop over numeric values (like in MATLAB or using `range` in Python). Let's say you have a series of 10 .sac files, with names "signalx.sac", where x is a number from 0-9. To read all 10 files in using a loop, we can do so as follows:

```
# SAC Macro to read in 10 files and plot
do i = 0, 9
    read more signal$i$.sac
```

```
enddo
plot1
```

This will read in all 10 files and plot them all on the same axes. Optionally, you can add an increment using `do i = 2,8,2` to increment by 2 each time. Another way to write this is to use the syntax `do i from 2 to 8 by 2`.

If you want to loop over a specific set of items, the syntax is a bit different. If we wanted to read all three components of a particular station and plot them, we might modify the above example to be the following:

```
# SAC macro to read 3 components of a station from disk and plot
# must input the station name when calling
$keys stat
do comp list N E Z
    read more station$stat$$comp$.sac
enddo
plot1
```

This version explicitly loops over all three components specified by whatever follows list.

If you want to get some files from the shell using a particular pattern, you can loop over files matching a certain pattern, which is very useful for reading in files. In particular, if you wanted to load the vertical component of any station in the current directory, you can do

```
# SAC macro to plot vertical components in the current directory
do file wild station*Z.sac
    read more $file
enddo
plot1
```

You can also specify a directory other than the current one if you use `do <variable> wild dir <directory> expr`.

Finally, you can do a standard `while` loop in SAC. The syntax is `while <condition> ... enddo`, and `break` will work as we have seen previously in Python and MATLAB to exit loops.

13.3 Summary

While programming in SAC is a bit less refined than programming in Python or MATLAB, it still lets you get the job done fairly well if your work requires processing .sac data files.

Exercise: Write a SAC macro that generates three sine functions with the same frequency and a fixed phase difference between them. Plot all three functions on the same axes, using a different color for each function. The frequency and the phase offset should be specified through command line variables, as should the number of points and the time spacing used in generating the signal.

Default values should be 100 time points, time spacing of 0.05 s, and a frequency of 1 Hz (there should not be a default for the phase offset).

CHAPTER

FOURTEEN

SEISMIC ANALYSIS CODE 3

In this lab on SAC, we will talk about Fourier transforms, plotting spectra, and filtering data, all useful techniques when processing time signal data.

14.1 Discrete Fourier Transforms

When analyzing time signals in geophysics (and practically all other disiplines), it is often convenient to look at functions in the frequency domain. To transform a signal $h(t)$ from the time to frequency domian, we take a Fourier Transform $H(f)$, defined as

$$H(f) = \int_{-\infty}^{+\infty} h(t) \exp(-i2\pi ft) dt.$$

If we have $H(f)$, we can invert it to compute $h(t)$ using

$$h(t) = \int_{-\infty}^{+\infty} H(f) \exp(i2\pi ft) df.$$

On a computer, we only have discrete samples from the time series, so we need to define a discrete version of the transform. Noting that the discrete version of an integral is a sum, if we have a vector h_n of length N, representing the values of the function at the time points in question, then the DFT is

$$H_k = \sum_{n=1}^{N} h_n \exp\left(-i2\pi k \frac{n}{N}\right)$$

where $k = 1, \ldots, N$ represents the various frequencies. Each entry in H_k is a complex number, so it has both a real and an imaginary part (or alternatively an amplitude and a phase). However, this would suggest that we start with N numbers and get out $2N$ numbers (since the transform is N complex numbers), but if the input data are real, then the transform coefficients H_k are symmetric about $N/2$ (known as the Nyquist point, corresponding to the Nyquist frequency or half the sampling frequency of the data, which is the highest frequency signal that can be captured by a discretely sampled time signal) and the imaginary parts are anti-symmetric about $N/2$. Thus, there are really only N meaningful numbers in the discrete transform if we started with real data.

The inverse of the FFT to transform H_k back to h_n is

$$h_n = \frac{1}{N}\sum_{k=1}^{N} H_k \exp\left(i2\pi k\frac{n}{N}\right)$$

Some definitions omit the leading factor of $1/N$, so if you use a software package to do a transform and its inverse and don't get back the same value, you are probably missing this leading factor. Note that the FFT definitions require that the data be uniformly spaced in time, and the algorithm does not depend on what that spacing is. Therefore, to translate the coefficients F_k into specific frequencies, you may need to figure that out on your own using the time spacing of your data (many, including SAC, do this for you).

Looking at the definition of the discrete Fourier transform, it looks like we need to do two nested loops over N terms each in order to calculate this. This means that the number of calculations that you need to do will grow like N^2, which can get rather large for long time series (this prevented Fourier analysis from being very useful in scientific applications). However, some smart people in the 60's figured out that a naive implementation can be improved upon to instead grow like $N \log N$, which is much more efficient. This made discrete Fourier transforms much easier to calculate, and implementations of this algorithm are usually referred to as **Fast Fourier Transforms**, or FFT for short. FFT algorithms usually require that the signal have a length that is a power of 2 for computational efficiency, so if the signal in question is not already an appropriate length, it will be padded with extra zeros to make it have a length that is a power of 2.

Because this algorithm is so widely used, most software packages come with FFTs already installed (this includes MATLAB and SAC, and it is available as an add-on Python package). We will cover FFTs and spectra in SAC in this lab, but you should be aware that many of these same tools are available in MATLAB and Python as well.

14.2 Spectral Analysis

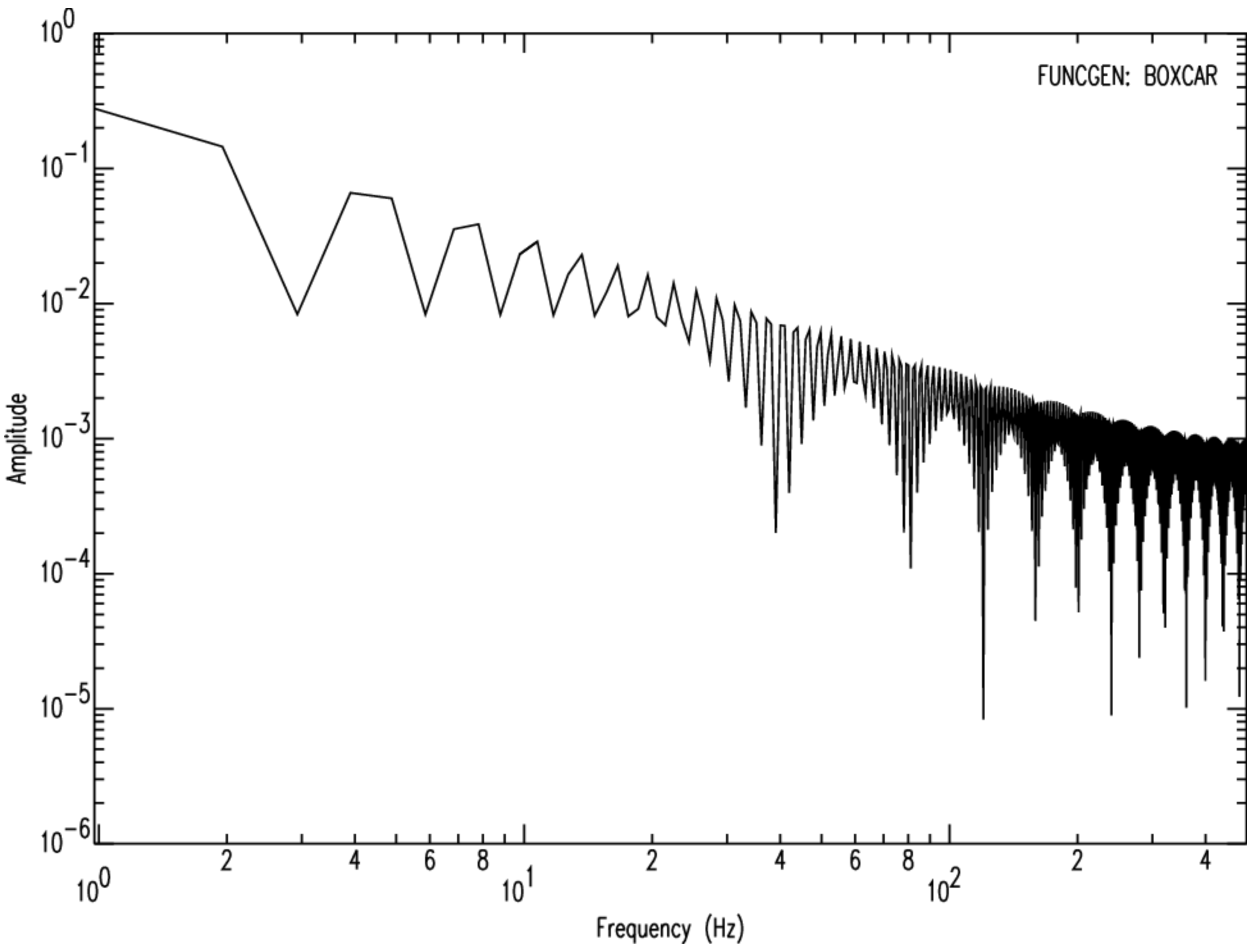
SAC has many functions for doing analysis of spectra. To take the Fourier transform of the signal in memory, use the `FFT` command. `fft` takes two options. First, you can specify `wmean` or `womean` to include or remove the constant DC offset from the transform, respectively. Second, you can obtain two versions of the transform, either `rlim` to obtain real and imaginary parts of the coefficients, or `amph` to obtain amplitude and phase information on the coefficients. The default values are `wmean` and `amph`.

An inverse FFT can be taken with `IFFT`. `ifft` requires that the data in memory is in the frequency domain, either by being the result of a previous `fft` or read from a file containing spectral data. SAC automatically recognizes whether the FFT is stored in real/imaginary or amplitude/phase format, so you do not need to specify anything about the format of the data.

When you take an FFT, the header information on time (beginning, time increment, and number of points) is automatically changed to reflect the beginning frequency, the sampling frequency, and number of points in the transform. An inverse FFT restores the header information to the time domain.

To plot the spectrum once you take an FFT, use the `PLOTSP` command (shortened to `psp`. `plotsp` has two optional arguments. First, you can specify the type of spectral plot: `asis` plots whatever form the data is already in (determined by the command issued when you took the FFT), `rlim` plots the real part of the FFT followed by the imaginary part, `amph` plots the amplitude, then the phase, and `rl`, `im`, `am`, `ph` plots only the real, imaginary, amplitude, or phase, respectively. The default value is `asis`. Second, you can specify the plot style to be linear (`linlin`), semi-log on the horizontal (`loglin`), semi-log on the vertical (`linlog`), or log-log (`loglog`). The default is `loglog`. For example, you can plot the amplitude spectrum of a boxcar function over one second on a log-log scale up to the Nyquist frequency of 500 Hz as follows:

```
funcgen boxcar npts 1000 delta 0.001
fft amph
plotsp am loglog
```



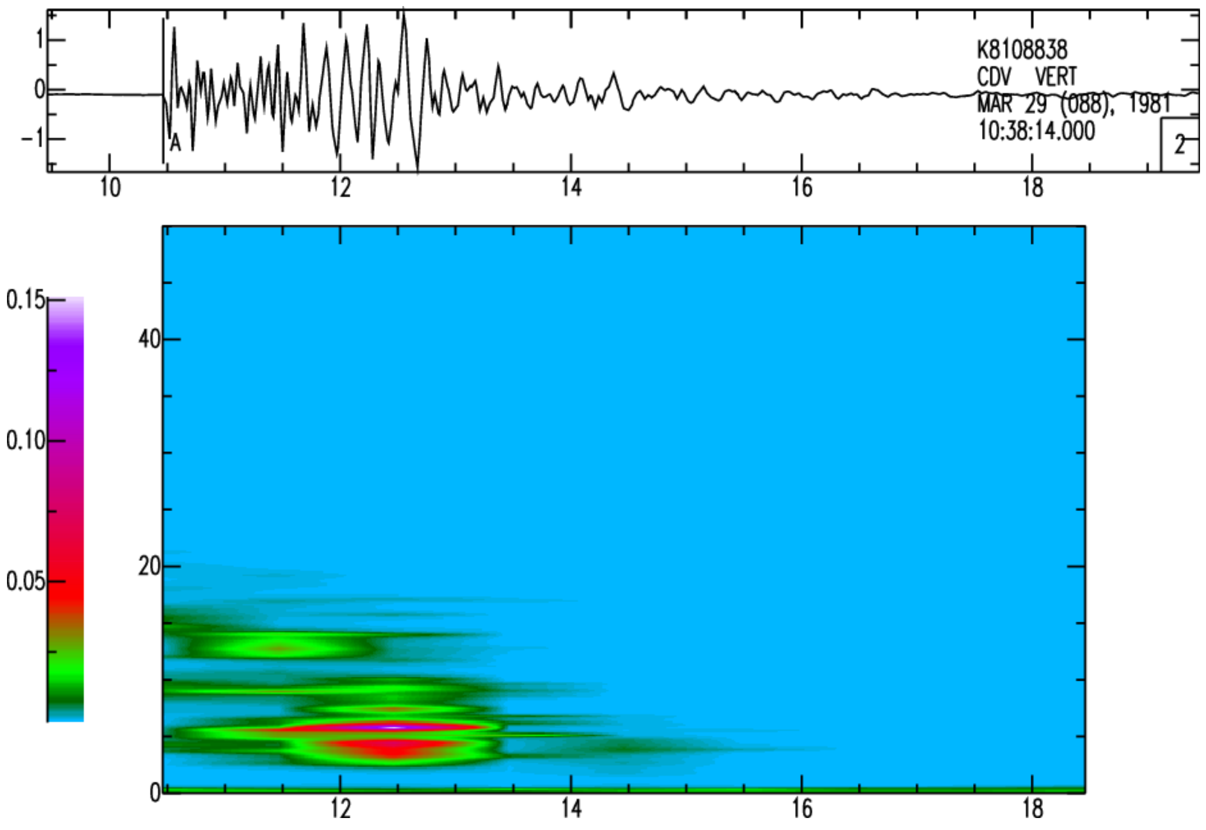
Exercise: Make a spectral plot for the triangle function over the same times and frequencies.

Another common spectral plot is a spectrogram, which plots the spectral content of a window of particular length as a function of time. The SAC command to do this is `SPECTROGRAM`. There are many options for `spectrogram`. The basic options include `window`, which determines the length of the window (in seconds) used to calculate the spectrum (and therefore determines the range of frequencies in the spectrogram), and `slice` which determines the spacing (in seconds) between successive windows. Other options include specifying the details of how the spectrum is estimated, and plot appearance attributes. See `help spectrogram` or the SAC manual for more details on these options.

An example spectrogram can be seen using the following commands:

```
funcgen seismogram
spectrogram
```

which will display the following plot:



One peculiar thing about spectrogram is that since it calculates the spectrogram, places it in memory (overwriting the original signal in the process), and plots it in on single command, you cannot execute it more than once without reloading the data (otherwise it tries to calculate the spectrogram of a spectrogram). Therefore, if you want to change the spectrogram options, you will need to reload the data into memory before executing the command again.

Exercise: Create a spectrogram for a seismogram that you have downloaded from the IRIS website. Note that in many cases, the default parameters might not give a very interesting plot – if necessary, change the `window`, `slice`, and `ymax` parameters to make a more useful plot. (Hint: plot the spectrum of the signal first to determine the relevant frequency band, and then choose `window` to be large enough to resolve those frequencies in the FFT – remember that the minimum nonzero frequency resolved in a DFT is the sampling frequency divided by the number of data points. Set `ymax` to make the vertical range of the plot reasonable for the frequencies in the data. You will also want to increase `slice` so that the time windows are more reasonably spaced, though it is not necessary.)

14.3 Filtering

Many signal processing situations call for filtering data in different frequency bands. SAC has a number of built-in filters, which you can apply using the `BANDPASS`, `HIGHPASS`, and `LOWPASS` commands (shortened to `bp`, `hp`, and `lp`). SAC has four built-in filters, which can be selected with the following options: Butterworth (`butter`), Bessel (`bessel`), and Chebychev type 1 and type 2 filters (`c1` and `c2`, respectively). The details of all of these filters is beyond the scope of this class, but they have the following basic properties:

- **Butterworth:** Designed to have a uniform gain within the passband. The group/phase delay is moderate. The transition from passband to stopband is reasonably sharp. In most cases, this should be the first filter you try for filtering a signal. See the following figure for an example of the gain as a function of frequency.
- **Bessel:** Designed to have a maximally flat group/phase delay, meant to preserve the shape of signals in the passband. However, the consequences of this is that the amplitudes of the filtered signal are not particularly accurate, and the fall-off from the passband to the stopband is the most gradual of all of the filters.
- **Chebychev Type 1:** Designed to have a sharp transition from passband to stopband. The tradeoff is that there are "ripples" in the passband (see below), and poor group/phase response.
- **Chebychev Type 2:** Same as the Chebychev Type 1, but with the ripples in the stopband.

For additional details on these filters, Wikipedia has lots of good information. For more details on their implementation in SAC, see the manual.

The Butterworth and Bessel filters are the simplest to implement. The arguments for the `bandpass` command in SAC involve the number of poles, which describe how the filter transitions from the passband to the stopband, and the frequency range (low to high) to keep in the signal. For example, to implement a butterworth filter with 2 poles from 2 to 10 Hz, the SAC command would be `bandpass butter npoles 2 corner 2 10`. `highpass` and `lowpass` work in a similar fashion, but only take one input frequency. We can see how a bandpass filter affects a sample seismogram as follows. First, generate the seismogram, plot it, and look at its spectrum:

```
funcgen seismogram
qdp off
plot
fft amph
beginwindow
plotsp am
```

The `beginwindow` command simply opens a new figure window, so I can look at these side by side. Now filter the data and plot again:

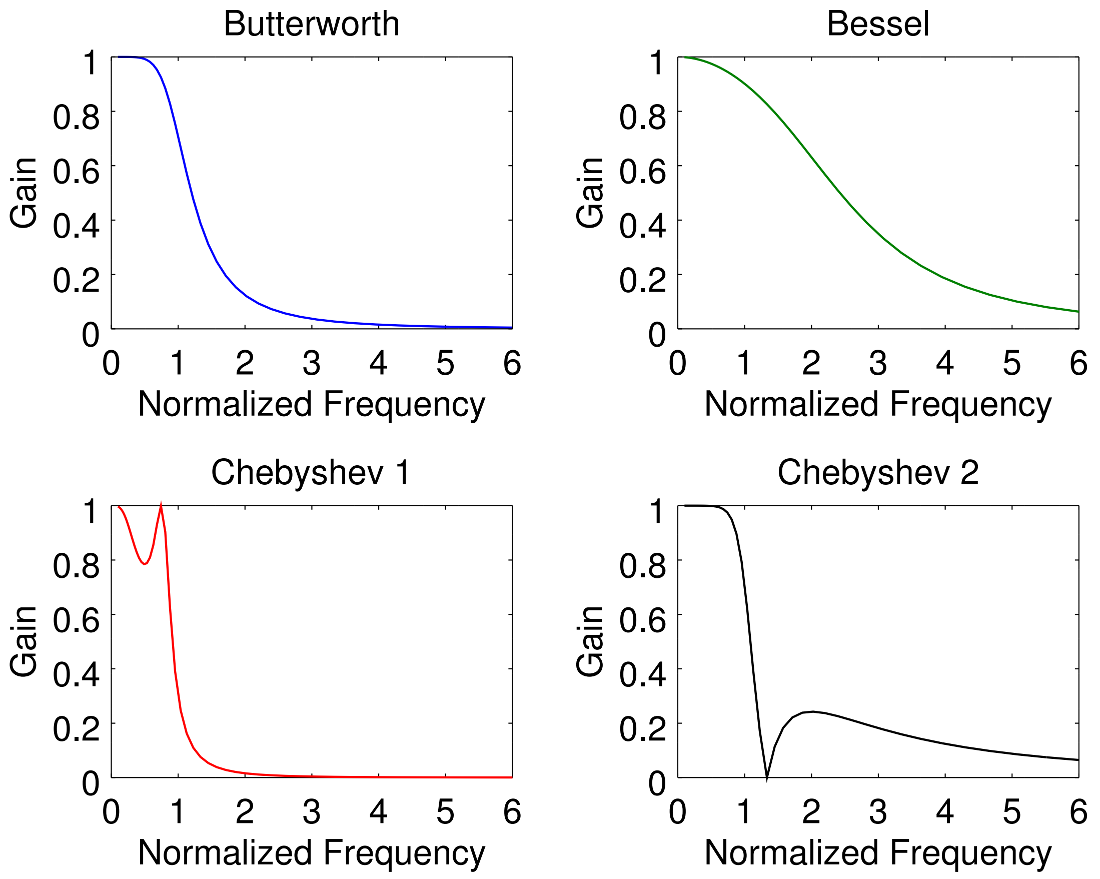


Fig. 14.1: Gain as a function of frequency for several linear filters. The Butterworth is flat in the passband, but has a slower transition from one to the other than the Chebyshev filters. The Bessel filter has the most gradual fall off in the gain, but the group delay is flat in the passband (not shown), helping to maintain the shape of signals. The Chebychev filters are more complicated to implement, and have ripples in either the passband (type 1) or the stopband (type 2).

```
funcgen seismogram
bandpass butter npoles 2 corner 2 10
beginwindow
plot
fft amph
beginwindow
plotsp am
```

You should see that the lower and higher frequencies are reduced relative to the original seismogram.

There are a few other things that you should know about the `bandpass` arguments. First, the minimum corner frequency cannot be zero (you will cause SAC to crash if you do this), and you should use `lowpass` to handle this situation. Also, while `npoles` can take any value from 1-10, in practice people do not usually use `npoles` greater than 3 or 4. While you will get a faster transition from passband to stopband using a higher value for `npoles`, this comes at the cost of a significant effect on the group/phase delay in the passband.

To take the complement of a bandpass filter, which removes the frequencies between the two frequencies specified using `corner` use the `BANDREJ` command. `bandrej` takes all the same options as `bandpass`, so if you know how to use `bandpass`, `bandrej` is straightforward.

Exercise: Try out the `bandpass`, `highpass`, and `lowpass` commands with different values on different signals. The highpass and lowpass functions take only a single frequency, but the other filter details are the same. Also try using the Bessel filter, which takes the same arguments as the Butterworth filter.

CHAPTER

FIFTEEN

SEISMIC ANALYSIS CODE 4

The final two labs on SAC will cover how to do two common types of signal processing in SAC. This lab focuses on stacking, a common method of reducing noise in signals. SAC provides many handy tools for stacking seismic waveforms.

15.1 Stacking

Stacking involves adding signals together so that the signals constructively interfere, while the noise adds incoherently, improving the signal to noise ratio. Seismologists often use stacking when recorded waveforms are relatively weak so that the coherent signal adds, while the random noise cancels out. Stacking can be done in a rather rudimentary fashion, or using a more sophisticated set of commands in SAC known as the "signal stacking subprocess."

What is a subprocess? A subprocess is like a module in Python – it adds some extended functionality to SAC, but we have to tell SAC that we intend to use it. When you run a subprocess, you have additional commands available that are not normally part of SAC, though you have a restricted set of basic SAC commands, usually just commands that can be used to manipulate data. To learn more about the commands available in a particular subprocess, consult the SAC manual.

15.2 Basic Stacking

To perform a basic stacking operation in SAC, we need the `ADDF` command. `ADDF`, short for "add file," is used to add together a signal in memory with a signal loaded from a file. The two signals must have the same number of data points and the same sampling rate, or SAC will complain and not let you add them together. If you aren't sure whether or not your signals can be added, check the header files.

`addf` works in a different manner depending on the number of files in memory and the number of files listed in the `addf` command. If the number of files in memory matches the list of files provided with `addf`, then the first file listed in the `addf` command will be added to the first file in memory, the second file in `addf` will be added to the second file in memory, etc. So for instance,:

```
read file1.sac file2.sac
addf file3.sac file4.sac
```

will add `file1.sac` and `file3.sac` and place it as the first signal in memory, and add `file2.sac` and `file4.sac` and place it as the second signal in memory.

If the number of files in `addf` is less than the number of files in memory, then the last file provided with `addf` will be added to all remaining files in memory that do not match a file in `addf`. Thus, I can add a single file to many different files using:

```
read file1.sac file2.sac file3.sac
addf file4.sac
```

Exercise: Use the `addf` function to combine an impulse function with a boxcar function. Try changing the number of points and time spacing on the functions to verify that you need two signals with the same time sampling to use `addf`.

Once we have `addf` working, it is simple to stack many files: we simply need to repeatedly call `addf` with all of the files that we wish to stack. This can be done most easily using a macro with a `do` loop. So for instance, if I have a series of files that I want to stack, I could stack them as follows:

```
read station1.sac
do file list station2.sac station3.sac station4.sac station5.sac
    addf $file
enddo
plot
```

This will first read in the data for the first station, then loop over the remaining stations and stack them on the signal already in memory. If I have a large number of files with complicated names and did not want to type all of them in, I could use a wildcard loop to read them in:

```
# first generate a signal of all zeros using funcgen
funcgen line 0 0
do file wild station[1-5].sac
    addf $file
enddo
plot
```

The first line is to initialize the signal in memory as all zeros before stacking the first file on top of that; this is simply to avoid the awkward case of reading in the first file but then excluding it from the loop. Note that this assumes that each station has time data that matches the default for `funcgen`. If the default signal length and spacing differs from the signals to be stacked, you will need to specify `npts` and `delta` to match your data when generating the initial string of zeros. Also note that `addf` does not accept wildcards, so if you want to use a wildcard to stack a number of files, you will need to use a `do` loop to accomplish this.

Exercise: I have provided a series of 100 noisy signals ("stack.tar.gz") to practice stacking, which are available on the course website and in my public folder. Read in a few of them to verify that they appear quite noisy – while you can see the strongest arrivals at around 8 seconds, the rest of the signal is not particularly useful (see one example below). Write a SAC macro to stack them, plotting the result (see the second plot below for the stacked version). Try only using a subset of them to see how the signal improves as you add more data.

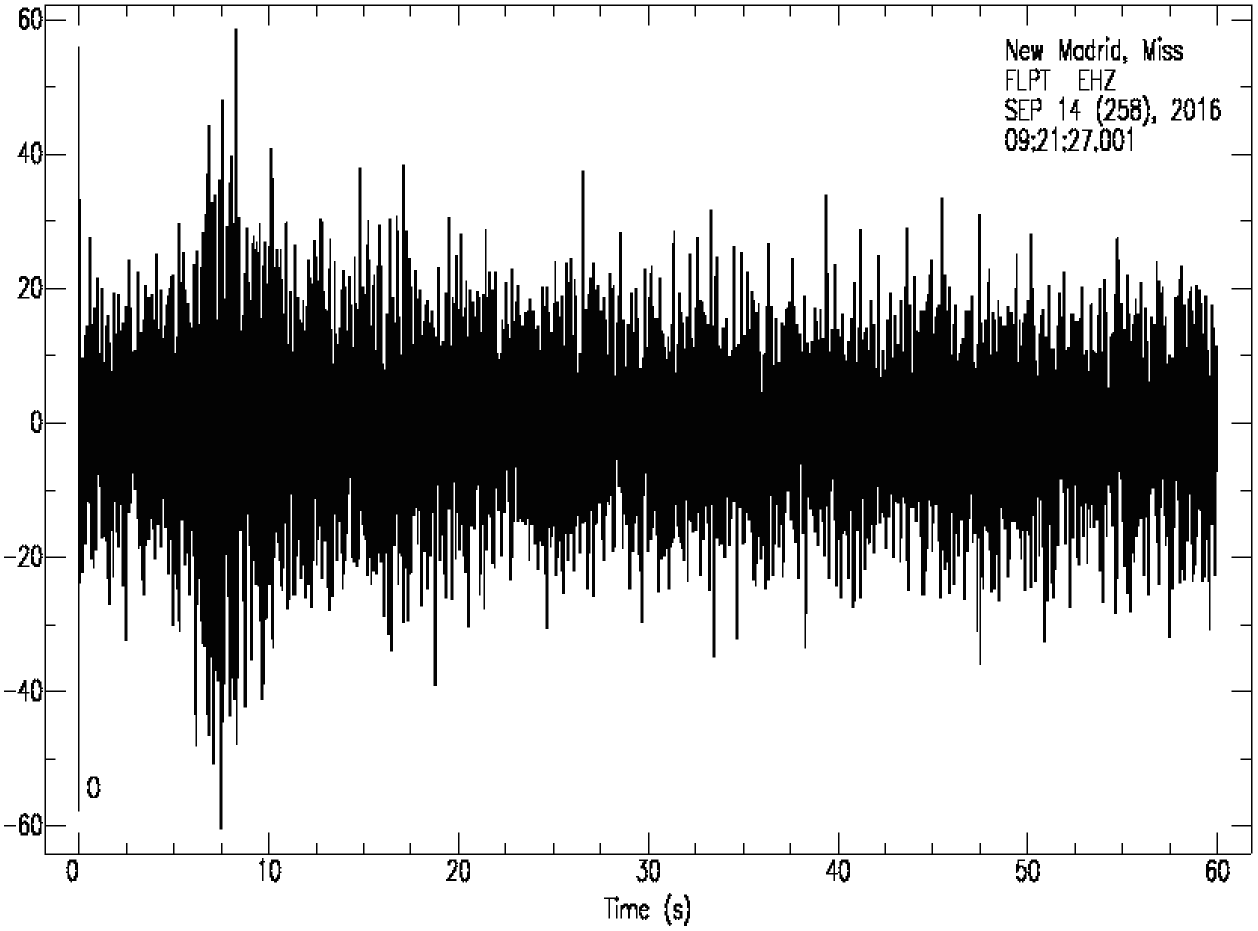


Fig. 15.1: A noisy time signal. There are 100 files like this, where noise has been added that drowns out much of the signal.

Exercise: Write a SAC macro using `addf` to stack a given number of sine functions that have had high amplitude noise added to them (the number of records to stack should be input to the macro as an argument). Your macro should first use a do loop to generate the records using `funcgen sine` and `funcgen random`, add them together using `addf`, and write to disk. Then, use a do loop to read the files and stack them, divide by the number of records (use `div <number>` to divide the signal in memory by a constant number), and plot the data.

Note: Random number generators do not produce truly random numbers. Instead, they produce a deterministic sequence of numbers that follow the same distribution as a set of random numbers. This means that calling `fg random` repeatedly will always produce the same sequence of num-

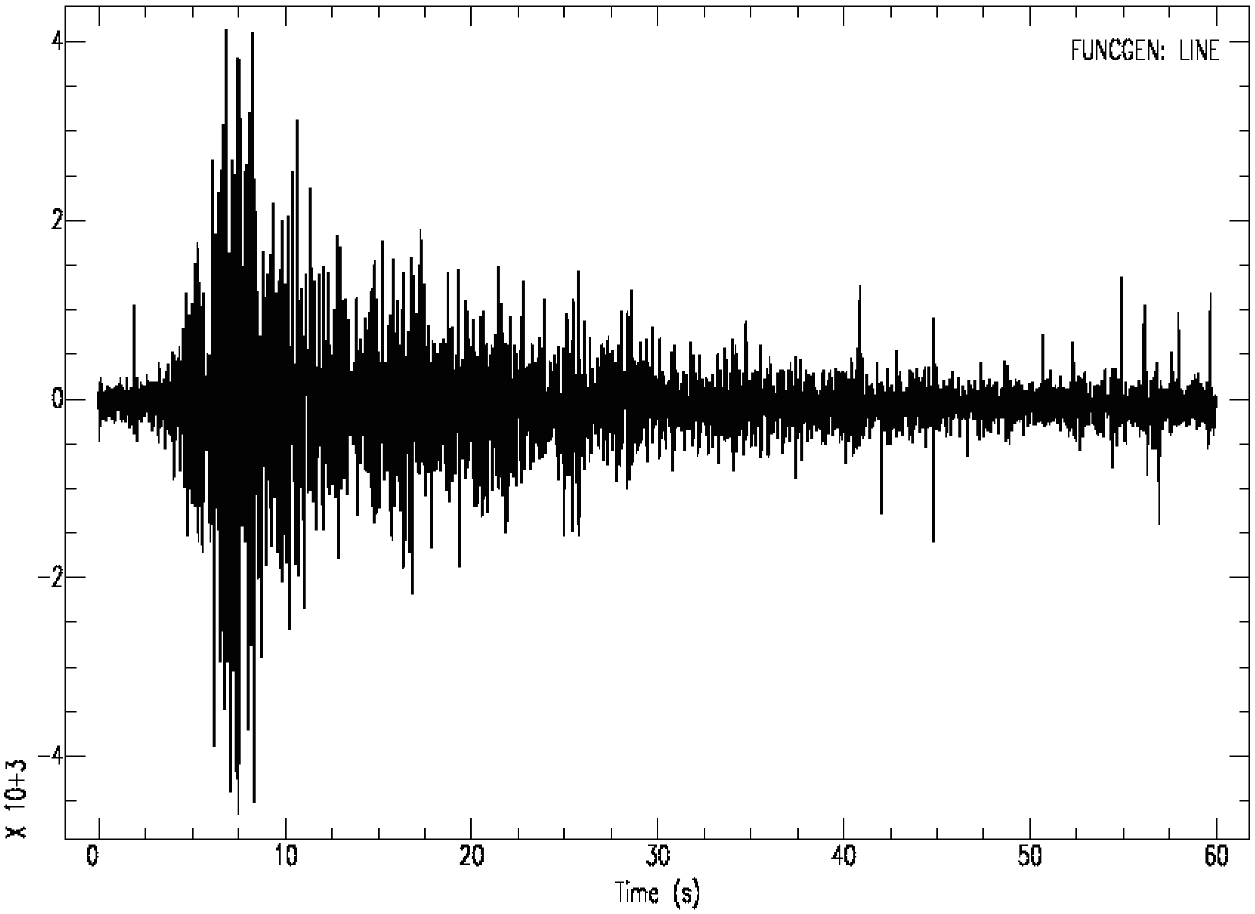


Fig. 15.2: The stacked version signal, which combines 100 noisy waveforms. This signal a much better signal to noise ratio, making the coda decay much clearer.

bers, which is not what we really want here (note however that this gives us a way to reproduce any calculation using random numbers, so this is in many ways a good thing). To produce a different sequence of numbers each time, we need to provide a different "seed" to the random number generator, which will ensure that the deterministic sequence starts at a different place. This can be done in SAC by calling `funcgen random 1 <seed>`, where `<seed>` is an integer that should be different each time through the loop.

15.3 Signal Stacking Subprocess

You can do more complicated things with stacking using the Stacking Subprocess. This gives you more sophisticated tools for aligning signals with different origin times so that they will produce coherent signals when stacked. We will not cover these time alignment tools in this class, but once you know the basics of how to stack signals using the subprocess it is not difficult to figure them out. To start a signal stacking subprocess, enter `sss` into SAC. Once you enter a subprocess, you will not have access to the full extent of SAC commands, so see the manual to know what commands are allowed.

Once you have started the stacking subprocess, you can stack waveforms in the same way as before, the only difference is the commands that you will use. Useful commands include:

- First, to do a stack you are required to define the time window of interest by entering `timewindow <start> <end>`, which defines the window to be used for the stack. Only time values within this window will be read into memory, when you add additional files to be stacked.
- To zero out the stack, enter `zerostack` to start with a clean signal.
- Once the time window is defined and the stack is zeroed, you can use `addstack <file>` to add a file to be stacked. This is the equivalent of `addf` within the stacking subprocess, but is different in that the original file in memory is not overwritten. This allows you to visually inspect the signals before performing the stacking operation.
- Once you have added files to be stacked, you can plot them by entering `plotstack`. This will show all of the files that have been added to the stack with their time axes aligned. If you have chosen a particular velocity model to align wave arrivals, the time axis will account for this correction, but since we are doing a simple stack here, it will plot all of the signals over the same time range.
- When you have added all of the desired files, stack them by entering `sumstack`. To see all of the signals, plus the stacked signal, you can enter `plotstack` to see the results.
- To write the stacked signal to disk, use `writestack <filename>`.
- Finally, you can exit the stacking subprocess by entering `quitsub` into SAC, which will return to normal SAC functionality.

This does essentially the same thing as the simple procedure outlined above for stacking files.

However, the additional tools for defining time windows, the simple way for zeroing out the data in memory, and the ability to see the signals separately along with the stacked sum make the subprocess a better way to carry out stacking operations.

Exercise: Modify the macro you wrote above to stack the noisy signals that I have provided to use the signal stacking subprocess. Write your result to disk so that you can retrieve the stacked result in the main part of SAC.

CHAPTER

SIXTEEN

SEISMIC ANALYSIS CODE 5

This class focuses on using SAC to perform a common signal detection operation: cross correlation. This is another technique to combat poor signal to noise ratios in time series data. The basic idea is to take a known signal and find other times where similar signals occur in a noisy time signal. To perform such signal detection, we will make use of the cross-correlation function in SAC.

16.1 Cross-Correlation, Autocorrelation, and Convolution

Common techniques in signal processing frequently make use of cross-correlation, autocorrelation, and convolution operations. These three operations are all related, and involve integrating the product of two functions with a varying time lag. Given two functions $f(t)$ and $g(t)$, then the cross correlation $(f \star g)(\tau)$ of the two functions is defined to be

$$(f \star g)(\tau) = \int_{-\infty}^{\infty} f(t)g(t+\tau)dt = \int_{-\infty}^{\infty} f(t+\tau)g(t)dt$$

This can be thought of as a moving inner product between the two signals, where τ indicates the time lag between the two signals. If the cross correlation is large at a given time lag, that means that the two signals are similar when they are offset by that value of the time lag – this is true if the cross correlation is positive or negative, with negative indicating the signs are reversed. If the cross correlation is small at a given time lag, then the signals are different at that given offset.

The autocorrelation is simply the cross-correlation of a signal with itself. The autocorrelation function thus has its maximum at a time lag of zero, and is symmetric about $\tau = 0$. The convolution $f * g$ is also similar, but one of the functions is reversed in time. This is illustrated in the figure below, along with a visual representation of how each function takes the values that it does.

16.2 Waveform Cross Correlations

SAC can also be used to perform auto- and cross-correlation analysis, which can be combined with stacking as a useful technique for identifying events in a low signal to noise signal. Basically, when

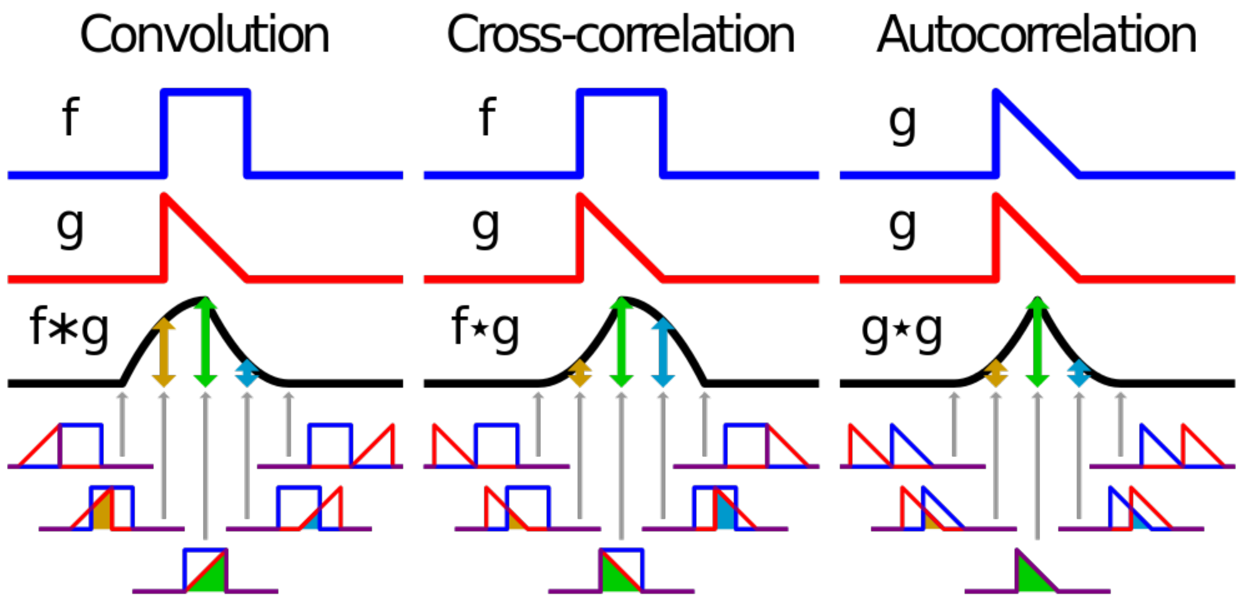


Fig. 16.1: Examples of convolution, cross correlation, and autocorrelation for basic functions. The cross correlation measures the similarity between two signals, while the convolution is essentially a time reversed cross correlation. Autocorrelation measures the similarity of a signal with itself, so it always peaks at zero offset and is symmetric.

the two signals line up in a coherent way, the cross correlation is larger. This can be used to isolate signals from noise, particularly if multiple stations and components are stacked to find situations where several different signals find something similar. Correlations can be easily computed in the frequency domain using Fast Fourier transforms – while a cross correlation in the time domain requires N^2 operations (where N is the length of the signals), in the frequency domain this only requires N operations. However, we have to do the FFT in order to accomplish this, which requires $N \log N$ calculations, so the overall dependence on the signal length is $N \log N$. Regardless, this is an improvement over the time domain calculation, so using this trick cross-correlations can be computed quickly even for long signals.

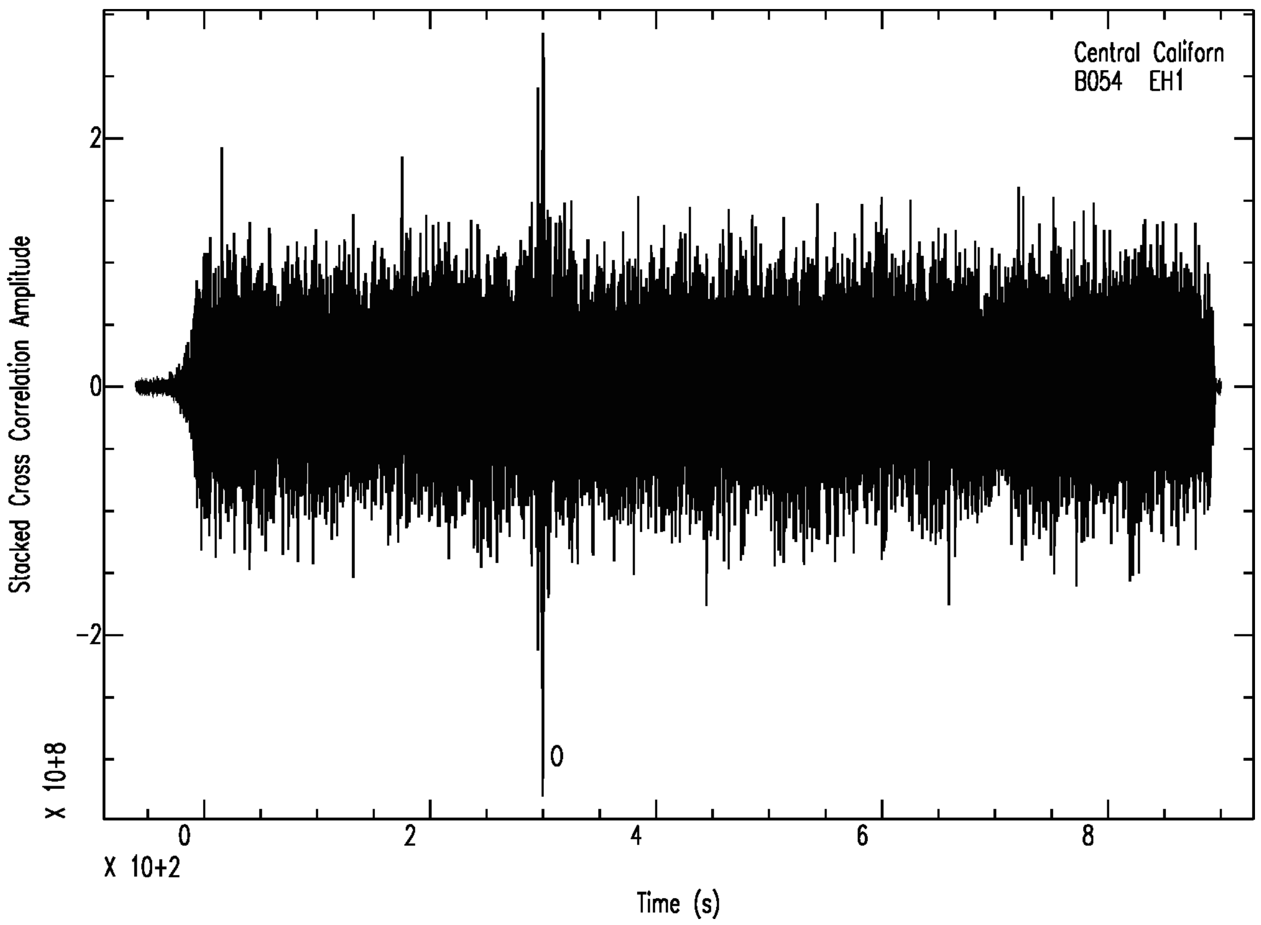
To calculate a cross correlation in SAC, we use `CORRELATE`. Correlate requires that at least two signals be loaded into memory. We need to choose which of the signals to treat as the "master" and this is designated by the order in memory. If we want the first signal to be the master then we enter `correlate master 1`, while if we want the second signal to be the master we type `correlate master 2`. SAC will then correlate the master signal with all signals in memory. Thus if the master is 1, then the first signal in memory will be the autocorrelation of the first signal, while the second signal in memory will be the cross correlation between the signals.

Exercise: Calculate the cross correlation of a boxcar and a triangle function. While the triangle function is not quite the same as in the previous figure, you should see similar results. Also calculate the autocorrelation of each function to verify that the peak is at a time lag of zero and it is symmetric in time regardless of the input function.

Cross correlation can be used to take a known signal and find times when that signal reoccurs in another set of data. It frequently works even if there is a large amount of noise, particularly if many waveforms are used and the results are stacked. The idea is to cross correlate the known signal with a long time series and find times when the cross correlation is large (meaning that the two signals are similar, and thus detect a similar occurrence to the previous event). While the cross correlation may not be high for one particular signal, if we stack many signals we can further improve the signal to noise ratio.

Exercise: I have included a series of 6 waveforms for you to try implementing this method (they are all in the zipped tarball `waveformcc.tar.gz`). The first three are a set of waveforms for a small earthquake in Northern California (`template1.sac`, `template2.sac`, and `templateZ.sac` for the two horizontal and vertical components). These are waveforms recorded on a nearby seismometer. I have also included a recording of another small event that occurred close by where I added noise that completely drowns out the signal (`noisy1.sac`, `noisy2.sac` and `noisyZ.sac`). Your goal is to write a macro that uses the template waveforms to detect the earthquake in the noisy data.

Hint: Cross correlate each component of the template waveform with the corresponding component of the noisy signal. Write the resulting cross correlations to disk, and then stack them. It is helpful to remove the mean of the data before doing the correlation by using `rmean` as it makes the cross correlation values more stable. Once the cross correlation function for all three components are stacked, you should be able to clearly see the time at which the event occurred on the resulting signal (the stacked cross correlation should show a large spike at the time of the event, see below). Plot the data as needed to see what is going on.



Note that this is only for three components; repeating this operation over many stations will further improve the signal to noise and make the event detection even more clear. This is a frequently used method to improve detection of seismic events: using previously detected catalog waveforms, this method can quickly scan long time signals and detect the time of any additional events with similar waveforms.

This lab illustrates how we can use simple features in SAC to perform more complicated signal processing tasks. While we have not covered all aspects of SAC, hopefully you should have a working knowledge of how to use it and can figure out how new commands work from the manual. There will be one homework assignment using SAC, which will involve downloading waveforms, rotating components, decimating/filtering, and plotting what is known as a record section.

CHAPTER

SEVENTEEN

AWK 1

In Homework 1, we wrote a Python function to look at earthquake catalog data that was read from a file. If we want to reuse that function on multiple earthquake catalogs that we might download from the internet, we will need to have all of the catalogs in that same format. Unfortunately, many public data sources provide their output in different formats. They all contain the same basic information, but this information is usually organized differently each time, plus there may be extra information that we do not need in our Python function. Therefore, we need a simple way to reformat text files in order to make our code useful in a more general way. This can be done in Python, but in fact the easiest tool to accomplish this task is another programming language known as AWK.

We will spend two labs learning how to use AWK to process text files. The first lab will focus on basic use for reformatting simple text files (the task described above). Lab 17 will focus on more complicated text processing and pattern matching, which allows you to do more complex tasks to reformat data that is not in simple columns. I find AWK to be very useful when using the Generic Mapping Tools (which we will cover at the end of the semester), so having a basic understanding of how to use AWK will be needed for those classes.

17.1 What is AWK?

AWK derives its name from the initials of its original designers (Aho, Weinberger, and Kernighan). They initially developed AWK in 1977 at AT&T's Bell Labs (the same place where Unix originated). Since then, several further versions of AWK have been developed – New AWK (`nawk`) and GNU AWK (`gawk`) are the other versions of AWK that you are likely to come across. On many operating systems, `awk` and `nawk` are one and the same, and `gawk` is an implementation of `nawk`, so for the most part any AWK program will run equally well regardless of the specific implementation. On the Mac, I believe that the `awk` command is an implementation of New AWK, but there is no `nawk` command. `gawk` is also available on the Mac Lab computers; its path is `/sw/bin/gawk` if you wish to put it in an AWK script. I have written all of the following examples to use the `awk` command, but they should also work using `gawk`.

AWK is a fairly simple, yet powerful command line tool for pattern matching and processing text files. There are several other programs that you can use to so similar things when processing text, including SED, Perl, and Python. SED (Stream Editor) is more basic than AWK and does process text, but it does not have the capability to save variables or do math that is available in AWK. Perl and Python are full fledged programming languages with many more bells and whistles, as we have seen when using Python in this class. Because of this, Perl and Python require a bit more overhead in writing a program. I feel that AWK gives you the most "bang for your buck" for text processing tasks, in that you can learn enough in a couple of hours to be quite competent, and that it is relatively easy to write simple programs. Most of the AWK work that I do involves short "one-liners" to reformat files, and I find that writing such programs are easier in AWK than in Perl or Python.

17.2 Running an AWK Program

AWK programs can be run in two ways. First, you can run AWK directly from the command line. Here is a simple example that prints every line in the input file:

```
% awk '{ print $0 }' inputfile.txt
```

We will see shortly what the specific program commands mean, but there are three parts to this. First, we enter the `awk` command, followed by the AWK program (the part in single quotes, in this case `'{ print $0 }'`), and finally the text file that you wish to process. Try entering this command for a text file (or use the provided "newmadrid2015raw.txt" as the input file, which is the original version of the file that I used to create "newmadrid2015.txt," which we used in the MATLAB classes, using AWK) – the output should just be the contents of the text file.

When calling AWK from the command line, we put the entire AWK program in single quotes so that the shell will interpret the entire program as a single string (the shell has a complex set of quoting rules, as we will see in the upcoming classes on shell scripts). Otherwise, the shell would interpret any spaces in your program as the end of the program, and you would get an error.

Here, we give AWK a single input file, but you can specify as many input files as you like. AWK will process one file after the next without stopping when you supply multiple input files. Additionally, AWK can take standard input from another Unix command through a pipe. By default, AWK sends its output to standard ouput. If you want to save the output of AWK as a text file, use output redirection (> and >>).

Alternatively, you can write an AWK program in a stand alone file (much like you would write a Python program as a stand alone file). To do this, create a text file using your favorite editor called `awkscript.awk` that contains the following lines:

```
#!/usr/bin/awk -f

# awk program prints out entire file

{ print $0 }
```

The first line specifies the command that will be used to interpret the commands in the script, and the second line is a comment (using the # character). The one line that is executed is simply is the same command that we entered above. Here the -f option tells awk that the commands to be executed are contained in a file. This option needs to be in place for AWK to correctly interpret the script.

Change the permissions for awkscript.awk so that you can execute the file (type chmod 755 awkscript.awk into the terminal). Then enter the following to run your script on the input file "inputfile.txt" as before:

```
% ./awkscript.awk inputfile.txt
```

All of this is not strictly necessary; you could just put { print $0 } in the file and then execute the script from the shell using

```
% awk -f awkscript.awk inputfile.txt
```

though it is usually considered good programming practice to specify the command at the beginning of the script so that either you or someone else reading your script are aware that it is an AWK script.

17.3 AWK Program Structure

AWK programs have the following basic structure:

```
BEGIN { <begin commands> } <condition1> { <commands1> } <condition2> {
↪<commands2> } ...
        ... END { <end commands> }
```

- First, AWK will execute all the commands in <begin commands>. This is often where you will define variables that you will use later in your program. This command is optional.
- Then, after executing the begin commands, AWK reads through the input text files one "record" at a time. By default, each record is a line in the file (it splits the file up using newline characters \n), but you can change the behavior of AWK so that it divides a file into records using whatever character you choose. We will see in the next lab how to change this behavior.
- If the current record matches the specified condition <condition1>, then the program executes <commands1>. AWK then proceeds to do the same for all of the pairs of conditions and commands that follow for the current record. Any conditions that are true will result in the corresponding commands being executed. The conditions are optional – if you do not provide a condition, then the corresponding commands will be executed for every record in the file. Technically, you do not need these types of lines, but nearly all AWK programs have at least one condition/execution block.

- Once AWK has executed all conditions and commands for the first record in the text file, it proceeds to the second record, starting at the first condition and testing for all conditions and executing the appropriate commands for all conditions that are true. It continues in this manner for all records in all input files.
- Finally, after going through all records in all input files, AWK executes all of the commands in `<end commands>`. This is where you often do things relating to the total number of lines in the file. As with `BEGIN` commands, this part is optional.

This may seem a bit abstract, but we will give a number of examples shortly to make this structure a bit more concrete.

17.4 Fields

When AWK reads a record from an input file, it automatically parses the record into a number of "fields" separated by spaces. For example, if a line of input text is

```
Eric Daub egdaub CERI
```

Then the first field would be "Eric," the second field would be "Daub," the third would be "egdaub," and the fourth field would be "CERI." You can refer to these fields using the variables `$1` for the first field, `$2` for the second field, etc. The entire line is represented by `$0`. By default, AWK uses any number of consecutive spaces to separate fields, but this can be changed (we will see how this is done later).

One thing to be careful about is the fact that the `$1` syntax has a different meaning in shell scripts (where it signifies command line inputs – this is similar to how SAC macros accept inputs). When you invoke AWK in a shell script, you must be careful about how you use quotes. This is why we generally use single quotes when using AWK in the command line, because it enforces literal interpretation of `$1`, rather than substituting a command line input in its place. We will cover the details underlying this next week when discussing shell scripts.

17.5 Examples

Here are a number of sample AWK programs to give you an idea of how AWK works. These examples use the input file "newmadrid2015raw.txt" Try them out to see how they work, and make changes to confirm your understanding. You cannot try too much code here!

First, here is the AWK program we showed earlier that prints every line in the file:

```
% awk '{ print $0 }' newmadrid2015raw.txt
```

Now that we know a bit about AWK syntax, we can explain why the program does what it does. This program only contains a single condition and command, so it omits the optional `BEGIN` and END commands. Further, the program omits the optional condition, so that the sole provided command in the braces is executed for every line in the file. This command is a print statement that prints the entire record, so the program simply prints the file to standard output.

You can put multiple commands within the same set of braces if you separate them with a semicolon. The following prints the first two fields of each line in the file (in this case, the network code and the event date) on separate lines:

```
% awk '{ print $1; print $2 }' newmadrid2015raw.txt
```

Alternatively, you could have put each print statement in a separate set of braces. Since neither one of the set of commands have a condition associated with them, both are executed for all records:

```
% awk '{ print $1 } { print $2 }' newmadrid2015raw.txt
```

If you want to print more than one field on the same line, simply put all fields in the same print statement:

```
% awk '{ print $1, $2 }' newmadrid2015raw.txt
```

Here is an example that contains a condition; this program prints the entire record of each earthquake that was detected from the New Madrid Network. Note that you do not need to include any sort of if command in an AWK condition; this is implied by the program syntax.

```
% awk '$1 == "NM" { print $0 }' newmadrid2015raw.txt
```

If you provide only a single condition and no statments to be executed in curly braces, by default AWK prints out the entire record. Therefore, the following AWK program is identical to the previous one:

```
% awk '$1 == "NM"' newmadrid2015raw.txt
```

AWK stores all fields as strings, but can perform numerical comparisons using them. If AWK can interpret the string as a number, then it will make a numerical comparison; if the string contains non-numeric characters, then a string comparisons is made based on alphabetical order, with numerical values coming before the alphabetical characters. Thus, to print the full record of each event with a magnitude greater than or equal to 2.5, we can use

```
% awk '$7 >= 2.5 { print $0 }' newmadrid2015raw.txt
```

You can also combine conditionals using && (and) and || (or). The following prints the full record of all events with a latitude between 35° and 35.5°:

```
% awk '$4 >= 35. && $4 <= 35.5 { print $0 }' newmadrid2015raw.txt
```

As you can see, this allows you to very quickly reformat a file with a short piece of code if all you are doing is simply collecting rows from a text file. To make a file similar to the one that we used previously in MATLAB (doing all of the reformatting except for the date/time), we simply need to enter the following into the shell:

```
% awk '{print $2"T"$3, $4, $5, $7}' newmadrid2015raw.txt >␣
↪newmadrid2015.txt
```

Concatenating strings is very easy in AWK; the code above combines the date (field 2) and time (field 3) with a "T" when printing to file. More complex string operations are available, and are described below.

17.6 Formatting AWK Scripts

For writing short AWK scripts on the command line, you can usually just write the entire command out in the terminal on a single line. However, for longer scripts, both on the command line and within a file, programs formatted in such a way can be hard to read and understand. Therefore, it is a good idea to format your AWK scripts by putting statements on different lines in a logical way. This includes putting each `<condition> { <statements> }` block on its own line, and breaking up conditional and loop statements into separate lines to make them easier to read, much like you would do so when writing a MATLAB script.

Within an AWK script saved as a file, putting these statements on separate lines is simple. However, when writing an AWK program directly in the C shell, pressing return to create a new line will cause the program to start running. In the C shell, you can break up commands into separate lines by putting a backslash at the end of the line. This will cause the shell to wait until you have finished entering all of your code before it begins executing the command. For example:

```
% awk '$1 == "NM" { print $4 } \
? $2 == "2015/01/01" { print $7 }' newmadrid2015raw.txt
```

Note that here each set of condition/statment blocks is on its own line. This is important because if you try to break up one of these units, AWK will not understand that they are linked. If you prefer spread your AWK program over several lines and not have to worry about maintaining the integrity of grouped statements, you need to put a double backslash before the new line:

```
% awk '$1 == "NM" \\
? { print $4 } \
? $2 == "2015/01/01" \\
? { print $7 }' newmadrid2015raw.txt
```

Because of this, I only use double backslashes if I ever need to break up lines in the C shell as it is robust and works in all cases.

(Note: the above statements about line breaking and line break characters apply only to the C shell, the default shell in the Mac Lab. If you are using `bash`, then you do not need to place backslashes before carriage returns in command line AWK programs, and only need a single backslash to break up statements like print and conditionals.)

17.7 Variables in AWK

You can define your own variables in AWK. As mentioned above, the only real trick with AWK variables is that they are always stored as strings, though you can do math with them. There is no special syntax needed to assign values to variables, do math with variables, or access values of varibles. Here are a number of examples of AWK programs that use variables.

The following calculates the mean magnitude of all events in the catalog and then prints it at the end. Note that we first initialize the variable in the BEGIN statement, add up the values found from each line, and then print the final value:

```
% awk 'BEGIN {total = 0} { total = total + $7 } \\
? END { print "Average magntidue:", total/NR }' \\
? newmadrid2015raw.txt
```

You do not need to put spaces between variables and operators. There is also a simplified syntax for a number of math operations: `a++` increments `a` by one, `a--` decrements `a` by one, `a += 2` sets `a` equal to its previous value plus 2, `a -= 2` sets `a` to its previous value minus 2.

There are also a number of built-in variables in AWK. One of the most common is `NR` which is the number of records that AWK has processed since being initialized. This variable can also be used to get the total number of lines if accessed in the `END` commands, which we used above to calculate the average magnitude.

If you are dealing with more than one file, `FNR` will give you the number of records that have been read from the current file. Another useful variable is `NF`, the number of fields in the current record. This way, you can print out the last entry of each record, even if the number of fields varies.

```
% awk '{ print $NF }' newmadrid2015raw.txt
```

17.8 Arrays in AWK

You can also define arrays in AWK. One crucial difference between AWK and all of the other languages that we have considered thus far is that array in AWK is what are known as an associative array. Associative arrays differ from regular arrays in that the indices need not be a consecutive set of integers. In MATLAB, if you type `a(10) = 0`, then MATLAB automatically creates an array with size 10 and initializes all of the additional entries to zero. After doing this, you can access `a(1)` without a problem. However, in AWK if you create an array using the syntax `a[10] = 0`, then the array only contains a single entry, and NOT ten entries. The indices need not be integers; you can just as easily write `a["ten"] = 0` and AWK will add a single element that can be retrieved with that string.

(For those interested in the technical details, this type of an array is implemented using what is known as a "hash table," and its primary advantage over a standard array is that its elements can be accessed with a time that is typically independent of the size of the array. This is not necessarily the case for standard arrays; you can only achieve a constant look-up time for an array if all elements of the array have the same size and the size of the array is constant. However, since AWK stores everything as a string, and the length of each array entry thus cannot be guaranteed to have the same length, we cannot be assured that this will work in this case. The fast look-up times for a hash table are achieved by trading time for space and distributing the array storage over a large number of "buckets" that each contain a small number of elements, ideally either one or zero, so that looking up an array element involves simply finding the necessary bucket that hopefully only contains the desired item.)

Because array indices are not necessarily consecutive, this makes it difficult to iterate over an entire array. To iterate over all indices of an array, we use the syntax `for (<index> in <array>)`, followed by the statements to execute. `<index>` can be any variable name you choose. Here is an array example that stores all of the earthquake longitudes in an array, and then iterates over the array to print each value:

```
% awk '{ longs[NR] = $3 } END { for (l in longs) print l, longs[l] }'␣
↪newmadrid2015raw.txt
```

`l` will take all previously used index values that have been assigned to `longs` and then print out the value stored in the array for that index. One thing to note is that an associative array does not preserve the same order in which the indices were added – you will iterate over all of the elements that have been added to your array using `for (<index> in <array>)`, but not in the same order in which they were added. This is a quirk of how associative arrays work, related to the technical details of how they are implemented.

17.9 Other Programming Structures

Within the various command blocks, you can execute conditionals and loops and other flow control statements. The syntax for these are as follows:

- `if` statements have the syntax `if (<condition>) <statement>` with an optional `else <statement>` if desired. Example:

  ```
  % awk '$2 == "2015/01/01" { if ($4 > 36) print $4, "north"; else␣
  ↪print $4, "south" }' newmadrid2015raw.txt
  ```

  You can also use curly braces to indicate where the statements end for the `if` clause rather than a semicolon.

Because of the way AWK programs are executed where a conditional is implied in every group of commands, `if` statements are much less common than in other programming languages. I find that they are mostly useful if you need to use the `else` clause, or if you need to check some condition in the `BEGIN` or `END` blocks in the program.

- `while` loops have the syntax `while (<condition>) <statements>}`, for example, to print only the first three fields in each record, you can use a while loop:

```
% awk '{ i = 1; while (i <= 3) { print $i; i++ } }'␣
↪newmadrid2015raw.txt
```

  Note that you need to group all statements to be executed in the while loop in curly braces. Without that, AWK assumes that only the `print` statement is in the while loop, and since `i` is never incremented you have an infinite loop. Also note that you can use `$i` to refer to field `i`; as `i` varies through the loop, it prints the appropriate value each time through the loop.

- `for` loops have the syntax `for (<initialization>; <condition>; <increment>) <statements>`

```
% awk 'BEGIN { for ( i = 1; i < 4; i++) a[i] = 0 } \\
? $1 == "NM" { a[1]++ } $4 > 36 { a[2]++ } \\
? $6 > 6 { a[3]++ } END { print a[1], a[2], a[3] }'␣
↪newmadrid2015raw.txt
```

  This program determines the number of events detected by the New Madrid network, the number of events north of 36° north latitude, and the number that are deeper than 6 km, and prints all three values to the terminal.

- `do-while` loops are similar to `while` loops, with the key difference that the condition is tested *after* each execution of the loop, guaranteeing that they execute at least once. This differs from while loops, which test the condition *before* executing the statements and are not guaranteed to execute. The syntax is `do <statements> while (<condition>)`. In practice, `do-while` loops are only occasionally needed, as a normal while loop usually suffices.

- `break` exits from the innermost `for`, `while`, or `do-while` loop and continues execution with the statements that follow for the current record.

- `continue` stops executing the current `for`, `while`, or `do-while` loop and proceeds with the next execution of the loop.

- `next` stops executing the current record and proceeds to the next record in the file. Any subsequent conditions are not processed for that particular line.

- `nextfile` stops executing the current file and proceeds to the next file. Any subsequent records in the current file are not processed.

- `exit` stops the program from running altogether.

17.10 String Operations

When doing basic text formatting, we often encounter a situation where we need to further break a field up. Returning to our Earthquake Catalog example, dates are often presented as “month/day/year” or “year/month/day” in catalogs. However, you may want to separate those numbers out for another task. This is most easily done using various string operations in AWK. The ones that I use most commonly include:

- `index(<input>,<target>)` finds the location of the target string in the input string. Useful if you are looking for a specific string but are not certain where it occurs.
- `split(<input>,<array>,<separator>)` takes the input string and breaks it up into pieces, using the separator to divide the string. Useful for things like dates that have a specific character dividing numeric values that you would like to extract. The resulting strings are stored in the array variable given to the program, which you can access using numeric indices starting with 1. The separator can be omitted if you would like to use whitespace as the separator.
- `substr(<input>,<start>,<length>)` returns the substring of the given length, starting at the given position. Useful if you are looking for unknown text but know the position of the string and its length. You can omit the length if you want to take all remaining string characters.

For more information on string operations in AWK, look at the manual (it can be easily found with a web search).

17.11 Exercises

These exercises all use the file “newmadrid2015raw.txt.”

- Write an AWK program that reprints each catalog entry preceded by the number of that entry (i.e. 1. <event data> <new line> 2. <event data>, etc.)
- Write an AWK program that prints out the date and time of all events occurring in the box bounded by $35°$ and $36°$ latitude and $-91°$ and $-90°$ longitude.
- Write an AWK program that finds the average depth of all earthquakes detected by the New Madrid Network.
- Write an AWK program that determines the number of events that occurred in July.
- Reformat the catalog to work with your code from Homework 1. Do this 2 ways: (1) convert the date and time to a string with a format “2015-01-01T00:00:00.0000”, and (2) convert the date to a decimal year in AWK.

CHAPTER

EIGHTEEN

AWK 2

This class cover additional aspects of AWK. The examples presented here build on the material covered in the first AWK lab and cover more complex pattern matching and text processing. We will then use what we have learned to use AWK to reformat a text file containing geophysical data.

18.1 Changing the Field and Record Separators

So far, we have only been using the default AWK behavior of separating records by the newline character (`\n`) and separating fields by any amount of whitespace. However, what if you need to use a different character to designate these differences? For instance, what if you have a table of values separated by commas? Or what if you have a file with multiple lines that should be grouped together, with a blank line to separate? AWK can process these files without any trouble if you tell it how to do so:

- **Field Separator:** The field separator is denoted by the `FS` variable. We can change the field separator by setting it in the `BEGIN` commands. For example, let's say we have a file that contains comma separated values like this:

  ```
  0.1,20,1.4,7
  ```

  We can read these values into AWK and print them to the screen with the following code:

  ```
  % awk 'BEGIN { FS = "," } { print $1, $2, $3, $4 }' << END
  ? 0.1,20,1.4,7
  ? END
  ```

  Here we have done a few things. First, we set the field separator to be a comma (note that you need to put it in quotes when setting the value). Then AWK reads in every record and divides it into fields using commas instead of spaces to separate the fields, printing out the four fields that it finds.

  We have also done one thing new here: we have used standard input to enter our data, rather than reading from a file. This involves using the << operator, one that we only touched on briefly when introducing the shell. The << operator is followed by some end designation (here it is `END`, but in practice this can be any set of characters) that tells the shell the characters that you will enter to tell it that it has reached the end of the file. Once you type the line containing the AWK code, the shell lets you enter input, line by line, until you enter the set of characters that you specified in the first line. Here, we only enter one line, followed by `END`, and AWK proceeds to process that input like it was input from a file. This technique is useful for entering short input into AWK, particularly in a shell script, where saving it as a separate file does not make much sense.

- **Record Separator:** The record separator is denoted by the `RS` variable. As with the field separator, this is most frequently changed in the `BEGIN` portion of the program. For example, let's say we have a file of addresses, each separated by a blank line. We can have AWK read each address as a record, and then separate the record into name, address, and city/state/zip fields as follows:

```
% awk 'BEGIN { FS = "\n"; RS = "" } { print $1, $2, $3 }' << END
? Eric Daub
? 3890 Central Ave
? Memphis, TN 38152
?
? Barack Obama
? 1600 Pennsylvania Ave
? Washington, DC 20500
? END
```

  This will print each address on a single line. However, this code assumes each address is exactly three lines, which is not necessarily true. You can make this code more robust as follows:

```
% awk 'BEGIN { FS = "\n"; RS = ""; ORS = "" } \
? { for (i = 1; i <= NF; i++) \
? { print  $i; print " " } \
? print "\n" }' << END
? Eric Daub
? CERI
? 3890 Central Ave
? Memphis, TN 38152
?
? Barack Obama
? 1600 Pennsylvania Ave
? Washington, DC 20500
? END
```

  Here we used a `for` loop to iterate over all fields in the current record. To prevent AWK from printing each output field on a new line, I changed the output record separator variable `ORS` (AWK puts this character after all `print` statements) to be the empty string, and then explicitly printed spaces and new lines to format the output.

You do not necessarily need to specify the field and record separators in the `BEGIN` section of the program; in fact, in many cases you may need to change the separators as you proceed through the program depending on whether or not certain conditions are met. Also, the field separator does not have to be a single character; it can be a multi-character string or even a regular expression (see below). However, the record separator must be a single character.

18.2 Regular Expressions

In the Unix terminal, we often can use wildcards and other special characters to reduce the amount of typing and match particular files. We can also perform similar tricks in AWK, using what are known as **regular expressions**.

Regular expressions are used as search patterns in AWK in the conditional portions of the program. Regular expressions are enclosed in forward slashes, and are case-sensitive. For example, the following program will print all lines in a file that contain the characters "the:"

```
% awk '/the/ { print $0 }' file.txt
```

If you simply place a pattern within forward slashes as the condition, AWK will execute the commands in braces for any record that contains that pattern. If you want to match a pattern only within a certain field, we use the tilde operator ~:

```
% awk '$1 ~ /the/ { print $0 }' file.txt
```

This will print all lines where the first field contains the characters "the," including "there" and "they," but not "The." If you would like to find all lines that do not contain "the," use `!~` (i.e. does not match) in place of ~.

There are sophisticated rules for constructing search patterns. Here is the syntax for some of the more common patterns:

- To require that the pattern appears at the beginning of the string, use ^ at the beginning of the string. For example, `/^the/` will onlymatch lines that begin with "the." Lines containing "the" in the middle of the string will not be matched.
- To require that the pattern appears at the end of the string, use `$` at the end. For example, `/the$/` will only match lines whose final three characters are "the."
- To match one of a number of characters, use square brackets. `/[Tt]he/` will match either "The" or "the" at any location in a string.
- To match any character *except* certain ones, precede the square bracket expression with ^. Note that the caret has a different meaning inside of square brackets as it does outside of square brackets. For example, `/t[^h]e/` will match any character between "t" and "e" except "the:" it will match any of "tee," "tre," "t1e," "tHe," and many more.
- Inside square brackets, you can match a range of characters with a dash. `/[a-z]he/` will match any lower case alphabetical character followed by "he." It will not match any upper

case or numeric characters followed by "he." You can specify numbers and upper case using `[0-9]` and `[A-Z]`, or any alphanumeric character with `[0-9a-zA-Z]`.

- The vertical bar designates a logical or. `/(^The)|(^Start)/` matches any line that starts with "The" or "Start." Note how we needed to use parentheses to group the expressions – parentheses are used in general within regular expressions to group characters.
- A dot (period) represents any single arbitrary character. `/.he/` will match any character followed by "he."
- An asterisk represents an arbitrary number of occurrences (zero or more) of the previous character. `/th*e/` will match "the," "thhe," "thhhhhhhhhhe," and "te." Note that this is different from the star wildcard in the terminal.
- A plus represents one or more occurrences of the previous character. `/th+e/` will match "the," "thhe," and "thhhhhhhhhhe," but not "te."
- A question mark represents zero or one occurrence of the previous character. `/th?e/` will match "the" and "te," but not "thhe."
- A number in curly braces matches the previous character a specified number of times. You can also specify a range using two numbers separated by a comma, or a minimum with a single number and a comma. `/th{2}e/` will only match "thhe," `/th{2,}e/` will match "thhe" and any other combination with more than two "h" characters. `/th{2,4}e/` will match "thhe," "thhhe," and "thhhhe."
- If you need to match any of the special characters outlined here, put a backslash before the character.

Regular expressions are a bit complicated, but are very useful for dealing with complex data files. Here are some exercises to practice with. You can either make up text files to test your programs with, or enter your own test text using the example above using standard input.

- Write a regular expression that matches postal codes. A U.S. ZIP code consists of 5 digits and an optional hyphen with 4 more digits.
- Write a regular expression that matches any number, including an optional decimal point followed by more digits.
- Write a regular expression that finds e-mail addresses. Look for valid characters (letters, numbers, dots, and underscores) followed by an "@", then more valid characters with at least one dot.

18.3 Formatting Output

We have been using the `print` function to print output. For more control over output formatting, we can use the `printf` function. The syntax is `printf <format>,<item1>,<item2>,...` where `<format>` specifies how to format the output, and the remaining arguments are the items to be printed. `printf` uses a syntax for formatting that is very similar to MATLAB's `fprintf`, so you may recognize some of this (both are derived from the same command that is available in the shell). An example use of `printf` is as follows:

```
% awk '{ printf "%i\n", $1 }' << END
? 1
? 2
? 3
? END
```

This will print field 1 formatted as an integer, followed by a newline character. You can also use `%f` for a floating point number, `%e` for a number in exponential notation, and `%s` for a string. `%d` can be used interchangeably with `%i` as both produce integer results. You can also specify additional information on the number of digits to print:

```
% awk '{ printf "%4.3f\n", $1 }' << END
? 1
? 2
? 3
? END
```

Here, the "4" tells AWK to print at least 4 digits (it will pad with spaces to the left if the width is less than 4), and include 3 figures beyond the decimal point. The meaning of the second formatting digit changes depending on the formatting specifier – for integers, it specifies the minimum number of digits to print, while for strings it specifies the maximum number of characters to print. There are many additional details regarding the use of `printf` to format output, which you can find in the user manual. Spend some time formatting output differently. I find that the majority of my work with AWK only needs simple print statements, and `printf` is only needed occasionally (though it is absolutely necessary in those cases).

18.4 Exercise

Here is an example of how you would use AWK to process a text file in your research. I have provided a google maps "KML" file that contains fault trace information for the Imperial Fault at the California/Mexico Border, taken from the USGS fault data that is publicly available. KML files contain information on how to draw markers, line segments, and other things on a map. However, you may need to take that information and put it into a different format to plot in MATLAB or GMT (one problem on HW5 involves doing this in GMT for a different fault, so it is probably a good idea to save your work as a script for later use).

KML files use XML tags to describe how to plot information on a map. There are many different tags to designate different types of information. Here we would like to extract all of the line segments in the file, each of which represents coordinates used to draw fault traces on a map. Each path to be drawn on the map has the following format:

```
<LineString  id="g39911"><altitudeMode>clampedToGround</altitudeMode>
↪<coordinates>-120.200876555996,40.2943269812509,0
-120.200833555813,40.2943569808352,0
</coordinates></LineString>
```

This represents a line segment, going from the first coordinate listed to the second coordinate listed. The coordinates are separated by newline characters, and each set of coordinates is in the form (longitude, latitude, height). While this example has only two coordinates, other lines may have many more segments. There are also additional markers in the file that we would not like to extract; these markers contain coordinates but do not use the `<LineString>` tag.

Use AWK to extract longitude and latitude coordinates that are used to draw the line segments representing each fault section in the provided KML file. You should print a line containing a > character between each set of segments that make up a LineString (this output format is how GMT reads x-y data segments, so we choose this format because we will eventually be plotting this data in GMT). Separate each longitude and latitude pair with a space.

Hint: It is easiest to do this in stages. First, write AWK code that uses a regular expression to identify any line that contains a set of coordinates. Set the field separator to be a comma, and print out the latitude and longitude that you extract from each set of coordinates. You should try to make this work for the general case, as you will be using this again in your homework. It is usually easiest to build up the regular expression gradually by adding one piece at a time and printing out all lines that match to be sure that it is working correctly.

However, not all of these coordinates are ones that you would like to keep. The ones that you want to keep satisfy one of the following conditions: the coordinates are the first thing on the line, or the tag `<LineString` (followed by some additional information) occurs somewhere else on the line. Modify your regular expression (you may need to add a second matching pattern) so that only these two lines are kept in the ouput.

After doing that, there will still be two problems. First, we have not put the > separator between segments, and there is still text that precedes the coordinates for the first entry in each segment. The > separator should be easy for you to fix. Removing the additional text requires additional string functions.

To remove the text, I found two string functions to be useful: the `match(<string>,<regexp>)` function, which gives the leftmost position in a string that matches the given regular expression, and the `substr(<string>,<index>)` function (covered in the previous lab), which returns the substring of `<string>` beginning at the position given by `<index>`. Use these to correct the output. When all is done correctly, the tail end of your output on the terminal should look something like this:

```
-115.543829900054 32.9199728287905
-115.543870909139 32.9197048434163
-115.54375391094 32.9194648530375
-115.543577910021 32.9192018633972
-115.54355891634 32.9189568754859
```

```
-115.543631926001 32.9187188887966
-115.543614930613 32.9185388975864
-115.543487931054 32.918292906931
-115.543549942905 32.9179639250401
-115.5438819708 32.917499955812
-115.544415016053 32.916745004646
-115.544718052504 32.9159310516597
>
-115.543157660434 32.9275224330804
-115.543366676364 32.9273054482584
-115.543620695267 32.9270404666198
-115.543728702943 32.9269414746468
>
-115.537694060571 32.9399966852325
-115.537746051277 32.940417665069
-115.537799046526 32.9406706533491
-115.537915039282 32.9411296322393
```

This is a tricky problem, so do not worry if you do not get it right on the first try. Be methodical and write your script incrementally.

CHAPTER

NINETEEN

SHELL SCRIPTING 1

This is the first of three labs on Shell Scripting. Shell scripting lets us write scripts that carry out tasks within the Unix terminal. Much like MATLAB `.m` files and SAC macros, we can do things like define variables and control flow using if statements and loops, which gives us more powerful tools to have the computer manipulate files and data. Without shell scripts, if you needed to process 1000 data files with a single program, you would need to type in the same command 1000 times, which would take you a long time and is likely to result in a mistake. A shell script lets you do the same thing with a loop, eliminating typing errors and greatly speeding up the process.

In this document, I will introduce a number of commands, and give simple examples of many of them. You should type each of the simple examples into a shell script file on your computer, run them to ensure that everything is working, and then tweak some things to ensure that you know how the different commands work, particularly for shell variables and quoting strings. This is your chance to experiment with how the commands are used. In the next two labs we will write more complicated scripts that use the commands from today.

19.1 Shell Scripts

To create a shell script, use your favorite text editor to open a new file called `example.csh` (feel free to change the name if you like). You can use any shell that you like, and it is common to use the shell that the script is written for as the file extension. Many people tend to write their shell scripts in either the C Shell or the Bourne Shell, because many of the interactive bells and whistles that have been added in the other shells are not commonly used when shell scripting (the `tcsh` and `bash` shells are supersets of the `csh` and `sh` commands, respectively). Thus, they use the least common denominator to make things as compatible as possible with older systems in case they need to run their script on another computer. However, I will note that on the Mac, running `csh` just runs `tcsh`, and running `sh` just runs `bash`, so in many cases the two are identical. I will try and point out some of the differences between the different shells as we go along, but the text that I type will in general be for the C shell. It is a good idea to be familiar with the syntax for both shells in case you are on a computer with only one of the shells, or you get a shell script from someone else.

Once you have your file open in a text editor, type the following in as the first line:

```
#!/bin/csh
```

This is usually the first line of a shell script, and if it is present it must be the first line of the file and have no leading spaces. This line tells the computer when it is executing your shell script that it should interpret all of the commands using the C shell. It is also telling the person reading the code that the script is intended to be interpreted by the C shell, so that they know what syntax to use when modifying. Remember that there are slight differences between the different shells, so it is important to use the syntax that corresponds with this shell. If you want your shell script to execute without running the default startup file, then the first line should be `#!/bin/csh -f`.

This line to specify the command meant to interpret the commands is generally present in all other types of scripting languages, such as Perl and Python. When a script is executed in the terminal, the shell reads this first line, and then calls the program corresponding to that line with the remaining lines in the script as input. This line is not strictly necessary, but if we do not include it we must specify the program when executing the script. For our shell script, we would need to type `csh example.csh` into the terminal without this first line.

After this line, you will enter the commands that you wish the shell to carry out when you run this script. Any of the commands we have learned so far can be put in a shell script, so to start with put some valid Unix commands into this file (things like `ls` and `cat` that print things out to the screen are good things to put in here while you learn how shell scripts work). You can include comments using # to designate a comment.

19.2 Executing a Shell Script

Once you have entered some commands, save your file and return to the terminal. There are several ways to execute your script. First, as mentioned above you can type `csh example.csh`, which invokes `csh` with your file as its input. This works whether or not you include the shell command as the first line of the file. However, if you do not want to type `csh` before every script that we run, we can avoid this. First, you need to give yourself execute permissions on your file (if you do not have execute permissions, use `chmod` to do so). Then you can enter `./example.csh` into the terminal, which should execute your script. (Note: if you give the file the `.csh` extension, the shell can usually figure out what to use to execute your shell script if no command is specified in the script or the command line. However, it is generally good programming practice to specify the command in the first line of any script.)

19.3 Why `./example.csh`?

Why do we need to enter the period prior to giving the name of the script? If you try to execute your shell script without it, it will not work (you will see a “Command not found” error message). This has to do with how Unix systems look for commands.

Whenever you type a command into Unix, it looks for a command that matches the command that you entered in certain places and in a certain order. It decides the places and the order using what is known as an **environment variable** called `PATH`. An environment variable is a pre-defined variable that provides information on the shell that you are working with. Example environment variables include information like your login name (`USER`) and the location your home directory (`HOME`). These are very useful for making shell scripts that work for different users on different systems, in case the specific configuration of a computer is different from the one where you wrote the script.

One very important environment variable is `PATH`. `PATH` is a string of directories where your shells looks for commands that match the one that you typed into the terminal. When you type in a command, the shell starts looking through the directories in `PATH` until it finds a command that matches the one you entered. It begins in the first directory, and if it does not find a command matching the one you typed, it goes to the next one, continuing all the way through the list of directories. If it gets to the end of all the directories in `PATH` without finding one, it gives an error message "Command not found." Note that this applies to *any* command entered: standard shell commands like `ls` and `cd` are programs too that must be located before they are executed, and they are located in directories in your path.

You can look at your path using the `env` command, which will list all of the environment variables (there are lots). We are interested in `PATH`, which may require that you scroll up a bit. You can also access `PATH` by entering `echo $PATH` into the terminal. `echo` is a useful command in shell scripting, as it can be used to print things to the screen (including variables, which can help you debug your script). In either case, you will see a long list of directories, each separated by a colon (for example, the start of `PATH` might be `/usr/local/bin:/usr/bin:/bin:...`). These are the directories where the shell searches for commands, and the order in which they are searched.

If you ever want to find out if a command is located in one of the directories on your path, use the `which` command. `which` prints out the full path to a command that is entered, and is a convenient way to figure out (1) if a command is in your path, and (2) if a command is located in more than one directory on your path, the one that will be executed.

Note that `PATH` does not contain your current directory, hence you got an error when it looked for this command. To tell the shell to treat example.csh as an executable, we use the "dot" notation `./example.csh` to tell the shell that the executable we want to execute is in the current directory. This is generally true for other types of executables and scripts, such as the AWK scripts we have looked at last week.

There is one way to avoid typing `./example.csh` every time: we can add `.` to our path (in Unix `.` refers to the current directory), so that the shell will know to look in the current directory. This is generally frowned upon by system administrators, as someone could put malicious programs in one of our directories that gets executed because of this (making it a security risk). I do not put "dot" in my path for this reason, and I don't really mind typing two extra characters. You can add additional directories to your path by typing `set PATH = ${PATH}:<directory>` (you will learn what this is doing shortly; basically you are appending a colon and a directory name to the existing string stored in `PATH`). This will only take effect in the present terminal; to make the effect take place in future sessions you will need to add this line to your startup file.

19.4 Variables

As we have seen countless times, variables are a powerful tool for programming as they can save typing and make programs more flexible and robust. Variables in the shell must start with a letter, and they are defined using `set <name> = <value>`, for instance `set x = 1` sets `x` to be the string `1`. To access the value of the variable at another time, use `$x`. Thus, we might write a shell script to copy a file to a new directory as follows:

```
#!/bin/csh

# shell script to copy a file to a new location

set file = myfile.txt

echo "Copying $file to home directory"
cp $file $HOME
```

This script is rather simple, but illustrates how variables are used in the shell. First, note that by using a variable, we only have to change the initial place where `file` was set in our script (I hope the usefulness of this is clear from our previous programming experience). Then, both the `echo` statement and the copy command use the correct filename. Note that I also used the environment variable `HOME` to designate the target directory for the copy command. By using `HOME` instead of the path to my home directory, this means that this script will work on any Unix machine, regardless of how the home directories are set up.

`set` is used to set the value of a variable in the current instance of the shell. However, if your shell script calls some other shell script, you will not have access to that variable within that shell script. To make a variable visible to other instances of the shell, you need to add it to the list of environment variables, as environment variables are passed on when a shell creates another shell. To set an environment variable, type `setenv <name> <variable>`. Try this, and then type `env` to verify that the variable has been added to the list of environment variables.

If you are using `bash`, the syntax for defining variables is different. Regular variables are just set normally in the shell (`x=1`), while environment variables are set using `export`: `export x=1`. Note that you cannot have spaces for setting variables in `bash` like you can using the C shell.

19.5 Quoting Rules in Shell Scripts

Note that in the previous example, I was able to print out the value of `file` in my print statement. This is because I used double quotes to designate the string to be printed out. What if I wanted to literally print out "$file" in that echo statement? Or what if you want to set a variable equal to some special character? We have already seen that one way to include spaces and other special characters in commands is to use the backslash \ immediately before the character. You can also put quotes, either single or double, around the entire string to have the shell interpret the entire string as a single argument in the event that your variable name include spaces.

The shell has complex rules for using quotes to denote commands and variable names. In particular, The shell treats single and double quotes in a different manner when interpreting the contents with respect to variable names. If you include a variable value `$x` inside double quotes, then the shell will replace the variable with its contents within that string. This was illustrated above. A further example:

```
#!/bin/csh
set x = 1
echo "number$x"
```

will print `number1` to the terminal. If you would like to include further text immediately after the variable, use braces to enclose the variable name, like `${x}`:

```
#!/bin/csh
set x = 1
echo "number${x}2"
```

to print `number12` to the terminal. However, if you want the literal characters `${x}` to appear for the `echo` command, then we use single quotes:

```
#!/bin/csh
set x = 1
echo 'number${x}2'
```

which will print `number${x}2` to the terminal. This quoting works the same for setting variables (to set a variable to the string `$x`, put it in single quotes).

Including quotes within a string is a bit trickier. One way to do this is to use the backslash to precede a quote to put it into the string. You can also put single quotes within a double quote string, and vice versa. However, neither of these work if you need to put quotes around a variable name to get the shell to substitute that variable. In this particular case, the way to put quotes inside of a quoted string is to concatenate several strings that use quotes appropriately: for example, `echo "'"'$x'"'"` concatenates three strings together: the first is `'` (the result of `"'"`, or using a single quote within a double quote string), the second is `$x`, which uses single quotes to avoid expanding the shell variable within, and the final string is again `'` (the result of `"'"`). The result is that the shell will print `'$x'` to the terminal.

One other quoting trick: if we want to include the output of a Unix command in a string, surround the command with backwards quotes (`` ` ``). For instance, ``echo "The current directory is `pwd`"`` will print "The current directory is " followed by whatever the output of the `pwd` command is.

Practice using quotes to get the following to print out in the terminal using `echo`. Precede the command with the commands `set x = 1` and `set y = 2` so that you will have some variables to work with. I will refer to the values stored in the variables *x* and *y* in italics, while I will refer to the actual letters x and y in regular script. Other text in italics should print what that text refers to on the screen.

- The final score is *y* to *x*
- Shell scripts print variables using $x and $y
- *x* can't be less than *y*
- Don't forget quotes when printing "$x" and "$y"
- "I won't make mistakes when using *x* and $y"
- The current path is *the current path*
- My home directory is *your home directory*
- The file 'example.csh' contains the echo command *number of times echo appears in example.csh* times

Using quotes properly in a shell script can be tricky, but it is very useful as it lets you control string output. Since much of automating tasks involves reading and saving things from files, being able to specify file names from variables is essential (otherwise you would need to rewrite your shell script every time you wanted to change the target files). Thus, using these quoting rules to correctly construct strings with appropriate file names is a necessary skill for shell scripting.

19.6 Input Arguments

Other uses of variables in shell scripts allow you to make shell scripts into functions. This lets you use one shell script to perform a task that might have different input values. When you call a shell script, you can give it any number of input arguments as follows:

```
$ ./example.csh <var1> <var2> ...
```

Within the shell script, you can access these variables using `$1`, `$2`, etc. As a simple example, the following shell script takes three options, with the first option specifying a search string, and the second two options specifying files in which to search for that string using `grep`

```
#!/bin/csh
grep $1 $2 $3
```

These variables behave exactly like regular variable set with the `set` command. However, note that these have a different meaning than in AWK, so you need to be particularly careful if you need to write an AWK program within a shell script (you need to use single quotes around anything that you plan to pass to AWK).

19.7 Math with Shell Variables

In the C shell (not in `bash` – the syntax for doing arithmetic in `bash` is totally different, and won't be covered here) you can do math with variables if you begin the line with `@`. For example, the following script should print out 1, then 2:

```
#!/bin/csh
set x = 1
echo $x
@ x = $x + 1
echo $x
```

Note that you must have a space between `$x` and the addition sign, and the one and the addition sign as otherwise the shell thinks you want to access the variable `$x+1`. You can also increment a variable by one using `@ x++` and decrement a variable by one using `@ x--`. In these cases, no spaces are necessary.

There are also comparison operators for variables, which do string comparisons. == tests for equality between two strings, != tests if two strings are different, ~ compares a string to a string pattern on the right (i.e. can contain wildcards), and !~ tests if a string does not match a string pattern. Also it is important to note that all arithmetic is *integer* arithmetic, meaning that you cannot do math with a decimal value (you will get an error), and division always returns an integer value (i.e. 3/2=1 in a shell script). Valid arithmetic operators include `+`, `-`, `*`, `/`, `++`, `--`, and `%` (modulo or remainder). We will see how to do non-integer arithmetic a bit later (it requires more complicated commands than integer math).

19.8 Array Variables

In addition to scalar variables, you can define arrays in a shell script. As with scalar variables, all array variables are stored as strings, but you can do integer arithmetic with them. Arrays are defined and you can access the elements as follows:

```
#!/bin/csh
set x = (1 2 3)
echo $x
echo $x[1]
@ x[1] = $x[1] + 1
echo $x
```

Indices in the shell begin at one, just like the indices for arrays in MATLAB. However, unlike MATLAB you cannot do array math (you need to iterate over all of the indices of an array to change everything) or floating point arithmetic, only integer arithmetic.

Other special ways that you can access array variables: `$x[*]` returns a list of all elements of the array `$x`, while if you give a range of numbers `$x[2-4]` it will return a slice of the array. `$#x` returns the number of elements in the list. You can also use an array to define another array, for instance `set y = ($x[1-5])` will make `y` equal to the array containing the first five elements of `x`. The parentheses are important here, as it tells the shell that you want `y` to be an array. Other useful tricks regarding variables and arrays in the C shell include `$?x`, which tells you if a variable is defined (it returns 1 if it is, zero if it is not).

One other useful application of arrays in the C shell is that using `path`, rather than `PATH`, gives you the path as an array of strings instead of as one long string with colons separating the entries. This can be much more useful within a shell script when compared to the standard version with the entries separated by colons.

19.9 Floating Point Math

I rarely need to do floating point math in the shell, but if I ever do, it can be done with quoting. It is a bit awkward, but possible, so if you ever need to do some basic math in a shell script, you can do so. It uses the Unix Basic Calculator command `bc`. `bc` can be run interactively, or run commands supplied in a file or from standard input. The standard input version is most useful in shell scripts. For instance, to add two decimal numbers and store as a variable,

```
#!/bin/csh
set a = 1.4
set b = 2.2
set c = `echo "$a + $b" | bc`
echo $c
```

As you can see, it is a bit awkward as you need to use backwards quotes to get the standard output of `bc` into the command to set the value of the variable `c`. `bc` is useful for a number of other things, such as making less than and greater than type comparisons, which we will see in the next class when we cover conditionals and loops in shell scripts.

19.10 Summary

We have introduced a number of commands and concepts useful for writing shell scripts in the C shell (and Tenex C shell by extension). While these types of commands are generally used in writing shell scripts, they are equally valid in interactive terminal sessions. This means you can define variables in any terminal sessions, and you can also do so in startup files to have variables available for your use when using the shell interactively.

- `echo`
- `env` and environment variables
- `set`

- `setenv`
- `PATH` and `path`
- Variables, input arguments, and arrays
- `@` (for variable arithmetic)
- Quoting (double, single, and backwards versions)
- `bc`

CHAPTER

TWENTY

SHELL SCRIPTING 2

This lab continues covering shell scripts, looking at conditionals, loops, and other useful tools.

20.1 Conditionals

You can include `if` statements in shell scripts to control the flow of your script. For simple one-line if statements, the syntax is `if (<boolean>) <command>` all on a single line. This will execute `<command>` if `<boolean>` is true. For a simple example:

```
#!/bin/csh
# simple if statement example
set a = 1
if ($a == 1) echo "a is one"
```

More complex `if` statements use the syntax below:

```
#!/bin/csh
# complex if statement
set a = 1
set b = 2
if ( $a == 2 ) then
    echo "a is two"
else if ( $b == 2 ) then
    echo "b is two"
else
    echo "neither variable is two"
endif
```

The syntax here differs from `bash` shell scripts, which end `if` statements with `fi` rather than `end if`. Besides the string comparisons that I mention above, there are a number of file booleans that you can use:

- `-e <file>` – test if `<file>` exists

- `-d <directory>` – test if `<directory>` exists and is a directory
- `-f <file>` – test if `<file>` exists and is a regular file
- `-r <file>` – test if `<file>` exists and has read permissions
- `-w <file>` – test if `<file>` exists and has write permissions
- `-x <file>` – test if `<file>` exists and has execute permissions
- `-z <file>` – test if `<file>` is empty

The above are useful for checking necessary conditions before performing some action that requires the above. There is not a built in assertion in the shell, so I frequently check things using conditionals instead if there are conditions that will trip up my shell script.

Other useful tools for comparisons inlcude `!=` (not equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to)‘‘&&‘‘ (logical and), `||` (logical or). Comparisons involving math can be done with `bc`. To test if the sum of variables `x` and `y` is less than some value, you can do this with

```
if ( `echo "$x + $y < 3" | bc`) echo "x and y sum to less than three"
```

To do this, we use backwards quoting to put the output of the command `echo "$x + $y < 3" | bc` into the `if` statement. Note that we need double quotes here in order to substitute the variable values into the string before we pipe it into the `bc` command. The resulting expression evaluates to `1` if the statement is true, and `0` if the statement is false, two values that are compatible with the `if` statement. The backwards quotes send the standard output of the `bc` command to the `if` statement. Note that this can also be done if you do shell math and save the sum in some other variable, so this is not strictly necessary, but it is a useful trick.

20.2 For and While Loops

You can also use loops in a shell script, both of the `for` and `while` flavors. `while` loops have the following syntax:

```
#!/bin/csh
# while loop example
set a = 1
while ( $a != 10 )
    echo $a
    @ a++
end
```

You can use `break` or `continue` within a while loop to either terminate execution or continue to the next loop, respectively. `bash` uses a different syntax (`while (<boolean>); do <commands>; done`).

For loops use the following syntax to iterate over all elements of an array:

```
foreach <varname> (<array>)
    <commands>
end
```

So to print out the same numbers as we saw with the above `while` loop, we could use the following script:

```
#!/bin/csh
# for loop example
set x = (1 2 3 4 5 6 7 8 9)
foreach var ($x)
    echo $var
end
```

`bash` again uses a different syntax for this; in `bash` the syntax is `for <varname> in <array>; do <commands>; done`. If you put the `for`, `do`, and `done` on separate lines, the semicolons are not necessary.

Loops are probably the most important construct in the shell for automating data analysis. They allow you to perform analysis on many different files in a robust, reproducible way. For instance, if you need to loop over all files matching a certain wildcard pattern, you can use backwards quoting and a loop:

```
#!/bin/csh
# perform an action on all files ending in .txt
foreach file (`ls *.txt`)
    echo "Processing $file"
    myprogram $file > output_${file}.txt
end
```

The backward quoting stores the result of the `ls` command in an array that is then looped over, with the value of `file` taking on all results from the `ls` command. This is a very handy way of repeating actions on many files without having to do large amounts of typing. You can also create an array of integers using the `seq` shell command with backwards quotes. `seq 0 1 9` gives a list of integers starting with 0, ending with 9, and incrementing by 1:

```
#!/bin/csh
# perform an action on 10 files with numerical identifiers
# each file is named filen.txt, where n is a number 0-9
foreach n (`seq 0 1 9`)
    echo "Processing file $n"
    myprogram file${n}.txt > output_${n}.txt
end
```

However, note that `seq` is not a standard UNIX program, so may not be available on all systems. If it is not available, you can always use AWK to create the numbers, again using backwards quotes.

20.3 Functions

Shell scripts in `bash` can define internal functions, which is not possible in the C Shell. However, you can write external shell scripts that take input arguments (`$1`, `$2`, etc.) that can serve as functions. More lightweight, single command tasks that are repeated can be shortened using aliases (recall the syntax for setting an alias is `alias <name> <commands>`). I find aliases in the shell particularly useful for writing GMT scripts that work for both GMT 4 and GMT 5, which we will see starting next week when we cover GMT.

20.4 Standard Input Using <<

As mentioned in the AWK labs, the << operator to give standard input is a useful tool in shell scripts when short input to a command is necessary. Why is this a good idea? If we need to give input to a command via standard input, we have a few options:

1. You could simply type out the entry into a separate file. This is fine in many cases, but it has two drawbacks. First, the file exists separately from the shell script, so it is not clear that the two files are related to each other, and you now have to keep track of multiple files and might inadvertantly delete the input file. Second, and most importantly, what if the input changes depending on some other conditions in the shell script? There is no way to tailor the input to the specific task being done.
2. To get around this, a better approach would be to produce the input file within the shell script. That way, you can use variables to modify the input accordingly if the input needs to be adapted to the task. One way to do this is to use a print statement and output redirection:

   ```
   #!/bin/csh
   echo "1 2 3 4 \
   5 6 7 8" > tmp.txt
   cat tmp.txt
   ```

   This should write the desired numbers to file, and print it to the terminal. If you needed to change the input based on some variable value, you would replace the appropriate entry with a variable.

While the second approach is more robust, it does have one downside in that it still produces an extra file that is saved in the current directory. This leads to unneeded clutter unless you explicitly clean up after yourself (which is nevertheless a good idea in shell scripts!).

It turns out that we can eliminate the file altogether by using the << operator. << tells the shell that input text will follow, to be terminated by a user-defined keyword. This keyword can be anything that you want (provided it does not appear in the text to be input to the command). A common choice that is easily understood is `END` or `EOF` (End Of File). As an example, the following shell script will do the same thing as above, without writing the data to a file (thus avoiding the need to clean up after ourselves).

```
#!/bin/csh
cat << END
1 2 3 4
5 6 7 8
END
```

Essentially, the << operator says that the following lines are input, until you run into the word that follows (`END` in this case). You can put shell variables into this text, giving you flexibility to use one script for multiple situations, and there are no additional files to keep track of or maintain. Obviously, this is only practical for short snippets of text to be entered, as otherwise this would make our shell script unreadable, but this is by far the best method for small files.

We will use this method below for calling SAC from a shell script, and in the upcoming classes on GMT, so it is a good idea to become aquainted with this method of entering input to shell commands.

20.5 Calling SAC From a Shell Script

I personally use the above tricks frequently when using SAC, and do most of the complex programming, looping, and file manipulation in the shell rather than directly in SAC. I find that the variable definitions and looping abilites of the shell are more powerful in the Shell than in SAC, so I take advantage of it by writing shell scripts that call SAC for nearly all of my SAC work.

SAC can take commands from standard input. I use the above technique to give standard input to embed short SAC macros into shell scripts, so that I only have one file to maintain and can use shell variables to handle input and things like loops.

Here is an example. Imagine that I have downloaded a series of SAC files from the internet, and want to low-pass filter them all using a user-specified corner frequency, save the files to disk, and then plot them all. The following is how I would do this using a shell script:

```
#!/bin/csh

# Shell script to filter all SAC files in the current directory␣
↪(corner frequency first input argument)
# data is written to file, then plotted
# script then cleans up afterwards

# loop over files

foreach file (`ls *.SAC`)
sac << END
read $file
lowpass butter npoles 2 corner $1
write $file:r_filt.SAC
```

```
quit
END
end

# now read all files in at once and plot

set files = (`ls *_filt.SAC`)

sac << END
read $files
plot1
quit
END

# clean up

rm *_filt.SAC
```

Here, I used `:r`, which strips off the file suffix (so that I could add in the `_filt` part to the filename before the `.SAC` suffix. One thing to be careful about here is that if you want to read from standard input, you must have the input text flush to the left edge of the script (so that the shell can find the `END` delimiter correctly), so I cannot indent my loops in this case. This is the one exception I make to always indenting blocks of code. Note that the variable names are substituted in the input. Note also that I need to put `quit` commands in each set of SAC commands, as otherwise sac will continue running and the script will hang.

I find this approach to be much more powerful than doing everything directly in SAC. Feel free to borrow on these ideas when using SAC.

Exercise: Adapt this shell script to your macros for the 4th and 5th SAC labs, moving all of the loops from the SAC macros to shell scripts.

20.6 Summary

This class covered a number of useful programming techniques in the shell, including:

- `if` statements
- Comparisons in the shell
- `while` loops
- `foreach` loops
- << to give input to a command
- Application of these to writing shell scripts that call SAC macros.

CHAPTER

TWENTYONE

SHELL SCRIPTING 3

In this lab, we will apply the shell scripting commands from the previous lab to create more complex scripts involving several commands. Since getting used to the syntax in Unix commands is not always straightforward, feel free to refer back to to the lab handouts for the Unix and Shell script labs to revisit the commands, or use the `man` pages.

1. Write a shell script that uses the various `env` variables to print the following to the terminal: user and login names, the current directory, the user's home directory, the current shell, and the path. Example output:

```
$ ./listinfo.csh
User: egdaub (login: egdaub)
Current Directory: /gaia/home/egdaub/Documents
Home Directory: /gaia/home/egdaub
Current Shell: /bin/tcsh
Current Path: /usr/bin:/bin:/usr/sbin:...
```

2. Write a shell script that lists the name of the file and the size of the file for all files in the current directory.

3. Write a shell script to check if you can execute a file. The script should take the file name as an input argument. The script output should print the file name, followed by one of the following three messages: `has execute permissions` if the user can execute, `does not have execute permissions` if the user cannot execute, or `does not exist` if the file does not exist.

4. Write a shell script that copies a file, adding the date to the end of the filename of the copied file. The file should be given as a command line input, and the date should be formatted as YYMMDD (i.e. January 10, 2016 is 160110). Append the date just before the file extension. For example, for a file `test.txt` I would run the script as follows:

```
$ ls test*
test.txt
$ ./adddate.csh test.txt
$ ls test*
test.txt           test161108.txt
```

You will find the `date` command useful, and in particular its formatting capabilities (try `date +%y%m%d`). You may also find the following string operations useful:

```
$ set file = file.txt
$ echo $file:r # prints file
$ echo $file:e # prints txt
```

If the input file does not exist, then print an error message to the terminal and do not change any files.

5. Write a shell script to find all executables within the directories in your `PATH` that contain a text pattern (which is specified as a command line input). Recall that you can get the `PATH` formatted as strings separated by colons with `$PATH`, and as an array using `$path`.

6. Write a shell script to copy all files in the current directory that have been modified in the past n hours (n should be specified as a command line input) to a target directory specified as a command line input. If the target directory does not exist, your script should create it.

CHAPTER

TWENTYTWO

GENERIC MAPPING TOOLS 1

This is the first of several labs on the Generic Mapping Tools (GMT). GMT is a set of Unix commands for drawing maps and plotting geophysical data on maps, though you can also use it to make basic x-y plots. GMT is a common tool used by many geophysicists.

22.1 About GMT

GMT is a collection of open-source tools for manipulating geographic data and producing plots and maps. The GMT tools were originally created by Paul Wessel and Walter Smith, though now there are several more developers involved in maintaining and adding new features. GMT features 25 different map projections, and tools to plot a variety of different types of data on them. GMT follows the Unix philosophy (small programs that do one thing well; string together several programs to do complex things), which makes it very powerful (you have control over every detail of your plot) and difficult to learn (you have to specify every detail of your plot).

Additional mapping tools are available in addition to GMT. MATLAB has a mapping toolbox (I have never used it, so have no idea how useful it is). I have used the Python Basemap and Cartopy packages in my research and found that they produce results on par with GMT. Because of its large usage, high quality output, and free and open source availability, GMT is a very useful tool in geophysics research, which is why we cover it in this class.

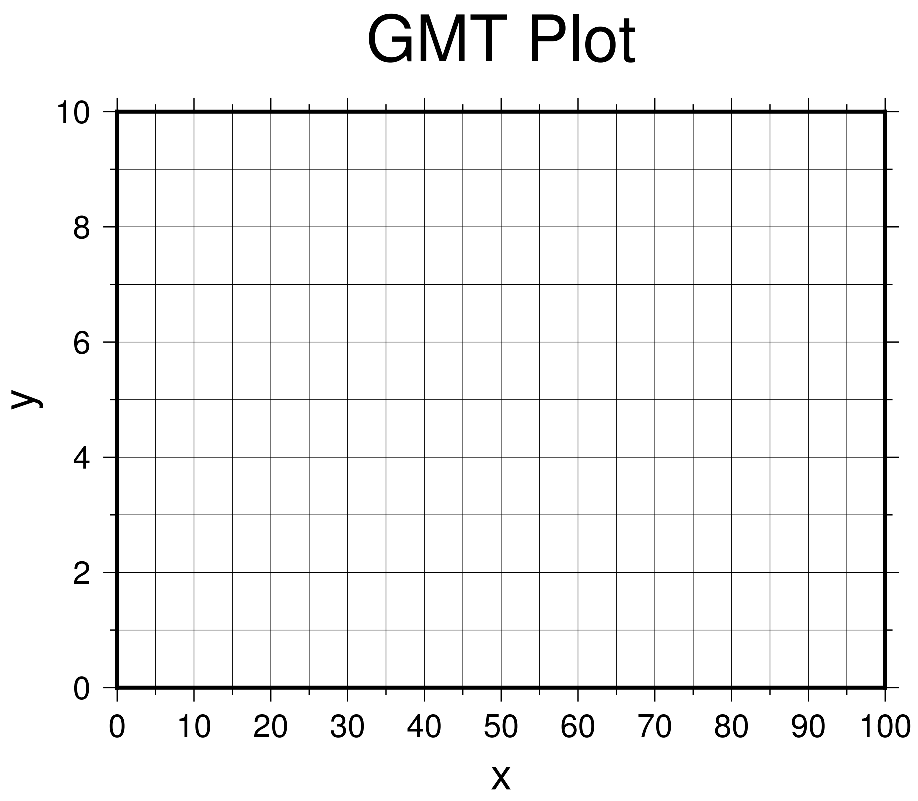
GMT commands are typically combined in a shell script. While they can be used interactively from the command line, they are quite complex to type out, making a shell script a convenient way to use them. Additionally, because many maps are built up in a similar way, this means you can often reuse scripts with some simple modifications for the particular map that you would like to make. This is particularly the case if you use shell variables to form commands. Using shell variables in your commands provides a number of advantages. First, GMT commands are often difficult to understand, but using shell variables means that you can use descriptive variable names to remind you of what everything is doing in a GMT command. Additionally, re-use is easier, as you can set apart the variable definition and command execution so that you can focus on setting parameters and not have to change the parameters, increasing the chances that you get correct code.

I have found that the best way to learn GMT is to get a hold of some example scripts and modify them to figure out what is going on (incidentally, this is how I learn how to use pretty much any type of computing tool). For each lab on GMT, I will provide a shell script that makes a few different maps. You should run them yourself to confirm that they work, and then change some of the shell variables to understand how they work. GMT has extensive documentation (I find myself reading the manual online on a frequent basis), though those pages are often incomprehensible when there are many options to specify.

GMT is currently on version 5. This is the version installed in the Mac Lab, though some older machines may still have GMT 4 as the latest version. Further complicating things is the fact that the Mac Lab lets you use the GMT 4 method for denoting command names, even though it is calling GMT 5. Having these two different versions makes things tricky because the GMT developers decided to change the syntax for a number of commands in going from GMT 4 to GMT 5. The shell scripts I provide can be used for GMT 4 or 5 (I do this by using shell variables and setting them to appropriate values depending on whether GMT 5 is detected on the system), but it is not necessary for you to do this in your homework – I just provide these to illustrate how you might make your scripts work for multiple versions. I will introduce the differences between GMT 4 and 5 as we proceed.

22.2 A Basic Axis Frame

The provided shell script `gmt1.csh` will produce three plots: `gmt1_1.pdf`, `gmt1_2.pdf`, and `gmt1_3.pdf`. Each one contains a number of comments explaining the GMT commands and options. Read through the comments for the first set of commands in the script, which produces the plot `gmt1_1.pdf`, shown below.



This is just a basic axis frame – since GMT follows the Unix philosophy, making axis frames and plotting data are two separate operations, and thus require separate programs. Here is the GMT 4 command that made this plot:

```
psbasemap -R0/100/0/10 -JX4i/3i \
    -Ba10f5g5:"x":/a2f1g1:"y"::."GMT Plot":WSne -P > gmt1_1.ps
```

And in GMT 5, the command is

```
gmt psbasemap -R0/100/0/10 -JX4i/3i \
    -Bxa10f5g5+l"x" -Bya2f1g1+l"y" -BWSne+t"GMT Plot" -P > gmt1_1.ps
```

Note that these are pretty incomprehensible by themselves. To make this easier to read, I define a number of different shell variables containing the relevant information and give them descriptive names. With the shell variables, this is actually

```
set code = "psbasemap -R${xmin}/${xmax}/${ymin}/${ymax} -JX${size} $
↪{bvals} $portrait > $outfile"
eval "$code"
```

Note how this is easier to understand, and it allows you to define all the pieces you need without looking at the command to figure out where everything goes.

One thing to note: when executing these GMT commands, I define shell variables to be strings containing double quotes for any axis label so that when they are substituted into other commands, the quotes remain behind so that GMT can figure out what my axis labels are. However, you do not need to include the quotes if your labels do not contain spaces – they are only strictly necessary if you want to have spaces in your labels or title.

However, if you just try to put shell variables containing spaces inside of quotes into shell command using variables, the shell trips up on them, mistaking the space for the start of a new flag. To avoid this, I put the entire command into a separate shell variable, and then use the `eval` command to evaluate the entire quoted shell variable. This ensures that the shell literally interprets exactly what I want as an entire string forming a single command. This is not needed if there is no label containing spaces, so if you are wondering why some of my plots use this method while others do not, this is why.

The details of how to draw this plot are specified by all of the various command line options above. We will go through these one at a time to explain them.

1. First is the `psbasemap` command. In GMT 4, you simply call `psbasemap`, while in GMT 5 you need to put `gmt` before the command name. I am able to use `psbasemap` in either case by defining an alias at the beginning of the script if GMT 4 is not detected on the system. In the Mac Lab, we use GMT 5 but these aliases are already defined, so you can use either version in your own scripts. This is the command to produce an axis frame, though it does not plot any data (that is a separate commamnd).
2. `-R0/100/0/10` `R` here stands for "region" and tells GMT the range of the plot. Note that unlike MATLAB, GMT does not assume anything about our plot, including the fact that we want our plot to show all of the data. We have to set it ourselves for every plot. Here, this says that we want the x range to go from 0 to 100, and the y range to go from 0 to 10. `R` is one of the required options for the `psbasemap` command. Note that when I use it, I set up shell variables for the min and max for each axis so that I can set the limits in a way that is easier to understand.
3. `-JX4i/3i` `J` specifies the projection (I do not have a mnemonic to remember the meaning of `J`). In this case, `X` means that we are plotting an x-y plot, and that we will tell GMT the plot size (you can alternatively give a lower case `x` which signifies that you want to specify the scale per unit). Here we give `4i`, which means the x-axis should be 4 inches long, and `3i` means the y-axis should be 3 inches long. `J` is also a required argument for `psbasemap`. Note that I use a shell variable for this as well, so that I know what to change if I want to alter the size of my plot later.
4. `-B` The `B` option ("boundaries") is probably the most complex and confusing of all of the GMT options. It tells GMT how to label the axes, how to draw grid lines, and how to title the plot. The manual for `B` is nearly incomprehensible as far as I am concerned, but hopefully you can figure out by trial and error by modifying the example here to see how it works.

   This is one of the other places where GMT 4 and 5 differ. I account for this by putting all of the `B` options in a shell variable that is set depending on the version that is detected.

   There are several parts to the `-B` command in this example:

   - `a10f5g5:"x":` the first part of the option prior to the slash specifies how to draw and label the x axis. You can then give any of three letters `a`, `f`, and `g`, each followed by a number, which describes the annotations (`a`), ticks (`f`), and grid lines (`g`). The number following each of them determines how frequently each item is drawn. Finally, `:"x":` says to label the axis as "x". There are more options beyond these, which you can try to decipher from the manual.

- `a2f1g1:"y":` does the same thing for the y axis as was done above for the x axis. If you want both axes to be annotated in the same way, then you do not need to provide the forward slash and the second set of annotation commands. However, if you want different labels for each axis, you need to give both options, even if the annotations are the same.
- `:."GMT Plot":` This additional text gives the title. Note that the period prior to the text signifies that this is the title and not an axis label.
- `WSne` These trailing letters determine how GMT will draw the axis frame. The letters `W`, `S`, `n`, and `e` each stand for the west, south, north, and east borders, respectively. If a letter is capitalized, that means to both draw and annotate the border. If it is lower case, then that means to only draw the border. If that letter is not present, then GMT neither draws nor annotates that border. Note that you must draw at least one border, so if you call `B` with no letters, it will draw and annotate all four by default.
- For GMT 5, you must call `-B` a separate time for each axis to be annotated, plus once more to set the display of the axis frame and title. If the first letter of the `-B` call is `x`, then that is for the x-axis, if `y` is the first letter then you are annotating the y-axis, and if there is no `x` or `y` then that call will set the frame. Labels and titles are preceeded by `+l` or `+t`, respectively and must appear in the `-B` call starting with x or y for a label, and without for the title. Hopefully the included examples make this more clear.

As you can see, `B` is very complex and can be difficult to understand. This is why I generally set everything in `B` through shell variables so that I have a much clearer picture as to what is going on. You should try experimenting with the options on this script until you get an idea of how they work.

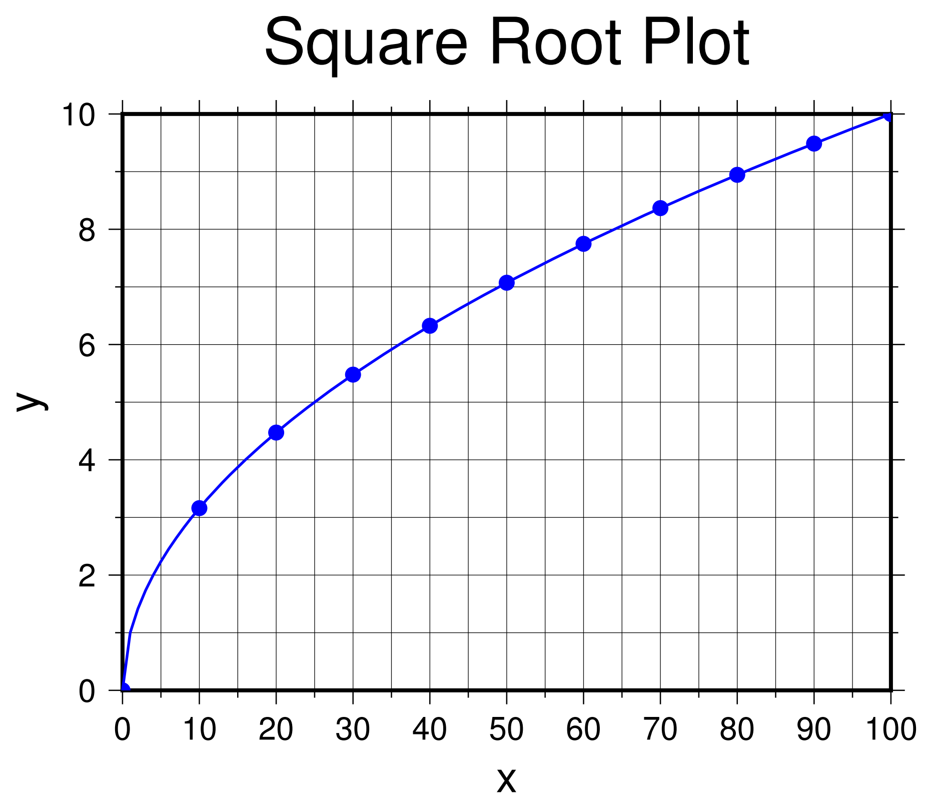
5. The final option is `-P`, which says to make the plot in portrait mode (default is landscape, which will be rotated by 90 degrees from vertical). I generally use portrait mode for all my maps, so there is a `-P` in all of my commands. However, to make it easier to toggle portrait mode on/off, I define a shell variable `portrait` that is included with every call. To enable portrait mode, I include `set portrait = "-P"` in the shell script, and `set portrait = ""` to turn it off. This way, it is set in one place and you do not have to go looking for `-P` flags to delete from every command.
6. The file is then output to `gmt1_1.ps`, a postscript file. Because most GMT commands write output to a file more than once, it is a good idea to use a variable for this so that you only have to set it once. PostScript is the default output format of GMT. You can use Preview on the Mac to open PostScript files and convert to PDF and other formats. GMT also provides a command `ps2raster` (GMT 4 and some versions of GMT 5) or `psconvert` (later versions of GMT 5) to convert PostScript to other formats, which I use here. There are

many options for `psconvert`: here I use `-A0.1i` which says to crop the image 0.1 inch around the edge, and `-Tf` says to give PDF output. See the manual for other output format options with `psconvert`. I also delete the postscript file when I am done with it, as the PDF is easier to deal with for viewing.

You should tweak the commands within this script to see how it changes the final plot appearance. Try changing one thing at a time, in all three of `R`, `J`, and `B` (using the shell variables that I defined). When you have an idea of how this works, proceed to the next plot to see how to actually plot some data.

22.3 A Basic Plot

The second portion of the shell script produces a plot that shows some actual data, and outputs the file `gmt1_2.pdf`. This plot requires three separate commands to produce the following output:



Here is the full set of commands for GMT 4:

```
psbasemap -R0/100/0/10 -Jx0.04i/0.3i \
    -Ba10f5g5:"x":/a2f1g1:"y"::."Square Root Plot":WSne \
    -X2i -Y3i -K -P > gmt1_2.ps
awk "BEGIN { for (x = 0; x<=100; x++) { print x, sqrt(x) } }" | \
    psxy -R -J -W1p,blue -K -O -P >> gmt1_2.ps
```

```
awk "BEGIN { for (x = 0; x<=100; x += 10) { print x, sqrt(x) } }" | \
    psxy -R -J -Sc6p -Gblue -O -P >> gmt1_2.ps
```

For GMT 5, the commands are:

```
gmt psbasemap -R0/100/0/10 -Jx0.04i/0.3i \
    -Bxa10f5g5+l"x" -Bya2f1g1+l"y" -BWSne+t"Square Root Plot" \
    -X2i -Y3i -K -P > gmt1_2.ps
awk "BEGIN { for (x = 0; x<=100; x++) { print x, sqrt(x) } }" | \
    gmt psxy -R -J -W1p,blue -K -O -P >> gmt1_2.ps
awk "BEGIN { for (x = 0; x<=100; x += 10) { print x, sqrt(x) } }" | \
    gmt psxy -R -J -Sc6p -Gblue -O -P >> gmt1_2.ps
```

Using the variable names instead, here is the actual code in the shell script

```
set code = "psbasemap -R${xmin}/${xmax}/${ymin}/${ymax} -Jx${scale}
↪$bvals -X$xshift -Y$yshift $firstplot $portrait > $outfile"
eval "$code"
awk "BEGIN { for (x = ${xmin}; x<=${xmax}; x++) { print x, sqrt(x) } }
↪" | \
    psxy -R -J -W${pen},$color $midplot $portrait >> $outfile
awk "BEGIN { for (x = ${xmin}; x<=${xmax}; x += 10) { print x, sqrt(x)␣
↪} }" | \
    psxy -R -J -S${symb}${symsize} -G$color $finalplot $portrait >>
↪$outfile
```

`psbasemap` contains some of the familiar options from above, plus a few extras. In this case, `R` is the same as before, as is `B`. The `J` option is slightly different than before, as I have used a lower case `x` to tell GMT that I am making an x-y plot. When you use a lower case letter for the map projection, you will specify a scale rather than an absolute size. Here I would like the plot to be the same size as before. Since x goes from 0 to 100, then if each x-unit is 1/25 of an inch (or `0.04i`) the total plot will be 4 inches wide. Similarly, y goes from 0 to 10, so to make a 3 inch high plot, each y-unit should be 3/10 inch.

I have also moved the plot from the lower left corner of the page using `-X2i -Y3i`, which tells GMT to place the plot 2 inches to the right and three inches above the lower left corner. Once you call `-X` and/or `-Y`, you do not need to call them again in subsequent plotting commands.

Finally, I intend to combine several plotting commands to make a complex plot. To do this, we add the `-K` option on `psbasemap` (I think of this as standing for "Keep open"). This allows you to call additional commands to add more things to the plot. I like to handle this using shell variables – I include `set firstplot = "-K"` and then put `$firstplot` into the first command in the sequence so that I have a descriptive reminder as to what that option is doing.

Next, we plot the actual data on the plot using the `psxy` command, which is for plotting x-y" data. `psxy` can either read data from a text file, a binary file, or standard input. Here I provide the data using standard input using AWK. First, I use AWK to print out 100 (x, y) pairs consisting of x and

$\sqrt{x}$, and pipe that into `psxy`.

`psxy` has many of the same options as `psbasemap`, and fortunately we do not need to specify them every time. If we just give `-R -J`, then GMT knows to use the same settings as we used in `psbasemap`. We give four additional options on `psxy`: `-W -K -O -P`, two of which we have seen before. `-O` (I think of it as meaning the file is already "Open") tells GMT that we are appending plot items onto an existing plot. As with the first plot, I define a shell variable for all the intermediate plots `$midplot` that handles the `-K -O` for me. `-W` gives information on how to draw the lines in the plot: `-W1p,blue` says that the lines should be 1 point (1/72 inch) wide and blue. All of this is appended onto the existing file "gmt1_2.ps" using Unix output redirection. Also note that in my AWK program, I used double quotes so that the x limits are correctly substituted into the program before execution.

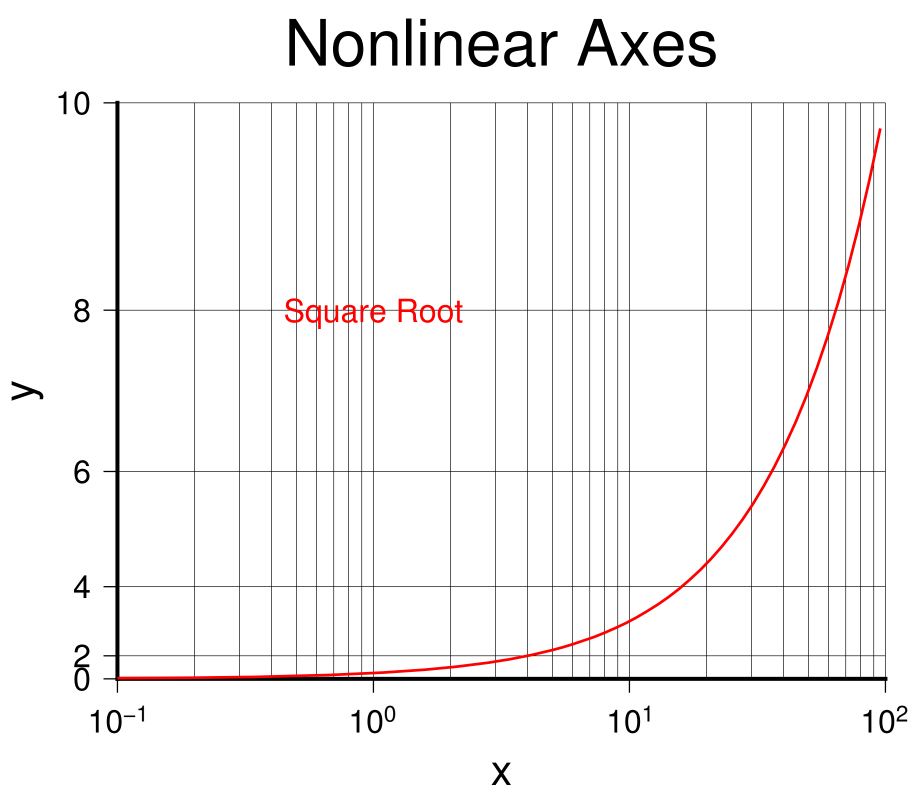
Finally, I want to also plot symbols on the figure in addition to the lines. This requires calling `psxy` a second time, as in the GMT Unix philosophy, plotting lines and plotting symbols are two separate operations. I only want to put a marker for every 10th data point, so I modify my AWK program to only print out every 10th point. You can also change the sampling rate of a dataset using the `sample1d` GMT command if you were making a plot from a text file – see the manual for more details.

I pipe my AWK results into `psxy` again. This time, I call two new options for `psxy`: `-S` and `-G`. `-Sc6p` says to plot each data point as a symbol that is a circle with a diameter of 6 points (6/72 in). `-Gblue` says to color the symbols blue. Note that we did not call the `-W` option here – since we gave the `-S` option, `-W` refers to the outline of the symbol, rather than lines connecting the points. The symbol outlines are drawn by default, so if you do not want the symbol outlines drawn you need to call `-W0` to make the line width zero. Here we want them drawn in the same color as the markers, so we do not need to do anything for `-W`. As with the previous command, we also call `-R -J` with no additional text needed, and `-O -P` to make a portrait layout and tell GMT that we are appending to a previous plot. In the shell script, the `-O` option is handled by a variable `$finalplot` so that it is easier to understand what is going on. Finally, we append to `gmt1_2.ps` as before. I convert to PDF and delete the postscript file when I am done with it.

As you can see, because we need to specify every single detail about our plot, even a basic plot requires a number of commands that can be difficult to understand if you are not GMT savvy. I try to make things easier to understand using shell variables for as many details as I can manage. Anything that is used more than once should *always* be defined in a shell variable so that it only has to be changed in one place, and anything that is hard to understand is usually better being replaced with a descriptive name that is set separately. Look through my use of variables to see how they can make GMT scripts easier to understand.

22.4 Nonlinear Axes

The final example shows how to make nonlinear axes and put text annotations on a GMT plot. The final plot produced by the shell script is the following:



The GMT 4 commands to make this plot are as follows:

```
psbasemap -R0.1/100/0/10 -JX4il/3ip2 \
    -Ba1g3p:"x":/a2g1:"y"::."Nonlinear Axes":WS -K -P > gmt1_3.ps
awk "BEGIN { for (x=0.1; x<=100; x = x*1.1) { print x, sqrt(x) } }" | \
    psxy -R -J -W1p,red -K -O -P >> gmt1_3.ps
pstext << END -R -J -O -P >> gmt1_3.ps
1 8 12 0 0 CM @;red;Square Root
END
```

GMT 5 requires a different syntax for `pstext`:

```
gmt psbasemap -R0.1/100/0/10 -JX4il/3ip2 \
    -Bxa1g3p+l"x" -Bya2g1+l"y" -BWS+t"Nonlinear Axes" -K -P > gmt1_3.ps
awk "BEGIN { for (x=0.1; x<=100; x = x*1.1) { print x, sqrt(x) } }" | \
    gmt psxy -R -J -W1p,red -K -O -P >> gmt1_3.ps
gmt pstext << END -R -J -F+f+a+j -O -P >> gmt1_3.ps
1 8 12,0,red 0 CM Square Root
END
```

The actual commands in the shell script:

```
set code = "psbasemap -R${xmin}/${xmax}/${ymin}/${ymax} -JX$size
↪$bvals $firstplot $portrait > $outfile"
eval "$code"
```

```
awk "BEGIN {for (x=${xmin}; x<=${xmax}; x = x*1.1) {print x, sqrt(x) }␣
↪}" | \
        psxy -R -J -W${pen},$color $midplot $portrait >> $outfile
pstext << END -R -J $textopts $finalplot $portrait >> $outfile
$textinput
END
```

`psbasemap` uses some new options on `-J` and `-B` to create nonlinear axes. First, we tell GMT to use a logarithmic x-axis and an exponential y-axis using `-JX4il/3ip2`. The `X` option, with a width of 4 inches and a height of 3 inches should be familiar. Here the `l` appended on the x-axis says to make the x-axis logarithmic, and the `p2` appended on the y-axis says to make the y-axis exponential with a base of 2.

The other differences are in the `B` options. We give `-Ba1g3p:"x":` or `-Bxa1g3p+l"x"` for the x-axis. For a logarithmic axis, the annotation, tick, and gridline option can be 1, 2, or 3. To annotate only every power of 10, we give 1 as the option. To annotate at 1, 2 and 5 within every decade, we give 2 as the option. Finally, to annotate a every integer 1-9 per decade, we give 3. This is why the annotations are at every power of 10 (we gave `a1`) and there are 9 grid lines for each decade (we gave `g3`). Finally, the trailing `p` tells GMT to write the annotations in exponential notation (if we replaced the `p` with `l`, the annotations would have been logarithmic and read -1, 0, 1, and 2).

The other options on `psbasemap` should be explained above. We then call `psxy` in the same way as above, though because of the logarithmic x-axis, I have distributed the data points differently.

The final command calls `pstext` to place a text annotation on the plot. `pstext` can read from a text file or from standard input; in our case we use standard input to give the x-coordinate, y-coordinate, size, angle, font number, justification, and text. I wish to place my text at $(1, 8)$, with 12 point font, no rotation, font number 0 (Helvetica), and center-middle justified (`CM` means center horizontally, middle vertically). The `@;red;` tag is the syntax in GMT for changing the font color (for other options, see the manual). This input is given to `pstext` along with familiar options `-R -J -O -P`, and the output is appended to `gmt1_3.ps` to complete the plot.

GMT 5 requires different input for `pstext`. The standard way to give input is to give the x value, y value, and the text string, but here I provide additional formatting information, which is indicated by the `-F` switch. Since I give `-F+f+a+j`, this means that I am adding a font specifier (`+f`), and angle (`+a`), and justification (`+j`) prior to the text itself. The font contains 3 comma-separated entries: size, font number, and color, and the angle and justification are described above. Note that I use shell variables to handle the differences between GMT 4 and 5, writing the input text and `-F` switch to shell variables that hold the correct value depending on the version of GMT that is being used. Note that this improves comprehension (both versions are fairly hard to understand in my mind, so using variables gives more descriptive names for both versions) and makes the script more flexible.

22.5 Exercises

The best way to figure out how things work in GMT is to take an existing script and tweak it. Use the provided shell script as a starting point. Here are some plots to try (feel free to make up your own examples, too):

- Make a linear plot of x^2 where x goes from 0 to 10 with annotations at every integer in x and annotations every 10th integer in y.
- Make the same plot as above, but change the axes that are drawn or annotated on the frame.
- Make the same plot as above, but change the frequency of annotations and either add or remove grid lines.
- Make the same plot as above, but add markers or your choice (for the command details, look at the manual) at every integer.
- Make a plot of $\exp(x)$ with a logarithmic y-axis with x going from 0 to 10. (AWK has a built-in `exp` command.)
- Add a text annotation to the previous plot.

CHAPTER

TWENTYTHREE

GENERIC MAPPING TOOLS 2

Now that we have a basic idea of how GMT works, we can start actually making maps. As with x-y plots, there are many options to specify, particularly the map projections. This lab gives several examples of how to plot different map projections, which we discuss before getting to the details of how to plot these projections in GMT.

23.1 Map Projections

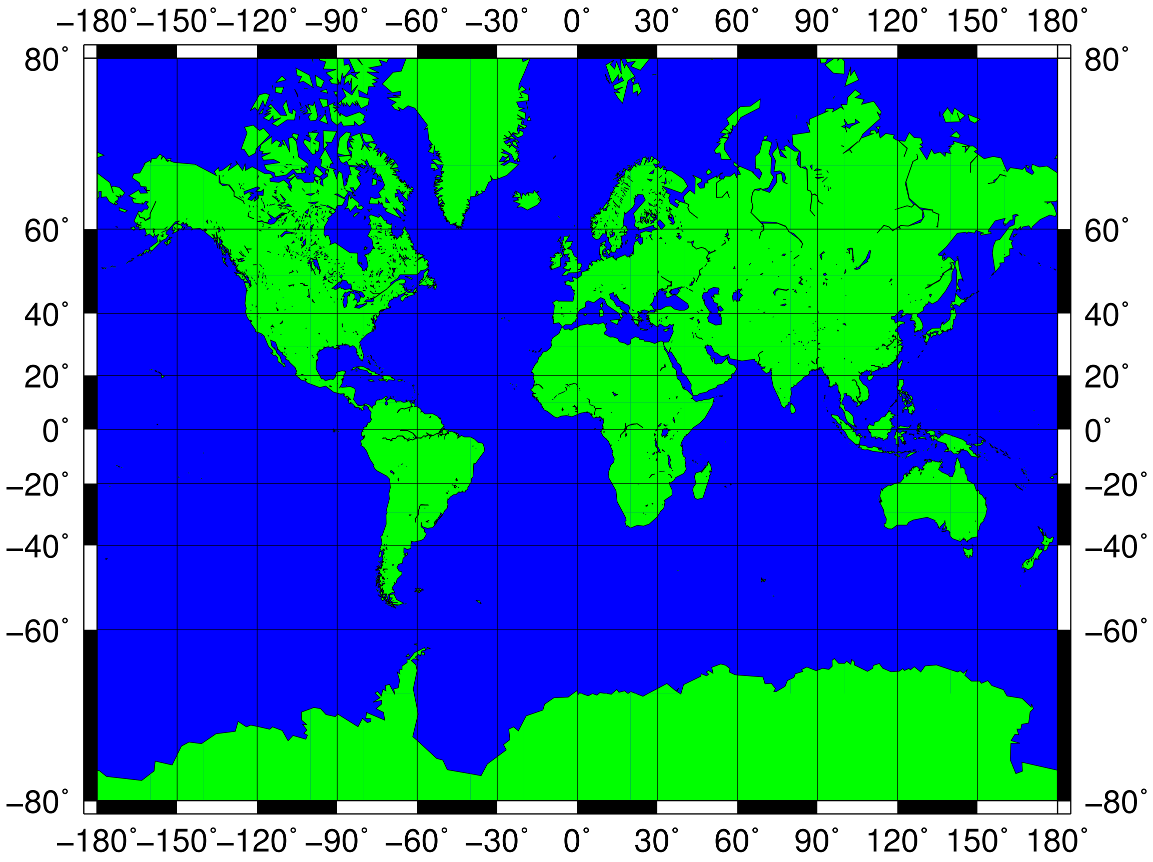
Because the earth is spherical, it is not possible to draw a perfect map projection onto a flat surface. Different map projections are designed to preserve a particular quantity. Some of the most common quantities include:

- Area
- Shape
- Distance
- Bearing
- Direction

These things can all be measured independently, and we can choose a map projection that preserves at least one of these quantities. Because we may need different maps for different purposes, not all maps will want to preserve the same quantities.

For instance, we may want to preserve areas in order to show some quantity per unit area. Equal area projections tend to skew shapes far from certain regions. If we want to maintain shapes (aka have a conformal map), then areas are not usually well-represented. Some maps are better for navigational purposes in that they show bearings correctly, or we may need to get accurate distance estimates. Thus, when considering which map projection to use, you need to consider the data that will be represented on the map.

For instance, most of you have probably seen the Mercator projection. The Mercator projection projects the earth's surface onto a cylinder, and is conformal (it preserves small shapes and angles). It also has the property that straight lines are lines of constant course, useful for nautical navigation:



It is well known, partially because of its longtime use for navigation, partially because it is a nice rectangular projection (i.e. nice for hanging on walls or putting on rectangular book pages), and most recently, because it is the projection used in Google Maps and other online map applications. However, it greatly exaggerates areas near the poles, and makes Greenland appear to be the same size as Africa (when I was in Elementary School, I had a placemat showing a Mercator projection, and was amazed to learn that the area "rankings" I had deduced from the placemat were wildly inaccurate when I compared it to a globe).

There are 5 types of projections available in GMT. We will consider an example of each of them (four in this lab, with the fifth type being the non-geographic x-y "projections" from the last lab). You can get more information on the various map projections on the web (the GMT manual has basic information on each projection, and Wikipedia also has good information on various projections). They break down into the following categories:

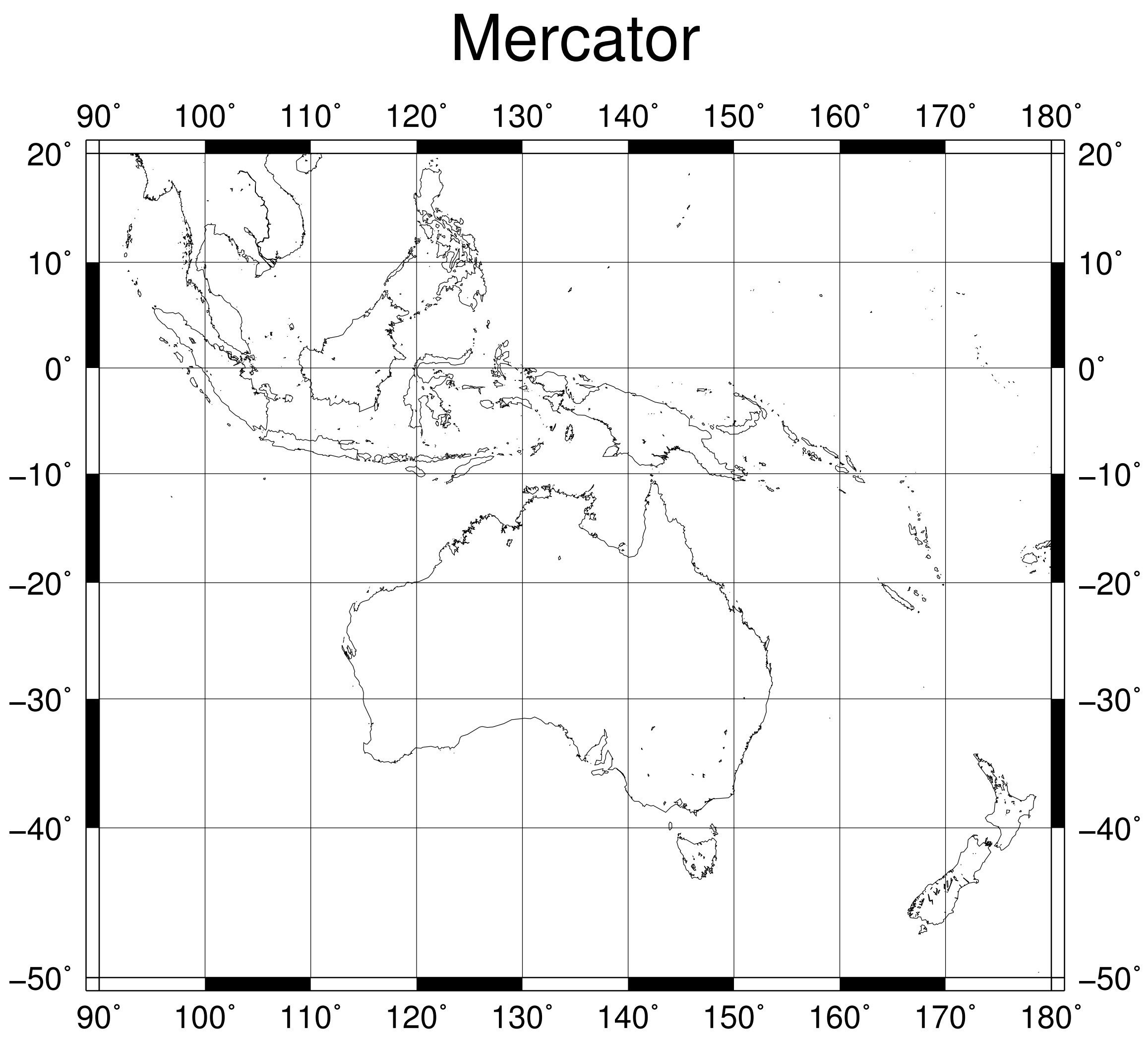
- **Cylindrical Projections** (project the earth's surface onto a cylinder): Cassini, Cylindrical Sterographic, Miller Cylindrical Projectioin, Mercator, Oblique Mercator, Cylindrical Equidistant, Transverse Mercator, Universal Transverse Mercator, Cylindrical Equal-Area
- **Conic Projections** (project the earth's surface onto a cone): Albers, Conic Equidistant, Lambert, (American) Polyconic
- **Azimuthal Projections** (directions from central point are preserved): Lambert, Azimuthal Equidistant, Gnomic, Orthographic, General Perspective, General Stereographic

- **Miscellaneous** (other qualities, or compromise projections that do not preserve any particular quantity): Hammer, Sinusoidal, Eckert IV, Eckert VI, Robinson, Winkel Tripel, Van der Grinten, Mollweide
- **Non-Geographic Projections**: Polar Coordinates, Cartesian (x-y) coordinates

23.2 A Simple Example

In GMT, we use `pscoast` to draw landforms on maps. `pscoast` takes many of the same options as `psbasemap`, and in many cases it is not necessary to call `psbasemap` because all of the necessary details are provided when calling `pscoast`. However, I tend to call both on most of my maps, and specify `-B` when I call `psbasemap` rather than on `pscoast` (as I think of the grid lines belonging more to the axis frame than to the map within). Here I will give examples of both, and you should feel free to do whichever you prefer.

The provided shell script `gmt2.csh` should produce four maps. The first one, `gmt2_1.pdf` is a simple example of a map of a Mercator projection of Oceania:



This map requires only one GMT call, to `pscoast`. In GMT 4:

```
pscoast -R90/180/-50/20 -JM5i -Ba10g10:."Mercator": -W -P > gmt2_1.ps
```

and in GMT 5:

```
gmt pscoast -R90/180/-50/20 -JM5i -Bxa10g10 -Bya10g10 -B+t"Mercator" -
↪W -P > gmt2_1.ps
```

As with the previous set of notes, I use shell variables extensively to make these commands more tractable. The actual command in the shell script is

```
set code = "pscoast -R${xmin}/${xmax}/${ymin}/${ymax} -JM$size $bvals
↪$continents $portrait > $outfile"
eval "$code"
```

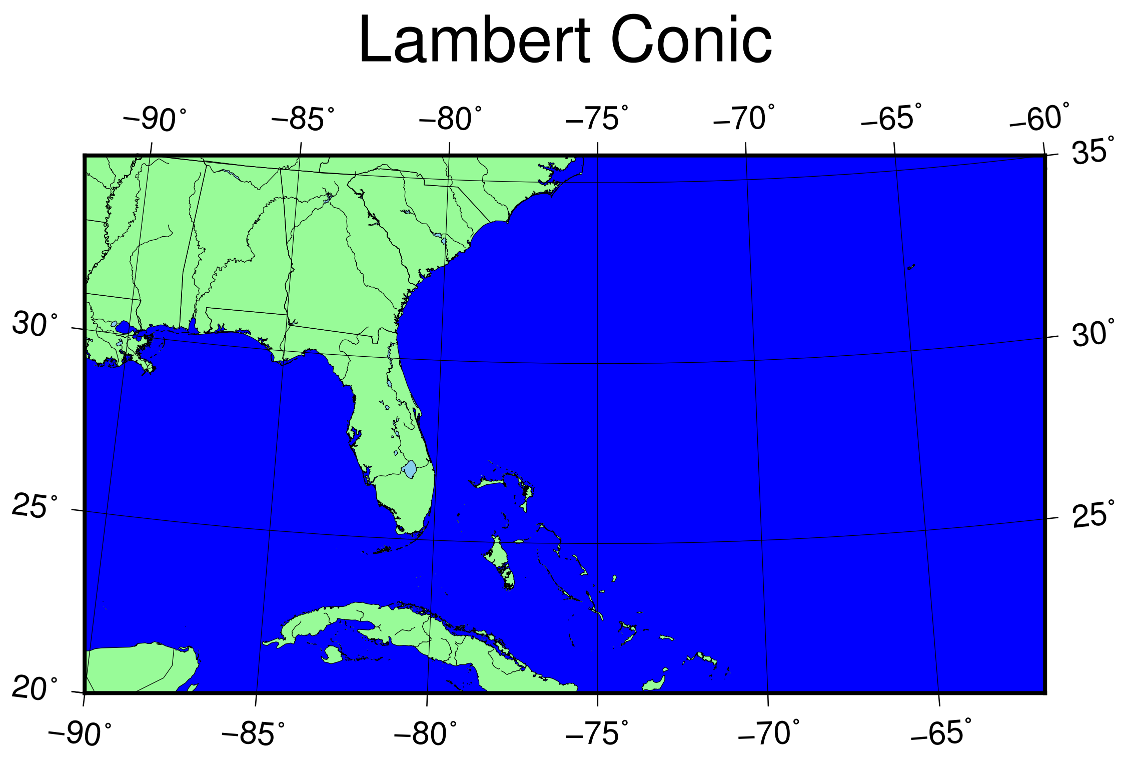
Here I use the trick of setting the entire command to a variable in case we need to include a title that includes spaces. The command options should all be familiar to you from the first GMT lab, though their exact usage is a bit different here. Let's look at them one at a time:

- `-R90/180/-50/20` are the `-R` options for this map: we give minimum and maximum longitude and latitude values to determine the map region. Here I am going from $90°$ to $180°$ in longitude, and $-50°$ to $-20°$ in latitude. Unlike when doing an x-y plot, the region can only take valid sets of geographical coordinates. Longitude must be in the range $(-180, 180)$ or $(0, 360)$, and latitude must be in the range $(-90, 90)$. Certain map projections like the Mercator cannot include the full globe, as they are not defined at the poles. If you try to extend the region of such a projection to a pole, you will get an error message.
- `-JM5i` are the `-J` options for this map, which sets the map projection. `M` stands for the Mercator projection, and the fact that it is a capital letter means that we set the map width of 5 inches (lower case means we would specify a length per degree longitude). Note that the map projection determines the map height, so we only need to specify the width. Some projections require additional information to define the projection that is independent of the region, though the Mercator projection is not among these (the min/max lat/lon from the `-R` option is all that is needed).
- `-Ba10g10:."Mercator":` (GMT 4) or `-Bxa10g10 -Bya10g10 -B+t"Mercator"` (GMT 5) sets the boundary annotation to every 10 degrees, and the grid lines to every 10 degrees. The title of the plot is "Mercator"
- `-W` tells GMT to draw coastlines using the specified line properties (none specified sets it to the defaults). You need to specify at least one additional option in `pscoast` regarding how to draw or fill the continents; here I set how to draw them (we will see later how to fill land and water areas).
- `-P` sets output to portrait mode, and `> gmt2_1.ps` redirects output to file. I then convert to PDF using `psconvert` and delete the postscript file.

This example is quite simple, as the Mercator projection is fairly easy to understand.

23.3 A Non-Rectangular Projection

The second example is of a Lambert Conic projection, which would produce a non-rectangular map if we specify the region using minimum and maximum longitudes and latitudes. However, what if you want your map to be rectangular? We can still make the final map output rectangular for this projection by changing the syntax for specifying `-R`. This produces a map `gmt2_2.pdf` that looks like the following:



Note that while the map is rectangular, the boundaries are not lines of constant latitude and longitude. This effect can be produced in GMT as follows (GMT 4):

```
pscoast -R-90/20/-60/35r -JL-75/27.5/20/35/5i \
    -Gpalegreen -Sblue -Cskyblue -Na -Ir -W -Di \
    -K -P > gmt2_2.ps
psbasemap -R -J -Ba5g5:."Lambert Conic": -O -P >> gmt2_2.ps
```

and for GMT 5:

```
gmt pscoast -R-90/20/-60/35r -JL-75/27.5/20/35/5i \
    -Gpalegreen -Sblue -Cskyblue -Na -Ir -W -Di \
    -K -P > gmt2_2.ps
gmt psbasemap -R -J -Bxa5g5 -Bya5g5 -B+t"Lambert Conic" -O -P >> gmt2_
 ↪2.ps
```

The commands using variables is the following:

```
pscoast -R${xmin}/${ymin}/${xmax}/${ymax}r -JL${clon}/${clat}/${minlat}
 ↪/${maxlat}/$size \
    -G$land -S$ocean -C$lake $natbounds $rivers $continents -D
 ↪$resolution \
    $firstplot $portrait > $outfile
set code = "psbasemap -R -J $bvals $finalplot $portrait >> $outfile"
eval "$code"
```

I have split this into two commands, one `pscoast` specifying how to draw the map, and `psbasemap` specifying how to draw the frame since the `pscoast` command is already fairly complicated. This example introduces several new syntax items:

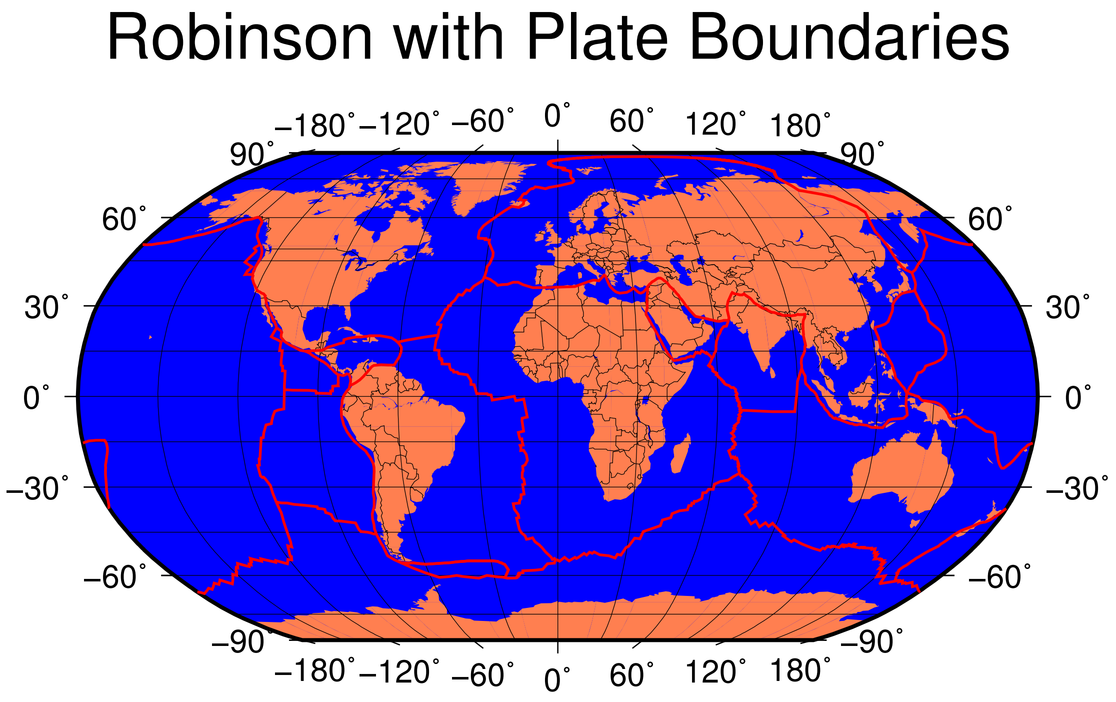
- `-R-90/20/-60/35r` In order to make a non-rectangular projection into a rectangular map, you need to specify upper left and upper right coordinates, rather than minimum and maximum values for each coordinate. The final lower case `r` tells GMT to interpret the `-R` command as lower left/upper right coordinates. Thus, I am telling GMT to plot the rectangular portion defined with the lower left coordinate at $(-90°, 20°)$ and upper right coordinate at $(-60°, 35°)$.
- `-JL-75/27.5/20/35/5i` tells GMT to make a Lambert Conic projection centered at $(-75°, 27.5°)$, with minimum latitude of $20°$ and a maximum latitude of $35°$, and width of 5 inches. Note that this defines how the projection is constructed, but does not determine the final area that will be shown on the map (that is what `-R` specifies). Note that the two are independent of one another, though it is likely that the values will be related, as my central point is at the center of the map and the latitude range matches the range in the region that I have selected.
- The next three options tell GMT how to color the map. `-Gpalegreen` says to color all land areas pale green, `-Sblue` says to color all water areas blue, and `-Cskyblue` says to color all lakes and rivers sky blue. If `-C` is not specified, lakes and rivers have the same color as water areas. You do not need to specify all three of these – you can make a map that only colors land, or only colors water – but you must specify one of `-G`, `-S`, or `-W`.
- The three options following the coloring options tell GMT how to draw map boundaries, including political borders (`-Na`), rivers (`-Ir`), and coastlines (`-W`). There are several options for political borders (`-N1` draws international borders, `-N2` draws national borders within the Americas, `-N3` draws marine borders, and `-Na` draws all three). There are many options for drawing rivers. `-Ir` draws all permanent rivers, and for more options, see the manual. As we saw above, the `-W` option tells GMT how to draw coastlines, and I call `-W` with no options to use the default line style to draw the coastlines.
- `-Di` specifies the map resolution. The following letter can be `f` (full), `h` (high), `i` (intermediate), `l` (low), or `c` (crude). The default is low, which is reasonable for maps at the scale of the first example. This example depicts a smaller region, so I have increased the resolution to intermediate.

The other options should be familiar to you. The output is sent to the file `gmt2_2.ps`. We then add the axis frame and grid lines by calling `psbasemap`. These options should be familiar to you, here we call `-Ba5g5:."Lambert Conic":` or `-Bxa5g5 -Bya5g5 -B+t"Lambert Conic"` to annotate and draw grid lines every 5 degrees. We also give a title. The other options should be familiar, and note that we just give `-R -J` to use the same region and projection as the `pscoast` command.

One important thing to recognize here was that we needed to put `psbasemap` after `pscoast`. This is because successive GMT commands are plotted on top of any previous commands. Since we colored the water and continents, if we drew the grid lines before coloring the land and water, the grid lines would not show up. This is important to keep in mind when you construct complicated maps, as you need to build up the plot layer by layer with different commands.

23.4 Plotting Data on a Map

While making pretty maps is nice, in doing scientific research we would like to plot data on maps. Here is a basic example using `psxy` to draw plate boundaries (included in the file `nuvel_plates.txt`) on a global map `gmt2_3.pdf` in the included shell script:



This plot uses the Robinson projection, a compromise projection designed to look "right" when plotting the entire globe and avoid the area distortion in the Mercator projection. The commands to make this map are as follows (GMT 4):

```
pscoast -Rd -JN5i -N1 -A10000 -Dc -Gcoral -Sblue \
    -X1.5i -Y1.5i -K -P > gmt2_3.ps
psbasemap -R -J -Ba60g30/a30g15:."Robinson with Plate Boundaries": \
    -K -O -P >> gmt2_3.ps
psxy nuvel_plates.txt -R -J -W1p,red -m">" -O -P >> gmt2_3.ps
```

For GMT 5:

```
gmt pscoast -Rd -JN5i -N1 -A10000 -Dc -Gcoral -Sblue \
    -X1.5i -Y1.5i -K -P > gmt2_3.ps
gmt psbasemap -R -J -Bxa60g30 -Bya30g15 -B+t"Robinson with Plate␣
↪Boundaries" \
    -K -O -P >> gmt2_3.ps
gmt psxy nuvel_plates.txt -R -J -W1p,red -O -P >> gmt2_3.ps
```

where the actual commands in the shell script are

```
pscoast -R$region -JN$size $natbounds $area -D$resolution -G$land -S
 ↪$ocean \
    -X$xshift -Y$yshift $firstplot $portrait > $outfile
set code = "psbasemap -R -J $bvals $midplot $portrait >> $outfile"
eval "$code"
psxy $infile -R -J -W${pen},$color $marker $finalplot $portrait >>
 ↪$outfile
```

First, we draw the coastlines using `pscoast`. Here we give a couple of options that you are not familiar with:

- `-Rd` is shorthand for a global domain: `-Rd` is equivalent to `-R-180/180/-90/90`. Also available is `-Rg`, which is the same as `-R0/360/-90/90`.
- `-JN5i` means a 5 inch wide Robinson projection.
- `-A10000` means to not draw any land areas with an area of less than 10000 km ^{2}. This option is useful on large global projections to avoid trying to draw tiny islands.
- The other options should be familiar. We color the continents and water, draw international boundaries, translate the map (to prevent cutting off the title), choose portrait mode, and intend to add more output in later commands.

Next, we draw the basemap frame to draw grids and label the axes. These commands should all be familiar to you. Remember that we do this second so that the grid lines are not obscured by the continents and oceans.

The third command uses `psxy` to draw lines on the map. The command reads data from file (`nuvel_plates.txt`) and plots red lines of width 1 point on the map (`-W1p,red`). `-R -J` should be familiar to you, as well as `-O -P`. The additional option for this command is `-m">"`, which tells `psxy` how to separate different line segments in the input file. If you look at the input file, you will see it is a long list of longitude and latitude coordinates, with each segment separated by a line that begins with >. The option `-m">"` tells GMT to use this to separate the different segments.

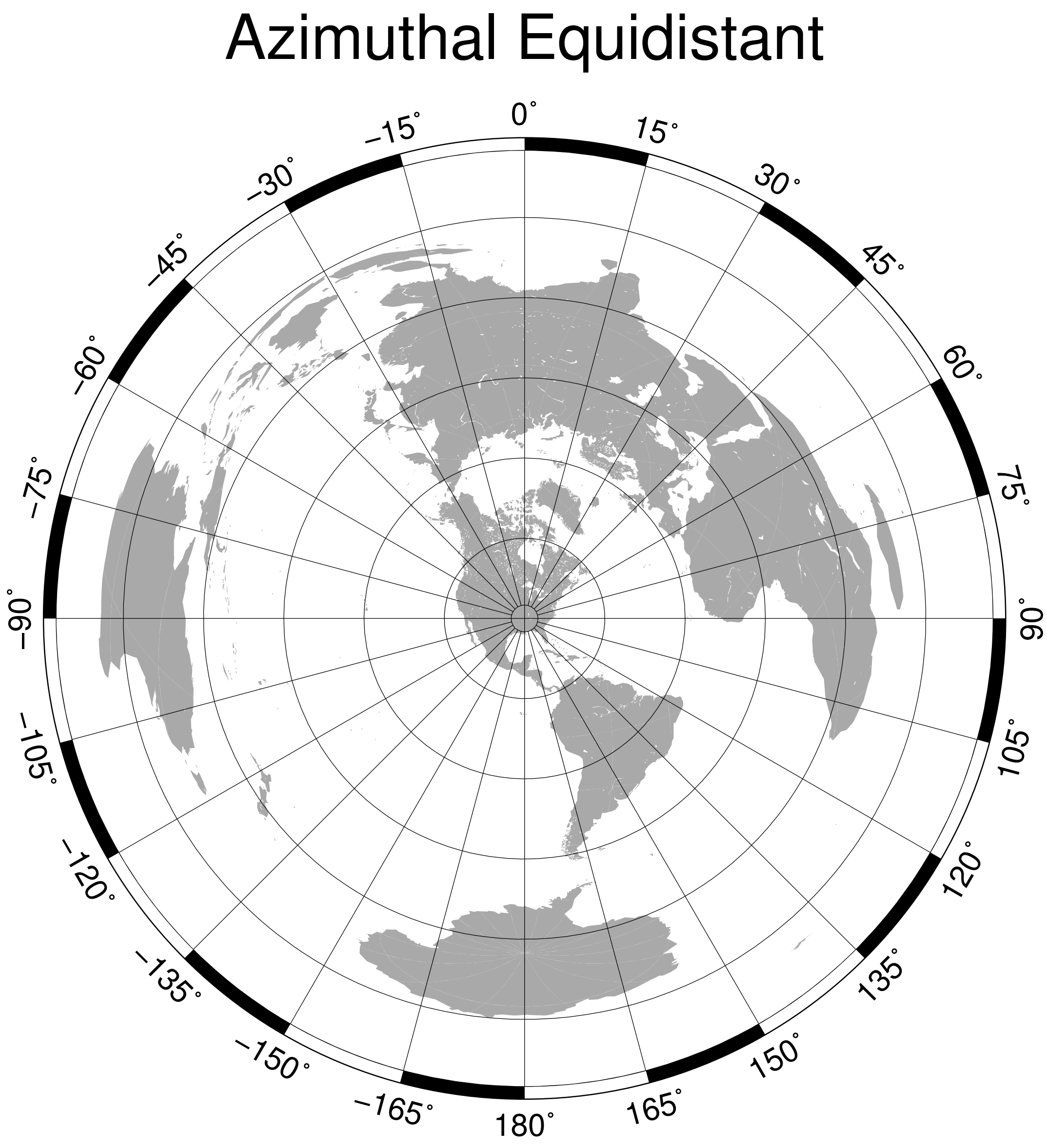
The `-m` option has been deprecated in GMT 5, and the > character is the default for separating line segments in a data file, so that argument is not necessary in GMT 5. To handle this, I set the `marker` shell variable to be an empty string if GMT 5 is detected.

This is a relatively simple example of how to plot data on a map. We will explore this topic in much greater detail in the next lab, showing how to plot additional types of data on map projections with much more complexity.

23.5 A More Complicated Map

The final example shows how to construct a map that is more difficult to make in GMT. This is mostly because the default version of the projection does not make the exact map that I desire, but there are sufficient tools that I can piece it together using several different commands (this is an example of how “Unix makes difficult tasks possible”). This is pretty tricky, so do not worry if you do not understand every detail. If you need to make this kind of map, you can use my script as a “cookbook” example and change the necessary pieces.

One nice map projection for determining earthquake distances and back azimuths to a particular seismic station is the azimuthal equidistant projection. This projection is centered on a particular point, and all lines to that point are great circle paths (this is the “azimuthal” part) and all distances from the central point are accurately represented (this is the “equidistant” part). For instance, here is azimuthal equidistant projection centered on Memphis:



If we plot an earthquake epicenter on this projection, we can easily determine the distance and back azimuth. How do we make this map?

```
psbasemap -R0/360/-90/0 -JS0/-90/90/5i \
    -Ba15:."Azimuthal Equidistant": -K -P > gmt2_4.ps
pscoast -Rg -JE-89.97/35.12/175/5i -Dc \
    -Gdarkgray -K -O -P >> gmt2_4.ps
psbasemap -Rg -JE0/-90/175/5i -Bg15/g30 -O -P >> gmt2_4.ps
```

For GMT 5:

```
gmt psbasemap -R0/360/-90/0 -JS0/-90/90/5i \
    -Bxa15 -B+t"Azimuthal Equidistant" -K -P > gmt2_4.ps
gmt pscoast -Rg -JE-89.97/35.12/175/5i -Dc \
    -Gdarkgray -K -O -P >> gmt2_4.ps
gmt psbasemap -Rg -JE0/-90/175/5i -Bxg15 -Byg30 -O -P >> gmt2_4.ps
```

The code containing variable names is

```
set code = "psbasemap -R0/360/-90/0 -JS0/-90/90/$size $bvals
↪$firstplot $portrait > $outfile"
eval "$code"
pscoast -Rg -JE${clon}/${clat}/${dist}/$size -D$resolution \
    -G$land $midplot $portrait >> $outfile
psbasemap -Rg -JE0/-90/${dist}/$size $bvals2 $finalplot $portrait >>
↪$outfile
```

This is a bit tricky, because GMT's azimuthal equidistant projection does not draw the frame or grids as I want them to appear. However, we can piece together additional commands to draw the frame and grid lines.

The first command draws the exterior basemap frame, and nothing else. Because this frame is not available in the azimuthal equidistant projection, I am instead using a polar stereographic projection centered at the South Pole (`-JS0/-90/90/5i`), but only plotting the frame. Here, the first two numbers are the longitude and latitude of the center point $(0, -90°)$, the third number is the horizon (chosen here to be $90°$ so that the map is for the entire southern hemisphere). The final number is the map size (5 inches).

`-R0/360/-90/0` says to plot the southern hemisphere centered at the south pole, which gives the frame with the correct azimuth annotations that I desire (every 15 degrees, specified by `-Ba15:."Azimuthal Equidistant":` or `-Bxa15 -B+t"Azimuthal Equidistant"`), but without any grid lines (which I will add later). I choose portrait mode (`-P`) and will add more plotting commands later so I add `-K`. Output is sent to `gmt2_4.ps`.

Once I have created the frame with `psbasemap`, I now plot the actual map. I use `pscoast` to plot an azimuthal equidistant projection centered on Memphis. The option `-JE-89.97/35.12/175/5i` says that I want my projection centered at $(-89.97°, 35.12°)$ (the geographic coordinates of Memphis), with a horizon of $175°$ (meaning that I only plot points up to $175°$ away, with a width of 5 inches. The depicted region is global (`-Rg`), and resolution is set to be coarse (`-Dc`). Continents are colored dark gray (`-Gdarkgray`), I use portrait mode (`-P`), and I append to an existing file, with the intention of continuing with further commands (`-K -O`).

A few comments about this projection: in GMT 4 it is known to have some bugs that can lead to problems in depicting the continents. These have supposedly been solved in GMT 5. It also has trouble coloring the map if the antipode (the point $180°$ away) is land. I had trouble getting this map to plot in GMT 4 when I colored both the land and the water, though some of the problems seem to be resolved in GMT 5, and I was able to successfully plot this map in both GMT versions.

The final `psbasemap` command is a trick to draw the desired gridlines on the plot. GMT is only able to draw gridlines that are meridians or parallels on map projections. However, what I wanted for this plot was a set of great circles radiating from the center of the map, and lines of constant distance from the map center. To do this, I used a trick where I plot the grid for the azimuthal equidistant projection centered on the south pole. By choosing this particular map projection, the “meridians” become the great circles, and the “parallels” become the distance contours. (Bob Smalley deserves credit for this, I got this from his GMT notes.)

To draw these gridlines, I call `psbasemap` for a global domain (`-Rg`), with the projection defined by `-JE0/-90/175/5i`. This is the same size and horizon as the `pscoast` command, but because it is centered at the south pole $(0, -90°)$ it will draw the grid lines in the places that I want. Note that `-Bg15/g30` or `-Bxg15 -Byg30` only tells it to draw the grid lines, and the $15°$ increment in longitude matches up with the annotation interval specified in the first `psbasemap` command. The grid commands are appended onto the file.

As I said earlier, I do not expect you to totally understand what I did here. This should illustrate how you can stick together different pieces of code to do something complex, and if you need to make this map projection in the future you can use this code as a cookbook example starting point.

23.6 Exercises

As with the last lab, you should try changing different options as you go through the scripts. Here are some ideas of things to try to change on some of the maps:

- Plot the same projection for a different region.
- Plot a different projection for the same region. Either use one of the other ones we covered today, or learn the syntax for a new projection (see the manual for details).
- Change the grid and annotation spacing, or remove the grid entirely.
- Add or take away political boundaries and rivers.
- Change the fill colors or the coastline outline colors.
- Change the resolution (it is best to do this on a small map, as global maps at high resolution create huge files).
- Plot symbols on the map using `psxy` and standard input.
- Add a text annotation to a map using `pstext`.

CHAPTER

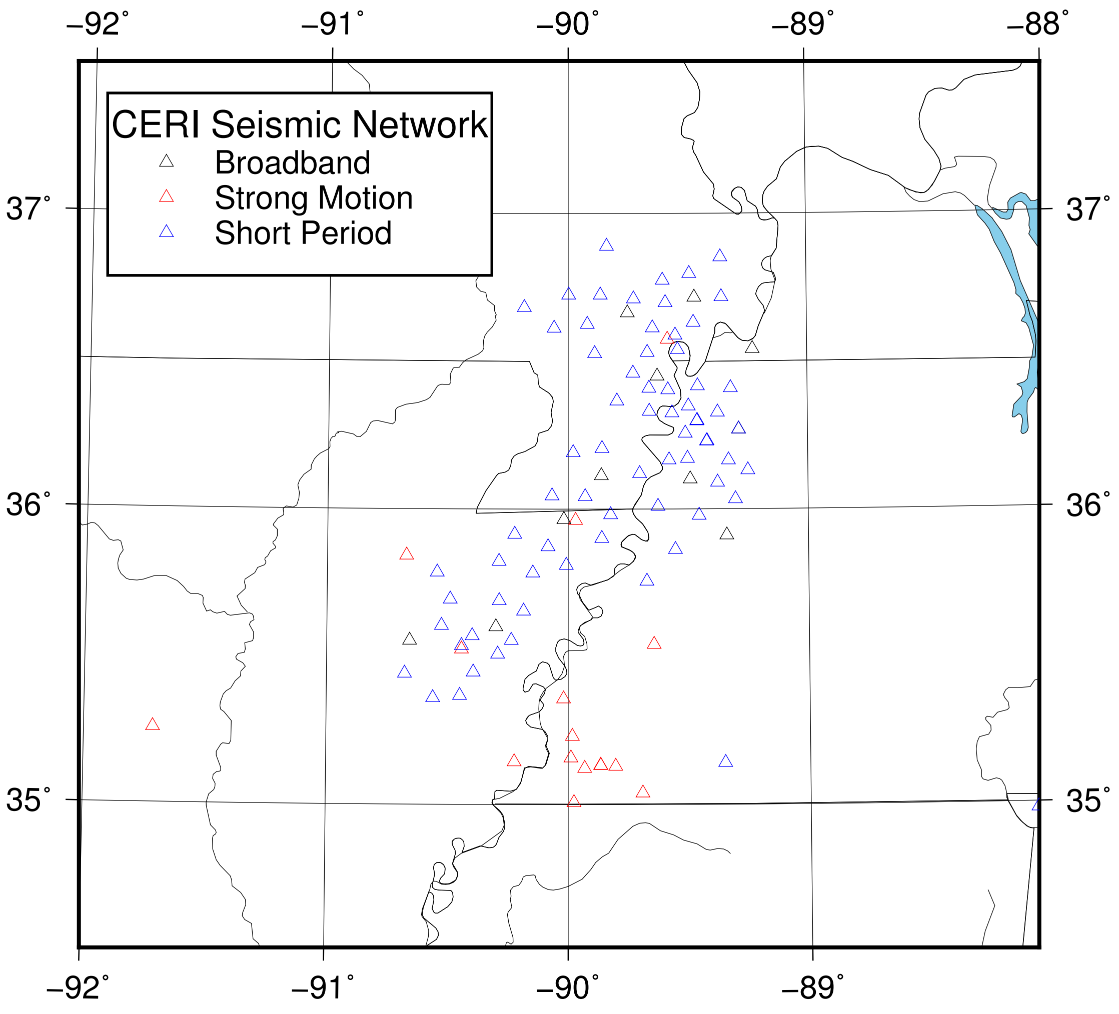
TWENTYFOUR

GENERIC MAPPING TOOLS 3

Geophysicists usually need to make maps in order to display geophysical data. The next two labs will show you various ways that you can do this in GMT. Today we will examine how to display point data on a map (i.e. we have data that has been collected for a certain number of discrete locations), and the final GMT lab we will look at grid-registered data (i.e. data that has been collected for every point gridded over an entire region).

24.1 Displaying Point Data

The simplest type of plot for a point data set is a collection of locations. An example of this might be locations of seismic stations. Here is a plot of the CERI New Madrid Seismic Network, which should be produced by the shell script `gmt3.csh` from the data files `ceribroadband.txt`, `ceristrongmotion.txt`, and `cerishortperiod.txt`.



The code to produce this plot is as follows (GMT 4):

```
set files = ( "ceribroadband.txt" "ceristrongmotion.txt"
↪"cerishortperiod.txt")
set colors = ("black" "red" "blue")
set legendtext = ("Broadband" "Strong Motion" "Short Period")

pscoast -R-92/34.5/-88/37.5r -JB-90/36/34.5/37.5/5i -Ba1g1 \
    -Dh -N2 -Ir -Cskyblue -W -K -P > gmt3_1.ps
foreach i ( 1 2 3 )
    awk '$0 !~ /Station/ {print $4, $3}' $files[$i] | \
            psxy -R -J -St6p -W$colors[$i] -K -O -P >> gmt3_1.ps
end
pslegend << END -R -J -Dx0.15i/3.5i/2i/0.95i/LB \
    -F1p,black -Gwhite -O -P >> gmt3_1.ps
H 14 0 CERI NMSZ Network \\
S 0.25i t 6p - $colors[1] 0.5i $legendtext[1]
S 0.25i t 6p - $colors[2] 0.5i $legendtext[2]
```

```
S 0.25i t 6p - $colors[3] 0.5i $legendtext[3]
END
```

and in GMT 5:

```
set files = ( "ceribroadband.txt" "ceristrongmotion.txt"
↪"cerishortperiod.txt")
set colors = ("black" "red" "blue")
set legendtext = ("Broadband" "Strong Motion" "Short Period")

gmt pscoast -R-92/34.5/-88/37.5r -JB-90/36/34.5/37.5/5i -Bxa1g1 -
↪Bya1g1 \
    -Dh -N2 -Ir -Cskyblue -W -K -P > gmt3_1.ps
foreach i ( 1 2 3 )
    awk '$0 !~ /Station/ {print $4, $3}' $files[$i] | \
            gmt psxy -R -J -St6p -W$colors[$i] -K -O -P >> gmt3_1.ps
end
gmt pslegend << END -R -J -Dx0.15i/3.5i+w2i/0.95i+jLB \
    -F+gwhite+p1p,black -O -P >> gmt3_1.ps
H 14 0 CERI NMSZ Network \\
S 0.25i t 6p - $colors[1] 0.5i $legendtext[1]
S 0.25i t 6p - $colors[2] 0.5i $legendtext[2]
S 0.25i t 6p - $colors[3] 0.5i $legendtext[3]
END
```

Using variables, these commands are written as:

```
set code = "pscoast -R${xmin}/${ymin}/${xmax}/${ymax}r -JB${clon}/$
↪{clat}/${minlat}/${maxlat}/$size"
set code = "${code} $bvals -D$resolution $natbounds $rivers -C${lakes}
↪$continents $firstplot $portrait > $outfile"
eval "$code"

foreach i ( 1 2 3 )
awk '$0 !~ /Station/ {print $4, $3}' $files[$i] | \
    psxy -R -J -S${symb}${symsize} -W$colors[$i] $midplot $portrait >>
↪$outfile
end

pslegend << END -R -J ${legscale} \
    ${legvals} $finalplot $portrait >> $outfile
H $headerfont 0 $header
S $legendoffset1 $symb $symsize - $colors[1] $legendoffset2
↪$legendtext[1]
S $legendoffset1 $symb $symsize - $colors[2] $legendoffset2
↪$legendtext[2]
S $legendoffset1 $symb $symsize - $colors[3] $legendoffset2
↪$legendtext[3]
```

```
END
```

Note that I broke up the command to create a variable holding the `pscoast` onto two lines to make it more readable – you cannot put line breaks into variable definitions, so this is a better way to handle long commands that need to be saved to a variable. Also note that I used a loop in calling `psxy` to draw the stations on the map, using arrays to handle the parts of the variable that change through the loop.

This plot involves several calls to GMT. I will describe them one at a time.

- The first call to `pscoast` draws the map outlines and gridlines. These should all be familiar based on the last GMT lab. If not, refer back to the examples from the previous labs. We use a new conic projection (the Albers Conic projection), which takes the same arguments as the Lambert Conic projection that we used in the last lab.
- The next three calls read x-y data from text files containing station information for the CERI New Madrid Seismic Zone Network. I use AWK to format the data to be used by `psxy` (here just longitude and latitude coordinates for each station). The options for `psxy` were discussed in the first GMT lab. Here we use triangles (`-St6p`) to plot stations (triangles are often used on maps to represent seismic stations).
- The final command uses `pslegend` to draw a legend on the plot, giving the legend data using standard input. `pslegend` takes several options. First are `-R -J`, which define the region and map projection. After that, the `-D` option describes how to draw the legend on the map. You can draw the legend either using map coordinates (degrees) or page measurement units (inches, centimeters, etc.). Here I choose to use page units by prepending the `x` before the measurements. The next four numbers give the x and y location (relative to the lower left corner of the plot) and then the length and width of the legend box (GMT 5 requires you prepend `+w` to designate the size of the box). Finally, the `LB` characters at the end describe the reference point for the location measurements (`LB` means that the x and y location specified in the call refer to the location of the lower left corner of the legend). GMT 5 requires that you place a `+j` before these characters.

  How to specify the style of the legend box is different between GMT 4 and GMT 5. For GMT 4, `-F1p,black` says to draw a rectangular outline in black with a thickness of one point (1/72 inch), and `-Gwhite` says to color the legend box white. GMT 5 requires this information all go under the `-F` option with the format `-F+gwhite+p1p,black`.

  The remaining options have been discussed previously.

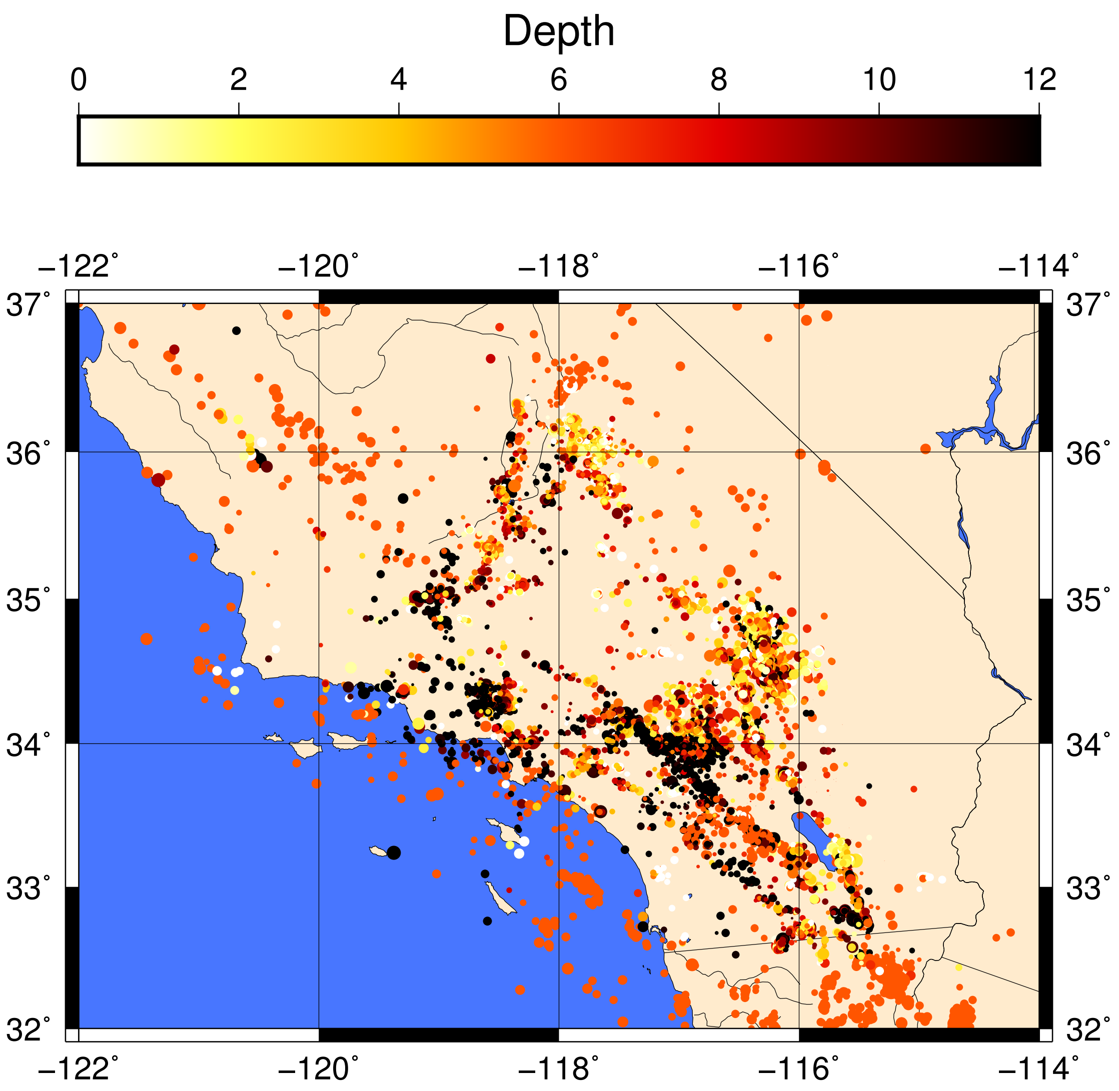
  After the command comes the legend text. Each line begins with a character describing the meaning of the line, and then the details of how to draw that line. The first line begins with `H`, which means that this line contains header text (the legend title). The text that follows (`14 0 CERI NMSZ Network`) means to write "CERI NMSZ Network" in font 0 (Helvetica) with a size of 14 points.

  The following three lines begin with `S`, meaning that they describe symbols to be draw in the legend. the next 5 entries are `0.25i t 6p -$colors[1]`, which describe how to draw the symbol. This means: 0.25 inches from the left of the legend box (`0.25i`), draw a triangle (`t`) with a size of 6 points (`6p`) with no fill (`-`) and a black outline (`$color[1]`, in this case black). After that comes another measurement unit and a text line: the text will be displayed that distance from the left side of the legend box, so in this case the text `$legendtext[1]` ("Broadband" in this case) is displayed 0.5 inches from the left side. The other symbol lines are the same as the first one.

As you can see, legends are a bit tricky. If you have trouble getting the legend to display correctly, you can always draw the symbols and text manually using `psxy` and `pstext` (a trick I use in some of the examples below).

24.2 Displaying Point Data with Size and Color Data

Sometimes, we have point data that are not all uniform, and would like to display this information using the size and color of the points. We can do this using GMT, as illustrated in the following example, which produces the following plot of earthquake epicenters (`gmt3_2.pdf`):



This file uses the earthquake catalog for Southern California for 1999 (`1999.catalog`), and is produced with the following commands in GMT 4:

```
pscoast -R-122/-114/32/37 -JM5i -Dh -Gblanchedalmond -Sroyalblue1 \
    -N1 -N2 -Ir -W -K -P > gmt3_2.ps
makecpt -Chot -I -T0/12/2 -Z -D > quake.cpt
awk '$0 == "" { next } $0 !~ /#|(year)/ { minus = match($8,/-/); \
    print substr($8,minus)-$9/60, $7+substr($8,0,minus-1)/60, $12,
↪$11*0.05 }' \
    1999.catalog | psxy -R -J -Sc -Cquake.cpt -K -O -P >> gmt3_2.ps
psbasemap -R -J -Ba2g2/a1g1 -K -O -P >> gmt3_2.ps
psscale -Cquake.cpt -D2.5i/4.75i/5i/0.25ih -Al -B2:"Depth": -O -P >>␣
↪gmt_2.ps
```

while in GMT 5, the commands are:

```
gmt pscoast -R-122/-114/32/37 -JM5i -Dh -Gblanchedalmond -Sroyalblue1 \
    -N1 -N2 -Ir -W -K -P > gmt3_2.ps
gmt makecpt -Chot -I -T0/12/2 -Z -D > quake.cpt
awk '$0 == "" { next } $0 !~ /#|(year)/ { minus = match($8,/-/); \
    print substr($8,minus)-$9/60, $7+substr($8,0,minus-1)/60, $12,
↪$11*0.05 }' \
    1999.catalog | gmt psxy -R -J -Sc -Cquake.cpt -K -O -P >> gmt3_2.ps
gmt psbasemap -R -J -Bxa2g2 -Bya1g1 -K -O -P >> gmt3_2.ps
gmt psscale -Cquake.cpt -D2.5i/4.75i/5i/0.25i+h+jCT+m -Bx2+l"Depth": -
↪O -P >> gmt_2.ps
```

Using the variable names defined in the shell script, the actual commands are:

```
pscoast -R${xmin}/${xmax}/${ymin}/${ymax} -JM$size -D$resolution -G
↪$land -S$ocean \
        $natbounds $rivers $continents $firstplot $portrait > $outfile
makecpt -C$cmap $cinv -T${cmin}/${cmax}/${cint} $cmapcontinuous
↪$cmapoutliers > $cmapfile
awk '$0 == "" {next} $0 !~ /#|(year)/ {minus = match($8,/-/); \
    print substr($8,minus)-$9/60, $7+substr($8,0,minus-1)/60, $12, $11*
↪'"$symbscale"'}' \
    $infile | psxy -R -J -S$symb -C$cmapfile $midplot $portrait >>
↪$outfile
psbasemap -R -J $bvals $midplot $portrait >> $outfile
psscale -C$cmapfile $scalvals $bvalscmap $finalplot $portrait >>
↪$outfile
```

This involves several new commands, plus some old ones used in a new way.

- `pscoast` does not involve anything new. See the previous lab notes if you do not understand something.
- `makecpt` is a command to make a color palette to plot color data on a map. Here, we create a color palette for plotting our earthquake based on their depth. GMT has several built-in color scales (you can see the list by typing `makecpt` with no arguments into the command line). I choose the "hot" scale by giving the `-Chot` option. However, we need to re-scale the color scale to our data to use it in the plot, which is why we call `makecpt`. I give the options `I` to invert the scale (I want shallow earthquakes to be yellow and deep to be black; since depth is a positive quantity this is backwards), `-Z` to make the color scale continuous (as opposed to discrete), and `-D` to set events that are off scale to be the maximum and minimum colors in the colorscale.

  The final option is `-T0/12/2`, which sets the minimum and maximum values in the color scale range and the increment. The increment is unnecessary here because I chose a continuous color scale, but if you want a discrete color scale this determines the number of colors. You can also give `-T` a file containing all of the values in your data, and it will scale the color palette to the maximum and minimum values in the dataset.

The color palette is redirected to file as `quake.cpt`, which we will use in additional GMT calls that follow.

- Next, we plot the data using `psxy`. Because we would like to set the color from the color palette file, and we would like to scale the marker size by the event magnitude, I give 4 values to `psxy` (longitude, latitude, depth, and magnitude) using AWK to pull the data from the file. This particular AWK program is a bit complicated because of the way the data is formatted (there is no space between the latitude and longitude, so I need to use `match` and `substr` to break the field into two parts). When you call `psxy` with four entries, then the third defines the color and the fourth defines the size. Thus, my plot will show the epicenter as a circle, where the size represents the event magnitude and the color represents depth. Note also that I had to combine single and double quoting when writing my AWK program so that `$symbscale` is correctly expanded to be fed into the AWK program, but the other uses of `$1`, etc. are not owing to my use of quotes. If you are unsure as to why I have to do this, review the notes for the first class on shell scripting.

  The additional options on `psxy` are `-R -J` to use the preceding region and projection, `-Sc` to draw the symbols as circles (note that I do not give a size so that GMT knows to use the data columns to determine the size), `-Cquake.cpt` to use the color palette to color the circles (note that I do not provide `-G` to fill the symbols, so GMT knows to use the data in the input file to set the color), and `-K -O -P` for portrait mode and to append code to a previous file, and to alert GMT that I will append more code to this plot.

- `psbasemap` should be familiar to you. I call it here so that the grid lines are overlaid on the earthquake epicenters.

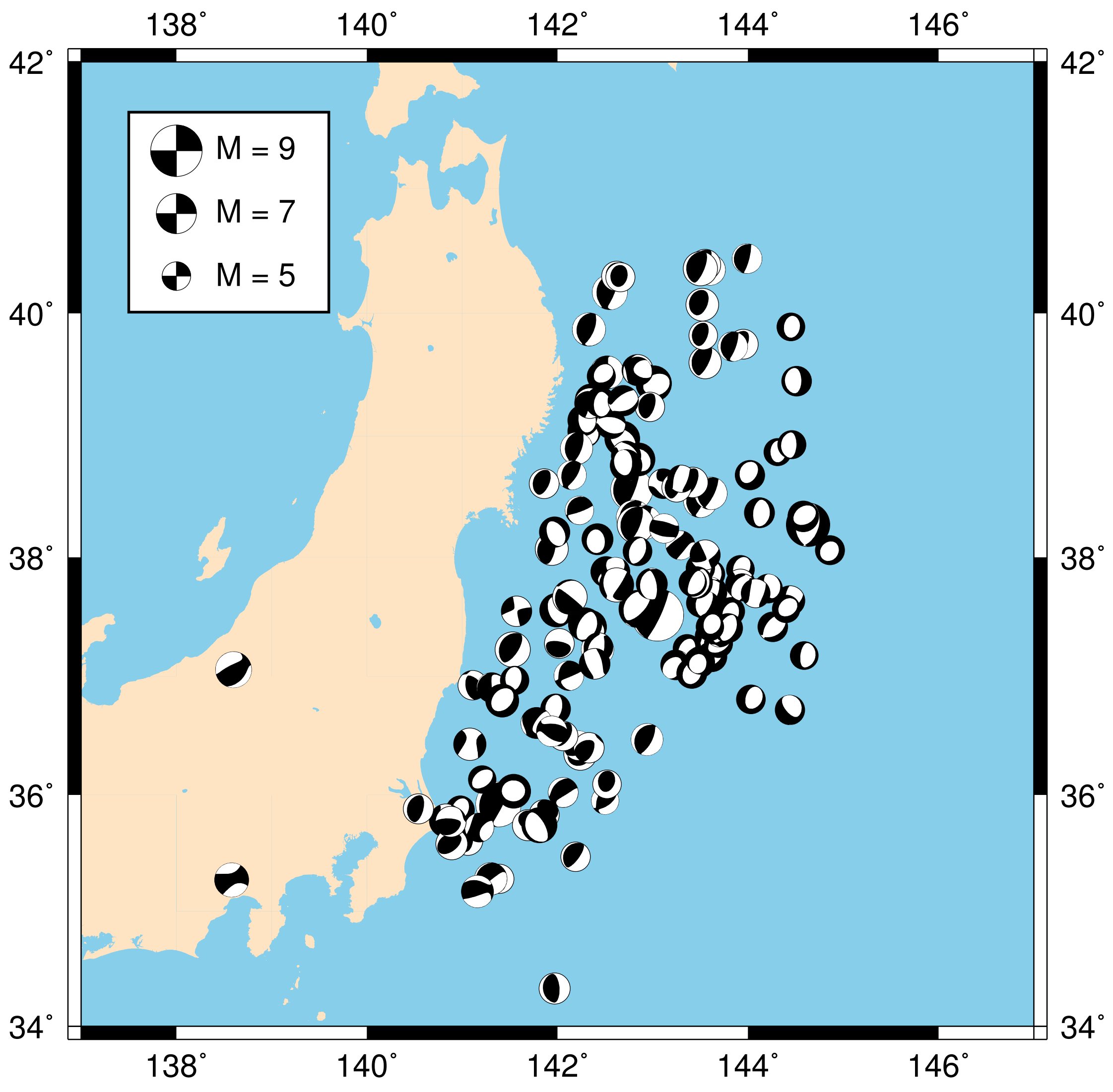
- `psscale` draws the colorbar on the figure. Options are: `-Cquake.cpt` says to use the color palette from before, and the `-D` option tells GMT where to draw the colorbar. The command is slightly different between GMT 4 and GMT 5. In both cases, the first two numbers are the x and y distances from the lower left corner of the plot (relative to the center top of the colorbar), then the next two are the width and height (GMT 5 requires the string `+w` before them). So the colorbar is 2.5 inches to the right and 4.75 inches up from the bottom left corner of the plot. GMT 5 requires that you specify the anchor point, which we give using `+jCT` to ensure compatibility with GMT 4, where that location is the default. The final `h` says to make a horizontal colorbar (default is vertical), and GMT 5 requires a + before the `h` for it to be recognized properly.

  `-Al` tells GMT to put the label on the opposite side of the colorbar in GMT 4. GMT 5 instead appends a `+m` to the `-D` option. Default is to have both the labels and annotations on the bottom side for a horizontal colorbar, and on the right for a vertical colorbar. `-B2:"Depth"` or `-Bx2+l"Depth"` sets the annotation frequency and the label. The rest of the options are ones you have seen before.

As this example shows, you can make very complicated plots using GMT, and have control over a large number of details. Hopefully you get the idea of how all of this works so that you can modify this code should you want to produce a map like this example.

24.3 Earthquake Focal Mechanisms

Another common data type that seismologists often want to show on a map is an earthquake focal mechanism. While the official GMT programs do not include a command to plot focal mechanisms, others have developed a focal mechanism program (`psmeca`) that is now included with the other GMT commands. `psmeca` can plot focal mechanisms in a variety of formats. Here, I show you examples of two of them in the plot `gmt3_3.pdf` created by the shell script.



These are the focal mechanisms of magnitude 5 and greater earthquakes starting with the 2011 $M = 9$ event off of Japan and continuing for 3 months after the event. Here is the code to produce the plot, using the file `japancmt2011.txt`:

```
pscoast -R137/147/34/42 -JM5i -B2 -Gbisque -Sskyblue -Df -K -P > gmt3_
↪3.ps
psmeca japancmt2011.txt -R -J -Sm0.15i -K -O -P >> gmt3_3.ps
```

```
pslegend << END -R -J -Dx0.25i/3.75i/1.05i/1.05i/LB -F1p,black -Gwhite␣
↪\
    -K -O -P >> gmt3_3.ps
G 0.04i
S 0.25i c 1p - white 0.4i M = 9
G 0.11i
S 0.25i c 1p - white 0.4i M = 7
G 0.11i
S 0.25i c 1p - white 0.4i M = 5
END
psmeca << END -R -J -Sc0.15i -O -P >> gmt3_3.ps
138 41.3 0 180 90 0 90 90 0 4 29 138 41.3
138 40.8 0 180 90 0 90 90 0 4 26 138 40.8
138 40.3 0 180 90 0 90 90 0 4 23 138 40.3
END
```

And for GMT 5, the commands are:

```
gmt pscoast -R137/147/34/42 -JM5i -Bx2 -By2 -Gbisque -Sskyblue -Df -K -
↪P > gmt3_3.ps
gmt psmeca japancmt2011.txt -R -J -Sm0.15i -K -O -P >> gmt3_3.ps
gmt pslegend << END -R -J -Dx0.25i/3.75i+w1.05i/1.05i+jLB -
↪F+gwhite+p1p,black \
    -K -O -P >> gmt3_3.ps
G 0.04i
S 0.25i c 1p - white 0.4i M = 9
G 0.11i
S 0.25i c 1p - white 0.4i M = 7
G 0.11i
S 0.25i c 1p - white 0.4i M = 5
END
gmt psmeca << END -R -J -Sc0.15i -O -P >> gmt3_3.ps
138 41.3 0 180 90 0 90 90 0 4 29 138 41.3
138 40.8 0 180 90 0 90 90 0 4 26 138 40.8
138 40.3 0 180 90 0 90 90 0 4 23 138 40.3
END
```

where I have used variables to make the original commands the following:

```
pscoast -R${xmin}/${xmax}/${ymin}/${ymax} -JM$size $bvals -G$land -S
↪$ocean -D$resolution $firstplot $portrait > $outfile
psmeca $infile -R -J -Sm$m5size $midplot $portrait >> $outfile
pslegend << END -R -J $legscale $legvals $midplot $portrait >> $outfile
G $legendvoffsets[1]
S $legendhoffsets[1] $legendsymb $legendsymsize - $legendbackground
↪$legendhoffsets[2] $legendlabels[1]
G $legendvoffsets[2]
```

```
S $legendhoffsets[1] $legendsymb $legendsymsize - $legendbackground
 ↪$legendhoffsets[2] $legendlabels[2]
G $legendvoffsets[2]
S $legendhoffsets[1] $legendsymb $legendsymsize - $legendbackground
 ↪$legendhoffsets[2] $legendlabels[3]
END
psmeca << END -R -J -Sc$m5size $finalplot $portrait >> $outfile
$legendmecx $legendmecy[1] $legenddep $legendmec $legendmantissa
 ↪$legendexp[1] $legendmecx $legendmecy[1]
$legendmecx $legendmecy[2] $legenddep $legendmec $legendmantissa
 ↪$legendexp[2] $legendmecx $legendmecy[2]
$legendmecx $legendmecy[3] $legenddep $legendmec $legendmantissa
 ↪$legendexp[3] $legendmecx $legendmecy[3]
END
```

The commands do the following:

- `pscoast` should be familiar to you. There is nothing new here.
- `psmeca` reads focal mechanism information from the file `japancmt2011.txt`. The file is already in the appropriate format, so I do not need to use AWK to reformat. In this case, the format is

  lon lat dep mrr mtt mpp mrt mrp mtp exp lon1 lat1 text

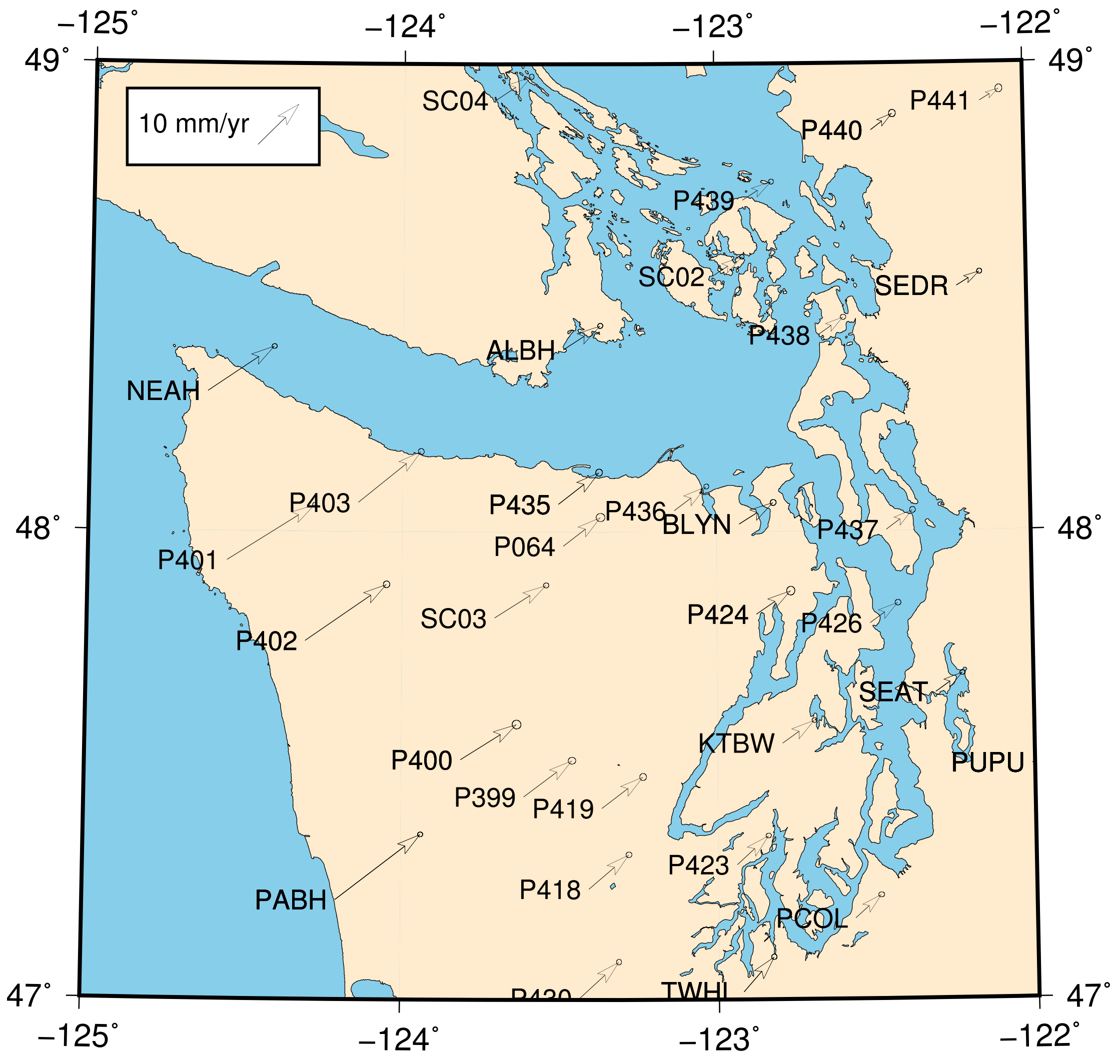
  where the fields have the following meaning: event longitude, event latitude, event depth, 6 component trace-free moment tensor in Harvard CMT format, plus an exponent (assumes seismic moment is in dyne-cm), then the latitude and longitude location for the symbol (I use the same location as the event), and an optional text tag. If you give a location that differs from the event location, the event location will be shown with a small dot that is connected to the beach ball by a line. This is one of the standard formats that you can download from the web (you will use this format to plot focal mechanisms in the homework).

  The other options in `psmeca` should be familiar with the exception of `-Sm0.15i`. This option tells `psmeca` the format for the focal mechanisms (the `m` specifies the format the I just described), and the size (`0.15i` corresponds to the size of a $M = 5$ event). If you want all events to have the same-sized beach ball, use the `-M` option.
- Next, I use `pslegend` to make a legend, where I will illustrate how the size corresponds to magnitude. Since `psmeca` is not an official GMT program, there is not an option within `pslegend` to draw a focal mechanism. Therefore, I will just plot a legend with no symbols, and draw my own symbols using `psmeca`. The syntax here is mostly identical to the previous use in the first example. The only difference is my use of the `G` option in the legend text. `G 0.04i` tells GMT to put a vertical gap of 0.04 inches into the legend, and I use this to get my text lined up with the focal mechanisms that I am about to draw. the `S` lines give the "symbols" that I wish to draw, but since I use a white pen on a white background, the symbols do not show up and I can just add the desired text to the legend in the desired place.

- Finally, I call `psmeca` again to place focal mechanisms on the legend. This time I use the format where I specify the two nodal planes using strike, dip, and rake for each plane (`-Sc0.15i` specifies this format), followed by the mantissa and exponent of the moment in dyne-cm. Since these are only supposed to be representative of size, I draw pure strike-slip mechanisms for each (strike of 180° and 90°, dip and rake of zero in both cases). The moments are chosen to give magnitudes 9, 7, and 5 for the three representative symbols. The other entries (latitude, longitude, depth, and plot position) are the same for all formats. Other options for `psmeca` should be familiar from the discussion above.

24.4 GPS Velocity Vectors with Errors

The final type of point data that we consider in this lab is GPS velocities. GMT has another supplemental program `psvelo` for plotting velocity vectors and error ellipses. Using the file `pbo_velocities.txt`, the shell script will produce the following plot:



The GMT 4 code for this plot is:

```
pscoast -R-125/-122/47/49 -JT-123.5/48/5i -B1 -Df -Gblanchedalmond \
    -Sskyblue -W -K -P > gmt3_4.ps
awk '/^ [0-9A-Z]{4}/ {print $9, $8, $21, $20, $24, $23, $26, $1 }' \
    pbo_velocities.txt | psvelo -R -J -Se30i/0.95/10 \
    -W -K -O -P >> gmt3_4.ps
pslegend << END -R -J -Dx0.25i/4.35i/1i/0.4i/LB -F1p,black -Gwhite \
    -K -O -P >> gmt3_4.ps
G 0.05i
L 10 - LB 10 mm/yr
END
psvelo << END -R -J -Se30i/0.95/10 -W -O -P >> gmt3_4.ps
-124.475 48.825 0.00707 0.00707 0 0 0
END
```

and the GMT 5 code is:

```
gmt pscoast -R-125/-122/47/49 -JT-123.5/48/5i -Bx1 -Bx2 -Df -
↪Gblanchedalmond \
    -Sskyblue -W -K -P > gmt3_4.ps
awk '/^ [0-9A-Z]{4}/ {print $9, $8, $21, $20, $24, $23, $26, $1 }' \
    pbo_velocities.txt | gmt psvelo -R -J -Se30i/0.95/10 \
    -W -K -O -P >> gmt3_4.ps
gmt pslegend << END -R -J -Dx0.25i/4.35i+w1i/0.4i+jLB -F+gwhite+p1p,
↪black \
    -K -O -P >> gmt3_4.ps
G 0.05i
L 10 - LB 10 mm/yr
END
gmt psvelo << END -R -J -Se30i/0.95/10 -W -O -P >> gmt3_4.ps
-124.475 48.825 0.00707 0.00707 0 0 0
END
```

and finally, the version with variables is:

```
pscoast -R${xmin}/${xmax}/${ymin}/${ymax} -JT${clon}/${clat}/$size
↪$bvals -D$resolution -G$land \
        -S$ocean -W $firstplot $portrait > $outfile
awk '/^ [0-9A-Z]{4}/ {print $9, $8, $21, $20, $24, $23, $26, $1}' \
        $infile | psvelo -R -J -Se${veloscale}/${err}/${velofontsize} \
        -W $midplot $portrait >> $outfile
pslegend << END -R -J $legscale $legvals $midplot $portrait >> $outfile
G 0.05i
L $velofontsize - $legendjust $velolabel
END
psvelo << END -R -J -Se${veloscale}/${err}/${velofontsize} -W
↪$finalplot $portrait >> $outfile
```

```
$velolegendx $velolegendy $velolegendcomp $velolegendcomp 0 0 0
END
```

Here are the details of the commands:

- `pscoast` involves a new projection, the Transverse Mercator. Unlike the standard Mercator, you need to provide a central longitude (and optionally a central latitude) to define the projection. This is done by the command `-JT-123.5/48/5i`. Otherwise there is nothing new.
- Next is our first call to `psvelo` The format for input to `psvelo` is

  lon lat Nvel Evel Nerr Eerr NEcor text

  The entries indicate the location of the station, the north velocity component, the east velocity component, the north velocity error (1 sigma), the east velocity error (1 sigma), the correlation between the north and east components, and the optional text representing the station name.

  There are other input formats that are possible (you select the format with the `-S` option). Here we give `-Se30i/0.95/10` as the format: the `e` selects the format detailed above, `30i` gives the length scale for the vectors, `0.95` gives the desired error ellipse meaning (95% confidence bounds in this case), and the final 10 is the size of the text label in points.

  I use AWK to reformat the file and pipe the result into `psvelo`. All of the columns that I need exist in the file, so the AWK program is straightforward other than the regular expression to find a pattern matching station names (line must start with a space followed by 4 alphanumeric characters) and just reprints the information in the correct order.

  The other options in `psvelo` are common ones that we have seen previously.
- Next we create the legend box and text using `pslegend`. The arguments are the same as above so I will not go into detail explaining them. We do use a new type of legend string, `L`, which just prints a text string in the legend. The arguments are font size, font name (– means use the default font), and `LB` describes the justification of the text to be left and bottom aligned. The text is 10 mm/yr, which is the length of the reference velocity vector that I give in the final command.

  Note: it is possible to draw vectors using an `S` entry in `pslegend`. However, I always find it tricky to get the length correct to correspond with the scale of the other arrows since you have to specify the length of the vector in the `pslegend` command. By using `psvelo` to draw the reference arrow, GMT can calculate the length of the vector for me, and I just have to arrange it where I want it in the legend using the latitude and longitude values.
- Finally, we plot the reference velocity vector on the legend. I chose a reference velocity of 10 mm/yr, and I want the vector to be oriented 45° from horizontal. Thus, since my other velocities are in m/yr, I give the velocity components of (0.00707,0.00707) (or each component has a length of $0.01/\sqrt{2}$), which has a magnitude of 0.01. I do not provide label

text, as I provided my own when I called `pslegend`. I do not need an error ellipse for this vector, so I set the uncertainty and correlation to zero.

24.5 Exercises

If you understand how all of these examples work, you will have a nice base of GMT scripts to use to create maps with data for yourself. There are also many, many examples that others have freely shared on the web, so take advantage of them. To get comfortable changing these examples, come up with your own tweaks, or try some of these.

- Change the projection.
- Change the annotations and grid lines.
- Change the color of the continents and oceans.
- Change the symbols that are plotted (for a list see the `psxy` manual page).
- Try a different built-in color scale (type `makecpt` to see them).
- Change the options on the color scale (limits, continuous vs. discrete, invert the color scale etc.).
- Change the scale of the symbols (either the earthquake epicenter sizes, the focal mechanisms sizes, or the velocity vector lengths), or make the focal mechanisms all the same size.
- Change the region for the GPS velocity plot (the data file contains stations throughtout the US, so you will see velocity vectors for most parts of the US).

CHAPTER

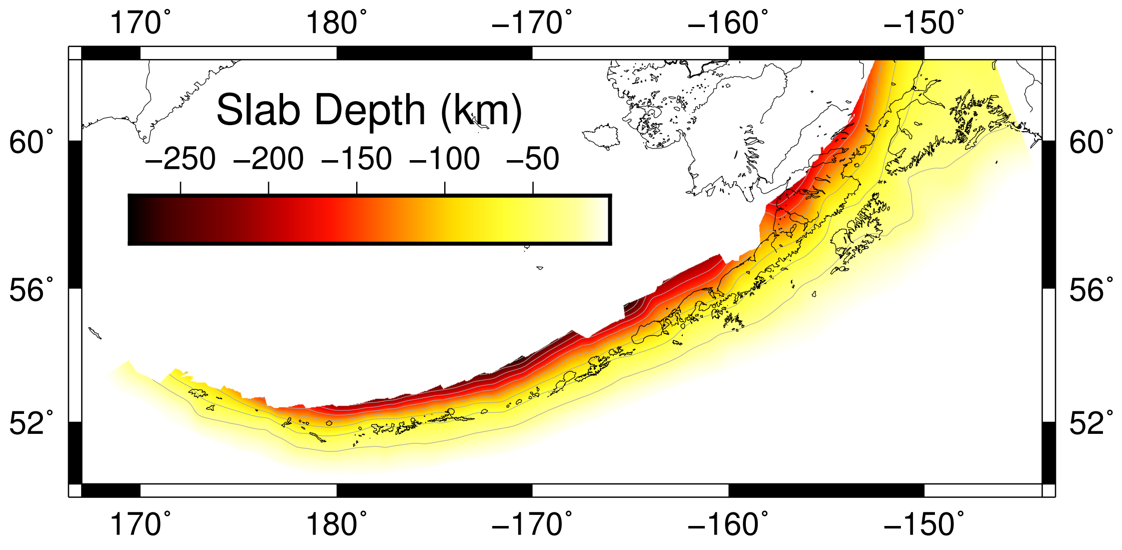
TWENTYFIVE

GENERIC MAPPING TOOLS 4

This final GMT lab will look at plotting grid-based data on maps, which can be a bit more complicated than plotting point-based data. With the increase in availability of densely sampled, high quality geophysical datasets, knowledge of how to plot and display grid-based data in GMT is an essential skill for any geophysicist.

25.1 ASCII Data

The first example uses a dataset containing ASCII data of the geometry of the Alaskan subduction zone. You will need to either copy this file from my public folder into your working directory, modify the shell script to include the full path to its location in my public folder (`/gaia/home/egdaub/Public/dataanalysis/alu_slab1.0_clip.xyz`), or download it from the USGS site. This is the first plot (`gmt4_1.pdf`) produced by the shell script (`gmt4.csh`):



Here are the commands (GMT 4):

```
xyz2grd alu_slab1.0_clip.xyz -Galaskaslab.grd -R167/216/50/62 -I0.02
grd2cpt alaskaslab.grd -Chot -Z > alaskaslab.cpt
grdimage alaskaslab.grd -Calaskaslab.cpt -R -JM5i \
    -Q -K -P > gmt4_1.ps
grdcontour alaskaslab.grd -C30 -J -R -Wdarkgray -A- -K -O -P \
    >> gmt4_1.ps
pscoast -R -J -B10/4 -W -Di -Ir -N1 -K -O -P >> gmt4_1.ps
psscale -D1.5i/1.5i/2.5i/0.25ih -Al -Calaskaslab.cpt \
    -B50:"Slab Depth (km)": -O -P >> gmt4_1.ps
```

and for GMT 5, this is:

```
gmt xyz2grd alu_slab1.0_clip.xyz -Galaskaslab.grd -R167/216/50/62 -I0.
↪02
gmt grd2cpt alaskaslab.grd -Chot -Z > alaskaslab.cpt
gmt grdimage alaskaslab.grd -Calaskaslab.cpt -R -JM5i \
    -Q -K -P > gmt4_1.ps
gmt grdcontour alaskaslab.grd -C30 -J -R -Wdarkgray -A- -K -O -P \
    >> gmt4_1.ps
gmt pscoast -R -J -Bx10 -By4 -W -Di -Ir -N1 -K -O -P >> gmt4_1.ps
gmt psscale -D1.5i/1.5i+w2.5i/0.25i+h+jCT+m -Calaskaslab.cpt \
    -Bx50+l"Slab Depth (km)" -O -P >> gmt4_1.ps
```

with the commands given in the shell script being:

```
xyz2grd $infile -G$gridfile -R${xmin}/${xmax}/${ymin}/${ymax} -I
↪$gridspace
grd2cpt $gridfile -C$cmap $cmapcont > $cptfile
grdimage $gridfile -C$cptfile -R -JM$size \
        $masknan $firstplot $portrait > $outfile
grdcontour $gridfile -C$contourint -J -R -W$contourcolor -A
↪$contourlabel $midplot $portrait \
        >> $outfile
pscoast -R -J $bvals -W -D$resolution $rivers $natbounds $midplot
↪$portrait >> $outfile
set code = "psscale $scalvals -C$cptfile $bvalscbar $finalplot
↪$portrait >> $outfile"
eval "$code"
```

This plot introduces several new commands, which I describe below.

- `xyz2grd` is a command that translates an ASCII file (or binary) containing grid-registered data into a GMT grid file. It requires the arguments given in this command, namely a file from which to read, a destination file to write the gridded data (`-Galaskaslab.grd`), the `-R` flag to specify the extent of the data, and the `-I` flag, which specifies the sampling interval of the grid (in this case, the file is sampled at 0.02 degrees). All of these arguments are required.

- `grd2cpt` then takes the grid file and sets up a color palette for plotting the data. This works in a manner similar to `makecpt` that we saw in the previous lab, but uses the limits of the grid file to determine the color scale (rather than the way I manually set the limits in the example from the last lab). The command takes the grid file as an argument, `-Chot` specifies the color scale, and `-Z` makes the colorscale continuous. The resulting color palette file is written to `alaskaslab.cpt`.
- Now I begin doing the actual map plotting. I would like the coastlines to act as an overlay on the subduction geometry, so I use `grdimage` to display the color scaled image first. `grdimage` takes several arguments, most of which should be familiar to you:
  - First is the grid file, `alaskaslab.grd`
  - `-Calaskaslab.cpt` defines the color palette used to display the data
  - `-R` specifies the region, which is the same as the previous use of `-R`
  - `-JM5i` sets the projection
  - `-Q` is a new option. For this example, we have a grid file that is defined over a region that is larger than the range of the actual data. To distinguish between data points and non-data points, the non-data points have `NaN` as the entry. Each color palette has a set color that it uses to display `NaN` values. Here, I do not want the `NaN` values to display any color, because I would like to see the map coastlines that I will plot later. The `-Q` option makes the `NaN` values transparent, and they will not show up in the final image.
  - `-K` says that more commands are to follow, and `-P` sets portrait mode. Output is then written to file.
- Next, we use `grdcontour` to draw contour intervals on the color image. The options for `grdcontour` are as follows:
  - First, we give the grid file, `alaskaslab.grd`
  - `-C30` defines the contour interval to be 30 km. Alternatively, if you have a discrete color palette, you can call the `-C` option with the cpt file and the contour intervals will be set by the color intervals in the color palette file.
  - `-J -R` are the usual projection and region options
  - `-Wdarkgray` sets the pen used to draw the contour lines
  - `-A-` means that I do not want the contours labeled (I will put a colorbar on the plot later to show the values)
  - `-K -O -P` have their usual meanings, and output is appended to the file
- `pscoast` draws the coastlines. This command does not contain any new options.
- `psscale` draws the colorbar. I have not used any new options here, so if you are unsure of this command, refer back to the notes for the previous class.

As you can see, dealing with gridded data usually involves some additional commands to set everything up before plotting the data.

25.2 Topographic Data

We might expect that there are certain types of data that are likely to be needed by all users of GMT, such as topography and bathymetry data. These types of data are often added to plots in addition to the data in question, as it makes plots look nice and makes them easier to interpret. Because these datasets are also quite large as they are often defined for the entire globe, it makes sense to have them available in one central location that is readable by GMT so that each user does not have to obtain a duplicate copy.

There are a number of such datasets available in the Mac Lab. To see what is available, type `grdraster` (`gmt grdraster` for GMT 5) into the command prompt. You will see something like this (you may need to scroll up to see this):

```
#   Data Description            Unit    Coverage                    ␣
↪Spacing     Registration
-----------------------------------------------------------------------
↪------------------
1   "ETOPO5 global topography"  "m"     -R0/359:55/-89:55/90    -I5m  ␣
↪      GG
2   "US Elevations from USGS"   "m"     -R234/294/24/50         -I0.5m␣
↪      PG
```

This will be followed by a number of other datasets. These are the sets available to you thanks to your friendly system administrators, who went to the trouble of finding, downloading, and setting them up for you. This tells us some of the details of each of these files. The first one is “ETOPO5 global topography” which gives the meters of elevation (negative values for bathymetry) over a grid from $0°$ to $359.55°$ and $-89.55°$ to $90°$ with a spacing of 5 minutes, and the `GG` means the data is grid-registered with geographic coordinates (i.e. values are given for every grid point rather than a pixel value, and the registration of the grid values in the file are in geographic, rather than cartesian order, meaning longitude varies first).

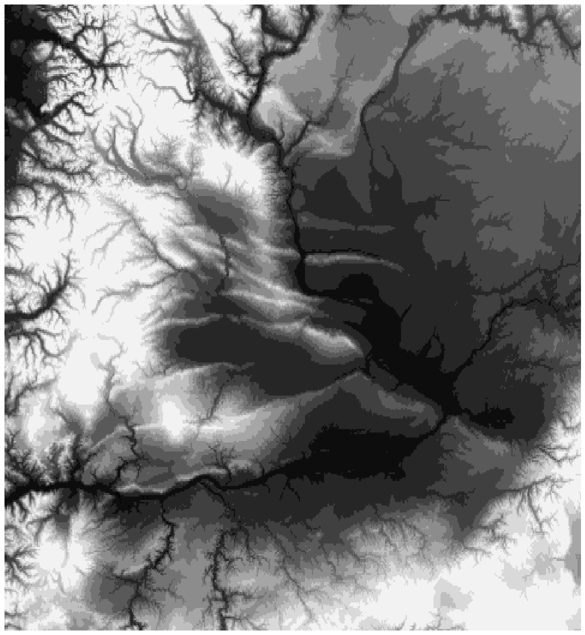
To use these files, we use the `grdraster` command, followed by the file number, and then the other options that are similar to `xyz2cpt` above and write the data to a grid file. We can then display the data using the `grdimage` command as it was used above.

If you want to install GMT on your own computer, then you will need to install these databases yourself, as they do not come with GMT. All of the data is freely available, so as long as you can find the datasets online, download them, and configure the grid information file, you should be able to use the following GMT commands on your own machine without any problems. However, it is even easier to just copy the files off of the Mac Lab computers and into the appropriate directory in your GMT installation. In the Mac Lab, the GMT databases live in `/usr/local/GMT4.5.7/share/dbase/` (you will need to use a terminal to see them).

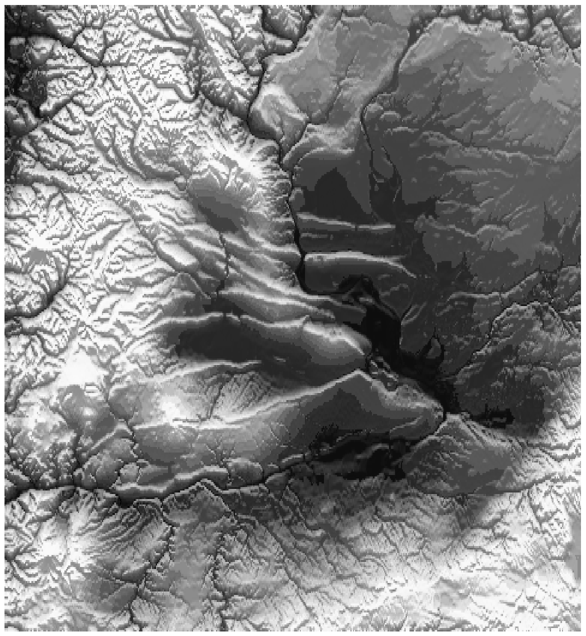
That directory contains the databases, plus a grid information file `grdraster.info` that contains information about them. You need to copy any datafiles that you want to use (note: some of these files are very large, meaning you may need to bring a portable hard drive to copy them), plus the `grdraster.info` file to the directory `/<path_to_GMT>/share/dbase/` on your own computer. (Note: I am not sure how these data files work on Windows.) You are of course free to find your own gridded data to use with GMT in this way, and as more and more high resolution geophysical data becomes available you will no doubt want to use some of it in GMT to produce maps.

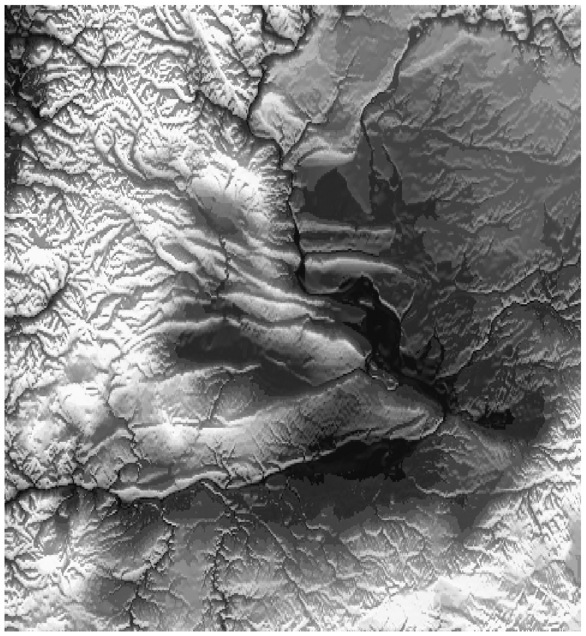
25.3 Topographic Shading

However, if we go ahead and display one of these topographic gridfiles using `grdimage`, we get something like this:



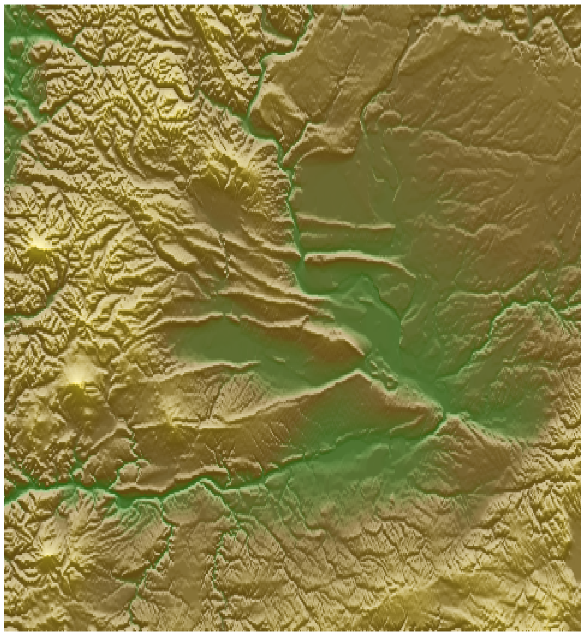
While this looks nice, it is not clear what this is topographically, since we do not have a color scale. This is because our brains are used to looking at topographic maps that are illuminated to help us interpret the topography. However, as with most things where we implicitly fill in the details, this means our brains can play tricks on us. Take the following two images of the same terrain, illuminated differently:

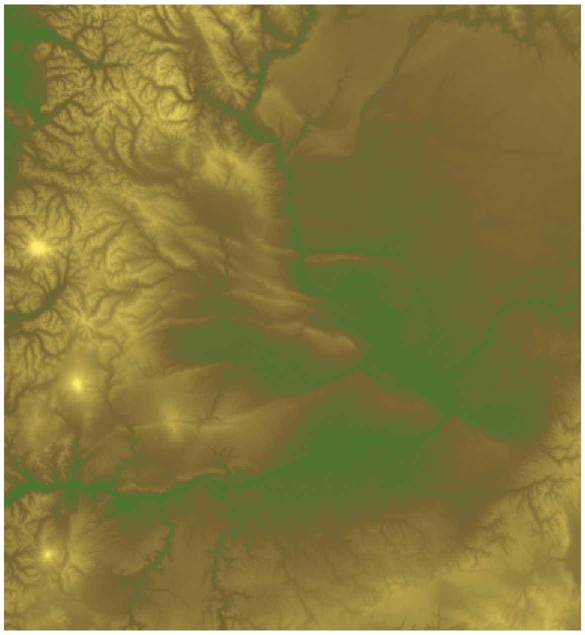




Can you tell what kind of terrain this is? It looks different on the left versus on the right, because one of them is illuminated in a way that you are not used to. The left image is illuminated from the top, while the right image is illuminated from the bottom, and usually these types of images are illuminated from the north (or an angle close to that of north).

Unfortunately, by illuminating we lose some of the quantitative information on altitude (particularly for grayscale images) even if the light and shadows help our brain interpret the image. Thus, the best option is usually to combine both color and illumination. Here is the color version of the image illuminated from the north, side by side with the non-illuminated version:

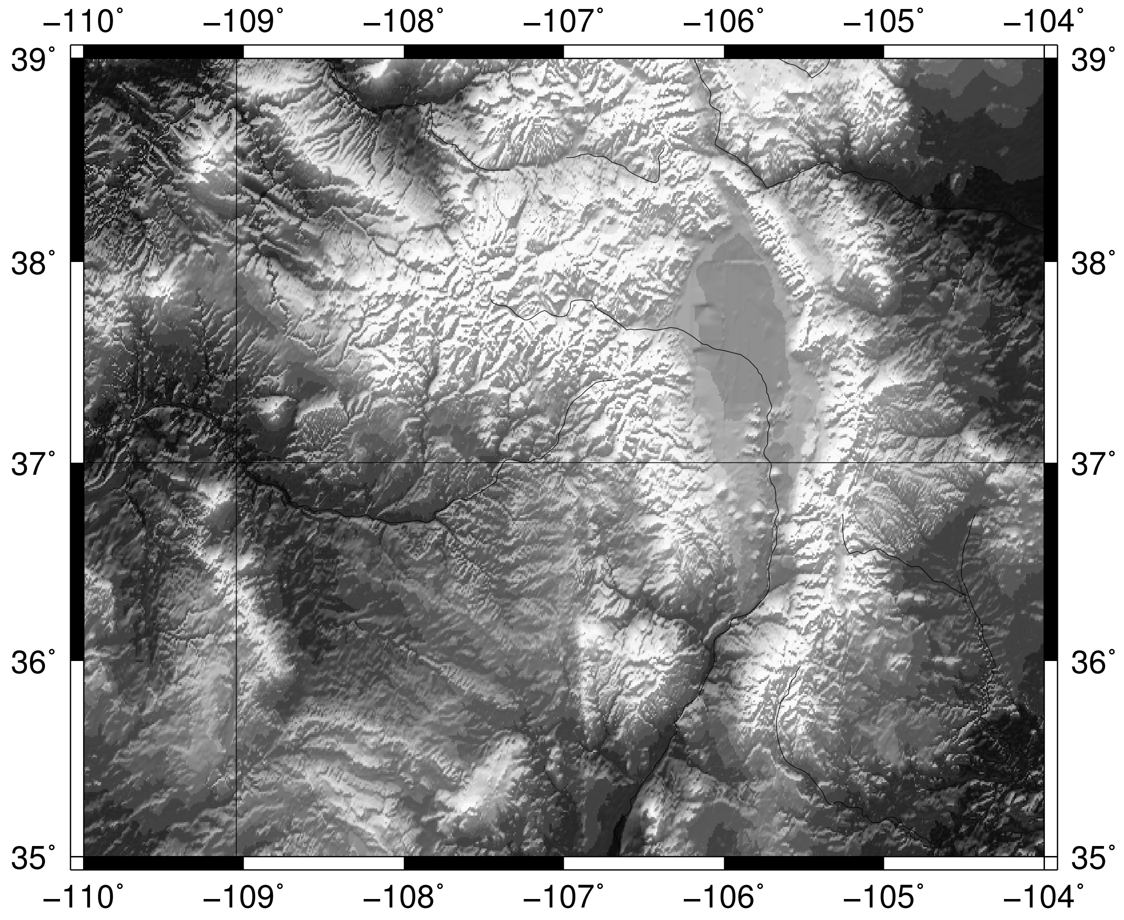




As you can see, the shading helps your brain interpret the topography, and by keeping the color scale, we can better determine which areas are high and which areas are low.

25.4 Making a Shaded Topographic Plot

How do we make shaded plots? Here is an example showing how it is done (the resulting file is `gmt4_2.pdf`):



And the code for this plot is as follows for GMT 4:

```
grdraster 11 -R-110/-104/35/39 -I0.5m -Gfctopo.grd
grd2cpt fctopo.grd -Cgray > fctopo.cpt
grdgradient fctopo.grd -A345 -Gfctopo.intns -Ne0.6
grdimage fctopo.grd -Ifctopo.intns -Cfctopo.cpt -R -JM5i \
    -K -P > gmt4_2.ps
pscoast -R -J -B1 -Dh -Na -Ir -P -O >> gmt4_2.ps
```

In GMT 5, the same plot is produced using:

```
gmt grdraster 11 -R-110/-104/35/39 -I0.5m -Gfctopo.grd
gmt grd2cpt fctopo.grd -Cgray > fctopo.cpt
gmt grdgradient fctopo.grd -A345 -Gfctopo.intns -Ne0.6
gmt grdimage fctopo.grd -Ifctopo.intns -Cfctopo.cpt -R -JM5i \
    -K -P > gmt4_2.ps
gmt pscoast -R -J -Bx1 -By1 -Dh -Na -Ir -P -O >> gmt4_2.ps
```

Using the variables defined in the shell script, the actual code is:

```
grdraster $datasetno -R${xmin}/${xmax}/${ymin}/${ymax} -I$grdinterval -
↪G$gridfile
grd2cpt $gridfile -C$cmap > $cptfile
grdgradient $gridfile -A$intnsdirection -G$intnsfile -N$intnsnorm
grdimage $gridfile -I$intnsfile -C$cptfile -R -JM$size \
    $firstplot $portrait > $outfile
pscoast -R -J $bvals -D$resolution $natbounds $rivers $finalplot
↪$portrait >> $outfile
```

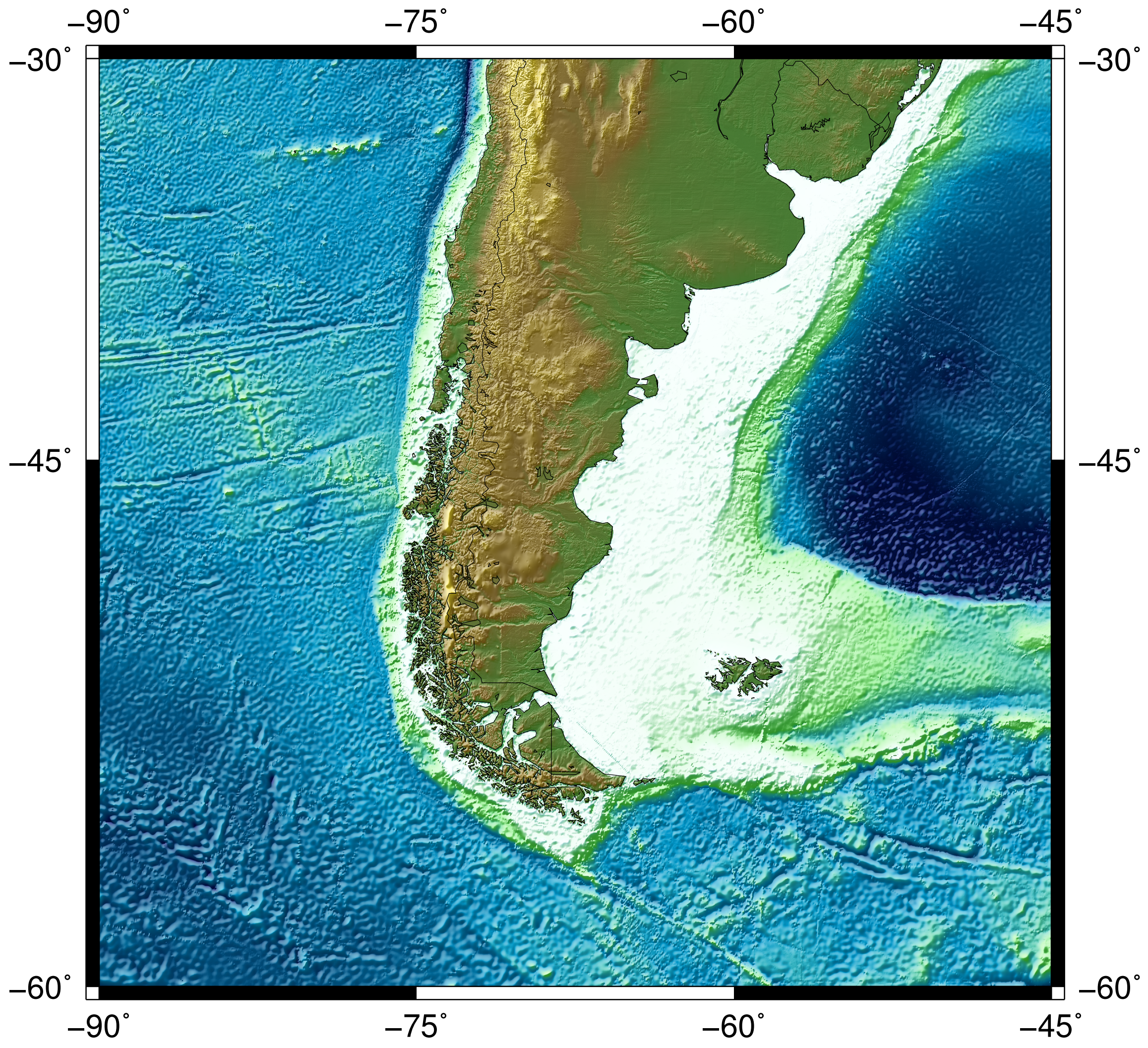
These commands do the following:

- `grdraster` turns the data file into a grid file that GMT uses for plotting. The first argument is `11`, which is saying that I want to use file number 11 (`ETOPO30s` which is global land topography sampled every 30 seconds – type `grdraster` with no arguments to see the different files available). Following that, we have three required arguments: `-R` to define the desired region, `-I0.5m` to determine the grid resolution, and `-Gfctopo.grd` to specify the destination grid file name.
- `grd2cpt` sets the color palette to the range of values in the data. Its use is the same as above (save for a few options that I chose not to use this time).
- `grdgradient` is used to calculate the gradient of a grid in a particular direction, and one of its uses is to calculate the intensities for the topographic illumination. It takes a grid file as its input, and the `-A345` option selects the direction in which to take the gradient (relative to degrees north, clockwise positive). Here I pick an angle 15° from north (345°). `-Gfctopo.intns` sets the output intensity file (really just a grid file, but I like to use `.intns` to remind me that the file is normalized for adding illumination).

  The final option is `-Ne0.6`, which tells GMT how to normalize the calculated gradient. If I wanted the true gradient, then I would not call this option. For doing illumination, we need the gradient to be in the range $(-1, 1)$, as it determines how much to lighten or darken the colors relative to their color value. The option `-Ne0.6` says to normalize the gradient such that the maximum value is ± 0.6, and to have the intensities follow a cumulative Laplace distribution. There are additional distributions and parameters that you can use when calling `grdgradient` (I have found this option produces plots that look nice, but feel free to try different distributions or parameters if you would like a different effect).
- `grdimage` plots the grid file. As before, we give it a color palette (`-C`), and region and projection information (`-R -JM5i`). The intensity for illumination is specified by `-Ifctopo.intns`. `-K -P` have their usual meaning.
- The final command is `pscoast` to draw the international and national borders, plus I color the rivers blue. Output is appended to the file `gmt4_2.ps`.

25.5 Merging Datasets

What if the last map had included an ocean area? The `ETOPO30s` dataset only covers continental areas, but what if we wanted to show an oceanic area's bathymetry? GMT contains tools to combine and merge datasets, which allows you to show high resolution topography on land and lower resolution bathymetry in the oceans on the same plot. While you can do this using transparency, the best way to do this is to merge the two datasets. This allows you to produce plots showing both topography and bathymetry (`gmt4_3.pdf`):



The code to create this plot includes the following commands (GMT 4):

```
grdraster 11 -R-90/-45/-60/-30 -I0.5m -Gpattopo.grd
grdraster 10 -R -I2m -Gpatbath.grd
grdsample patbath.grd -I0.5m -Gpatbathinterp.grd
grdmath pattopo.grd patbathinterp.grd AND = pattopobath.grd
makecpt -Crelief -Z > pattopo.cpt
```

```
grdgradient pattopobath.grd -A330 -Gpattopo.intns -Ne0.6
grdimage pattopobath.grd -Ipattopo.intns -Cpattopo.cpt -R \
    -JM5i -K -P > gmt4_3.ps
pscoast -R -B15 -J -W -N1 -O -P >> gmt4_3.ps
```

For GMT 5, the commands are:

```
gmt grdraster 11 -R-90/-45/-60/-30 -I0.5m -Gpattopo.grd
gmt grdraster 10 -R -I2m -Gpatbath.grd
gmt grdsample patbath.grd -I0.5m -Gpatbathinterp.grd
gmt grdmath pattopo.grd patbathinterp.grd AND = pattopobath.grd
gmt makecpt -Crelief -Z > pattopo.cpt
gmt grdgradient pattopobath.grd -A330 -Gpattopo.intns -Ne0.6
gmt grdimage pattopobath.grd -Ipattopo.intns -Cpattopo.cpt -R \
    -JM5i -K -P > gmt4_3.ps
gmt pscoast -R -Bx15 -By15 -J -W -N1 -O -P >> gmt4_3.ps
```

Within the shell script, this is called as:

```
grdraster $topofileno -R${xmin}/${xmax}/${ymin}/${ymax} -I
↪$topogrdinterval -G$topofile
grdraster $bathfileno -R -I$bathgrdinterval -G$bathfile
grdsample $bathfile -I$topogrdinterval -G$bathfileinterp
grdmath $topofile $bathfileinterp AND = $mergedfile
makecpt -C$cmap $cmapcont > $cptfile
grdgradient $mergedfile -A$intnsdirection -G$intnsfile -N$intnsnorm
grdimage $mergedfile -I$intnsfile -C$cptfile -R \
    -JM$size $firstplot $portrait > $outfile
pscoast -R $bvals -J -W $natbounds $finalplot $portrait >> $outfile
```

Here are the details on these commands:

- The first two commands call `grdraster` as above. The first call to `grdraster` uses the same land-based `ETOPO30s` dataset that we used above, which is sampled at 30 second intervals. The second call to `grdraster` uses dataset 10, which is a global 2 minute sampling rate topography/bathymetry dataset (type `grdraster` for more information on this dataset). We choose the same region for both, and sample both datasets at their full resolution (to sample the data at a lower rate, you can change the sampling rate on the `-I` option).
- The two grid files need to be merged, but first we must fix the fact that they are sampled at a different rate. This is done using `grdsample`, which resamples a dataset to the desired grid spacing. We choose to resample the bathymetry dataset to match the higher resolution of the land topography dataset. The sampling interval is set by the flag `-I0.5m` to 30 seconds so that the data sampling rate matches between the two datasets.

  If the new sampling rate is chosen to be higher than the previous sampling rate, `grdsample` interpolates. If the new sampling rate is more coarse than the existing data, it desamples the data (i.e. it simply takes every nth point, without applying an anti-aliasing filter, so take care when desampling data that varies at a high rate).

- The next command merges the two datasets using the `grdmath` command. `grdmath` does mathematical operations on datasets, using what is known as Reverse Polish Notation, or RPN. RPN puts the operator after the operands, using what is known as a "stack" to hold results. When using an unary operator, the operation is applied to the first item in the stack, and leaves the result in the first position on the stack. When using a binary operatory, the operation is applied to the first two items in the stack, and leaves the result in the first position on the stack (if there were any items higher up in the stack, they are shifted down a position). Array math is done element-by element (i.e. you cannot do matrix multiplication using `grdmath`, only element-by-element array multiplication).

  For example, if you have one grid file that you would like to take the additive inverse of (the `NEG` operator), then you would enter the following:

  ```
  grdmath a.grd NEG = b.grd
  ```

  This would write the additive inverse of the data in `a.grd` to the grid file `b.grd.` To add two grid files, use the `ADD` operator and put the two grid files on the stack:

  ```
  grdmath a.grd b.grd ADD = c.grd
  ```

  This will write the sum of the two grids to a third grid file. For our case, we need to use the `AND` operator. `AND` is a binary operator, which keeps the result in the first grid file, unless that file contains a `NaN`, in which case the operation keeps the result in the second grid file. Because all of the oceanic areas are `NaN` entries in our `ETOPO30s` data, we can replace the `NaNs` on the oceans with the bathymetry data from the second dataset. Thus, the resulting command is:

  ```
  grdmath pattopo.grd patbathinterp.grd AND = pattopobath.grd
  ```

  The result is written to the file `pattopobath.grd`, which we then use to plot our data.

- `makecpt` creates the color palette for displaying the data. I set `-Crelief` to use the "relief" color palette, and since it is already scaled for use with topography and bathymetry, I use it as is without modifying the limits. My only change is to make it continuous using `-Z`.

- `grdgradient` calculates the gradient and normalizes so that we can use the intensities to illuminate our plot. The options are the same as above.

- Next, we use `grdimage` to plot the topographic data on our map. The command syntax is discussed above, and does not use anything new.

- Finally, we draw the coastlines and boundaries. There is nothing new here, either.

The end result is a map showing both topography and bathymetry. You can also illuminate and shade the bathymetry and topography from slightly different directions, then merge the two instensity datasets in a similar fashion as the grid merging example above. This will make the final file look a little bit different, though I think the example above with a single illumination direction is just fine and do not see the need to go to extra trouble.

25.6 Miscellaneous GMT

25.6.1 Setting GMT Defaults and Parameters

GMT lets you specify absolutely any detail that you wish for your plots. As you have seen, there are many details that you specify with various command options. However, there are even more details, such as default line styles, default font names and sizes, default tick mark lengths, and on and on and on. There are several ways to set these parameters.

If you recall back to our labs on MATLAB, there were two commands to query and specify the style of plots: `get` and `set`. `get` allowed you to see the value of a particular option, while `set` allowed you to change a particular option. GMT has a similar syntax for querying and setting parameter values, using the commands `gmtget` and `gmtset`. These work in much the same way as the equivalent MATLAB commands. For a list of options, enter `gmtdefaults` into the terminal, or see the manual page for `gmt.conf`.

Alternatively, you can save default GMT values a file called `gmt.conf` in your working directory, home directory, or in the `~/.gmt/` directory that contains configuration options for GMT. If you do not have a `gmt.conf` file in your current directory, you can create one by typing `gmtset -D` into the terminal, or by setting a parameter value. The one in the current working directory always takes precedence, so you may want to keep a set of defaults in your home directory, but override the defaults in a working directory for a particular project.

25.6.2 Converting PostScript to Other Formats

Once you have created a nice GMT map, you may need to save it to a different format for use in a paper or presentation. While the Preview application on the Mac is able to do this, you can also convert formats on the command line using the `psconvert` command that is included with GMT. We have in fact been using this to convert files to PDF in all of the shell scripts, so this isn't the first time we have come across this command, but here I provide a bit more detail of what it does. `psconvert` uses Ghostscript to convert between file formats. An example usage to convert a PostScript file to PDF would be:

```
psconvert -A0.1i -Tf gmt4_1.ps
```

The flag `-A0.1i` means to crop the file with a border of 0.1 inch (leaving the border size blank crops it using a tight bounding box, I usually like to add a bit of a cushion and find 0.1 inches is a good number), and `-Tf` sets the output format to PDF. For other output formats (there are several), see the manual. If you are converting to a raster format like PNG or TIFF, the `-E` option lets you specify a resolution; `-E300` will save as 300 dpi.

25.7 Exercises

If you understand how all of these examples work, you will have a nice base of GMT scripts to use to create maps with data for yourself. There are also many, many examples that others have freely shared on the web, so take advantage of them. To get comfortable changing these examples, come up with your own tweaks, or try some of these.

- Change the projection.
- Change the annotations and grid lines.
- Change the color palette.
- Change the illumination direction.
- Try a different built-in dataset (type `grdraster` to get more information on them).
- Change the options on the color scale (continuous vs. discrete, invert the color scale etc.). If you want to set the limits yourself, use `makecpt` with the `-T` option (see the previous lab).
- Change the grid resolution either using `grdsample` to resample the data or by using the `-I` option when calling `grdraster`.
- Change the region for one of the topography plots.

CHAPTER TWENTYSIX

INDICES AND TABLES

- genindex
- search