Interactive Data Language Guide

Chris Lowder

May 10, 2010

Contents

Chapter 1

WebGDL

In your time as an REU student, you'll most likely be coding in the IDL language, or Interactive Data Language. This software is installed on the main solar servers, and you'll connect remotely from another machine to run the programs you design. Since you won't be able to access an IDL compiler until you arrive for the program, we've found a free alternative that you can play about in. Think of it as a sandbox to experiment with coding in IDL before you begin your research.

1.1 Web access

A link will be provided to you via email that will enable use of the WebGDL interface. For server security reasons, please do not share this address with anyone online. Thanks!

1.2 Command layout

There are several sections laid out in the online interface. There is a graphic window in the upper left corner, a command line below this, and a file system to the right. Commands are entered into the command line one at a time, or run through in a series as defined in a program (.pro file).

1.3 Persistence of files

To keep your programs intact between online sessions, it would be wise to compose them in an external program, such as notepad. One can then copy and paste the code into the WebGDL interface as a .pro file. After successfully debugging and running any potential code online, one can simply copy and paste the entire program back into your local machine for safe-keeping.

Chapter 2

Introduction to Programming

While computers are quite quick to perform calculations, all of this raw power is useless without some sort of purpose to it. Programming, from the earliest days of punching code into notecards to now punching code on a keyboard, is designed with the intent to harness this raw computing power purposefully and efficiently.

There are many languages in which one can program a computer, and IDL is just one of many. Just as many actual languages share the same structure and sentence components, computer languages are quite similar. Much of the logic and structure is identical across languages, and ultimately it comes down to the syntax where things differ. So, let's learn the syntax.

2.1 IDL Prompt

Begin by opening the GDL / IDL program in a terminal. After an opening sequence it should display a small, lonely prompt of,

```
GDL>
```

You can enter commands directly into the prompt, or choose to write a series of commands bundled together in a program file. We'll be dealing more with writing code into a program file, and then compiling and executing that program in the prompt.

2.2 Variables

Similarly to the memory function on a calculator, variables enable an IDL program to store information. Note that information implies that a number of things can be stored in a variable. This includes arrays of numbers and even text! There are many classifications for a variable, but we'll just cover a few that will be more relevant to your project.

2.2.1 Scalar

Integer

An integer is a whole number, defined in a way such as:

```
a = 42
```

Float

A floating number allows the variable to contain non-integer numbers, such as:

```
b = 42.23
```

Or even,

```
c = 42.
```

You may ask yourself, why go to all the trouble of adding that period for the definition of our variable c? What's the difference between 42. and 42? When doing any sort of math operations between our variables a, b, and c here, the careful programmer wants to make sure that all variables involved in an operation are of the same data type. If this isn't the case... strange things will soon become afoot.

String

A string is a series of text characters stored into a variable. To define a string, one would issue the command,

```
d = 'I do not like Spam!'
```

Now, just as there are operations to act on our numerically oriented variables, there are operations which act on string variables as well. We'll discuss some of these a bit later.

2.2.2 Vectors and arrays

Just as a string is nothing more than a series of letters arranged in a particular order, an array is most often just a series of numbers arranged in a particular order. You could define one as,

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

Numbers can then be referenced based on their position within the array. For instance, to access the third element in the array,

```
print, f[2]
```

This is quite strange, however, as I thought we wanted the third element in the array. As it turns out, IDL references elements in an array beginning with 0. So to access the first element, one would use the command,

```
print, f[0]
```

Arrays, however, aren't limited to just one dimension though. They can extend even beyond the perceivable three dimensions. To define an array of numbers, the simplest way is via,

```
g = fltarr(dim1, dim2, dim3, ...)
h = intarr(dim1, dim2, dim3, ...)
```

Where fltarr will create an array of float-type variables, and intarr will do the same with integers. Note that the number of elements in each dimension are then listed. For example, to create a 512 by 512 by 100 array of floating numbers,

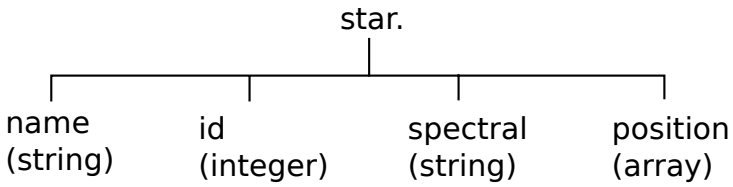
```
i = fltarr(512, 512, 100)
```

2.2.3 Structures

Just as we defined earlier a regular number variable, and then an array of those individual numbers, we can also go one step further. Think of a structure as a way to group together multiple sorts of information into one package. The information doesn't even need to be of the same type! You can bundle together strings, numbers, and even arrays into your structure. This

makes dealing with large datasets a little easier. Now, how do we visualize, define, and then access these structures?

The visualization step is crucial to understand how your data is floating around inside IDL's memory. Let's say we have a particular star that is of interest to us, and we wish to bundle several bits of data together regarding this star. First, the structure must have an overall name, let's call it... 'star.' Think of this as a large folder, which can contain many smaller folders. We can then define sub-categories within our 'star' structure.



To actually define the above structure, we'd use the following command,

```
star = CREATE_STRUCT('name', 'sol',\\
          'id', 0, 'spectral', 'G', 'position', [0,0])
```

This will create the structure 'star', with the elements that we've provided it. Now, how does one retreive this stored information. The command, "help, star", won't yield very much useful information. Instead, we must append the tag, /str, so that IDL knows what it is dealing with. This is accomplished via, "help, star, /str" This should print all of the sub-structure elements contained in our structure, as well as the data-type of each.

To access the data in any of these fields, simply type the structure name, a period, and then the field name. For instance, to print the coordinates of the given star, we'd type,

```
print, star.position
```

We can then either recall this information for use, or even make modifications to the entry in this fashion.

2.3 Mathematical and String Operations

2.3.1 Normal math operators

For the types of variables we've defined thus far, the regular mathematical operations you're used to will work. For instance, try the following commands in the IDL prompt, and study the results.

```
print, b + c
print, b - c
print, b / c
print, b * c
```

And as you might guess from the statement above, the print command can output operations as well as regular variables. Also note the keyboard characters used for the division and multiplication operations.

2.3.2 String operators

Now strings are another story altogether. You can't exactly add 'I do not like Spam!' to 'Some other string' in the traditional sense. That just doesn't make any sense. But, strings can be thought of as a sequence of letters which can be broken up, combined, and rearranged.

First of all, the addition command will work, but not in the way one is used to. For example, consider the command,

```
print, 'I do not ' + 'like Spam!'
```

Which outputs,

```
I do not like Spam!
```

In this case, the addition operator simply appends the strings to one another, creating a new string containing both.

Say we now wish to break a string apart into multiple chunks. To accomplish this, we'll use the command, strmid. Consider the following general command structure,

```
print, strmid(string, lettertostartbreakingat, numberofletterstobreakoff)
```

And consider the example now,

```
theword = 'something'
print, strmid(theword, 4, 5)
```

These commands will define the string, `theword`, and then print the chunk of it displaying, `thing`.

2.3.3 Boolean logic

While humans are truly capable of thinking in shades of grey, the computer is not so lucky. Boolean logic can be used to compare sets of numbers, arrays, etc, based on the premise that the state 1 is true, and that 0 is false. For instance, let's say we wish to compare two numbers, to see which is larger. First, we can make sure they aren't equal,

```
print, 5 eq 4
  0
```

The command `eq` will check to see if the two statements are equal, in which case it will return 1, and otherwise, 0. The corresponding command, `ne`, will check to see if the two elements are **not** equal. While on the subject of reverse commands, one can reverse any logic statement with the addition of the $\sim$ character. For instance,

```
print, ~(5 eq 4)
  1
```

One can also use the standard suite of greater than, greater than or equal to, less than, and less than or equal to, expressed respectively below.

```
print, 5 gt 4
  1

print, 5 ge 4
  1

print, 5 lt 4
  0
```

```
print, 5 le 4
  0
```

Boolean statements can also be combined via the `and` or the `or` commands. They require the component boolean statements to either both be true, or for at least one to be true, respectively. For further study, consider the two examples below.

```
print, (2 gt 1) and (1 lt 2)
  1
print, (7 gt 3) or (2 gt 6)
  1
```

Now, one can also compare matricies using boolean logic. For sake of argument, let's say we wished to isolate the elements of a 5x5 array that are greater than a particular value. Consider the matrix, **m**, below.

$$\begin{pmatrix} 3 & 5 & 1 & 0 & 6 \\ 8 & 2 & 1 & 4 & 3 \\ 2 & 8 & 6 & 2 & 3 \\ 0 & 1 & 8 & 2 & 6 \\ 2 & 5 & 3 & 3 & 9 \end{pmatrix}$$

We can in essence create a 'mask' using boolean logic to pick out the values greater than a certain value, say 2. To create this mask, we can use the regular boolean logical statements, except that now it will return a matrix of boolean values, rather than just a single number.

The command,

```
mask = m gt 2
```

will yield the boolean matrix,

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

We can then multiply this mask by our original array, and only those values that meet our criteria will remain!

2.3.4 Sizing

Knowing the size of a vector or array is quite important when dealing with them in a more automated program. For a one-dimensional array, we can use the command, `n_elements`.

```
m = [1,2,3,4,5]
print, n_elements(m)
  5
```

When dealing with a two-dimensional (OR MORE) matrix, we need to employ the `size` command.

```
m = [[1,2,3],[4,5,6],[7,8,9]]
print, (size(m))[1]
print, (size(m))[2]
```

The output will return the number of columns and rows, respectively. For futher information on the output parameters of `size`, see the IDL help files.

2.4 Display

Now, we've learned how to manipulate data within IDL. However, we still need a useful way to output this data, or all of it will be in vain

2.4.1 Print

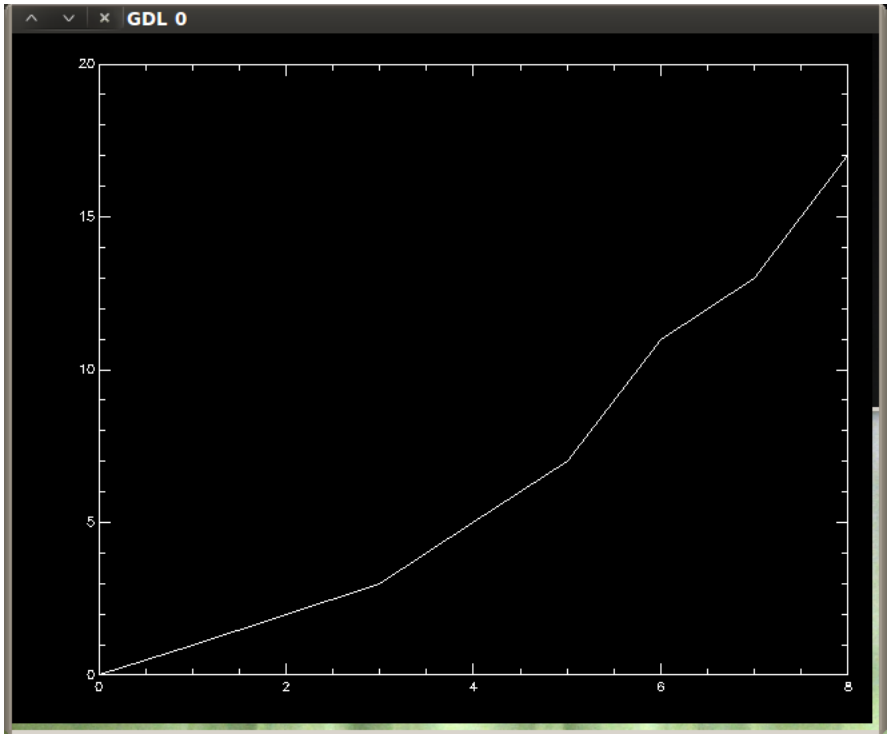
You've seen the `print` command through usage in almost all of the preceeding examples. So hopefully you should be familiar with it's usage by now. However, were you aware that you can also print multiple items on the same line? For instance,

```
print, a, b, c, d
```

2.4.2 Plot

Now, the print command is useful for single number variables, strings, and perhaps small vectors and arrays. However, for arrays with a large number of items, or for situations where direct visualization of your data is simpler, plot is the way to go. Try the following command.

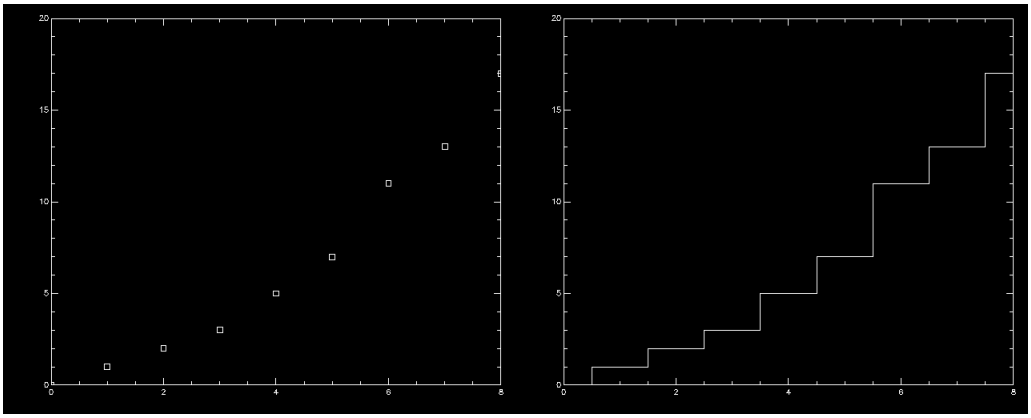
```
v = [0,1,2,3,5,7,11,13,17]
plot, v
```



You can see that IDL plotted each number accordingly, with values specified on the y-axis. One can supplement the `plot` command with many options, one of which defines the line-type. For instance, by using a command such as,

```
plot, v, psym=5
plot, v, psym=10
```

One can produces plots similar to those below. Try different values of psym to suit the data that you are plotting.



In addition to plotting single arrays on the same plot, one can overlay multiple plots together by simply plotting the initial dataset as usual, and then calling plot again, but replacing plot with the command oplot. IDL will not rescale your plot, so make sure to plot the data with the largest range first.

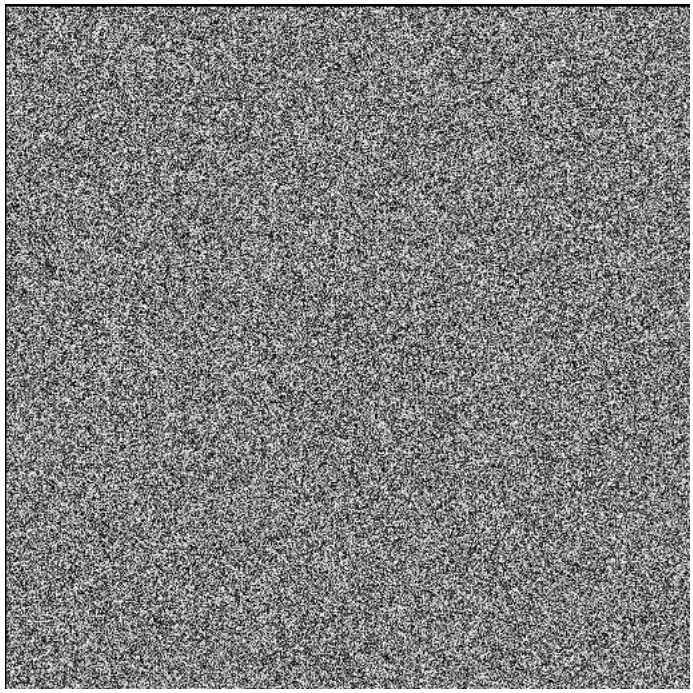
```
plot, a, psym=1
oplot, b, psym=2
```

2.4.3 TV and TVSCL

Now, plotting is very useful for one-dimensional vectors, but what of 2d arrays? For this, we can use the tv command to display the data. Let's generate a random 512x512 array, and use IDL to display the resulting data. Notice that we artificially scale the data within the TV command to allow IDL to show more of the range within the plot. Note also that the TVSCL command should do this automatically, but this command is for some reason not included within GDL.

```
m = randomn(1, 512, 512)
tv, m*1000
```

This seems terribly useless at the moment, given that we're just displaying random noise. However, this will prove much more useful in the future.



2.5 Writing it down

At this point, we've reviewed many useful commands that can be entered within the command line. However, to allow tasks that are more automated, we need to begin creating IDL script programs that can be run. One can write down a series of commands, that the program will execute in sequence. One point to note, however, is that unless declared in the program statement, the variables created and used within the program will not survive in memory **after** having run the program.

2.5.1 Program template

A program in IDL is characterized by calling it in the command line via,

```
programname, variable1, variable2
```

It will commence running the code, calling inputs and outputs as defined in the list of variables following the program name.

The most basic IDL program template would be something similar.

```
PRO programname, variable1, variable2

print, 'HELLO WORLD!'

end
```

2.5.2 Function template

A function differs from the program form in IDL in the way that it is called. To invoke a function, one would type,

```
a = function(input1, input2)
```

This is accomplished via a `return` statement within the function code. A basic template would be given by,

```
FUNCTION functionname, input1, input2

print, "This is where I would normally do something"

return, result

end
```

2.5.3 The importance of comments

When writing a program, one is often tempted to just write out code in a steady stream of inspriation. However, this can be quite dangerous, as the author has experienced firsthand. Personally. Regretably.

The note of point here is that comments can be inserted as inert blocks of text into your code, to remind you of exactly what you were thinking upon writing said code. This is accomplished via the ; character. Consider the following program.

```
pro test, input, output

;First, I'll print the input
print, input

;Then, I'll calculate the mean value of the input array, and output it
output = mean(input)

end
```

Sometimes seemingly redundant, coming back to look at code that one has written months before can be quite a taxing task without comments. Ultimately, the moral of this story is to take the extra bit of time to nicely comment your code. You (and anyone else using your code) will appreciate it.

2.6 Program control statements

We've learned so far how to allow IDL to store your variables, manipulate them, and how to encode commands into a script-like file. Ultimately, we wish the computer to do as much of workload as possible. The following program control statements allow the computer to behave in a certain way, depending on commands given by the program author.

2.6.1 IF statement

Think of the IF statement as a logical gate. You provide as input a boolean statement, which if true will allow the code within the IF statement to be run. If false, this code will be skipped over entirely. A simple example would be,

```
a = 5

if a lt 10 then begin
     print, a
endif
```

Notice the statement syntaxx involving `thenbegin`, and `endif`. In this case, the statement was true, and thus the value of a would be printed. The IF statement can also be appended with the addition of an ELSE statement. If the first statement is not true, then the else statement code will run. If the first statement is true, then the else statement code will not run. Consider the example,

```
a = 7

if a gt 10 then begin
     print, 'a is greater than 10'
endif else begin
     print, 'a is less than 10'
endelse
```

2.6.2 FOR loop

A FOR loop is useful when a chunk of code needs to be repeated under certain conditions. Usually some counter index is used, and advanced through a set of values. Consider the following example. Our counter index, `i`, is iteratively set through a series of values. The chunk of code within the FOR loop is then run, each time using the particular value of i as directed. Let's use the power of FOR loops to create a sine wave, encoded in an array. Notice the specific use of syntax.

```
f = fltarr(100)
for i=0, 99 do begin

     f[i] = sin(i)

endfor

plot, f
```

2.6.3 WHILE loop

Now, suppose that we wished to run a particular chunk of code, but we weren't sure of a particular index that would be useful. Suppose we just

wished to run our chunk of code as long as a particular condition was met. Suprise, surpise, a WHILE loop is the perfect solution here. Let's consider a quick example. In each iteration of the while loop, let's generate a random number. We'll add that random number to our current number, and create an array to document this progress. We can continue this process until the absolutely value of this number passes 10.

```
num=0
f = 0
while abs(num) lt 10 do begin

     num = num + randomn(systemtime_seed)
     f = [f,num]

endwhile
```

Try playing around with this code to modify it a bit. The only way to truly learn programming is to take code, and to tear it apart.

2.7 Other tricks

Here's a list of helpful commands that can make programming in IDL a little bit easier.

2.7.1 Save files

More of a feature than a trick, save files allow one to save data in IDL's proprietary format. You can even save multiple variables into the same save file. Consider the syntax for creating and then restoring a save file.

```
save, a, b, c, d, filename='file.sav'

restore, file.sav
```

2.7.2 Debug mode

Debug mode is useful when in the process of writing a program. Ordinarily, if your IDL program isn't working correctly in terms of the form of the

output, but is still managing to execute altogether, you have no access to the variables created and destroyed in the process of the calculation. By including a simple keyword into your program, one can halt the program before the end (or anywhere else you wish), and keep variables intact to study exactly where the problem is occuring.

```
pro test, input, output, debug=debug

... program goes here ...

if keyword_set(debug) then stop
end
```

One would then call this program via,

```
test, input, output, /debug
```

Chapter 3

Getting to know Linux

3.1 The terminal

First, we need to find the terminal itself. If using Macintosh OS, locate the terminal through opening spotlight (the magnifying glass in the top right corner of the screen), and type in terminal. After opening terminal, by right clicking the dock icon you will have the option of retaining it in your dock.

If using Microsoft windows, follow the instructions provided by the physics IT staff after having installed terminal software. For the curious user with an ssh account on the physics server, try the program PUTTY to get access to a linux machine over an ssh tunnel.

Once you have the terminal open in front of you, there it will sit, awaiting your commands...

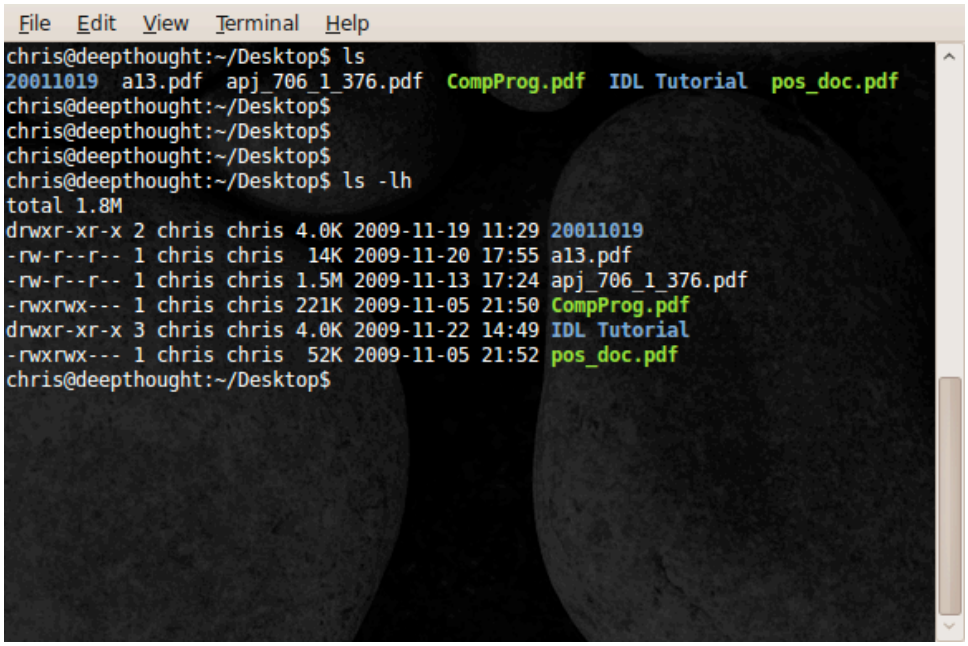
3.2 Terminal commands

3.2.1 Listing a directory

Probably the first command that will be of use is the ls, or list command. With your focus in the terminal window, type the command ls, followed by the return key to execute the command. It should return a list of files and directories in your current folder. For more details, try the command ls - lh. This will execute the same list command, but with the added instructions,

noted by the - symbol, to do a full list, l, and to make any values human readable, h.

Both examples are shown in the figure below. Try it out yourself for practice!

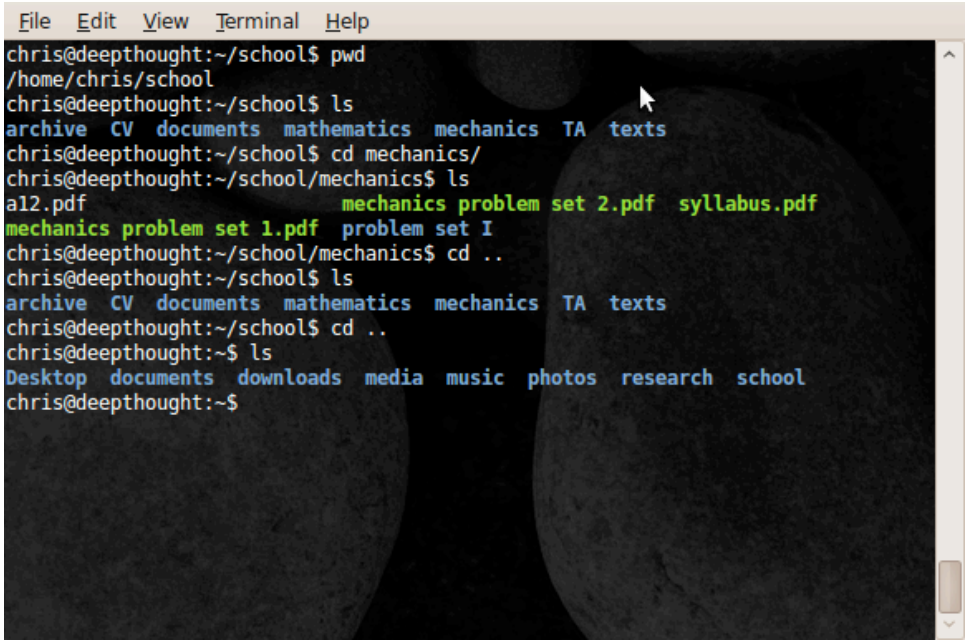


3.2.2 Changing directory

For this section, follow along with the commands shown in the figure. We begin by typing into the terminal, `pwd`. This will display the current directory that you're in. From there, we can use the ls command to see what lies in this brave new directory. I see the folder, mechanics, and decide that I wish to move into that directory. I then issue the `cd` command for change directory, followed by the name of the folder or path that I wish to move into.

Once I'm in this directory, for some reason I decide I no longer wish to be here. So I decide to move back into the folder above the mechanics folder. So I issue the command, `cd ..`, `cd` followed by two periods. This will move back into the next level folder. For example, if I type this command while in the folder, `/home/chris/school/mechanics/`, it will move me to the

directory, `/home/chris/school/`. What do you think would happen if you nested two of these together, `cd ../..`? Try moving around the directories in your `/home/yourname/` folder, and familiarize yourself with visualizing the directory structure.



3.2.3 Creating a directory

Now that you can move around through directories, what if you want to create a new one? To accomplish this, issue the command `mkdir nameofdirectory`.

3.2.4 Copying, moving, and renaming files

Things should start to move more smoothly now that you're beginning to become familiar with Linux and UNIX commands. Let's begin with how to copy a file. The command is fairly simple,

```
cp originalfile newfilelocation
```

Keep in mind that one can use file directories along with the copy command, that is to say that the file you're copying doesn't need to be in the same directory! Let's say I wish to copy a file from my home directory to my Desktop directory, /textbfwhile I'm in my documents folder! Sounds complicated, but here we go,

```
cp /home/chris/file.txt /home/chris/Desktop/
```

When making a copy of this file, no one said I had to keep the same name for it. When specifying the directory to move the file to, you can also specify the name of the copied file. For instance,

```
cp /home/chris/file.txt /home/chris/Desktop/someotherfilename.txt
```

Now, a helpful linux trick that may come in handy. Recall that the two dots, .. referred to the directory above the one currently in. A single dot, ., refers to the current directory. So, just to throw a lot of what we've learned together all at once, consider the following scenario. I'm located in my documents folder, and I wish to copy a file from my downloads folder into the documents folder. Rather than a complicated mess of changing directories and the like, consider the following command. Try to visualize the directory structure as you read through the command.

```
cp ../downloads/file.txt .
```

Pretty sweet, eh? These linux commands seem quite complicated at first glance, but with practice they become a powerful and time saving tool. Up until now we've considering copying a file, leaving the original file behind. The syntax for moving a file is quite similar to the copy command. Just swap out the cp for a mv, and you're good to go. As an example, say I wish to move a file from my Desktop to my documents folder.

```
mv /home/chris/Desktop/file.txt /home/chris/documents/
```

All the same tricks we learned for the copy command will work here as well. But, there is another trick up our sleeves. Say we wish to simply rename a file. We could think of that as just moving the original file to the same directory, but specifying a new name. It's just that easy,

```
mv file.txt newfilename.txt
```

3.2.5 Deleting files and directories

To everything there is a time and a purpose. A text file you wrote long ago may not have a place in your current research. As such, there should probably be some sort of garbage maintenance in a Linux system. To remove a file, simply type,

```
rm file.txt
```

The question of the afterlife for a file in a Linux system is quite complicated; best not to ask. Now, this command should work with any file, but what about a directory. Here, things get a bit trickier. The main di?erence is that you must ensure the directory is empty of any files and or subdirectories. Then, issue the command,

```
rmdir directory
```

Your training is now complete. The next section deals with advanced Linux tricks that may be of use in your journey.

3.3 Other terminal tricks

3.3.1 Home directory shortcut

When invoking a directory structure in a command, to quickly reference the home directory use the character $\sim$. For instance, it can shorten things as follows.

```
cp /home/chris/documents/file .
```

This now becomes the more wieldy,

```
cp ~/documents/file .
```

This may not seem the most useful tool, but after typing your home directory a few hundred times you'll wish your pinky finger wandered up to the tilde key.

3.3.2 Tab autocomplete

The terminal isn't dumb. It has a knowledge of the files and directories you're trying to access. And it wants to help you. Say you were trying to access your documents folder by typing the following.

```
/home/chris/docu
```

...But at this point you're feeling quite impatient with how long it actually takes to type the word document. Problem? Nope. By hitting the TAB key, the terminal will attempt to finish your statement for you, and will most likely type 'ments' for you. If nothing happens, multiple files or documents could exist with that same prefix. Two TABs will display a list of all matching files and documents.

3.3.3 Wildcard tool

How can you find what you're looking for, if you don't know what its name is? The wildcard tool is your answer. Say you have a directory full of files. Named file0001, file0002, file0003. Instead of typing rm file0001 and all of the rest, the wildcard symbol will enable the terminal to match anything matching the input you've provided. For instance, to delete the files as shown, the command issued would be.

```
rm file*
```

But... suppose you want to delete all of the image files, but not the text files in a directory. And just to make things more difficult, the names overlap, but all begin with the prefix file. No problem with the wildcard, as it can be inserted into the middle of a statement, even multiple times!

```
rm file*.jpg
```

3.3.4 Manual files

If ever in doubt about a command,

```
man command
```