IDL for Astronomers

Austin McDowell
Joe Zalesky
Will Schultz
Acknowledgements: We would like to acknowledge Baylee Bordwell and Pauline Arriaga for their work in assembling much of the base topics covered in this class, and for creating many of the resources which influenced the production of this textbook.

Written Summer 2015
# Version Control & Git

## Bash

### 1 Bash

#### 1.1 Bash on a Conceptual Level

#### 1.2 Why Use Bash?

#### 1.3 Getting to Know Bash’s Syntax

#### 1.4 The Basics

- **1.4.1** Documentation ........................................... 9
- **1.4.2** Viewing Files and Paths ................................... 9
- **1.4.3** Manipulating Directories and Files ...................... 10
- **1.4.4** Finding Things Quickly .................................... 12
- **1.4.5** Flags and Optional Input ................................ 12
- **1.4.6** Summary of the Basics .................................... 13

#### 1.5 More Advanced Bash Commands

- **1.5.1** SSH/SCP .................................................. 14
- **1.5.2** Screen .................................................... 14
- **1.5.3** Tarballs .................................................. 15
- **1.5.4** Top/Kill .................................................. 15
- **1.5.5** Passwd .................................................. 15
- **1.5.6** Alias .................................................... 16

#### 1.6 Summary

### 2 Version Control & Git

#### 2.1 What is Version Control? Why do I care?

---

**Contents**

1. Bash ................................................................. 7
1.1 Bash on a Conceptual Level ................................... 7
1.2 Why Use Bash? .................................................. 7
1.3 Getting to Know Bash’s Syntax ................................. 7
1.4 The Basics ...................................................... 9
1.4.1 Documentation ................................................. 9
1.4.2 Viewing Files and Paths ..................................... 9
1.4.3 Manipulating Directories and Files ....................... 10
1.4.4 Finding Things Quickly ..................................... 12
1.4.5 Flags and Optional Input .................................. 12
1.4.6 Summary of the Basics ..................................... 13
1.5 More Advanced Bash Commands .............................. 13
1.5.1 SSH/SCP ...................................................... 14
1.5.2 Screen ......................................................... 14
1.5.3 Tarballs ....................................................... 15
1.5.4 Top/Kill ....................................................... 15
1.5.5 Passwd ....................................................... 15
1.5.6 Alias ......................................................... 16
1.6 Summary ......................................................... 16
2. Version Control & Git .............................................. 17
2.1 What is Version Control? Why do I care? .................. 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Data Analysis and Plotting</td>
<td>55</td>
</tr>
<tr>
<td>6.1 Reading and Writing Data</td>
<td></td>
</tr>
<tr>
<td>6.1.1 Bash and IDL</td>
<td>55</td>
</tr>
<tr>
<td>6.1.2 Constant System Variables, Save, and Restore</td>
<td>56</td>
</tr>
<tr>
<td>6.1.3 Reading Data with Readcol</td>
<td>58</td>
</tr>
<tr>
<td>6.1.4 Reading and Writing Data</td>
<td>60</td>
</tr>
<tr>
<td>6.2 Plotting in IDL</td>
<td>61</td>
</tr>
<tr>
<td>6.2.1 A Simple 2-D Plot</td>
<td>61</td>
</tr>
<tr>
<td>6.2.2 A More Complicated Plot</td>
<td>64</td>
</tr>
<tr>
<td>6.3 Summary</td>
<td>69</td>
</tr>
<tr>
<td>7 Strings and FITS Files</td>
<td>71</td>
</tr>
<tr>
<td>7.1 Advanced String Manipulation</td>
<td>71</td>
</tr>
<tr>
<td>7.2 FITS Files</td>
<td>73</td>
</tr>
<tr>
<td>7.2.1 Reading in FITS Files</td>
<td>74</td>
</tr>
<tr>
<td>7.2.2 Playing with the Image and the Header</td>
<td>75</td>
</tr>
<tr>
<td>7.2.3 Displaying Images</td>
<td>76</td>
</tr>
<tr>
<td>7.2.4 Writing to FITS Files</td>
<td>78</td>
</tr>
<tr>
<td>8 Data Structures and MCMC</td>
<td>79</td>
</tr>
<tr>
<td>8.1 Data Structures</td>
<td>79</td>
</tr>
<tr>
<td>8.1.1 Creating Data Structures</td>
<td>79</td>
</tr>
<tr>
<td>8.1.2 Playing with Structures</td>
<td>80</td>
</tr>
<tr>
<td>8.2 Markov Chain Monte Carlo: MCMC</td>
<td>82</td>
</tr>
<tr>
<td>8.2.1 Humble Beginnings: Probability Density Functions</td>
<td>82</td>
</tr>
<tr>
<td>8.2.2 The Random Walk Problem &amp; Markov Chains</td>
<td>83</td>
</tr>
<tr>
<td>8.2.3 MCMC Summary</td>
<td>84</td>
</tr>
<tr>
<td>9 Error Analysis and Fitting</td>
<td>85</td>
</tr>
<tr>
<td>10 Glossary</td>
<td>87</td>
</tr>
<tr>
<td>10.1 Chapter 1: Bash</td>
<td>87</td>
</tr>
<tr>
<td>10.2 Chapter 2: Version Control and Git</td>
<td>87</td>
</tr>
<tr>
<td>10.3 Chapter 3: Basic IDL</td>
<td>87</td>
</tr>
<tr>
<td>10.4 Chapter 4: Control Statements</td>
<td>87</td>
</tr>
<tr>
<td>10.5 Chapter 5: Arrays</td>
<td>87</td>
</tr>
<tr>
<td>10.6 Chapter 6: Data Analysis and Plots</td>
<td>87</td>
</tr>
<tr>
<td>10.7 Chapter 7: Strings and FITS Files</td>
<td>87</td>
</tr>
<tr>
<td>10.8 Chapter 8: Data Structures and MCMC</td>
<td>87</td>
</tr>
</tbody>
</table>
10.9 Chapter 9: Error Analysis and Fitting
1. Bash

1.1 Bash on a Conceptual Level

To understand how and why computers are used ubiquitously throughout astrophysics we begin by understanding how computers work on a more basic level. Almost all computer operating systems (other than Windows) are really based off an older operating system called UNIX which was initially developed at Bell Laboratories in the 1970’s. In order to make computing efficient, many different types of user interfaces have been developed for the human user to communicate with the computer’s operating system. One interface developed in 1989 was Bash which allows the user to interact with UNIX via different text commands. The first part of this course will acquaint you with various Bash commands so that they can be used to easily and quickly control the operating system.

1.2 Why Use Bash?

At this point you may be wondering; why do we bother to teach such an old way of interacting with computers? To answer that let’s take a common situation you will often encounter in an upper division lab course or a research experience. Say you have a large number of data files (~1000) and you want to edit line 57 of each file and then place the edited files in a different part of the computer while retaining the unedited original copy. Using a more user-friendly interface this could take ages as you open, edit, and move each file. Using Bash this can be done by in less than a few seconds. In addition lets say you still want to edit by hand; what if you make a mistake on file 485? How would you know about your error until it caused a problem later on? Using Bash removes the possibility for this type of error called user-error (or as we like to say; a problem between the chair and the keyboard).

1.3 Getting to Know Bash’s Syntax

Now that I’ve hopefully convinced you of the usefulness of interacting with a computer this way let’s try to understand how Bash works (or as we say, let’s get to learning about Bash’s syntax). The first thing we need to do is to open up a Terminal window which allows users to input Bash commands.
For Ubuntu users this can be accomplished by hitting Ctrl-Alt-T on your keyboard and Mac users can simply do Cmd-Spacebar followed by typing ”terminal” and pressing enter. Below is an example of what a new Terminal window looks like on my personal Ubuntu setup.

Ubuntu is a UNIX-like operating system like Mac OSX in that there is a Graphical User Interface (GUI) overlaid on top of UNIX to make certain actions more user-friendly. Ubuntu is currently the operating system installed on the UGAstro computers in the undergraduate lab so becoming familiar with this OS will be beneficial to you.

Figure 1.1: A typical terminal, with two common Bash commands (ls and cd) sent through.

Let’s try to understand what’s happening in the above figure. Looking at the first line you can see the prompt which reads, 
joseph@joseph:~$ ls

What’s going on here? The first line joseph@joseph:~ corresponds to my computer’s name followed by a colon which is then followed by the current directory. You can think of a directory much like a ”file” in Windows or Mac OSX. The current directory that you are ”in” is often called the working directory. In Bash ~ is shorthand for the ”home” directory which is where all new Bash terminals default to. At the end of the line is a $ which just acts as a separator between the terminal’s prompt and user inputs. Following this you can see that I’ve input the Bash command ls which is short for ”list”. This command lists all the files and directories that are within the current working directory (which in our case is the home directory). You can see in the output that I have directories for music, pictures, videos, etc. within my home directory. In English you can read this line as saying: Please list all the files and directories under my home directory.
Let's jump to the next line which reads,

```
joseph@joseph:~$ cd Documents
```

This looks very similar to our previous line except now I’ve input `cd Documents` as the command instead of `ls`. `cd` stands for ”change directory” and in this instance we’ve called this command with the input ”Documents”. What this line is saying in English is: Please change my working directory from the home directory to Documents. You can see in the prompt that the working directory has changed from `~` to `~/Documents`. After this I ask Bash to list the files and directories within Documents and you can see directories I have for school, research, etc.

If you haven’t understood everything in this section don’t be discouraged; the way everyone learns Bash is by using it more than reading from a textbook and you’ll have plenty of practice in the coming weeks.

### 1.4 The Basics

#### 1.4.1 Documentation

Now that you’ve at least seen the terminal and how commands are input let’s get to discussing what Bash can really do. You’ve seen in the above section that we can list everything within our working directory using `ls` and we can change directories using `cd` but as you can imagine Bash is much more useful than that. The first thing to discuss is the command `man` which is short for ”manual”. This command allows you to view the manual on any Bash command that you desire. For instance typing

```
man ls
```

will give you the manual for the `ls` command. This should be your first stop if you’re unsure about syntax of a particular command or if you’re wondering if a command can do something special that you may require.

#### 1.4.2 Viewing Files and Paths

Being able to view files in our working directory is great but let’s say we wanted to look at what files are within a different directory. Using our example from above lets say I wanted to see if I had a particular album downloaded in my `Music` directory but I’m currently in the `Documents` directory. We can easily look at Music with either of the commands;

```
ls ~/Music
ls ../Music
```

What’s the difference between these two? `ls ~/Music` will give us the list of items within Music no matter what directory we’re currently in. Why is this? The `~` symbol is really a shorthand for the home directory (which on my system is actually `/home/joseph/`) so entering

```
ls ~/Music
```

is really the same as entering,

```
ls /home/joseph/Music
```
which as you can see has nothing to do with the current working directory. So what is the second command really saying then? The symbol .. stands for “parent directory” which means the directory above your current working directory. For a concrete example take the path /home/joseph/Music/. Here the parent directory of Music is /home/joseph/. So by typing `ls ../Music` while we’re in the Documents directory is saying in English; go up a directory, find Music, and list its contents.

Another less efficient way to solve this would be to enter,

```
cd ../Music
ls
```

The first command says; go up a directory and to Music followed by the list command. As you can see much of the syntax of our `ls ../Music` command has carried over to the `cd` command. Bash has the nice property that most of its syntax is very intuitive and each command does not have its own unique syntax.

### 1.4.3 Manipulating Directories and Files

Now that we can move around our directory tree as it is commonly referred to, lets learn how to start editing our directories. The first command we’ll introduce is `mkdir` or ”make directory”. The syntax of the standard input is,

```
mkdir <path of desired directory>
```

which allows you to make a directory anywhere in the directory tree. For instance lets say I downloaded some music and wanted to add the album to my Music directory. I’d input

```
mkdir ~/Music/New_Album
```

after which I could `cd` into and input `ls` but of course there’s nothing in this directory as we just made it. Notice that my directory New_Album does not use a space. This is deliberate as Bash handles spaces in a unique way. As you most likely noticed commands and their inputs are separated by spaces and so if I did `mkdir New Album` then I would have 2 new directories called New and Album. If I really wanted to make New_Album have a space then I would have to input

```
mkdir ~/Music/New\ Album
```

Where the \ character allows for a space as the input.

So now we’ve made a directory for our new album and lets suppose we have multiple mp3 files stored somewhere on our Desktop that we’ve downloaded and we’d like to move them into our New_Album directory. Assuming we’re in our Desktop directory this is simple enough with the command `mv`;

```
mv file1.mp3 ~/Music/New_Album
mv file2.mp3 ~/Music/New_Album
mv file3.mp3 ~/Music/New_Album
```

etc. You can imagine that inputting this would get tedious for large albums so is there a better way to do this? With Bash, yes there is! All we have to do is use a wildcard with the syntax;

```
mv *.mp3 ~/Music/New_Album
```
What’s happening here? The * is our wildcard which stands for any *and* multiple characters. In English this command reads: move anything with .mp3 at the end to New_Album. This prevents us from having to issue the same command multiple times and is a nice shortcut. However suppose we only wanted a few files from our desktop to go into New_Album and not all of them are mp3’s. How do we solve this issue? Hopefully the files have some well-defined naming structure like songname-albumname.fileformat. If this is true we can just input,

```
mv *-albumname.* ~/Music/New_Album
```

Notice that we had to use two wildcards one to remove the necessity to input each song name and the other to take care of the file format. Hopefully this example illustrates the reason for not using unnecessary capitalization, simple but clear file names, and a self-consistent naming system.

In addition to being the “move” command, mv also serves as the renaming command in Bash. So if we wanted to change the name of one of our songs we could simply;

```
_mv songname.mp3 new_songname.mp3
```

which reads; rename songname.mp3 to new_songname.mp3. You may be wondering how mv distinguishes between moving and renaming but don’t let the details concern you for now. This will be explored in the lab portion of the course and, with practice, will become obvious to you. Continuing with our music example let’s suppose we listened to the album and decided we didn’t like it and so want to remove it from our computer. There is a simple command to do this; rm which is the most useful but dangerous Bash command. Inputting

```
rm ~/Music/New_Album/*
```

which reads: remove everything within New_Album is not like using the Recycle Bin in Windows. If you use rm on any file or directory then those files are gone forever with no hope of recovery. I’ve personally removed entire directories before that I did not mean to and would like you to avoid the same mistake. So now that we’ve removed all the files in New_Album we need to remove the directory with rmdir where we simply input,

```
rmdir ~/Music/New_Album
```

You may be wondering why we didn’t just use rmdir in the first place and that is because rmdir can only remove directories that are empty. This acts as a preventative measure to stop you making the mistake of removing directories that are important to either you or the computer’s operating system. There is actually a way to remove all the files and the directory in one line using rm but I leave that to you as an exercise.

Next is cp which stands for ”copy”. The syntax of a standard input is,

```
cp <what you want to copy> <where you want to copy it to>
```

So for an example lets say we wanted to copy a few images to Desktop that we wanted to edit in something like GIMP or Photoshop. To copy these we’d input

```
_cp ~/Pictures/*.png ~/Desktop
```

where I’ve assumed they all have PNG formatting for simplicity. By now you should see that this command reads: copy every .png from Pictures to Desktop.
1.4.4 Finding Things Quickly

Now that you can move around the terminal easily along with being able to create and destroy files and directories you almost have complete mastery of the Bash basics. There are two more commands to help you find and list things beyond the `ls` command and they are `grep` and `more`.

`grep` is really an acronym for “Globally search a Regular Expression and Print”. Regular expressions are a bit beyond our reach here but feel free to google to your heart’s content. All you need to understand is that `grep` is the holy grail of search commands in Bash.

Let’s say for instance you wanted to find a file in a directory with over 1000 files of similar name (such as a directory full of data labeled by the date it was generated). You could do `ls` and try to find the name of the file you wanted but it would quickly become annoying to visually sift through everything you didn’t want. To make matters easier let’s say you’re looking for something that happened on 05_01 (May 1st). To reduce the number of files you’re looking at simply issue,

```
ls | grep 05_01
```

and Bash will output the names of the files with the string of characters 05_01 in your terminal.

How does this command work? The `|` symbol stands for `pipe` and simply means `use the output of the left command as the input of the right command`. So in our case the output of `ls` (Re: the list of files) is now the input to `grep 05_01`. In English this reads: list all the files in this directory then search that list for 05_01 and print those files containing 05_01. Pipes have endless utilities beyond the simple two-command example shown here and should be explored when working with Bash for a larger project.

The last command we have to detail here is `more` which, as its name suggests means to show ”more detail” of the input file.

Sticking with our dated-data example from above let’s say we found our file using `grep` and its name was 2015_05_01.txt and now imagine we’re interested in finding if the string hello exists anywhere in our file. We can just enter,

```
more 2015_05_01.txt | grep hello
```

and if the string hello exists within this file, `grep` will tell us. Again we have used a pipe and in English this reads: show the full detail of 2015_05_01.txt and then search for hello and print where it exists.

`grep` has a large number of helpful optional inputs than can be found using the `man` command; many of which operate using a special syntax detailed in the following section.

1.4.5 Flags and Optional Input

In the examples above we’ve shown multiple cases of using Bash’s optional inputs. For instance if you input just `ls` then you’ll get a list of everything in your working directory. However if you use an optional input and specify a path such as `ls ~/Documents` then you get a list of everything under that directory. You can imagine that Bash has much more powerful optional inputs than this simple example and here we explore a special kind of optional input called flags.

We’ll begin by showing a simple example followed by an explanation. Going back to our music example let’s assume we wanted to list all the songs in our Music directory. We could try `ls` but that only outputs a list of the albums, not the songs themselves. How do we access the songs? We use a flag! You can think of flags as a way to tell whatever command you’re using to do something special if you issue the flag. In our example we would input

```
ls -l
```
1.5 More Advanced Bash Commands

`ls -R ~/Music`

to get the list we desire. This reads in English: list everything under `Music` AND any of the subdirectories under `Music`. Assuming we’ve stored our music under the format of `~/Music/Album/<allsongs>` then this solves our issue. You may be wondering why the flag here is `-R` and the answer is that it stands for recursion which you can Google as its an extensive topic in itself. All flags begin with `-` and all flags for any command can be found in the command’s manual which is accessible via the `man` command.

1.5.6 Summary of the Basics

As requested by previous years we’ve added a section which offers a short summary of the commands we’ve detailed so far in this chapter (and two we didn’t but they are useful).

1. **pwd**: "Print Working Directory"; This command will print out the full path to your working directory.

2. **clear**: "Clear Current Terminal"; Removes all text on the current terminal window. Useful when output becomes overwhelming and unreadable.

3. **man**: "Manual"; Lists the manual page for the input command

4. **ls**: "List"; Lists all items in either the working directory or specified path

5. **cd**: "Change Directory"; Change your working directory to the input path

6. **cp**: "Copy"; Copy desired input from A to B

7. **mv**: "Move/Rename"; Moves first input to second input OR renames first input to second input

8. **rm**: "Remove"; **Powerful Command**! Completely removes whatever is input

9. **mkdir**: "Make Directory"; Make a directory at the specified path

10. **rmdir**: "Remove Directory"; Removes directory at the specified path. Only works if directory is empty.

11. **grep**: "Globally search a Regular Expression and Print"; A search function that allows for various types of input and output.

12. **more**: "More Detail"; will display any ASCII text within the file in the terminal window.

1.5 More Advanced Bash Commands

In this section we detail some of the more useful non-conventional bash commands that may be of help to you in either the lab courses or research applications in order of importance. Please note that Bash has many more commands than detailed in this text and if you would like a full list of possible commands please refer to the Bash Resources portion of the class website.
1.5.1 SSH/SCP

Having the ability to easily, quickly, and securely connect to remote computers is one of the most important parts of any computer network. UNIX has a nice system called SSH which stands for Secure-Shell and allows UNIX users to connect to any other UNIX computer provided a login system has been setup. In this class the main function of SSH will be to allow you to login to the computers in the undergraduate lab from home. In general the way one ssh’s into any network is;

```bash
ssh <username>@<network>
```

But for our purpose the way to ssh into the ugastro network is

```bash
ssh -X <username>@ugastro.berkeley.edu
```

where the -X flag stands for X-11 forwarding and allows for graphical interfaces to function over this type of connection which is usually text-based. Once you’ve logged in using this command you’ll have access to the ugastro computers just as if you were sitting at a computer in the lab (though of course now its over an internet connection so it will be slightly slower). Please note that Windows users attempting to use Bash must have extra software to enable X-11 forwarding. See the Remote Connection Guide on our website for a complete guide to SSH from a Windows machine.

Now lets say you wanted to transfer something from your home computer to ugastro or vice versa. To do this we need the command scp which stands for Secure-Copy and utilizes UNIX’s built-in SSH networking capabilities. A general scp command should be input as;

```bash
scp <copy this> <to here>
```

where you can see that this function operates much like the cp command except now it is to an exterior network. As a specific example lets say we wanted to copy some text document from our home computer to our home directory on ugastro. We can simply input

```bash
scp file.txt <username>@ugastro.berkeley.edu:/home/<username>/
```

Or if we wanted to copy some data from ugastro onto our local computer’s Document’s directory we could simply do

```bash
scp <username>@ugastro.berkeley.edu:/home/<username>/datafile.txt ~/Documents/
```

or to any path we desired.

However imagine that you’re running a simulation over SSH and it takes ~10 hours to complete. If the internet connection drops out at any point (thanks airbears!) then you’re simulation will fail and you’ll have to run it again. How do you get around this issue? That’s the main point of the next section.

1.5.2 Screen

One of the most common issue when working over SSH is the possibility of disconnections. The command screen makes this issue virtually disappear by allowing your processes to continue after the disconnection occurs. To begin a screen session just enter screen into the terminal and you’ll be prompted with a little information window. Hit return to exit out of this and you should be back in a normal Bash terminal. It may not appear so but you are actually now using a screen session.

Screen commands revolve around the Ctrl-a keyboard shortcut. Let’s say you’re working on some
project on a remote host in a screen session and the connection cuts out. Once you’ve ssh’d back into your remote host simply input Ctrl-a r to resume the session from where you left off. You can also open multiple screen session in one terminal window with Ctrl-a c to start a new session and Ctrl-a k to kill a session. With multiple sessions open you can use Ctrl-a Ctrl-a to switch between them and each session is assigned a number starting at 0 for whatever screen session it is. There are many more useful screen commands (such as changing the label on each, controlling multiple sessions with one command, etc.) but I’ll leave it to you to dig deeper via man and Google.

1.5.3 Tarballs

Tarballs is a jargon term for a compressed group of files akin to .zip files that any Windows user will be accustomed to working with. By compressing the information in each file into a smaller size, it becomes faster to send from one computer to another. You may see these when downloading large software packages for any UNIX OS or when working with an older researcher so we want you to be aware that they exist. All you’ll really need to know about tarballs is how to create and open them. Let’s say we wanted to send an album comprised of different .mp3 files to a friend via a tarball. To create the tarball we just enter,

```
tar -czf album.tar *.mp3
```

where tar tells Bash we want to work with a tarball. The 3 flags shown are often remember with the mnemonic Create Zee File!. Then you can send the created album.tar over email, facebook, etc. Now let’s say you’re on the other end of this and you receive an album from a friend via tarball and you want to open it and listen to some music. To open, aka extract, album.tar we just enter,

```
tar -xzf album.tar
```

and then all the files within album.tar will be placed within our working directory. The mnemonic for remembering the flags to open tarballs is, Extract Zee File!

1.5.4 Top/Kill

Another powerful tool Bash offers is the ability to quickly see what processes are running both in the foreground (Re: things you can see from the Desktop GUI environment) and the background (Re: subprocesses that cannot be easily seen without Bash). The command to show all processes by CPU usage is top.

From the top session you should see somewhere a column called PID which stands for Process Identification. You can manipulate this process using various Bash commands but the most widely used command is kill which abruptly ends any process. For instance say we notice that Google Chrome with PID 10873 is taking up way too much memory (which it often does and is why you should try Firefox) and we need to stop it. We can simply input kill 10873 to kill the process.

1.5.5 Passwd

If you’re reading this portion of the text it most likely means you need to reset your password on ugastro. To create a new password simply type passwd and follow the prompts. Note that a common feature of Bash is to display nothing when asking for a password so if you don’t see anything come up on the screen as you type don’t fret; that’s just Bash convention for hiding sensitive information from appearing on the screen.
1.5.6 **Alias**

In Bash **aliases** are commands which rename any given text string as something else. For instance let's say you often find yourself needing to `cd` far into your directory tree which is annoying to type out. For a specific example: on my computer I store all the resources for this Decal in `/home/joseph/Documents/fall2015/idl_decal/current_decal`. You can imagine that I don’t want to type `cd /home/joseph/Documents/fall2015/idl_decal/current_decal` every time I need to work on things. To solve this I made a permanent alias. A general alias can be defined in any Bash session but will be lost when the session closes. To make a permanent alias simply open your `.bashrc` file in your home directory with your favorite text editor (Emacs right?) and place whatever command you want to enter into this file and save it. The general syntax is,

```bash
alias new_name='<desired command>'
```

Or in my case for the decal I have,

```bash
alias decal='cd /home/joseph/Documents/fall2015/idl_decal/current_decal'
```

If you're wondering why we need to save it in this special file it is because every time we open a new Bash session, Bash searches this file for any predefined commands or procedures to run through before opening. This offers more control on the user's end while being able to keep commands all in one place.

1.6 **Summary**

Hopefully this has given you a general idea of a new and more efficient way of interacting with a computer when compared to a Graphical User Interface (GUI) system. The next chapter focuses on some special software called **git** which allows for easier control of directories and files over time.
2. Version Control & Git

2.1 What is Version Control? Why do I care?

Version Control is a unique type of software that is aimed at solving a specific problem: controlling files over time. Let’s say that you’re working on a piece of code called *derp.pro* (you’ll learn why I call it .pro in the Basic IDL chapter) and everything seems to be working well but then you have a new idea for a better way for the code to run. You then edit *derp.pro* to the point that the code has a completely different structure, it refuses to work, and you have no idea how to fix it. How do you go back to the previous way the code ran? You could try making multiple backups of *derp.pro* with cp but you’ll soon find how tedious that is. Better yet you’d have to come up for a naming convention for each file. Version Control software takes care of this for us all in one package and that’s just if you use version control for personal use.

You can imagine that if you took the previous problem and then added that more than one person had to work on the same code then without version control then it would very difficult to document and keep track of everyone’s code as they added in various components. Version Control is used by a majority of software companies for keeping track of their development team’s progress while also allowing for new ideas to be tested without threatening the integrity of the software that they know works and is very useful when working with any type of collaboration (Re: Final Project groups, assignments in the upper-div lab courses, and in research positions with an international community such as astronomy).

If version control all sounds too good to be true then I’ll say now that the downside to all of this is that Version Control, and more specifically the type of version control that we’ll be using called *git*, has a somewhat steep learning curve when you start out. The biggest challenge is visualizing how the computer keeps track of information without having a lot of graphics on the screen for you to reference. We’ve placed a lot of resources on the website that will hopefully ease your transition into learning *git* and hopefully this chapter will provide a good overview of how to think about *git* and the basic commands associated with it.
2.2 Basic Git

2.2.1 Getting Started

The first step to using git is setting up the directory where you want all of your git action to take place. This is done using two familiar Bash commands from the previous chapter, mkdir and cd. After creating and entering your new directory, you’ll want to run your first git command, git init. This command initializes (hence the ’init’) a new local repository in your directory. A repository, or ’repo’, is where the files you wish to keep updated with git are stored. The repository is ’local’ because it exists on your computer.

Now that we have initiated a repository (sounds fancy, huh?), we can do some simple configuration. If you’re working in a group, it’s good to keep track of who is making changes. In order to set up a username for yourself, type,

```
git config --global user.name "<your name>"
```

Since you’ll eventually be working with online repositories, it is helpful to associate an email address with your git actions. The command is similar to the one for your username,

```
git config --global user.email <your email>
```

Doing so allows other git users to contact you if they have questions about your code or if they want to invite you to take a look at their code. The final configuration is mainly one to improve the aesthetics of git. However, it arguably makes git a little easier to understand. By typing,

```
git config --global color.ui true
```

you can enable colors when running certain git commands.

Great, now that you’ve gotten your directory set up and everything is configured, you can start adding files to your repo.

2.2.2 Add and Commit

Before learning more git commands, it’s good to know that git comes with its very own help command. This help command is the git equivalent to the bash command man. Typing,

```
git help
```

shows the most commonly used git commands and provides a quick explanation of their usage. If you’re curious about a particular command, you can type,

```
git help <command>
```

This will open the git manual for the command you entered which provides a more detailed explanation as well as possible flags.

Now you need to create a file that you want to keep updated with git. Open your favorite text editor and make a file,

```
emacs -nw myfile.txt
```

Once you’ve finished editing your file be sure to save and exit.

Now that you’re back in the terminal you can type
2.2 Basic Git

`git status`

and you’ll be able to see what has changed since your last update. In this case, `git` will tell you that you have an untracked file called `myfile.txt`. This means that `git` isn’t keeping track of any changes that you make to this file. You should also notice that `myfile.txt` is colored red as a result of the color configuration you did in the previous section. In order tell `git` to start tracking `myfile.txt` you need to type,

`git add myfile.txt`

This adds your file `myfile.txt` to what’s called the **staging area**. Files in the staging area are ready to be **committed**. Committing a file saves the changes that you made before adding the file to the staging area. If you again type `git status`, you’ll see that `git` tells you that you have added a new file called `myfile.txt` and that your new file is ready to be committed. Additionally, the staged file is highlighted in green. Finally, you want to commit your file by typing,

`git commit myfile.txt -m 'Created myfile'`

The `-m` flag in the above command tells `git` that the following string is your commit message. When working on a project, it’s important to include these messages in your commits so your team knows what you’ve changed or so you can look back on commits and remind yourself of the recent changes. Suppose you’ve added multiple files to the staging area and you don’t want to have to type `git commit <filename>` for each file. You can instead simply type,

`git commit -m 'Commit message'`

which will commit all of the files in the staging area. The only downside to this is that all of the files will have the same commit message.

**Undoing Adds and Commits**

However say you didn’t really mean to add `myfile.txt` to the staging area. How do we remove it from there? There are multiple ways but,

`git reset myfile.txt`

is the safest and easiest option and the one most often used for this type of issue. However what if you’ve also committed the file and wanted to go back to the state of your previous commit. Simply enter,

`git reset --hard HEAD~1`

and all your files will be brought back to the state of the previous commit. If you’re wondering about the `HEAD~1` portion of this command; suffice to say that `HEAD` is a special word used by `git` to point to the most recent changes on the timeline (Re: the `HEAD` of the timeline) and `HEAD~1` will go back to the commit right before the latest one. There are many more ways to alter your timeline and the state that it is in but for now these commands should be enough to get you started working with your local repository.
2.2.3 Remote Repositories: Push and Pull

**Push**

Now let’s say you’ve gotten your local repository all set up in a state you are okay with and want to share your work with others. The easiest way to share git repositories is to use an online storage system such as Bitbucket or GitHub. Personally I would say Bitbucket is your best choice for most situations as it allows for free private repositories and GitHub, along with many others, will force you to pay for such a service. From here I will assume you’ve chosen to work with Bitbucket but the following steps should be more or less identical no matter what hosting site you use. First you’ll need to make an account on Bitbucket. We have a very explicit tutorial on how to do this on the website if you ever need to refer back to it. Once you have that setup you should see a ”Create” button at the top of the page which lets you link a new repository on your account to eventually be shared with others. You can chose to make the repository private or not and if you do you will have to manually give all members working on the project access to view and edit changes in the repository.

Once the online repository is created you will want to fill it with whatever you’ve created. To do so you will need to copy the url that is associated with the online repository. On Bitbucket you can find this link as shown in the first figure below or, if you’re repository is currently empty, it can be found in the instructional portion of the page.

![Figure 2.1: URL location highlighted in red.](image)

Once you have this url go back to your terminal and the directory where you ran `git init` and enter,

```bash
$ git remote add origin <your url>
```

What is happening in this command? `git remote` is telling git that you’re going to be working with a remote (Re: non-local) repository. `add origin <your url>` tells git that we’ll be defining the name `origin` to reference `<your url>`. Once this is done you can see this reference by running

```bash
$ git remote -v
```
where the -v flag stands for "verbose" meaning to list all the information about the connected remote repositories.

So now your local repository has a way to reference the online repository. However, the online repository still does not contain your newest commit. You still need to push the latest version of your timeline from your local repository to the online repository. To do this simply enter,

```
git push -u origin master
```

which tells git to push the latest commit on the branch master to origin which we just defined to be the url of our remote repository. Branches extend somewhat beyond what needs to be covered in this course so for now just know that we have only one branch of our repo and we call it master by convention. If you’re wondering what the -u flag does it basically sets the optional inputs origin master (which we often call arguments to differentiate them from flags) as the default input so that when you want to push changes in the future you can simply enter,

```
git push
```

instead of the entire command. Now everything should be synced and you can share your repository with whomever you are working with.

**Pull**

Imagine now that a friend has told you that they’ve made some edits to your code on the repo (see how fancy our speech is getting?) and you want to see what changes have been made. To do this you’ll need to pull the code from the online repo to your local repo. To do this we enter,

```
git pull origin master
```

where using git pull origin tells git where we’ll be pulling from and master is the name of the branch (which again you can think of as just the name of our local repo for now) we’ll be pulling it to. Should there be no problems between your local repo and the online repo, that is to say that you haven’t edited any files locally and committed changes that the online repo isn’t already tracking, then everything should be fine, git will fast-forward your local repo to the state of the online repo, and you’ll be able to see the changes made. However what if there is some inconsistency between files? How are those issues handled? For a great visual description of more advanced git commands including resolving errors, branches, and merges you can watch the third video in the series of Git Videos we have placed under IDL Misc. Resources on the website.

Note that the commands described in this section should be enough to get you started using git and more advanced commands will come into use as you run into issues with the software. Just like in Bash your skill will come with practice and general use more so than reading a textbook so don’t feel discouraged if you don’t understand everything right away.

### 2.3 Summary

Below we give you a laundry-list of common git commands most of which were covered in detail in the above sections.

1. **git init:**
   - Initializes a repository in your working directory.

2. **git config --global user.name <your name>:**
   - Sets <your name> as the name listed in the log files for every commit you do.
3. `git config --global user.email <your email>`:
   Same as above but also lists your email.
4. `git config --global color.ui true`:
   Enforces color highlighting making git much easier to work with.
5. `git help`:
   The git equivalent to Bash’s `man` command
6. `git log`:
   Shows you a log of all commits made on whatever branch you are working on.
7. `git status`:
   See what is in the staging area and what is being tracked by git.
8. `git add <files or directory>`
   Adds whatever files or directories are input as arguments.
9. `git reset <file or directory>`
   The opposite of `git add`; removes files or directories from the staging area.
10. `git commit <files or directories> -m "<message>"`:
    If nothing is input for `<files or directories>`; commits everything in the staging area to whatever branch you are currently on (usually master).
11. `git reset --hard HEAD~1`
    Purges the last commit completely from your branch.
12. `git remote add origin <remote repo address>`
    Defines `origin` as `<remote repo address>`
13. `git remote -v`
    Lists all currently defined remote repositories on your branch.
14. `git push -u origin master`
    Pushes files from the branch `master` to `origin`.
15. `git pull origin master`
    Pulls files from `origin` to the branch `master`.

We’ve also included a list of useful commands from the Git Videos that we did not cover in this text for ease of reference.

1. `git branch <name>`
   Creates a new branch called `<name>`. If nothing is input for `<name>` then it returns a list of all branches.
2. `git checkout <name>`
   Switches working branch from your current one to `<name>`
3. `git merge <name>`
   Merges the branch called `<name>` into your current working branch
4. `git branch -d <name>`
   Deletes the branch `<name>`
Now that you have some experience working with Bash and git it's time to jump into some real programming using a language called the Interactive Data Language or IDL. IDL was developed by Exelis Inc. back in 1977 and is used mainly within astronomy, medical imaging, and geographic satellite mapping communities. Why do astronomers use IDL? Today most astronomical science images are saved in the format of FITS files. We will cover FITS files in a later chapter but for now just know that IDL allows for easy manipulation and control of these files along with a large list of other capabilities that are useful for manipulating and displaying data.

### 3.1 The IDL Interpreter

How do we begin to interact with this language? IDL is what we call an interpreted language in contrast to a compiled language. The details of this difference is beyond this class (Re: google for more information if you’re interested) but just know that an interpreted language basically means that there is some software called an interpreter that we will interact with to use IDL. To start up the interpreter on the UGastro network simply enter Bash (Re: open Terminal) either through the computers in lab or over SSH and type,

```
idl
```

and a start-up script will begin and open the interpreter in the terminal window along with two plotting windows which for now you can close. Your terminal should look something like this,

```
IDL>
```

where IDL is now waiting for an input. To see things output on the screen we will use something called the print **procedure** which for now think of as the IDL equivalent of a Bash **command**. We can call this procedure by saying,

```
IDL> print, 'Hello World'
```

Hello World
Where IDL has now printed our argument *Hello World*. We can also use IDL as a glorified calculator by issuing something like,

IDL> a=5
IDL> b=10
IDL> print, a+b
15

What happened in these few commands? In the first two we defined the variables `a` and `b` to be 5 and 10 respectively. If we enter

IDL> print, a
5

you can see that we get the expected output. We then told IDL to print the output of the sum of these two variables and we get out 15 as expected. Simple enough so far right?

You may be wondering why I had to put apostrophes around *Hello World* in my first command but didn’t have to when I printed `a`. If you’ve been playing around with the interpreter than you may have also noticed that sometimes IDL seems to get basic math wrong. For instance if you enter:

IDL> print, 1/2
0

why didn’t IDL give you 0.5 as it should have? Both of these questions stem from the fact that IDL splits up different types of data into things called **datatypes**.

### 3.2 Datatypes

#### 3.2.1 Integers, Floats, Longs, and Doubles

You can imagine that since computers have both finite memory and processing capabilities that it would be efficient to store different types of information, aka data, into different categories. An obvious example is that the number 5 should obviously have different capabilities than the phrase *hello*: the most notable one being that if we try inputting `5 + 10` into the interpreter then we should get 15 however if we try `5 + hello` then the interpreter should give us some kind of error message. However when programming anything one should try to be as efficient as possible and as such IDL has many more datatypes to address different subtleties beyond the simple example above.

As an example, let’s resolve the mystery of why `print, 1/2` does not return `0.5`. To understand why we’ll need to talk about our first two datatypes (which are the most common numerical datatypes) called **integers** and **floats**.

Integers are, as you would expect from your knowledge of math, whole numbers that do not include decimals while floats (short for floating point) do contain decimals. To see this let’s enter into the interpreter,

IDL> i=42
IDL> f=42.0
IDL> print, i
42
IDL> print, f
42.0000
We’ve defined two variables; i as our integer and f as our float. You can see that i really does contain less information (Re: uses less computer memory) to store the number 42 than f and as such is more efficient. To see the datatype more explicitly we can enter,

IDL> help, i
I INT = 42
IDL> help, f
F FLOAT = 42.0000

You may now see why print, 1/2 only returns 0; both 1 and 2 are integer datatypes and, as convention, IDL can only return an integer datatype (Re: our result 0) as output when given only integer datatypes as input (Re: the numbers 1 and 2). So how do we fix the issue? We make one of the numbers a float;

IDL> print, 1/2.
0.500000

and our issue has been resolved (Note that 1/2. is not a typo but rather the short way to make 2 a float).

Since integers are the least memory intensive datatype they are also the ones prone to the most common errors. For instance let’s define a simple variable

IDL> test=32767

which is the same number as $2^{15} - 1$ (you’ll see why that’s important in a moment). Lets try a simple mathematical operation on test;

print, test+1
-32768

Wait what? How did we get a negative number from addition of two positive numbers? The key here is again that test is an integer datatype which itself is only a 16-bit datatype. What does that mean? In computing a bit is the most basic unit of information corresponding to a 1 or 0, yes or no, true and false, etc. statement. Since integers are 16-bit they can only hold $2^{16} = 65536$ unique numbers where we also need one bit for 0 and so we can only store up to $2^{16} - 1 = 65535$ unique, non-zero numbers thus leading to our limit of $2^{16} - 1 = 32767$. After reaching this cap IDL “rolls over” the variable into the negative region rather than changing its datatype to suit what you as a human may expect to happen. Its all about keeping things efficient. So how do we go beyond the number 32767? We use other datatypes and in this case, assuming we still want test to remain a whole numbered datatype we can cast it as a long datatype whose name is rather explicit; a long integer referring to the extra bits it uses to store larger numbers. To cast it as a long and see the difference we enter,

help, test
TEST INT = 32767
test=long(test)
help, test
TEST LONG = 32767
print, test+1
32768
and you can now see that we get the output of 32768 that we expect. The syntax test=long(test) may be odd to those of you who have never programmed before but is a common sight when looking at code. All this statement says in English is: take what has been previously defined as test (Re: the integer 32767) and make test now a long with the value 32767. Or another way to read it is: redefine test from whatever datatype it previously was and now make it a long. You’ll become accustomed to this way of thinking with more practice.

Just like there is a limit on the amount of information stored in integers there is also a limit on floats where floats are 32-bit. Longs also have a limit of being 64-bits but if you ever need a variable larger than 64-bit then you are most likely doing something wrong. There is also a 64-bit datatype corresponding to floats which are called doubles standing for doubling the precision of the float datatype. You should not need any numerical datatypes beyond these 4 for this class but if you’re interested and want to read further we have added some links to other resources which talk about the subject in-depth on the course website.

3.2.2 Strings

The first command we entered into the interpreter;

print, 'Hello World'

used a datatype called a string which commonly stores text-based information. The syntax for defining a string is to surround whatever you want to be a string in apostrophes or quotation marks. Strings are commonly used for testing various aspects of a program or otherwise giving the user some form of feedback through text.

One subtlety of IDL’s strings compared to other programming languages is that numbers can also be cast as strings and used for calculations;

\[
\begin{align*}
a &= 5 \\
b &= '10' \\
\text{print, } a+b \\ &= 15
\end{align*}
\]

but, if we try to add text or non-numerical symbols then they cannot be used for numerical calculations;

\[
\begin{align*}
a &= 5 \\
b &= 'hello' \\
\text{print, } a+b \\ &= \text{Type conversion error: Unable to convert given STRING to Integer.} \\
\text{% Detected at: $\text{MAIN}$} \\ &= 5
\end{align*}
\]

We can also add strings to each other. A common situation is that you want to output the result of a calculation stored in some variable, let’s say its called mass. To do this we just concatenate a few strings together with +,

\[
\text{IDL> print, 'MASS: ' + string(mass) + 'g'}
\]

```
MASS: 100g
```

There are many other ways to manipulate strings (search, find and replace, etc.) but we save that for a future chapter. For now just understand that text is stored as strings, numbers can also be stored as strings, and to concatenate strings together use +.
3.3 Writing Procedures & Functions

In the above section we often used the print and help procedures and I told you that you could think of them as the IDL equivalent to Bash commands. This is still true except that procedures are not the only "Bash command equivalents". IDL also supports things called functions which act a lot like procedures except for requiring something called a return statement which we will explain as we go on to explore writing things outside of the interpreter.

3.3.1 Scripts

So far everything we’ve done just requires us to run idl in Bash and we start interacting with IDL. However writing projects, or as we call them: scripts, programs, etc. that are 100’s of lines of code is much more easily done using a text editor and saving the file in some format that IDL can read. Doing this has a few distinct advantages: we can make changes and save as we go along, we don’t have to input all the commands every time we want to run the code in a script, and if we’re working with other people on a project they can easily access and edit our script using something like git.

Let’s try to write a script that prints Hello World just like we did in the interpreter in the previous section.

3.3.2 Differences in Syntax and Application

To begin we’ll start in Bash in some directory where we want our script to be saved. Maybe in something like ~/Documents/scripts. To start writing a script we’ll open a new file in Emacs,

```
emacs hello_world.pro
```

where the .pro is the format that IDL can read as code and that Emacs knows how to use proper syntax highlighting on for IDL. Syntax highlighting just allows Emacs to use colors to differentiate parts of the code to make it more readable by human eyes. Our finished script will look something like this,

```
pro hello_p

    print, 'Hello World'

end
```

The top line says that we want to write a procedure (as opposed to a function) called hello_p. Also note that every procedure and function must have an end statement so IDL know where one procedure/function ends and another begins. Cool so how do we actually get IDL to run this procedure? We first we open an IDL interpreter and use .com,

```
IDL> .com hello_world.pro
% Compiled module: HELLO_P.
IDL> hello_p
Hello World
```

What happened here? First we told IDL, using .com to look for a file called hello_world.pro which it did (Note we could have given it any path just like in bash but I’m assuming we’re in the same directory for this example). From there IDL told us that everything went smoothly and there were no immediate errors. If we had made a mistake like typing en instead of end then the
Chapter 3. Basic IDL

Interpreter would be complaining that there is some syntax error that must be fixed. Finally we called our hello_p procedure and it ran it where the only command was to print the string Hello World. We could have written this using a function,

```idl
function hello_f

  return, 'Hello World'

end
```

where you can see the most notable difference between the two is the need for the return statement. However there must be a reason for splitting procedures and functions into two different categories. The reason is that functions allow us to return output to something which in most case is a variable. Lets run our hello_f function like before and see what happens,

```
IDL> .com hello_world.pro
% Compiled module: HELLO_F.
IDL> hello_f
% Attempt to call undefined procedure: 'HELLO_F'.
% Execution halted at: $MAIN$
```

Something went wrong; we tried to call hello_f as if it were a procedure when really it is a function. To properly call our function we need to,

```
IDL> hello_f()
Hello World
```

and our output displays properly. Using our hello_f function we can also return our output to something like a variable,

```
IDL> a=hello_f()
IDL> print, a
Hello World
```

however we cannot do this with hello_p because if you look back at our code, hello_p simply says to print the string Hello World and not return it,

```
IDL> a=hello_p
% Variable is undefined: HELLO_P.
% Execution halted at: $MAIN$
```

It seems odd with this simple example to bother with the differences between procedures and functions; really all we wanted to do here was print a simple string. However such nuances are important to understand because, though it may not be very relevant for this simple case, being able to use both procedures and functions effectively will make coding a lot easier when we get to more complicated examples in data analysis.
### A More Relevant Example

Let's go beyond the standard Hello World example and try to do something more involved. Let's say you have a bunch of data points stored in some variable called `foo` and you want to find the average using IDL. To do this we can write a short function which will return the average of `foo`. You may be wondering how we can store multiple numbers into one variable and the way we do this is by using a new datatype called an **array** which for now you can think of as a list of numbers. To define `foo` we can just enter,

```idl
IDL> foo=[42,4,7,8,5,3,5,7,8,42]
```

and now we’ve stored our variable into the interpreter’s temporary memory (if we quit the interpreter now, `foo` will be lost). Let’s get to writing our script and let’s call it `average_foo.pro`.

When writing any type of code it is important to be explicit but terse in your naming conventions so that (a) you can understand what’s happening in your code months after you wrote it and (b) others can read your code and quickly figure out what you’re trying to accomplish.

To begin we’ll need to make sure that the user of this function can input what they want to average,

```idl
function average_foo, input_arr
end
```

where now no matter what the input is called, be it `foo`, `bar`, or any other name our script will see it as `input_arr`. To write our averager we’ll need to know some information about `input_arr` namely the total sum of all the values within `input_arr` and how many numbers there are in `input_arr`. One of the signs of well written code is generality; we don’t want to make this function work for *only* `foo` but rather any array we input into the function. As such we don’t want to just count by hand and **hard-code** in the numbers that happen to work for `foo`. How do we get this information?Luckily for us IDL has some pre-built functions which will return the information we desire,

```idl
function average_foo, input_arr
    total_sum = total(input_arr)
    total_elements = n_elements(input_arr)
    average = total_sum / total_elements
    return, average
end
```

Hopefully my naming conventions are clear in that first we used the **total** function to find the sum of `input_arr`, **n_elements** to find the number of **elements** within `input_arr`, and then used basic math to find the average and return it. Let’s see if this functions works using the interpreter,

```idl
IDL> .com average_foo.pro
% Compiled module: AVERAGE_FOO.
IDL> print, average_foo(foo)
12.1818
```
and you can see that our output is as we expect (check the math if you want to). We’re pretty much done but there are a few improvements that could be made to this code. For one thing we don’t need to use so many variables and we can add some **comments**.

```idl
function average_foo, input_arr

    ; Finds the average of input_arr and returns
    average = total(input_arr)/n_elements(input_arr)
    return, average

end
```

Comments are what the name suggests; comments written so that the human user can more easily read the code. In IDL the comment symbol is ; which means that anything after ; on the same line will not be read by the interpreter. Here a comment is somewhat redundant given the explicit names we’ve given our function and variables however they are invaluable for larger projects and when working with others. There are also some other good programming practices that are important in addition to having well written comments.

### 3.4 Good Programming Practices

Now that you understand the basic syntax of how to write a script lets detail some good coding practices that are applicable no matter what language you are using. To begin there are a few things unique about IDL’s interpreter that we need to list;

1. **IDL does not care about capitalization**
   - Declaring a variable as test is the same as TEST or TeSt.
2. **IDL does not care about indentation**
   - In the procedures/functions above notice how I indented the code so it looks like it’s within whatever procedure/function we were defining? IDL’s interpreter doesn’t notice any of that.
3. ; is IDL’s comment symbol
   - You should comment as much as needed to make operations explicit

However you should note that just because IDL’s interpreter doesn’t care about these things that does not mean that you can follow the same path. Computers read code using just the basic syntax that we’ve described in the previous section. However just because the interpreter can read the code does not mean that you or someone else is able to read the code.

What do I mean by this? Let’s look at an example of “well written” code which I’ve provided below. **Do not bother trying to understand what’s happening in the code** as it contains a variety of things we haven’t covered yet. Just look at the **readability** of the code and see how well organized it is.
As you can see there are a variety of comments detailing what is happening and why the code is written the way it is. Also indentation is widely used to separate different parts of the code and variable names are explicit. Now lets remove all of the things that are for human eyes and just get the bare-bones that the interpreter can work with,
function aprint,thing,nocompress=nocompress,_extra=_extra
if size(thing,/type) eq 7 then begin
  delimiter='", "
  delim2 = ',
end if size(thing,/type) eq 1 then begin
  return, aprint(fix(thing), nocompress=nocompress,_extra=_extra)
end if size(thing))[0] gt 1 then begin
  nd = (size(thing))[0]
  dims = size(thing,/dimensions)
  nz = dims[nd-1]
  leading_dims = dims[0:nd-2]
  thing2 = transpose(thing,[nd-1,indgen(nd-1)])
  thing2 = reform(thing2,[nz,n_elements(thing2)/nz])
  retval = "[ "
  for i=0,nz-1 do begin
    if i gt 0 then retval += ", "
    thing3 = reform(thing2[i,*],leading_dims)
    retval += aprint(thing3)
  endfor
  retval += " ]"
end
if keyword_set(nocompress) then return,"[ "+delim2+strjoin(thing,delimiter)+delim2+" ]"
return,"[ "+delim2+strc(thing,/join,delimiter=_extra=_extra)+delim2+" ]"
end
See how hard this is to read? **DO NOT WRITE CODE LIKE THIS.** If you do we will take off
points on assignments as it shows that you do not understand the importance of having well written
code.
To summarize there are three basic things to good code that are independent of the language you are
using:

1. **Keep variable names short but explicit**
   Do not name variables things like `a`, `data`, `result`, etc.
2. **Indent your code**
   Indenting and whitespace allows for easier readability of code
3. **Comment your code**
   Comments are invaluable and are present in all programming languages for a reason.
3.5 Predefined Procedures & Functions

As we began to discuss earlier in the chapter, IDL contains a wide variety of predefined procedures and functions that make our lives easier since we don’t have to write them. These often solve simple problems such as: *How do I obtain how many elements are in an array?* or *How do I sum all the elements in my array?* Here we detail some of the more relevant procedures and functions so that you can use them in your own code.

- **n_elements()**: Hopefully the purpose of this function is clear. As mentioned above, `n_elements` returns the number of elements in an array or list.
- **total()**: This function should also be straightforward; `total` returns the sum of all the elements in the array.
- **min()**: `min` returns the minimum value of the elements in an array or list.
- **max()**: Similar to `min`, `max` returns the maximum value.
- **avg()**: Remember that function you wrote to find the average of an array? Well, you could have just used `avg`, but then you wouldn’t have learned anything.
- **reverse()**: The name of `reverse` should be a dead giveaway. This function reverses the position of all the elements. The first element becomes the last element, the second element becomes the second to last element, and so on.
- **size()**: This function, as you might have guessed, returns the size of an array. However, by simply calling `size`, it might be difficult to guess what its output is telling you. The first number indicates the dimension of the array (we’ll get into multi-dimensional array in a later chapter). The next few numbers indicate the number of elements in each dimension. The penultimate number tells the data type. Finally, the last number is the total number of elements in your array.
- **stdev()**: Despite what you might be thinking, this function returns the standard deviation of an array or list.
- **readcol,<filename>,v1,v2,...**: `readcol` is a procedure that is incredibly useful when dealing with data files. As its name suggests, if the data file is arranged in columns, these columns will be read into the vectors v1, v2, etc.
At this point you should feel comfortable writing a basic procedure or function that can do simple mathematical tasks on some type of input. However in most situations we don’t want to perform some operation on just one file, image, set of data, etc. but multiple. We could, in principle, just copy and paste whatever function we need and use it on every file, image, set of data, etc. however that is both tedious and is prone to user-error. A better solution is to use control statements which, as you may guess from the name, control the way our input is manipulated. Control statements exist in all programming languages, the most basic of which are generally split into loops and conditional statements.

4.1 For Loops

As the name suggests, loops allow us to easily repeat a process over and over again until we decide we want the process to stop. For loops are probably the easiest to understand. Lets say we wanted to print our old friend Hello World 5 times. We could call the procedure we wrote in the previous chapter 5 times or we could just use a for loop. Lets look at our old hello_p procedure,

```idl
pro hello_p
  print, 'Hello World'
end
```

Pretty basic so how do we add a for loop? Much like our procedures and functions, each loop in IDL must have a declaration that we want to use a for loop followed by an endif statement,
Chapter 4. Control Statements

```idl
pro hello_p

   for
       print, 'Hello World'
   endfor

end
```

Notice how what we want to repeat is indented to be within the for loop. This is common notation and is one part of what we mean when we say to indent your code. Continuing on with our example we still haven’t told IDL how to begin and end this loop. For loops use something called an index variable which by convention is taken to be i however if a different name suits your purposes any variable name can be used.

```idl
pro hello_p

   for i=0, 4 do begin
       print, 'Hello World'
   endfor

end
```

How does this loop operate? The easiest way to think about how any program operates is to do what I call following the code which means to look at every statement and think about what’s happening in each step. This is a general rule that is applicable far beyond loops and can help you understand otherwise gibberish code very quickly.

Let’s apply this method to our loop here. Our index variable is first declared to be i=0, we then tell IDL we want the loop to stop when i=4, and then the loop starts. Next our procedure prints Hello World and reaches the end of the loop. Though it isn’t written explicitly, IDL checks to see what value i is which, in the first run-through of the loop, is i=0. Since i=0 is not i=4, then the loop jumps back up to the for statement, adds 1 to i so that now i=1 and begins the loop over again. This continues as i=2, i=3, to i=4 where then IDL stops our loop and since there is no code left, ends the procedure.

We can then call this in IDL,

```
IDL> .com hello_world.pro
% Compiled module: HELLO_P.
IDL> hello_p
Hello World
Hello World
Hello World
Hello World
Hello World
```

which has successfully printed Hello World 5 times. In English you can read this code as: For i=0 through i=4 do these things: print Hello World. You may be wondering why I didn’t write the code as follows,
4.1 For Loops

```idl
pro hello_p
    for i=1, 5 do begin
        print, 'Hello World'
    endfor
end
```

since starting with i=1 and ending on i=5 seems to make more sense. You’ll see why I chose i=0 in the next example.

**Arrays and For Loops**

In most uses of for loops you will not hard-code in things like i=0 and i=4 but rather use for loops in combination with arrays. So in a slight deviation from our discussion on loops there are a few things we must understand about arrays before continuing. As in the last chapter you can still continue to think of arrays as a simple list of numbers.

First we’ll declare an array as a variable in our interpreter,

```idl
IDL> foo=[8,5,6,7,9,10]
```

and now say we want to access the number 6 out of the array and use it for a calculation. How do we do this? In the same way that our for loop is indexed with a variable so too is our array and we use brackets to denote which element of the array we want to access. You’re first impression may be to do something like

```idl
IDL> print, foo[3]
7
```

but as you can see this gives you the wrong element. To get the correct element we must enter,

```idl
IDL> print, foo[2]
6
```

Why does IDL give me the 3rd element when I ask for foo[2]? This is because in IDL indexing starts at 0 such that if we want the first element, which in our case is 8, then we must enter,

```idl
IDL> print, foo[0]
8
```

When I first took this course I thought this was really stupid; why define 0 as the 1st element? The reason is because of how computers, not humans, interpret numbers. From a human’s perspective it seems logical to use 1 as the 1st element mostly because of our 5-digit fingers where 1 means raising the first finger. However computers do not have hands (yet) and so do not need to abide by these rules. In which case it seems perfectly logical to use 0 as the index for the 1st element.

Hopefully this discussion has given you a fair reason why I chose to use i=0 instead of i=1 in our Hello World example. However to be explicit lets use an array, not hard coded numbers, with a for loop to see it in action.
Chapter 4. Control Statements

Let's say we just want to print out each element of foo using a loop. To do this we'll need to iterate through each element, print that element and then have the loop repeat itself. You probably guessed that the structure of this procedure will look very much like our hello_p procedure,

```idl
pro print_elements, input_arr

; prints the elements of input_arr

for
  print, input_arr[i]
endfor
end
```

How do we access the elements? We use our index variable i to tell us what element to print. However how will the loop know where to end? The easiest thing to do is use our predefined function n_elements to tell the loop when to stop.

```idl
pro print_elements, input_arr

; prints the elements of input_arr

for i=0, n_elements(input_arr)-1 do begin
  print, input_arr[i]
endfor
end
```

How does this work? In the first iteration i=0 and so during the first pass our print statement will see print, input_arr[0] which prints the first element. Then, because for loops by default add one to the index variable, i becomes i=0+1=1 and on the next pass the print statement reads print, input_arr[1] which prints the second element and so on until we reach n_elements(input_arr)-1. Why the -1 you may ask? This goes back to the fact that in IDL, indexing starts at 0. To be explicit let's take foo as defined above and enter into the interpreter,

IDL> print, n_elements(foo)
6

and you can see that n_elements print out how many elements there are as if 1 was the index for the first element. However if we enter,

IDL> print, foo[6]
% Attempt to subscript FOO with <INT (       6)> is out of range.
% Execution halted at: $MAIN$

you can see that we get an error because arrays index as if 0, not 1, is the first element. So in our loop we can't enter something like print, foo[i] where i=n_elements(foo). Rather we have to ensure that the last element is n_elements(foo)-1 to correct for the indexing.

Looking at the output of print_elements,
4.2 Conditional Statements

However what if we wanted to print only the even-indexed elements, I.E. `foo[0]`, `foo[2]`, etc? We can just change how our index variable changes after every iteration of the loop by saying that instead of adding 1 it should add 2. The syntax for this is as follows,

```idl
pro print_elements, input_arr
   ;prints the elements of input_arr
   for i=0, n_elements(input_arr)-1, 2 do begin
      print, input_arr[i]
   endfor
end
```

where in a general case 2 can be any positive (iterate upwards) or negative (iterate downwards) number that we would like to iterate by. We can then look at the output,

```
IDL> print_elements, foo
8
6
9
```

which is what we desired.

4.2 Conditional Statements

Before getting into the second type of loop known as a **while loop** we must first discuss the control statement that allows while loops to function which are called conditional statements. Conditional statements operate much in the same way that truth tables work; IF some type of condition is met THEN preform an operation. There are various types of conditional statements within IDL but we begin with the basic **If Then Else** statement.

4.2.1 If Then Else

As their name suggests, *if then else* statements check some type of condition before preforming some type of action. Lets make a program that will print some output, here a string of numbers, only if the correct password is input,
pro secret, guess

;gives you the output upon correct guess of the password
if <some condition> then begin
    print, 'See You Space Cowboy'
endif
end

(brownie points for those of you who recognize the output without the help of google). What type of condition do we need here? Well we would want guess to *equal* the correct password. To do this we need to use IDL’s *relational operator* for equal which is written as EQ,

pro secret, guess

;gives you the output upon correct guess of the password
if guess EQ 'spike' then begin
    print, 'See You Space Cowboy'
endif
end

Lets try some false answers followed by the correct password,

IDL> secret, 'idk'
IDL> secret, 'what'
IDL> secret, 'password'
IDL> secret, 'spike'
See You Space Cowboy

where you can see that our password of *spike* gives the output we desired. The possible relational operators are:

1. **EQ**
   Equal to
2. **GT**
   Greater Than
3. **GE**
   Greater than or Equal to
4. **LT**
   Less Than
5. **LE**
   Less than or Equal to
6. **NE**
   Not Equal to

What if we wanted to include a statement that tells the user that they didn’t get the password right? Then we need the *else* part of the if then else statement,

pro secret, guess
4.2 Conditional Statements

; gives you the output upon correct guess of the password
if guess EQ 'spike' then begin
  print, 'See You Space Cowboy'
endif else begin
  print, 'Guess Again Faye'
endelse

end

So now incorrect guesses will generate,

IDL> secret, 'password'
Guess Again Faye
IDL> secret, 'spike'
See You Space Cowboy

Now what if we wanted a different type of output that corresponded to a different password? We nest if-else statements together,

pro secret, guess

  ; gives you the output upon correct guess of the password
  if guess EQ 'spike' then begin
    print, 'See You Space Cowboy'
  endif else begin
    if guess EQ 'julia' then begin
      print, 'Its All A Dream'
    endif else begin
      print, 'Guess Again Faye'
    endelse
  endelse

end

where now our output will be,

IDL> secret, 'idk'
Guess Again Faye
IDL> secret, 'spike'
See You Space Cowboy
IDL> secret, 'julia'
Its All A Dream

as we expected. While we could nest many if-else statements IDL allows us to avoid the cumbersome syntax by use of case statements.

4.2.2 Case and Switch Statements

Case Statements

Like nested if-else statements, case statements allow us to pick out one case that we can follow. For instance writing something like
pro test_case, input_case

    case input_case of
        1: print, 'one'
        2: print, 'two'
        3: print, 'three'
        4: print, 'four'
    else: begin
        print, 'You entered: ', input_case
        print, 'Please enter a value between 1 and 4'
    end
endcase
end

will result in the following output,

IDL> test_case, 2
    two

IDL> test_case, 5
    You entered: 5
    Please enter a value between 1 and 4

IDL> test_case, 'nope'
    You entered: nope
    Please enter a value between 1 and 4

where you can see that if we want multiple statements within one case then we must wrap that case in a begin and end statement just as we did here for the else case. Note that if we change the code to be,

pro test_case, input_case

    case input_case of
        1: print, 'one'
        2: print, 'two'
        2: print, 'also two'
        4: print, 'four'
    else: begin
        print, 'You entered: ', input_case
        print, 'Please enter a value between 1 and 4'
    end
endcase
end

we will never be able to access the 'also two' output because case statements only allow one case to be chosen;

IDL> test_case, 2
    two
4.2 Conditional Statements

**Switch Statements**

Unlike a case statement which will only follow one possible case, a switch statement, as the name suggests, will switch on and execute all of the statements which follow the first switch. To be more explicit say we have the code,

```idl
pro test_switch, input_switch

    switch input_switch of
        1: print, 'one'
        2: print, 'two'
        3: print, 'three'
        4: print, 'four'
    endswitch

end
```

then the output will be

IDL> test_switch, 2
two
tree
four

where our input 2, acted as our switch.

Imagine that, for whatever reason, we didn’t want to print out four when we input input_switch=1 but we still wanted to use our switch statement. How do we exit the switch statement when we want to? The way we break out of any loop, case, or switch statement is to use a break statement as shown,

```idl
pro test_switch, input_switch

    switch input_switch of
        1: print, 'one'
        2: print, 'two'
        3: print, 'three'
        4: begin
            if input_switch EQ 1 then break
            print, 'four'
        end
    endswitch

end
```

which will generate the output,

IDL> test_switch, 2
two
tree
four
IDL> test_switch, 1
one
as desired. Note that break statements can be used to break out of loops, case, or switch statements but will not break you out of the more simplistic if then else statements.

4.3 While Loops

Now that you understand the basics of both for loops and conditional statements we can move on to our last topic; while loops. As the name suggests while loops will continue to loop while some conditional statement is true. The general syntax for such a loop is as follows,

```
pro test_while
  while <some condition> do begin
    <do something>
  endwhile
end
```

The most basic (and useless) type of while loop is one that goes on forever which we call in programming an infinite loop. In IDL conditional statements 0 corresponds to False and 1 corresponds to True. Such that if we write the following code,

```
pro test_while
  while 1 do begin
    print, 'Hi'
  endwhile
end
```

then IDL will continuously print Hi until you exit the interpreter or stop the process through some external means. Where while loops come into use is where we can control the conditional statement. There are multiple ways to do this. One way is to use a while loop as a pseudo-for-loop where we again create an index variable and iterate,

```
pro test_while
  x=0
  while x LE 5 do begin
    print, 'Hi'
    x+=1
  endwhile
end
```

and this code will output,

IDL> test_while
Hi
The `x+=1` syntax and its equivalent `x=x+1` still may be odd to those of you new to programming. Let's follow the code and see how the above while loop works. First we declare `x=0`. Next the while statement checks to see if `x LE 5` which it is and so the loop begins. The print statement is then executed. However unlike our for loop we must manually increment the index variable `x` by the statement `x+=1`. How does this work? Let's look at the equivalent `x=x+1`. In our first pass through the loop `x=0` and we are telling IDL to redefine `x=x+1`, which in the first pass is really seen as `x=0+1=1`. The best way to think of statements that redefine variables like `x=x+1` is to imagine that the right-hand side of the equation operates by using previous definitions of variables and the left-hand side is what will be defined.

In addition to being glorified for loops, while loops are most often used when some process needs to be repeated until some condition is met. In the past a lot of student's final projects have revolved around some type of game where there is a possible winning condition. Although it is not efficient, many students will wrap the code of their game into a huge while loop and when the player wins the game the code will check to see if the winning condition is met.

### 4.4 Summary

All of these control statements allow us to control when different parts of the code are accessed. On their own, each loop or conditional statement is relatively simple. However when they are combined together they have powerful data processing capabilities. I should note that in this text we have not detailed all of the possible control statements and have skipped phrases such as `continue`, `goto`, etc. However the need for such phrases will come with practice and are well documented in the online documentation. If you're rereading this chapter and looking for something for your final project I highly suggest checking out the online documentation to see if there isn't something more efficient than the statements we detailed here that could solve your problem.
5. Arrays

In the previous chapters we have told you that you can think of arrays as simply a list of numbers like,

IDL> foo=[42, 9, 3, 6, 5]

but in this chapter we will begin to explore the true power of storing data in arrays. Firstly we haven’t discussed how arrays deal with datatypes. Secondly there are many tricks to simple array manipulation that we have yet to discuss. Thirdly, above is an example of what we call a one-dimensional array since there is only one dimension to the array itself. This is the simplest and thus the most often used type of array in computer programming however IDL supports two, three, four, and up to N-dimensional arrays.

5.1 Arrays, Datatypes, Lists

One of the most limiting (or convienent, depending upon what you’re trying to accomplish) aspects of IDL arrays are that they can only store one datatype. Above we defined

IDL> foo=[42, 9, 3, 6, 5]

and this works because every element is one datatype; an integer. We can try to mix integer and floats,

IDL> bar=[15, 2, 14.56, 8, 5.67]
IDL> print, bar[0]
15.000000

and while IDL has let us define bar you can see that the elements we intended to be integers just are changed to floats. If we try to force something like a mix of strings and numerical datatypes, IDL yells at us saying that we can’t do that;
IDL> foo=[2,10,15.5,'test',4]
% Type conversion error: Unable to convert given STRING to Integer.
% Detected at: $MAIN$

Why restrict arrays to only one datatype? One of the benefits of this is to prevent routine errors from occurring. Say you’re loading an array with some data you want to analyze and you accidentally change all your code to input a string into your otherwise numerical array. Rather than finding out about it after you realized something has gone wrong in your calculation IDL clearly points out the problem beforehand.

However you can imagine there would be instances where we want multiple datatypes stored in something easily accessible; for instance the name of a star and associated spectra, telescope images, etc. How do we accomplish this in IDL? The smart way is to use something called a structure which is involved enough to warrant its own separate discussion in a later chapter however the quick and dirty way is to use a list.

### 5.1.1 Lists

To create a list we use the aptly named list function. In the simplest case, we just input the elements we want to be in the list.

IDL> foo = list(2, 10, 15.5, 'test', 4)

This will create a list called ‘foo’ that will contain all the values that were giving us errors before.

Another way to create a list is to use the list function to join two arrays

IDL> x = [3, 4, 2, 5]
IDL> y = [1, 6, 7, 2]
IDL> z = list(x, y)
IDL> print, z
3 4 2 5
1 6 7 2

The only problem here is that IDL interprets each array as an element of the list. By saying z=list(x,y) we have told IDL to make a list of arrays. Therefore, typing print, z[0] gives all the elements in the first array. If instead we want to use the elements in the arrays to make a list (a subtle but important difference) we have to use the extract keyword when we call list

IDL> z = list(x, y, /extract)
IDL> print, z
3 4 2 5 1 6 7 2
Now, typing \texttt{print, z[0]} will return 3 as expected. Lists also have some useful features called ‘methods’ that allow them to be edited and analyzed quickly. Let’s say you wanted to add an element to the list you just made. Instead of recreating the list, you can use the \texttt{list::add} procedure

\texttt{IDL> z.add, 8675309}

This adds the value 8675309 to \texttt{z}. Another useful method is the \texttt{list::count} function. This can be thought of as an \texttt{n_elements} function specifically made for lists. For example,

\texttt{IDL> print, z.count()}
\hspace{1cm} 9

Lists contain a wide variety of other useful methods allowing you to swap elements, reverse the order of the list, remove elements, etc.; however we do not detail them here. We have included a link to the online documentation that covers lists on the class website. Really, just be aware that if you need multiple datatypes stored into one object and do not want to use a structure then lists are your best choice.

Note that on the online documentation there are methods called Map, Reduce, and a discussion of things called Lambda functions. If you are interested feel free to read about these higher-order functions but know that we will not cover them in this class and that Lambda functions are not supported by IDL 8.2.

### 5.2 Advanced Array Manipulation

While lists allow you to use multiple datatypes to store information, arrays can be extremely powerful when you are able, as is often the case, to restrict yourself to only using one datatype. In our previous discussions we’ve always begun by declaring some array by explicitly hard-coding in the elements such as,

\texttt{IDL> foo=[5,6,3,4,6]}

However there are other ways to create arrays using \texttt{array-generating functions}.

#### 5.2.1 Array Generators

While declaring arrays explicitly is fine for smaller applications you can imagine that if we would need a very large array, say of order $10^3$ elements, then it would be painful and prone to user error to create an array by typing out $10^3$ 0’s. Luckily IDL has built-in functions called array-generators which allow us to do this using rather intuitive syntax. Starting with a simple example we can construct a 10 element array filled with only 0’s by simply entering,

\texttt{IDL> ten_int=intarr(10)}
\texttt{IDL> print, ten_int}
\hspace{1cm} 0 0 0 0 0 0 0 0 0 0
Chapter 5. Arrays

Here you can see we’ve used the `intarr` function and have told IDL to build an array (hence the `arr` suffix) containing only integers (hence the `int` prefix). Our input of 10 told IDL that we want 10 elements in this array and thus that is what we see in the output.

This class of `arr` functions extends to supporting multiple datatypes. For instance if we wanted a 10-element array consisting of floats instead of integers we could have used the `fltarr` function as shown,

```
IDL> ten_flt=fltarr(10)
IDL> print, ten_flt
0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
```

We can also go beyond creating arrays filled with zeros and use another class of array generating functions that use the `gen` suffix. For instance, if we replace `intarr` with `indgen` then we get the following output,

```
IDL> ten_int=indgen(10)
IDL> print, ten_int
0 1 2 3 4 5 6 7 8 9
```

and a similar output with floats would follow if we used the `findgen` function. The fact that the prefixes have changed from `int` and `flt` to `ind` and `find` may seem odd but are simply for easy of pronunciation. Note that there are more array generating functions available in IDL however these are the four most widely used.

### 5.2.2 Playing With Arrays

Now that you understand the different ways you can create arrays we’ll begin discussing how to manipulate them. For instance we’ll declare an array filled with the numbers 0 through 9,

```
IDL> gunter=indgen(10)
IDL> print, gunter
0 1 2 3 4 5 6 7 8 9
```

Say however we didn’t want 0 through 10 but instead 10 through 19. We could manipulate each element individually by calling

```
IDL> gunter[0]=gunter[0]+10
IDL> print, gunter
10 1 2 3 4 5 6 7 8 9
IDL> print, gunter
10 11 2 3 4 5 6 7 8 9
```

all the way through our array but as you can see that is tedious and prone to user error. A better solution would be to use a for loop to do this for us automatically,
```
function add_one, input_arr

    return_arr=intarr(10); declare an output array

    for i=0, n_elements(input_arr)-1 do begin; for every element
        return_arr[i]=input_arr[i]+10; add 10 to each element
    endfor

    return, return_arr; return the output

end
```

Then using our newly created function,

IDL> print, add_one(gunter)

10 11 12 13 14 15 16 17 18 19

So we get the correct output but you can also notice how much code we’ve written for such a simple problem. The best way (Re: the most efficient way) to solve this problem is the one-line solution,

IDL> gunter=indgen(10)+10
IDL> print, gunter

10 11 12 13 14 15 16 17 18 19

which says; “first make an array of 0-9 integers and then add 10 to each integer”. This takes no script writing and no for-loop usage. This is a perfect example of how semi-understanding programming can lead to ”going down a rabbit hole” of unnecessary complication and illustrates how understanding syntax can lead to quick and efficient solutions.

### 5.3 N-Dimensional Arrays

While restraining oneself to 1-dimensional, a.k.a 1-D arrays, can be beneficial for their simplicity and ease of control, expanding out to N-D arrays are useful when there is cause to do so.

#### 5.3.1 2-D Arrays

We’ll begin our discussion by only adding one dimension and discuss 2-D arrays. In the context of astronomy these arrays are most often used to store data regarding scientific images from telescopes (a discussion which in itself warrants its own chapter). Here however we detail how to use and control 2-D arrays.

To begin with we’ll define a simple 2x2 element 2-D array,

IDL> brule=[[9,8],[7,5]]
IDL> print, brule

<table>
<thead>
<tr>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

The first question you should ask is: *How do I access the different elements?* The simplest (though not often the best) way to index arrays is still by the `arr[i]` indexing we discussed when using 1-D arrays. For example we can access every element in `brule` by simply entering,
IDL> print, brule[0]
9
IDL> print, brule[1]
8
IDL> print, brule[2]
7
IDL> print, brule[3]
5

However you can imagine that we would like at access multiple elements at a time. IDL supports this possibility by allowing us to access both rows (in our brule example the elements 9, 8 make up the first row) and columns (in our brule example the elements 9, 7 make up the first column). However, since we have two dimensions there are two possible conventions to index arrays; either we first reference the column then the row or first the row then the column. By convention IDL follows the column then row reference convention for reasons that will be made clear in the FITS file chapter. This type of referencing convention is called column-major indexing since the column comes before the row in the reference.

If you’re like me then the above section was nothing but a bunch of gibberish; lets see whats discussed above with a visual example. If I want to call the first column then I can enter,

IDL> print, brule[0, *]
9
7

and you can see the first column has been printed. You may recognize the * symbol from our discussion in Bash where it served as a wildcard meaning that it symbolized all possibilities. This notation carries over to IDL and in this example by entering print, brule[0, *] we’ve asked IDL to print everything in column 0 which is equivalently saying to print the first column. We can also ask IDL to print the second column,

IDL> print, brule[1, *]
8
5

and you can see we get the output we expect. Hopefully you can see that we could also reference the first and second rows in a similar way,

IDL> print, brule[*, 0]
9
8
IDL> print, brule[*, 1]
7
5

and that we can also reference any single element we desire if we replace * with a specific reference,

IDL> print, brule[0, 1]
7

We can also create 2-D arrays by using our normal array generators. Though not discussed in the last section, the input of all array generators actually references the dimensionality of the desired array. To see what this means we can create a 2x2 integer array filled with zeros by using the intarr function,
and this type of convention carries over to all array generator functions.

### 5.3.2 N-D Arrays

IDL’s referencing system for 2-D arrays of column then row continues by logical extension to N-D arrays though it becomes harder to visualize since the output on your screen is 2-D. As an example lets make a 3x3x3 ”box array” that contains a total of $3 \times 3 \times 3 = 27$ elements,

IDL> jake=intarr(3,3,3)
IDL> print, jake

```
0 0 0
0 0 0
0 0 0

0 0 0
0 0 0
0 0 0

0 0 0
0 0 0
0 0 0
```

How should you visualize this output? You can see that each 3x3 ”slice” is seperated by a bit of whitespace. Imagine that the 2nd slice is ”behind” the 1st slice and the 3rd slice is ”behind” the 2nd slice such that this array is a cube rather than a collection of slices. A visual example of what I’m describing is provided in Figure 5.1. Using this visual analogy it becomes easier to understand the indexing system for 3-D arrays. Since now we have 3 dimensions we have $3! = 3 \times 2 \times 1 = 6$ possible ways to index this array. IDL uses the convention of column, row, slice. To see this lets add 5 to the first column;

IDL> jake[0,*,*]+=5

![Figure 5.1: A visual example of a 3-D array.](image-url)
IDL> print, jake
    5  0  0
    5  0  0
    5  0  0

    5  0  0
    5  0  0
    5  0  0

    5  0  0
    5  0  0
    5  0  0

As you can see the first column, in all 3 slices, increased by 5. We can also add 6 to the first row and 7 to the first slice to produce,

IDL> jake[*,0,*,] += 6
IDL> jake[*,*,0] += 7
IDL> print, jake
    18  13  13
    12  7  7
    12  7  7

    11  6  6
    5  0  0
    5  0  0

    11  6  6
    5  0  0
    5  0  0

You can see now that the first row in each slice has increased by 6 (and in the first columns 5+6=11) and the first slice has increased by 7 (where in the first element of the first slice 5+6+7=18). This convention of column, row, slice,... continues out to 4-D, 5-D, etc. arrays but the spatial visualization becomes impossible at 4-D since we live in a 3-D universe. Do note that 4-D and above arrays are used in certain instances to save on computer memory but that they will not be a common sight at the level of programming we do in this class.

5.4 Summary

Arrays are one of the most useful objects within IDL and are the powerhouse behind image processing and data analysis. Now that you are acquainted with the very basics of any programming language (writing scripts, control statements, and arrays) we can now begin to do some real science using IDL to help us. In the next chapter we detail how to output the results of calculations, done using control statements and arrays, in a visual manner that can be interpreted to give a scientific result.
6. Data Analysis and Plotting

6.1 Reading and Writing Data

The key elements of any type of scientific process are; I have an idea! Followed by, Is my idea right? Followed by Let’s test it! Then, assuming you can develop a reliable test, you’ll see if your idea is correct or not. This chapter is concerned with the logistics of the Let’s test it! phase where you have developed an idea but need to compare it to data in order to verify it. In astronomy we are often dealing with very large volumes of data (often on the gigabyte or terabyte scale) so in order to analyze this large volume of data we must use automated processes, created with a programming language like IDL, to assist us. To begin to analyze data with IDL we must learn how IDL can read and write data from and to different types of files.

6.1.1 Bash and IDL

Before delving into playing with data within IDL it is useful to know that its possible to issue Bash commands using IDL. There are two ways to interact with Bash through IDL and the first and simpliest way requires you to use the interpreter directly (Re: you can’t use this method in a script) and uses a $. If we want to ls the contents of our directory from in the interpreter we can just enter

IDL>$ls

or any other Bash command we desire like,

IDL>$cd ~/Desktop

if we wanted to get to our desktop.

The second way allows us to use either the interpreter or a script. This is done using the spawn command. The name comes from the idea that IDL will ”spawn” an instance of Bash and then issue commands using the same syntax that Bash uses. Other languages have similar capabilities (see Python’s os module if you’re interested in an example) but they are often cumbersome and non-intuitive in comparison to IDL’s simple syntax.
To illustrate the syntax of `spawn` lets look at a real-world example. Lets say you’re beginning to analyze your data stored in a file like `/home/jzalesky/project/data.txt` (do not name files like this, I’m just using the names for simplicity). However in your analysis you want to edit `data.txt` and therefore it is possible to screw up the file and ruin your data if you mistype something in your code that edits the file. One way to prevent complete calamity is to create a copy of `data.txt` before touching the file. We know that Bash can easily do this with the `cp` command and thus we can use `spawn` to issue the command in our script as follows,

```idl
pro data_analysis, path_to_file

;create a copy before analyzing
spawn, 'cp '+path_to_file+' /home/jzalesky/project/backup/data_backup.txt'

;analysis code
end
```

As you can see `spawn` takes a string which contains a Bash command as an input. Assuming we call the procedure with the following input,

```
IDL> data_analysis, '/home/jzalesky/project/data.txt'
```

then this is equivalent to issuing the following command in Bash,

```
 cp /home/jzalesky/project/data.txt /home/jzalesky/project/backup/data_backup.txt
```

You may have been wondering why I’ve used complete paths containing `/home/jzalesky`/instead of just relative paths like,

```
 cp data.txt backup/data_backup.txt
```

since the relative paths require less work. The reason I don’t do this is because using relative paths with `spawn` is a bad idea. If I open IDL while I’m in the `/home/jzalesky/project` directory the relative-paths work out fine because I’m operating within that directory. However there is no guarantee that when I want to use the `data_analysis` procedure at a later date (Re: months after writing the code) that I’ll remember to `cd` into the projects directory. The general rule of use complete path references when creating anything automated will save you a lot of headache in the future despite it requiring a little bit of extra work when writing the code.

### 6.1.2 Constant System Variables, Save, and Restore

#### Constant System Variables

Those of you who have taken some courses in the physics or astronomy department will know how helpful it is to have a reference sheet containing a lot of physical constants: the speed of light, the universal gravitational constant, the conversion ratio between radians and degrees, etc. IDL contains a wide variety of Constant System Variables which are pre-defined in the back-end of IDL and can be used in any IDL session. For instance to call up the numerical value of \( \pi \) and \( G \) we can just enter,

```
IDL> print, !pi
3.1415927
IDL> print, !const.G
6.6742799999999995e-11
```
where $G$ has been given in SI units. IDL has a ton of physical constants stored under !const and other useful references stored under other names but always beginning with ! (for instance you can get RGB values of a ton of colors using !color). For a complete list of these constants look in our IDL Resources section of the website for the IDL Constants link.

**Save & Restore**
You can imagine that it would be useful to not only create your own constants but other procedures and functions that you use quite often. In addition if someone else creates a procedure/function that would be of use to you it would be nice to easily access that function without having to .com a ton of .pro files. To solve both of these issues IDL has save and restore to help us out.

Save, as the name suggests, can save variables and procedures/functions that have been defined within an session of IDL. As an example let say I start up IDL and define,

```
IDL> kimiko_ross='mez'
```

and then I quit the IDL session, start IDL back up, and try to access my definition of kimiko_ross,

```
IDL> exit
jzalesky@aquarius:~$ idl
IDL> print, kimiko_ross
% Attempt to call undefined procedure: 'KIMIKO_ROSS'.
% Execution halted at: $MAIN$
```

And as you can see IDL has no idea that kimiko_ross='mez' because that definition was only meant to last within the first session of IDL. However if I wanted to store kimiko_ross='mez' as a variable that I could easily access later then I can use the save procedure,

```
IDL> kimiko_ross='mez'
IDL> save, /variables, filename='/home/jzalesky/Desktop/dresden_codak.sav'
```

In the last step I’ve told IDL to save all currently defined variables (the only one here is kimiko_ross) into a file on my Desktop called dresden_codak.sav. The .sav extension is so that IDL knows its a file which save and restore can access.

Now let’s exit IDL, start a new session, and restore the variable called kimiko_ross so that we can use it in the new session,

```
IDL> exit
jzalesky@aquarius:~$ idl
IDL> restore, '/home/jzalesky/Desktop/dresden_codak.sav'
IDL> print, kimiko_ross
mez
```

and you can see that IDL now knows what kimiko_ross is.

You can also use save and restore to do the same thing with procedures and functions that you often use. Simply .com whatever functions you would like to use so that they’re defined within your current IDL session and save them into a different .sav file. Note that you cannot save both procedures/functions and variables into the same .savfile,
IDL> .com sav_and_res.pro  
% Compiled module: KAITO_KUSANAGI.  
% Compiled module: TINYCARLJUNG.  
IDL> save, /routines, 'kaito_kusanagi', 'tinycarljung', $  
filename='/home/joseph/Desktop/routines.sav'

and then we could restore routines.sav in the same way as before with our variables. Note that the $ allows me to extend the save command beyond one line and is used for simple readability of the command.

Save and Restore are useful if you're using them for just yourself but their real power comes from combining your ideas with things others have created. You can often find great scripts online that solve complicated issues for you as IDL has been around for almost 40 years and there's a good chance someone else has run into the same issue you're having. A particularly good reference is Coyote's Guide to IDL which contains a lot of useful procedures and functions, mostly involving plotting. For a link see our IDL Resources on the class website.

6.1.3 Reading Data with Readcol

For almost all scientific applications, data is often stored in column format. For instance let's say you're looking at a simulation of a planetary system and someone has recorded the orbital parameters of one of the planets into an ASCII file (that's just a fancy name for any text file). It would make sense that you would store all of that information into columns as shown in Figure 6.1. We need to read this data into IDL so that we can use IDL to manipulate it. For column-formatted data we have a procedure called readcol. readcol can take a variety of keywords and arguments to format the data in the way you desire so let's try to use it on the data shown in Figure 6.1.

Note that I have this file stored on my computer in planet1.aei. Note that readcol does not require the extension to be .txt but rather just that the data is stored in the ASCII format (google ASCII for more info).

You can see from the figure that there are some problems we'll have to address with this file. First of all we need to note that we have 8 columns of data, corresponding to the 6 orbital elements, 1 column for a record of how much time has passed within the simulation, and 1 column that contains the mass of the planet (if you're wondering that's in case of collisions where the mass will change for any specific object). The other problem is that there are a few lines at the top that do not contain any data so we won't want to read those into IDL.

To read this data in we'll issue the following command,

IDL> readcol, 'planet1.aei', skipline=5, t, a, e, i, peri, node, mass
IDL> print, t
10000.000 20000.000 30000.000 40000.000 50000.000 60000.000 70000.000 80000.000 90000.000 100000.000 110000.000 120000.000 130000.000 140000.000 150000.000 ...

This tells IDL to read in the columns of planet1.aei into the variables t, a, e, etc. and to skip the first 5 lines from the top. Note that t is assigned to the first column only because we listed it first in our readcol command. If we had switched the places of t and mass for instance,

IDL> readcol, 'planet1.aei', skipline=5, mass, a, e, i, peri, node, t
Figure 6.1: An example of how data can be stored using column formatting.
IDL> print, mass

10000.000  20000.000  30000.000  40000.000  50000.000
60000.000  70000.000  80000.000  90000.000  100000.00
110000.00  120000.00  130000.00  140000.00  150000.00
...

Then mass is now the variable that contains the time passed within the simulation. We could have called these variables anything we wanted but the names above are the most intuitive (to those of you who are interested in planetary astronomy look up orbital parameters in google to understand why). Additionally if we only wanted the first 3 columns (time, a, and e) then we could have just issued,

IDL> readcol, 'planet1.aei', skipline=5, t, a, e

and this would work just as well. Note that here we’ve only done one example of readcol and that other problems could creep up with different data files. For instance there could have been extra text at the bottom of the file that we would have needed to screen out or maybe we only wanted the first 10 lines of each orbital element. To address all of these issues look into the readcol documentation online to see if it offers a way to solve your specific problem.

### 6.1.4 Reading and Writing Data

IDL also supports much more robust file manipulation beyond the simple readcol and the class website contains a link listing out all the possible routines. However it is rare to need such procedures and functions at the level of an undergraduate and thus we detail only the vital few commands that should get you started.

A common problem is that you’re running some code but the output in the terminal scrolls past far too quickly for you to interpret. In order to address this it is rather simple to write to a log file which contains a log of output regarding your code. This is useful to see if some conditional is met, if you’re code is stuck hanging, or if you’re debugging code. Let’s assume you want to check to see if some conditional is met. The easiest thing to do is use printf which is short for ”print to file” and it operates much like the print procedure. We will use printf in combination with openw and close.

```idl
pro data_ana

;some code that does some cool data analysis stuff

;check a conditional
if condit EQ 1 then begin
   openw, 1, '/home/jzalesky/logs/log_1.txt'
   printf, 1, 'condit is True'
   close, 1
endif

end
```

In this code we first use an if statement to check if condit is true. If it is then we first use openw, short for ”open, write” to write to our log_1.txt file (note the full path is specified). We use the number 1 to enumerate log_1.txt as it is common to need to write to multiple files at once. We
then use printf to write the string condit is True to log_1.txt and finally close log_1.txt. Note that what we’ve simply output a string but we could also output some type of numerical result.

```idl
pro data_ana
    ;some code that gets the mass (M) of the host star and semi-major axis (a)

    ;calculate a planet’s period^2 assuming a simplistic two-body problem
    p_squared=((2.*!pi)^2.)*(a^-3)/(const.G*M)

    ;write to a file
    openw, 2, '/home/jzalesky/log/planet_period.txt'
    printf, 2, p_squared
    close, 2
end
```

where now have output the variable p_squared instead of some type of string.

6.2 Plotting in IDL

Now that we can read and write data from files its time to do something useful with it. In any type of scientific endeavor it is important to create plots showing relationships between different variables. In astronomy there are many types of commonly used plots with needlessly complicated names like spectra (intensity vs. wavelength), rotation curve (velocity vs. radius), light curve (intensity vs. time), etc. Being able to make clear and concise plots is essential in order to illustrate your ideas.

> A paraphrased quote from Prof. James Graham, "People should be able to look at only your plots and captions and understand the central point of what you’re trying to say in your paper."

6.2.1 A Simple 2-D Plot

Making The Plot

Lets jump right in an make a simple X-Y scatter plot which contains nothing but a few points of data (we’ll keep it simple and use only 10 data points). In IDL (and almost all programming languages) 2-D plotting is done using two 1-D arrays where one array contains the x-values you’d like to plot and another array contains the corresponding y-values. For example we can declare two arrays,

```idl
IDL> x_vals=[2,5,6,10,18,19,21,22,24,29]
IDL> y_vals=[3,5,7,19,22,30,31,33,34,40]
```

which are filled with our data (these numbers could mean anything). Your intuition should tell you there is a simple plot procedure and there is! If we use plot with our data,

```idl
IDL> plot, x_val, y_val
```

Then we get the output shown in Figure 6.2A. There are a few notable problems; one is that IDL has interpolated when we didn’t want it to and the other is that there are no labels on the axes or title for the plot. We can easily correct these by specifying a few optional inputs,
Chapter 6. Data Analysis and Plotting

Figure 6.2: (Left) Figure 6.2A: The simplest Plot. (Right) Figure 6.2B: A Little bit better.

IDL> plot, x_val, y_val, psym=1, title='A Sample Plot', xtitle='X-Axis [Units]', ytitle='Y-Axis [Units]
and we get the output shown in Figure 6.2B where now the axes and plot are labeled and we’ve used a psym, short for “plotting symbol”, to remove the interpolation.

While plotting this data is great and gives us a visual way to interpret it, in science we often want to extract some type of trend from our data. This data looks somewhat linear so it may be useful to overplot a linear fit which is nothing more than a line whose attributes are derived from the data. The Error Analysis and Fitting chapter will detail how to calculate such fits but for now just know that we’re going to want to overplot the resulting fit on top of our data. Say we’ve stored our x and y values of our fit into x_fit and y_fit arrays respectively. We can overplot this fit easily enough with oplot.

IDL> plot, x_val, y_val, psym=1, title='A Sample Plot', xtitle='X-Axis [Units]', ytitle='Y-Axis [Units]
IDL> oplot, x_fit, y_fit
and the output of this is shown in Figure 6.3A. It may also be helpful to add some color to this plot to more easily differentiate the data from the fit and change the font size to make the plot readable. I’m going to switch to a script since this can get annoying to retype,
6.2 Plotting in IDL

Figure 6.3: (Left) Figure 6.3A: Over plotted a fit. (Right) Figure 6.3B: A very nice plot.

```
pro plot_test
  ;define data
  x_vals=[2,5,6,10,18,19,21,22,24,29]
  y_vals=[3,5,7,19,22,30,31,33,34,40]
  
  ;code to generate x_fit and y_fit
  
  ;plotting
  loadct, 0 ;load b-w colortable
  plot, x_vals, y_vals, psym=1, title='A Sample Plot', xtitle='X-Axis [Units]', ytitle='Y-Axis [Units]', charsize=1.5, color=255, background=0, /nodata
  loadct, 13 ;load rainbow
  oplot, x_vals, y_vals, psym=1, color=100 ;plot data
  oplot, x_vfit, y_fit, color=200 ;plot fit
end
```

The resulting plot is shown in Figure 6.3B but lets discuss what happened here. To incorporate color, IDL uses things called colortables which can be loaded using loadct where every colortable has a unique number; 0 is black and white, 13 is rainbow, etc. We've provided a link to the list of all colortables on the class website. First we loaded in the black-white colortable with the first loadct and then specified that the axes, labels, and title should all be white with a black background.
using /nodata and a font size of charsize=1.5. We then loaded in the rainbow colortable and overplotted our data in blue and our fit in gold (I actually didn’t do that on purpose, weird).

**Saving the Plot**

So now we have a fancy little plot but its stuck in the plotting window. How do we save it so we can send it, import it into our paper, etc? We can create a PostScript file using psopen and psclose much in the same way we used openw and close. All we need to do is to place our plotting code between these statements as follows,

```idl
pro plot_test
  ;define data
  x_vals=[2,5,6,10,18,19,21,22,24,29]
  y_vals=[3,5,7,19,22,30,33,34,40]
  ;code to generate x_fit and y_fit
  ;open .eps file
  psopen, 'first_plot.eps', /encapsulated, /color,
  ;plotting
  loadct, 0 ;load b-w colortable
  plot, x_vals, y_vals, psym=1, title='A Sample Plot', xtitle='X-Axis [Units]', $
    ytitle='Y-Axis [Units]', charsize=1.5, color=255, background=0, /nodata
  loadct, 13 ;load rainbow
  oplot, x_vals, y_vals, psym=1, color=100 ;plot data
  oplot, x_vfit, y_fit, color=200 ;plot fit
  ;close .eps file
  psclose
end
```

where the /encapsulated, and /color keywords simply keep the formatting that we’ve already set in our plot and oplot procedures. Note that psopen has a few other keywords but the two used above should be enough to solve most issues. If you run this code there should be a new file called first_plot.eps in your working directory. If you’d like to change the formatting to a .pdf you can just use the bash command available on ugastro called ps2pdf and can combine it with spawn to automate the process.

### 6.2.2 A More Complicated Plot

While a lot of scientific analysis does consist of looking at plots like the 2-D example shown above there are also many other ways to visually show data including histograms, contour plots, 3-D maps of simulations, etc. IDL supports many ways to plot things and we’ll go through an example of making a contour plot. A visual explanation of a contour plot is shown in Figure 6.4. In the most general sense, a contour plot is a graphical technique for representing a 3-D surface by plotting constant z slices, called contours, on a 2-D format. That is, given a value for z, lines are drawn for connecting the (x,y) coordinates where that z value occurs. As you can see it is much quicker to
interpret a contour plot than it is a 3-D surface and this is why contour plots are used so often in a wide variety of scientific fields.

So how do we get IDL to make these plots? First I’m going to pull some example data from *A Coyote’s Guide to IDL* using his predefined `cgDemoData` function (google about it if you want to know more) and load it into an IDL session;
Figure 6.5: The most basic contour.

IDL> data=cgdemodata(2)
IDL> help, data
D2    FLOAT = Array[41, 41]

and as you can see this is just a 2D array. How will this help us create a contour plot? Think of it as the x and y values being specified by the indexes in the array and the z values being specified by the values of each element. We can use the rather intuitive contour procedure to help us (I’m again switching to a script),

```
pro plot_contour
 ;example of a contour plot

data=cgdemodata(2) ;load in data

contour, data
end
```

and we get the output shown in Figure 6.5. What you’re looking at are a few contour lines specifying regions that have higher and lower z-values (Re: higher and lower valued elements contained within the 2-D array). Since our data array was 41x41 (look at when I called help, data) then our plot goes to 40 in both the x and y axis. Why not 41? Try to figure it out; you should be able to by now. Some problems with this plot: we have no idea what actual values these curves correspond to, there are no axis labels, and its pretty bland being all white and black. We’ll first add labels to the axes, a title, and increase the number of visible curves,
6.2 Plotting in IDL

Figure 6.6: Throw in some labels.

```
pro plot_contour
   ;example of a contour plot
   data=cgdemodata(2); load in data
   loadct, 0
   contour, data, xtitle='X-Values [Units]', ytitle='Y-Values [Units]', title='A Sample Contour Plot', charsize=1.5, nlevels=10, c_labels=replicate(1,12)
end
```

where nlevels has set the number of visible levels and c_labels controls the numerical labels on each curve (don’t worry too much about how that works unless you’re super interested). The output is shown in Figure 6.6. Now we could read these tiny numerical labels or we could replace them with color and add a colorbar to assign each color to a numerical value. This may seem needlessly complicated but it makes reading your figure both much easier conceptually and much more interesting to look at. To change the colors up we must edit our script,

```
pro plot_contour
   ;example of a contour plot
   data=cgdemodata(2); load in data

   ;color axes black and white
   loadct, 0
```
Figure 6.7: Added some color.

```
contour, data, xtitle='X-Values [Units]', ytitle='Y-Values [Units]', title='A Sample Contour Plot', charsize=1.5, nlevels=10, /nodata

;color plot with rainbow
loadct, 13, ncolors=13
contour, data, xtitle='X-Values [Units]', ytitle='Y-Values [Units]', title='A Sample Contour Plot', charsize=1.5, nlevels=10,
c_colors=reverse(indgen(13)+1), /overplot
```

and the output is shown in Figure 6.7. This is very familiar to the way we added color to our 2D plot. One notable difference is that in calling loadct 13 (the rainbow) I scaled the colortable using ncolors=13 so our values would correspond to noticeably different colors. Also notice that in the second call of contour I used c_colors=reverse(indgen(13)+1) which sets the contour to match our loadct scale and reverses the color so red corresponds to larger values and blue corresponds to smaller values. 

Finally we’ll want to add a colorbar onto our plot so we can tell what colors map to what numerical values,
6.3 Summary

Reading in data into IDL and displaying it is straightforward, in most instances, once you understand and become accustomed to the syntax. Hopefully now you see the reason for well-documented and consistent record keeping both in code and in stored data as it makes manipulation of both much, much easier. We’ll see this trend continue as we discuss Images in the next chapter.
Figure 6.8: A detailed contour plot.
Strings are one of the building blocks of any programming language and as such have a wide variety of applications. However in the context of astronomy they are most often used to specify names of paths, data files, variables, etc. It is therefore beneficial to know how to manipulate strings beyond simple concatenation and we detail how IDL handles this in this chapter. In addition we finally get to play with the mythical FITS files. We’ll quickly come to learn that the reason FITS files are so ubiquitously used in astronomy is due to the way they store the type of information we need.

7.1 Advanced String Manipulation

In Chapter 3 we learned the basics of strings: how to create and concatenate them. Here we’ll learn some of IDL’s prewritten functions that allow us to further manipulate strings.

The first function we will look at is strpos. As in most cases, the best way to understand this function is through an example. Suppose you have the string 'This is bat country' and you want to know the position of the word 'bat'. In IDL you would type

IDL> print, strpos('This is bat country', 'bat')
8

Here you can see that IDL returns the value 8 for the position of 'bat' (you’re welcome to check this by counting, if you’d like). If the string you’re looking for is not in your larger string, then strpos will return -1. For example:

IDL> print, strpos('This is bat country', 'hat')
-1

Now, let’s go in the opposite direction. Suppose you want to know the elements of a string at a certain position. Again, consider the string 'This is bat country' and let’s say we want to know the elements from the 8th position to the end of the string. Using the function strmid, we can find what we’re looking for.
Chapter 7. Strings and FITS Files

IDL> print, strmid('This is bat country', 8)
bat country

Here we gave `strmid()` a string and told it give us all the elements from the 8th position to the end of the string. We know from using `strpos` that 'bat' starts at the 8th position, so we could have guessed that the output of `strmid` would be 'bat country'. If we wanted to further refine our search, we could add a third argument to `strmid`. The third argument, an integer, tells `strmid()` how many elements to include. For example:

IDL> print, strmid('This is bat country', 8, 3)
bat

In this case, we told `strmid` to start at the 8th position in 'This is bat country' but to only count three elements, which gives us bat.

The next function we will discuss is `repstr`. As its name suggests, `repstr` is used to replace substrings in a larger string. For example, if you wanted to replace 'bat' with 'cat'

IDL> print, repstr('This is bat country', 'bat', 'cat')
This is cat country

Be weary of spaces in your strings; sometimes you will have to include spaces in your substrings in order to create the string you want. Imagine you are no longer in bat country and want your string to read 'This is not bat country'. Your initial thought might be to use `repstr`, and you would be right. However, you have to be careful with how you write your substrings and adding spaces where they are needed. Without considering spaces

IDL> print, repstr('This is bat country', 'is', 'is not')
This not is not bat country

This is obviously not what you wanted. What’s important to notice is that 'This' also contains the string 'is' and that `repstr` doesn’t know the difference. Now, if you consider the spaces, you can fix this discrepancy.

IDL> print, repstr('This is bat country', ' is', ' is not')
This is not bat country

In this case you got exactly what you wanted by considering the spaces included in the substrings. This is important to keep in mind when the substring you want to replace is also found in other places.

Lastly, we will be introducing the function `strmatch`. This function takes two strings as inputs and returns a one if the second string appears in the first and a zero if it doesn’t. For example:

IDL> print, strmatch('This is bat country', 'bat')
0

This result is probably not what you expected, but it highlights subtleties of `strmatch` and how to get around them. What we told `strmatch` to look for was simply the string 'bat', nothing else. However, in the string 'This is bat country', the substring 'bat' appears surrounded by other characters; namely, the rest of the string. In order to get around this, we have to use some wildcards that were introduced many chapters ago. For this example, we want the * wildcard. As you may recall, * refers to just about anything. In this case, we’ll use it as follows:
IDL> print, strmatch('This is bat country', '*bat*')
   1

This time IDL returned 1 because we used the * wildcard to represent anything that came before or after 'bat'. There is one more wildcard that we can use. A ? is used to represent any single character. Once again (last time, I promise), let’s look at 'This is bat country':

IDL> print, strmatch('This is bat country', '*b?t*')
   1

As you can see, the ? acted as a wild card for the a character.

7.2 FITS Files

Flexible Image Transport System (FITS) Files, as the name suggests, are used to transfer astronomical images between computers and serve as the main storage mechanism of modern astronomical data. Astronomers no longer do their job by looking through an eyepiece and hand writing their observations. Astronomy has become a highly precise science and cannot afford the threat of user error associated with visual interpretations and hand-recording of data.

Okay but why do we need these special FITS Files; wouldn’t it be easier to use standard ASCII or binary if we’re looking to save memory? Not really. To see why let’s think about how we go from the light of an astrophysical object to FITS Files.

Photons, a.k.a light, from some astrophysical source reach the telescope and are focused, most often, by a set of mirrors. The focused light then hits a panel containing a 2-D grid of pixels which are made of Charge-Coupled Devices or CCDs. An example of a CCD array is shown in 7.1. For

Figure 7.1: The CCD Array aboard the Kepler Space Telescope. Note that each blue pannel contains many pixels.

now (Re: until you take a lab course) you can think of a CCD as something that counts and records how many photons have hit a pixel. After some specified time the pixel array is ”read out” and the information is stored on a computer. At the same time, the observatory is recording things like windspeed, temperature, the exact local time, weather conditions, etc.

To store all of this information, both the measurements from the telescope (often confusingly referred to as ”the data”) and notes from the observatory (often confusingly called the header) we use FITS Files because they contain a convenient way to store and access both types of information.
Chapter 7. Strings and FITS Files

7.2.1 Reading in FITS Files

Without IDL

Sometimes its useful to simply glance at what is in the header or what the image looks like. The quick and dirty way to do this is to use some software called ds9 which provides a nice little GUI to play around with an image. For instance if we wanted to look at an image I had saved from optical lab called Idaphne101713_108.fits I can simply enter into Bash,

```
joseph@joseph:~$ /home/apps/ds9/ds9 Idaphne101713_108.fits
```

and we open up ds9 as shown in Figure 7.2. You can open up the header using DS9’s GUI or the better way to do it is to pipe Bash’s fold command into more as shown below,

```
joseph@joseph:~$ fold Idaphne101713_108.fits | more
```

Figure 7.2: ds9’s GUI interface and an image of an asteroid and background stars. Can you pick out which one is the asteroid?

SIMPLE = T / conforms to FITS standard
BITPIX = -32 / array data type
NAXIS = 2 / number of array dimensions
NAXIS1 = 1056
NAXIS2 = 1024
CRVAL1U = 2046 / COLUMN ORIGIN
CRVAL2U = 2046 / ROW ORIGIN
CDELT1U = -2 / COLUMN CHANGE PER PIXEL
CDELT2U = -2 / ROW CHANGE PER PIXEL
OBSNUM = 108 / OBSERVATION NUMBER
EXPTIME = 40.000000 / Exp time (not counting shutter error)
As you can see the header basically stored information into keywords such as SIMPLE, BITPIX, etc. and each keyword has a comment by it (Re: the text following the /) to remind people what this information means. We’ll get more into the header as we play with images in IDL.

With IDL
So how do we access all of this data stored in the FITS files using IDL? Most image processing revolves around a little function called mrdfits which is pronounced Mister Dee Fits. Let’s load in Idaphne1017_108.fits into IDL.

IDL> img=mrdfits('Idaphne1017_108.fits', 0, hdr)
IDL> help, img
IMG FLOAT = Array[1056, 1024]
IDL> help, hdr
HDR STRING = Array[91]
IDL> print, hdr
SIMPLE = T / conforms to FITS standard
BITPIX = -32 / array data type
NAXIS = 2 / number of array dimensions
NAXIS1 = 1056
NAXIS2 = 1024

As you can see, img contains the information from the telescope as stored into a 2-D array. In this array the value of each element loosely corresponds to the intensity of light at each pixel in the CCD array. Why “loosely” and not “directly”? As you may have noticed from Figure 7.2 there often malfunctions in the instrumentation in the telescope like the column that is all white. A large part of image processing involves making adjustments to the image to correct for instrumental factors that influence your data. We will discuss one problem called dark current and how to correct for it in a moment.

We’ve also loaded our header as a string into hdr. If you’re interested in why we must specify 0, and there is a good reason, come talk to us outside of class as its somewhat of an aside and takes too much space to write out here. If you’re not too interested just know to always put 0 when calling mrdfits.

7.2.2 Playing with the Image and Header
The Header
First let’s pull some information from the header to look at it. We could write a complicated string manipulation function to pull out information from specific keywords but someone has already written that for us. For instance lets say we want the Right Ascension (RA) and Declination (DEC) of this observation. For those of you new to astronomy these two numbers will specify a specific point in the sky for a given time and location on Earth (Re: google or talk to us for more info). To load these from the header we just use the fxpar function,

IDL> ra=fxpar(hdr,'ra')
IDL> dec=fxpar(hdr,'dec')
IDL> print, ra, dec
19:36:29.5   -09:58:05.0

and we get the locations we desire. We can pull out any arbitrary keyword using this simple yet efficient function.
The Image: Dark Current

We can also play with the image data in the exact same way we made edits to 2-D arrays like we did in the array chapter. To detail this let’s look at a specific example you’ll have to do later if you take the optical lab course.

One correction you’ll have to make to your data is the correction for dark current. As discussed before, CCDs can be thought of as photo-sensitive plates that count photons and store the information using electricity. One problem with doing this is that the flow of electric charge through any device can be influenced by changing the temperature of the device. Random fluctuations of temperature can cause random fluctuations in the flow of electricity. We call this electrical charge that is owed to temperature fluctuations dark current as it happens even when we have the CCD array in a completely dark environment.

To correct for this effect we usually take a dark frame with the CCD isolated from any light source. From there we subtract this dark frame from our raw image to get an image that does not depend as heavily upon temperature variations at the telescope. For instance if we have a dark frame stored into a file called dark101713_01.fits we can load that into IDL and subtract it off from our raw data;

```
IDL> dark_frame=mrdfits('dark101713_01.fits')
IDL> science_img=img-dark_frame
```

where now science_img is the frame we would use in any future analysis. Note that dark current is only one possible source of error in an image and there are many more, and better, corrections to be done in more complicated image analysis.

7.2.3 Displaying Images

Our image is now loaded into IDL and we may want to look at the image from time to time as we make our edits. To look at our image we’ll use the display procedure. for instance we can call,

```
IDL> display, science_img, title='A Sample Image', charsize=1.5
```

and we get the output shown in Figure 7.3. However you can see that this image is somewhat dim and its hard to see the stars and the asteroid. To solve this we can modify the color scale to see different details in the image. The color scale is an important but subtle point. Each element of science_img contains a specific numerical value. For instance I can call a random element,

```
IDL> print, science_img[155,244]
542.038
```

and this value and the value of every other element will not change as we change the color scale. When modifying the color scale we are changing the way IDL maps from values in science_img to values in the color table and not the values in science_img itself. For instance we can set display so that any element at or below median(science_img)/10 (Re: $\frac{1}{10}$ of the median of the image) will be completely black and any element above 10000 will be completely white with a linear scale in between those values,

```
IDL> display, img, title='A Sample Image', charsize=1.5, max=10000, min=median(img)/10
```

and we get the output shown in Figure 7.4.

We can now see more stars and instrumental effects in our image using this color scale. What is the
Figure 7.3: A Sample Image.

Figure 7.4: A Sample Image with better scaling.
"right" color scale to use? That isn’t a good question since what is "right" depends on what you’re looking for. Here we’re looking for all the possible astronomical sources (Re: background stars) in our image so making the color scale "brighter" made sense. However if you ever look at more complex objects (circumstellar disks, nebula, galaxies, etc.) you can imagine wanting to look at various different settings to see different types of structure within the image. What we’ve detailed here is a simple example of using only the display procedure but if you combine this with contour using the overplot keyword then you can make some pretty interesting figures.

7.2.4 Writing to FITS Files

To close up the chapter lets take our defined science_img and our hdr and write them to a brand new FITS file so we don’t have to do the dark subtraction every time we start up a new IDL session. To easily accomplish this we use the mwrfits procedure and input the data as follows;

IDL> mwrfits, science_img, 'Idaphne_darksub_101713_108.fits', hdr

and now science_img along with hdr has been saved under the Idaphne_darksub_101713_108.fits file.

However what if we wanted to add a new parameter to the header? We can do this with the sxaddpar procedure. For instance if we wanted to create a flag (Re: label) called gb_flag stating if this specific image is good or bad for a specific type of analysis we just run,

IDL> sxaddpar, hdr, 'g_b_flag', 'good'

and now we can call our newly defined g_b_flag information with fxpar,

IDL> print, fxpar(hdr, 'g_b_flag')
good

But if we decide that flag is useless later down the line we can just remove it with sxdelpar,

IDL> sxdelpar, hdr, 'g_b_flag'

IDL> print, fxpar(hdr, 'g_b_flag')
0

and you can see that the flag has been removed. If print, hdr then you won’t be able to find the g_b_flag keyword. Keep in mind that there is more advanced manipulation tricks than shown here but these are the basics that should solve most of your problems.
Way back when we were discussing arrays I told you that arrays are confined to using a single datatype. If you want to store multiple datatypes within a single object the best way to do that in IDL is to use things called data structures. In this chapter we introduce you to data structures, their basic syntax, and provide an example of them working in action. In addition we introduce you to a statistical modeling method used in a variety of aspects of astronomy called Markov Chain Monte Carlo or MCMC for short.

8.1 Data Structures

As the name suggests, data structures provide a convenient way to organize different types of data. If you have a bunch of related variables, keeping them organized with a matrix or a series of lists can be cumbersome and confusing. Data structures give you a way to group associated variables together and access them easily. Additionally, data structures allow you to keep different data types together – a feature not available when using lists or arrays. Suppose you have a group of people and you want to keep track of their traits (name, age, height, etc.). One way to do this is by making a different array for each characteristic and remembering that the first element in each array gives the trait of the first person and so on. However, this is cumbersome and can be done more efficiently with structures.

8.1.1 Creating Data Structures

Let’s say you wanted to create a structure to contain information on person1.

IDL> person1 = {name:'Bob', age:10d, height:'2 meters'}

Here we created a structure for our 6 foot tall 10 year old in order to hold all of his information. We use the curly brackets { and } to indicate the beginning and end of the structure. In between the brackets are the tags and their values. Each tag-value pair is separated from the next by a comma.
In the example above, \texttt{name}, \texttt{age}, and \texttt{height} are the tags; \texttt{’Bob’}, \texttt{10d}, and \texttt{’2 meters’} are the values. What is the purpose of the \texttt{d} in \texttt{10d}? It tells IDL that we want the value for \texttt{age} to be a double. Without the \texttt{d}, the data type is an integer by default. Alternatively, we could have said \texttt{age:10.0}, which would have set the data type as a float.

As you can see, this was relatively simple and gives us an easy way to arrange different variables into a single package. Next we’ll look at how to access elements in a structure and how to add tags.

\textbf{8.1.2 Playing with Structures}

First off, when looking at a structure, it’s always good to know what the tags are. To see the tags, simply type

\begin{verbatim}
IDL> help, person1
** Structure <1107b78>, 3 tags, length=40, data length=16, refs=1:
   NAME STRING ’Bob’
   AGE DOUBLE 10.000000
   HEIGHT STRING ’2 meters’
\end{verbatim}

Luckily for you, IDL displays tags, values, and other useful bits of information such as the number of tags and the length of the structure. The series of numbers and letters, \texttt{<1107b78>}, identifies the structure in the way that a license plate identifies a vehicle. Like a license plate, we can customize this identification to create a \texttt{named} structure. The structure above is an \texttt{anonymous} structure because we didn’t give it a name when we created it. Named vs anonymous structures aren’t of primary concern right now, though, so we’ll continue.

Another useful way to access a structure’s tags is by using the function \texttt{tag_names} which allows us to create a string array whose elements are the tags contained in our structure. For example:

\begin{verbatim}
IDL> x = tag_names(person1)
IDL> print, x
NAME AGE HEIGHT
IDL> print, x[0]
NAME
\end{verbatim}

This is nice because it allows us to quickly extract the tags for future use.

Now, to see the usefulness of tags:

\begin{verbatim}
IDL> print, person1.name
Bob
IDL> print, person1.age
10.000000
IDL> print, person1.height
2 meters
\end{verbatim}

The tags provide a simple way to organize and retrieve stored values. They can also be used in a similar way to change the values of a structure:

\begin{verbatim}
IDL> person1.name = ’steve’
IDL> person1.age = 23d
IDL> person1.height = ’1.5 meters’
\end{verbatim}
Here, you can see that are structure still has the same identification yet all the values have changed.

Next, suppose you want to add another trait to the first person's structure. Unfortunately, once created, the size of a structure is fixed. However, we can to use a function called `struct_addtags` to combine our old structure with a new structure containing our new tags and values. Let's make a structure called `new_struct` that contains the person's hair color and then add `new_struct` to `person1`.

```
IDL> new_struct = {hair_color:'brown'}
IDL> person1 = struct_addtags(person1, new_struct)
IDL> help, person1
** Structure <111d778>, 4 tags, length=56, data length=56, refs=1:
   NAME      STRING   'steve'
   AGE       DOUBLE   23.000000
   HEIGHT    STRING   '1.5 meters'
   HAIR_COLOR STRING   'brown'
```

The function `struct_addtags` takes two arguments: our old structure and the new structure that we want to add. You'll notice that the identification for `person1` has changed. Since the size of a structure is immutable, calling `struct_addtags` technically creates a third structure out of the two inputs. In this case, however, we still called our structure `person1`, because we want brown hair to be attributed to the first person.

Now we have a whole room full of people we need to keep track of. Instead of having forty structures, we can create a structure of structures *Inception horn*.

```
IDL> people = {one:person1, two:person2, three:person3, <and so on>}
```

Accessing values from this structure tree is actually quite simple. If you wanted the age of person two

```
IDL> print, people.two.age
   25.000000
```

and so on for other attributes. Above we talked about anonymous vs named structures; we can see an example of the difference between the two when we use structure trees. We'll start by creating two new structures and nesting them in a single, larger structure.

```
IDL> a = {planet1, mass:1.23, radius:0.95}
IDL> b = {mass: 2.1, radius:1.4}
IDL> planets = {one:a, two:b}
```

Structure `a` was named as evident by the first element, `planet1`. But, shoot, we forgot to name our second structure. Let's see what happens when we look at the larger structure, `planets`:
** Chapter 8. Data Structures and MCMC **

IDL> help, planets
** Structure <812c28>, 2 tags, length=16, data length=16, refs=1:
    ONE STRUCT -> PLANET1 Array[1]
    TWO STRUCT -> <Anonymous> Array[1]

Our second substructure is <Anonymous> because we never gave it a name! For structure trees it’s easy to see why naming your substructures would be incredibly useful.

** 8.2 Markov Chain Monte Carlo: MCMC **

What I’d like to accomplish in this section is for you to walk away with a general understanding of what MCMC’s are and how they apply to astronomy rather than a memorization of some mathematical definition that you could look up. Let’s start with the elephant in the room; what is an MCMC? If you’re like me your first instinct is to google this. Wikipedia’s first sentence on the topic says that; In statistics, Markov Chain Monte Carlo (MCMC) methods are a class of algorithms for sampling from a probability distribution based on constructing a Markov chain that has the desired distribution as its equilibrium distribution. Which may be helpful to those of you who know some statistics but to me was a bunch of gibberish when I first read it. Let’s break up MCMC and take it one step at a time as it can be difficult to understand without a background in probability or statistics. The first subsection is really just to get you acquainted with terminology. After developing this vocabulary we will delve into what MCMC really does.

** 8.2.1 Humble Beginnings: Probability Density Functions **

One term you may not have known the meaning of within that Wikipedia definition of MCMC is probability distribution also called a probability density function or pdf for short. However I almost guarantee you know what these are since they govern almost all of your grades here at UC Berkeley. An example of a grade pdf is given in Figure 8.1. The pdf is the curve itself and the shape of this particular curve is called a Normal distribution.

![Figure 8.1: A Normal Distribution](image)

The x-axis is the grade and the y-axis counts how many people received that grade. Now if I ask you, What is the average grade? often represented by µ, you should see that the answer is 70 as it is the peak and center of the symmetric pdf.

Why is 70 the mean? An equivalent way to ask this question is If I pick a random student, which grade has the highest probability of being that student’s grade? Looked at in this manner, the curve
that describes our grade distribution really just describes how the probability density of getting a certain grade by random sampling changes as a function of the grade. Therefore the grade with the highest probability (Re: the x value that corresponds to the highest y value on the curve) is our mean value as it lies at the center peak of the symmetric curve.

I can give you many more distributions with the same average, a.k.a mean, value of 70 as shown in Figure 8.2 that look nothing like Figure 8.1.

![Figure 8.2: Many Normal distributions with the same mean value.](image)

We need another parameter to describe any Normal grade distribution uniquely. There exists a nice one that other people have defined and found useful called the standard deviation, often denoted by $\sigma$. The definition of standard deviation is illustrated in Figure ??.

![Figure 8.3: $\sigma$ for a Normal distribution.](image)

As you can see, $\sigma$ is a measure of the deviation from the mean value set at a standard that other people have predefined. In the case of a our Normal grade distribution, $\sigma$ is the length from the mean value such that the area underneath the curve and between the mean and 1 $\sigma$ from the mean comprises 34.1% of the total area underneath the curve. This definition is not arbitrary and we will see why in Chapter 9. For now just understand that we can describe any Normal pdf given $\mu$ and $\sigma$.

### 8.2.2 The Random Walk Problem & Markov Chains

Okay so now we know what a probability distribution is, how does this relate to MCMC? First we’ll look at the Markov chains half of MCMC. To understand Markov chains lets discuss a
real world example. Imagine you’re hiking in the Berkeley Hills and you see a very inebriated man stumbling back and forth on the path. Let’s assume, somewhat offensively, that he is so far gone that after each step he takes, he loses his memory about who or where he is. He therefore stumbles back and forth on the path with no conscious rhyme or reason to his movements. In physics we call the problem of trying to map out and predict his 3-D movement in the hills as a **random walk problem**. This drunkard is a prime example of a Markov chain. Wikipedia is a bit more helpful with its definition for this one,

A Markov Chain is any random process that undergoes transitions from one state to another... where the probability of being in the next state depends only on the current state and not on the sequence of events that preceded it.

This seems to describe our drunkard very well since he effectively has no memory and his next movement is based completely upon his current position. 

*Nice but I’m not seeing the connection to the probability distributions?*

This is the heart of MCMC so read carefully. Lets imagine in our drunkard example that the Berkeley Hills are a 3-D probability distribution where the latitude and longitude coordinates can be thought of as two parameter’s were interested in studying and the higher elevations can be thought of as higher probabilities. Now let’s ask a question; *how can we map the topology of the Berkeley Hills using only this drunkard?* Which is equivalent to asking, *How can I make a map of the probability distribution by wandering around it?*

To do this we’ll add in an extra rule our drunkard will follow; **the amount of time the drunkard spends standing still between steps will be proportional to his elevation.** So he may spend whole minutes between steps at spots like Vollmer Peak and may spend only a second between steps within the valleys. We’ll then ask the drunkard to record the time taken between all of his steps and the latitude and longitude where his steps occurred (the analogy is getting a bit far-fetched but bear with me). When he returns will have a nice contour map of the Berkeley Hills and our drunkard will be paid handsomely assuming he wasn’t eaten by a mountain lion.

What the drunkard did for us was **explore a probability distribution which was very complicated and hard to map out through other means.** This is **exactly** what MCMC does; if we have a complicated probability distribution we can “MCMC it” and have a reasonable approximation of the topology of the system.

### Some Subtleties

You may have noticed some problems with my drunkard analogy. For instance;

1. If we wanted to map the Berkeley Hills wouldn’t it have been easier to fly a helicopter over or use a satellite to map the terrain?
2. If we really were restricted to using drunkards, why did we only use one instead of multiple?
3. Why did you say “reasonable approximation of the topology”? Isn’t the map complete after the drunkard is done wandering around?

Which are great questions. Let me clear each one up,

1. For my analogy yes, it would have been **much** easier to ask someone with a helicopter to take pictures to create a map. However you should notice that for the Berkeley Hills I said it is a
good analogy to exploring two parameters. What if we wanted to explore more than two? Say three, five, or ten? Having N parameters would be equivalent to exploring a N-Dimensional map. So yes we could "fly over" our two parameter example but we have no way to "fly over" three, five, or ten dimensions because we live in a 3-D universe.

2. Ah you’re catching on! Most MCMC codes do use multiple "drunkards" (in most cases they’re called walkers) which explore any given probability distribution. The number of walkers ends up being limited by the number of available computing power you have at your disposal.

3. If our drunkard takes steps of 1ft in length we have a time corresponding to each 1ft interval but know nothing of the topology between these steps. Now in the Berkeley Hills its very reasonable to argue that the topology won’t change much between 1ft intervals (Re: there isn’t going to be a mountain that shoots up 500ft over 1 ft step) so it is reasonable to argue that we have a rather complete map of the Berkeley Hills. However this argument cannot be extended to an arbitrary probability distribution because we have no idea of the behavior of an arbitrary probability distribution (Re: it may be completely possible to have large peaks over small intervals). This is one of the major concerns when developing MCMC code; how big do you set your walker’s step size?

8.2.3 MCMC Summary

Hopefully this has given you an intuitive idea as to what MCMC is and does when applied in the correct situations. There is a lot of detail skipped here so if you’re interested in the topic search around for alternative explanations. However if you didn’t follow much of the discussion we will be discussing MCMC in lecture this week as well.
In previous chapters, tutorials, and homeworks we have been loosely referring to different ideas in probability and statistics that are used in astronomy. In this sections we hit some of the conceptual highlights that you, as an undergrad, should be aware of. In addition we finally discuss creating different types of fits from data and how to use IDL to do that. Much of my knowledge was gathered from two short (Re: less than 100 page) books titled, *An Introduction to Error Analysis* by John Taylor and *Statistical Data Analysis* by Glen Cowan. These are often on the shelves in the undergrad lab and I highly recommended reading these before the end of your time here as an undergrad.

### 9.1 What is Error?

You have probably been told in high school that the error of any given measurement is,

\[
\text{Percent Error} = \frac{(\text{Measured Value}) - (\text{Exact Value})}{(\text{Exact Value})} \tag{9.1}
\]

This is a very rudimentary way to estimate the error for any given measurement and it assumes that,

- You *know* the “Exact Value”
- There *is* an ”Exact Value”

which is generally not the case, especially within astronomy. For instance if I ask you to estimate the errors on your emission lines back in Tutorial 4 you can see that Equation 9.1 would be of no help.

A better way to describe error is the variation in a measurement. A good example is making a measurement with a ruler. When making a measurement with a ruler, the recorded measurement rarely falls exactly on a tick mark, however you can be sure that the measurement is between the two nearest tick marks. Thus the distance from the measurement to its nearest tick marks would
be a great way to state the error in the measurement. To report this measurement, you simply use the ± notation by stating measured value ± error in measurement. A numerical example using a typical ruler with millimeter markings would be 5.25 ± 0.05 cm for a measurement that was halfway between the 5.2 cm and 5.3 cm ticks. A good way to quantize the variation in a measurement, often called the variance, is the square of the standard deviation. The square is simply to remove any negative signs from the standard deviations. This type of error is often called "read error" and is a type of random error, which you will learn about later in this section.

Another good way to describe an error is the deviation of points from a fit. If you generate a fit, which we will discuss later in this chapter, then to quantify the error of that fit, you can use the average deviation of the points to the fit.

9.1.1 Random vs. Systematic error

There are two main types of error in any scientific measurement: random and systematic error. Random error, also known as statistical error, is error that occurs whenever you attempt to measure an exact value. It is very unlikely that you would get the exact number even if you took thousands of measurements; you would almost always get a number slightly higher or slightly lower than the expected value. This spread of numbers is what makes up random error. Statistical error can generally be identified by its,

- Inevitability - it will be present in all data
- Lack of preferred direction - the distribution of the measurements should be symmetric about the expected value
- Variability with each datapoint - no two data points will have the same statistical error
- Reduction with more data - the more points you take the smaller the uncertainty in your measurement which we will discuss in the next section
- Similarity to a Gaussian distribution (Bell curve) - the spread should be similar to an ideal grading curve

The second type of error, systematic error, occurs due to errors in your system of measurement. An extreme example would be if your were trying to measure the voltage at different points in a circuit and your volt meter was unknowingly mis-calibrated, always giving a value that was several volts higher than the actual measurement. Thus your systematic error in the measurements would be several volts, which could be bad if you are trying to measure small, low power circuits. Systematic error can generally be characterized by its,

- Ability to be minimized - though there will probably always be some systematic error in an experiment, it should be made much smaller than the random error of the measurement
- Unidirectionality - the error will all be in the same direction, consistently higher or lower than the expected value
- Consistency in each measurement - though the error might not be exactly the same, it will be close to the same shift
- Lack of reduction with more data - if you take more data, the error will remain constant
- Lack of similarity to a Gaussian distribution - a shifted Gaussian might be a better fit
9.2 Basic Statistics

A good example of where you could find both of these errors is in a stretched ruler. The user approximates which tick mark is closest to the true measurement, resulting in a random error as they are equally likely to overestimate and underestimate the measurement. However, because the tick marks themselves are stretched, the measurement will always be an overestimate resulting in a systematic error.

9.2 Basic Statistics

The basics of all error statistics are the mean and standard deviation of a set of data. Remember from Chapter 8 that the mean value, \( \mu \), describes the center of a Gaussian distribution and the standard deviation, \( \sigma \), describes the width from the center at which 34.1% of the total area is underneath the curve. If you take many measurements of the same value, you would expect the results to resemble a Gaussian distribution with most values being near a central value, with some outliers further away. How would you extract a single value as the "measured" value and quantify the error in such a set of data? Just like professors set their grading distribution to centered around the grade they want to give the most of, we can use the reverse to extract the most expected value by taking the mean of the sample. The error of the sample can be found my taking the standard deviation of the set. Thus your reported measurement would look something like \( \mu \pm \sigma \).

9.3 Reducing Error

One of the best and most basic fitting techniques is called the \( \chi^2 \) method. The basic idea is to find the fit of the data that minimizes the deviation of each point, or distance between the point and the fit. For simplicity, the distance is typically calculated from the difference in y values. The \( \chi^2 \) value that needs to be calculated for this type of fitting is

\[
\chi^2 = \sum_{i=1}^{N} \frac{(y_i - y_{fit}(x_i))^2}{\sigma^2}
\]  

(9.2)

where \( y_i \) is the y value of the datapoint, \( \sigma \) is the standard deviation of the data, and \( y_{fit}(x_i) \) is the y value of the fit at the x value of the point, \( x_i \). For example, the \( \chi^2 \) value for a linear fit is

\[
\chi^2 = \sum_{i=1}^{N} \frac{(y_i - mx_i + b)^2}{\sigma^2}
\]  

(9.3)

where \( m \) and \( b \) are the fitting parameters for the fit of \( y = mx + b \). A great visualization of \( \chi^2 \) can be seen in Figure 9.1. From the visualization, it is easy to see why minimizing \( \chi^2 \) would produce a better fit because the lower \( \chi^2 \) means that the fit is the closest to the points as it can be. \( \chi^2 \) is also related to a "goodness of fit" variable, usually denoted by \( r \), which is best (closest to 1) when \( \chi^2 \) is minimized.

This is not the only way to fit data, but it is a fairly easy way that is often used.

9.4 Using IDL in Error Analysis

So how do we, that is students of an IDL course, use IDL to reduce \( \chi^2 \) for a given dataset? That is the subject of this week’s tutorial so you’ll get a full run through there. In this text we describe how
to use linear algebra (Re: concepts from Math 54) within IDL so that we can use these concepts to quickly compute $\chi^2$ reduced fits for a given dataset.

### 9.4.1 Arrays are Matrices

Since most of you probably haven’t taken Math 54 yet let’s flesh out the main ideas that we care about from the course and link them to using arrays within IDL.

Right so let’s get right to it with some basic matrix notation. At this point the easiest way to think of a N-dimensional matrix is to imagine an N-dimensional array. For instance I can express a set of data using matrix notation,

$$a = \begin{pmatrix} 5 & 7 \\ 2 & 6 \end{pmatrix}$$

(9.4)

or I can express the same set using IDL’s arrays,

IDL> a=[[5,7],[2,6]]

IDL> print, a

5  7
2  6

and you can see they look the exact same. In the tutorial we will actually use IDL’s arrays as matrices when doing some calculations so this is a useful analogy to keep in mind.

Let’s explore matrices with a relevant example. You all know the equation for a straight line,

$$y = mx + b$$

(9.5)

where $m$ is the slope and $b$ is the y-intercept. Now let’s imagine we have a set of 3 data points. We could describe each point with the following equations,

$$y_1 = mx_1 + b$$

(9.6)

$$y_2 = mx_2 + b$$

(9.7)

$$y_3 = mx_3 + b$$

(9.8)
however this seems annoying to do and you can imagine its even worse for large dataset. You
already know that we could represent the same information using 1-dimensional arrays (the correct
mathematical term is *vector* but it’s the same thing) as we usually do when plotting,

\[ y = mx + b \]  

(9.9)

where,

\[
y = \begin{pmatrix}
y_1 \\
y_2 \\
y_3 
\end{pmatrix} \tag{9.10}
\]

\[
x = \begin{pmatrix}
x_1 \\
x_2 \\
x_3 
\end{pmatrix} \tag{9.11}
\]

However we could also use matrices to express this equation,

\[ Y = X \cdot A \]  

(9.12)

where,

\[
A = \begin{pmatrix}
m \\
b 
\end{pmatrix} \tag{9.13}
\]

\[
X = \begin{pmatrix}
x_1 & 1 \\
x_2 & 1 \\
x_3 & 1 
\end{pmatrix} \tag{9.14}
\]

\[
Y = \begin{pmatrix}
y_1 \\
y_2 \\
y_3 
\end{pmatrix} \tag{9.15}
\]

We’ll get to why the last part with matrices is true in a moment but look at how much more efficient
in terms of computing using matrices would be compared to arrays. Look at Equation 9.9. Using
arrays we would have 4 different variables; two variables for \( x \) and \( y \) and two variables for \( m \) and \( b \)
giving us a total of four variables.

Now in contrast look at Equation 9.12. We would only use three variables for \( X \), \( Y \), and \( A \). Not that
much computing power is saved in this example of three data points but for larger datasets and more
complex equations it becomes a lot more efficient to use matrices.

### 9.4.2 Basics of Math 54

About \( \frac{3}{4} \)ths of Math 54 concerns Matrix notation (and will likely be the only thing you remember
from the course 6 months after taking it) so here we just detail the basics that are relevant to computing
\( \chi^2 \) linear reduction fits. The main idea behind all of this is that *matrices do not follow the same
algebraic rules as normal numbers, aka scalars*. If you’ve already taken Math 54 and remember
dot product, transpose, and inverse you can probably skip to the next section.

Let’s start by looking at Equation 9.12; how does equation work? As you may notice there is
a dot on the right-hand side of the equation. This dot says to multiply the matrices in one of the two
possible ways. That is to take the dot product of the two matrices (the other way is to take the cross product but we won’t use that idea here). How do you take the dot product of two matrices? The visual interpretation is to take the columns of the matrix on the right, turn it \(90^\circ\) counter-clockwise, multiply the matching elements of each row in the matrix on the left and sum them. A cartoonish but good visual example is provided at, https://www.mathsisfun.com/algebra/matrix-multiplying.html under the Multiplying a Matrix by Another Matrix section. Try to understand how Equation 9.12 is the same as Equation 9.9 before moving on but know we’ll cover it in lecture as well if you ask us to.

So now we understand dot product. Let’s discuss how to take the transpose of any matrix or vector. To take the transpose of a vector like,

\[
Z = \begin{pmatrix}
z_1 \\
z_2 \\
z_3
\end{pmatrix}
\]  

(9.16)

we simply "flip" it in the same way as the dot product and call it the "transpose of \(Z\),

\[
Z^T = \begin{pmatrix}
z_1 & z_2 & z_3
\end{pmatrix}
\]  

(9.17)

In addition if we want to take the transpose of some matrix like

\[
W = \begin{pmatrix}
w_{00} & w_{01} \\
w_{10} & w_{11}
\end{pmatrix}
\]  

(9.18)

then the resulting transpose is

\[
W^T = \begin{pmatrix}
w_{00} & w_{10} \\
w_{01} & w_{11}
\end{pmatrix}
\]  

(9.19)

where two of the elements have been flipped over the diagonal. Hopefully you can see from our discussion of dot products that in general and for any given matrix or vector \(B\) that

\[
B \cdot Z \neq B \cdot Z^T \neq Z^T \cdot B \neq Z \cdot B
\]  

(9.20)

which is to say that none of these are equal to each other due to the specific way dot products are calculated.

Finally we need to discuss the inverse of a matrix. For normal scalars,

\[
x^{-1} = \frac{1}{x}
\]  

(9.21)

however this is not true for matrices,

\[
B^{-1} \neq \frac{1}{B}
\]  

(9.22)

and instead the inverse of a matrix is actually rather complicated to compute. The inverse for a 2-D matrix is given in Equations 5 and 6 in this link, http://mathworld.wolfram.com/MatrixInverse.html and it’s far more complicated for N-D matrices. Worse yet some matrices are non-invertible which means you cannot mathematically take the inverse; it’s impossible. However for the matrices we’ll be using to compute fits we’ll do a special trick in the tutorial to ensure we can take the inverse.
9.5 Why is the Linear Solution so Important?

To finish off our book (hoo-rah!) I want to make sure you all understand why everyone in the sciences and more specifically astronomy are crazy over linear fits. As you can see calculating a linear $\chi^2$ reduced fit is rather easy and requires little computational power. As such it would be nice
if we could just calculate linear fits instead of fits with super complicated functions. However anyone in physics, chemistry, or biology will know that the world is highly non-linear and this can cause problems for scientists.

In a paraphrase of our new department head Eugene Chiang, "As you can see the curve is non-linear, or to use the technical term; evil."

While we can’t make the external world submit to our desire of linearity we can make our plots do this very easily.

Consider that I have some data and it seems to follow a $y = x^2$ distribution with a little bit of noise thrown in as shown in Figure 9.2. Now you might say that *There’s no way to fit a line to that.*

Look at the asymptote on that curve! While it’s true I can’t fit a line to this plot. I can easily scale the plot to be linear.

How? I set the axis to plot not $x$ but instead $x^2$ as shown in Figure 9.3. Now all I have to do is fit not for a linear solution in $x$ but rather a linear solution in $x^2$.

Now you see the power of a linear least-squares solution. If we can make a guess as to the shape of the curve we can easily avoid the pain of having to write non-linear fitting code. Be aware though that you will need to write such non-linear fitting code to solve problems within the lab course. I just wanted you to be aware of this method as it can save time and computing power.
9.6 You’re Gonna Carry That Weight.

This is the end of our textbook! You survived quite the ordeal if you’ve read through to this point. This is the first attempt at writing a textbook for all three of us so I hope we didn’t confuse you too badly. In the coming weeks we’ll be discussing some IDL tactics relevant to the Radio Lab course followed by LaTeX and a very short intro to python. After those topics all that remains is to make your final project which we should be giving you details on soon.

Also as a note I (Joe) will not be around next year to teach this course and Austin and Will are looking for some help. If you’re interested in teaching (and have been doing most of the work we ask of you) please let us know. The IDL version of this course is currently at risk for being taken over by python year-round and I would like to see it continued given how many people and large astronomical campaigns still use IDL. However that requires dedicated students to make it happen so if you’re interested please get involved.

Figure 9.3: A non-linear dataset shown in a linear plot.
Here we list definitions for words that may be unfamiliar to those with no programming experience. Definitions are ordered by their order of use in the text.

10.1 Chapter 1: Bash

UNIX
Operating system developed in the 70’s from which many other OS were created.

Bash
A way for humans to interact with the computer via text commands.

User-Error
A mistake made by the operator of the computer.

Terminal
A window where users can input Bash commands to be interpreted by the computer.

GUI
Stands for ‘graphical user interface’. It is also how most people are used to navigating a computer

Directory
Think folder. A place where different types of files, or more directories, are stored.

Working Directory
The directory you are in now.

Directory Tree
The path linking directories to sub-directories and so on.

Wildcard
A character that is used to take the place of a multitude of other characters. Example: *

Pipe
Provides a way for multiple commands to be ‘piped’ together. Uses the output of the command on the left of | as the input of the command on the right of |
**Chapter 10. Glossary**

A predefined sequence that modifies a specific command. Example: `emacs -nw file.txt`

**SSH**

Stands for ‘secure shell’. Provides a way for the user to initiate a Bash session remotely.

---

### 10.2 Chapter 2: Version Control & Git

**Version Control**

A unique type of software which allows for easy control of code and files through time.

**Git**

A special brand of version control that we use to submit assignments and collaborate. Pretty standard software to use, other one is SVN.

**Staging Area**

The pre-commit area where you can easily add/modify what you’d like to commit to your version control timeline.

**Repository**

Can be thought of as a ‘version controlled directory’. It really is just a directory that something like git is keeping tabs on.

**Commit**

The process of adding things to the timeline. You commit changes you’d like to save.

**Push**

The equivalent of ‘uploading’ things from a local repository to a remote repository. You push your code to Bitbucket when you complete an assignment.

**Branch**

The name is rather telling; you can create timelines that branch off of the original to test code without affecting the code that you are sure works.

**Arguments**

Optional input in almost any computer language ranging from Bash to IDL.

**Pull**

The equivalent of ‘downloading’ things from a remote repository to a local repository. When we grade your homework you pull the code from Bitbucket to your local computer to look at our comments.

**Fast-Forward**

What happens when there are no errors or merge issues when using Git. The repository is fast-forwarded to the desired state.

---

### 10.3 Chapter 3: Basic IDL

**Interactive Data Language (IDL)**

The programming language we will be using this semester. Though some may argue against its age it really is an amazing language.

**Interpreter**

The bit of software that we use to interact with IDL. Equivalent to the Bash ‘Terminal’.

**Procedure**

A routine within IDL that does NOT have to have a return statement. A procedure, as the name suggests, just runs multiple lines of code in order until it reaches the end of the routine.
Variables
A structure in every programming language that can be modified and redefined at various instances according to the user’s desire.

Datatypes
Information stored within a computer must be finite. To increase efficiency it makes sense to store different types of information (Re: contrast a number with a letter) into different classes. This is why datatypes (literally made for holding different types of data) exist.

Integer
As in mathematics a whole rational number with the inability to be positive and less than 1. Example: \(2/3=0\) using integers.

Float
A floating-point number that can most easily be thought of as a decimal. Example \(2.0/3.0=0.66666\) using floats.

Bit
The most basic form of computer memory corresponding to a 1 or 0 state.

Long
A long integer. Can hold larger numbers than the standard integer datatype while still requiring the use of whole numbers. As with integers \(2/3=0\) using longs.

Double
A longer float. Equivalent to the Long datatype for integers but for floats. \(2/3=0.6666\) for doubles.

String
A datatype which can hold both numbers and other characters. If you restrict yourself to using only numbers within a string, IDL can still preform basic mathematical operations unlike most languages.

Concatenate
To add together. Most commonly referenced when discussing strings. To concatenate strings simply use the + symbol.

Function
A routine in IDL that MUST return some type of output. In contrast to a procedure which does not have this requirement.

Return Statement
The statement mentioned above that must be within functions. Syntax is \texttt{return, <whatever>}

Script
A.K.A Code. A script is a file that contains a list of commands for IDL to run. Often saved as .pro to differentiate from other languages/ to take advantage of syntax highlighting in text editors like emacs.

Syntax Highlighting
Using color to differentiate various structures within IDL. Only of benefit to the human user for readability.

Array
A datatype that can be most easily thought of as a list (possibly multi-dimensional) of one specific datatype.

Hard-Code
Chapter 10. Glossary

The process of not calculating but manually entering in numbers into a script so that one specific problem can be solved in one specific case. Usually one should avoid this practice as it can lead to unexpected errors later down the line.

Elements
The individual units of an array. For instance if I declare \( a = [1, 2, 3] \) then \( a \) has 3 elements.

Comments
As the name suggest these are little notes programmers leave for themselves and other to more easily understand the flow of information through the code.

Whitespace
The space between bits of code. When you indent your code you are adding whitespace. IDL, unlike some languages, does not require whitespace but is still suggested as it allows for easier readability by humans.

10.4 Chapter 4: Control Statements

Control Statements
Expressions within IDL that allow for control of information depending upon certain circumstances that the user can control.

Loops
A control statement which allows a certain process to be repeated many times.

Conditional Statements
A control statement that will only execute a single or set of commands if a certain condition (that the user controls) is met.

For Loops
A specific type of loop that uses an iterative variable. Usually read in English as "For this many times, preform this action".

Index Variable
AKA an iterative variable. Used in for loops (and sometimes while loops) to control the loop.

While Loop
A loop that continues WHILE some condition is true. This condition is checked everytime the loop starts over.

Truth Table
You may have learned about these in elementary school. See https://en.wikipedia.org/wiki/Truth_table

If Then Else
A conditional statement that will execute commands IF something condition is met. Can by accompanied by the ELSE part to run commands if the condition is not met.

Relational Operator
An operator used to set up conditions. For instance to test if \( 5 > 3 \) the proper syntax in IDL is \( \text{if } 5 \text{ GT } 3 \text{ then begin} \).

Nest
The process of putting loops or conditionals within other loops or conditionals. Should be avoided when possible as it can quickly increase runtime of your script.

Case Statement
A conditional that checks multiple cases and executes commands if they are found to be true.

**Switch Statement**
A conditional that checks multiple cases. When one case is found to be true, commands under that case and all following cases are executed. Literally acts as a switch when something is found to be true.

**Break Statement**
A statement used to break out of loops, case, and switch statements. Cannot be used to break out of IF THEN ELSE statements.

**Infinite Loop**
A loop that will continue on forever until stopped by some external process such as the user beating their computer to death. Or if they use Ctrl-C.

### 10.5 Chapter 5: Arrays

#### 1-D Arrays
An array using only 1 dimension. Can easily be thought of as a list of whatever datatype is being used.

**Structures**
Though used with abandon in other languages, structures have a very well-defined meaning in IDL. They can most easily be viewed as similar to arrays except allowing for multiple datatypes. This was the precursor to Python’s dictionaries.

**Lists**
A rather new creation within IDL that behaves much like an array and allows for multiple datatypes. Used rarely if ever.

**Array Generating Functions**
Functions such as `indgen()` or `fltarr()` which can easily generate arrays removing the need to manually type in a bunch of numbers.

**Rows**
Considering a 2D array rows are just as you’d expect in that the first row is the first set of numbers that go horizontally across your screen when you print the 2D arrays.

**Columns**
See Rows except the first column is the set of numbers that appear to go down your screen when you print the 2D array.

**Column-Major**
The idea that we reference 2D arrays by $a[\text{column},\text{row}]$ and not $a[\text{row},\text{column}]$. Though this is the opposite of matrix notation it is the standard (and more logical) convention when doing image processing. Since we are astronomers, image processing is much more important to use than array manipulation.

### 10.6 Chapter 6: Data Analysis and Plotting

**Spawn**
A procedure which allows you to "spawn" instances of Bash from within IDL. Effectively allows for execution of Bash commands from your IDL script. One of the best procedures in the entire language.
**Constant System Variables**
As the name suggests these are constant values that can be accessed using !. For instance !pi return 3.14159.

**Save**
Used in conjunction with Restore. Allows for the saving of routines and variables to be used in future IDL session.

**Restore**
Used in conjunction with Save. Allows for the recalling of saved routines and variables.

**Column Format**
ASCII text file data is usually stored in terms of columns with some type of delineation.

**Orbital Parameters**
There is a set of orthogonal orbital parameters that uniquely describe any possible orbit of a body around a central mass.

**ASCII**
Literally "American Standard Code for Information Interchange". Most easily thought of any type of text file. Note that the file need not have .txt at the end to be considered ASCII.

**Readcol**
A procedure which allows for easily reading in column formatted ASCII data.

**Log File**
A file containing messages and errors of what happened when a piece of code is run. Useful for longer projects.

**Interpolation**
Constructing guessed parameters from a discrete set of data. In the case of plotting this is implemented in assuming that each data point should be connected by a straight line. For scientific analysis this is not desirable because making this assumption for any given data set is not justified.

**Overplot**
As the name suggests, a procedure that allows you to put more than one set of data onto the plot.

**Colortables**
I could write pages on this topic. Allows for the assignment of various colors to be mapped to an integer value between 0-255 and used within plotting.

**PostScript**
A format that used to be the goto format to save plots. Now pdf is much more desirable.

**Contour Plot**
A type of plot which could be 3D but instead uses lines of constant Z value and often color to display the same information in 2D. These are useful when writing MCMC code.

**Colorbar**
A way to show people what colors on your plot correspond to what values. For instance I may use red to show a higher temperature and having a colorbar that shows what color maps to what temperature is very useful.
10.7 Chapter 7: Strings and FITS Files

Flexible Image Transport System (FITS) Files
A type of file which allows for easy storage and access of astronomical images (Re: images taken off the telescope).

Pixels
A small CCD that gathers photons and is the smallest element of a picture that can be modified in image processing.

Charge-Coupled Device
The device, typically made of doped semiconductors, that allows the pixels to capture energy from photons and produce an image.

Header
The table of useful information given by the FITS file generator detailing information about the parameters of the image’s origin. It typically includes date taken, location, CCD used, magnification, weather, etc.

Dark Current
The small amount of current that flows through CCDs even when there is no excitation due to photons.

Right Ascension (RA)
The longitudinal (East and West) location of an astronomical object relative to the Earth’s surface. It is measured in hours, minutes, and seconds. Hour 0 is defined to be the location of the sun during the vernal (spring) equinox.

Declination (DEC)
The latitudinal (North and South) location of an astronomical object relative to the Earth’s surface. It is measured in degrees, arcminutes, and arcseconds. There are 60 arcminutes in a degree, and 60 arcseconds in an arcminute. 0 degrees is defined as the celestial equator which is the plane that passes through earth’s equator.

Dark Frame
An image taken with no light so all the dark current can be adequately mapped and removed from the following images used for measurement. Typically the first image taken on an observing night.

Color scale
The color scheme a program uses to display the image. Changing the colors in the scale, or the scale from linear to logarithmic can allow a user to clearly see different features of an image.

10.8 Chapter 8: Data Structures and MCMC

Data structures
An array-like object that can store different datatypes and from which you can access different elements by their tag.

Markov Chain Monte Carlo (MCMC)
A statistical modeling technique that is typically used to model a complex system. It compares the results after stepping through a a number of steps and creating a resulting distribution.

Tags
The name of a defined variable in a data structure.

**Probability distribution, probability density function, pdf**
A distribution or function that maps the likelihood of different results to occur.

**Standard Deviation**
A statistical characteristic of a data set describing the typical deviation of the group as a whole.

**Random walk problem**
A problem consisting of any number of parts that travel or act in a random fashion.

## 10.9 Chapter 9: Error Analysis and Fitting

**Matrix**
A N-dimensional array with its own set of mathematical rules governing how to manipulate them.

**Vector**
A quantity with a direction and magnitude. They often make up the columns of matrices.

**Scalar**
Simply a number with no direction.

**Dot product**
A matrix operator that allows you to multiply two matrices together. It is easier to understand the physical meaning when considering vectors. The dot product of two vectors returns the component of the first that is pointing in the direction of the second. In order to take the dot product of two matrices, you need to take the transpose of one of the first matrix.

**Transpose**
The transpose of a matrix, $A$, denoted by $A^T$, is simply the reflection of $A$ across its diagonal. As can be seen in Figures 9.16-19.

**Inverse**
The inverse of a matrix, $A$, denoted by $A^{-1}$, is the matrix that satisfies $A \times A^{-1} = I$ where $I$ is the identity matrix, consisting of all 0s except for 1s on the diagonal. If you are curious about just how this is done, a quick internet search will reveal a lot, but the actual methods are outside the scope of this class, but are covered in Math 54: Linear Algebra or an equivalent class.

**Matrix operator**
A mathematical operator in IDL that is designed to specifically designed to be used on matrices.

**Non-linear**
Also described as evil by certain professors, this property simple means that the variables in an equation are related with higher powers than 1. Many characteristics of the world do not follow the linear equations like $y = mx + b$, and thus are described as non-linear.