Computational Physics

Release 0.1

Nov 29, 2022

Contents:

There are a few Latex syntax guidelines worth remembering:Our goal in Computational Physics is to develop the skills necessary to use computers to solve physics problems. You will take the tools that you learned in Phy280 and apply them to a variety of physical systems to learn about algorithm development, modeling and analysis.

CHAPTER 1

Course topics

Materials will be divided into lessons as follows:

Lesson	Content
1	Intro to UNIX; Remote login
2	The UNIX Filesystem; The Shell; Commands and wildcard
3	Scripting; Startup files; Emacs
4	Root finding techniques
5	Numerical differentiation and Integration
6	Latex ; Bibtex
7	Ordinary Differential Equations
9	Laplace's Equation
10	Project

CHAPTER 2

Contents

Lesson content, readings and due dates are subject to change

2.1 General info

This course is designed for students in any of the physics majors or minor at Eastern Michigan University as well as students that are taking introductory level physics.

Course materials are freely available.

2.1.1 Instructor

David Pawlowski
dpawlows@emich.edu
Strong 240D
734.487.8644

2.1.2 Course website

https://computationalphysics.rtfd.io/

2.2 Installing Python

In order to complete the tutorials and exercises, you should download a python distribution on your own computer. While the Mac computers in the campus computer labs have python installed on them, these versions may not have some of the packages that we will use in this course. The purpose of this page is to help you to install Python and different Python packages onto your own computer. While it is possible to install Python from the Python homepage https://www.python.org/, **we highly recommend using** Anaconda which is an open source distribution of the Python and R programming languages for large-scale data processing and scientific computing. Anaconda combines a basic python distribution with many packages that are commonly used to do science. This makes life a lot easier for us, the users, in the long run.

2.2.1 Install Python on Windows

Following steps should work on all Windows 7 and 10 computers.

Download Anaconda installer (64 bit) for Windows.

Install Anaconda to your computer by double clicking the installer and install it into a directory you want (you will need admin privileges). If promped, install it to **all users** and use default settings.

Test that Anaconda´s package manager called `conda` works by opening a command prompt as a admin user (http://www.howtogeek.com/194041/how-to-open-the-command-prompt-as-administrator-in-windows-8.1/) and running the command `conda --version`. If the command returns a version number of conda (e.g. `conda 4.5.9`) everything is working correctly.

2.2.2 Install Python on macOS

OSX users can visit the Anaconda downloads page and click to download the latest 64-Bit Graphical Installer for the Python 3 version of Anaconda.

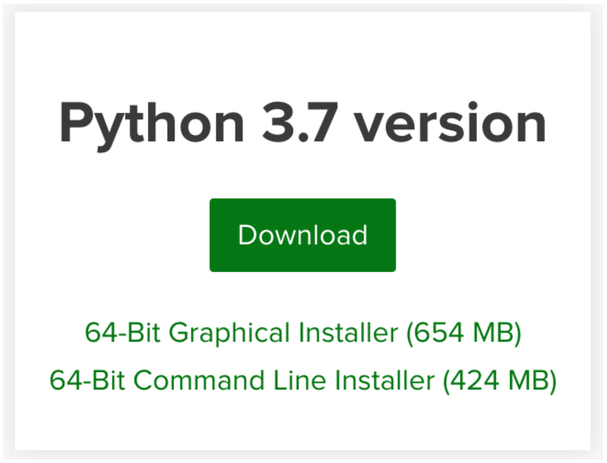

Python 3.7 version
Download
64-Bit Graphical Installer (654 MB)
64-Bit Command Line Installer (424 MB)

Fig. 1: You probably want the graphical option, but you do you.

Run the installer and follow the prompts to install Anaconda using the default options.

Then, open up the Terminal application and enter `conda --version` to verify a correct installation.

Note: The `conda` command in the Windows Command Prompt or Mac Terminal can be used to install packages that were not included with the base distribution, though we probably won't need to do that in this course.

2.2.3 Updating

Once you start using Anaconda or the Spyder IDE, you may get the occasional "software needs updating" dialog box popping up. Updating is easy from the Command prompt (windows) or Terminal (Mac). Simply execute the command `conda update spyder` and follow the instructions.

2.3 Learning Objectives

There are 4 main learning outcomes that you should achieve by taking this course. Specifically, you should understand

1. how to use the Linux operating system to streamline your workflow.
2. how to apply computational techniques to analyze a variety of physical systems.
3. how to perform data analysis tasks.
4. how to use computational tools to gain insight into the behavior of physical world.

While we will be working with the Python programming language, it should be noted that the concepts that we will cover can be utilized in any language.

2.3.1 Git and GitHub

In this course, we will use the version control software Git as well as the Git repository hosting service, GitHub.

Having a system for keeping track of different versions of the software that you create is one of the crucial set of "good coding practices". There are several systems for doing this. Git is one of the more popular ones. In this class, we will only scratch the surface of using Git.

While Git is the software that helps to keep track of your code, GitHub is a service that hosts your code so that you can access it from any computer and share it with other users. When you use Git to track your code, you create a code "repository", or "repo". GitHub stores your repositories on their own servers and makes it easy to clone (download a separate copy) the code and commit changes back into the repo when you are finished making modifications.

GitHub is free for anyone that is ok with making their repositories publicly available. However, since you are doing this as part of a class, the repositories that you create for this class will be private so that only you and I can access them.

You should have used learned how to use github in Phy280, but if you need a refresher, please see https://foundations-of-scientific-computing.readthedocs.io/en/latest/lessons/L3/coding.html.

2.4 Python Vocabulary

For reference, a few common terms that will be used throughout this course.

- **Variable**: a way of storing values into the memory of the computer by using specific names that you define.
- **Script**: a document that contains your Python code that you can execute. Python script files should always have the `.py` file extension.
- **Index**: the **location** of specific value stored in Python lists, arrays, or tuples. Not the value itself. The first index value of list is always 0.
- **Data types**
  - Integer (int) = Whole number
  - Float (float) = Decimal number
  - String (str) = Text
  - Boolean (bool) = True / False
  - List (list) = A "container" that can store any kind of values. You can create a list with square brackets e.g. `[1, 2, 3, 'a', 'b', 'c']`.

  - Tuple (tuple) = A similar "container" as list with a difference that you cannot update the values in a tuple. You can create a tuple with parentheses `(1, 2, 3, 'a', 'b', 'c')`.

- **Prompt**: The input area in the Anaconda Graphical User Interface (GUI) used to test single lines of code or interact with your code.
- **Argument**: Information that the computer needs to perform commands or execute functions.
- **Module**: Software that is not part of the base python distribution that needs to be imported before it can be used.
- **cwd**: Current working directory
- **Command line**: The terminal prompt
- **Filepath**: The location of a file within the filesystem.
- **root** (user): The superuser of the system. Capable of modifying any file on the computer.
- **root** (directory): The lowest level in the filesystem. Literally /.

2.5 Transferring Files

It is often necessary to move files from one computer to another. The process for doing this depends on the type of systems involved. There are a number of graphical application available for this purpose. On a Mac, I have used CyberDuck. On Windows, winSCP and FileZilla are both free options. Regardless of the software type, the process is similar from application to application. The important thing is that whichever software you choose, you must have the option to transfer files using either scp (secure copy) or sftp (secure file transfer protocol).

2.5.1 Graphical Options

Upon opening any of these software packages, you will first need to connect to the remote computer. Often, there is a "Connect" button somewhere in the menu or perhaps toolbar with some options. In all cases, you will be asked to specify the Host (e.g. emich.edu or similar) that you want to connect to and your credentials on that computer. Depending on the application, you may have to specify the port (typically 22) as well and you may have to select which protocol you wish to use to connect (scp or sftp). With this information entered, you should be able to initialize a connection to the remote computer, and if successful, you will be presented with two sub-windows: one containing a snapshot of your local filesystem and one with a snapshot of the remote filesystem.

At this point, transferring files is a simple as dragging and dropping a file from one computer to another. You should be able to navigate the remote file system by clicking through folders on the system like you would if it were local. You should be able to push files to the remote as well as pull files from remote to local.

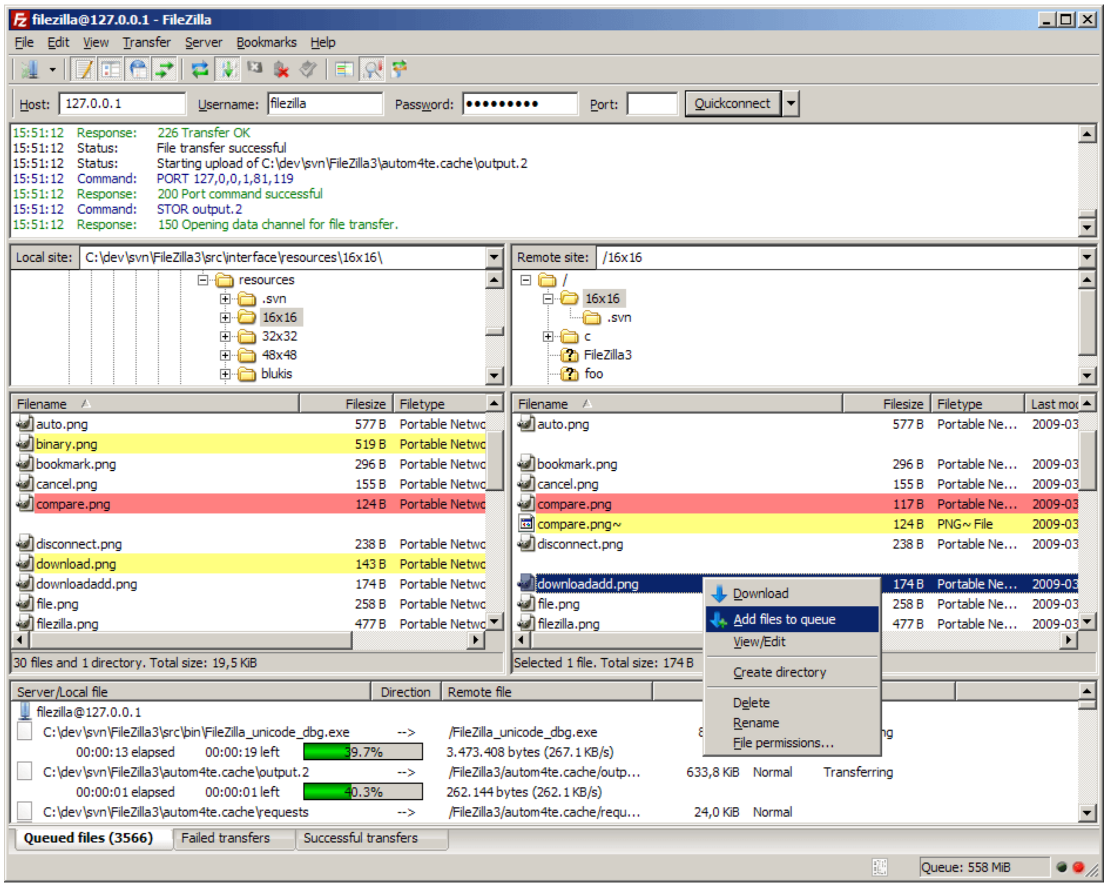
2.5.2 Linux scp

While graphical options are available on Linux computers, there is also a command line option: `scp`.

The `scp` command is similar to `cp`: you are copying a file from one location to another. The difference is that since we are moving a file from machine to machine, we need to include information about the remote machine in the process. Usage is:

```
scp [username]@hostname:’path of file to transfer’ [username]@hostname:’path to␣
↪transfer to’
```

The first argument contains information about the file that you want to transfer, and the second argument contains information about where you want to transfer to. The username is optional and only necessary if you username on the


filezilla@127.0.0.1 - FileZilla
File Edit View Transfer Server Bookmarks Help
Quickconnect

Host: 127.0.0.1 Username: filezilla Password: ••••••••• Port:

```
15:51:12  Response:  226 Transfer OK
15:51:12  Status:    File transfer successful
15:51:12  Status:    Starting upload of C:\dev\svn\FileZilla3\autom4te.cache\output.2
15:51:12  Command:   PORT 127,0,0,1,81,119
15:51:12  Response:  200 Port command successful
15:51:12  Command:   STOR output.2
15:51:12  Response:  150 Opening data channel for file transfer.
```

Local site: C:\dev\svn\FileZilla3\src\interface\resources\16x16\

```
resources
  .svn
  16x16
  32x32
  48x48
  blukis
```

Remote site: /16x16

```
/
  16x16
    .svn
  c
  FileZilla3
  foo
```

Filename	Filesize	Filetype
auto.png	577 B	Portable Netwo
binary.png	519 B	Portable Netwo
bookmark.png	296 B	Portable Netwo
cancel.png	155 B	Portable Netwo
compare.png	124 B	Portable Netwo
disconnect.png	238 B	Portable Netwo
download.png	143 B	Portable Netwo
downloadadd.png	174 B	Portable Netwo
file.png	258 B	Portable Netwo
filezilla.png	477 B	Portable Netwo

30 files and 1 directory. Total size: 19,5 KiB

Filename	Filesize	Filetype	Last moc
auto.png	577 B	Portable Ne…	2009-03
bookmark.png	296 B	Portable Ne…	2009-03
cancel.png	155 B	Portable Ne…	2009-03
compare.png	117 B	Portable Ne…	2009-03
compare.png~	124 B	PNG~ File	2009-03
disconnect.png	238 B	Portable Ne…	2009-03
downloadadd.png	174 B	Portable Ne…	2009-03
file.png	258 B	Portable Ne…	2009-03
filezilla.png	477 B	Portable Ne…	2009-03

Selected 1 file. Total size: 174 B

```
Server/Local file                                      Direction  Remote file
filezilla@127.0.0.1
C:\dev\svn\FileZilla3\src\bin\FileZilla_unicode_dbg.exe   -->     /FileZilla_unicode_dbg.exe
    00:00:13 elapsed    00:00:19 left    39.7%                3.473.408 bytes (267.1 KB/s)
C:\dev\svn\FileZilla3\autom4te.cache\output.2             -->     /FileZilla3/autom4te.cache/outp…   633,8 KiB  Normal  Transferring
    00:00:01 elapsed    00:00:01 left    40.3%                262.144 bytes (262.1 KB/s)
C:\dev\svn\FileZilla3\autom4te.cache\requests             -->     /FileZilla3/autom4te.cache/requ…   24,0 KiB   Normal
```

Queued files (3566) | Failed transfers | Successful transfers

Queue: 558 MiB

Fig. 2: The main window for FileZilla. Not pretty, but it gets the job done.

two machine differs. The hostname is only included for which ever system is remote. You should, in general, include the quote marks with the remote machine argument.

For example, if I wanted to send a file called “images.tar” to a computer called **yipe**, I would do:

```
% scp images.tar dpawlows@yipe.emich.edu:'/'
```

This will copy the file to my home directory on Yipe.

To get a file from yipe:

```
% scp dpawlows@yipe.emich.edu:'/scripts.tar' .
```

will bring the file to my local machine and put it in my current working directory. If you want to transfer an entire directory, simply include the -r flag.

```
% scp -r /Documents/ yipe.emich.edu:'/'
```

will transfer my Documents directory and all of its contents to yipe. Note again that I don’t actually need to include my username if it is the same on the remote system as it is on my local machine.

2.6 Lesson Overview

In this first lesson, we will start learning about UNIX and UNIX-like operating systems, including Linux. You will be introduced to the Linux Terminal and some basic commands that will allow you to start to get to know the Terminal and how to use it.

The main components of the first lesson are:

1. *an introduction to UNIX/Linux*
2. *a quick overview of the Terminal*
3. *logging in to a remote computer*
4. *basic Linux commands*

2.6.1 Learning Goals

After this week you should be able to:

- differentiate between different operating systems
- understand the difference between the Terminal and Windows PowerShell
- log into a remote computer using SSH (or PuTTy)
- be able to recall and execute basic Linux commands associated with creating and modifying files

2.7 Introduction to UNIX/Linux

2.7.1 A brief history

In the early days of computing in the 1960s, software developers needed to find a way for users to provide instructions to the computer so that it could perform calculations. The method of using a keypunch to punch holes in a piece of paper in a way that could be read by the computer was cumbersome, time intensive, and prone to error. Instead, early developers wanted a way to type programs in to a terminal while enabling a computational environment which facilitated sharing of resources between many users. In 1969, a Bell Labs team lead by Ken Thomsen and Dennis Ritchie accomplished this by creating the UNIX operating system (OS).

Throughout the 70s and 80s, UNIX's popularity grew in the computing world and Bell Labs was able to license the UNIX operating system to a variety of commercial and educational users. However, the cost of a license was seen to be prohibitive to an individual user and before long, members of the software community looked to make the OS more widely available by creating alternative versions. The most famous and widely used version of UNIX was created by Linus Torvalis in 1991. Linux is considered a UNIX-like OS in that it retains many of the key components of UNIX (specifically the kernel, the shell, and the programs). While Torvalis originally released Linux for no cost, today there exist a variety of free and not-free versions of Linux, including Rea Hat, Ubuntu, Suse to name a few.

The Linux OS is widely used in computing. It is found on personal desktop and laptop computers, tablet devices, mobile devices, servers, and supercomputers. The Android OS is based on a modified version of Linux. The Mac OS, while strictly speaking is a different UNIX-like system, has adopted some Linux specific features over the years.

2.7.2 Reasons to use UNIX-like systems

As personal computers came to be used more widely by the generally public, it became necessary to develop software that made using a computer relatively straightforward for the lay person. As a result, most people are unaware of some of the more useful features of the very computer that they use on a daily basis. Remember, UNIX, and subsequently Linux, was designed to 1) make it easy for a user to give instructions to the computer and 2) make it easy to multitask (or have multiple users). These features of Linux are still incredibly useful for today's software developers. While modern Linux distributions come with sophisticated graphical user interfaces (GUIs), all of them provide the user with access to a Terminal program that can be used to interact with the computer at a basic level.

I would argue that there are several reasons to use the Linux operating system and the Terminal:

1. Anything that can be done via the GUI can be done (usually more quickly) using the Terminal.
2. The Terminal makes it easy and extremely fast to perform common workflow operations such as creating files, moving files, copying files, deleting files, etc.
3. The Terminal provides access to tools for logging in to remote systems, such as your work computer or a supercomputer.
4. You can use the Terminal to quickly navigate your filesystem and to keep track of where you are within it.
5. You can use it to execute many programs and perform operations on multiple files simultaneously.
6. The terminal can be used to control access to files that are on a shared computer so that certain users can access your files while others cannot.
7. The commands that are used to do work in the Terminal can be combined in a file and executed from the file, allowing you to essentially "program" your workflow.

There are certainly many more, but in summary, using the Terminal makes performing routine tasks that are performed on a computer much faster and more efficient. One of the goals of this course is to get you started with using the Terminal so that you can use it as an interface to access the true power of your computer.

2.7.3 A note on Windows

You'll note that the Windows operating system was not mentioned in any examples above. Windows is not based on a UNIX-like OS and thus it lacks many of the features that make Linux systems useful for power users. Instead, from the beginning, Windows aimed to make computers user-friendly by pursuing an optimal GUI, one of the main reasons that Windows dominated the OS market share for many years. While Windows computers have always come with a Command Line Interface (CLI) (Note that the Terminal is often referred to as the command line), the CLI provided access to lower level system commands via MS-DOS. DOS was initially developed for single-user systems, was not built to support multiple processes per user, and lacks many of the basic commands that make Linux powerful.

Recently, Windows computers have included a different program that acts, in someway, to replicate the Linux Terminal: PowerShell. Windows Powershell offers the user many of the same commands and features that Linux users use regularly. If you use a Windows machine, while you will be logging in to a linux computer to complete assignments in this course, you should also learn to use the Powershell.

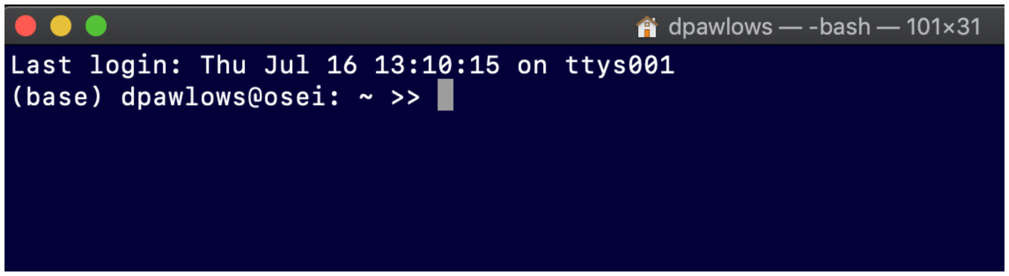
2.7.4 Linux vs. Windows

None of this is to say that Windows machines are inferior to Linux computers. Rather, they were initially designed to be good at different things, and those initial design choices are still evident in modern computers. Many software developers choose to use Windows computers as many choose Linux. My belief is that the Linux OS offers efficiency benefits to those that perform repetitive tasks on their computer and that these benefits can lead to a large time savings over the course of a project. I personally use both a Linux computer and a Mac for all of my work.

Many developers choose to purchase or build Windows computers and then partition their hard drive so that they can dual boot Linux. I have also used this option in the past. Ultimately, the decision of which computer to use often comes down to which computer you learned to use first (if not which computer does what you need it to do for less money) and that is fine. I would urge you to learn how to use whatever computer you are using at a more fundamental level and to understand how it can help you do the work that you normally do faster and more efficiently.

2.8 The Terminal

Before we start learning how to use Linux (remember, Linux is based on UNIX. I will use the two terms interchangeably throughout the course), I want to add a couple details about the Linux Terminal. Again, the Terminal is the program that allows the user to interface directly with the computer. Only Linux/UNIX computers have a Terminal program, which is why you all have been given an account on a remote computer.



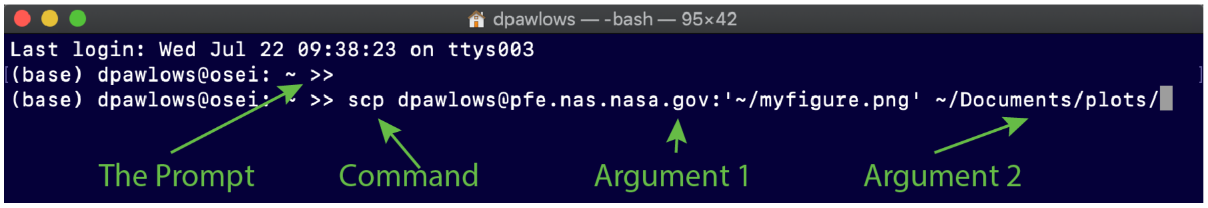
```
Last login: Thu Jul 16 13:10:15 on ttys001
(base) dpawlows@osei: ~ >>
```

Fig. 3: My Terminal at startup with some customization

On a Mac, you can open a Terminal by going to Finder > Utilities > Terminal. On a Windows computer, again, there is no Terminal, but you can open PowerShell by clicking Start and typing PowerShell in the search bar. In both cases you can perform actions by typing and executing specific *commands*. As you type, your text will be entered into the *prompt*. In the image above, the prompt is the line that ends with **>>**, but note that I have set my prompt to look that way. Yours will probably look different. We will spend some time learning about a variety of commands that allow us to create and manipulate files, perform basic data analysis tasks, access remote computers and more in this class.

When you execute a command commands using the Terminal you may need to include a(n) **argument(s)**; pieces of information that a specific command needs to do its job. Some commands do not require any arguments while others may require 1 or more.

In the above example, I am using the `scp` command to transfer a file from a NASA computer to my local computer (this command is not available on windows computers). We will talk about this command later, but you can see that



```
Last login: Wed Jul 22 09:38:23 on ttys003
(base) dpawlows@osei: ~ >>
(base) dpawlows@osei: ~ >> scp dpawlows@pfe.nas.nasa.gov:'~/myfigure.png' ~/Documents/plots/
```

Fig. 4: A useful command

the first set of characters that come after the prompt must always be a command. Then, any characters that come after a command are arguments, each separated by a space. In this case, there are two arguments.

Note: I want to stress that the first set of characters that come after the prompt must be a command. A common error for users is to try to type a filename or something, and hit enter. Even if the filename that you are typing exists in the location that you are working in on your filesystem, you will still get an error. Another common error is to simply misspell the command that you are trying to enter.

2.9 Remote Computing Using SSH

As part of being a physics major, you have access to a remote Linux server called *serenity*. Our first task will be learning how to access this computer using an application called SSH.

2.9.1 Linux/Mac Users

For Linux and Mac users, this is very easy. You simply need to open up the Terminal and use the `ssh` command. Execution looks like this:

```
>> ssh [username@]hostname
```

Again, the >> are meant to indicate the prompt. ssh takes 1 argument. That argument has a required part (hostname) and an optional part (username@). When looking at documentation for linux commands and often programing in general the [ ] symbols are often used to designate optional parameters.

Note: ssh stands for "Secure Shell" which is a remote computing protocol that is "secure", i.e. people that are snooping on your internet connection can't see the information that you are sending to or receiving from the remote computer. This was not the case for all previous remote login commands.

The hostname is the name of the computer that you are logging into. You all have an account on the computer **serenity.emich.edu**. That is the complete hostname for serenity. Your username on that computer is the same as your emich uniquename. The reason the username is optional is if the username on your local computer (the computer that you are sitting in front of) is the same as that on the remote computer, then you don't have to write it out.

So, if I wanted to log into serenity, I would execute either

```
>> ssh dpawlows@serenity.emich.edu
```

or

```
>> ssh serenity.emich.edu
```

depending on if my username on serenity is the same as that on my local machine.

2.9.2 Logging in for the first time

Let's say I was going to log onto serenity for the very first time using the command `>> ssh serenity.emich.edu`. When I execute this command, my local computer will check to see if the remote host (serenity) is to be trusted. It does this my looking in a file on your computer called the known hosts file for information about the remote computer. If it doesn't find any information about that computer, it will tell you and then ask you if you are sure that you want to proceed. Assuming you know the computer that you are logging into, you should answer "yes".

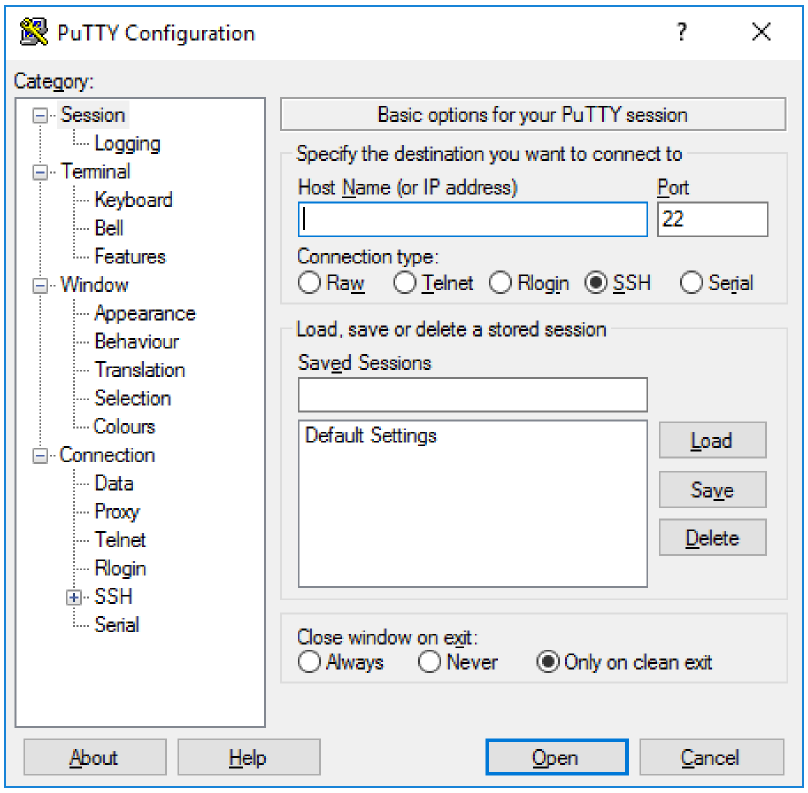
At that point, information about the remote computer will be added to the known hosts list on your computer and you won't get that message in the future (unless something about that machine changes). Then, you will be prompted for your password and you can enter it. Note that as you type your password, no characters will show up on the screen as this would be an obvious security flaw.

If you entered your credentials correctly, you should be logged into serenity, at which point you can interact with the computer via the Terminal as if you were sitting in front of it.

2.9.3 SSH using Windows

Only recently has Windows made it possible to use ssh to access remote computers via PowerShell. Still, doing so can be a bit cumbersome as additional software tools must be installed. Instead, historically Windows users have opted to use a different SSH client. One of the most popular of these is called **PuTTy** and it can be downloaded at https://www.chiark.greenend.org.uk/~sgtatham/putty/ PuTTy doesn't need to be installed; you just download it and put it somewhere, like the desktop or in "Program Files" or something. Once downloaded, just double click the icon, shortcut, or whatever to open it up.

PuTTy gives you access to ssh via a simple GUI. One you open PuTTy, you should see the configuration screen.



Host Name (or IP address):
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Saved Sessions: Default Settings
Close window on exit: Always Never [✓ Only on clean exit]

Fig. 5: The PuTTy configuration screen

The most important information that you need to specify is the Host Name (serenity.emich.edu) and the port (22, the default port for ssh). There are all sorts of configuration options, but you don't need to deal with any of these unless you want to do exciting things like change the color of the terminals that you open and use a specific font size and type. As you get the hang of using PuTTy feel free to change these at will. Additionally, you can save a session which will allow you to store information about a hostname and other configuration options for easy use in the future.

Once you have a hostname entered, you can click "open" and PuTTy will attempt to establish a connection to the remote machine. Again, if this is your first time connecting, you will get a message about the remote computer not being in the known hosts file and you can just say yes, you do want to connect. Then, you will be prompted for your username and password. If successful, you will be given a terminal window which will allow you to enter commands and run programs on the remote Linux computer.

SSH in Powershell

Like I mentioned, it is possible to use ssh via PowerShell. There are a variety of ways to make this happen. In my opinion, the easiest way is to install the OpenSSH Client using a package manager (a utility that handles the downloading, installation, and updating of software packages so you don't have to). I don't have much experience with Windows package managers, but I have heard ok things about **Chocolatey**.

You can find instructions for downloading and installing OpenSSH and Chocolatey here: https://medium.com/@haxzie/using-ssh-in-windows-powershell-complete-installation-guide-ae029a9e3615

Once OpenSSH is installed, the ssh command should be available to you via PowerShell and it can be used just like described for *Linux/Mac Users*.

2.10 Basic Linux Commands 1

In order to work with a Linux based operating system, you have to become familiar with Linux commands. There are many, and throughout this course you will become familiar with a lot of them. Here we will get started with some of the most basic ones.

Nearly every Linux command has documentation called **man pages**. You can access the man pages by typing: >> `man command-name` for the appropriate command. This will bring up a document in your terminal window that you can navigate through using the up and down arrows to scroll line by line, and the space bar to advance to the next page. Man pages tend to be a bit cryptic in that they are meant to be brief and understood by people with a strong Linux background. Understanding the details may be tricky, but you should be able to look at a man page and get an understanding of what a particular command is supposed to do, and also see some of the options for that command.

We talked about command arguments in the introduction to *The Terminal*. Additionally, just about every Linux command has some options that you can specify when using the command. These options are referred to as **flags** and are either indicated by a single hyphen "-" followed by a letter (e.g. -r), or a double hyphen "–" followed by one or more words (e.g. –recursive). These flags may be similar or different for different commands (often commands that do similar things have similar flags).

2.10.1 Files and Directories

Linux is all about manipulating files. The following commands will allow you to work with files (and directories); you will use these a lot. First, a little more detail about what is going on when you are working in the terminal. When you first login to a Linux system, the directory that you are accessing will be your **home directory**. The home directory has the same name as your user name, and it is where all of your personal files and subdirectories will reside. You will be able to create, remove, or move files and directories in you home directory, but you will not be able to do so in any directories higher in the file hierarchy (more on this later).

Lets start doing a few things in your home directory.

ls (list)

To find out what is in the current directory, you can use the `ls` command. This will list all the files and directories that are in your current working directory (which I will call **cwd** for short). On a typical computer you might see several files such as Desktop, Documents, Downloads, etc. These files are actually directories and may or may not contain other directories or files.

The ls command as you used it only told you the contents, nothing about the contents. To see more, use:

```
>> ls -l
```

The -l flag stands for "long". This tells you much more information about what is in a directory. We will talk about some of the information later, but a few notes on what you are looking at:

1. If there is a "d" in the first character of a line, that means the corresponding file is a directory. 2. You should see your username in the 3rd row. This tells you that you are the owner of the file, and therefore, you control the permissions (who can read, write, or execute) for the file. 3. The row before the date tells you the size of the file or directory in kilobytes. 4. The date is the timestamp that signifies the last time the file or directory was modified.

Another useful flag for ls is **-a**:

```
>> ls -a
```

This will show any "hidden" files. A file may be hidden in Linux by having the first character of the filename start with '..' You will notice that there are 2 strange files ' . ' and ' .. ' . The directory, ' . ' is Linux for "the directory that you are currently in", while ' .. ' is Linux for the directory one level up in the hierarchy. So, for example you could execute: `>> ls -a .` and you would see the contents of your cwd, exactly the same as ls -a. Or, you could do: `>> ls -a ..` to see the contents of the directory one level up.

In the last two examples, ' . ' and ' .. ' were arguments provided to the ls command. I could replace the either of those with another directory to see its contents. Also, you can combine flags:

```
>> ls -la Documents
```

would show you the contents of the Documents directory, in long format, including hidden files. There are many more flags that you can use with ls, and I'll leave it to you to explore them. Those covered here tend to be the most useful.

cd (change directory)

Now that you can see the contents of a directory, it is useful to be able to more around the file system. The command `>> cd directory` will change the cwd to "directory". For example, assuming you are in your home directory, you can change to Documents using:

```
>> cd Documents
```

Now you can do an *ls* to see the contents, if there are any. To get back to your home directory, which is one level up in the hierarchy, you do:

```
>> cd ..
```

You can confirm to yourself that you are back in your home directory by doing a ls.

Try it yourself: What happens if you run the command cd . ?

One last note on cd, if you type the `cd` command with no arguments, you will change directories to your home directory.

mkdir (make directory)

The `mkdir` command will create a new directory. mkdir needs one argument: the name of the directory to be created.

```
>> mkdir Linux
```

After executing this command, typing `ls Linux` will show the contents of your newly created directory, which should of course, be empty.

pwd (print working directory)

A **file path** tells you where you are in relation to the entire Linux filesystem. Let's say you are in your home directory. Executing the command:

```
>> pwd
```

will print the path of your current working directory. In my case, it is simply "/Users/dpawlows". We will talk more about the filesystem later, but you should know that each of your home directories should be in the /Users directory. /Users or /Home is where you would typically find the home directories of all users on a Linux computer. You can change into this directory and see this for yourself:

```
>> cd /Users
>> ls
```

Doing this should show you a list of everyone's home directory. What happens if you try to make a directory here?

```
>> mkdir temp
```

Since you are not the **owner** of this directory, and the permissions for the directory are set such that the owner is the only one that can write to it, you get an error. If you do a `ls -la`, you will see that the owner is someone named **root**. The root user, also know as the **superuser**, is the user that has permission to do anything they want on the system. root is an account on every Linux system, and it can be very dangerous to use the root account. When you are logged on as root, you can delete, rename, modify, etc. any file on the system, including those that are required for performing system tasks. For this reason, most people try to avoid logging on as root unless it's absolutely necessary. You will not have access to the root account on this machine for obvious reasons.

mv (move)

You can move a file from one directory to another **and/or** change the name of the file itself using the `mv` command. This doesn't copy the file; i.e. you only end up with one file after the command has been executed. If you change directories to your home directory, and make a new directory called 'temp', you can change its name to MyFiles by typing:

```
>> mv temp MyFiles
```

Alternatively, you could move the temp directory out of your home directory and into another directory completely by typing:

```
>> mv temp Documents/
```

or do both:

```
>> mv temp Documents/MyFiles
```

The system is smart enough to adapt its behavior based on the types of files that you are working with.

cp (copy)

If you want to copy a file or directory, you use the `cp` command. For example, try this:

```
>> cp /Users/dpawlows/Public/Phy380/assignment1.txt .
```

Note the “ . ”. cp requires at least **2** arguments, the source file (1st argument, the file that you want to copy) and the target file (the directory and name of the file that you want to copy the file to). If you just specify a directory for the target file, as in this case, then the target file has the same name as the source file. This command should put a copy of the first homework assignment in your working directory.

rm (remove file)

To delete (remove) a file, use the rm command:

```
>> cp assignment1.txt temp
>> ls
>> rm temp
>> ls
```

You can use rmdir to remove a directory (make sure its empty) or `rm -r` to recursively remove directories and all the files inside (**Careful, this can be dangerous!**). It is common to include the -i flag when using rm, which stands for interactive; e.g. ask before actually deleting anything.

cat (concatenate)

Lastly `cat`: a pretty useful and powerful command. Its primary use is to display the contents of a file to the screen. For example, in the directory that you copied the first homework assignment, you can type:

```
>> cat assignment1.txt
```

This will write out the contents of that file to the screen. Note that it doesn’t know what to do with formatting, or program specific characters (i.e. if you tried to use cat on a word document, you would get a big jumbled mess). `cat` can be used to read normal, basic, text. But cat can do more, you can also create a file using it:

```
>> cat > newfile
```

When you use cat like this, the Linux prompt doesn’t come back! Linux is waiting for you to do something. Anything that you type will be placed into a file called newfile. When you are done typing, press control-d to tell Linux that you are done, and you will be back at the prompt.

Here cat is being used as a really basic text editor. The “>” symbol has special meaning in Linux. When used, it is called **redirection** in that you redirect output from one command into a file (generally). Here, using the cat command with no arguments gives us a simple space to enter some text. Then, that text would normally be written to the screen. Instead, we use redirection to print the text to a file.

If you do an:

```
>> ls
```

you will see a file called newfile, and if you want to see the contents of the file, you can use:

```
>> cat newfile
```

You can use > with any command that gives output. For example, try:

```
>> ps aux > tempfile
```

This will print a bunch of information on the processes that are currently running on your machine into the file tempfile. You can see this by typing:

```
>> cat tempfile
```

You'll notice that if you run the ps command above over and over again, you don't actually add anything to tempfile. When one '>' is used, Linux creates a new file, and if one exists, it just overwrites it. However, if you use two of them '>>' then Linux will **append** the file that you are trying to redirect to. For example:

```
>> cat assignment1.txt >> tempfile
```

will place the contents of assignment1.txt at the end of the existing contents of tempfile.

In this manner, cat can be used to do powerful things. Later on we will talk about writing scripts, small programs that are comprised of one or more lines of Linux (or other) commands. Being able to manipulate files using cat can be extremely useful in this context.

2.11 Lesson Overview

This week, we we will continue learning about Linux usage. This includes some discussion of the linux filesystem, more details on the shell, the basics of shortcuts in linux, and more commands.

Specifically, you will learn about:

2.11.1 Learning Goals

After this week you should be able to:

- navigate the linux filesystem and figure out where useful files are located.
- use a variety of shortcuts to help work with large subsets of files.
- understand the concept of the shell and some key usage features.
- do more work in the Terminal with additional key commands.

2.12 The Linux filesystem

By now you should be able to perform basic operations within the Linux operating system including being able to: login, create and remove files, move around the filesystem, etc. Before we discuss more Linux commands and other tools, it is important to have an understanding of the file system including where and how to find things, and a little bit about what is going on in the background when you type a command.

2.12.1 Everything in Linux is a file

Every Linux computer has a similar filesystem hierarchy in which there are standard directories where certain files are placed. It is important to note that **Everything in Linux is a file** Again, everything in Linux is a file. The reason your mouse or printer works, the reason the terminal spits out a bunch of information when you type *ls -l*, and the reason that you are transferred to another computer when you use *ssh* is because there is a file somewhere that does something when you perform these tasks. Being able to locate the files, or figure out where they should be is important if you want to be able to use a Linux system effectively. So, let's explore the filesystem a little.

2.12.2 The root directory

When you login to a Linux system, you are placed inside your home directory. This is by no means the highest directory in the hierarchy, nor the most important. If you type the command:

```
>> pwd
```

and hit enter, you will see that you reside where you expect, i.e. for me, in the directory /Users/dpawlows. As you know already, typing

```
>> cd ..
```

brings you up one level in to the /Users directory. But that's still not the highest level in the hierarchy. There's one level more. Type

```
>> cd ..
```

to get there. Now, if you type `>> pwd` you should see that you are in the directory "/". This is a special directory called "the root directory" or just **root**. This is as high as you can go in the filesystem. Every file on this computer is contained somewhere in this directory.

Every Linux machine has several subdirectories in root that are the same or at least very similar from system to system. Figure 1 shows an example of the top levels of a typical Linux file hierarchy.

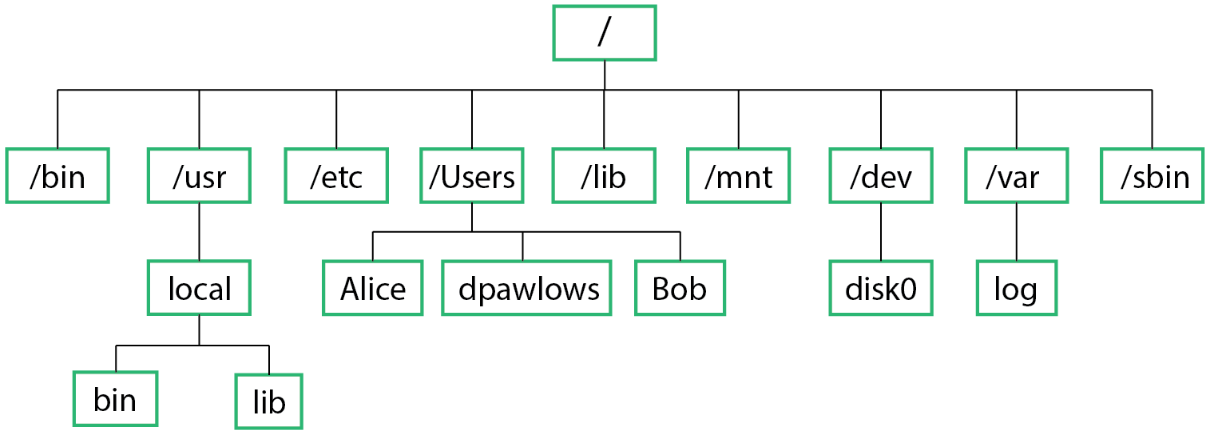


Fig. 6: The top levels of a typical Linux filesystem

Below are a few notes on some of these directories. For more information, the command

```
>> man hier
```

will bring up a man page that describes the Linux hierarchy.

2.12.3 Directories

/bin

Contains fundamental executable files (commands or programs) that are used by users. Only the most basic ones are here. Incidentally, the command `which` will tell you where an executable actually resides. For example, try:

```
>> which ls
```

/usr

Contains most of the user's programs and applications. Important subdirectories include:

- /usr/bin: Contains most of the executables on your system, e.g. emacs, man, sort.
- /usr/lib: Contains programming libraries- collections of program routines.
- /usr/local: Contains user installed programs and files.
- /usr/include: Contains include/header files, primarily for the C language.

/etc

Most of the configuration files are installed here, such as those needed for the shell, ssh, web server, etc. These files are used every time you start/use one of these things.

/Users

Contains home directories for all the users on the system. On some systems this is called /home instead.

/sbin

Executable files which are for system administration are stored here, as opposed to programs used by normal users. For example, the command `shutdown` resides here, which is a command that properly shuts the system down.

/dev

Contains files which give access to devices like the keyboard, mouse, screen, bluetooth, etc.

/var

Contains output and "temporary" files such as log files and email messages.

Others

In addition to these, Apple computers have other directories that are included in the root directory, most of which are needed for the graphical user interface (GUI) side of the OS or applications that are installed when using the GUI.

2.12.4 The Filepath

There are two ways to enter a filepath (or just path) to a file or directory. The **absolute** path of a file or directory is the path that you would get if you typed the pwd command. It starts from root and goes down the file hierarchy to your cwd. For example, try:

```
>> cd /opt/anaconda3/bin/python
```

You just navigated to the directory for the python executable by entering the directory's absolute path.

Instead, say you are already in */opt/anaconda3/*. Then, you could have simply entered

```
>> cd bin/
```

and you would have ended up in the same place. In this case though, you would have changed directories using the **relative** path. The relative path is based on the fact that the file or directory in the right most position of the path that you entered actually resides in the directory you are in. In other words, it is relative to your current position in the filesystem.

For example, had you entered

```
>> cd anaconda3/bin/
```

from your *home* directory, you would have received an error, because the file ~/anaconda3/bin doesn't exist. Entering a path either way is fine. It is just sometimes more convenient to enter a relative path, since it is almost always shorter. However, when using a path when creating a program or script, it is usually a good idea to use the absolute path, as you may want to run the program from any directory in the filesystem, which means that the relative path would usually not work.

2.13 Shortcuts and wildcards

The only device that we use to interact with the Terminal is the keyboard. So, in order to be able to do things like navigate the filesystem quickly and perform operations on multiple files at once, there are a variety of shortcuts that you must know.

2.13.1 Directories

Because a user's home directory is used so frequently, it is treated specially by Linux. To see this, navigate away from your home directory, if you haven't done so already. Now, enter

```
>> cd
```

(nothing after the cd command). After hitting enter and typing

```
>> pwd
```

you will see that you are back in your home directory. Linux assumes if you do not enter an argument after the cd command, you want to go to your home directory.

Next, type

```
>> cd -
```

(that's a hyphen). This should take you to the last directory that you were in. Using the hyphen is a quick way to bounce back and forth between two locations in your filesystem.

Now, let's say you wanted to go to your Documents directory. You could cd to your home directory, then cd again to Documents. Or you could enter the full absolute path, */Users/dpawlows/Documents/*. But, the easiest way is to do this is to use

```
>> cd ~/Documents
```

Linux gives special meaning to the tilde: ~; when used in the command line or in programming in general, ~ represents your home directory. Typing ~ is exactly the same as typing /Users/dpawlows/ (in my case). This is simply to make life easier, as all the directories that you will create and use will reside beneath your home directory.

2.13.2 Wildcards

One of the most useful features of Linux are called wildcards. Think of these as placeholders for omitted characters. For example, say you are looking for a file, but aren't sure if you named it physicsprogram or physicsscript. You can include a wildcard to stand for the part of the filename that you are uncertain about. For example, you can try

```
>> ls phyics*
```

This would list all files that start with the letters "physics". This could include, if they existed, physics, physicsprogram, physicspants, or physics help. You can use wildcards for just about any purpose in Linux, and many programming/scripting languages (such as C++, python, and perl) understand them as well. Here are a few guidelines:

- Use ? as a placeholder for one character or number.
- Use * as a placeholder for zero or more characters or numbers.
- You can include a wildcard at any place in a name; the beginning, the middle, the end, it doesn't matter. Also, feel free to use them in multiple places! i.e. *physics*.
- The ^ character can tell Linux to exclude a character

As an example, change directories to /usr/bin:

```
>> cd /usr/bin
```

Try listing all files that begin with z. What about all files that have the word "mirror" in them? What happens if you do this:

```
% ls z[^c]*
```

2.14 The Shell

One of the most important aspects of the Linux system is the shell. When you open a Terminal, you are gaining access to a shell session. The shell determines how you enter and re-enter commands and handles the execution of the commands themselves. One of the great features about Linux is there are different shells that you can choose from. While each shell has the same set of general features, all have different capabilities that may be useful depending on your situation.

In order to find out which shell you are using, you can use the command:

```
>> echo $SHELL
```

Note that capitalization matters on most Linux systems. `echo` is a command that tells Linux to display information about a variable. It is similar to *print* in python. In this case we pass 1 argument to *echo*: the variable named SHELL. The $ character is Linux's way of differentiating between a word and a variable (something that stores information).

The system's response to the echo call above will be to display the full path to your shell; /bin/bash. You are using the bash shell because that is what was specified when your account was created. By the way, bash stands for Bourne Again Shell, an extension of the original unix shell, sh. Typically, the shell you start learning with becomes the one you stick with.

You can see which shells are available to you if you type:

```
>> cat /etc/shells
```

and if you would like, you can change your shell with the `chsh` command.

2.14.1 Common features

There are a few features that are extremely convenient and available in most shells.

Tab completion

cd into your home directory and type (but don't hit enter):

```
>> cd Desk
```

Now, hit the tab key. Most shells have a feature which will automatically complete the word you are typing when the tab key is pressed. In this case, the shell should have auto-completed the rest of Desktop/ . By default, this will only work if there is only one possible match. If there is more than one possible match, the shell will beep at you and wait for you to type in enough letters to distinguish the two (or more) files. You can use tab completion to complete commands, directory names, filenames, and pretty much anything else you might enter into the command line that is sufficiently unambiguous.

It you enter some text into the shell and hit tab to autocomplete, but there are more than one possible entries that fit the text that you've entered, there is a way to have the shell return a list of possible completions instead of beep at you annoyingly. Ask me how to do this later (it involves modifying a configuration file. Don't worry, it's easy).

Viewing session history

Most shells will save a history of the most recent commands that you have entered. Use your shell for a little while, changing directories, listing files, redirecting output, etc. After you have entered several commands press the **up arrow** once. You should see the last command that you typed. You can continue to press up or down to cycle through your most recent commands, and when you find the a command you wish to re-enter, simply press enter.

As you cycle through the commands, you are able to modify any of the lines simply by using the left and right arrows. Next, use this `history` command:

```
>> history
```

You will get a list of the most recent commands and what time you entered them. You can re-execute any command by referencing the number in the history list:

```
>> !10
```

would re-enter the command that corresponds to the 10th entry in your history.

Scripting

All shells make it possible to combine one or more commands in a file for repeat execution. These files are called shell scripts and are extremely useful for expediting tasks that would otherwise take a long time, such as performing batch actions on many files at once, reorganizing a part of your filesystem, or handling routine computer maintenance tasks.

With in a shell script, it is possible to do all of the typical programming actions, including using loops, conditional and comparison statements, and logic. However, each shell has it's own syntax so if you become proficient at making bash scripts, you will still have to review syntax if you want to make tcsh scripts.

We wont address scripting specifically in this class as it's beyond the scope, but know that if you find yourself repeating the same tasks over and over when using Terminal, it might make sense to package those commands in a file and create a script.

2.15 Basic Linux Commands 2

You should be logged into a linux computer, such as serenity, to follow along with the examples in this lesson.

Now that we have a basic understanding of the Linux filesystem and know how to create and remove files and directories, it's time to add more commands to your tool belt so that you can start doing something useful.

2.15.1 more

You've learned how to quickly display the contents of a file using cat. But what if the file is huge? For example, try:

```
>> cat /var/log/system.log
```

You should get a lot of text printed to the screen. Now you could use the mouse wheel (if that even works) to scroll up through the text, or an alternative is to use *more*:

```
>> more /var/log/system.log
```

Now you should see the first "page" of text, and you can advance through it using the space bar or the page up/page down keys. Pressing *q* will allow you to exit without needing to go all the way to the bottom.

2.15.2 head/tail

Maybe you don't need to see the whole text, but you just want to see the first or last few lines. Then you'll want to use the head or tail commands:

```
>> head /var/log/system.log
```

will show you the first 10 lines of the linuxsystem.tex file, while

```
>> tail /var/log/system.log
```

will show you the last 10 lines. Adding the *-n M* flag will show M number of lines instead of the default 10.

2.15.3 Pipes

Recall that we can use > to redirect the output of one command into a file. Similarly, Linux offers us the ability to take the output of one command and **pipe** it into another command using the " | " character. For example:

```
>> ls /usr/bin | more
```

Here, we list the contents of ls, but instead of printing all of the files to the screen in one go, we are piping the output into the more command. This causes the output to be printed page-by-page as above.

Piping is an extremely powerful feature of Linux, and this simple example doesn't do the command justice. Piping can be used to do simple tasks like viewing output easier and to do some more complex tasks like perform data analysis operations.

2.15.4 grep

grep is an extremely useful command that searches a file or files for text. For example let's say created a program to solve laplace's equation but I can't remember the name of the file. So, I want to search my directory for python files that contain the word "laplace"

```
>> grep -Hi laplace *.py
```

In this context, the grep command has two required arguments, the string to search for (laplace) and where to search (*.py). This command tells Linux to search all the files in the above directory that end with ".py" for the word laplace. The result will be all the lines of text, all of which include that word. I've also included two optional flags: the -H flag tells grep to return the name of the file that the word appears in and the -i flag causes grep to ignore case sensitivity. grep is obviously useful for searching files, but it can also be useful attached to other commands:

```
>> ps aux | grep -Hi 'username'
```

will tell you information about all the processes that are being run by you on this computer. There probably aren't many. If you replace your username by mine, dpawlows, you should see a few more.

2.15.5 ps

The ps command is useful when you want to see what processes are eating up computational resources, and if you need to kill an unresponsive process, provided you are the owner of the particular process. This is done using the kill command:

```
>> kill 'pid'
```

where 'pid' is the process id number, the number in the 2nd column of the ps output. For example, try:

```
>> emacs -nw &
```

Hit enter a couple times after that to bring the prompt back. You've just started the program emacs, a text editor, and then put emacs in the *background*. This means emacs is still running, but you can't interact with it until you bring it back to the *foreground*. Let's say you can't get it back into the foreground in order to close it. You can use ps and kill:

```
>> ps aux | grep 'username'
```

note the pid of the emacs process. Then,

```
>> kill 'pid'
```

You will need this because sooner or later, you will start a process and put it in the background, forget about it and log out of the remote computer. Since the process is in the background, it will not necessarily end. Later on, you can find it again and kill it using this command.

2.15.6 Finding files

The most useful command for finding any file on your system is the *find* command. find has a ton of options which may be useful, but the best uses are the following:

```
>> find . -name 'string' -print
```

will search the current directory (.) for a file called string. You can include wildcards in both the directory (replace . with any directory), and in the string (if you include the quotes). For example:

```
>> cd dpawlows/UpperAtmosphere
>> find * -name 'champ*' -print
```

will search my UpperAtmosphere directory for files starting with the string champ. If you include the grep command, find can find a file anywhere in the system that contains a string inside the file.

```
>> cd dpawlows/idl
>>  find * -exec grep -Hi 'champ' '{}' \; -print
```

will output the lines and files that include the word 'champ' in them. The syntax is complicated, but necessary. Note that this is useful if you really have no idea where the file is, but remember a specific word or phrase contained in it. Depending on how you use this, it can take a very long time and search a lot of stuff. For this reason, the *locate* command can be better, if it is available.

2.15.7 Sort

The sort command is a easy to use utility that can quickly sort data that is stored in a file in a table format. Sort is pretty powerful, but knowing the basics is usually enough to make quick work of examining a dataset. Let's say you have the following data in a file called fruit.dat:

Banana	5.2	1001
Apple	32.1	1002
Pear	101.8	1003
Grape	7.5	1004
Melon	47.2	1005

We can quickly sort the data using sort:

```
>> sort fruit.dat
```

You should see the data has been sorted by the first column in the file. Easy enough. However, there are a couple flags that are needed to really use sort. First, what if we want to sort based on the 2nd column?:

```
>> sort -k2 fruit.dat
```

Note that something happened here, but maybe not what was expected. Sort treats all data in a file the same, as a character string. So in this example, sort looked at the data an put the string that started with a "1" first. But most likely we want that data to be sorted as if column 2 was numerical data. To do that, we need another flag:

```
>> sort -nk2 fruit.dat
```

That should give the expected result, with the banana row at the top of the output and Pear at the bottom.

Don't forget that it can be useful to combine a sort with redirection to send the output to a new file:

```
>> sort -nk2 fruit.dat > fruitsorted.dat
```

2.15.8 Linking files

It is often useful to link two files together. For example, try:

```
>> cd
>> ln -s dpawlows/Public/Phy380/ Phy380
```

This will create a **soft link** in your home directory that points to the Public Phy380 folder. Now, if you want to cd into that directory, you just have to do

```
>> cd /Phy379
```

and if you want to copy a file to that directory or a subdirectory of that directory, it's quick and easy

```
>> cp homework1 /Phy380
```

This works even though the folder that Phy380 is linked to isn't in a folder owned by you.

2.16 Lesson Overview

Having learned several types of Linux commands, it's now time to put them to real use in the form of a shell script! This week we discuss the basics of shell scripting as well as how to edit text from the terminal and how to modify the permissions of your files so that you can actually run your scripts.

Specifically, you will learn about:

1. *the EMACS text editor*
2. *file permissions and how to modify them*
3. *the basics of shell scripting*
4. *startup files*

2.16.1 Learning Goals

After this week you should be able to:

- Edit text from the Terminal using emacs.
- Determine the permissions for your files for the owner, group and other users.
- Modify file permissions.
- Write basic bash scripts and execute them.
- Modify your bash startup file to set some aliases and modify other shell features.

2.17 Editing Text

A computer would have limited use if we could only interact with it by typing commands into the command line. Instead, most of the power on a computer comes from being able to store commands in a file and to execute that file multiple times. However, remember that for early computers, the only way to interact with the machine was via the keyboard. This meant that there needed to be software programs that were accessible via the Terminal that could be used to write and edit text.

These **text editors** are fundamentally different than the **word processors** that have become ubiquitous. While such software, including Microsoft Word and Google Docs, is incredibly useful, they can't be easily used to write computer programs or scripts because what you type is not actually what gets saved in the file that you create. Instead, text editors are fundamentally what-you-see-is- what-you-get (WYSIWYG); the software doesn't add formatting rules or notes to your file, or attempt to automatically apply certain rules as you are writing.

Linux computers always come with a variety of basic text editors. There are wars over which of these is superior, but typically, the one that you learn first is the one that you will prefer. For me, and likely for you, that one is **Emacs**.

2.17.1 Emacs

Emacs is a free, portable, and extensible text editor that you will find on pretty much every Linux based system in existence. It works on a variety of architectures and other operating systems, and as it is quite portable, no matter what system you are on, you will find that the behavior of emacs is always the same. Emacs is extremely popular with programmers. If you use a common programming language, Emacs probably provides a mode that makes it especially easy to edit code in that language, which provides context sensitive indentation and layout, colors, and even the ability to run programming sessions, compile programs, debug your code, and interact directly with the language interpreter inside the Emacs editor itself.

Emacs is much more than a WYSIWYG editor. However, we will be using it mostly in this capacity. A fantastic description of emacs resides at the GNU Emacs website.

Notation

Emacs was designed to run on systems that did not have access to devices other than the screen and keyboard, i.e. no mice. Therefore, the only way to save, open, close, delete, move the cursor, etc. was by using the keyboard. For this reason, there is an extensive list of keyboard commands that will allow you to do everything that you need to do when editing text using the keyboard. Every keystroke in Emacs is a command. Typing the letter "A" is a command to insert the letter "A". There other commands that do not result in a character being printed to the screen. The standard notation that describes these keystrokes follows:

C-x

For any x, the character Control-x.

M-x

For any x, the character Meta-x. The Meta character is usually the escape key as few keyboards have the Meta character.

C-M-x

For any x, the character Control-Meta-x.

RET

The return key.

SPC

The space bar.

ESC

The escape key.

Many, but not all, commands in emacs can be performed using some combination of the keystrokes above. However, every command has a long name, like kill-line or deletebackward-char. Most typical commands are bound to keystrokes for convenience. This is called key binding. One of the great things about emacs is that it is possible to create custom key bindings, or change the default ones to suit your preferences.

2.17.2 Starting emacs

To get acquainted with emacs, lets start by opening the editor by typing:

```
% emacs temp.txt
```

at the Linux command prompt. The file temp.txt doesn't have to exist, it will create one for you and you should find yourself working within the emacs text editor and not the command line.

There are just three Emacs commands you absolutely need to know. To start, type some stuff in the editor. Once you do that, you'll actually want to **save** your work. So enter:

```
C-x C-s
```

to accomplish this. Note that is two separate key strokes. You type control-x then let go and then type control-s in succession.

Obviously, there will be a time when you need to **undo** a keystroke. This is accomplished using the

```
C-x u
```

key binding.

If you were done editing the file at this point, you would want to **exit** emacs and return to the Terminal. To do this you use:

```
C-x C-c
```

Note that if you enter this command without saving your work first, emacs is smart enough to know that and will ask you if you want to save (in the minibuffer). In that case, typing 'y' or 'n' will allow you to proceed.

When you execute a command, for example `C-x C-s`, emacs will tell you that you just did something at the bottom of the window in the part of the windows called the **minibuffer**. When you open a file in emacs, you open what is referred to as a buffer. You can have many buffers open at once, which is nice as they are easy to switch through. The minibuffer tells you information about a command that you may have entered, provides a space to enter longer, more complicated commands, and gives you options when working with certain commands.

2.17.3 More basic usage

Now that you saved a file, lets open another one. From within emacs, type:

```
C-x C-f
```

In the minibuffer, emacs wants you to specify a file. By default it sets you up to create a new file in, or open a new file from, your current working directory. In the minibuffer, type:

```
temp2.txt
```

Now the active buffer should be a file called temp2.txt. You have opened 2 files in emacs and it is easy to switch between them:

```
C-b
```

This tells emacs that you want to switch to a different buffer. Note this is not the same as opening a different file. The file already has to be loaded in the emacs's memory. The default is the last buffer that you were working with. Since temp.txt is the only other file that we've been working with, that's the only other option:

```
Ret
```

Now temp.txt should be the active buffer. Open up a few more temporary files, temp3.txt, and temp4.txt. Now lets switch back to temp2.txt:

```
C-b
```

Note that in the minibuffer, the default is not temp2.txt, since that was not the last one we worked with. You can just type temp2.txt in the minibuffer to access that one. Even better though, type:

```
temp2
```

in the minibuffer (leave off the .txt), and hit the tab key. Tab completion works in emacs just like in the shell. If there are multiple options based on the letters that you entered, then you will get a list of the possibilities.

If you do something that gives you access to the minibuffer, like type:

```
C-x C-f
```

to open a file, and then decide that you don't want to actually do that. You can quit the minibuffer using the:

```
C-g
```

key binding.

2.17.4 Other useful commands

When programming especially, it is useful to be able to delete text quickly without having to hit backspace for every character.

```
M-d
```

will delete the word directly after the cursor, and

```
M-delete
```

will delete the word preceding the cursor. Also,

```
C-k
```

will delete everything on a single line after the cursor. When doing any of these deletions, emacs automatically copies the word/words that were deleted into memory, and you can then paste them elsewhere using:

```
C-y
```

This is especially useful, especially for programmers when you often have multiple lines that are identical, or very similar. Instead of re-typing the entire line, you simply go to the beginning of the line, type:

```
C-k
C-y
C-y
```

and you will now have a copy. See, much quicker. Lastly, if you just want to move the cursor, but don't want to have to hit the arrow keys over and over again, there are a few commands to do that too:

```
M-f
```

will move the cursor one word forward.

```
M-b
```

will move it one word backwards.

```
C-v
```

will move the cursor down one page

```
M-v
```

will move it up one page.

2.17.5 Summary

For now, those commands will get you using emacs, and doing so more efficiently than a lot of people. Utilizing the word/line cutting key bindings and pasting along with being able to move the cursor through text quickly can really speed up the text editing process. And to be honest, it is remarkable how much time is saved by not having to rely on the mouse to do a lot of things. One last note. There are literally thousands of command options in emacs. These are just the basics of the basics. Many of the commands need to be bound to keys manually. This is done using a file that is read when ever emacs is started up, called .emacs (remember dot files!). This file must be created by you, and must reside in your home directory (where all startup files typically go). The syntax for adding or changing commands is a bit cryptic, which is why its usually just easier to google whatever it is you want to do and find someone that has done it before. A good place to start for some useful customization is my .emacs file, which you are welcome to copy and use however you want.

2.18 File Permissions

On a linux system, files can have different permissions for different users. A given file may be readable, writeable, or executable (or none of these things). In order to execute a command, it must be executable. If you run:

```
ls -l
```

and look at a given file in the list, you will see something like:

```
-rw-r--r--
```

in the left most column. There may be more than 10 characters printed, but we only care about the 1st 10. The first character, a hyphen in this case, tells us the file type. Generally, there are two options here: a regular file (which gets a hyphen) or a directory, in which case we would see a `d`.

Then, there are **three sets of three** characters. Each set describes the file permissions for three different groups of users. The first set are the permissions for the user (u) that owns the file (the owner is the third column in the output from ls -l). The 2nd set of three characters are the permissions for the group that the file belongs to (4th column in the ls -l output) and the 3rd set is for everyone else (o for other) that has access to the computer.

New files that you create are assigned, by default, read/write (rw) permissions for you (the owner) and read (r) permissions for the group and other. If you want to change the permissions of a file we use the command chmod.

chmod takes at least 1 file as an argument, as well as a sequence of letters or numbers that specify what the permissions should be. There are several different ways to assign permissions. In order to change the permissions for the u that owns the file, you would do something like this:

```
% chmod u=rwx file
```

Note that only the owner of a file, and root, can change the permissions of a file. This above example tells Linux to give the user (u) that owns the file read (r), write (w), and execute (x) permissions. Similarly, you can change the group's permissions:

```
% chmod g=rx file
```

gives all members of the group (g) that the file belongs to read and execute permission. Finally, you can change the permissions for everyone else:

```
% chmod o=r file
```

gives read only permission to other.

Chmod has a few shortcuts worth noting- let's say that you wanted to give everyone permission to do anything to a file:

```
% chmod ugo=rwx file
```

Note that we just combined the 3 types of users. Careful with that one, it means everyone with access to the computer can do whatever they want to the file, including delete it. Alternatively, we could have done

```
% chmod a=rwx file
```

where a stands for all types of users. You can also add permissions to the existing ones by

```
% chmod u+x file
```

This says to take the current permissions and add executable permission for the owner.

Finally, you may see usage examples where permissions are specified with a number sequence, such as:

```
% chmod 751 file
```

This is a shorthand which specifies the permissions for each of the groups separately, each number corresponds to a different user, e.g. u gets the number 7, g is assigned 5, and o assigned 1. The numbers range from 0 - 7 and are derived by adding up the bits with values 4, 2, and 1. Where 4 represents read permissions, 2 write, and 1 execute.

So in the example above, 7 = 4+2+1 means u gets rwx permissions, 5 = 4+1 means g gets rx, and 1 means o gets just x permissions. This is the quickest way to assign different permissions to all user classes.

2.19 Shell Scripting

At this point, you've been introduced to a several Linux commands and while they are useful on their own, what if you needed to execute a series of commands over and over again? It would be really annoying if you had to type each one of them every time you needed to use them, and not much better if you used the history feature of Linux. As it turns out, there is an easy way to handle repetitive tasks. Create a script! A script is a small program that executes one or more Linux commands automatically. Since we are creating scrips using shell commands, we call these shell scripts. Being able to create shell scripts quickly can be a really useful tool that makes doing routine tasks that may take a long time if done using only the command line very quick.

2.19.1 A first script

Let's start off by making a script that prints a message to the screen. Open up a file in emacs called first.sh. The .sh suffix doesn't really mean anything, other than it would be an indication that this file is a shell script. Add the folllowing lines to the file:

```
#!/bin/bash
#This is a comment!
echo "Hello World"
```

The first line of the script tells Linux the shell that should be used to interpret the script. In this case, we are telling Linux to use /bin/bash, or the Bourne Again shell. This is the shell that you use normally in the command line, but including this line tells the kernel which shell to use even if you aren't using bash.

The second line of that script begins with a special symbol: #. This marks the line as a comment, and is ignored by the shell. The only exception is when the very first line of the file starts with #!, like ours. Then Linux knows to interpret it as described above. The third line runs a command: echo, with one argument, the string "Hello World".

Now we are ready to run the script. To do this, we need to exit out of emacs and work in the command line. But, before using C-x C-c to exit emacs, we can use another Linux shortcut.

2.19.2 Background/foreground

Instead, type:

```
Control-z
```

This keystroke is Linux for "put this process in the background". This is analogous to setting emacs off to the side while you work on something else. The nice thing about putting a process in the background is that you can come back to it later and it will be in the exact same state that you left it in. This is called putting a process in the foreground. To do that, in the command line type:

```
fg
```

and press return. Now, you should have your emacs window back. One more thing, it is possible to put multiple processes in the background at a time. `fg` will bring the last process that you put in the background to the foreground.

So, if you have multiple processes in the background and want to bring a different one up, you need to use is's job number. Put emacs in the background again and type:

```
jobs
```

in the command line. This command tells you about the processes, or jobs, that you have running in your current shell session. If you only have the 1 emacs session running, you will see it has an id of 1. If there were more jobs, they would be listed here, each with a unique job number. You can use the job number to bring a job to the foreground like so:

```
fg %n
```

where "n" is the job number. With your one processes, `fg %1` should bring emacs back to the foreground.

2.19.3 Running your script

Ok, with emacs in the background, we are close to being able to run your script. Try to execute it like you would any command, by typing the command name:

```
% ./first.sh
```

The "./" tells Linux to search the current directory for the command. Upon entering this command, you should get an error. The reason is that Linux doesn't know that this is an executable file. Before you can execute this command you need to change the **permissions** of the file to be executable.

So back to our example. Give yourself permission to execute the file and then try to run it again. You should see that the shell outputs the words "Hello World" on the next line after the call to the script. That's a pretty simple script, but it does something. If you haven't created one before, congrats- that's your first program. Now, let's play around with scripting a little.

Create a new file called first2.sh:

```
#!/bin/bash
# This is a comment!
echo "Hello World" # This is a comment, too!
echo "Hello World"
echo "Hello * World"
echo Hello * World
echo Hello World
echo "Hello" World
echo Hello " " World
echo "Hello "* " World"
echo 'hello' world
echo `hello` world
```

The last line uses the backtick charater which is below the escape key (not the single quote).

Change the permissions and run it. It's ok if you get an error or two. Try to make some sense of whats going on with each line, and if you can't figure out every line, that's ok, we'll cover all the differences eventually.

2.19.4 Variables

Pretty much every programming language out there uses the concept of variables: a symbolic name for a chunk of memory to which we can assign, read, and manipulate its contents. Scripting with the shell is the same. You may be familiar with a few variables already, such as `user`. Lets play with the ability of Linux to be able to handle variables. Open up another file, call it var.sh or something, and enter the following:

```
#!/bin/bash
my_message="Hello World"
echo $my_message
```

This assigns the string "Hello World" to the variable my_message, then echos the value of the variable. The shell does not care about the type of variable used. It can store strings, integers, real numbers, etc.

Note: The shell is picky about spaces. This won't work if you put spaces around the equals sign.

Note how the shell refers to variables. The third line of the above script includes the $ character. If you remove this, then the echo command won't know that my_message is, in fact, a variable, and will simply write the word "my_message" in the terminal. Requiring a special character to signify that a certain word is a variable is not typical of most programming languages any longer, but it is the case when using the shell. It is possible to interactively set variable names using the read command:

```
#!/bin/bash
echo What is your name?
read my_name
echo "Hello $my_name! I hope things are going well."
```

Certain programming languages, like C and Fortran require you to declare variables and their type before you use them. This is not the case in most shells. What this means is that if you try to reference a variable that has not been set, you will get an empty string, and not an error. For example try running this:

```
#!/bin/bash
my_variable=500
echo $my_var
echo $mv_vra
```

You should see that the second echo doesn't actually do anything, since the variable name is misspelled. This can be a tricky source of bugs in your code, because in most programming languages, trying to print a variable that hasn't been set would give you an error.

2.19.5 Scope

If you were to set a variable in the command line and then try to use it in a program it wouldn't work. Try it. First create a simple script, myvar2.sh:

```
#!/bin/bash
echo "my_var is: $my_var"
my_var="hi there!"
echo "my_var is: $my_var"
```

Once you've done that, go back to the command line and run the script. You will see that the first echo gives you nothing, and the second spits out the string "hi there'!" Now, try setting the variable in the command line (not in the script):

```
% my_var=”hello”
```

and test that it is set using echo in the command line. Next run the code:

```
% ./myvar2.sh
```

The output should be the same! The reason is that when you set a variable, as we have been doing, the variable by default is local to the current shell. When a command is executed, Linux actually spawns a separate shell to run the command. If we want to make the variable available to all sub-processes started by the current shell, we want to change the **scope** of the variable. This is done in bash using the `export` command. So, execute this in the command line:

```
% setenv my_var hello
% ./myvar2.sh
```

Now you should see the variable is set already when we call our script.

The concepts covered here are just the basics of shell scripting and enough to get you started. Of course, we can do a lot more with scripts, including looping, logic, etc. We'll save that stuff for another lesson.

2.20 Startup Files

Every time a new shell session is started, Linux attempts to read several files that set certain variables, options, and other things that the system needs or makes your life easier. These are referred to as startup files. The first startup files that are read are buried in the /etc/ directory and set things for all users. After that, Linux reads an individual's startup files which are located in your home directory.

In your home directory, if you execute:

```
% ls -a
```

you will may see one or more files that start with '.'. In particular, the shell will try to read files called .bash_profile (for bash, .cshrc if you are using csh, etc.) and .login to name a few. In addition, certain programs may have their own "dot" file. For example, when emacs starts up, it looks for the file, .emacs. These files don't have to exist, but if they do, Linux will read them and execute whatever commands are in them. In this tutorial we will just talk about .bash_profile and .emacs in brief. There are endless possibilities for what you can put in these things. A good place to start is to just google ".bash_profile" and copy what someone else uses. The point here is to introduce you to a few common options, and show you how to use these files.

2.20.1 .bash_profile

.bash profile and .login basically do the same thing, except that .bash_profile is read first, and .login is only read in the login shell. I actually don't know anyone that uses .login, but I'm sure there's a use for it. .bash profile is typically used to setup variables that the system may or may not use. Aliases are one such example; they are variables that are basically shortcuts to execute commands. For example, let's create an alias for the command ls -l using the command line:

```
% alias ll='ls -l'
```

Here we are setting a variable called ll as an alias to a useful command. Note that there are no spaces surrounding the equal sign. Again, bash is picky about that. After hitting enter, type `ll`. You should get a long list of all the files in your current working directory. Doing this is quite useful, but defining an alias this way means that the alias is only good for your current shell session. If you startup another one, i.e. by logging into a second putty or ssh session, you'll notice `ll` is no longer defined.

They way to get around this is by putting the alias in a startup file that is executed each time you start a shell session, such as .bash_profile. Here is an example of a basic .bash profile file.

```
#!/bin/bash
# Sample .bash_profile
history=200
export EDITOR="emacs"
export PATH=".:~/bin:/usr/local/mpi/bin:/usr/local/bin:
/opt/local/bin:/opt/local/sbin:$PATH"
PS1="\u@\h: \W >> "
alias emcas="emacs"
alias emasc="emacs"
alias emcsa="emacs"
alias h="history"
alias j="jobs"
alias l="ls"
alias ll="ls -l"
alias la='ls -a'
export term="xterm-color"
export CLICOLOR=1
export LSCOLORS="cxfxcxdxbxegedabagacad"
export PYTHONPATH="~/Programming/Python"
```

This .bash profile file sets a few variables, a few aliases, and changes the look of the prompt. The first variable, `history`, is used by the history command. This sets the default number of commands for history to remember. The second variable, `EDITOR`, is used by the system whenever Linux wants you to edit text. Linux needs to know that you prefer to use emacs, like a good human. In this case, `EDITOR` is set as an environment variable via the use of the `export` command, which means that its scope extends beyond the current shell. Any program that is called by the shell will inherit the value of EDITOR.

Next, the path is set. `PATH` stores information telling Linux where to look for executable files. This way, you don't have to actually be in the directory where the executable is in order for it to work. Linux wouldn't work very well if you had to copy `ls` to every directory that you wanted to list files in.

After that, we define a new prompt. Mine is sort of complicated, but it basically prints out username@hostname:'current working directory'>>.

The next lines setup a few handy aliases. I misspell emacs a lot. Finally, the last lines tell the shell that you want files and directories to be color coded when you list them using ls. You can assign any color you want to different types of files. That's what that long line that looks like a bunch of garbage does. Take a look at the `ls` man pages to figure out how to change the colors to your preference. There are lots of other options that you can put in the startup files, but these will get you started.

2.20.2 Reloading your startup file

When you make changes to .bash_profile, you don't have to start a new shell session to make the changes take effect (most of the time). Instead, you can use the command:

```
% source .bash_profile
```

or if you aren't in your home directory:

```
% source /.bash_profile
```

because remember, .bash_profile should go in your home directory! This command re-executes the file.

2.20.3 .emacs

When you start an emacs session, emacs will look for the file .emacs in the home directory and execute it if it exists. When we first talked about emacs, I mentioned that you can define your own commands by binding them to a key sequence. This is where you would do that, so that those key-bindings automatically exist each time you use emacs. Below is a sample .emacs file.

```
;; Set up the keyboard so the delete key on both the regular keyboard
;; and the keypad delete the character under the cursor and to the right
;; under X, instead of the default, backspace behavior.
(global-set-key [delete] 'delete-char)
(global-set-key [kp-delete] 'delete-char)

;Use C-l to go to a specific line number
(global-set-key "\C-l" 'goto-line)

;;use C-t to start spell checking
(global-set-key [(control t)] 'ispell-buffer)

;; Enable wheelmouse support by default
(cond (window-system
(mwheel-install)
))

;; Visual feedback on selections
(setq-default transient-mark-mode t)

;; Always end a file with a newline
(setq require-final-newline t)

;; Turn on font-lock mode(language specific colors and
;; Settings) for Emacs
(cond ((not running-xemacs)
(global-font-lock-mode t)
))
```

The syntax here is especially cryptic. I recommend googling the feature that you are looking for to find examples on how it is done.

2.21 Lesson Overview

This week, we cover a few different topics. We will continue to enhance our linux command toolbox while also digging in a bit deeper into shell scripting. Then, we will start to learn about techniques for solving algebraic equations using a computer.

Specifically, you will learn about:

1. *more linux commands!*
2. *more shell scripting!*
3. *root finding techniques*

2.21.1 Learning Goals

After this week you should be able to:

- really know how to do some cool things in Linux.
- write bash scripts that can actually be useful.
- solve some basic algebraic equations numerically.
- discuss topics that are important to numerical methods such as round-off error, stopping criteria, iteration, and how numbers are stored on a computer.

2.22 Basic Linux Commands 3

There are many, many common commands that you will find on any Linux system. You've been introduced to several, here are a few more that will help you become a more skilled Linux user.

2.22.1 gzip filename

gzip is a command that will "zip" up, or compress, a file so that it takes up less space. Typically doing this will make the file take up half as much space.

2.22.2 gunzip filename

This will unzip a file that has been zipped.

2.22.3 tar

tar is a command that lets you package up and compress several files into a single "file". There are many flags that you can use with tar, but we will only deal with the most common here. First, copy dpawlows/Public/Phy380/images.tar to one of your directories.

Typically, a set of files that have been tarred together is referred to a "tarball". You can extract the files from the tarball using:

```
% tar -xvf images.tar
```

The -v flag is common among many Linux commands and stands for verbose, i.e. it will tell you what is going on. Here, you get a list of the files that were extracted. If you

```
% ls
```

you should see that, indeed, those files were extracted. The tar command can be used to extract files from a tarball as well as create a tarball. All you have to do is use the -c flag instead of the -x flag to create:

```
% tar -cvf allimages.tar *.ps
```

will create a new tarball called allimages.tar that contains all of those images (plus any other files that end in "ps" that you might have in your directory). Note the order of the arguments, the file to be created goes first, and the files to tar up second. Also, note the use of the wildcard.

In addition to packaging all the files together, it is also possible to compress the files:

```
% tar -czvf allimages.tgz *.ps
```

Will create a zipped tarball. Again, changing the -c to -x will extract the files from an existing tarball. As with any file, the extensions do not mean anything. .tar and .tgz are typical extensions given to a file that has been tarred and tarred/compressed, but you don't have to use those extensions. The common extensions do help let other people know what to do with them if they are given such files. Also, certain GUI programs need to have the correct extension in order to work.

2.22.4 scp

You should know how to login to a remote unix system using ssh, but what if you want to transfer a file? scp allows you to do this securely. The syntax is as follows:

```
scp [username]@hostname:'path of file to transfer' [username]@hostname:'path to␣
↪transfer to'
```

The first argument contains information about the file that you want to transfer, and the second argument contains information about where you want to transfer to. You should, in general, include the quote marks on whichever machine is the remote machine. "[ ]" indicated that username optional: you only need specify your username if it differs on the remote computer.

For example, if I wanted to send a file to a computer called **yipe**, I would do:

```
% scp images.tar dpawlows@yipe.emich.edu:'/'
```

This will copy the file to my home directory on Yipe.

To get a file from yipe:

```
% scp dpawlows@yipe.emich.edu:'/scripts.tar' '.'
```

will bring the file to my local machine and put it in my current working directory. If you want to transfer an entire directory, simply include the -r flag.

```
% scp -r /Documents/ yipe.emich.edu:'/'
```

will transfer my Documents directory and all of its contents to yipe. Note that I don't actually need to include my username if it is the same on the remote system as it is on my local machine.

2.22.5 awk

awk is an extremely powerful Linux utility that has entire books devoted to it. awk is a tool that is used to work with regular expressions, or a pattern of characters used to match the same characters in a search through text. The pattern can include special characters to refine the search, as well as do a bunch of other stuff. If you end up writing lots of shell scrips which manipulate text, learning regular expressions is well worth the time.

In this context, we are just going to use `awk` for a couple things. The first will be obtaining the length of a string. In the command line, execute the following two lines:

```
% string='Processing NGC 2345'
% nchar=`echo $string |awk '{print length($0)}'`
```

Single quotes surround the `print` statement, but backticks (shift - tilde) are also used on the second line surrounding the entire `echo` command. Those are special quotes that allow the execution of commands within the string itself. Basically, Linux sees that whole thing as a string, except it checks for commands and executes them if one or more is found. Now test to see if those lines worked:

```
% echo $nchar
```

This should tell you the length of the string that you entered.

It is also possible to determine the position of a substring within another string using awk:

```
% catalog='NGC '
% number=2345
% object="Processing $catalog$number."
% index1=`echo $object |awk '{print index($0,"NGC")}'`
% index2=`echo $object |awk '{print index($0,"2345")}'`
```

The index function for awk searches a string for a substring. In this case, we are using a pipe before the awk command so the *$0* "argument" tells awk to use whatever went into the pipe as input.

Finally, we can use awk to extract a substring from a larger string:

```
% littlestring1=`echo $object |awk '{print substr($0,12,8)}'`
% littlestring2=`echo $object |awk '{print substr($0,16)}'`
```

Compare $littlestring1, $littlestring2, and $object to see what awk is doing here. The numbers in the substr method of awk refer to the positions of a character in the larger string. It is useful to use the index method combined with the substr method: index can help you find the position of a substring, while substr will actually extract it.

There is much, much more that you can do with awk. For example, you can split up a string into an array, change the case of a string, and substitute one substring for another. The tools that we have dealt with are good enough for now though. They can be quite useful in scripts, especially for dealing with files and directories.

2.22.6 cmp

If you have two files that you want to compare and make sure that they are identical, the `cmp` command will do that for you.

```
% cp first.sh firstcopy.sh
% cmp for1.sh forcopy.sh
```

will return absolutely nothing, since those two files are definitely identical. However, if you do have two different files

```
% cmp first.sh second.sh
```

will tell you that the two files do indeed differ, and point you to the first differing occurrence.

2.22.7 diff

If you actually want some detail about how two files differ, then use diff:

```
% diff first.sh second.sh
```

This will tell you every line that differs between two files, with the less than and greater than symbols referring to the first and second files respectively.

2.22.8 w

Tells you who is logged in and what they're doing.

2.22.9 last -1 username

Tells you when the user last logged on and off and from where. If you leave off any arguments, you will get a list of everyone's login info.

2.22.10 write username

Allows you to write short messages to another user. This is often disabled by default.

2.22.11 du directory

Shows the disk usage of the files and directories in directory.

2.22.12 df

Shows the disk usage of all the disks connected to the system.

2.22.13 quota -v

Shows your disk quota (if you have one). If you have access to a public system, you will have a limited amount of disk space that you are able to use.

2.23 More Shell scripting

In this lesson, we will cover the a few topics that are essential to make shell scripting useful.

2.23.1 Escape Characters

Sometimes it is necessary to be able to write out or use a character that is reserved for use by the unix system. For example, at the command prompt, try echoing an asterix:

```
% echo *
```

Linux won't know what you are trying to do and it will basically list all files in your directory. If we want to actually print a "*", then we need to "escape" the special meaning from the character. This is done a few ways, but most easily using the forward slash "":

```
% echo \*
```

If you are writing a command and you get a weird error on a specific special character, you may need to escape that character. In linxu quotes do this for an entire string. Compare:

```
% echo "Hello * I am happy to be here\ Today is a great date"
```

to

```
% echo 'Hello * I am happy to be here\ Today is a great date'
```

However, beware! there is a difference between single and double quotes. Compare:

```
% echo "Hi $USER"
```

to

```
% echo 'Hi $username'
```

Double quotes allow for variables to be used within a string where as single quotes escape the variables so that literally what you typed is used by the command.

2.23.2 Loops

Most languages have the concept of loops: If we want to repeat a task twenty times, we don't want to have to type the same code twenty times. As a result, we have `for` and `while` loops in the shell. There are somewhat fewer features to loops than in other languages, but nobody claimed that shell programming has the power of C.

For Loops

For loops iterate through a set of values until the list is exhausted (note that we are using bash)

for1.sh:

```
#!/bin/bash
for i in 1 2 3 4 5
do
echo "Looping ... number $i"
done
```

Next, try this one:

for2.sh:

```
#!/bin/bash
for i in hello 1 * 2 goodbye
do
echo "Looping ... i is set to $i"
done
```

Try to understand what is happening here. If you don't see it right away, try it with out the *. Being able to use wildcards in your scripts makes life much easier.

While Loops

while loops are also very useful, but can often be a source of bugs in your code. If you don't use them correctly, you can often be stuck in an infinite loop. For example, try this in the command line:

```
% while :; do echo true; done
```

You'll have to hit C-c. This code will spit out the word true over and over again, forever, because the argument of the while statement (nothing in this case) always evaluates to being true.

So let's use a while loop properly:

while1.sh:

```bash
#!/bin/bash
input_string="hello"
while [ "$input_string" != 'bye' ]
do
echo "Please type something (bye to quit): "
read input_string
echo "You typed: $input_string"
done
```

While loops are often most useful for reading a file or document and doing something with that information. The ability to read in data is somewhat limited when shell scripting. That's ok though, as shell scripts are typically used to get ready to use data, not necessarily use the data itself.

Logic

One of the most important types of control statements in any programming language is the if statement.

if1.sh:

```
#!/bin/bash
echo $#
if [ $# -eq 0 ]
then
echo "There are no arguments"
fi
if [ $# -ne 0 ]
then
echo "There are arguments ($#): $*"
fi
```

Try running this script a few times, starting with:

```
% if1.sh
```

Then, add an argument or two. . . :

```
% if1.sh hello!
% if1.sh hello! how are you?
% if1.sh hello! “how are you?”
```

Make sure you understand what is happening here. Linux automatically sets certain variables when you run a script. In order to use if statements properly, you need to know how to relate 2 or more variables to one another. You do this by using operators. You've seen a couple of those already above, but there are more (note strings and integers use different operators):

```
Integer Comparison
-eq
is equal to: if [ "$a" -eq "$b" ]
-ne
is not equal to: if [ "$a" -ne "$b" ]
-gt
is greater than: if [ "$a" -gt "$b" ]
-ge
is greater than or equal to: if [ "$a" -ge "$b" ]
-lt
is less than: if [ "$a" -lt "$b" ]
```

(continues on next page)

(continued from previous page)

```
-le
is less than or equal to: if [ "$a" -le "$b" ]
<
is less than (within double parentheses): (("$a" < "$b"))
<=
is less than or equal to (within double parentheses): (("$a" <= "$b"))
>
is greater than (within double parentheses): (("$a" > "$b"))
>=
is greater than or equal to (within double parentheses): (("$a" >= "$b"))

String Comparison
==
is equal to: if [ "$a" = "$b" ] (note the whitespace between the =)
!=
is not equal to: if [ "$a" != "$b" ]
<
is less than, in ASCII alphabetical order: if [[ "$a" < "$b" ]]
if [ "$a" \< "$b" ]
Note that the "<" needs to be escaped within a [ ] construct.
>
is greater than, in ASCII alphabetical order: if [[ "$a" > "$b" ]]
if [ "$a" \> "$b" ]
Note that the ">" needs to be escaped within a [ ] construct.
```

In addition, it is possible to do file queries:

```
-e file file merely exists (may be protected from user)
-r file file exists and is readable by user
-w file file is writable by user
-x file file is executable by user
-o file file is owned by user
-z file file has size 0
-f file file is an ordinary file
-d file file is a directory
```

So, for example, you can test to see if a file exists, and if it is a directory:

filetest.sh:

```
#!/bin/bash
if [ $# == 0 ]
then
echo "You have not provided any arguments. You
must provide at least 1!"
echo "filetest.sh"
echo "Usage: filetest.sh filename1 [filename2,filename3,...]"
echo " "
else
for file in $*; do
echo "Testing to see if $file exists..."
if [ -e $file ]; then
comment="$file exists!"
else
comment="Sorry, $file does not exist"
fi
if [ -d $file ]; then
```

(continues on next page)

(continued from previous page)

```
comment2="and it is a directory"
else
comment2=""
fi
echo "$comment $comment2"
echo " "
done
fi
```

Try running this a few times and make sure you get what you expect. Note, you can enter any filename at all, as long as you include the entire path.

2.24 Root Finding

A common problem in scientific computing is to find the solutions, or roots, of some equation

$$f(x) = 0$$

in a single variable x. Numerical techniques are useful because we aren't required to come up with a general solution in a closed form, i.e. a function. Typically, a numerical solution is found by starting with approximation to the answer, p_0, and steadily improving the solution $p_1, p_2, \ldots$ until we have obtained a value that is within a certain error tolerance. There are a variety of methods for doing this. The best methods are ones that ensure convergence and do so with the fewest number of iterations.

So, what is meant by convergence? Given a sequence of approximations to a solution for p: p_0, p_1, p_2, p_3,... if the solution converges to the answer p, then the differences, or errors, $e_n = p_n p$ must get smaller and smaller as n approaches infinity. To put it another way, if we take the ratio of successive errors, it should be less than 1. In reality, as we perform more and more iterations, then this ratio should approach a non-zero constant that is less than 1:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|} = k < 1.$$

More generally, it is possible to have:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^{\alpha}} = k < 1.$$

where α is some positive power called the order of convergence. When $\alpha = 1$ then we have linear convergence. When $\alpha = 2$ then we have quadratic convergence, i.e. our solution improves much faster than with linear convergence.

Using such an iterative approach to problem solving means that n really could go to infinity, which really isn't practical for scientific computing. Instead, we define a tolerance, ϵ such that when $|e_n| < \epsilon$ we stop iterating and declare our solution good enough. This is called a **stopping condition**. Careful though, just because $|e_n| < \epsilon$ it is not necessarily true that our solution is with in ϵ of the unknown solution. It is always possible that the error could start growing again for one reason or another. For this reason it is helpful to have a physical understanding of the problem, or even to experiment with larger than normal values of n.

2.24.1 Roots of a Quadratic

One of the most common problems is finding the solution (roots) of a quadratic equation:

$$ax^2 + bx + c = 0.$$

For which the standard solution can be applied:

$$x = \frac{b \pm \sqrt{b^2 4ac}}{2a}.$$

Certainly this equation should produce the correct solution, and in fact there is no need to perform any sort of iteration. However, in general introducing a computer into the solution process can lead to one potential source of error: **round-off error**.

2.24.2 Precision and Round-Off Error

When we use a computer to store a number, the computer does this physically by utilizing a limited number of binary bits. Historically, floating point numbers (normal decimal numbers) take up 32 bits and double precision numbers take up 64 bits. Regarding precision: there is a limit on the number of significant digits that can be stored by the computer.

Most computers use the *IEEE 754* storage standard to represent numbers. In this manner, floating point numbers are stored as a combination of three parts: the **sign**, **mantissa**, and **exponent**. Consider as an analog a number displayed in exponential format, i.e.: 0.74723×10^5. In this case, the sign is +, the mantissa is 0.74723, and the exponent is 5. Single precision floats have roughly 23 bits reserved for the mantissa, 8 for the exponent, and 1 for the sign, while double precision numbers have 52 bits reserved for the mantissa, 11 for the exponent, and 1 for the sign.

What this all means is that a number that requires an infinite number of digits to be represented must be rounded to a finite number of digits. The problem isn't typically that the computer can't represent large enough numbers: single precision numbers can have exponents that range from ± 126. Rather, issues pop up when subtracting two numbers that have very similar magnitude. In this case, it is possible that significant digits may be lost. In the example of the quadratic formula, such errors can manifest themselves if the product of $4ac$ is much smaller than b.

It is important to note that round-off error is an issue with nearly every computer program. However, the precision of today's computers (python uses 64 bits for floating point numbers by default) typically makes round-off error less important than other sources of error, such as truncation error, and the most important source, user error.

2.24.3 Bisection Method

Say that you are given a function, $f(x)$ over the interval $[a, b]$, and at some point in the interval $f(x)$ changes sign. The bisection method is thus: our first step is to divide the interval in half and note that f must change sign on either the right half or the left half of the interval (or be zero at the midpoint of the interval). Next, the interval $[a, b]$ is replaced with the half-interval in which $f(x)$ changes sign and we repeat the process. We iterate by halving the interval and selecting the sub-interval that has a sign change until the sub-interval has a length of ϵ, or our tolerance

Note that since we are always reducing the interval by half, this method is linear in order of convergence. Additionally, as long as our function, $f(x)$ changes sign within our initial interval, $[a, b]$, the bisection method will converge, so it is an extremely reliable method, if not the most efficient.

The bisection method may or may not work if there are more than one roots within our interval. In particular, it will always work for an odd number of roots. If a given function has an even number of roots, it is necessary to choose the initial interval such that it only brackets an odd subset.

In order to implement such an algorithm, one would use the following steps:

1. Input the range, $[a, b]$
2. Input the tolerance
3. Test the ends of the range to see if they are one of the roots or if they bracket a root
4. Begin iteration
5. set $c = \frac{a+b}{2}$

6. if $f(a)f(c) > 0$ then $a = c$ else $b = c$
7. repeat until $abs(ab) < tolerance$

2.24.4 Newton-Raphson Method

In order to create an algorithm that is a bit more efficient than the bisection method, it is useful to have a little bit more information about the function. One such method is based on expanding our function $f(x)$ about the point $x = p$ as a Taylor series. If we do this and only keep first order derivatives, we get:

$$f(p) = f(x) + f'(x)(p - x) + \ldots$$

Since we got rid of higher order terms, we can't use this method to find an exact solution, but we can still get a pretty decent approximation. Solving for p in the above equation gives:

$$p = x\frac{f(x)}{f'(x)}.$$

This defines an iterative method, so if we start with an initial guess p_0 then we would have a method that looks like:

$$p_{n+1} = p_n\frac{f(p_n)}{f'(p_n)}$$

and we iterate over n until a sufficiently accurate value is reached. Unlike the bisection method that we already discussed, there is no guarantee that Newton's method converges. This is obvious for the case where $f'(p_n) = 0$.

Usually, Newton's method requires a that you choose your initial guess "close" to the actual solution. If this is done, the method converges quadraticaly. In addition, since there is no guarantee that our method converges, it is always a good idea to limit the number of iterations that your program can execute, less it runs forever. If the method doesn't converge by the time you reach some maximum number of iterations, you give up and try to find an alternative. Finally, in order to use Newton's method, you need to have knowledge of your function's derivative. Hopefully it is easy to find this analytically, as we have yet to discuss doing so numerically. Implementation of Newton's method should follow the following steps:

1. Input the initial guess p
2. Test if $f(p) = 0$. If it is, you're done!
3. Begin iteration
4. Calculate $f(p)$ and $f'(p)$.
5. set $p_{new} = pf(p)/f'(p)$
6. repeat until $abs(pp_{new}) < tolerance$ or $n > nmax$.

2.25 Lesson Overview

This week, we will learn how to do some calculus using a computer! Perhaps more importantly, you will how certain numerical schemes are developed so that you will be able to develop your own! Additionally, we will talk about the syntax behind reading data into python using some basic functions.

Specifically, you will learn about:

1. *python input and output*
2. *numerical differentiation*
3. numerical integration

2.25.1 Learning Goals

After this week you should be able to:

- use a Taylor series to develop a numerical differentiation scheme.
- understand the consequences of truncation error.
- understand the difference between first and second order numerical schemes.
- implement a numerical differentiation scheme.
- implement a numerical integration scheme.
- read from and write to a file from within python.

2.26 Python Input and Output

There are many methods to reading and writing input from/to a file in python. Here, we will cover the basics and emphasize an approach that will work for just about any programming language.

2.26.1 Reading data from a file

In order to read from a file, there are really 4 steps that you need to follow:

1. Open the file
2. Read the file
3. Parse the input
4. Close the file

Steps 1 and 4 are pretty simple and implementing them is basically the same for all circumstances. Steps 2 and 3 will depend on the particular problem that you want to solve.

Opening the file is done using the `open()` function. Given a file called "data.dat" we can open it with:

```
>>> f = open('data.dat','r')
```

Notice that the `open()` function requires two arguments- the name of the file that you want to open and a second string, 'r' in this case. The 'r' tells python that we want to *read* the file. We could have opted for 'w' if we wanted to *write* to the file, 'r+' if we wanted to read and write (update) to it, and 'a' if we wanted to read and append to it.

The other thing to note is that we saved the result of this function, which creates a python *file object*, to a variable, *f*. We must do that so that we can access the tools (or methods) of the file object that allow us to read the file, close it, etc.

One of the methods of the file object is `read()`:

```
>>> mydata = f.read()
```

This will read the entire file and store it as a character string in the variable *mydata*. I don't think that this is particularly useful, as parsing the data in that file can be a bit tricky.

At this point, if you try to read the file again:

```
>> mydata = f.read()
```

you will get an empty string. This is because python keeps track of the parts of the file that have been read already. Think of it as python setting a marker at the beginning of the file when it is opened, and then moving the marker as each part of the file is read. So, if the entire file has been read, python's marker will be at the end of the file, with no more data available to be read.

At this point, the only thing to do is to close the file:

```
>>> f.close()
```

The technique that I suggest to use when reading a data file is to read line by line. Python makes this really easy. Let's open the file again so we can start fresh:

```
f = open('data.dat','r')
for line in f:
   print(line)
```

Python treats a file object as an iterable! Further, each line is a single element of the iterable. So, we can use a for loop to iterate through the file and read it line by line.

At this point, all that is left is for us to parse the data. This is really where it comes down to the specific problem that you are working on. However, there are a few specific tools to be aware of.

First, when python reads a file, it stores the result as a string. You can verify that by using the type function:

```
type(line)
```

In physics, we often deal with numerical data that is organized in columns with in a file. For this reason, it can make sense to take that string and split it on a delimiter. For example, let's say you just read a single line and it has the following data:

```
2003 10 26 03 25 00 5.4 3.2 100 95.2
```

Upon reading, python stores all of that information as a single string, not very useful to us. Luckily, I can parse that string and turn it in to a python list using the `split()` method of the string object:

```
data = line.split()
```

This will result in a new variable called data which is a list that is made up of the data in the line variable. By default, `split()` will break the data up by whitespace. So our data list should have 10 elements. Note that if my data were separated by a different delimiter:

```
2003, 10, 26, 03, 25, 00, 5.4, 3.2, 100, 95.2
```

I can specify a delimiter when using `split()`:

```
data = line.split(',')
```

Splitting a string may be useful when reading data in from a file, but really, it depends on how the data is formatted in that file. Just know that there are many tools that come with the **string** datatype that can help you to parse your data, including the ability to search for a substring, extract a substring, etc. Additionally, at this point, you may need to perform some type conversion operations, store only part of the imported dataset, etc. This can all be handled within the for loop during the file read.

Finally, don't forget to close the file when you are done reading it:

```
f.close()
```

As a complete working example, imagine that I wanted to read a file called "euv.dat" that consisted of several lines formatted as above:

```
2003 10 26 03 25 00 5.4 3.2 100 95.2
2003 10 26 04 25 00 5.2 3.0 110 98.3
...
```

I only want to save information about the date and the 8th column of the dataset. The following code snippet should do the job, including type conversion:

```python
from datetime import datetime

filename = 'euv.dat'
f = open(filename,'r')
date = []
data = []
for line in f:
  temp = line.split()
  date.append(datetime(temp[0],temp[1],temp[2],\
    temp[3],temp[4],temp[5]))
  data.append(float(temp[7]))

f.close()
```

2.26.2 Writing to a file

Another method of a file object is write. Open up another file:

```
>>> g = open('somedata.txt','w')
```

Now we can use the write method to write to the file:

```
>>> g.write('Hello there!\n')
>>> g.write('This is some text...\n')
```

You need to include the “n” character if you want to start a new line. if you want to write something other than a string, you have to convert it to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> g.write(s+'\n')
```

Again, remember to close the file when you are done.

```
>>> g.close()
```

Once the file is closed, you can look at it in a text editor.

2.27 Numerical Differentiation

As any physics student knows, calculus is critical to solving most interesting problems. The implicit nature of calculus, which deals with the infinitely small, is a bit tricky for computers, which are fundamentally limited by the inability work with infinities. So, it shouldn’t be surprising that any techniques that aim to calculate derivatives using a computer are inherently approximations. That said, we have some control over the performance of our approximations.

Further, in the real world when an experimentalist takes data from some apparatus or system and wants to do analysis of that system, taking derivatives poses the same problem. After all, data isn’t continuous. Data is discrete, where each data point is separated by some "distance" in space, time, frequency, etc. So, numerical methods for calculus in general, and derivatives specifically, have existed for nearly as long as calculus. Our job is to implement those techniques on a computer.

2.27.1 Finite Differencing

While there are a few different approaches to taking numerical derivatives, we will use finite differences, which involves taking the difference between the values of a function at two points on some grid.

Analytically, the derivative of a function, $f(x)$, is given by:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}.$$

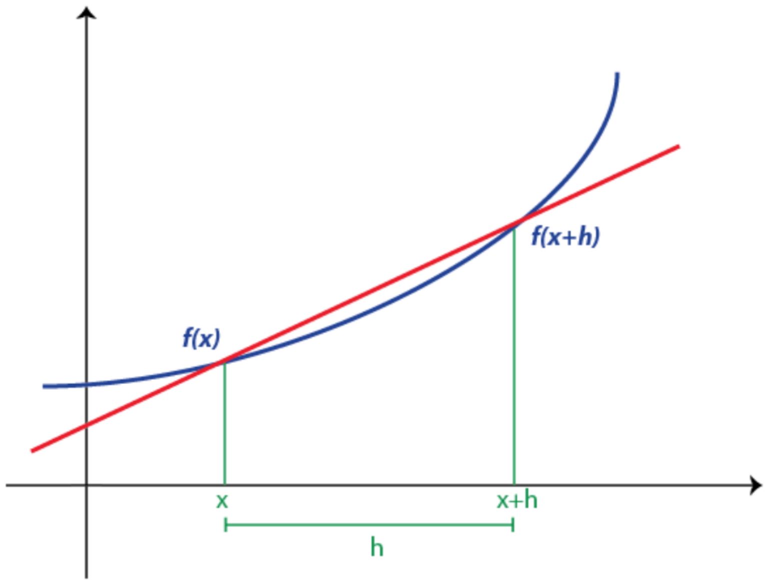
Computationally, we know that we can't let h go to zero as in that case, our solution will blow up. However, as we have discussed, computers are great at doing repetitive tasks very quickly. So that means we can make h small, and evaluate the derivative at many points on our grid that are very close together. Thus, in order to find the derivative of some function, we need to establish a grid, with grid spacing h. If our dependent variable is x, then our grid would generically be something like:

$$\ldots, x-3h, x-2h, x-h, x, x+h, x+2h, x+3h, \ldots$$

and since we know the function for which we want the derivative, we could evaluate the function at each of these points. Armed with this information, we can approximate the derivative:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{2.1}$$

Approximating the derivative in this way is logical. Essentially, we are assuming the function that we are given is linear between two successive points on our grid:



Clearly there will be some error, but as h gets smaller, we do a better job of approximating the true derivative.

2.27.2 Forward and Backwards Differencing

Eq (2.1) above isn't the only technique that we can use to approximate a derivative. To see other options, it helps to turn to our old friend, the Taylor Series:

$$f(x+h) = f(x) + \frac{f'(x)h}{1!} + \frac{f''(x)h^2}{2!} + \frac{f'''(x)h^3}{3!} + \dots \tag{2.2}$$

We can use this to approximate $f'(x)$ by keeping just the first two terms on the right side:

$$f(x+h) = f(x) + f'(x)h$$

which gives:

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

exactly what we have in eq (2.1). Approximating the derivative using this formula is called a **forward differencing scheme** because as it is written, to find the derivative at the grid point x, we use the forward value at $x+h$.

Alternatively, we could have chosen to find the derivative using **backward differencing**. That requires the Taylor series for the point at $x-h$:

$$f(x-h) = f(x) - f'(x)h$$

solving for $f'(x)$ gives $f'(x) = \frac{f(x)-f(x-h)}{h}$.

2.27.3 Truncation Error

You'll recall that we have already discussed one of the main sources of error for any given numerical scheme: *round-off error*. In deriving the forward (or backward) differencing scheme, we introduce another type of error, **truncation error**, named such because we *truncate* the Taylor series after the first 2 terms:

$$f'(x) = \frac{f(x+h) - h(x)}{h} + \mathcal{O}(h^2).$$

Here, $\mathcal{O}(h^2)$ tells that we have truncated terms that have an h^2 or higher. As such, we would consider this scheme to be "first order accurate"- only terms with an h to the first power are in our approximation.

The "order" of an approximation tells us how much better our solution gets if we make the grid spacing smaller. For a first order scheme, if we increase the step size by a factor of 2, we expect the solution to get 2x more accurate.

2.27.4 Central Differencing

That said, we can do better than a first order scheme. How to do that? Well, let's keep the 2nd order term in our Taylor series!

$$f(x+h) = f(x) + \frac{f'(x)h}{1!} + \frac{f''(x)h^2}{2!} \tag{2.3}$$

The problem is, I want to solve this equation for $f'(x)$ but if I do, that, I will have an $f''(x)$. In order to get rid of that term, I need be a little sneaky. Let's use the Taylor series about the backward point:

$$f(x-h) = f(x) - \frac{f'(x)h}{1!} + \frac{f''(x)h^2}{2!} \tag{2.4}$$

Then, I can subtract Eq (2.4) from Eq (2.3):

$$f(x+h) - f(x-h) = f'(x)h + f'(x)h$$

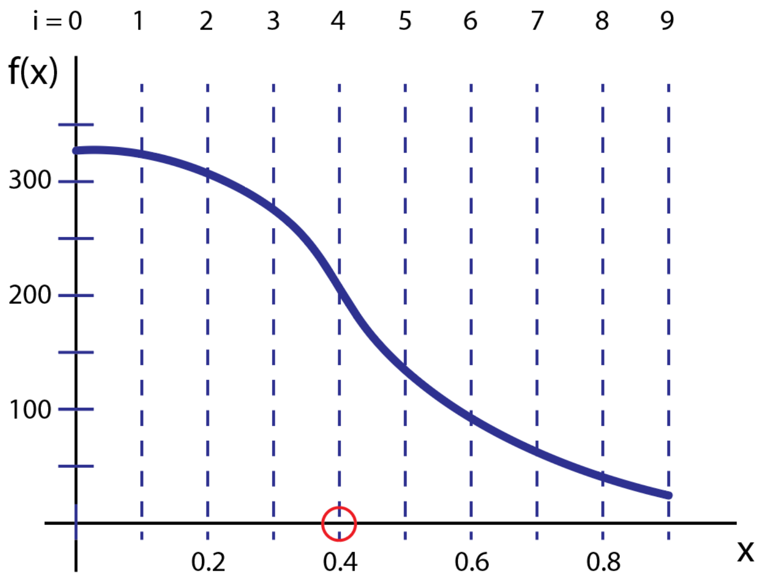
and solve for $f'(x)$:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^3). \tag{2.5}$$

This is called a central differencing scheme. We want the derivative at grid point x and to find it, we use the two grid points on either side $(x - h)$ and $(x + h)$. Keeping the 2nd order terms in the series means that this is a 2nd order scheme. But, that didn't really cost us anything as we were able to cancel those terms out by combining two Taylor series about two different points.

2.27.5 Implementation

The figure below shows a generic function as a function of x. Since our function only depends on x, it is inherently 1D. Units are arbitrary. You can imagine this line showing the velocity profile (vertical axis) of a car as it travels along a straight road (x axis).



In order to solve for the derivative of this function using a computer, we first break up the domain into discrete points, i.e. setup our grid. This is represented by the vertical dashed lines. For our purposes, there are two ways of referring to an individual grid point. For example, say I want to reference the grid point circled in red. I can either specify the value of the dependent variable at that location (0.4) or, I can refer to the grid point index (4), where the index is obtained by counting from the start of the grid (the origin). This is analogous to a list (or array) in python. Consider the line:

```
>>> position = numpy.arange(5,10.1,.2)
>>> print(position)

[ 5.   5.2  5.4  5.6  5.8  6.   6.2  6.4  6.6  6.8  7.   7.2  7.4  7.6
7.8  8.   8.2  8.4  8.6  8.8  9.   9.2  9.4  9.6  9.8 10. ]
```

This creates a numpy array that starts at value of 5 and ends at value of 10 with spacing of 0.2. For any given point I can refer to a position by value (say 5.4) or its location in the array (its index value). 5.4 is the 3rd element in the list, so its index is 2 since we start counting at zero. This may all seem obvious, but one of the biggest issues that students who are new to scientific computing have is understanding when they need to access an array's value or the index that corresponds to that value.

Back to the central differencing scheme. If I want to find the derivative of the blue line above numerically, I can implement central differencing as in Eq. (2.5). Let's say that I want to find the derivative of $f(x)$ on the interval x=[0,0.9] and I use the grid as shown. This means that I know the value of the function at 10 grid points. However, in order to find a derivative at some grid point using Eq. (2.5), I need to use the function evaluated at the grid point to the left and to the right. Thus, I can't determine the derivative at the first and last grid points.

To make this more explicit: let i stand for the index value of the grid point I am interested in. Then:

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2h}$$

where f_i represents the function value at the i^{th} grid point. As you can see, the lowest index value for which I can solve for f_i' is $i = 1$!

In this specific example, the $i = 0$ and $i = 10$ grid points (corresponding to values of x = 0 and x = 0.9) are considered part of the domain *boundary*. If I really needed the values of the derivative there, I would need to be told something about the boundary conditions for this problem (e.g. the function is constant across the boundary, the derivative is constant across the boundary, etc.).

With those details presented, implementation central differencing scheme is not very complex. Generically something like:

```
stepsize = 0.1
fprime = np.zeros(10)
i = 1
while i < len(fprime)-1:
    fprime[i] = (myfunction[i+1]-myfunction[i-1])/(2*stepsize)
    i += 1
```

would work assuming I've defined the variable "myfunction" to have information about the function that I am trying to differentiate.

2.28 Numerical Integration

In upper level physics, we constantly run in to integral problems that are difficult, if not impossible, to solve analytically. Systems that are more complex than the most basic examples often require the use of approximations that may not be applicable to a particular problem. Luckily, there exist a wide swath of numerical techniques that can be implemented to attack such problems. Here, we will cover a few of them as a study on how such techniques are developed and how to implement them. It should be noted that today's computing languages often come with all, and more, of these techniques implemented in pre-existing software libraries.

Given some function $f(x)$, we would like to find the solution the integral:

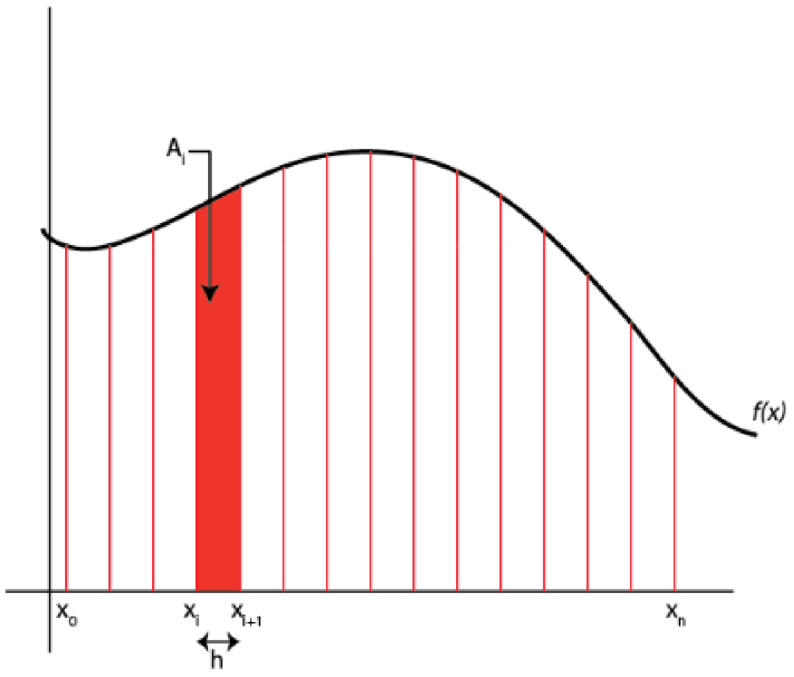
$$I = \int_a^b f(x)dx$$

Techniques for solving this problem all stem from the fundamental definition of the integral. Given some function, $f(x)$, the integral is equal to the area under the curve defined by $f(x)$ between the points a and b. Our job is to use the computer to find a estimate for that area.

Numerically, this means that we have to break up the area into discrete chunks and sum the area of each chunk. The question is: how to break up each chunk?

2.28.1 Rectangular rule

Since finding the area of a rectangle is pretty easy, perhaps the simplest method that we can choose would be to break up the domain that we are interested in into several rectangular chunks. This means that for each rectangle we must



define the width of the rectangle (the difference between two points on our grid) and the height of our rectangle (the function evaluated at some point, $f(x_i)$). There are a few options when it comes to selecting the height.

Let's say that I want to evaluate the area of the i^{th} rectangle. The width of my rectangle is $x_{i+1} - x_i$. The height can either be defined using $f(x_i)$ or $f(x_{i+1})$. As you can see in the figure below, this will result in some error, either an underestimate of the true area or an overestimate depending on the behavior of the function.

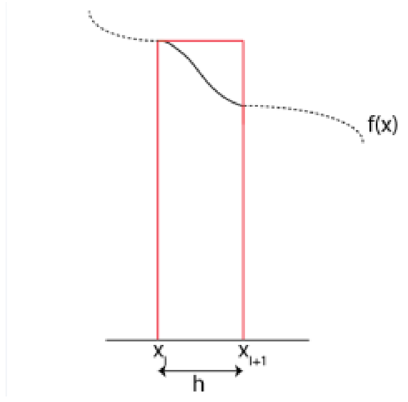


Fig. 7: Choosing the left point results in an overestimate.

This method of approximating the area under the curve is known as the rectangular rule. Once we set up a grid $(x_0, x_1, x_2, \ldots x_n)$ and evaluate our function at each point on the grid, it is easy to calculate the area of each rectangle and sum to get the integral. E.g. we can approximate the integral using the points on the left side of our rectangle using:

$$I = \sum_{i=0}^{n-1} f(x_i)h$$

A common adaptation to the rectangle rule is to use the midpoints of our grid as the point at which to evaluate the "height" of our function. If we don't actually know the value of our function at those points, we can interpolate. This

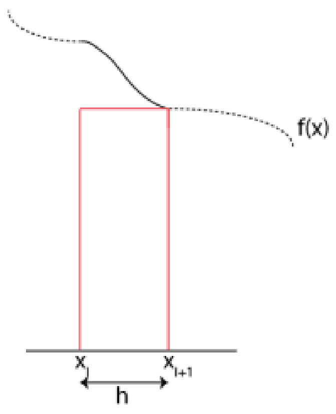


Fig. 8: Choosing the right point results in an underestimate.

gives us the *midpoint* rule:

$$I = \sum_{i=0}^{n-1} f(\frac{x_i + x_{i+1}}{2})h$$

2.28.2 Trapezoid rule

It is clear from the above figures that there can be significant error when implementing the rectangular rule. An alternative then, is instead of breaking the domain up into rectangles, use trapezoids!

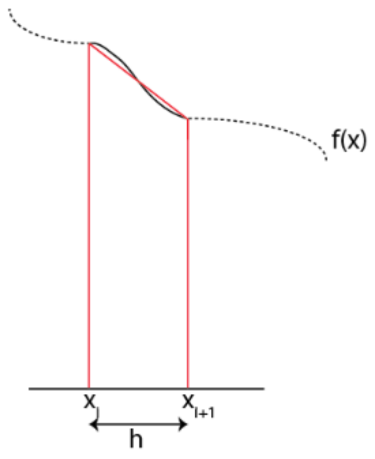


Fig. 9: Trapezoids allow us to use information about the function at both grid points

Calculating the area of a trapezoid is not much more difficult that doing so for a rectangle, and the sum is similar:

$$I = \sum_{i=0}^{n-1} \frac{f(x_i) + f(x_{i+1})}{2} h$$

As you can see from this sum, the function is evaluated twice at each point on our grid, with the exception of the first and last grid points. Written out, the sum looks like:

$$I = \left(\frac{f(x_0}{2} + f(x_1) + f(x_2) + \cdots + \frac{f(x_n)}{2} \right) h$$

In this form, the trapezoid rule looks quite similar to the rectangle rule (and also the midpoint rule). The only difference being that we divide the function evaluated at the first grid point by 2, and add and extra term: $\frac{f(x_n)}{2}$. This should tell you that while we might think the trapezoid rule is much more accurate that the rectangle rule, it actually isn't that much better.

In fact, we can calculate the error of these two methods:

$$error_{midpoint} \approx -\frac{h^3}{24} f''$$
$$error_{trapezoid} \approx \frac{h^3}{12} f''$$

where both error terms depend on the 2nd derivative of the function and the negative sign means that the approximation underestimates the solution when the function is concave up. By taking a weighted average of the two methods, we can effectively cancel out these errors and come up with a new method!

2.28.3 Simpson's Rule

The weighted average looks like:

$$I_s = \frac{2I_m + I_t}{3}$$

where I_m represents the integral calculated with midpoint rule and I_t represents using the trapezoid rule.

Based on that formula, we can combine the summations above to write down the Simpson rule:

$$I = \frac{h}{3} \left(f(x_0) + 2 \sum_{i=1}^{n/2-1} f(x_{2i}) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + f(x_n) \right)$$
$$I = \frac{h}{3} \left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n) \right)$$

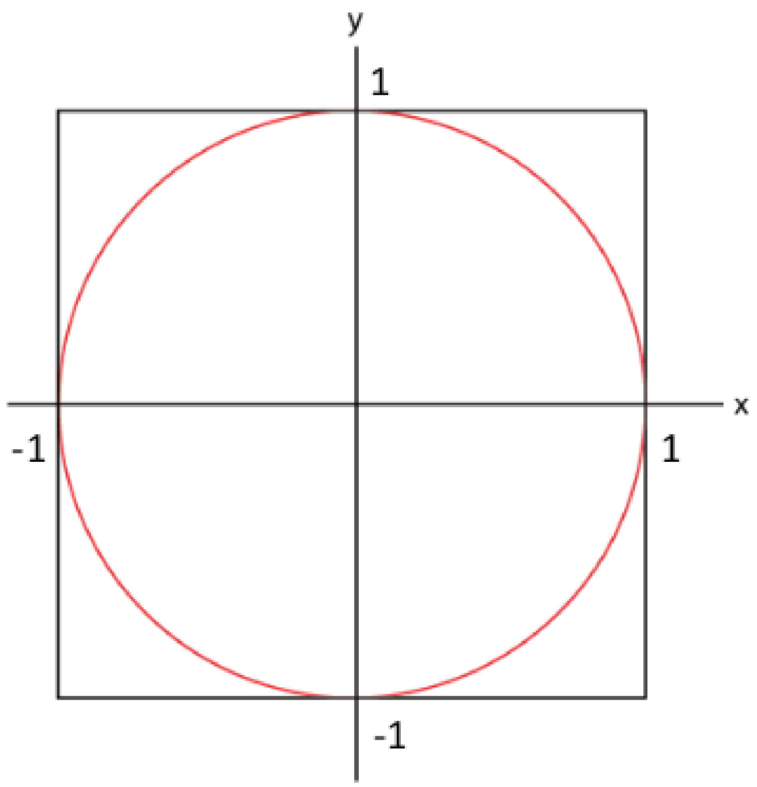
Note that **n must be even**. This technique results in an error proportional to h^5, which is much better than our options above. Additionally, the error depends on the fourth derivative of the function in question, which means that it is exact for any polynomial of 3 degrees or less, a nice bonus.

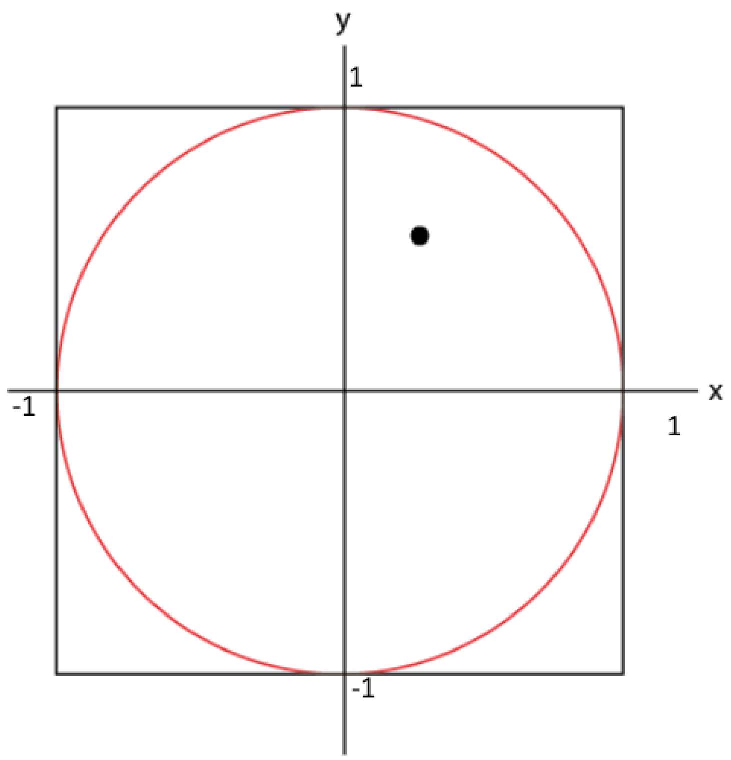
Implementation of Simpson's rule is only slightly more complex than the rules above, only because the coefficient changes depending on if we are dealing with a odd or even grid point. There is a slight computational expense associated with performing the extra floating point operation. However, the improved accuracy over the midpoint method makes Simpson's rule a good choice for typical integration tasks.

2.28.4 Monte Carlo techniques

The techniques discussed above are all based on the concept of interpolating the function that needs to be integrated in some way. However, there are other techniques that we can think of that attack the problem in a different manner. One class of such techniques are Monte Carlo methods, named because they involve some measure of randomness.

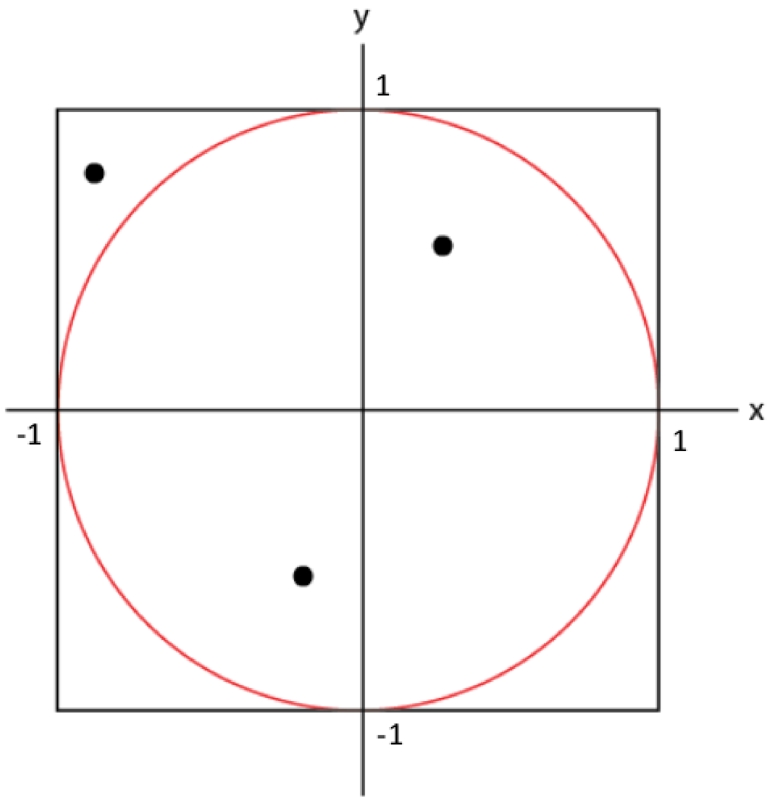
To illustrate the concept, imagine that you wanted to calculate the area of a circle, but you didn't know anything about π or any of that. Instead, you chose to surround the circle by a square, for which you **do** know how to calculate the area. One might draw such a diagram on a piece of paper:


y
1
x
-1
1
-1


y
1
x
-1
1
-1

Next, we throw darts at the paper and we take a tally of total number of darts that were thrown as well as the darts that land inside the circle.

Inside the circle = 1, Total 1



Inside the circle = 2, Total 3

Inside the circle = 4, Total 7

So, 40 out of 50 darts are inside the circle, or 80%. So, assuming the darts were thrown randomly, I could approximate the area of the circle by $a_c \approx 0.8A_s$.

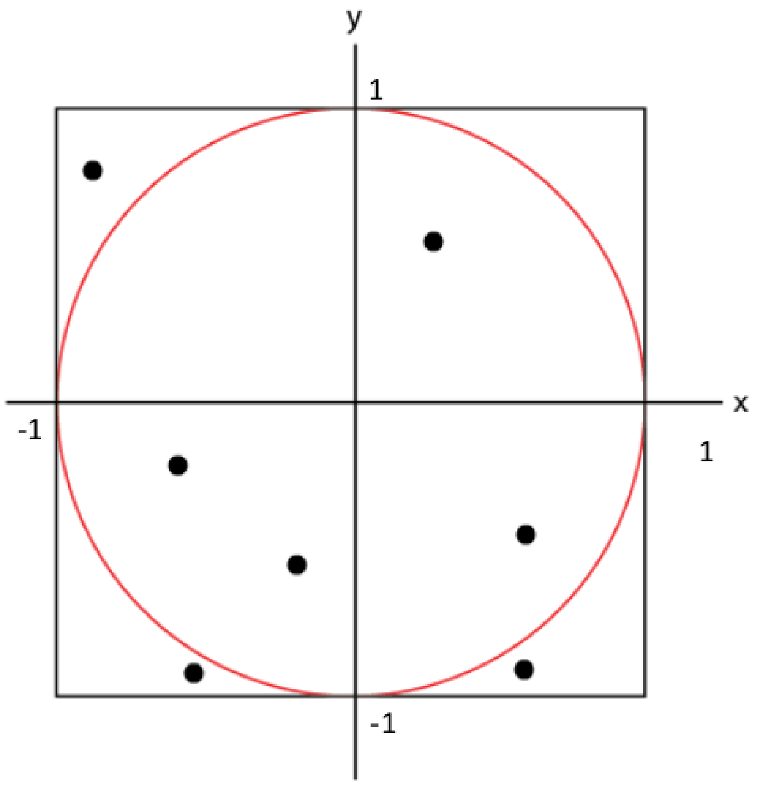
If my square has an area of 4 units, then

$$A_c = 4.0 * 0.8 = 3.2 \approx \pi r^2 = 3.1415$$

Not a bad approximation!

In other words, I can implement this to find the integral of a function but picking a **random** coordinate $(x, y...)$ in the domain. Then, solve the function at that coordinate $f(x)$. Assuming I am taking the integral with respect to y, I could check to see if the random y value that I picked in the first step is less than f(x). If that is the case, I would tally that point as "in". Repeat the procedure with some number, n, of random points. Then, the integral is approximated by the *area (volume, etc) of the domain * in / n*. Of course, the larger n, the better the approximation.

Monte Carlo techniques tend to be slower for low dimensional problems. If you are doing 1D or 2D integration, it is best to stick with Simpson's rule if possible. However, if you have a higher dimensional problem, 3D, 4D, etc. then Monte Carlo methods can be extremely beneficial.



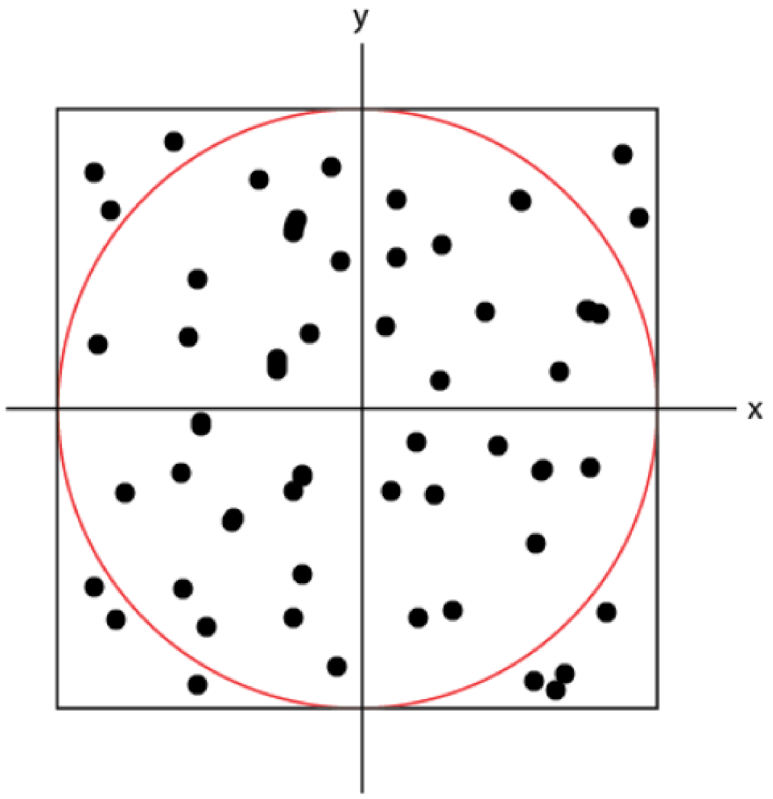


Fig. 10: Inside the circle = 40, Total 50

2.29 Lesson Overview

This week, let's switch gears a little and talk about everyone's favorite topic, typesetting! We will explore using Latex to create beautiful, self-consistent, documents that are publication quality.

Specifically, you will learn about:

1. *The basics of Latex*
2. *Referencing with Bibtex*

2.29.1 Learning Goals

After this week you should be able to:

- Use Latex to create professional documents
- Understand basic Latex syntax
- Include complex elements such as figures, lists and tables in your documents.
- Incorporate simple and complex mathematics in your documents.
- Use self-consistent cross-referencing within Latex to reference different document elements.
- Create a bibtex database and link it to your latex document to enable consistent citations and referencing.

2.30 Latex Basics

Latex is a document preparation system for high-quality typesetting. It is used by a large to create documents by a large part of the scientific community as well as to prepare technical documents in a wide range of fields. I'll start by saying Latex is not a word processor. The whole point of Latex is to allow the author to focus on their writing, and not on the document styling. Latex is great for preparing journal quality articles and has features that can handle sectioning, cross-references, tables, and figures. It makes the typesetting of math formulae straightforward and quick. It automatically generates bibliographies and indices, and seamlessly accommodates the inclusion of graphics. And yes, it also handles formatting, but it separates the formatting process out from the writing process so that, again, the focus can stay on the writing.

Learning how to use Latex can be a frustrating process. But, I absolutely promise that once you get through the learning pains, Latex will save you an huge amount of time when preparing your documents. Additionally, they will look much more professional than if you used other word processors. Ultimately, Latex makes things that are very difficult to do in other software relatively easy. The cost of this is a little more effort to get your documents set up. The figure below summarizes the idea quite well:

2.30.1 Basics

To get started, you will need access to a Latex distribution. There are many. If you don't have any other option, serenity has a distribution installed that you can use via the command line. However, in recent years, web based latex solutions have popped up online and are probably the easiest way to get started as you don't have to download any extra software. Today, Overleaf is one of the better options for web based Latex. You can set up a free account which even includes some limited collaboration options. If you would prefer to download a Latex distribution, MikTex and Texlive are both good options. Before you go any further, set up an account on Overleaf or download and install a latex distribution.

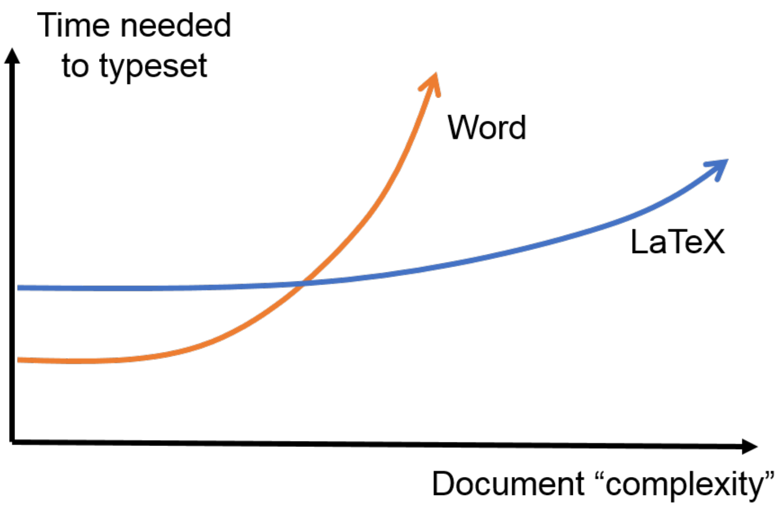


Fig. 11: Word vs. Latex: a summary (this version of the plot by JL Blanco, MappingIgnorance.org)

Many people make the claim that using Latex is like coding your papers. In a sense, it is. The file that contains your writing also has commands that define the document structure and formatting. In addition, in order to produce the final document, you have to compile the latex document. In that sense, using latex isn't as simple as opening up a document and starting to write. We need to specify some basic commands to tell Latex what we are doing.

Three things every Latex document must have

There are three commands every Latex document must have for Latex to compile (compiling takes the latex commands and allows them to operate on the content of the document to produce some sort of output, e.g. a final pdf) successfully. Those commands are: **'documentclass{ }'**, **'begin{document}'** and **'end{document}'**. Before we create our first document, let's talk about these commands in a general sense.

First, note the Latex syntax. All latex commands start with the character '. *The backslash is a special character in Latex, which means you can't just go around using it willy-nilly. If you type that key in a Latex document, you have to use it properly. Next, each of the commands above were followed by a set of curly brackets '{ }*. Anytime you see curly brackets in Latex, those represent a place where you need to include a *required argument*. Alternatively, you may see commands that use square brackets, *[ ]*. Those enclose *optional* arguments.

To the first required command. *documentclass{ }* is required of every Latex document and it requires 1 argument: the document class. There are several to choose from: *proc*, *report*, *book*, *slides*, etc. But, you will always be using the *article* class in this course.

The other two required commands, *begin{document}* and *end{document}* tell latex where the content of the document begins and ends. The *begin* and *end* commands generally define a Latex *environment*. You will use them often, but with different arguments. The basically allow you to add a variety of your elements, including figures, tables and lists, to your document. But the main *environment* is of course the document. Anything that you want to actually show up in your final document must go between the *begin{document}* and *end{document}* commands. With that, let's create a very simple document:

```
\documentclass{article}

% preamble

\begin{document}
```

(continues on next page)

(continued from previous page)

```
Hello World!!

\end{document}
```

Use whatever Latex distribution you are using to create a first document and compile it (typically, if you are using web based software, there is a recompile or compile button that you can click on). If you have just these elements, you should see the text "Hello World!" in the output.

In this simple example, I made use of a Latex comment, which is designated using the % character. Any lines that start with that character are ignored by the Latex compiler. In this example, I wrote the word "preamble". Any content in our document that comes after the *documentclass* but before *begin{document}* is referred to as the preamble. This space is reserved for configuration options for our document, defining and redefining commands, and generally setting up preferences. We will talk more about some of this stuff later, but **generally** the preamble contains information about document-wide formatting, as well as information about Latex packages that you may use in the body. The body contains the content of the document, as well as small scale text formatting commands.

2.30.2 The Preamble

Before we get into the syntax, let's briefly discuss the preamble. Remember that I said the goal of a good typesetting program is to separate the formatting from the actual writing. In some sense, this is the goal of the preamble. It is here that we specify the document wide formatting options, including the fonts, margins, spacing, list formatting, section formatting, and more. But, there are many other things that we can do in the preamble as well. The preample is the place that we will "import" additional functionality into our documents by adding Latex packages. This is done via the \usepackage{} command. This command takes at least one required argument, the name of the package that you want to import. Standard latex distributions come with many, many packages that you may find useful. Among those, you will probably use the *amsmath*, *graphicx* and *natbib* packages regularly. In order to use them, you must include them with the \usepackage command in the preamble.

2.30.3 General syntax

There are a few Latex syntax guidelines that you should remember:

1. Spaces and line breaks aren't important with one exception: one or more blank lines starts a new paragraph.
2. Commands always start with a backslash, .
3. Curly brackets are used for required arguments for example: *documentclass[11pt]{article}*.
4. Square brackets are used for optional arguments.
5. Commands are case sensitive.

The most common optional arguments to *documentclass* are:

- 11pt- uses 11-point font instead of default size
- 12pt- uses 12-point font instead of default size
- twocolumn- produces two column output

2.30.4 Sectioning

Latex has several levels of sectioning that make it easy to structure your document:

```
\section{section name}
  \subsection{subsection name}
    \subsubsection{subsubsection name}
```

The title of each section goes in the braces. Latex will automatically number your sections, and there are options for different number schemes.

2.30.5 Font Styles

Latex will automatically set the font for you, but you can specify other styles on the fly:

- *{\em text}*- italics
- *{\tt text}*- fixed-width typewriter-like font
- *{\bf text}*- bold font

The use of the curly brackets allows the inclusion of multiple words. If you only wanted to boldface a single word, you don't need the braces, e.g.: `\\bf Hello world` will result in **Hello** world.

2.30.6 Lists

To create any time of list, you need to enter a list environment. Environments are common ways in Latex to perform formatting on a block of text. This is opposed to inline formatting, where the formatting is applied to a text element. To enter an environment, you enclose the text with in a `\begin{}..\end{}` block, just like you enclose the text of your latex file using the `\begin{document}` and `\end{document}` commands. There are 4 types of lists in Latex:

- Bulleted- to create a bulleted list, you use enclose your list with the commands `\begin{itemize}` and `\end{itemize}`. Each item in your list is prefaced with the `\item` command (no braces).
- Enumerated- to create a numbered list use `\begin{enumerate}` and `\end{enumerate}`, and again, use `\item`.
- Descriptive- composed of subheadings followed by one or more indented paragraphs. To create a descriptive list, use `\begin{description}` and `\end{description}` and use `\item`.

You can also make nested lists by defining another list environment within a list environment. Latex will handle the nesting and make an alternative bullet or numbering scheme.

2.30.7 Special Characters

Since certain characters are used in Latex commands (e.g., the backslash and curly braces), if you want to actually print these characters in your document, you have to **escape** them (not to treat them as part of a command). Generally, this is done with a leading backslash. However, there are some exceptions:

```
Character   Command
\           $\backslash$
$           \$
%           \%
^           \^
&           \&
_           \_
~           \~
#           \#
```

(continues on next page)

(continued from previous page)

```
{           $\{$
}           $\}$
```

2.30.8 Math

One of the reasons that a lot of people transition to Latex is due to the ease of rendering mathematical expressions. Today, you can find Latex math syntax incorporated in many services, including add-ons for gmail as well as in Canvas.

There are two ways to use math mode: inline and display. In inline, math is rendered inline with the text: e.g. $y = \cos^2(\theta)$. In display mode, the math content is separated from the text:

$$y = \cos^2(\theta)$$

Inline

Entering inline math mode is done using the special symbol: $. The dollar sign lets latex know to treat certain symbols in a special way. For example, entering *$a^2+b^2=c^2$* results in $a^2 + b^2 = c^2$.

Display

For display mode, there are several options. The simplest option is to use two dollar signs instead of one: *$$a^2+b^2=c^2$$* will produce:

$$a^2 + b^2 = c^2$$

Again, Latex doesn't care about whitespace, so I can type the above expression inline with the text in my Latex document, as I've done here, but it will be rendered separate from the text. The use of $$ is the quickest way to enter display math mode, but by default, it does not result in numbered equations nor will it align them if you are trying to display an equation set. For this reason, many people prefer to use the *amsmath* when writing extensive mathematics in their documents. Again, to do this include the `\usepackage{amsmath}` command in your document's preamble. Amsmath gives us access to a modified *align* environment which allows us to align math expressions and number them at the same time. For example:

```
\begin{align}
  \sum F&=ma \\
  &=F_N_x - T_x\\
  &=mg\sin(\theta)-T\cos(\phi)
\end{align}
```

results in:

$$\begin{aligned} \sum F &= ma && (1) \\ &= F_{Nx} - T_x && (2) \\ &= mg\sin(\theta) - T\cos(\phi) && (3) \end{aligned}$$

Fig. 12: Using the amsmath package gives us an align environment that produces numbered and aligned equations.

Note the use of the & character. This is Latex's *alignment* character. It is not printed in the rendered document. Instead it results in alignment of the equations where that character is located.

Basic Math

Once we enter math mode, the syntax and key combinations to do various things is meant to be somewhat intuitive. I'll provide an overview of the basics here but I'll refer you to the summary here: https://en.wikibooks.org/wiki/LaTeX/Mathematics.

Arithmetic Operations: The plus (+), minus (-), division (/) symbols have the usual meaning. To denote multiplication explicitly (this is rarely necessary), use `\cdot` (pro- ducing a centered dot) or `\times` (producing an "×"). The equal, less than, and greater than symbols on the keyboard work as expected; to get less than or equal, use `\le`; similarly, `\ge` gives greater than or equal. Square roots are generated with the command `\sqrt{...}`: `$z=\sqrt{x^2+y^2}` gives $z = \sqrt{x^2 + y^2}$.

Subscripts and superscripts: These are indicated by carets ^ and underscores _, as in `$n^2$` or `$a_1$` which produce n^2 and a_1 respectively. If the sub/superscript contains more than one character, it must be enclosed in curly braces, as in `$2^{x+y}$`.

Fractions: Fractions are typeset with `$\frac{x}{y}$`, where x stands for the numerator and y for the denominator. An example: `$\frac{f'(x)(x-a)}{n!}$` produces $\frac{f'(x)(x-a)}{n!}$.

Sums and Integrals: The symbols for sums and integrals are `\sum` and `\int`, respectively. These are examples of "large" operators, and their sizes are adjusted by TeX automatically, depending on the context (e.g., inline vs. display math). Note that the symbol generated by `\sum` is very different from the capital sigma Greek symbol, `\Sigma`; the latter should never be used to denote sums. TeX uses a simple, but effective scheme to typeset summation and integration limits: Namely, lower and upper limits are specified as sub- and superscripts to *\sum* and *\int*. For example, `$\sum_{k=1}^n k = \frac{n(n+1)}{2}$` produces $\sum_{k=1}^n k = \frac{n(n+1)}{2}$. (Note that the "lower limit" k=1 here must be enclosed in braces, because it is more than 1 character long). Typically "large" operators are used in display mode as opposed to inline mode:

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

Further, if one uses parenthesis or brackets in an expression that uses large operators or fractions, the `\left` and `\right` commands are often used to assist in the correct sizing of those symbols. Compare `$$[\sum_{i=1}^n\frac{f^{i}(x)}{i!}(x-a)^i]$$`:

$$[\sum_{i=1}^n\frac{f^{i}(x)}{i!}(x-a)^i]$$

with `$$\left[\sum_{i=1}^n\frac{f^{i}(x)}{i!}(x-a)^i\right]$$`:

$$\left[\sum_{i=1}^n\frac{f^{i}(x)}{i!}(x-a)^i\right]$$

Greek Letters: The commands for Greek letters are easy and intuitive: Just type `$\epsilon$`, `$\delta$`, `$\nu$`, `$\phi$`, etc. To get upper case versions of these letters, capitalize the appropriate command; e.g., `$\Delta$` gives a capital Delta.

2.30.9 Tables

Tables are produced in Latex using the tabular environment, as in begin{tabular} and end{tabular}:

```
\begin{tabular}{|c|l|}
  \hline
  n & n! \\
  \hline
```

(continues on next page)

(continued from previous page)

```
  1 & 1\\
  2 & 2\\
  3 & 6\\
  4 & 24\\
  5 & 120\\
  6 & 720\\
  7 & 5040\\
  8 & 40320\\
  9 & 362880\\
  10 & 3628800\\
  \hline
\end{tabular}
```

When beginning the tabular environment, there is a required format specification, the stuff you see in the second set of curly braces. That tells latex how many columns to use, how to justify the text in the columns, and where to put vertical bars (using the | key). In this example, we have two columns. The first is center-justified and the second is left justified. In addition, there is a vertical bar on both sides of the table, as well as one separating the two columns. Other options here are:

```
l         specifies a column of left-justified text
c         specifies a column of centered text
r         specifies a column of right-justified text
p{width}  specifies a left-justified column of the given width
|         inserts a vertical line between the columns
@{text}    inserts the given text between the columns
```

Once you've set the table up, its time to add the content. Horizontal bars are added using the `\hline` command. Then, each row in the table is written. Columns are separated by the ampersand, &. Since we only have two columns, only one ampersand is used in each row. Since whitespace doesn't matter in latex, you let Latex know to start a new row using the new line command, two backslashes, `\\`. Notice that there is no need to give any information on the dimensions of the table. Latex does all that for you.

You can have text that spans multiple columns very easily. Also, you can include normal Latex typesetting commands:

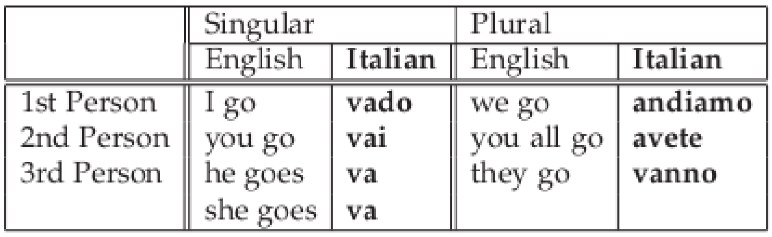
```
\begin{tabular}{|l||l|l||l|l|}
  \hline
  &\multicolumn{2}{l|}{Singular}&\multicolumn{2}{l|}{Plural}\\
  \cline{2-5}
  &English&{\bf Italian}&English&{\bf Italian} \\
  \hline\hline
  1st Person&I go&\textbf{vado}&we go&\textbf{andiamo}\\
  2nd Person&you go&\textbf{vai}&you all go&\textbf{avete}\\
  3rd Person&he goes&\textbf{va}&they go&\textbf{vanno}\\
  &she goes&\textbf{va}& & \\
  \hline
\end{tabular}
```

Results in:

Notice how the last line has two blank cells. Also, the `\cline` command has been used, which "clears the line".

Table as a float

A table created as described above will place the table "in line" with the text. Typically, we want our tables to "float", or adjust position in the text so that the table is at the top or bottom of the page. Floats exist to deal with the problem of an object that won't fit on the present page. The are not part of the normal stream of text, but separate entities, and



	Singular		Plural	
	English	**Italian**	English	**Italian**
1st Person	I go	**vado**	we go	**andiamo**
2nd Person	you go	**vai**	you all go	**avete**
3rd Person	he goes	**va**	they go	**vanno**
	she goes	**va**		

Fig. 13: An advanced table with text that spans multiple columns.

are position in a part of the page to themselves (top, middle, bottom, left, right, etc. As such, we need to wrap our tabular environment in another environment:

```
\begin{table}
  \centering
  \begin{tabular}{|c|l|}
  \hline
  n & n! \\
  \hline
  1 & 1\\
  2 & 2\\
  3 & 6\\
  4 & 24\\
  5 & 120\\
  6 & 720\\
  7 & 5040\\
  8 & 40320\\
  9 & 362880\\
  10 & 3628800\\
  \hline
  \end{tabular}
  \caption{\small A table showing the result of taking the factorial of the numbers 1␣
↪-- 10
\end{table}
```

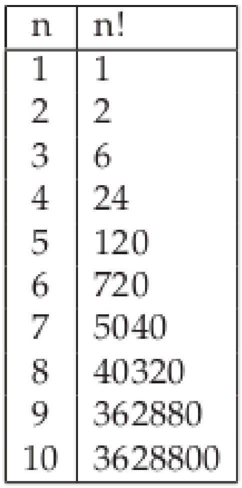
This produces the following table:

Note that we nested the tabular environment within the table environment. One of the advantages of this is we are able to include a caption (I like to make my captions have smaller text than the rest of the document, hence the `\small` command.). Tables should always be created this way, e.g. as floats and not inline.

All floating environments (tables, figures, etc) take optional positioning arguments when defining the float: `\begin{table}[placement specifier]`. The placement specifier can be one or more of the following:

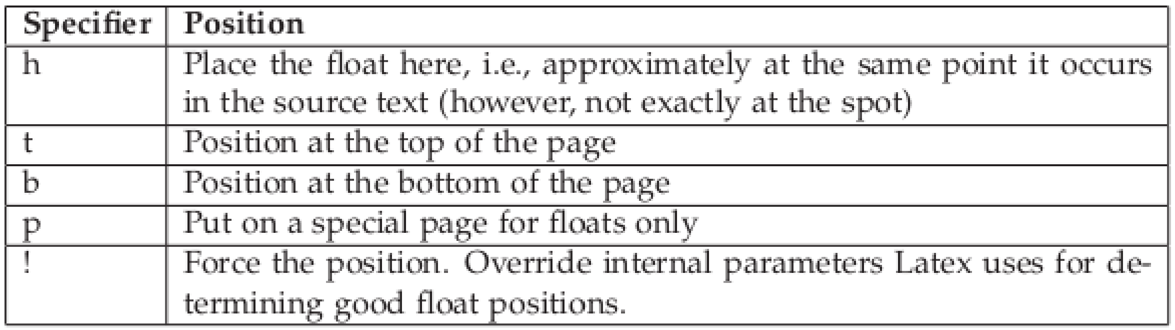
Latex tries to put the table (or figure) where you want it, but it does have some guidelines that it also tries to obey. For one, Latex really wants the floats to be either at the top or bottom of the page. This means that the float is never sandwiched by the text. Latex will really struggle with positioning if you have several floats very close together in the text. In some cases, they may overlap or run into one another, so it becomes sort an art to space things properly.

Note: Don't worry to much about the position of a float! Let Latex do what it wants even if the float comes on a different page than the text that is referencing it!



n	n!
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Table 1: A table showing the result of taking the factorial of the numbers 1 – 10.



Specifier	Position
h	Place the float here, i.e., approximately at the same point it occurs in the source text (however, not exactly at the spot)
t	Position at the top of the page
b	Position at the bottom of the page
p	Put on a special page for floats only
!	Force the position. Override internal parameters Latex uses for determining good float positions.

Fig. 14: The same positioning arguments are used for figures as well, as you will see later.

2.30.10 Labels and Cross-referencing

One of the advantages to using Latex is that it handles labeling, numbering, and cross-referencing for you. The way that this works is you attach a label to some part of your document, then you reference the labeled object in the text. For example, I can add a label to our factorial table:

```
\begin{table}
  \centering
  \begin{tabular}{|c|l|}
  \hline
  11n & n! \\
  \hline
  1 & 1\\
  2 & 2\\
  3 & 6\\
  4 & 24\\
  5 & 120\\
  6 & 720\\
  7 & 5040\\
  8 & 40320\\
  9 & 362880\\
  10 & 3628800\\
  \hline
  \end{tabular}
  \caption{\small A table showing the result of taking the factorial of the numbers 1␣
↪-- 1
  \label{factorial}
\end{table}
```

Once I have a label, I can reference the table using the `\ref{factorial}` command. The argument of the label just has to be the same as the argument for the \ref. Note that the `\ref{factorial}` command only puts the number in the text, not the word "Table". I have to do that, e.g. "See Table \ref{factorial}" Note that Latex only counts floats, not the tables that were defined without the table environment, so if you have use an inline table, it will not be counted (again, you should always use floating tables).

Warning: When referencing floats, the label should always come after the caption, if there is one. Otherwise the numbering will be wrong if you reference the object!!!!

Cross-referencing in this way (defining a label and using it with \ref{}) can be used on tables, figures, equations, and sections. For example:

```
\section{Introduction}
\label{intro}
```

By assigning a label to the Introduction section, I can then reference the section using the `\ref{}` command: "For background information see Section \ref{intro}". Not that the label that I used (intro) is totally up to me and can be anything I want. Then, I use that text string as the argument to the \ref{} command.

2.30.11 Figures

The use of figures is much the same as tables from Latex's point of view. The only difference is that Latex alone isn't capable of handling and interpreting graphics files without the use of additional packages. I've mentioned one of the most widely used packages already: `graphicx`. You tell Latex that you want to use this pack- ages by including it in the Latex document preamble like so: `\usepackage{graphicx}`. The graphicx packages can handle almost any type of image, i.e., pdf, jpg, bmp, png, etc. One notable exception is ps, or postscript files. These are commonly produced on unix-like systems. There are a variety of unix tools available that will convert ps to other standard file types such as imagemagick (a common unix utility).

Again, when you use a figure, Latex wants to create a float object. This is accomplished using the figure environment:

```
\begin{figure}[htp]
  \centering
  \includegraphics[width=8cm]{flare.eps}
  \caption{\small A solar flare!}
  \label{flare}
\end{figure}
```

The `\includegraphics` command has several optional arguments that are useful, particularly those dealing with size like width and height. As with tables and sections, we can reference the figure using the `\ref{}` command: e.g. "See Figure \ref{flare}."

2.31 Latex + Bibtex: Bibliographies

Latex offers many capabilities that are difficult or impossible to do using other software. One such is Latex's ability to work with Bibtex, reference management software, to create a self-consistent bibliography. The seemless integration of Bibtex and Latex make the referencing and citation process extremely easy.

2.31.1 Bibtex database

In order to do referencing seamlessly in Latex, it is necessary to create a database containing the works that you wish to cite in your documentation. The beauty of separating the bibliography out into a separate file as opposed to including references within the document itself is that you can maintain a single database for all of your documentation.

Each entry in your bibtex database is defined using keywords and linked to a bibliography label. For example, consider the following entry:

```
@ARTICLE{pawlowski_params,
   author = {{Pawlowski}, D.~J. and {Ridley}, A.~J.},
    title = "{Quantifying the effect of thermospheric parameterization in a global␣
↪model}",
  journal = {Journal of Atmospheric and Solar-Terrestrial Physics},
 keywords = {Thermosphere, Global climate model, Parameterization},
     year = 2009,
    month = dec,
   volume = 71,
    pages = {2017-2026},
      doi = {10.1016/j.jastp.2009.09.007},
   adsurl = {http://adsabs.harvard.edu/abs/2009JASTP..71.2017P},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

This entry is labeled as "pawlowski_params" and all of the relevant information about the article is specified. Keywords such as "author", "journal", and "year", etc. contain the necessary information to properly populate the reference. Simply put, including a reference in a document is as simple as adding all of your references to your database using the above format, saving that database (latex requires the .bib extension on your bibliography database, so you might call your file "mybib.bib" or something), and making sure that your project has access to this file. For those that use overleaf.com, this means uploading that database to your project on overleaf.

Since bibtex is such a widely used piece of software, most reference management programs have the ability to export citations in this format. This means that if you have a reference that you want to cite, you can almost always export the bibtex entry or otherwise copy it directly from whatever software you use to find and read documents (e.g. Mendeley Desktop, EndNote, SciRef, Zotero to name a few) and paste it into your bibtex database. Further, common literature databases, including Google Scholar, also have the ability to export the bibtex entry for documents that you find within. All of this means that adding entries to your database can be handled by software so you don't have to manually type in the relevant information.

2.31.2 Connecting Bibtex to Latex

Armed with a populated bibtex database, the only thing to do is to tell Latex that you want to include a bibliography, specify a bibliography style, and cite some documents. This is all accomplished with a few commands added to your Latex document.

Natbib

There are many bibliography styling options available and if you are submitting an article to a journal or other publication there is probably a style that the publisher requires. I require my students to use the natbib package by including `\usepackage{natbib}` in the preamble of the latex document. This provides two easy to use commands for adding citations and formats the citations and references in the manner that I prefer.

The bibliography

To include a bibliography in your latex document, use two commands at the end of your document (but before the \end{document}):

```
\bibliography{databasename}
\bibliographystyle{stylename}
```

where "databasename" is the name of your bibtex database (e.g. mybib.bib) and "stylename" is the appropriate style. I suggest `plainnat` which comes with the natbib package.

Citing a reference

Now that latex knows about your database, you simply need to add citations to your latex document. Natbib provides two citation commands: `\citep{}` for parenthetical citations and `citet{}` for textual citations.

Parenthetical citations are those that are not meant to be read as part of the text. For example, the following text:

```
In a previous study, it was demonstrated that uncertainty in specification
of the thermal conductivity can result in extremely different circulation
patterns in the atmosphere \citep{pawlowski_params}.
```

will result in the following citation:

```
... circulation patterns in the atmosphere [Pawlowski and Ridley, 2009].
```

The citation is fully enclosed in brackets. Alternatively, I could structure the text slightly differently to use a textual citation:

```
\citet{pawlowski_params} show that uncertainty in specification
of the thermal conductivity can result in extremely different circulation
patterns in the atmosphere.
```

which will result in the following citation:

```
Pawlowski and Ridley [2009] show that uncertainty...
```

Note that in both citation styles, the appropriate bibtex entry label was used as the argument to the citation command.

When either citation command is called, Latex notes the entry that was used, creates the citation inline with the text as appropriate and also adds an entry to the References section. No other work is required to ensure that there is a connection between the citation and the reference and at no point to we have to manually populate the references section itself. Latex handles all of that automatically.

2.31.3 Summary

To summarize, the steps to add the ability to cite and reference other works in your document are:

1. Create a bibtex database and populate it with the entries that you want to cite.
2. Add that database to your project (e.g. the folder that contains your latex document).
3. In your Latex document, add the bibliography and styling using the `\bibliography` and `\bibliographystyle` commands. Be sure to add Latex packages as necessary (.e.g `\usepackage{natbib}`).
4. Cite the referenced works using the relevant `\cite` command (.e.g `\citet` or `\citep`).

2.32 Lesson Overview

Back to numerics! Our goal this week is to solve ordinary differential equations using a computer! So, we will look at a few techniques for doing just that.

1. *Solving ODEs*

2.32.1 Learning Goals

After this week you should be able to:

- understand the development of algorithms behind several different ODE solvers.
- tell the difference between the Euler method and Runge-Kutta schemes.
- implement a variety of ODE solvers.
- describe why one type of solver might be unstable.

2.33 Ordinary Differential Equations

An ordinary differential equation is a differential equation in which all dependent variables are functions of a single independent variable. ODEs show up all over the place in physics. From analyzing Newton's laws, to harmonic motion, to circuit analysis, these types of problems are very common and are typically encountered with when working on realistic physical systems.

For a given ODE, if it is reduced to its simplest form, the highest derivative (first, second, etc.) tells us the "order" of the ODE. In general, we would like to be able to describe an algorithm that enables us to solve an n^{th} order ODE. As an example, consider

$$\frac{d^2x_j}{dt^2} = \frac{F_j(x_1, ..., x_n, t)}{m_j}$$

where x_j is the position of the j^{th} object, m_j its mass, and F_j is the net force acting on that object, which may depend on any number of other variables (x_i, t, order ODE as a series of n 1:math:^*{st}* order ODEs and apply techniques that can solve just 1:math:^*{st}* order equations. For example, we can rewrite our equation above as

$$\frac{dx_j}{dt} = v_j,$$
$$\frac{dv_j}{dt} = \frac{F_j(x_1, ..., x_n, t)}{m_j}$$

As a shorthand, let's denote our first derivative using prime notation: y'. We are given the ODE as a function of some number of independent variables:

$$y' = \frac{dy}{dx} = f(x, y, ...)$$

where in this case, we let the prime denote that we are taking the derivative with respect to x. Again, we know $f(x, y, ...)$ and we seek the solution to the ODE: $y(x, ...)$. In other words, we know the derivative of y, which is a function of y and other variables, we want to find y as a function of those other variables.

Note: For all of the methods discussed here, I've tried to assume that our ODEs are a function of a general number of variables. Of course, it is possible that any given ODE only be a function 1 (or even no) variables.

For any technique, in order to solve this equation, we need some other information about our function $y(x, ...)$. Typically, we are given some initial value boundary condition. For example, let's assume that the function that we are after is only a function of a single variable, such that $y = y(x)$. Then, we may be given

$$y(x_0) = y_0 \tag{2.6}$$

the value of the function y at some location. In order to find the value of y at all other locations, we must use this value. As is the case for many of the problems that we deal with in this course, before we can proceed, we need to discritize the domain that we are interested in. In other words, we will find y at all points x_i where

$$x_i = x_0 + ih$$

and h, as always, is the grid spacing. Given the ODE, an initial value and a grid, we can then proceed to implement a few different techniques to solve our ODE.

2.33.1 Euler's method

The most basic technique that we can employ to solve our 1_{st} order ODE is Euler's method. This technique exploits the fact that we know the 1st derivative of a function, $y'(x)$ and an initial value of the function itself, $y(x_0)$. We then find the value of the function at a neighboring point, $y(x_1)$ by assuming that the derivative is constant between the two grid points x_0 and x_1. We can write the technique as:

$$y_{n+1} = y_n + y'_n h \tag{2.7}$$

Since we are given information about y' everywhere, we can determine y_{n+1} if we know y_n. Thus we require the boundary condition in Eq. (2.6). This process is depicted graphically below.

In Eq (2.7), it may seem as though we don't use information about the grid. However, remember 1) that we must know y_0 at some initial value for x at x_0 and 2) our ODE, $y'(x, ...)$ may also be a function of x. Generally, Euler's method is relatively straightforward to implement.

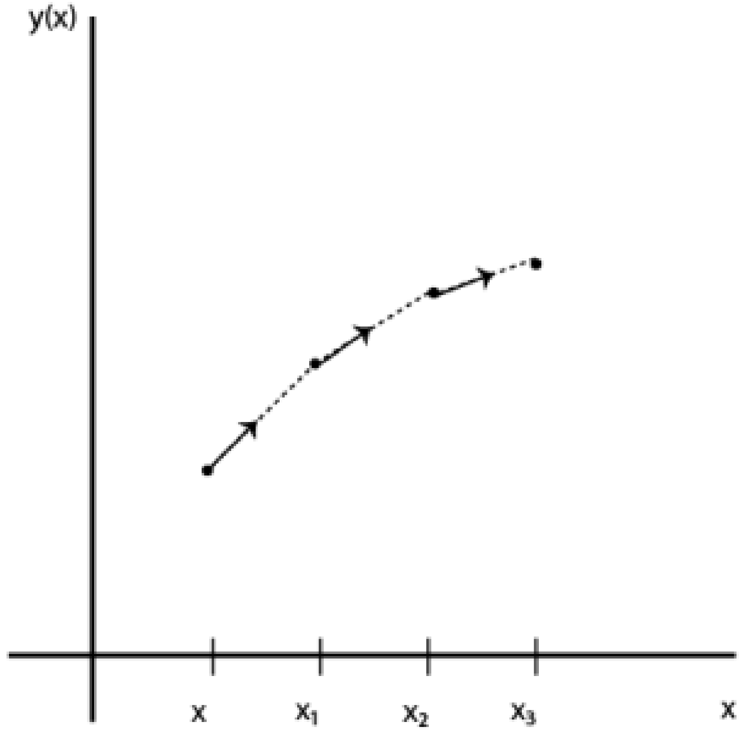


Fig. 15: We are given $y(x_0)$ and from there we can use information about the derivative of y to find subsequent values of y.

1. Establish a grid that spans the domain of interest.
2. Initialize variables, including specifying the initial value condition according to Eq (2.6).
3. Create a loop that covers each grid point.
4. Evaluate the value of the ODE, y'_i at that grid point, x_i given y_i .
5. Apply Eq (2.7) to find the value of the function, y_{i+1} at the next point.
6. Repeat until you've reached the end of the grid.

Although the algorithm is relatively simple, there are 2 issues to consider: 1) error and 2) stability.

2.33.2 Runge-Kutta Techniques

The methods that are typically used by scientists to solve ODEs are called Runge-Kutta schemes (after a couple German mathematicians). The idea is this: since Euler's method is asymmetric, it only depends on derivatives taken at the beginning of a particular interval of interest. This makes the errors relatively high. Runge-Kutta methods attempt to perform a more symmetric step. First, take an Euler step to the midpoint of the interval:

$$y_{n+\frac{1}{2}} = y_n + y'\frac{h}{2}$$

Then, use the values at that point to calculate the solution at the real interval. This results in the 2^{nd} order Runge-Kutta method. Given some ODE that is a function of some number of variables, $y'(x, y, ...)$:

$$\begin{aligned} k_1 &= hy'(x_n, y_n, ...) \\ k_2 &= hy'(x_n + h/2, y_n + k_1/2, ...) \\ y_{n+1} &= y_n + k_2 + \mathcal{O}(h^3) \end{aligned}$$

The result of this is the first order errors are canceled out and we get a method that is 2^{nd} order accurate. We gain accuracy by using the derivative evaluated at an extra point, and by being smart about it.

Of course, we don't need to stop at taking the derivative at just one extra point. We could keep going! In fact, the 4_{th} order Runge-Kutta method is one of the most popular methods of integrating ODEs currently in use:

$$\begin{aligned} k_1 &= hy'(x_n, y_n, ...) \\ k_2 &= hy'(x_n + h/2, y_n + k_1/2, ...) \\ k_3 &= hy'(x_n + h/2, y_n + k_2/2, ...) \\ k_4 &= hy'(x_n + h, y_n + k_3, ...) \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + \mathcal{O}(h^5) \end{aligned} \tag{2.8}$$

The 4^{th} order RK method isn't necessarily limited by round off error as you increase n, but rather, the computational effort. For every step, the ODE must be evaluated 4 times (or in general, n times for an n^{th} order RK method). As n gets large, this means many calculations are performed for each step. For this reason, higher order methods are not frequently used. They don't give enough benefit in error reduction to make the decrease in performance worthwhile.

Implementation of Runge-Kutta techniques are very similar to that of Euler's method. The difference being that we must evaluate the k values before we can find y_{n+1}:

1. Establish a grid that spans the domain of interest.
2. Initialize variables, including specifying the initial value condition according to Eq (2.6).
3. Create a loop that covers each grid point.
4. Find k_1 by evaluating the value of the ODE, y'_i at that grid point, x_i given y_i.

5. Using the previous step, find k_{n+1} by evaluating the ODE using updated values. Repeat as necessary for remaining k s.
6. Apply final equation in Eq (2.8) to find the value of the function, y_{i+1} at the next point.
7. Repeat until you've reached the end of the grid.

2.34 Laplace's Equation

Unlike ordinary differential equations, which can be solved using general techniques such as Euler's method or a Runge-Kutta schemes, general numerical solutions to partial differential equations (PDEs) do not exist. Instead, there are a wide range of numerical techniques that can be applied to different classes of PDEs. The goal here is to numerically solve one such class referred to as elliptic PDEs, one example of such an equation being Laplace's equation, which describes the electric potential in space in absence of any electric charges:

$$\nabla^2 V = 0$$

or

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = 0. \tag{2.9}$$

To solve this equation, we will focus on the use of numerical schemes called relaxation methods.

To start, it is necessary to discretize the independent variables, x, y, and z. In other words, we create a grid for which we will determine the potential $V(i,j,k)$, where i,j,k refer to the indices of a particular grid point. Then, we need to rewrite the derivatives in Eq. (2.9) in terms of a finite difference. We begin with

$$\frac{\partial V}{\partial x} \approx \frac{V(i+1,j,k) - V(i,j,k)}{\Delta x} \tag{2.10}$$

where we have used forward differencing. We could have just have easily used the backward or central difference. Note that Eq. (2.10) is effectively centered at the point $i + \frac{1}{2}$. We can then approximate the second derivative of the function as

$$\frac{\partial^2 V}{\partial x^2} \approx \frac{1}{\Delta x}\left[\frac{\partial V}{\partial x}(i+\frac{1}{2}) - \frac{\partial V}{\partial x}(i-\frac{1}{2})\right], \tag{2.11}$$

where it is explicit that the 1st derivatives are really evaluated at $i \pm \frac{1}{2}$. Putting it all together, we have

$$\begin{aligned}\frac{\partial^2 V}{\partial x^2} &\approx \left[\frac{V(i+1,j,k) - V(i,j,k)}{\Delta x} - \frac{V(i,j,k) - V(i-1,j,k)}{\Delta x}\right] \\ &\approx \frac{V(i+1,j,k) + V(i-1,j,k) - 2V(i,j,k)}{(\Delta x)^2}.\end{aligned} \tag{2.12}$$

This equation says that in order o determine the function at the grid point that we are interested in, $V(i,j,k)$, we must know the function at the surrounding grid points $i+1, i-1$, etc. In other words, the value of V at a particular grid point is the average of V at all the neighboring points. The solution, $V(x,y,z)$, then is a function that satisfies that condition at all points simultaneously.

Generalizing this problem to 3-dimensions follows the same steps as above and results in

$$V(i,j,k) = \frac{1}{6}[V(i+1,j,k) + V(i-1,j,k) + V(i,j+1,k) + V(i,j-1,k) + V(i,j,k+1) + V(i,j,k-1)]], \tag{2.13}$$

where we have rearranged and solved for the point of interest. Note that the factor of $\frac{1}{6}$ comes from the 3-D geometry. If this were a 2-D problem, the factor would be $\frac{1}{4}$. There is an issue here though. In order to solve this problem, we need to have prior information about the function at all grid points, not just the boundary. The solution to this is to start with a guess of the function. In many cases, it is sufficient to guess $V = 0$ everywhere (except on the boundary). To improve our guess, we evaluate Eq (2.13) at each point on our grid using our initial guess for the function in all terms on the RHS of the equation. Once we have solved for (i, j, k) at all points on our grid, we can repeat the procedure using our updated guess to fill in the RHS. In that way, we *relax* our initial guess into the solution (to some confidence level) by iterating over the grid, updating the RHS of the equation with our latest and greatest guess at the function. This processes is referred to as the **Jacobi method** or **Jacobi Relaxation**.

This method works due to the uniqueness theorem of Laplace's equation, which basically states that if you find a solution that satisfies the equation, then it is the one and only solution. That means that given information about the boundary conditions, we are free to simply guess the solution to Laplace's equation within our grid. If we guess a solution that works, then we are done. Jacobi relaxation simply amounts to 'strategically guessing'.

Implementation of Jacobi Relaxation is relatively straight forward. Setup a grid and start with an initial guess for the value of $V(i, j, k)$ on the grid while also including the boundary conditions. Update $V(i, j, k)$ repeatedly for each grid point using Eq. (2.13) while simultaneously checking for convergence. Convergence is achieved when the change to $V(i, j, k)$ is sufficiently small (based on the problem).

2.34.1 A note about relaxation techniques

The use of relaxation in this way is in many ways similar to taking a time-independent problem and turning it to a time dependent one. Imagine a toy problem where we know the potential of some 2D region of space at the boundary: $V(0, y) = 1$ and $V(1, y) = -1$. We wish to find the potential everywhere inside the domain $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$. Jacobi relaxation essentially solves the time dependent problem where at $t < 0$, $V = 0$ everywhere, then at $t = 0$ the boundary conditions are suddenly applied (i.e. two conducting walls are held at +/- 1V.). At exactly $t = 0$, the potential is zero everywhere except at the walls. As time evolves, the potential between the walls change to accommodate the boundary conditions, and after some finite amount of time (a very, very small finite amount of time since the electric field is propagating at the speed of light), the potential inside the domain is finite everywhere and reaches a steady-state. Each iteration of the Jacobi method gives a snapshot of the potential before that 'steady-state' value is reached, but given enough time, the solution relaxes to a steady, unchanging, value.

2.34.2 Higher-order methods

Generally, Jacobi Relaxation is a very conservative, and thus inefficient, algorithm. In fact, if a toy 2D problem has L points on each side, then the number of iterations required for a given level of convergence goes as L^2. In other words, if we increase the number of grid points by a factor of 2, then the computational effort is increased by a factor of 4. We can get a small amount of improvement by using new information about the function as it becomes available. For example, Jacobi Relaxation for a 2-D problem can be expressed in slightly more detail as:

$$V_{new}(i,j) = \frac{1}{4}[V_{old}(i+1,j) + V_{old}(i-1,j) + V_{old}(i,j+1) + V_{old}(i,j-1)]. \tag{2.14}$$

Here V_{old} is the potential from the previous guess. A different method, the Gauss-Seidel method uses the new values of V as they become available, depending on how you loop through the domain. For example, if you iterate from small i, j to large i, j, then

$$V_{new}(i,j) = \frac{1}{4}[V_{old}(i+1,j) + V_{new}(i-1,j) + V_{old}(i,j+1) + V_{new}(i,j-1)]. \tag{2.15}$$

Employing the Gauss-Siedel method improves the speed of convergence by a factor of 2, which is nice, but not great. A more widely used method that can improve convergence speed by an order of L is the method of *simultaneous over-relaxation* (SOR), which uses a problem-specific over-relaxation factor that maximizes the efficiency of the scheme. There is plenty of discussion on how to implement this method in the literature so it will not be discussed here. However, the basic premise is much the same as that of Jacobi Relaxation and the Gauss-Siedel method.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search