The basics of Linux and programming languages

We will be using the Linux operating system in this course. The chief advantages of Linux are the ease with which you can do complicated things, and the large amount of high quality free software available. You will also probably find that certain things run a bit faster and more smoothly, and there is also the advantage that there are almost no viruses being written for Linux because it isn't used by enough people to justify the effort by the virus programmers[1]. The advantages relative to MacOS are quite a bit smaller than the advantages it has relative to Windows. Relative to MacOS, the main advantage is that the hardware on which it runs is cheaper.

Linux has a much wider range of command line tasks available than Windows does. It is also a little bit more straightforward to make what are called "scripts" in Linux than in Windows, although this is possible in both operating systems. We will not get into scripting in the course, but you may eventually want to do it. It is a handy way to do things like run the same program repeatedly with different parameter values, or run through a long list of tasks. Installations of Linux also have a window manager, and you can still do most of the same thing by pointing and clicking in Linux that you can do in Windows when pointing and clicking is easier.

The first thing to do with Linux is to open up a terminal window. [figure out how we do that in that particular flavor].

Next, let's also open a web browser. Go to my course web-site:

and download the file helloworld.cc

Now, let's move the file. In Linux, we use the term *directory* to refer to a place where a group of files are located. It's essentially the same as a folder on a Mac or a Windows machine, but we just use a different name for it. In the window managers, they'll be represented with icons that look like folders.

To change directories to the directory called *directory*, we type:
`cd` *directory*

To copy files, we use `cp` *filename new_filename*.

To move files, we use `mv` *filename new_filename*.

There are a few key shortcuts for locations on the computer. The current directory is represented by a single dot (.), and the directory one level up the tree is represented by double dots (..). The asterisk key is a wildcard.

So, to copy all files from the previous directory to the current one, we can type: `cp ../* ..` If we only wanted files starting with an "a", we could type: `cp ../a* ..`

The other key command that we need to learn is the command to list files: `ls`.

And one final command to learn is the `man` command, which gives the manual page for other commands. A lot of the commands above have options. For example, `ls -l` gives a lot more information about each file than `ls` does. If you type `man ls`, it will give you the manual page for using the `ls` command.

[1] And, also, Linux users tend to be the most computer-savvy computer users.

There are a ton more things you can do in Linux that I'm not discussing here, but what I've told you so far will give you a decent foundation for doing the basics. Eventually, at some point, you may want to learn to use tools to write scripts in Linux (sets of tasks you run one after another), or use commands like `awk` which can manipulate files in a convenient way. With the foundation from this course, you should be able to teach yourself whatever else you need to know in the future.

Programming languages

First generation languages

Computers work with binary codes. Sending different sets of "bits" to a computer's central processing unit will give it different instructions about how to move other bits around in its memory or how to apply mathematical operations to the bits in its memory. The most "direct" way to program a computer, then, is to feed it a series of binary codes that tell it exactly what to do. This is how the earliest computers, which were mechanical devices, worked, and for some very specific applications, binary coding is still used. Machine language programming is called a "first generation programming language".

Second generation languages

The next step was to develop tools that allowed programmers to write programs in something resembling plain English, and to be able to translate those programs into machine code. These "second generation programming languages", are often referred to as Assembly (since the code needed to be assembled into machine language). Assembly is generally machine-specific – that is, the same program will not work on a PC and a Mac. This has major disadvantages, of course, in that it requires the programmer to write separate programs for different types of computers. It has some major advantages for some very specific types of programming, in that it makes it possible for the programmer to do some non-standard things that will allow the program to run an the very fastest possible speed on that particular system. Getting the graphics processing to run smoothly in video games is one particular application of assembly language that still exists.

Compiled versus interpreted languages

Computers will only actually run machine language codes. There are two ways to generate these – through the use of compiled languages and interpreted languages. Compiling a computer program means taking the full program and converting it from a language similar to plain English into machine code. Interpreting a program means doing this line-by-line. The programming process is often a bit easier for an interpreted language – especially the process of going through and finding mistakes. The big disadvantage of interpreted languages is that they are often 30-100 times slower for the same task – for certain types of tasks they spend far more time translating the same line of your program into machine language than they do executing the tasks you're interested in.

Third generation languages

The major breakthrough was the development of third generation programming languages. These are languages that look something like plain English *and* can be transported from one operating system to another. They are writ-

ten with the aim of being broadly applicable to a range of types of programs, rather than with a few specific applications in mind. In this course, we will use the C programming language. It is one of several third generation languages widely used in physics. The other two are Python and FORTRAN.

Why C?

The C language is sort of a compromise choice among the three languages. I wanted to give you some experience with a compiled language, so that if you eventually need to work on a problem that will be very computationally intensive, you can do it. I also wanted to give you experience to a language that is widely used outside academia, so that you will have better job prospects when you finish your degree if you decide not to go to graduate school.[2]

Python has some properties that allow it to do certain mathematical tasks in ways that are much easier to program than in C or FORTRAN (e.g. you can apply an operation to a vector in one line of code, instead of having to go through all the elements of the vector), but it is much slower once it's running than FORTRAN or C. Eventually if you learn both C and Python, you can decide which you will use based on how long you expect your program to spend running. Generally speaking, C or FORTRAN programs will run about 30-100 times faster than python programs for the same job. If the job will take an hour in python, then it will take a minute in FORTRAN or C. If you know you only need to run the program once, and it will take an extra half hour to write the program in FORTRAN or C than in Python, then you are probably better off with Python – you can take a coffee break or work on another project while it's running. On the other hand, if you will have to run that same program 100 times, it may be worth that extra half hour of your human time to write the program in FORTRAN or C.

Fourth generation languages

Fourth generation programming languages also exist. These are languages designed to be very easy to program, but for a fairly narrow range of tasks. Some examples include MATLAB, which is designed for engineering purposes, and R/S which are languages optimized for statistical analysis. Both of these can be of some value to physicists, but whether they are in a particular case depends on how big your computational project is and how well matched it is to the computer language you're using. For example, MATLAB has a scripting language, which is an interpreted computer language, and it has some compiled routines. If you need to do tasks that can be done mostly by MATLAB's compiled routines, then the relative ease in developing the program, coupled

[2]FORTRAN has historically been a little better for certain kinds of computational projects than C because it gives the user a bit less freedom to manipulate the computer's memory – meaning that the compiler can assume certain things to be true and not use computational power checking that they are. There are now ways to force C compilers to assume the same things, removing much or maybe all of FORTRAN's advantages. FORTRAN is still a little easier to program for a lot of tasks, and there are some very valuable old programs in FORTRAN that are still modified and used. It's not dead yet. C has a much better system set up for user-interfaces, which is its main advantage – and it also caught on with a greater level of popularity in computer science departments in the 1980s and 1990s, which is why it's so widely used now.

with what are sometimes better algorithms for doing the numerical tasks, may make it advantageous to use MATLAB. If you need to do most of your tasks under the MATLAB scripting language, which is interpreted, then you will probably find your programs running extremely slowly. The other disadvantage of many of the 4th generation languages is that they have expensive licensing agreements (although there are some open-source, free alternatives for the most prominent ones, which can provide nearly all of the same functionality).

Different levels of languages: an analogy to eating

I'll lay out a good, if slightly imperfect analogy for you.

1. First level languages are like cooking food that you grew yourself. It's pretty hard to do anything complicated, but you really know what's going on with everything.

2. Second level languages are like picking food, then cooking it. It's a lot easier than the former, and you get pretty much all the same choices.

3. Third level languages are like cooking for yourself after buying food at the grocery store. You might lose out a little bit in terms of just how well you can do what you want, but it's a lot easier process.

4. Fourth level languages are like eating at a restaurant. It's a lot less work for you, but it costs more. You also have a lot less control over how things are done. Sometimes the expertise of the person doing the work for you is high enough that it allows you to try something you couldn't try otherwise. Sometimes it just takes longer and isn't any better. There are some things which are 4th level languages that are aimed at specific applications in the pure sciences which are free – but if it's something engineers would also be interested in using, it will often carry a charge. There are also knock-off versions of many of these languages which are free.

Which type of language you should use is something that you will eventually have to choose for yourself in a variety of different situations. For a lot of you, after you settle into a career, you'll find a fourth-level programming language like MATLAB to be what you use most of the time. That's fine, if the MATLAB routines exist for the purposes you have in mind. You'll generally find that you can get the programs written faster in MATLAB than in some other language.

I'm not going to teach you MATLAB, however, for a few reasons. First, you might end up going to a job or to a graduate school which doesn't have a MATLAB license. Second, you might end up needing to write programs to do the kinds of things MATLAB isn't optimized for. And third, it's generally easier to start with more a more general programming language and adapt to a specific one than the other way around.

At the same time, I'm not going to teach you assembly language (i.e. a second level language). The uses of assembly language these days are mostly limited to optmizing codes for a very small range of specific applications. If you think you might like to work in the video game industry, you should look into

that, but it's not really something that is of use to many physicists – I've never used it myself.

C vs C++

We will use the C++ compiler g++ for turning our programs into machine code. The C++ compiler is "backward-compatible" with C programs – i.e. it can compile programs written in standard C. It also has structures set up to allow the use of a technique called object-oriented programming. The object oriented approach can sometimes be better for very large group projects, but is generally a bit more burdensome for doing small projects on your own or in small groups. I will not teach any object-oriented techniques to you, so effectively I will be teaching you to program in C, even though we will be using the C++ compilers.

An aside: LaTeX

Many of you may be impressed with how nice the equations look in the notes I will send around to you. I am using a software package called LaTeX for writing them. It is the standard way to write scientific papers for most physicists, astronomers, mathematicians and computer scientists. For papers with a lot of math in them, it can be a lot easier to work with than the Microsoft equation editor. I am not going to teach LaTeX in class, because it's one of these things that you can learn by doing once you know what it is, and it's not really computational physics, but if you want to see a few sample copies of the original files of my notes so that you can start teaching it to yourself, please feel free to ask me for them. I didn't start learning it myself until grad school, so don't feel like this is something you really need to do.