Introduction to Computing Part I

In this lab you will log into a Linux workstation, make a subdirectory, discuss a problem to solve, use an editor to create a program, compile the program and run it.

Unix was developed in 1971 by Thompson, Ritchie, Ossanna and Canaday at Bell Labs and publicly released in 1975. Because U.S. Federal Law prevented Bell Labs from selling products due to its status as a unique, monopoly institution, Unix was made available at no cost with complete source code. In 1982 the Department of Justice broke AT&T into several firms. Corporate interests combined with intellectual property laws soon made the original Unix system very expensive.

In 1984 Stallman began the GNU project to create a new operating system with a license that guaranteed individual programmers would have the right to extend and modify the software as needed. The remaining kernel needed to complete the GNU operating system was provided by Torvalds in 1991 at the University of Helsinki and called Linux. The Linux kernel consists of about 15 million lines of code while the rest of the GNU operating system adds another 15 million lines. A typical Linux workstation includes an additional 40 million lines of code devoted to mouse interfaces, web browsers and office applications.

Currently 94 percent of the worlds top 500 supercomputers run the GNU/Linux operating system. Linux is also widely used for scientific computing on personal computers. Most scientific codes are written in Fortran. Over the years Fortran has been extended to include support for structured, object oriented and parallel programming. It has all the features of a general programming language, however, it still remains a specialty language used almost exclusively by scientists.

We will focus on C because it is more generally used in the software industry and because it is just as good as Fortran from a suitability, availability and performance point of view. In particular, the C99 standard includes native complex data types and variable length arrays, while the MIT/Intel CilkPlus extensions add support for SMP parallel programming. Given the role of Linux in scientific computing, we will use Linux for this course and write our programs in C.

Logging In

The computers in the ECC computing lab have been set up to accept your UNR netid and password

```
http://www.unr.edu/it/login-ids/netid
```

as login credentials. Please enter your netid and password in the login manager window. All students, whether registered under the Math prefix or the CS prefix of this course, should be able to log in. If you can't log in, try again making sure that the keyboard caps lock is turned off. If you still can't log in, partner with someone who can log in for today.

If you were unable to log in, please verify your netid over the weekend and send a polite email to the system administrator

Ronald Ray `<rlray@unr.edu>`

indicating that you are registered for Math/CS 466/666 and that your netid needs to be added to the list of valid accounts in the lab. Please send an email to me at `ejolson@unr.edu` if you continue to have difficulties.

Making a Subdirectory

Once logged in, open a terminal window. Depending on which mouse interface comes up by default, opening a terminal window will either be obvious or not so obvious. The strategy of mousing around until you find an icon that looks like terminal and clicking on it, usually suffices. If you have figured out how to open a terminal window and your neighbor is still searching, please help him or her. If you are searching and you notice your neighbor has a nice terminal window open already, please ask for help. However, please hold off all talking when the teacher is telling something to the class.

Unix pioneered the idea of a hierarchical filesystem that can be used to organize the operating system and your own work, as well as the idea of a working directory that defines your current workspace. Most mouse interfaces refer to subdirectories as folders, but lack the notion of a working directory. The prompt on your terminal should look like

```
netid@hostname:~$ _
```

where `netid` is your UNR netid, `hostname` is the name of the workstation you have logged into and `:~` indicates the current working directory. The working directory is currently the same as your home directory.

Make a subdirectory by entering the command `mkdir lab01` as shown in the following line:

```
netid@hostname:~$ mkdir lab01
```

To check the subdirectory was actually made, enter `ls` as

```
netid@hostname:~$ ls
lab01
```

If you do not see `lab01` displayed immediately beneath your command, something went wrong. Try making the subdirectory again until you get the correct output. Now change the working directory to be the subdirectory.

```
netid@hostname:~$ cd lab01
ejolson@remote:~/lab01$ _
```

Note that the prompt has now changed to indicate that your current working directory is `lab01`. You may also check what the current working directory is with the command `pwd` as

```
netid@hostname:~/lab01$ pwd
/nfs/home/netid/lab01
```

The string `/nsf/home/netid/lab01` is called the absolute path of your working directory. It may be slightly different than the one indicated here, but should, at least, end with `lab01`.

A Numerical Problem to Solve

Before writing a program, we need a problem to solve. As this is a course on scientific computing, the problem should involve computing something. Here is the problem.

Problem 1: Write a program to compute

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

for $x = j/10$ where $j = 0, 1, 2, \ldots, 10$.

This integral can't be expressed in terms of the elementary functions; however, the exponential function has a convergence power series

$$e^z = \sum_{k=0}^{\infty} \frac{1}{k!} z^k \qquad \text{so that} \qquad e^{-t^2} = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!} t^{2k}$$

and consequently integrating term by term

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \sum_{k=0}^{\infty} \frac{(-1)^k}{k!} t^{2k} dt = \frac{2}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k}{k!(2k+1)} x^{2k+1}.$$

It follows that we can approximate

$$\operatorname{erf}(x) \approx \frac{2}{\sqrt{\pi}} \sum_{k=0}^{n} \frac{(-1)^k}{k!(2k+1)} x^{2k+1}$$

for n large. Theoretically, the larger we take n, the better approximation. However, if x is also large, then we need to take n very very large. But then the fact that computer arithmetic works only with finitely number of significant digits makes a mess of the approximation. This is called loss of precision and will be discussed in class later.

As is often the case with problems that students are asked to solve, this exact problem has been solved before. In particular, there is already a function called `erf` built into the C programming language. The fact that a program to compute $\operatorname{erf}(x)$ has already been written doesn't make writing another program of any less educational value. The thing to remember is that we are solving this problem to learn something about numerical methods and scientific computing, not because this problem is new, original or really needs to be solved again. The ideas, techniques and methods which are taught by solving the problem are much more important than the actual answer. Of course, if we don't get the correct answer, then the only thing we can claim to have learned is that it is easy to get wrong answers. We'll use the built-in function to get started with our program.

Create A Program

In this section we use an editor to create a program. If you are familiar with standard editors such as `vi`, `joe` or `emacs` feel free to use your preferred editor. Otherwise use `gedit`. Do this because `gedit` interacts with the mouse in the least astonishing way. Create a file called `main.c` using

```
netid@hostname:~/lab01$ gedit main.c &
```

This will open a new window on your workstation. Don't forget to type the `&` sign at the end of this command. Otherwise, `gedit` will lock your terminal window and you will have to exit the editor to use it again. Of course you can always open more terminal windows if needed, but don't do that now. After starting the editor properly, move your mouse cursor to the editor window and, if necessary, click on the window to focus it.

It should be pointed out the traditional mouse interface under X Windows uses a system called focus follows mouse to decide which window receives input from the keyboard. This is different than click to focus used by Microsoft Windows and Apple Macintosh systems and likely reflects the fact that Unix was a multi-tasking system from the beginning. At anyrate, to accomidate beginners, it is possible your Linux session will default to click-to-focus compatibility mode. Feel free to change the preference.

Let's develop the program in stages: First we'll write the main routine. Then we'll add a subroutine to compute the approximation for the integral. Here is the first version of the code:

```
#include <stdio.h>
#include <math.h>
int main(){
    printf("%5s %20s\n","x","erf(x)");
    for(int i=0;i<=10;i++){
```

```
 6          double x=i/10.0;
 7          printf("%5g %20.12e\n", x, erf(x));
 8      }
 9      return 0;
10  }
```

Note the numbers to the left of each program line are for reference and should not be typed into the editor. If you would like these line numbers to appear in `gedit` then select

Edit → Preferences → View

with your mouse and place a check next to the Display line numbers option. Okay, now you have to type the program in.

While typing may seem tedius, scientific computing is a literate activity that involves both reading and writing computer programs. Writing anything on a computer involves typing. If you have not learned typing, consider this another reminder that it is time to learn. Make sure you save the program after you are done typing it in. Do this by selecting

File → Save

with your mouse.

While our C code may be self-explanatory to many, if you have never written a computer program, that is likely not the case. I will describe each line of the program, but don't panic if you don't understand everything exactly. It typically takes about a month to get used to C programming. The effort is worth the time, because C-programming skills are valuable for scientific computing, for computer science in general and in many other fields of industrial research and development.

Line 1: The program will access the standard library for input and output. In particular, this is for the `printf` function.

2: The program will use the standard mathematics library. This is for the `erf` function.

3: All programs have an `int main()` routine.

4: This line prints a key for output in line 7. This formatted print statement reserves 5 characters for printing the string `"x"` and then 20 characters for printing the string `"erf(x)"`. The widths 5 and 20 are the same as used on line 7.

5: This is a `for` loop for a counter called `i` which starts at 1 and takes values up to and including 10. Note that C99 allows the declaration `int i` within the `for` loop in a way similar to C++.

6: If we wrote `double x=i/10` with the decimal point missing, then `x` would be rounded down to the greatest integer less than or equal the quotient, which is not what we want.

7: The format specifier `%5g` indicates that 5 characters should be used for printing `x` in a general format; the specifier `%20.12e` indicates 20 characters should be used for printing `erf(x)` in an exponential format with 12 digits after the decimal point.

Compiling the Program

At the beginning of 1st grade my teacher wrote a letter of the alphabet on the black board and everyone copied that same letter again and again onto their papers. We learned a couple of letters a day. Within a month we were writing our own sentences, paragraphs and stories. You have copied the program letter by letter into the computer. This is how I learned to write English. This is also how I learned to write computer programs. As far as I know, the only way to learn these kind of technical skills is by doing.

So far we've used the C programming language to express a sequence of steps that we want the computer to perform. We shall refer to such a sequence of steps as an algorithm, numerical code or a computer program. As you gain experience, you will be able express your own thoughts and algorithms in the C programming language. First, let's see if you managed to type the program in correctly.

Return to the terminal window and type `ls` to make sure you saved your file. If all goes as planned, you should obtain the output

```
netid@hostname:~/lab01$ ls
main.c
```

We shall use the GNU C compiler. The compiler will read the file `main.c` which we just created, and hopefully produce another file, called the executable. This executable contains the instructions needed by the hardware to perform the algorithm described by the C program we wrote. However, if there are errors in our file, the compiler will report those errors instead of producing an executable. Type the following to compile the program.

```
netid@hostname:~/lab01$ gcc -std=gnu99 main.c -lm
```

Hopefully there won't be any errors. At this point it is likely that about half the people in the classroom will have errors. Errors issued from the compiler direct you to take a closer look at your C source code.

If there are errors, go back to your `gedit` window and search for any difference between what appears there and what you were supposed to type. The messages issued by the compiler may contain a line number or description that helps focus where to look. If you don't have errors, please help your neighbor find their errors. Every semicolon and every parathesis is important. Also check that your neighbor didn't type the letter `O` for the number `0` or the letter `l` where the number `1` belongs.

Depending on your luck and ability to notice small details, finding the error could be easy or very difficult. As before, please hold off all talking when the teaching is telling something to the class. If everything else fails, copy my version of the program directly into your directory using

```
netid@hostname:~/lab01$ cp /nfs/home/ejolson/lab01/main.c main.c
```

After the program compiles with no errors, type `ls` to check that the executable was produced.

```
netid@hostname:~/lab01$ ls
a.out  main.c
```

The executable is called `a.out`. We are almost done for today.

Running the Executable

Notice that it took almost an hour for us to do something that would take an experienced programmer less than 5 minutes. However, within a month we will be discussing C programming techniques for scientific computing new to most people in the class. Among other things, we will use the native complex data types and variable length arrays which are part of the C99 standard as well as the MIT/Intel CilkPlus extensions which add support for SMP parallel programming. Before leaving, let's run the program.

```
netid@hostname:~/lab01$ ./a.out
```

The output should look like

```
  x               erf(x)
  0   0.000000000000e+00
0.1   1.124629160183e-01
0.2   2.227025892105e-01
0.3   3.286267594591e-01
0.4   4.283923550467e-01
0.5   5.204998778130e-01
0.6   6.038560908479e-01
0.7   6.778011938374e-01
0.8   7.421009647077e-01
0.9   7.969082124228e-01
  1   8.427007929497e-01
```

Next Friday we will write our own subroutine to approximate erf(x) by summing the series. Have a good weekend and see you on Wednesday.