

Assignment 1

Getting Acquainted with UNIX and C

Prof. Darrell Long
CSE 13S – Winter 2023

Due: January 22nd 2023 at 11:59 pm

1 Introduction

The purpose of this assignment is to get you acquainted with a UNIX system and to get familiar with the syntax of a basic C program. You won't need to write any C code for this assignment, but it is expected that you go through and understand the code given to you. This assignment itself will require the use of bash, a general purpose scripting language. You may wonder why, especially in a course dedicated for introductory C programming, you're expected to learn the basic of bash, but it will prove useful to you as a programmer down the road and will aid in getting you comfortable with working in a UNIX command line, or a *shell*.

2 UNIX and the Shell

To understand why a *shell* is used to interact with UNIX, we need to first delve into why UNIX was developed the way it was. The development of UNIX started by Ken Thompson and Dennis Ritchie, with contributions by many others, for use at Bell Laboratories (Bell Labs). The development continued throughout most of the 70s, eventually leading to the licensing of UNIX to other parties, especially universities in the late 70s.

Along with UNIX came the UNIX philosophy. This philosophy is based on the belief that a tool should have a limited, yet specific function. All the tools on the original UNIX system shipped with this philosophy. Examples include `head`, which prints out the first n lines of a file, `ls`, which lists out files in a directory, and `diff`, which reports differences between two files. The idea is that the output of one tool, or *command*, can become the input of another command.

Commands are the main way that users are intended to interact with UNIX. Each command follows the pattern of `command [arguments]` (here the square brackets mean the arguments are optional). Each command can be executed an optional number of arguments. Consider the following command:

```
$ wc -l roster.csv
```

This particular command uses the `wc` command, which is a program that counts the number of words, lines, and characters in a file. The argument `-l` tells `wc` to only consider the line count. The final argument, `roster.csv`, is the file to get the line count from. Note the `$` prefixing the command. The `$` is not itself part of the command, but is commonly used to indicate that the command is run in a *shell* and acts as a *shell prompt*.

A shell is a program (just like any other) that runs in an infinite loop, parsing and executing commands. In other words, it acts as a *command line interpreter*. Examples of popular shells include `bash` and `zsh`. Both these shells have two modes of operation: interactive and batch. Using a shell in interactive mode means issuing commands one at a time, with the result of the command appearing immediately after the command is executed. Using a shell in batch mode means giving the shell a *series* of commands to run in sequence. Typically this means placing all the commands in a file. Files that contain a series of commands to run are typically called *shell scripts*. You will be writing a shell script for this assignment.

3 Gnu Plot (gnuplot)

`gnuplot` is a utility that can be run on a variety of platforms (Linux, macOS, Windows, and many more) used to generate high-quality plots and graphs. Unlike Excel, `gnuplot` is used through the command-line—the shell. Although you can use it interactively, issuing commands and having the current plot be redrawn, `gnuplot` is capable of running commands in batch, lending itself very naturally for use in a bash script. To install `gnuplot` on Linux:

```
$ sudo apt install gnuplot
```

`gnuplot` is mostly used to plot data points in a file. The basic format is very simplistic: just a series of (x, y) coordinates on each line, separated by whitespace.

Example data file: `example.dat`.

```
1 5
2 4
3 3
4 2
5 1
```

After you've created this file, start up `gnuplot`. This should open up an interactive prompt. From here, all we need to do is specify what data file to plot and how to plot it.

```
$ gnuplot
```

```
GNUPLOT
Version 5.4 patchlevel 2    last modified 2021-06-01
```

```
Copyright (C) 1986-1993, 1998, 2004, 2007-2021
```

Thomas Williams, Colin Kelley and many others


```
gnuplot home:      http://www.gnuplot.info
faq, bugs, etc:    type "help FAQ"
immediate help:    type "help"  (plot window: hit 'h')
```

Terminal type is now 'qt'

```
gnuplot> plot "example.dat" with linespoints
```

Entering this command into gnuplot should result in the plot popping up in a window. We would also like to be able to export a plot to a file, say a PDF. We'd also prefer if we didn't have to interactively type out what to plot each time. There is where the non-interactive mode for gnuplot comes in. We can create a file listing the commands we'd ordinarily issue to gnuplot interactively and instruct it to output the plot to a PDF of our choosing.

Example gnuplot file: `example.plot`.

A screenshot of a terminal window with a dark background. The prompt is 's'. The commands are: 1 et terminal pdf, 2 set output "example.pdf", 3 set xlabel "x", 4 set ylabel "y", 5 plot "example.dat" with linespoints.

```
s
1 et terminal pdf
2 set output "example.pdf"
3 set xlabel "x"
4 set ylabel "y"
5 plot "example.dat" with linespoints
```

We now redirect the contents of `example.plot` into gnuplot, which should generate the plot in a file named `example.pdf`. The plot should resemble Figure 1. We'll learn more about file redirection when we discuss basic bash scripting in §6.

```
$ gnuplot < example.plot
```

You can learn more about gnuplot and view more extensive examples, respectively, with the following links:

<http://gnuplot.info/documentation.html>

<http://gnuplot.info/demos/>

There is also a built-in help system in gnuplot. Just type “help” when you are in interactive mode.

Do you have to use gnuplot for your course assignments? Yes. Can you use Excel? No. What about some random online tool? No. You have to learn and use a UNIX command line tool. If you are extremely insistent on not using gnuplot, you can use matplotlib for Python as an alternative. Those are your two alternatives.

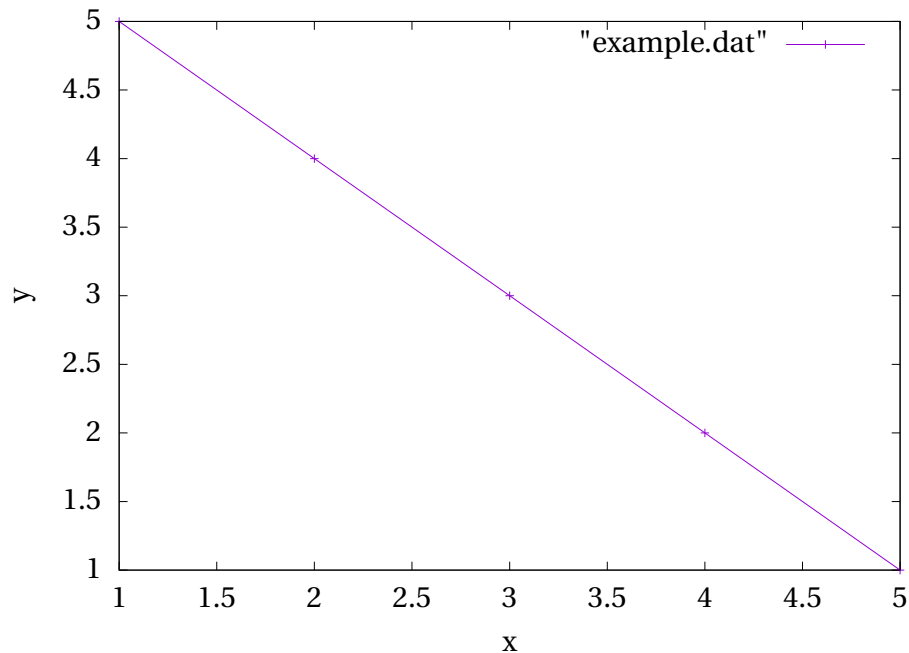


Figure 1: Example plot generated by gnuplot.

4 \LaTeX (pdf \LaTeX)

What is \LaTeX ? It is a tool used by scientists, mathematicians and engineers to produce high quality documents. It is especially useful when your documents have mathematics in them.

Here is an example minimal \LaTeX document. There is a much more complete example in the course resources repository. If your GitLab repository has been created, then you should have access to this repository. It contains the materials for each assignment: header files, source examples, and example programs. You can access the repository here:

<https://git.ucsc.edu/cse13s/winter2023-section01/resources.git>

```
\documentclass[11pt]{article} % Document type
\title{Example Document}
\author{Conrad Veidt}
\date{\today} % Sets the date to \today, or any date you specify
\begin{document}\maketitle % Start the document
For the left Riemann sum, approximating the function by its value at the
left-end point gives multiple rectangles with base  $\Delta x$  and height
 $f(a + i\Delta x)$ . Doing this for  $i = 0, 1, \dots, n-1$ , and adding
up the resulting areas gives

$$A_{\text{left}} =$$

```

```

\Delta x \left[f(a) + f(a + \Delta x)
+ f(a+ 2 \Delta x)+ \cdots+f(b - \Delta x)\right].
$$
\end{document}

```

You will need to install \LaTeX before you can start using it. You will most likely want the full installation:

```
$ sudo apt install texlive-full
```

There is a complete example in the resources repository that you can modify. \LaTeX is a compiled language, just like **C**. That means if you make a syntax error it will complain!

Do you have to use \LaTeX ? Yes, and it works well with `gnuplot`. Life will be much easier if you take a few minutes to learn and use \LaTeX . All you need to do is to mimic what you find in the example documents.

5 Your Task

As stated previously, your task for this assignment is to write a shell script, or more specifically, a bash script. You will use this script to produce plots of a provided **C** program. The provided program can be found in the resources repository.

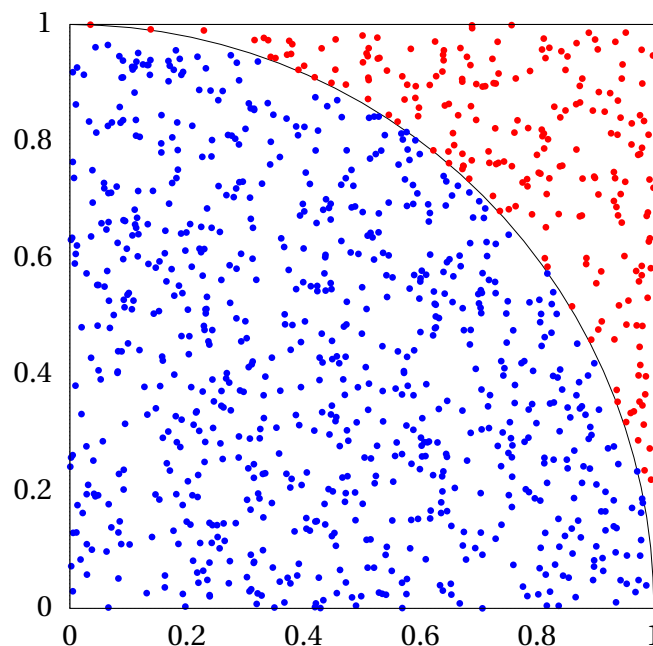


Figure 2: Blue points have distance less than or equal to 1, and hence belong both in the square and the circle, red points are a part of the square but not the circle.

All the program does is print the Monte Carlo estimation for π after each random point it tests. For those unacquainted with Monte Carlo estimation of π , we use a large number of points in a square with an inscribed quadrant. The ratio of the number of points in the quadrant to the number of points in the square is the ratio of the to areas:

- Area of Square with side l is l^2
- Area of a quadrant of a circle with radius l is $\frac{\pi l^2}{4}$
- The ratio of these areas is $\frac{\pi}{4}$

The process is as follows: We will uniformly scatter a number of points in a square, and measure the number of points that fall within the quadrant (distance from origin less than or equal to 1) and the other points. The C program will output the estimated value of π after every point for n number of points. We have provided an example of this scatter in Figure 2.

Using the provided Monte Carlo program, you are expected to:

1. Familiarize yourself with the syntax of C.
2. Understand what the provided program is doing. You will want to attend section for any parts you are unsure about.
3. Produce interesting plots using the Monte Carlo estimations produced by the provided program. You should use `gnuplot` to produce these plots, and everything should be automated by means of a bash script.

{Output for the monte_carlo program}

```
$ ./monte_carlo -n 10
```

Iteration	Pi	x	y	circle
0	4	0.445822	0.55245	1
1	4	0.772659	0.506885	1
2	2.66667	0.999439	0.0812311	0
3	3	0.727855	0.515805	1
4	3.2	0.199699	0.390919	1
5	3.33333	0.579692	0.536917	1
6	3.42857	0.358613	0.367796	1
7	3.5	0.241639	0.905172	1
8	3.55556	0.974999	0.0754683	1
9	3.2	0.862372	0.706275	0

Here is an example output for the provided program up to 10 points. The output provides you with the point number, the estimated value of π at the current point, its x and y coordinates, as well as a 1 or 0 value indicating if it is in the circle, where 1 means it is inside the circle.

Refer to Figures 2, 3 for the example plot that were created using the provided program. Now that you know your task, we will now give a brief overview on bash scripting.

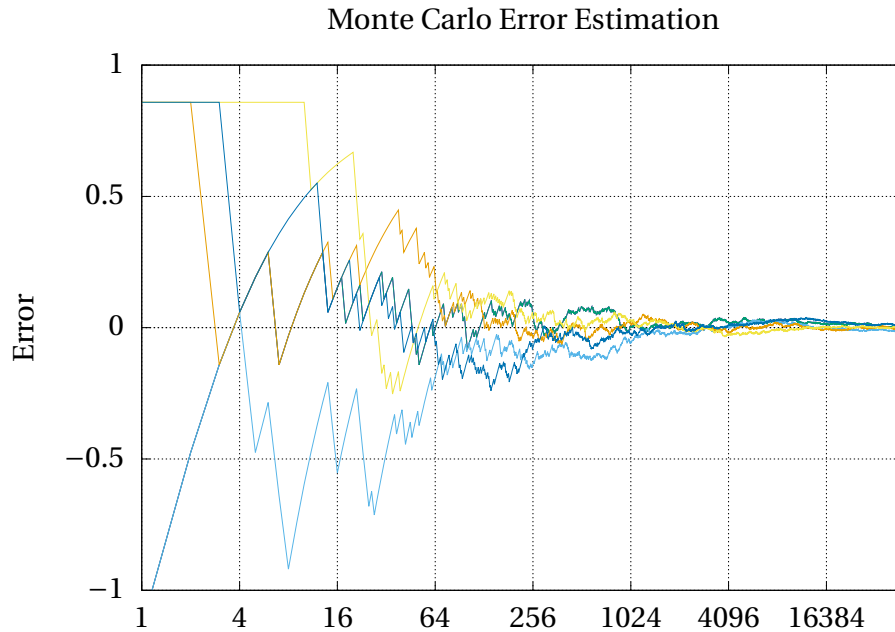


Figure 3: The value of the difference between the estimated π and π_i gets closer to zero as we increase iterations. The different colors represent different seeds for the random number generator.

6 bash

We will preface this section by saying this is by no means the most comprehensive guide to scripting in bash. For such a guide, a copy of the official bash guide can be found here:

<https://folk.ntnu.no/geirha/bashguide.pdf>

The amount of bash knowledge needed for this assignment is very minimal, but will be crucial to your career in Computer Science. We will start with issuing commands in bash. **You will want to study this section and reference it in the future.**

6.1 Commands and Arguments

Commands in bash are made of up words delimited by whitespace, where a word is just a sequence of characters. Most commands usually span a single line, but it is possible to write a more complex command that spans across several lines. As stated in §2, a command follows the pattern `command [arguments]`. Consider the following command:

```
$ head notes.txt
```

bash parses this specific command into two words: `head` and `notes.txt`. The command to execute is `head` and its sole argument is `notes.txt`. The program `head`, by default, prints out

the first 10 lines of a specified file, in this case `notes.txt`, to *standard output*, or `stdout`. This is what you think of as the console, or terminal window. Any UNIX process has the following three open input/output streams open at creation:

1. Standard input (`stdin`) – Unix file descriptor number 0
2. Standard output (`stdout`) – Unix file descriptor number 1
3. Standard error (`stderr`) – Unix file descriptor number 2

What is a UNIX process? We'll learn more about that later on in the course, but for now you can think of it as a program in execution. We'll also discuss input and output more in-depth later on in this section. Returning back to the discussion about arguments, it's important to note that there can be any number of arguments to a command. For instance, the `head` program also accepts arguments to specify how many lines of a file to print out. For example, to print out the first 2 lines of `notes.txt`:

```
$ head -n 2 notes.txt
```

There's not a whole lot more to discuss regarding commands and arguments other than the different kinds of commands: aliases, functions, built-ins, keywords, and executables. We will only concern ourselves with built-ins and executables for now, as they are the bare minimum needed to get moving around in UNIX and writing bash scripts.

6.2 Built-ins

Built-ins are functions that are built into bash. A bash function can accept arguments and is composed of a series of commands to execute. For example, here is a function called `mkcd` that creates a directory and enters the created directory:

```
$ function mkcd { mkdir $1 && cd $1 }
```

Let's break down this function a bit more. `mkdir` is a utility that creates a directory. A directory is the equivalent of a Windows folder. The `$1` accesses the first argument passed to the function. This operates by parameter expansion (more on this later), as commented in the following example:

```
$ mkcd directory # Runs "mkdir directory && cd directory".
```

The `&&` is a *control operator* and indicates logical AND. So basically, make the directory with `mkdir` *and* enter it with `cd`. `cd` is arguably one of the most important built-ins since its sole purpose is to move around directories. `echo` is another built-in.

6.3 Executables

An executable on UNIX is just another name for a program. The terms *program*, *executable*, *executable binary*, and sometimes just *binary*, all refer to a program that can be run, or *executed*. The distinction between an plain old executable and an executable binary is in its representation. An executable binary is platform specific and is simply a file full of machine code: pure binary. An executable could be a shell script, which contains readable text. The only thing these two share is that they must have the executable bit set in order to be run. Refer to the discussion of UNIX file permissions in assignment 0 if you have forgotten what the executable bit refers to.

6.4 Parameters

Parameters in bash store values. There are two kinds of parameters: *variables* and *special parameters*. Variables are used to store values specified by the programmer. Special parameters have values that are set by bash itself. The value of a parameter can be accessed via *parameter expansion*, which just means prefixing a parameter's name with a \$. A parameter name can consist of letters, numbers, and underscores, but cannot start with a number. An example of bash variable assignment and parameter expansion:

```
$ greeting="Bonjour"
$ prime=13
$ echo $greeting # Prints "Bonjour".
```

Note that bash *does not* allow spaces around the assignment operator due how words are delimited by whitespace. Adding spaces would cause the assignment to be parsed as three different words instead of one. Special parameters include @ and ?, and even 1 that was used in an earlier example. Each of these can also be expanded. @\$ expands to all the arguments passed to a command. \$? expands to the exit status of the last run command. This is typically 0 for a successful exit, and non-zero for an unsuccessful exit. \$1 expands to the first argument supplied to a command.

6.5 Conditionals

Conditionals allow programming languages to handle decisions and take form as expressions that can be evaluated as either true or false. bash uses a command called test to test whether or not an expression is true. If an expression is true, test exits with an exit code of 0. Otherwise, it exits with an exit code of 1. For example:

```
$ test "hello = hello"
$ echo $? # Prints out last command's exit code, which is 0.
$ test "hello = goodbye"
$ echo $? # Prints 1.
```

bash, unlike several programming languages, uses = instead of == to test for equality. As you may expect, bash also provides if-else statements.

```
$ if [ -f mystery ]; then echo "file"; else echo "directory"; fi
```

This example tests if `mystery` is a file. The square brackets indicate a test (`[` is a reserved keyword for the test command), and `-f` tests if its argument is a file. `bash` has a more robust keyword for testing, `[[`, which should be preferred over `[` since it provides more functionality.

```
$ if [[ -d mystery ]]; then echo "directory"; else echo "file"; fi
```

6.6 Loops

Loops allow for commands to be executed multiple times in succession. We will discuss the `while` and `for`-loop in `bash`, since they also appear in `C`. The `while`-loop loops until a command exits unsuccessfully.

```
$ while true; do echo "infinitely looping"; done
```

There are two `for`-loop flavors. One of them closely resembles a `for`-loop in `C`.

```
$ for (( i=0; i < 10; i++ )); do echo $i; done
```

What is the `((...))`? That is an *arithmetic context* in `bash`. `bash` inherently handles variables as just text. Try the following example:

```
$ x=3
$ y=29
$ if [[ $x > $y ]]; then echo "x > y"; else echo "x <= y"; fi
```

Why does it print `"x > y"`? Because it compared `x` and `y` *lexicographically*. To test things numerically, we need to use an arithmetic context.

```
$ x=3
$ y=29
$ if (( $x > $y )); then echo "x > y"; else echo "x <= y"; fi
```

You can think of an arithmetic context as just an environment in which numerical operations such as addition and multiplication can be performed. This is needed for the `for`-loop because `++` is the *postfix increment operator*. You will learn more about the differences between the postfix and prefix operators when you start writing `C` programs.

The other `for`-loop in `bash` uses *brace expansion*, like so:

```
$ for i in {0..9}; do echo $i; done
```

`bash` handles `{0..9}` by expanding it to the list of words `0 1 2 3 4 5 6 7 8 9`. The variable `i` is then assigned to each expanded word in sequence. The `..` indicates a *range* in `bash`. Ranges in `bash` are inclusive.

You can iterate over more than just integers in `bash`; you can even iterate over all the files in the current directory that end in `.txt`.

```
$ for f in *.txt; do echo "$f seems like a text file"; done
```

The `*` used in the `for`-loop is a *glob*. Globs are used for matching patterns. The `*` glob in particular matches any string. Thus, `*.txt` matches any string as long as it ends with `.txt`. Globs are very closely related to *regular expressions*, which you'll become acquainted with later on in the course.

6.7 Input and Output

As stated earlier, any UNIX process starts up with `stdin`, `stdout`, and `stderr`. We described them as input/output streams earlier, but that was a slight lie. To be more exact, these are all *file descriptors*. File descriptors are simply integers that act as indices into a table of open files for a process. By default, `stdin` is file descriptor number 0, `stdout` is file descriptor number 1, and `stderr` is file descriptor number 2. `stdin` is where input goes, such as a typed password when a program prompts for one. `stdout` and `stderr` both print to the console, with the only difference being that `stderr` is usually designated for error messages. `bash` has two main mechanisms to provide input/output redirection: *file redirection*, and *pipes*.

6.7.1 File Redirection

File redirection works for both input and output. That is, you can specify that a program take input from a file (redirecting `stdin`) and place its output into another file (redirecting `stdout`).

```
$ head < full.txt > first10.txt
```

This example first redirects the `stdin` of `head` to `full.txt`. In other words, it feeds the contents of `full.txt` to `head`. Then, the `stdout` of `head` is redirected to `first10.txt`. This takes the output produced by `head` and places it in `first10.txt`. As you can imagine, `<` is the input redirection keyword and `>` is the output redirection keyword. To redirect `stderr`, prefix the output redirect keyword with the file descriptor of `stderr`: 2.

```
$ head < full.txt > first10.txt 2> errors.txt
```

We can build on this example further, redirecting `stdout` to `/dev/null`, then redirecting `stderr` to what `stdout` is pointing at. `/dev/null` can be thought of as a black hole: it consumes anything that is written to it. Note that `>` by default redirects `stdout`. We could also manually specify to redirect `stdout` with `1>`.

```
$ head < full.txt > /dev/null 2>&1
```

6.7.2 Pipes

It is with *pipes* where you can truly see and appreciate the beauty of the UNIX philosophy. Pipes, originally called *hoses*, allow for the `stdout` of one process to be connected, or piped, to the `stdin` of another process.

```
$ ls | sort -r
```

This is an example of a UNIX *pipeline*: two or more commands piped together. This particular pipeline lists the contents of the current directory in reverse lexicographic order. Hopefully it's easy to see why the UNIX philosophy is so loved and still widely used today. Just consider the following example, which prints out your most commonly used commands and the number of times you've used the command.

```
$ history | awk '{ $1=""; print $0 }' | sort | uniq -c | sort -nr
```

6.8 Here-Documents

A *here-document*, or *heredoc*, is another way of passing data into a command. This is typically used if you don't have too much information to pass along to a command and don't want to write it to a file first and then redirect it. We'll use an example of a here-document in the next section when we tackle a simple script using `gnuplot`.

6.9 A Simple bash Script

We can now start to write bash scripts. We'll start with a simple script that plots the function $\sin(x)$ using values of x within the range $[-2\pi, 2\pi]$ in increments of 0.01. This script is provided as `plot.sh` in resource repository. Refer to Figure 4 for what the produced plot should resemble.

```
plot.sh
1 #!/bin/bash
2
3 make clean && make sincos # Rebuild the sincos executable.
4 ./sincos > /tmp/sin.dat   # Place the data points into a file.
5
6 # This is the here-document that is sent to gnuplot.
7 gnuplot <<END
8     set terminal pdf
9     set output "sin.pdf"
10    set title "sin(x)"
11    set xlabel "x"
12    set ylabel "y"
13    set zeroaxis
14    plot "/tmp/sin.dat" with lines title ""
15 END
```

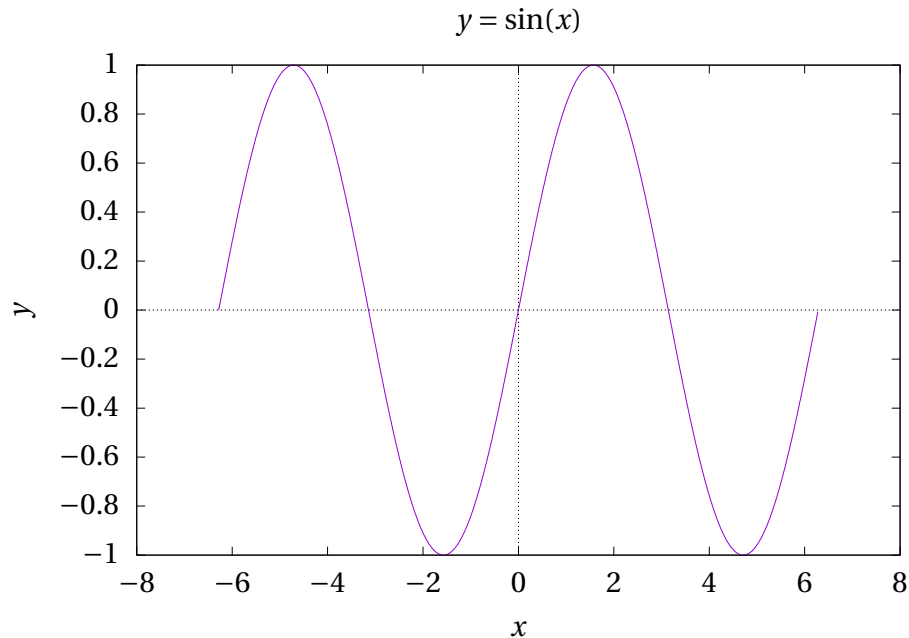


Figure 4: Values of $\sin(x)$ over x in $[-2\pi, 2\pi]$.

The first line, although it starts with the comment character `#`, is imperative. It acts as the *interpreter directive*. When `plot.sh` is executed, the first two bytes of the file, `#!`, indicate that `plot.sh` is an executable script and the program to use to interpret the script is `/bin/bash`. To set the executable bit of `plot.sh`, use `chmod`:

```
$ chmod +x plot.sh
```

To run the script:

```
$ ./plot.sh
```

The syntax for a here-document may look a little strange. An identifier is required to delineate the start and end of the here-document. In the case of `plot.sh`, the identifier `END` is used. Any identifier is fine as long as the contents of the here-document don't also use the identifier. It should be clear that `gnuplot` is the intended destination command for this particular here-document, but here-documents in general can be sent to any command.

7 Useful Commands

This section will cover some commands that you'll find useful for this assignment and for the course in general. This is not meant to be an exhaustive list. We will cover all the commands that were used to create Figure 3, but you may end up using commands that aren't listed.

- `head` — Prints out the first 10 lines of a file by default. Can be instructed to print out a specific number of lines.
- `ls` — Lists the contents of the current working directory.
- `sort` — Prints the sorted contents of a file. The contents are sorted lexicographically by default, but can be specified to be sorted numerically if desired.
- `uniq` — Filters out repeated lines in a file. Can be used to both filter out repeated lines and count the number of times a line was repeated.
- `wc` — Prints out the word, line, and character count of a file.

We have provided a brief description for each command, but should you find yourself wanting more information, refer to the command's man page. You can do so using `man <command>`. A man page (short for manual page) is software documentation for tools and programs found on UNIX systems. To view a man page:

```
$ man <function, program, tool>
```

These manual pages are typically divided into sections, depending on their respective purposes. General commands are found in section 1, system calls in section 2, and library functions, such as the `printf()` function used in this assignment, are found in section 3. So, to view the man page for `printf()`:

```
$ man 3 printf
```

You will want to get into the habit of checking man pages. They are written by programmers for programmers and will be invaluable when you get to writing C code.

8 Deliverables

You will need to turn in the following source code and header files:

1. `plot.sh`: This bash script should produce the Monte Carlo method plots used in your report. This script should produce plots similar to Figures 2, 3. You can create more plots than these if you choose to.
2. `monte-carlo.c`: This file is provided and contains the implementation of the Monte Carlo program.
3. `Makefile`: This file is provided and directs the compilation process of the Monte Carlo program.

You may have other source and header files, but *do not make things over complicated*. **Any additional source code and header files that you may use must not use global variables.** You will also need to turn in the following:

1. `README.md`: This must use proper *Markdown* syntax. It must describe how to use your script and `Makefile`. should also list and explain any command-line options that your program accepts.
2. `DESIGN.pdf`: This document *must* be a proper PDF. This design document must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. **This does not mean copying your entire program in verbatim.** You should instead describe how your program works with supporting pseudocode.
3. `WRITEUP.pdf`: This document *must* be a proper PDF. This writeup must include the plots that you produced using your bash script, as well as discussion on which UNIX commands you used to produce each plot and why you chose to use them. **Use should use \LaTeX to produce this document.**

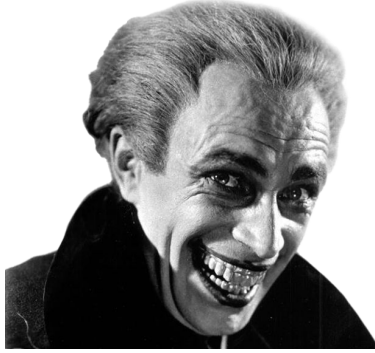
9 Submission

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. “I forgot to push” and “I forgot to submit my commit ID” are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

10 Supplemental Readings

- `$ man bash`
- `$ man cd`
- `$ man head`
- `$ man ls`
- `$ man mkdir`
- `$ man pushd`
- `$ man pwd`
- `$ man sort`
- `$ man tail`
- `$ man uniq`
- `$ man wc`



Honni soit l'homme qui rit.