CMPE-380: Applied Programming

Laboratory Exercise 14

Shell Programming & Simulation

Table of Contents

Pre-Lab 3

Kernel, Shell & Terminal 3

Shells in Linux OS 4

Why Do We Need a Shell? 4

Bash Scripts 5

Interactive Exercises 10

Basic commands 10

Functions 12

Advanced Examples 14

Assignment – 60 pts 15

Objective 15

Assignment Files 15

Initial steps 16

Makefile 21

Analysis 22

Grading Criteria 22

Laboratory Grading Sheet 23

Shell Programming

Pre-Lab

In this lab session, we will review Shell Programming. Shell programming allows you to simply automate tasks which are done repeatedly or to extend operating system commands. Most operating systems now provide access to system commands via graphical shells (GUI) (Windows, Mac, etc.) but all still support command line shells (CLI) (PowerShell, CMD, BASH, etc.). When you drag and drop a file you are interacting with the GUI shell. When you issue copy or cp commands you are interacting with the CLI shell.

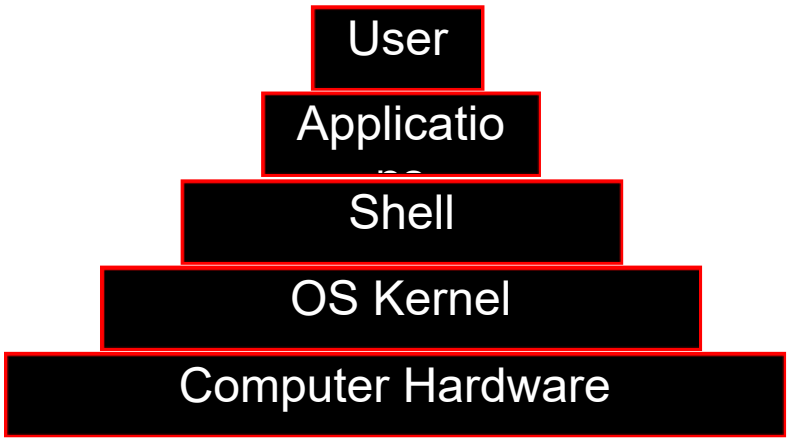
Kernel, Shell & Terminal

The **Kernel** is the core of the operating system which is used for: device management, file & I/O management and process management. The kernel is the guts of the operating system.

The **Shell** enables a user to interact with the **Kernel**. Before user-friendly operating systems, the only way to access the OS was the **Shell**. Now, computers start up in desktop mode, which enables users to access the **Kernel** through drag and drop utilities/applications, however you can still access the **Shell** using a **Terminal**!

The **Terminal** is a special program in the Linux OS or Windows Command Prompt (CMD) which is used to type human readable commands such as “ls”, “cd” and pass them to the shell.

The terminal is an application to display the output from the shell



Shells in Linux OS

- **BASH (Bourne Again SHell)**– The default shell in Linux systems and in Mac OS. It can also be installed on Windows OS.
- **CSH (C SHell)**– A special Shell which has a syntax that is similar to the C programming language.
- Others like Ksh, Zsh etc.

Please note that each shell has a different command set to interact with the OS. For more details about shells please read the article at: https://opensource.com/business/16/3/top-linux-shells

Why Do We Need a Shell?

- **Avoid repetitive work**: If we want to try the same set of commands to run/test our programs, or backup our data etc. we should avoid typing the same commands each time!
- **Add new functionality to the existing shell**: We may need to implement a new shell command!

Bash Scripts

A BASH shell script is composed of specific shell keywords, commands, or functions to achieve a certain task. In a shell script you can declare variables, use if statements, while or for loops, case conditions and call functions.

Shell scripts are traditionally named with a "**.sh**" extension.

Bash shell scripts must start with **"#!/bin/bash**".

Shell scripts are created with text editors, saved and then made runnable by setting the execute bit: e.g. chmod +x <file.sh>

Note: If you create a shell script on your local PC, you may get an "unexpected token" error when you try to run the script in a Linux environment. Windows terminates each line with a CR/LF while Linux only uses a CR. If you receive a token error use the *dos2unix <file1>* command to convert the (DOS or Mac) format of your file (file1) to UNIX.

Note: bash is VERY white space sensitive!

The following is a simplified list of commands:

Command	**Action**	**Example**
#	Comment	# I am a comment
X=	Create a variable, NO SPACE between the variable and the string or number	X=dog
$X	Uses a variable	echo $X
$1 $2 $3 etc	Positional passed parameters	aka argv[] or function parameters
echo	Prints something with a new line	echo “Hello”
printf	Like a C printf command	Printf “number %d\n” 5
read	Reads from stdin	read name
` cmd `	Executes the Linux commend in the BACKQUOTE (next to the 1 key)	directory=`pwd`
+ - * / % ** += -= *= /= %=	Mathematical operations	Y=$((Y+1))
let	Use complex mathematical operation	let Y+=1

if [condition] ; then
<command>
else
<command>
fi
	If statement	if [$age -lt 10] ; then
		echo "Not old enough"
		fi

< <= == => > -lt -gt -le -ge -eq -ne	string comparison numeric comparison	
&& || !	Normal Boolean operations	
while [condition]	While loop	

do Command done		
until [condition] do Command done	DO until loop	
case value in a) command ;; b) command ;; *) command ;; esac	Case statement, note the double semicolons. *) is the none of the above case	
for item in list do command done	Iterates through a list	A list variable is just a normal space delimited string variable.
break continue	Normal loop modifiers	

Script Functions:

The bash scripting language supports functions like C. Functions can take parameters and return values. Passed parameters are positional using an integer (aka $1, $2). Functions return a value using the echo command and specific calling formats. Functions must be defined before use. The syntax of a function is:

```
function someFun {
  echo "In the function, the passed parameters are:" $1 $2
}
```

The calling syntax to capture the output of a function is: **result="$(returnFun)".** Anything in the echo command will be captured

```
function returnFun {
  retnum=999
  echo $retnum
}
```

. In this case result will have the value 999.

For more information search the web or see:

https://linuxconfig.org/bash-scripting-tutorial-for-beginners#h1-bash-shell-scripting-definition

Please change to the “prelab” directory to test the exercises in this lab.

Create a “**hi.sh**” bash script. Your script:

1) Prompt for a name using the echo command
2) Print “Hello <name>” - the name entered - using echo

3) Prompt for a name AND ENTER DATA AT THE END OF THE PROMPT using the printf command
4) Print “Hello <name>” using printf

Pre-lab

Show your **hi.sh** script to your lab TA and be prepared to answer simple bash shell questions.

Interactive Exercises

Basic commands

In this section we will exercise basic bash shell commands. Change to the "exercise" directory and use the provided **lab_a.sh** code as a template for your solution.

1) Create a variable containing the phrase "dog cat"
2) Print the variable out at the end of the string: "The variable is"

3) Write code to read a number from the command line
4) Print the variable at the end of the string :"The number is"

5) Use the number in an "if" statement, printing out a message **containing** the number stating if it is greater than 3 or less and or equal to 3.

6) Write code to add 2 to the number using simple addition
7) Write code to clearly identify what operation occurred and the new value.

8) Write code to add 1 to the number using "+="
9) Write code to clearly identify what operation occurred and the new value.

10) Write a while loop to count from 1 to 3 and print out the value of the looping variable.

11) Write code to cycle through the list: "one two three" and print out the value of the list item being processed.

12) Write code to prompt for the string: "Enter dog (or not)"
13) Use an if statement to determine if the string entered was the phrase "dog" or not, clearly identify the result of the if statement.

14)Write code to prompt for: “Enter a number [1-3]"

15)Write code using a case statement and check for the value “1”, “2”, “3” and other.

16)Clearly print out the case executed.

17)Test with the values: 1, 2, 3, 999

18)Write code to get and then clearly print the current directory

Show and demonstrate your code to your TA.

Functions

In this section we will write a shell scrip that demonstrates passing parameters to a function and returning values from a function

1) Write a bash shell script called **lab_b.sh** with the proper header
2) Create a function called “parmFun” that passes two parameters and returns nothing.
3) The function should clearly print the value of the two passed parameters.
   e.g: "In the function, the passed parameters are:" <variable values>

4) Create a function that returns the number 999 and takes no passed parameters

5) Create a function called **globVar** that creates the “local” variables **var1** **var2** and set them to **“dog cat”** respectively
6) The function should clearly print out the values of var1 & var2 BEFORE setting them in the function and after setting them in the function.
   e.g “Entered globVar" <variable values>
   "Leaving globVar" <variable values>

7) Write the body of the script that calls the “parmFun” function passing the values using the variables **var1** and **var2** initialized to “**one two**” respectively.
8) Clearly identify you are calling the function parmFun utilizing the variable var1 and var2. e.g “Calling a function with 2 parameters:” <variable values>

9) Call the globVar function to show that variables are global. Clearly identify the values of **var1** and **var2** before and after calling the function.
   e.g: “before calling globVar" <variable values>
   "after calling globVar" <variable values>

10) Add to the body of a script the call to the function that returns a number and clearly show that the number was returned.

Show and demonstrate your code to your TA.

Advanced Examples

Examine and run the provided script **lab_c.sh** and **lab_d.sh.** Document your answers in **exercise.txt**. **Be sure to save backup copies of your exercise.txt file as you work.**

1) What does the **lab_c.sh** script do?
2) Are there any new programming “tricks”, “commands” or “features” used that were not covered in this lab?

3) What does the **lab_d.sh** script do?
4) Are there any new programming “tricks”, “commands” or “features” used that were not covered in this lab?

Show your exercise.txt file to your TA.

Assignment – 60 pts

Objective

The purpose of this assignment is to implement the Runge-Kutta RK3 and RK4 simulators based on the simulation class notes and to implement a simple spring mass simulation with dampening. You should rely on the simulation class notes to help with the implementation of the RK3 and RK4 simulators. An implementation of the Eulers and Heun's - RK2 simulation method is included in the provided simulator code. An implementation of the fluid tank problem reviewed in the class notes is also included to aid in debugging your RK3 and RK4 implementation.

Assignment Files

hw14.c	A complete testing interface, no changes are required in this file. See program help for information. Simulation output is in the form of 3 columns of data: time stamp, parameter 1, parameter 2
myplot2	A binary file to plot the results of your simulation. See program help for usage.
genErrors	A binary file which will take the results of each spring simulation, with no dampening, and compare it to the expected analytical results cos(8t). The output from this program can be used to plot the simulator error.
ODEsolver.c	Contains the EU & RK2 simulation code and RK3 and RK4 framework functions for you to implement.
simulations.c	Contains the fluid tank simulation and a spring/mass framework for you to implement
ODEsolver.h ClassErrors.h	Provided .h files, no changes needed

Note: If some of the programs provided do not run, change their executable bit: **chmod a+x filename** (this is what makes a file executable in Linux).

Initial steps

Verify you understand the operation of the simulator and plotting code by executing the following. (This does not need to be included in your lab writeup.)

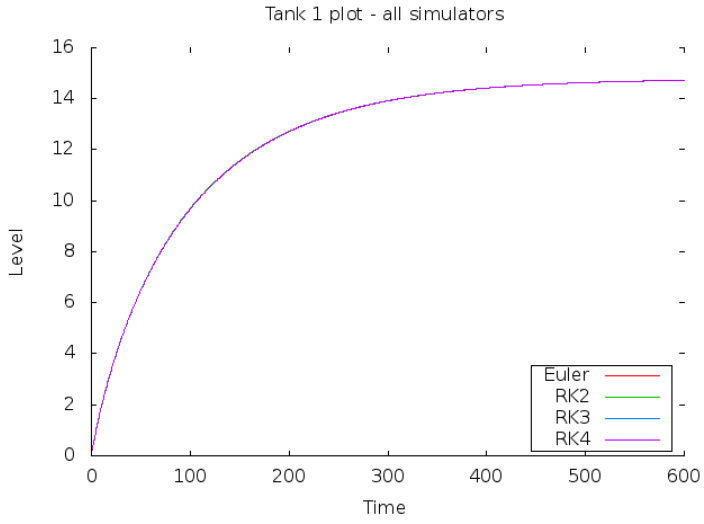
1) Test compile the provided code using:
   gcc -g -std=c99 hw14.c ODEsolvers.c simulations.c -lm -o hw14

   a. Run the code to produce 3 column sample output using:
      ./hw14 -tank -x1 0 -x2 0 -eu -step 5 -ftime 600 > t1.sim
      ./hw14 -tank -x1 0 -x2 0 -eu -step 5 -ftime 600 > t2.sim
      ** yes twice each because the plot routine assumes four simulators **
      ./hw14 -tank -x1 0 -x2 0 -rk2 -step 5 -ftime 600 > t3.sim
      ./hw14 -tank -x1 0 -x2 0 -rk2 -step 5 -ftime 600 > t4.sim

2) Build the 12 column plot data using the following:
   paste t1.sim t2.sim t3.sim t4.sim > tank.sim

3) Generate a plot using: **./myplot2 -tank tank.sim tank.png**

4) View the tank.png plot using the Linux display command: **display tank.png**



Implement RK3 & RK4

The implementation of a RK3 and RK4 simulation engine is an extrapolation of the RK2 implementation, following the same segment approach but with additional stages and different weightings as identified in the class notes (and in the code comments).

The RK2 algorithm has an initial evaluation stage:

```
/* k1 = f(tk, xk)  */
f(sim, t0, x0, k1);
```

followed by a single RK correction stage

```
for ( i = 0; i < sim->neq; i++ ) {
   /* Build k2 function parameters   xk+hk1*/
   xtilde[i] = x0[i] + sim->h*k1[i];  }
/* k2 = f(tk+ h, xk+hk1) */
f(sim, t0+sim->h, xtilde, k2);
```

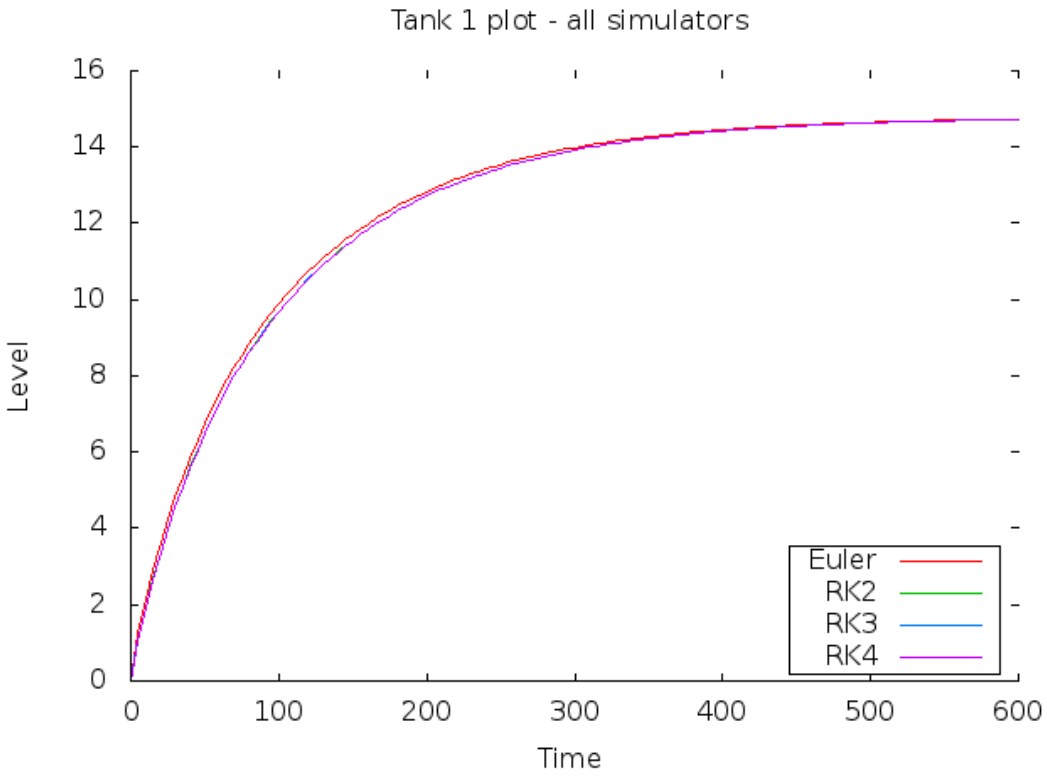
Followed by a final evaluation stage utilizing the RK weightings

```
/* update dx xk+1 = xk+h(½k1+½k2)  */
for ( i = 0; i < sim->neq; i++ ) {
   x0[i] += sim->h*(k1[i] + k2[i])/2.0;     }
```

The RK3 and RK4 algorithms add additional correction stages with their own specific weighting values and a final evaluation stage with specific weighting values.

Test your RK3 and RK4 simulation engines using the provided "tank" simulation comparing to the provided "RK2" simulator. This time, you will generate one .sim file for each simulator. RK3 and RK4 results will be very close to the RK2 results.

The following is a graph of the expected results:



Spring Simulation

You will be implementing the classic spring and block simulation with optional dampening loss. As a start, review the provided tank simulation code in **simulations.c** to develop a feel for how to setup a state variable vector and how the simulator will act on the values.

Assume the following for our problem:

k = 128 N/m	spring constant
d = 0.2 N/m	dampening constant - optional
M = 2 kg	mass
l – 1.0 m	initial displacement

1) Hook's law states a stretched spring provides a force of **f = -kl** Where k is a spring constant and l is the displacement from rest. The force is in the opposite direction of displacement.
2) Newtons second law of motion is: **f=ma** so for any given force the mass will affect the acceleration and therefore the period of the oscillations.

   Given: f = ma , f = -ky and a=dv/dt

   Results in -ky = ma -> a = -k/m*y -> **dv/dt = -k/m*y**
3) The dampening force will dissipate energy from the system eventually stopping the spring. Without a dampening force, the spring/mass system will oscillate forever. The dampening force will be passed on the command line and stored in the variable: **sim->damp**. The dampening force is proportional to the velocity.

   dv/t = -d*v

FYI: A spring/mass system like this without dampening has an analytic solution of:

x(t) = cos(8t)

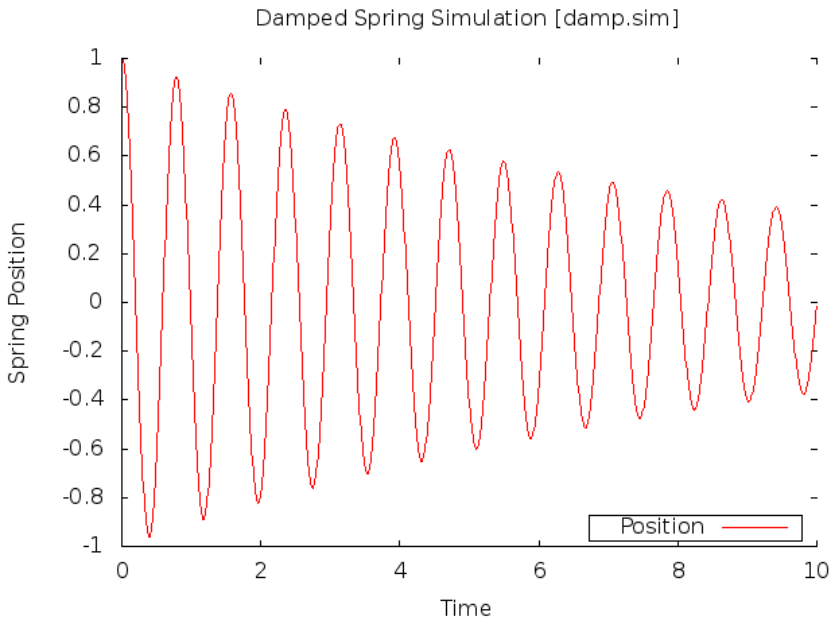
The first step in developing a simulation system is to determine which state variables will be required. The selection of state variables is based on the problem space. In this problem space we need to know the spring displacement because it is required by Hook's law for springs. We also need to know the velocity because the dampening force is proportional to velocity.

The simulator utilizes a vector (array) of state variables so we will make the following assumption:

x[0] - position, x[1] - velocity

In summary, the differential equation for position is just the integration of velocity and the differential equation for velocity utilizes the modified newton equation and the optional dampening value.

Expected damped spring solution:



Makefile

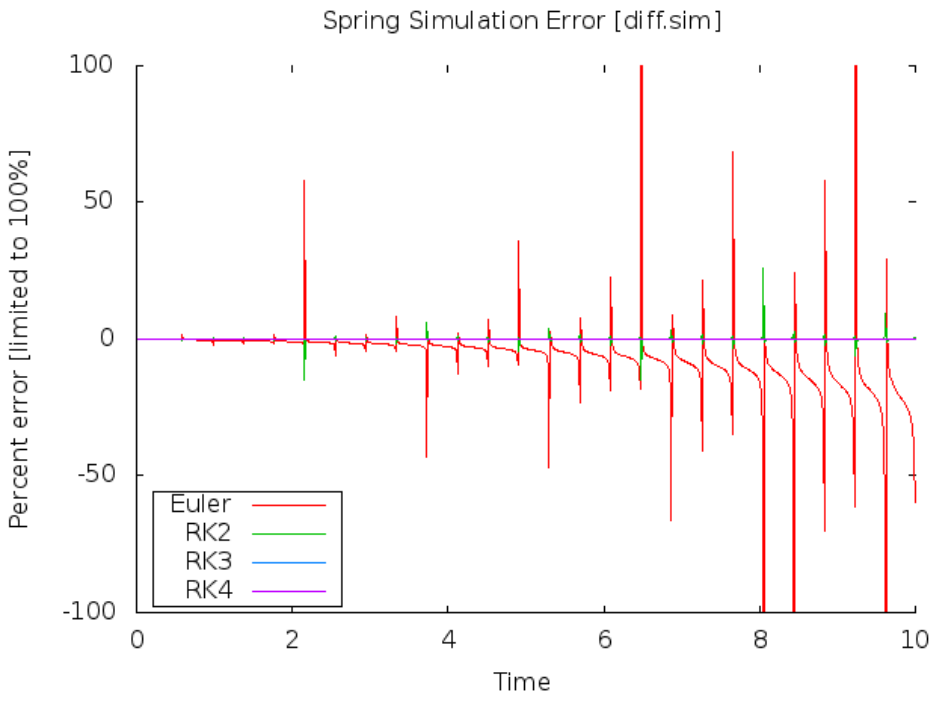
You must provide a quality Makefile with the following targets: all, tank, spring, damp, tests, mem, help and clean. Quality Makefiles utilize .SILENT, .PHONY, use make variables and only compile and/or link the minimum number of files. A Quality makefile is robust to usage.

E.g. make clean followed by make mem will automatically rebuild the binary.

Make Targets;

all	should make hw14
tank	should run the tank simulation with all four simulators and provide a single output plot file called **tank.png** of the position parameters. Use the "paste" Linux command to combine the output of each simulation into a single file **tank.sim** file to plot. Use **myplot2 -tank tank.sim tank.png** to generate the plot. Each simulation should use the following parameters: **-step 5 -ftime 600 -x1 0 -x2 0**
damp	should run the spring simulation using RK2 and a dampening value of **0.2** to produce a **damp.png** file. (use myplot2 with -damp option.) Use the following parameters: **-step .01 -ftime 10 -x1 1 -x2 0 -damp 0.2**
spring	should run the spring simulation (without dampening) with all four simulators. Use the paste command to combine the output of each simulation into a single **spring.sim** file. Pass the spring.sim file to the **genErrors** program which will produce a new data difference file which can be plotted using myPlot2 -spring. The resulting plot should be called **diff.png** Each simulation should use the following parameters: **-step .01 -ftime 10 -x1 1 -x2 0**
tests	should run the **tank, damp** AND **spring** tests
mem	should run the tank simulation with RK3 and RK4 simulators and store the combined valgrind results in **mem.txt**
help, clean	should do the normal things

Expected spring results:



Analysis

Your analysis should address the spring simulation error and how the error relates to time and the simulator used.

Grading Criteria

1. (15 points) Correct implementation of the RK3 code.
2. (15 points) Correct implementation of the RK4 code.
3. (15 points) Correct implementation of the spring code.
4. (5 points) No memory leaks or access errors
5. (8 points) Correct make file
6. (2 points) Analysis

Student Name: _____

Laboratory Grading Sheet

Lab 14 - Shell Programming & Simulation

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab: hi.sh code	**8**		
Pre-Lab: quiz	**2**		
Exercise: Basic commands lab_a.sh	**15**		
Exercise Functions lab_b.sh	**10**		
Exercise: Advanced lab_c/d.sh	**5**		
Total	**40**		

You must turn this signed sheet in at the end of lab to receive credit!