**C Programming Mini-Challenges**

For the following Mini-Challenges, you may use the Linux box set up for you.  It is equipped with gcc and valgrind for your benefit.  The IP address is: 129.244.245.11 (hostname Diophantus.utulsa.edu). Its open SSH port is 53997. Your individual credentials have been provided to you in the Harvey grade center under Diophantus creds in the "Feedback to Learner" section. The text therein should contain your name, your account ID and your password. (Pay no mind if you get a linebreak due to a hyphen in the password, just ignore the linebreak.)

To ssh into that box, you would type something like: ssh -p53997 <ACCTID>@129.244.245.11

To use scp for copying files into it, that would go something like: scp -P 53997 myfile @129.244.245.11:~/

And now, on to the Mini-Challenges:

1. Print “Hello, <NAME>” where NAME is input from the keyboard.
2. Implement Archimedes algorithm to estimate pi for inscribed/circumscribed polygons with n sides, up to 100, doubling n at each step, and time it.
3. Implement matrix – vector multiplication.  Read in the following text file (mv.txt) which contains the matrix and vector to be multiplied.  Print your answer to the screen and time the computation.  The format of mv.txt is:  line 1 contains numrows, numcols.  The next numrows contains the rows of the matrix, each with numcols integer values.  The next line contains the length of the vector. The next line contains the vector of that length.
4. Compare the speed of \*,/,sqrt, sin operations/functions.
5. Use the attached code snippets as a basis for comparing the performance of row-major vs. column major computations.  One snippet uses a static allocation for the array, the other allocates the array dynamically.  Do a little experimentation with each approach. Vary the size of the square array from 128 X 128 on up, doubling it in size each time.  Chart your results.  Is there a difference in performance or behavior between static and dynamic?  Between row-major and column-major?  In terms of the latter, valgrind has been installed on your Linux VM, and its cachegrind tool facility may help provide some insights.  Do some OSINT research to learn about valgrind....

**Submit:** Your source code for each mini-challenge, zipped up in a folder named Ex0.  As a naming standard, name each source code file <LAST NAME>\_ Ex0\_<challenge\_number>.c. If you have more than one source code file for a given challenge, append a,b,c, etc. to that last part. For example, a file for the 5th challenge might be Hawrylak\_Ex0\_5a.c. Also, submit a PDF with your output (screenshots will suffice), along with responses to the following:

Name: David Fantin

ID: 1525813

Class: CS 4373

Date: 1/31/2022

C Programming Mini-Challenges

1. For the Hello Name challenge: (20 points)

a. Did you try passing your name as an argument from the command line or did you use scanf?  Why?

I used scanf() simply because I looked up how to take in command line inputs and it was the most prevalent result. Also it seemed easier.

* **What did you find with scanf() when you had a space in your name (e.g., between first and last name)?**

At first, when I was using “%s” to read in the characters, it would only print the first word (everything before the space).

* **How did you handle this so that you can get your first and last name into the system?**

I switched to using “%[^\n]” to read the characters instead of “%s”, which solved the issue. “%[^\n]” tells the system to read every character until there is a new line, while “%s” tells the system to read every character until there is a space.

b. How did you manage or allocate the strings? (Static or dynamic)

I allocated the strings statically. I set the character array to be size 780, since the longest name in known history was 747 characters long (or 773 with spaces).

* **Did you check if the dynamic memory allocation succeeded? If so how did you do this?**

I did not, because I did not use dynamic memory allocation.

Output:



2. For Archimedes algorithm (20 points)

a. How did you time your program?

I included the time.h package at the beginning of my code. Then before my for loop I found the current the clock time (start) and after the for loop I found the current the clock time (end). Subtracting the final time from the start time then dividing it all by how many clocks cycles occur per second (CLOCKS\_PER\_SEC), results in how many seconds it took to run the algorithm.

b. Were there any issues with precision and/or convergence that you noticed?

I also tested my results in mathematica, and at around the 6th decimal point, mathematica is more accurate than my program. I believe it has to do with the sine function, and the conversion ratio I had to use to get 180 degrees converted to radians (since I could not use Pi).

* **Did you use integer or floating point types for your calculations?**

I used integers for the number of sides (n), but floats for the actual approximations because I wanted the decimal points.

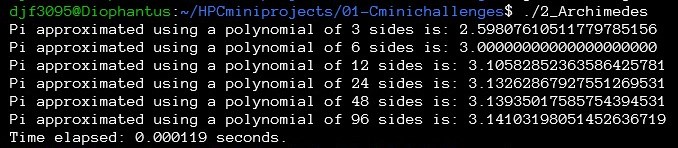
* **How many decimal places did you print PI to? How did you do this?**

I printed it to 20 decimal places by using “%.20lf” instead of “%lf” in the printf() statement.

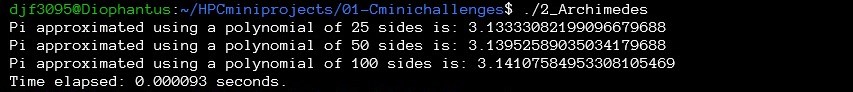
* **Did you try this with the float and double types? If so, did you see a difference?**

I did, and there was a difference. Using the double type, the result for n = 100 was slightly more accurate than the float type with the first difference occurring on the 7th decimal place. For speed, on average they seemed about the same (floats may have been very slightly faster on average but I could be imagining it).

Output 1: Starting from n = 3:



Output 2: Starting from n = 25:

(I was not sure if you wanted this, but was the only way to show n = 100 and still double n)

3. For Matrix-vector multiplication (20 points)

a. How did you allocate and access your matrix?

I first multiplied the number of rows by the number of columns to get the total number of elements, and created an array of that size. I then had a for loop to add all the elements of the matrix to the array. In order to access each row, I had a for loop that looped over 10 integers, then when it reached the 10, and there were still elements left, I added 10 to to the stopping condition so that it would continue.

b. Were there any challenges in reading in the file?

It took me a while to figure it out, but once I found the fscanf() function, and understood how it worked, it was easy to integrate it into my program. There is probably a better way of doing it than I did it though.

c. Was there anything special about the actual computation?

Not specifically beyond the fact that it is matrix multiplication. I had to loop quite a few times though, in order to iterate through all the rows.

d. What was your strategy for timing?

I timed it the same way as I timed problem 2: I included the time.h package at the beginning of my code. Then before accessing the file I found the current the clock time (start), and after closing the file I found the current the clock time (end). Subtracting the final time from the start time then dividing it all by how many clocks cycles occur per second (CLOCKS\_PER\_SEC), results in how many seconds it took to run the algorithm.

* **Did you check if the dynamic memory allocation succeeded? If so how did you do this?**

Yes, if I understand correctly, the dynamic allocation did succeed, because I tried my code with 10 different input matrices / vectors of different sizes and it worked perfectly every-time.

* **Did you perform any error checking on the input from the file (e.g., correct number of rows and columns)?**

Yes. I made an if statement that checks if the number of columns = the number of elements in the vector, and if they do not equal, then the matrix and the vector cannot be multiplied, so the program terminates.

Output:



4. For measuring the speed of arithmetic computations (20 points)

a. What was your timing strategy?

The timing strategy I used for the previous two questions seemed to work, so I used a similar idea: I had four for loops (one for each function) before each one I found the clock time, and after each one I found the clock time. Then at the end of the for loops I found the total time in seconds by subtracting each final time from its respective start time then dividing it all by how many clocks cycles occur per second (CLOCKS\_PER\_SEC).

b. Are all arithmetic operations created equal?

No the square root function took a very long time, the sine function took a very short time, and multiplication and division were very close, but multiplication seemed slightly faster (although that could have been due to how I set them to calculate).

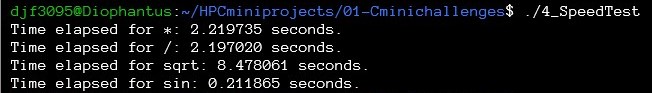
* **What do you think causes the differences in timing for the operations?**

I believe sine is the quickest because the function’s outputs are always between -1 and 1, so no matter how high the inputs go, it is fairly easy to calculate. The square root function is the slowest because it is the most complicated to accomplish. Multiplication and division are similar operations, which is why I they are so close, but multiplication is slightly faster since for division you need to multiply by the inverse, which is an extra step. For the square root function I was curious if perfect squares would make it go faster, but it did not. I did notice that when I kept the input to the functions static for every iteration they square root functions were a lot faster than when I changed the input for each loop, however there was barely a change for the other three operations.

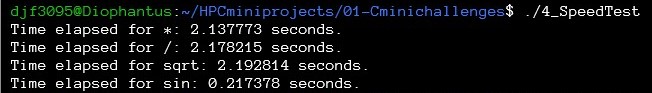
* **What did you do to investigate this further?**

Like I said before, I changed the inputs from static to dynamic (counting up by 1 after each iteration) to see if that had an effect (it only majorly affected the square root, but somehow it also made multiplication slightly faster than division). Besides that, I messed with the inputs for each of the functions just to see if I could change the results substantially, however I could not change them by much.

Output 1: Dynamic speed test: (Each function’s input varied with each iteration)



Output 1: Static speed test: (Each function’s input was the same number for each iteration)



5. For the row-major/column-major exercise (20 points)

a. What did you observe about differences in program behavior in static vs dynamic allocation of arrays, and how do you explain it?

The dynamic allocation was always faster for the ones I tested. The interesting thing though was that for all the tests, no matter what size matrix I used, the static always finished in around the same time as all the other sizes I tried. The dynamic one however, gets slower as the matrix size increased, which is expected.

b. What did you observe about differences in program behavior in row-major vs. column major computations and how do you explain it?

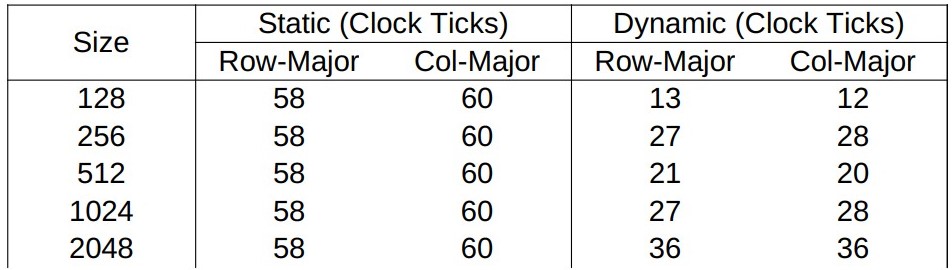
I did not see much difference between the two at first

* **How did the timing of the row-major compare with that of the column-major? Why do you think this is?**

The column major was very slightly slower on the static, but on the dynamic it was about the same. I believe they are close because they are both doing the same thing, just traversing the matrix in different ways. Although I think the column major may be sightly slower because of how the matrix is read in. The rows are read in first so it might take more flip flopping around with pointers to achieve the result, making it a bit slower.

* **How do different architectures and OSes impact the results of this part?**

A slower OS or older architecture would result in a slower time. In addition, some architectures might have different conventions to read matrices and be more or less efficient in doing so.

Output: Speed Chart:

Output: Static:

Output: Dynamic: