Matthew Lacaire
UID: 205288439
CEE/MAE M20
October 6, 2020

Homework 1

1. String Manipulation

a. Introduction

Problem 1 involves a script that is to ask the user for his/her name and student ID number. Then, manipulating these strings, the program capitalizes every letter of the name and reverses the student ID. Lastly, the program outputs this information and tells the student, in full caps, what their student ID is backwards.

b. Model and Methods

The script calls the input function twice, and, upon the user's input in the command window, immediately assigns these inputs to the variables "name" and "number". See below:

```
name = input('Enter your name\n','s');
number = input('Enter your UID\n','s')
```

Then, in a succinct, one-line command involving the fprintf function and arguments upper(name) and fliplr(number), I output the .

```
fprintf('Hello %s, your UID backwards is %s', upper(name),
fliplr(number));
```

The act of capitalizing the name and reversing the number are done by the fliplr() and upper() commands. For conciseness and cleaner code, I have these functions process the strings within the fprinrt arguments. I could have just as easily written something along the lines of

`uppercasename = upper(name)` and `backwardsnumber = fliplr(number)`

And taken uppercasename and backwardsnumber as arguments. In my technique however, the processing of strings and subsequent printing all happens on the last line of code.

c. Calculations and Results

The program works, and, after testing with various names and strings (IDs) of various length and permutation, holds up and does its assigned task. Below is the entire command window dialog when working with my own name and student ID:

```
        Enter your name
        Matt Lacaire
        Enter your UID
```

```
205288439
Hello MATT LACAIRE, your UID backwards is 934882502
```

d. Discussion

This program shows the power of MATLAB's inherent functions, and is entirely dependent upon upper() and fliplr(). Also important to note is that if the "input" command is given characters or non-numbers, it attempts to execute them as a sort of keyword or key term. To override this (and guarantee all input is treated just as a string), I had to add the 's' portion:

```
name = input('Enter your name\n','s');
```

Besides this peculiarity, the problem works exactly as expected, and, quite simply, gets inputs, processes them, and outputs the results according to a set of rules.

2. Sterling's Approximation

a. Introduction

Problem 2 is a question that prompts the user for a number (n), and, using two methods (one exact, one approximate), calculates factorial values. Also, a percent error calculation is performed and displayed along with exact/approx. factorial values.

b. Model and Methods

The program is split into three sections: Inputs, Algorithms, and Display (output). This is done to organize the code and make it more readable.

In the inputs section, the script simply request a number from the user by using the input() function and assigning the result to variable *n*:

```
n = input('Enter a value of n:');
```

In the following *Algorithms* section, computations are performed and assigned to new variables. The *factorial*() function provides an exact factorial value, and this is immediately assigned to the new variable a. Meanwhile, the Sterling Approximation is represented in MATLAB format by the following line:

```
b = sqrt(2 * pi * n) * (n / exp(1) )^n;
```

In a similar fashion, the following line finds the percent error by simply rewriting the equation given on HW in a MATLAB-conducive format:

```
c = ((a - b) / a) * 100;
```

Lastly, the exact, approx, and percent error values are printed with *zero* decimal points. I do this by inserting the highlighted section. See below example:

```
fprintf('n! approx) %.0f\n', b);
```

I did this entirely for cleanliness, considering that factorials (at least of integers) should never be fractional; but rather should be integers themselves.

c. Calculations and Results

The program appears to work, though I cannot say that I have tested a massive array of numbers to test its limits. The following is an example code I just ran:

```
Enter a value of n:15
```

```
n! exact) 1307674368000
n! approx) 1300430722199
error: 0.553933%
```

After experimenting with a fair deal of numbers, I have not run into any errors or any particular issues worth debugging.

d. Discussion

The code works in three particular steps, much like no. 1: it takes an input, alters/processes it, and then returns the result.

Though the MATLAB-native factorial() command promises to calculate exact factorials, I doubt its limits. For example, entering a massive number (ex. 2444) results in MATLAB returning

```
n! exact) Inf
n! approx) Inf
```

This proves that MATLAB and computer processing in general has an upper limit to how precisely, or frankly, how successfully, it can calculate difficult tasks.

Mathematically, this program is interesting. It shows MATLAB's ability to make analyzing numbers and trends fairly straightforward. For example, after testing many numbers and logging error percents, the script has allowed me to make the conclusion that Sterling's approximation is *usually very accurate within 1-2%* and *always seems to underestimate slightly*.

3. Law of Sines and Cosines

a. Introduction

Question 1.3 involves calculations with the law of sines and the law of cosines. Given a, b, and c values (side lengths), the program solves the triangle, finding all angles. A challenge is presented in that one angle is inaccurate if calculated with the law of sines. See following sections for more information.

b. Model and Methods

The program is split into four subsections: Variable Assignment, Law of Cosines Calculation, Law of Sines Calculation, and, lastly, Output.

Variable Assignment is simply setting variables a, b, and c equal to the provided lengths.

In the *Law of Cosines Calculation*, I rearrange the law of cosines to solve for alpha. I wrote this out on paper, solving for alpha, and then input the equation in MATLAB-friendly language:

```
alpha = acos(((c^2) + (b^2) - (a^2))/ (2 * b * c));
```

Note that acos() is an arccos-finding function that expresses itself in radians.

In *Law of Sines Calculation*, I solve for the β and γ angles. I do the same thing as I did for the previous subsection- rearranging the equation (this time, law of sines) to solve for the desired angle.

```
beta = rad2deg(asin(sin(alpha)* b / a));%
gamma = rad2deg(asin(sin(alpha)* c / a));
```

Note that the entire equation is nested inside of the rad2deg function, which converts from radians to degrees. This is necessary according to HW specifications.

I write the line:

```
correctgamma = 180 - gamma;
```

because gamma, visually, is obviously obtuse. Remember identity sin(x) = sin(180-x) in degrees.

Also note that I *did not* use the rad2deg function in the Law of Cosines part. This is because the alpha variable is referenced in the beta/gamma calculations. It would make no sense to have a degree-valued alpha used to calculate something in radians, then finally convert to degrees. That's why I elect to convert alpha to degrees later in the program.

In *Output* I solve this by having rad2deg(alpha) as an argument of fprintf. The rest of the function is just printing values and explaining for discrepancies.

c. Calculations and Results

The output of the function is verbose, but it calculates correctly, as confirmed by calculator check. The following is the output:

```
According to Law of Cosines, alpha is 22.33 degrees.
According to Law of Sines, beta is 49.46 degrees, and gamma is
108.21 degrees.
Note also that the Law of Sines produces either the required
angle or 180 minus the angle.
The original gamma is calculated as 71.79, which is not
visually accurate (gamma is obtuse).
For confirmation, the correct gamma, beta, and alpha sum to
180, as 22.33, 49.46, and 108.21 sum to 180.
```

By using basic arithmetic and/or actually seeing the triangle, we can confirm the validity of the last statement.

d. Discussion

This script involved pretty serious equation manipulation at first, and made me privy to a technique- solve equations on paper for a given value, and then, translating them into MATLAB-friendly format, use them to calculate the value.

Also of note is the fact that gamma, as originally calculated, was wrong. This is because MATLAB takes the standard range of asin (arcsin) to be [-90, 90]. I had to correct for this by writing and displaying the "correctgamma" variable.

4. Oblate Spheroid

a. Introduction

This is a script that solicits the user for two inputs (r1, r2- equatorial/polar radii), performs exact and approximate surface area calculations, and then outputs both values.

b. Model and Methods

The script is organized into three sections: inputs, calculations, and display. In inputs, I simply ask questions, and assign answers to r1, r2. Example:

```
r2 = input('What is the polar radius?\n');
```

In the calculations section, I define an intermediate variable "gamma". Gamma is a constant found within the exact area equation, and, to make the code less messy, I simply calculate gamma instead of writing its definition each time in the equation. Variables "approx" and "exact" are fairly standard, and are simply assigned the HW equations formatted in such a way that MATLAB understands them:

```
approx = (4 * pi * ((r1 + r2) / 2)^2);
exact = 2 * pi * (r1^2 + (r2^2/sin(gamma)) * log(cos(gamma) /
(1 - sin(gamma))));
```

The exact SA equation is lengthy, and, for the sake of safety/accuracy, I made extensive use of parentheses to respect order of operations.

Finally, with the fprintf, the results are displayed. See discussion for more info.

Calculations and Results

The following is an example of a user-script interaction. I used random values for r1, r2:

```
What is the equatorial radius?
69
What is the polar radius?
67
Exact value is 5.8676e+04 km^2, and approximate val is
5.8107e+04 km^2>>
```

As you can see, the program clearly asks for radius values, and then displays results with units as well.

c. Discussion

The program works, and after comparing results with the Earth's real surface area, I can see that the math checks out. Due to the large value of the surface area, I elected to display the results using the "%e" argument of the fprintf function, which displays scientific notation. See below snippet.

```
Exact value is %.4e km^2
```

I also elected to insert the ".4" function to display four digits of scientific notation right of the decimal. This has the express goal of making it possible to see the differences between the approximate and exact calculations. If I didn’t want scientific notation, I could have used the “%f” argument.