ENGR131 – Homework 8: Advanced Function Techniques

Requirements:

- Submit a single .zip file containing all .m files by the posted due date.
- Every .m file should include a comment at the beginning with your name and network ID.
- All work should be your own, as explained in the Academic Integrity policy from the syllabus.
- Problems will be graded for completeness (reasonable attempt for each rubric item). They will not be checked for correctness.

Using demo programs: Demo programs are provided as .p files. There are three steps for running a .p file: (1) copy it to a folder that's *different* from where you are writing your own program, (2) select that folder in Matlab as the current folder, and (3) type the script name in the Command Window. *WARNING:* Having a .p file in the same folder as your .m file may cause an error message.

Problem #1: Estimating π

This problem uses function handles. Write a program that allows the user to check the accuracy of different functions that estimate the value of π. Two such functions are provided on the website (madhava.m and monteCarlo.m) and are described below. Your overall program should include those files as well as the following functions in separate .m files. You will create three new functions: two new estimation functions and another function that checks them.

a. Create a function named "newton" that implements the Newton algorithm shown in the equation below. It receives the desired number of iterations *n* and returns an estimate for π. A larger *n* value should produce a better estimate. To help you, the file madhava.m is provided for you and implements the Madhava algorithm which is very similar to the Newton algorithm. The Madhava and Newton algorithms both use a numerical summation as shown in the equations below.

Madhava	Newton
$\pi \approx \sqrt{12}\sum_{k=0}^{n}\frac{\left(-\frac{1}{3}\right)^k}{2k+1}$	$\pi \approx 2\sum_{k=0}^{n}\frac{2^k k!^2}{(2k+1)!}$

Sample output: Below are examples of using both functions with *n* = 5. Note that madhava.m is provided, but you must write newton.m. They will each give a different result because the algorithms are slightly different.

```
>> madhava(5)
ans =
    3.1413
>> newton(5)
ans =
    3.1215
```

Tip (factorials): The Newton algorithm uses the factorial operation "!" which can be computed using the built-in "factorial" function in Matlab. For example, 3! can be computed as shown below.

```
>> f = factorial(3)
f =
     6
```

For fun (not for credit): You can test the accuracy of your "newton" function by using small and large values for *n* and compare the results to the built-in value of "pi" in Matlab.

b. Write a function named "monteCarloVectorized" that uses a vectorized Monte Carlo algorithm. A "Monte Carlo" algorithm uses random samples to solve a problem. A Monte Carlo algorithm to estimate

π involves generating random points with coordinates (x, y) inside a circle that is surrounded by a square. This is described at http://math.fullerton.edu/mathews/n2003/montecarlopimod.html. The value of π is then estimated by counting the number of random points inside the circle and computing the following:

$$\pi \approx 4 \times \frac{number\ inside\ circle}{total\ number}$$

The file monteCarlo.m is provided for you and implements this Monte Carlo algorithm using a loop. It receives the desired number of random points *n* and returns an estimate for π. Test the function yourself and you should notice that you need larger values of *n* than you needed for the algorithms in (a). Your function should implement the same algorithm using a *vectorized* approach (without a loop).

Sample output: Below are examples of using both functions with *n* = 5. Note that each function may return a different result each time it is executed because the random numbers will be different.

```
>> monteCarlo(5)
ans =
    2.4000
>> monteCarloVectorized(5)
ans =
    3.2000
```

Tip (array of random numbers): You can create an array (vector) of random numbers by using rand(1, *n*) which creates a 1-by-*n* array of random values. If you do this for the x- and y-coordinates, you can use the ".^" operator to compute the distance. Then use the "find" function to determine which distances are less than 1.

c. Write a function that can receive a handle to any one of the estimation functions described above and the number of desired iterations or random coordinates. It returns the error of the estimate. Calculate the error as the absolute value of the difference between the estimate and the value of the built-in "pi" in Matlab. Use the "abs" function to compute the absolute value. For example, if the estimate is 3.14, then the function should return abs(3.14 – pi).

Sample output: Below is an example of how this function should work. Note that the functions from (a) should yield the same result every time they are called, but the functions from (b) will not because the random values will be different every time.

```
>> problem1(@madhava, 4)
ans =
   0.001012092073291
>> problem1(@newton, 4)
ans =
   0.043179955177095
>> problem1(@monteCarlo, 100)
ans =
   0.141592653589793
>> problem1(@monteCarlo, 100)
ans =
   0.138407346410207
>> problem1(@monteCarloVectorized, 100)
ans =
   0.298407346410207
```

```
>> problem1(@monteCarloVectorized, 100)
ans =
   0.021592653589793
```

COMMENTS: Interested students can find more about these approximation algorithms, including links to information on Madhava and Newton, at http://en.wikipedia.org/wiki/Approximations_of_%CF%80. The general Monte Carlo approach to solving problems is described at http://en.wikipedia.org/wiki/Monte_Carlo_method.

Rubric

Item	**Points**
Function (a): header	5
Function (a): has a loop	5
Function (a): has computation	5
Function (b): header	5
Function (b): has computation	5
Function (b): uses "find" function	5
Function (c): header	5
Function (c): calls function	5
Total	40

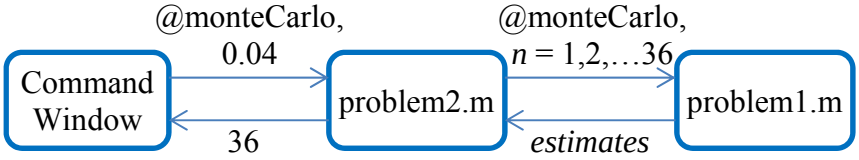
Problem #2: Analyzing algorithms for π

Write a function that finds the number of repetitions that are necessary to estimate π to within a certain accuracy. It receives a handle to one of the estimation functions described in problem #1 and a desired error limit. It returns the minimum value of *n* (number of iterations or random coordinates) for which the error is smaller than the desired error limit. Use the function from problem #1 (c) to do this. For example, suppose that we wanted to know the number of iterations required for the Newton algorithm to achieve an error less than 0.04. We can start with $n = 1$ and use a loop to repeatedly call the function from (c), incrementing *n* by 1 each time that the error is not less than the desired limit. In the example from #1 (c), we know that the error for $n = 4$ is 0.43 which is too large. If the error for $n = 5$ is less than the desired limit of 0.4, then the new function should return 5.

Note that the user will pass the function handle to this new function which will then call the function from #1 (c), and that will call the estimation function. Additionally, the randomized algorithm for the Monte Carlo approach will not produce the same answer every time. Below are some flow diagrams and Command Window examples.

Command Window	Information Flow Diagram

`>> problem2(@newton, 0.04)`
`ans =`
`5`
	

`>> problem2(@monteCarlo, 0.04)`
`ans =`
`36`
`>> problem2(@monteCarlo, 0.04)`
`ans =`
`29`
	

Rubric

Item	Points
Header	5
Has a loop	5
Calls function	5
Total	15

Problem #3: Random numbers for games

This problem uses a variable number of arguments. Write a function named "getRand" that returns random numbers for different games of chance like flipping a coin, craps, and poker. The function returns an array of integers with random values from 1 up to a certain limit. It receives the upper limit of the random values and an optional second argument that specifies how many random values should be returned. If no second argument is given, return only one random integer. Use the following expression discussed earlier this semester to generate the number(s):

```
floor(rand(1, n) * limit) + 1
```

where *n* is the quantity of random numbers and *limit* is the upper limit. You should try using this expression in the Command Window to be sure you understand how it works. You can create a separate file for this function or use it as a subfunction in the primary program described below. If you were to use it as a separate .m file, below are examples of how it could be used in the Command Window.

Five playing cards for poker	`>> pokerCards = getRand(13, 5)`
	`pokerCards =`
	`10 10 4 9 9`

Two dice for craps	`>> crapsDice = getRand(6, 2)`
	`crapsDice =`
	`1 1`

One playing card	`>> singleCard = getRand(13)`
	`singleCard =`
	`7`

One coin (1 = heads, 2 = tails)	`>> coinFlip = getRand(2)`
	`coinFlip =`
	`2`

Rubric

Item	Points
Header	5
Handles different number of arguments	5
Uses "rand" function	5
Total	15

Problem #4: Playing audio files

Write a function that receives words as separate arguments and plays the sequence of .wav audio files that have those words as their filenames. It should receive any number of string arguments. Four .wav files are provided on the website that contain recordings of a person speaking four different words. (You can create your own .wav files if you wish.) Assume that a .wav file exists for every string that is passed to the function. Below is an example of using the function to play the following sequence of audio files: this.wav, is.wav, audio.wav.

```
>> problem4('this', 'is', 'audio')
```

Below is an example of using the function to play the following sequence of audio files: this.wav, audio.wav, is.wav, digital.wav.

```
>> problem4('this', 'audio', 'is', 'digital')
```

Tip: This problem uses a variable number of arguments.

Tip (loading .wav files): Use the "wavread" function which can receive the name of the file with or without '.wav' at the end. For example, the following would load the file 'this.wav' into a variable named "array".

```
array = wavread('this');
```

Tip (playing .wav files): Use the "sound" function to play the data in the array that was loaded using the "wavread" function. For example: `sound(array)`

Rubric

Item	**Points**
Header	5
Has a loop	5
Calls the "sound" function	5
Total	15

Problem #5: Playing audio files *safely*

The previous problem said to assume that a .wav file exists for every string that is passed to the function. Make a new version of the function that prints a warning message when a .wav file is not found, including the string that was the issue. Below is an example in which the file "great.wav" does not exist. The function would still play the first three .wav files.

```
>> problem5('this', 'audio', 'is', 'great')
ERROR: great.wav not found.
```

Tip: Use the "fopen" function to determine if the file can be opened. You will *not* read the file as we did in Chapter 9, but you can check for a valid file ID. If "fopen" was successful, just close the file before using the "wavread" function. Note that you will need to add '.wav' to the filename using one of the techniques described in Chapter 7.

Rubric

Item	**Points**
Has same parts from problem 4	5
Uses "fopen" function	5
Checks file ID to see if file exists	5
Total	15

Summary of point values for all problems

Item	**Points**
Problem #1	40
Problem #2	15
Problem #3	15

Problem #4	15
Problem #5	15
Total	100