Lab Assignment #2

Due 2/7/2020 at 11:59pm on bCourses

The assignment will be graded by an autograder which will check the outputs of your functions. For your functions to be scored properly, it is important that the names of your functions **exactly match** the names specified in the problem statements, and input and output variables to each function are in the **correct order** (i.e. use the given function header exactly). Instructions for submitting your assignment are included at the end of this document.
For some problems, a **skeleton code** is provided to you on bCourses. Follow the instructions, if any, in the skeleton code and fill in your implementations. For the rest of the problems, you will write the functions from scratch.

1 Mathematical language to programming language

`function y = mathOperation(x)`		
Input	Type	Description
`x`	`1x1 double`	The input, a number.
Output	Type	Description
`y`	`1x1 double`	A number which is calculated by the mathematical function in details based on `x`.

Details

From the previous lectures, you see how to translate a math function to program. In this question, you are going to translate a simple mathematical function a programming language (MATLAB). The mathematical function includes input `x` and output `y`. The operational rule is as followed.

$$y = \begin{cases} 4x,\ for\ x \le -2 \\ x^3,\ for -2 < x \le 2 \\ 4x,\ for\ 2 < x \end{cases}$$

2 Persistent variables

`function output = timeAddition(input)`		
Input	Type	Description
`input`	`1x1 double`	An integer.
Output	Type	Description
`output`	`1x1 double`	An integer, which is the sum of `input` and a persistent variable that is defined in the function. See Details.

Details

Recall the definition of a mathematical function that Prof. Sengupta discussed in the lecture. What does a non-mathematical function look like? This problem is a really simple function that illustrates the idea that a MATLAB function is not necessarily a mathematical function. Recall that a mathematical function must always give the same result for the same input value. Our function violates this definition through the use of a persistent variable.

In this function, you will define a persistent variable (see **Tips**) `count` that records the number of times that the function has been called. This persistent variable `count` is initialized with 0 and incremented by 1 each time the function is called. The output of the function is simply the sum of `count` and `input` (after `count` has been incremented).

Tips

Refer to MATLAB documentations about persistent, which gives detailed instructions and examples of how a persistent variable is defined and initialized. Pay particular attention to the initialization of a persistent variable since it is more complicated than that of an ordinary variable.

3 Decimal-Quaternary converter

`function quaternary = decimalToQuaternary(decimal)`		
Input	Type	Description
`decimal`	`1x1 double`	An integer between 0-65535.
Output	Type	Description
`quaternary`	`1x8 double`	Array of 0s, 1s, 2s or 3s that represents a 8-bit quaternary when read left to right.

Details

Quaternary is the base-4 numeral system. It uses the digits 0, 1, 2 and 3 to represent any real number, as opposed to our standard integers. As such, each digit in a quaternary number can be a 0, a 1, a 2 or a 3, with each digit representing a factor of 4 increase from the previous digit as opposed to the factor of ten increase in the decimal system.

In this problem, you will write a function which converts a decimal value to a 8-bit quaternary number (returned as an array of doubles).

Tips

- The decimal number 2 is 00000002 in 8-bit quaternary, while the decimal number 5 would be 00000011 in 8-bit quaternary and the decimal number 4 would be 00000010 in 8-bit quaternary.

- `quaternary` is a 1-by-8 array. For example, output array $[0, 0, 0, 0, 0, 0, 0, 2]$ represents the integer 2.
- Note that the factor of 4 increase starts with 4^0 and increases as you go from right to left.
- To ensure the output `quaternary` always contains 8 elements, you can use leading 0s in the array.
- Since the number of digits in quaterary is limited to be 8, it is possible to build this function by conditional statement. You might find the MATLAB function floor to be helpful.

4 Rotating a 2D vector

`function [u_rotated] = rotateVector(u,angle)`		
Input	Type	Description
`u`	`2x1 double`	A 2D column vector
`angle`	`1x1 double`	The angle, in **radians**, that the vector should be rotated counterclockwise.
Output	Type	Description
`u_rotated`	`2x1 double`	The vector `u` after being rotated.

Details

In vector mathematics, you can define matrices for different vector transformations. One such matrix is the rotation matrix, associated with a rotation by the angle θ counterclockwise about the origin. This matrix is defined as:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

To apply the rotation to any vector u, one must simply multiply that vector by the rotation matrix. For this problem, you should write a function that based on the given 2D vector and rotated angle, produces the corresponding 2D vector after rotation.

Tips

- Note the different input unit for MATLAB functions: `sin()`, `cos()`, `tan()`, `sind()`, `cosd()`, `tand()`

5 Recovering a rotated 2D vector

`function [u_recovered] = recoverVector(u,angle)`		
Input	Type	Description
`u_rotated`	`2x1 double`	A 2D column vector after `angle` counterclockwise rotation
`angle`	`1x1 double`	The angle, in **radians**, that the vector has been rotated counterclockwise.
Output	Type	Description
`u_recovered`	`2x1 double`	The recovered vector(the original vector without rotation).

Details

Problem 4 asks you to rotate a 2D vector with a given angle. Now imagine you have a 2D vector which you know has been rotated counterclockwise by a given angle θ, how can you recover the original one? To achieve this goal, one can utilize the rotation matrix (referring to Problem 4) by simply multiplying the rotated 2D vector by the inverse of the rotation matrix, which recovers the original 2D vector.

Tips

- You might find the MATLAB function `inv()` to be helpful.
- You **should not** call function from Problem 4. Solve Problem 4 and Problem 5 independently.

6 Datatype matters

`function [add,charAdd,strOut,array,charArray] = dataTypeMatters(x,y)`		
Input	Type	Description
`x`	`1x1 double`	An integer between 0-9.
`y`	`1x1 double`	An integer between 0-9.
Output	Type	Description
`add`	`1x1 double`	The sum of the two input integers $(x + y)$.
`charAdd`	`1x1 double`	The sum of the ASCII values of the two input integers.
`array`	`1x2 double`	The row matrix whose entries are the two input integers respectively.
`charArray`	`1x2 char`	The char array that concatenate the two input integers.
`strOut`	`1x1 str`	Summation of the first element of `charArray` and the second element of `charArray` after the second element has been converted to a string.

Details

In lecture, we talked about different data types. In this question you will see why data types matter. Your function will be given two integer inputs (e.g. 6 and 9), and you are asked to:
1. Return their arithmetic sum. In this case, 15
2. Return sum of their ASCII numbers. In this case, 111
3. Return a 1x2 matrix, with the first entry has value of first input, and the second entry has value of second input. In this case, [6, 9]
4. Return the char array that concatenate the inputs as a new integer. In this case, '69'
5. Return a 1x1 string array. In this case, "69"

Tips

- A basic frame of script is provided to you. Follow the rest of the tips and filling the blanks in the script.
- Look up Convert integers to characters and Convert characters to strings
- Outputs `add`, `charAdd` and `strOut` are generated by the same operator
- Outputs `array` and `charArray` are generated by the same operator

- Now think about why same operators can result in different outputs. Your thoughts will not be graded but will be helpful for your future success.

7 My Time Conversion

`function [hours, minutes, message] = myTimeConversion(totalmins)`		
Input	Type	Description
`totalmins`	`1x1 double`	A non-negative integer, which represents the number of total mintues.
Output	Type	Description
`hours`	`1x1 double`	A non-negative integer, which represents the number of hours after time conversion.
`minutes`	`1x1 double`	A non-negative integer, which represents the number of minutes after time conversion.
`message`	`1x char`	A char array which describes the followings: `totalmins` minutes = `hours` hour(s) and `minutes` minutes

Details

- The input argument `totalmins` should be a nonnegative scalar integer representing the total number of minutes in a specified time interval.
- The output arguments `hours` and `minutes` should be nonnegative scalars such that `hours`*60 + `minutes` = `totalmins` and the value of Minutes must be less than 60.
- The output argument `message` must be a char array. Notice that time units should be singular (i.e. minute, hour) if the value is 1, and plural (i.e. minutes, hours) if the value is not equal to 1 (including 0).

Tips

- For `totalmins` = 151, `message` should equal to the char array: '151 minutes = 2 hours and 31 minutes'
- For `totalmins` = 181, `message` should equal to the char array: '181 minutes = 3 hours and 1 minute'
- For `totalmins` = 60, `message` should equal to the char array: '60 minutes = 1 hour and 0 minutes'
- For `totalmins` = 1, `message` should equal to the char array: '1 minute = 0 hours and 1 minute'
- You might find the MATLAB functions `rem` and idivide to be helpful.

- To format data into char array, see function sprintf.

8 Cell Arrays

`function [C_joined, C_nested] = cellConcatenate(C1, C2)`		
Input	Type	Description
`C1`	`Mx1 cell`	Cell array 1.
`C2`	`Mx1 cell`	Cell array 2.
Output	Type	Description
`C_joined`	`Mx2 cell`	A cell matrix with `C1` as the first column and `C2` as the second column.
`C_nested`	`1x2 cell`	A cell array holding `C1` within the first element and `C2` within the second element.

Details

Cell arrays and matrices are a more complex data structure than traditional arrays and matrices. A cell array can store elements of different types and sizes; a cell can even store another cell array! You will write a function that accepts two cell arrays: `C1` and `C2` as inputs and combines them in two different ways to form a larger structure with the information from both of the original cell arrays. `C_joined` will be formed by concatenating `C1` and `C2`, while `C_nested` will be created by nesting `C1` and `C2` into the cells of a larger cell array.

Tips

- Experiment with different combinations using [ ] and {} and see the result.

9 Array access time

`function [time1, time2, time3] = myArrayAccessTimer(matrix)`		
Input	Type	Description
`matrix`	`1000X1000`	any 1000-by-1000 array.
Output	Type	Description
`time1`	`1x1 double`	Time cost to get value of the first entry in a matrix
`time2`	`1x1 double`	Time cost to get value of the last entry in a matrix
`time3`	`1x1 double`	Time cost to get value of a random entry in a matrix

Details

We have learned a lot of interesting features about computers. Now think about one question: how long does a computer need to access the memory? To observe this feature in reality, you are going to use a 1000-by-1000 matrix as input and observe the time required for your computer to access a specified index.

Tips

- Look up tic-toc stopwatch of MATLAB, and think about how to use this feature to measure memory access time.
- Look up random integer, and think about how to use this feature to access a random entry in the matrix
- Observe what is printed to your Command Window and think about "What do these indicate?" Your thoughts will not be graded, but you can discuss it with your Lab GSI, which will be helpful to your future success in this class.

Submission Guidelines

When you are ready to test one or more of your function(s), download the `Lab02Tester.m` script from bCourses and run it in the same directory as your function(s). This will run a series of test cases for your function(s) and display the outputs. You should check these results with the example outputs displayed. Note that the series of tests that the script will run is not exhaustive and that it is up to you to make sure that your function fully satisfies the requirements of the problem. If you want to run all the test cases for one function, select that part of the tester script and push the `Run Section` button. If you wish to only check a single test case, you may do so by highlighting the appropriate lines in MATLAB and then pressing F9 (on Windows and linux) and **`shift + fn + F7`** (on a Mac) to run the selected text in the Command Window.

The script will also pack your function into a zip file for final submission. Your function file names and headers should match the examples given in the problem statement. If they do not match, the tester script will not be able to find them and will issue a warning. If you do not correct this problem, your missing function(s) will not be able to be graded and **you will receive 0 points**, so be sure to check carefully! Make sure to upload the `Lab02.zip` file to bCourses before the deadline. You may resubmit your zip file by re-uploading to bCourses before the deadline, but please be aware that only the most recent zip submission is considered for grading. That said, it is critical that you include your most recent version of **all** of your functions in each new zip submission.

Your final zip file for this assignment should contain the following:

- `mathOperation.m`
- `timeAddition.m`
- `decimalToQuaternary.m`

- `rotateVector.m`
- `recoverVector.m`
- `dataTypeMatters.m`
- `myTimeConversion.m`
- `cellConcatenate.m`
- `myArrayAccessTimer.m`