Programming Assignments

Introduction to Programming with MATLAB

Lesson 5

- Unless otherwise indicated, you may assume that each function will be given the correct number of inputs and that those inputs have the correct dimensions. For example, if the input is stated to be three row vectors of four elements each, your function is not required to determine whether the input consists of three two-dimensional arrays, each with one row and four columns.
- Unless otherwise indicated, your function should not print anything to the Command Window, but your function will not be counted incorrect if it does.
- Note that you are not required to use the suggested names of input variables and output variables, but you must use the specified function names.
- Also, read the instructions on the web page on how to test your functions with the auto-grader program provided, and what to submit to Coursera to get credit.
- Note that starred problems, marked by ***, are harder than usual, so do not get discouraged if you have difficulty solving them.
- We have not covered loops yet, so they are neither necessary nor allowed!
- You need MATLAB r2012a or newer or MATLAB Online to run the grader! Older versions are not supported.

1. Write a function called **`eligible`** that helps the admission officer of the Graduate School of Vanderbilt University decide whether the applicant is eligible for admission based on GRE scores. The function takes two positive scalars called **`v`** and **`q`** as input. They represent the percentiles of the verbal and quantitative portions of the GRE respectively. You do not need to check the input. The applicant is eligible if the average percentile is at least 92% and both of the individual percentiles are over 88%. The function returns the logical **`true`** or **`false`**.

2. Write a function called **`fare`** that computes the bus fare one must pay in a given city based on the distance travelled. Here is how the fare is calculated: the first mile is $2. Each additional mile up to a total trip distance of 10 miles is 25 cents. Each additional mile over 10 miles is 10 cents. Miles are rounded to the nearest integer other than the first mile which must be paid in full once a journey begins. Children 18 or younger and seniors 60 or older get a 20% discount. The inputs to the function are the distance of the journey and the age of the passenger in this order. Return the fare in dollars, e.g., 2.75 would be the result returned for a 4-mile trip with no discount.

3. Write a function called **`sort3`** that takes a 3-element vector as its sole arguments. It uses if-statements, possibly nested, to return the three elements of the vector as three scalar output arguments in non-decreasing order, i.e., the first output argument equals the smallest element of the input vector and the last output argument equals the largest element. NOTE: Your function may not use any built-in functions, e.g., **`sort`**, **`min`**, **`max`**, **`median`**, etc.

4. Write a function called **`day_diff`** that takes four scalar positive integer inputs, **`month1`**, **`day1`**, **`month2`**, **`day2`**. These represents the birthdays of two children who were born in 2015. The function returns a positive integer scalar that is equal to the difference between the ages of the two children in days. Make sure to check that the input values are of the correct types and they represent valid dates. If they are erroneous, return -1. An example call to the function would be

   ```
   >> dd = day_diff(1,30,2,1);
   ```

   which would make **`dd`** equal 2. You are not allowed to use the built-in function **`datenum`** or **`datetime`**.

5. Write a function called **holiday** that takes two input arguments called **month** and **day**; both are scalar integers representing a month (1-12) and a day (1-31). You do not need to check that the input is valid. The function returns a logical **true** if the specified date is a holiday; if not, it returns **false**. For the purposes of this exercise, the following dates are considered holidays: January 1st, July 4th, December 25th, and December 31st.

6. Write a function called **poly_val** that is called like this **p = poly_val(c0,c,x)**, where **c0** and **x** are scalars, **c** is a vector, and **p** is a scalar. If **c** is an empty matrix, then **p = c0**. If **c** is a scalar, then **p = c0 + c*x**. Otherwise, **p** equals the polynomial,

   $c0 + c(1)x^1 + c(2)x^2 + \cdots + c(N)x^N$,

   where N is the length of the vector **c**. Here are three example runs:

```
>> format long
>> p = poly_val(-17,[],5000)
p =
   -17
>> p = poly_val(3.2,[3,-4,10],2.2)
p =
   96.920000000000030
>> p = poly_val(1,[1,1,1,1],10)
p =
   11111
```

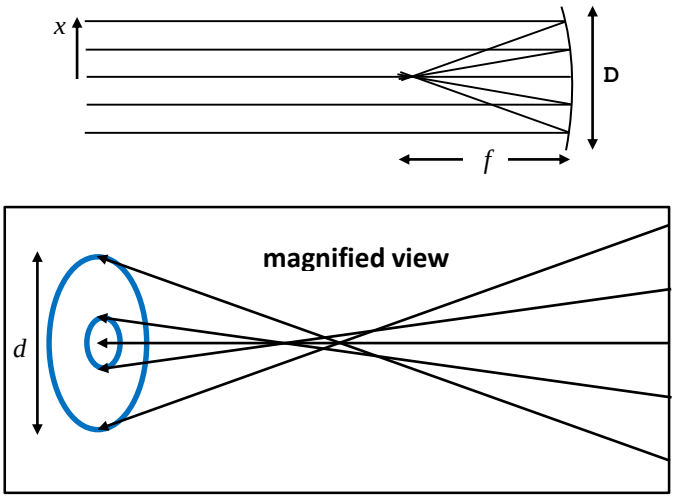
7. *** Write a function called **exp_average** that computes the "exponentially weighted moving average," or "exponential average" for short, of a sequence of scalars. The input sequence is provided to the function one element at a time and the function returns the current average each time. If we denote the n^{th} element of the input sequence, that is, the function input at the n^{th} invocation, by in_n, then the rule for calculating the corresponding average out_n that is to be returned by the function is:

   $out_1 = in_1$

   $out_n = b \cdot in_n + (1 - b) \cdot out_{n-1}$

   where b is a coefficient between 0 and 1. You do not need to check b. In plain English, the current average depends on the current input and the previously computed average weighted by b and (1 - b), respectively. Here is how the function is expected to work. When called by two input arguments, the input sequence is reset, the first input argument is considered to be in_1 and the second input argument is the value of the coefficient b. When called with a single input argument, it is considered to be in_t, that is, the current value of the input sequence. In both cases, the output is calculated according to the formula above. If the function is called for the very first time with a single input argument, the value of the coefficient b must default to 0.1.

8. Write a function that is called like this: **`mbd = spherical_mirror_aberr(fn,D)`**, where all arguments are scalars, **`fn`** is the "f-number" of a concave spherical mirror, **`D`** is its diameter in millimeters, and **`mbd`** is the mean blur diameter in millimeters. The f-number equals the focal length f of the mirror divided by its diameter. Ideally, all the rays of light from a distant object, illustrated by the parallel lines in the figure, would reflect off the mirror and then converge to a single focal point. The magnified view shows what actually happens. The light striking a vertical plane at a distance f from the mirror is spread over a circular disk. The light from the center of the mirror strikes the center of the disk, but light arriving from a point a distance x from the center of the mirror strikes the plane on a circle whose diameter d is equal to $2f\tan(2\theta)\left(\frac{1}{\cos\theta}-1\right)$, where $\theta = \sin^{-1}\left(\frac{x}{2f}\right)$. The function calculates d for all values of x in the vector **`x = 0:delta_x:D/2`**, where **`delta_x = 0.01`**. Then it calculates the mean blur diameter using this weighted formula:



$$\mathbf{mbd} = \frac{8 * \mathbf{delta\_x}}{\mathbf{D}^2} * \sum x_n d_n \quad ,$$

where the sum includes all **`x(n)d(n)`**, and returns it.

Here are three example runs:

```
>> format long
>> mbd = spherical_mirror_aberr(8,152)
mbd =
   0.029743954651679
>> mbd = spherical_mirror_aberr(8,300)
mbd =
   0.058695642823892
>> mbd = spherical_mirror_aberr(10,300)
mbd =
   0.037543964166654
```