Programming Assignments

Introduction to Programming with MATLAB

Lesson 6

- 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.
- You need MATLAB r2012a or newer or MATLAB Online to run the grader! Older versions are not supported.

1. The function **move_me** is defined like this: `function w = move_me(v,a)`. The first input argument **v** is a row-vector, while **a** is a scalar. The function moves every element of **v** that is equal to **a** to the end of the vector. For example, the command

   ```
   >> x = move_me([1 2 3 4],2);
   ```

   makes **x** equal to `[1 3 4 2]`. If **a** is omitted, the function moves occurrences of zeros.

2. Write a function called **halfsum** that takes as input an at most two-dimensional array **A** and computes the sum of the elements of **A** that are in the lower right triangular part of **A**, that is, elements in the counter-diagonal (going from the bottom left corner, up and to the right) and elements that are to the right of it. For example, if the input is `[1 2 3; 4 5 6; 7 8 9]`, then the function would return 38.

3. Write a function called **small_elements** that takes as input an array named **X** that is a matrix or a vector. The function identifies those elements of **X** that are smaller than the product of their two indexes. For example, if the element **X(2,3)** is 5, then that element would be identified because 5 is smaller than 2 * 3. The output of the function gives the indexes of such elements found in column-major order. It is a matrix with exactly two columns. The first column contains the row indexes, while the second column contains the corresponding column indexes. For example, the statement **indexes = small_elements([1 1; 0 4; 6 5]**, will make **indexes** equal to `[2 1; 1 2; 3 2]`. If no such element exists, the function returns an empty array.

4. Write a function called **approximate_e** that uses the following formula to compute e, Euler's number:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \cdots$$

   Instead of going to infinity, the function stops at the smallest **k** for which the approximation differs from **exp(1)** (i.e., the value returned MATLAB's built-in function) by no more than the positive scalar, **delta**, which is the only input argument. The first output of the function is the approximate value of e, while the second is **k**. (Note: if your program or the grader takes a long time, you may have created an infinite loop and need to hit Ctrl-C on your keyboard.) You are not allowed to use the built-in function **factorial**.

5. Write a function called **spiral_diag_sum** that takes an odd positive integer **n** as an input and computes the sum of all the elements in the two diagonals of the n-by-n spiral matrix. For example, starting with the number 1 and moving to the right in a clockwise direction, a 5-by-5 spiral is formed as follows:

```
21 22 23 24 25
20  7  8  9 10
19  6  1  2 11
18  5  4  3 12
17 16 15 14 13
```

   The sum of the red elements above is 101. Hint: the problem does not ask for the matrix itself. (Inspired by Project Euler.)

6. *** Write a function called **triangle_wave** that computes the sum

$$\sum_{k=0}^{n} \frac{(-1)^k \sin\big((2k+1)t\big)}{(2k+1)^2}$$

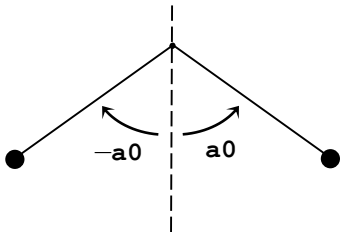
   for each of 1001 values of *t* uniformly spaced from 0 to 4π inclusive. The input argument is a scalar non-negative integer **n**, and the output argument is a row vector of 1001 such sums—one sum for each value of *t*. You can test your function by calling it with **n** == 20 or greater and plotting the result and you will see why the function is called "triangle_wave".

7. *** Write a function **max_product** that takes **v** a vector and **n**, a positive integer, as inputs and computes the largest product of **n** consecutive elements of **v**. It returns the product and the index of the element of **v** that is the first term of the product. If there are multiple such products in **v**, the function must return the one with the smallest starting index. As an example, the following call

```
>> [product, ind] = max_product([1 2 2 1 3 1],3);
```

   will assign 6 to **product** and 3 to **ind** since the max 3-term product in the input vector is 2*1*3. If **v** has fewer than **n** elements, the function returns 0 and -1, respectively.

8. *** Write a function called **pendulum** that is called like this: **T = pendulum(L,a0)**, where all arguments are scalars and **a0** is a positive number less than π. The function calculates the period **T** of a simple pendulum, which is the time required for a weight attached to a rod of length **L** and negligible weight to start from rest, swing with no friction under the influence of gravity from an initial angle **a0**, to – **a0** and back to **a0** again, as shown in the figure. The motion is determined by physics using the following definitions, where units [square brackets] are provided but are not needed:



θ = angle [radians]
ω = angular velocity [radians/s]
α = angular acceleration [radians/s^2]
g = acceleration due to gravity = 9.8 [m/s^2]
t = time [s]

The function starts its calculation with the pendulum angle θ equal to **a0** and then calculates a sequence of decreasing pendulum angles, each at a time separated from the one before it by $\Delta t = 1 \times 10^{-6}$ s. It continues until the pendulum has passed its lowest point, at which $\theta = 0$. The elapsed time equals **T/4**.

The calculation at each time step proceeds as follows: The angular acceleration α is set equal to $-g\sin\theta/$**L** . Then the angular velocity ω is increased by the product of the angular acceleration and Δt. That new angular velocity is then used to obtain a new θ by adding the product of the angular velocity and Δt to the old θ.

Here are two sample runs:

```
>> format long
>> T = pendulum(2, pi/2)
T =
   3.350344000012992
>> T = pendulum(0.22952, pi/4)
T =
   1.000000000000917
```