Solutions Lecture 2 (Sections 3.6-3.7 and 4.1-4.3)

Make sure to import Numpy to be able to use all its functionality.

```
import numpy as np
```

Note that all the questions below should be solved without using for-loops.

Question 1

Create the two-dimensional array

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{bmatrix}$$

by combining two functions seen in Chapter 3.

```
M = np.arange(1,25).reshape(6,4)

print(M)
```

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

Question 2

Create the array

$$x = [1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12]$$

from the first three rows of M (using reshaping functionality in combination with transposing a matrix)

```
x = M[0:3,:].T.flatten()

print(x)
```

```
[ 1  5  9  2  6 10  3  7 11  4  8 12]
```

Question 3

Implement the following function

$$f(x) = \begin{cases} -x+3 & \text{if } x < 0 \\ x^2+3 & \text{if } 0 \leq x < 1 \\ \sqrt{x^2+3}+2 & \text{if } x \geq 1 \end{cases}$$

so that it can handle both single numbers x and one-dimensional arrays x as input. Do not use for-loops or if/else statements. You might want to look at the Heavyside function example in Section 4.1 for some inspiration.

Your function should give the following output on the given input x.

```
x = np.arange(-5,6)
print(f(x))
```

```
[8.         7.         6.         5.         4.         3.
 4.         4.64575131 5.46410162 6.35889894 7.29150262]
```

```
def f(x):
    term1 = (-x + 3)*(x < 0)
    term2 = (x**2 + 3)*((x >= 0) & (x < 1))
    term3 = (np.sqrt(x**2+3)+2)*(x >= 1)
    return term1 + term2 + term3

x = np.arange(-5,6)
print(f(x))
```

```
[8.         7.         6.         5.         4.         3.
 4.         4.64575131 5.46410162 6.35889894 7.29150262]
```

Question 4

We will write a function that can compute the cumulative mean of a one-dimensional array. Take $n = 10$ (define this as a variable in your script).

a) Create the array $y = [1, 1/2, 1/3, \ldots, 1/(n-1), 1/n]$ using `arange()` in combination with a division.

```
n = 10
y = 1/np.arange(1,n+1)
```

```
print(y)
```

```
[1.         0.5        0.33333333 0.25       0.2        0.16666667
 0.14285714 0.125      0.11111111 0.1       ]
```

b) By combining your solution in part a) with `cumsum()`, create a function `cum_mean()` that takes as input a one-dimensional array $x = [x_0, \dots, x_{n-1}]$ and outputs the cumulative means of the array. This is the array that has at position i the value

$$\frac{1}{i+1}\sum_{j=0}^{i} x_i$$

for $i = 0, \dots, n-1$.

It should give the following output on the given input x.

```
# Some test data
x = np.array([1,4,2,5])
print(cum_mean(x))
```

```
[1.         2.5        2.33333333 3.        ]
```

```
# Define cumulative mean function
def cum_mean(x):
    # Input: One-dimensional array x
    # Output: Cumulative means of x

    n = np.size(x)
    y = 1/np.arange(1,n+1)
    return np.cumsum(x)*y

# Some test data
x = np.array([1,4,2,5])
print(cum_mean(x))
```

```
[1.         2.5        2.33333333 3.        ]
```

c) Vectorize your function of part b) so that it takes as input two-dimensional arrays, and outputs the cumulative mean of every row of the two-dimension array. Your function should give the following output on the given input matrix M.

```
# Some test data
M = np.array([[1,4,2,5],[1,10,12,8],[-1,9,3,-10]])
print(cum_mean(M))
```

```
[[ 1.          2.5         2.33333333  3.        ]
 [ 1.          5.5         7.66666667  7.75      ]
 [-1.          4.          3.66666667  0.25      ]]
```

```
# Cumulative mean function
def cum_mean(x):
    # Input: Two-dimensional array x
    # Output: Cumulative means of every row of x

    n = np.size(x,axis=1) #Or np.shape(x)[1]
    y = 1/np.arange(1,n+1)
    return np.cumsum(x,axis=1)*y

# Some test data
M = np.array([[1,4,2,5],[1,10,12,8],[-1,9,3,-10]])
print(cum_mean(M))
```

```
[[ 1.          2.5         2.33333333  3.        ]
 [ 1.          5.5         7.66666667  7.75      ]
 [-1.          4.          3.66666667  0.25      ]]
```

Question 5

Consider the following function

$$g(x_0, \dots, x_{n-1}) = \sum_{i=0}^{n-1} \sin(x_i) \cdot (x_i)^{2 \cdot i}$$

that takes as input an array $x = [x_0, \dots, x_{n-1}]$ and outputs $g(x)$.

a) Implement the function g. It should give the following output on the given input x.

```
# Some input data
x = np.array([1,4,2,6,4,5])
print(g(x))
```

```
-9427125.80618379
```

```
def g(x):
    n = np.size(x)
    return np.sum(np.sin(x)*(x**(2*np.arange(0,n))))

# Some input data
x = np.array([1,4,2,6,4,5])
print(g(x))
```

```
-9427125.80618379
```

b) Vectorize the function g so that it can take as input a two-dimensional array, and return the function value $g(x)$ for every row x of the array.

It should give the following output on the given input M.

```
# Some input data
M = np.array([[1,4,2,6,4,5],[1,4,2,6,4,5],[7,4,9,6,3,5]])
print(g(M))
```

```
[-9427125.80618379 -9427125.80618379 -9373912.93333995]
```

```
def g(x):
    n = np.shape(x)[1]
    return np.sum(np.sin(x)*(x**(2*np.arange(0,n))),axis=1)

# Some input data
M = np.array([[1,4,2,6,4,5],[1,4,2,6,4,5],[7,4,9,6,3,5]])
print(g(M))
```

```
[-9427125.80618379 -9427125.80618379 -9373912.93333995]
```

Note: The 5 in the bottom-right position `M[2,5]` used to be a 50. However, this number caused numerical inaccuracies in the output. The third number outputted by Python was of the order 10^8, although it should have been of the order 10^{16}. Some of the other numbers on the last row of M have also been adjusted. The first and second row are chosen the same to illustrate that the function indeed returns twice the same number of the rows are the same.

Question 6

Write a function `geom(x)` that takes as input a two-dimensional array, and outputs the geometric mean of every column of the array. For an array $x = [x_0, \dots, x_{n-1}]$, the geometric mean is defined as

$$\left(\prod_{i=0}^{n-1} x_i \right)^{1/n}$$

It should give the following output on the given input M.

```
# Some input data
M = np.array([[1,4,2,6,4,5],[1,4,3,7,1,5],[1,4,2,6,8,50]])
print(geom(M))
```

```
[ 1.          4.          2.28942849  6.3163596   3.1748021  10.77217345]
```

```
def geom(x):
    m = np.shape(x)[0]
    return np.prod(x, axis=0)**(1/m)

# Some input data
```

```
M = np.array([[1,4,2,6,4,5],[1,4,3,7,1,5],[1,4,2,6,8,50]])
print(geom(M))
```

```
[ 1.          4.          2.28942849  6.3163596   3.1748021  10.77217345]
```

Question 7

In this exercise we will normalize the data in an array, so that all entries are between 0 and 1.

a) Write a function `normalize()` that normalizes a (nonzero) array $x = [x_0, \dots, x_{n-1}]$ by replacing every entry x_i by

$$\frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$

where $x_{\min} = \min_i x_i$ and $x_{\max} = \max_i x_i$.

It should give the following output on the given input x.

```
# Some input data
x = np.array([1,4,2,-6,4,5])
print(normalize(x))
```

```
[0.63636364 0.90909091 0.72727273 0.         0.90909091 1.        ]
```

```
def normalize(x):
    return (x - np.min(x))/(np.max(x) - np.min(x))

# Some input data
x = np.array([1,4,2,-6,4,5])
print(normalize(x))
```

```
[0.63636364 0.90909091 0.72727273 0.         0.90909091 1.        ]
```

b) Vectorize your function so that it can normalize every column of a two-dimensional array using the formula in part a).

It should give the following output on the given input M.

```
# Some input data
M = np.array([[1,4,2,-6,4,5],[-4,3,5,1,3,2],[9,8,7,6,5,4]])
print(normalize(M))
```

```
[[0.38461538 0.2        0.         0.         0.5        1.        ]
 [0.         0.         0.6        0.58333333 0.         0.        ]
 [1.         1.         1.         1.         1.         0.66666667]]
```

```
def normalize(x):
    col_min = np.min(x,axis = 0)
    col_max = np.max(x,axis = 0)
```

```
    return (x - col_min)/(col_max - col_min)

# Some input data
M = np.array([[1,4,2,-6,4,5],[-4,3,5,1,3,2],[9,8,7,6,5,4]])
print(normalize(M))
```

```
[[0.38461538 0.2        0.         0.         0.5        1.        ]
 [0.         0.         0.6        0.58333333 0.         0.        ]
 [1.         1.         1.         1.         1.         0.66666667]]
```

Question 8

In this exercise we will implement a different type of data normalization.

a) Write a function `normal()` that normalizes a two-dimensional array (matrix) M such that the entries in each row have mean 0 and standard deviation 1. You can do this by substracting the mean of a row from every element in a row, and dividing every element by the standard deviation of the row.

It should give the following output on the given input M.

```
# Some test data
M = np.array([[ 1,  2,  3,  0],
       [ 5,  6,  -7,  0],
       [ -9, 10, 11, 0],
       [13, -13, 15, 0],
       [17, 18, 19, 0],
       [-21, -22, -23, 0]])
print(normal(M))
```

```
[[-0.4472136   0.4472136   1.34164079 -1.34164079]
 [ 0.77702869  0.97128586 -1.55405738 -0.19425717]
 [-1.47153441  0.85839508  0.98102294 -0.3678836 ]
 [ 0.82181649 -1.48815418  0.99950654 -0.33316885]
 [ 0.4472136   0.57498891  0.70276422 -1.72496673]
 [-0.47108153 -0.57576631 -0.6804511   1.72729894]]
```

```
def normal(x):
    row_means = np.mean(x, axis = 1) # Note that this is row array
    row_means = row_means[:,None] # Turn it into column array

    row_std = np.std(x, axis = 1)
    row_std = row_std[:,None]
    return (x - row_means)/row_std

# Some test data
M = np.array([[ 1,  2,  3,  0],
```

```
       [ 5,  6,  -7,  0],
       [ -9, 10, 11, 0],
       [13, -13, 15, 0],
       [17, 18, 19, 0],
       [-21, -22, -23, 0]])
print(normal(M))
```

```
[[-0.4472136   0.4472136   1.34164079 -1.34164079]
 [ 0.77702869  0.97128586 -1.55405738 -0.19425717]
 [-1.47153441  0.85839508  0.98102294 -0.3678836 ]
 [ 0.82181649 -1.48815418  0.99950654 -0.33316885]
 [ 0.4472136   0.57498891  0.70276422 -1.72496673]
 [-0.47108153 -0.57576631 -0.6804511   1.72729894]]
```

b) Verify that the rows have mean 0 using the `mean()` function from NumPy.

```
mu = np.mean(normal(M),axis=1)
print(mu) # All numbers are (almost) zero
```

```
[ 0.00000000e+00 -6.93889390e-18  0.00000000e+00  2.77555756e-17
  0.00000000e+00  0.00000000e+00]
```

c) Verify that the rows have standard deviation 1 using the `std()` function from NumPy.

```
sigma = np.std(normal(M),axis=1)
print(sigma) # All numbers are (almost) one
```

```
[1. 1. 1. 1. 1. 1.]
```