Lab Instructions - session 3

Row and Column Space, Linear Maps

Column Space and Row Space

The following code creates a figure with two subplots. In the left subplot, we plot a bunch of random 3D points in the column space of matrix **A**. The right subplot shows a set of 2D points in the row space of **A**.

`plot1.py`

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# create a 3 x 2 matrix
A = np.array([[1, 2],
              [3, 4],
              [-2,1]])

fig = plt.figure()

# A 1 by 2 subplot grid, subplot 1 (3D)
ax1 = fig.add_subplot(1,2,1, projection='3d')
ax1.set_title('column space')

for i in range(200):
    # create a random column vector
    u = np.random.randn(2,1)
    # create a point in the column space of A
    v = A @ u
    ax1.scatter(v[0,0], v[1,0], v[2,0], color='b')

# A 1 by 2 subplot grid, subplot 2 (2D)
ax2 = fig.add_subplot(1,2,2)
ax2.set_title('row space')

for i in range(200):
    # create a random row vector
    u = np.random.randn(1,3)
    # create a point in the row space of A
    v = u @ A
    ax2.plot(v[0,0], v[0,1], 'ro')
plt.show()
```

- Rotate the 3D plot. Do all the points lie in a lower-dimensional subspace?
- What is the dimension of the column space? What is the dimension of the row space?

Task 1 - Practice vectorized coding

You have to write the above without using the **`for`** loops. To create an m by n (normally distributed) random matrix use **`np.random.randn(m,n)`**. Notice that for a **2** by **n** matrix **A** containing **n** points as its columns, you may plot the points by giving the list of the x- and y-coordinates as the first and second argument of the **`plot`** function respectively:

```
ax.plot(A[0,:], A[1,:], 'o')
```

Similarly, for a **3** by **n** matrix containing 3D points, you may use

```
ax.scatter(A[0,:], A[1,:], A[2,:])
```

Likewise, you may plot the points represented as rows of a matrix.

Task 2

Repeat task 1 for the matrix

```
 1,  2
 3,  6
-2, -4
```

- What are the dimensions of the row and column spaces?

Task 3

Create a 2 by 3 subplot using **`fig.add_subplot(2,3,i, projection='3d')`** for plotting the column and row spaces of the following 3 by 3 matrices:

```
A = 1,  2,  1,
    2, -1, -1,
   -1,  1, -2

B = 1,  2, -3
    3,  1,  1
    2,  1,  0

C =  1,  2, -3
     3,  6, -9
    -2, -4,  6
```

The row and column spaces must be plotted in the subplot's first and second rows, respectively. The columns of the subplot correspond to the matrices **A**, **B**, and **C**.

- Rotate the plots. For each matrix, what are the dimensions of the row and the column spaces?
- What can you say about the row and column spaces of a matrix?
- Plot (the points in) the row and column spaces of matrix B in the same axes using two different colours. Repeat the same for matrix C. Are the row and column spaces of matrices equal in general?

Linear Transformations

Remember representing the shape of a face as a set of points from the previous lab. Here, we apply a linear transformation to each point.

`face1.py`

```
import matplotlib.pyplot as plt
import numpy as np
from face_data import Face1, edges

def plot_face(plt,X,edges,color='b'):
    "plots a face"
    plt.plot(X[:,0], X[:,1], 'o', color=color)

    for i,j in edges:
        xi,yi = X[i]
        xj,yj = X[j]

        plt.plot((xi,xj), (yi,yj), '-', color=color)

        plt.axis('square')
        plt.xlim(-100,100)
        plt.ylim(-100,100)

th = np.pi/6
A = np.array([[np.cos(th), np.sin(th)],
              [-np.sin(th), np.cos(th)]])

X = Face1 @ A
plot_face(plt, X, edges, color='b')
plt.show()
```

- Why does the above rotates the face counterclockwise, while the matrix **`A`** corresponds to a 30 degrees clockwise rotation (-30°)?

Task 4 - Linear Transformations

A. Animate the face to rotate around the origin by varying **`th`** from 0 to 2π. Use what you learned from the previous lab.
B. Apply a scaling transformation:

```
A = [[  α,     0  ],
     [  0,     α  ]]
```

- Animate by varying α from 3/4 to 4/3.
- What happens when alpha is negative?

C. Apply a non-uniform scaling transformation:

```
A = [[  α,     0  ],
     [  0,     β  ]]
```

- Animate by varying α from 3/4 to 4/3 and taking β = 1/α.

D. Shear the face (horizontally) by applying the transformation

```
A = [[  1,     0  ],
     [  s,     1  ]]
```

- Animate by varying **s** from **-0.7** to **0.7**.
- The matrix **A** above represents a vertical shear. Why does it perform a horizontal shear here?

Measuring the execution time

To measure the execution time of an operation or a piece of code put it inside a function and pass it to **timeit.timeit**:

measure_time.py

```
import numpy as np
import timeit

n = 1000

A = np.random.rand(n,n)

def f():
    np.linalg.inv(A)

t1 = timeit.timeit(f, number=1)
t2 = timeit.timeit(f, number=100)/100

print(t1)
print(t2)
```

- The execution time of what operation is measured?
- Which measurement is more reliable? **t1** or **t2**?

This can be done in a more compact way using the **lambda** functions:

```
timeit.timeit(lambda : np.linalg.inv(A), number=100)/100
```

Diagonal matrices

Execute the following and see the result.

diagonal.py

```
import numpy as np

D1 = np.diag([2,3,4])
D2 = np.diag([10,20,30,40])

print('D1=\n', D1)
print('D2=\n', D2)

A = np.array([[1,1,1,1],
              [1,2,2,2],
              [1,2,3,4]])

print('A=\n', A)
print('D1@A=\n', D1 @ A)

print('A@D2=\n', A @ D2)
```

- What is the effect of multiplying a diagonal matrix to the left and right?

Scaling rows and columns using broadcasting

This is an alternative to scaling the rows of a matrix using the concept of Broadcasting you learned in Lab 1.

scale_rows.py

```
import numpy as np

d1 = np.array([2,3,4]).reshape((3,1))

A = np.array([[1,1,1,1],
              [1,2,2,2],
              [1,2,3,4]])

print('d1=\n', d1)
print('A=\n', A)
```

```
print('d1.shape=\n', d1.shape)
print('A.shape=\n', A.shape)

print('d1 * A=\n', d1 * A)
```

- Write an equivalent code to scale columns of a matrix with numbers [10,20,30,40]. Is reshaping **`np.array([10,20,30,40])`** to shape **(1,4)** necessary for scaling columns? Why? (refer to the Broadcasting rules)
- Measure the execution time of **`d1*A`** and **`D1@A`** using **`timeit`** Which one is faster? Why?

Task 5

Compare the execution time of **`d1*A`** with **`D1@A`** for random matrices **`d1`** and **`A`** and **`D1=diag(d1.ravel())`**, where **`A`** is 100 by 200. Which one is faster? (Do not count the time of creating **`D1`** when computing **`D1@A`**.)

Task 6- Practice vectorized code

Consider the following:
`task2.py`

```
import numpy as np

m,n,p = 100,50, 2000

A = np.random.rand(m,n,p)
s = np.random.rand(p)

for i in range(p):
    A[:,:,i] *= s[i]
```

- In the above, replace the **`for`** loop with a single command.
- Compare the execution time of your code with the for loop using **`timeit`**.

Task 7 - Practice vectorized coding

Cosine distance is a measure of dissimilarity between two vectors, which is widely used in AI applications like natural language processing (NLP) for comparing word embeddings.
Given two vectors $\boldsymbol{a}$, $\boldsymbol{b}$ of the same dimension, cosine distance is calculated as follows:

$$CosineSimilarity(\boldsymbol{a}, \boldsymbol{b}) = \frac{\boldsymbol{a} \,.\, \boldsymbol{b}}{\|\boldsymbol{a}\| \; \|\boldsymbol{b}\|}$$

$$CosineDistance(\boldsymbol{a}, \boldsymbol{b}) = 1 - CosineSimilarity(\boldsymbol{a}, \boldsymbol{b})$$

Given two matrices `A` (shape m×d) and `B` (shape n×d), where each row represents a vector, the following code calculates the pairwise cosine distance between the rows of `A` and `B`.

`cosineDistance.py`

```
import numpy as np

m, n, d = 4, 3, 3
A = np.random.rand(m, d)
B = np.random.rand(n, d)

D = np.zeros((m, n))
for i in range(m):
    for j in range(n):
        dot_product = np.dot(A[i], B[j])
        norm_A = np.linalg.norm(A[i])
        norm_B = np.linalg.norm(B[j])
        cosine_similarity = dot_product / (norm_A * norm_B)
        D[i, j] = 1 - cosine_similarity
```

- Rewrite the above code in a fully vectorized form, without using loops, and ensure it performs the same computation. The new implementation should only require **one line**. You are not allowed to use `np.linalg` and `np.dot()`. You must print and compare the result of the above code with your own result.
- Test your code with `A=A**2` and `A= αA (where α is a scaler)`, and compare it with the previous result. What do you observe?
- Considering the previous question, explain why cosine distance is appropriate for NLP?