1928
K. N. Toosi University of Technology

Department of electrical engineering

Course: Linear Algebra

Professor: Dr. Mahsan Tavakoli

Lab Instructions - session 2

Introduction to Python

Deadline: 1403/01/25

Spring 2024

Prepared by
Ramin Tavakoli

Lab Instructions - session 2

Linear Combination, Span, Basis

How to drawing 3D vectors?

To draw 3D objects first add these three lines after importing matplotlib:

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

A vector can be plotted either as a *point* or an *arrow*. To plot a set of 3D points you can use the **scatter** function.

plot1.py

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

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# plot multiple points
u = np.array([1, 2,   3])
v = np.array([2, 0, -2])
w = np.array([-1, -1, -1])

xs = [u[0], v[0], w[0]]
ys = [u[1], v[1], w[1]]
zs = [u[2], v[2], w[2]]

ax.scatter(xs, ys, zs)
plt.show()
```

To plot an arrow you may use the **quiver** function:

`Plot2.py`

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

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# plot multiple points
u = np.array([1,2,3])
v = np.array([2, 0, -2])
w = np.array([-1, -1, -1])

xs = [u[0], v[0], w[0]]
ys = [u[1], v[1], w[1]]
zs = [u[2], v[2], w[2]]

# base of the vectors set to the origin
tail_x = [0,0,0]
tail_y = [0,0,0]
tail_z = [0,0,0]

ax.set_xlim(-3,3)
ax.set_ylim(-3,3)
ax.set_zlim(-3,3)

ax.quiver(tail_x, tail_y, tail_z, xs, ys, zs, color='r')
plt.show()
```

- Rotate the plot to view it from different angles. Do you think **u, v** and **w** are linearly dependent? If yes, how can you write one of them as a linear combination of the others?

Linear combination/span

The following code generates 2 random scalars **a** and **b** using the **`numpy.random.rand`** function and plots the linear combination **w = a u + b v** of the vectors **u** and **v**.

`Plot3.py`

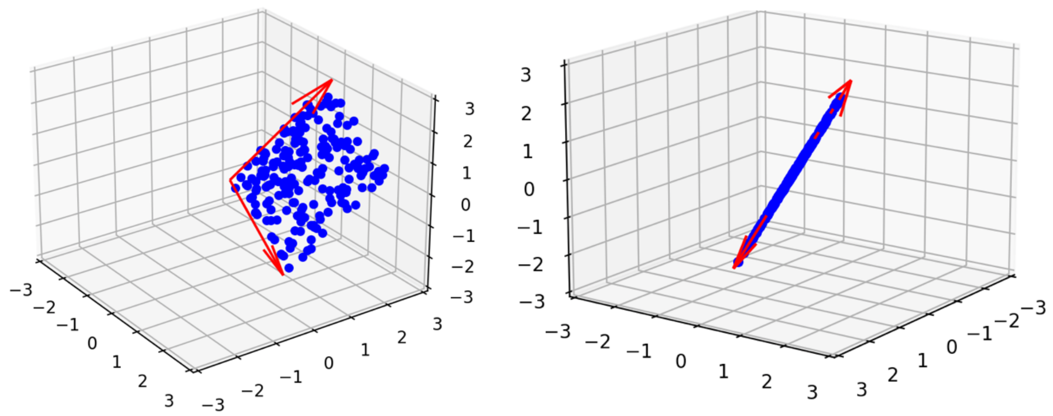
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# plot multiple points
u = np.array([1,2,3])
v = np.array([2, 0, -2])

xs = [u[0], v[0]]
ys = [u[1], v[1]]
zs = [u[2], v[2]]

# base of the vectors set to the origin
tail_x = [0,0]
tail_y = [0,0]
tail_z = [0,0]

ax.set_xlim(-3,3)
ax.set_ylim(-3,3)
ax.set_zlim(-3,3)

ax.quiver(tail_x, tail_y, tail_z, xs, ys, zs, color='r')

a,b = np.random.rand(2)
w = a * u + b * v
ax.scatter(w[0], w[1], w[2], color='b')

plt.show()
```

- Change the code to repeat plotting the linear combination **w** 200 times. This can be done by putting the following three lines in a loop:

```
a,b = np.random.rand(2)
w = a * u + b * v
ax.scatter(w[0], w[1], w[2], color='b')
```

Rotate the plot to view it from different angles and then you may see something like this:



- Notice that the plotted points are in **span(u,v)**. Rotate the plot to see this. Why is the shape of the scatter like that? Notice that the function **`numpy.random.rand`** generates random samples in the interval **[0,1)**.

- Replace the function **`numpy.random.rand`** with **`numpy.random.randn.`** What happens? and why?

Animating a plot

Run the following piece of code. What does it do?

Plot4.py

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

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

u = np.array([1, 2, 3])
v = np.array([2.0, 0, -2])

rng = np.linspace(0, 1, 20)

for alpha in rng:
    w = (1-alpha) * u + alpha * v

    ax.set_xlim(-4, 4)
    ax.set_ylim(-4, 4)
    ax.set_zlim(-4, 4)

    ax.quiver(0,0,0, u[0], u[1], u[2], color='r')
    ax.quiver(0,0,0, v[0], v[1], v[2], color='r')

    ax.quiver(0,0,0, w[0], w[1], w[2], color='b')
    ax.scatter(w[0], w[1], w[2], color='b')

plt.show()
```

- Rotate the plot to observe it from different angles.

- Add the following lines at the end of the body of the for loop. What happens?
  `plt.draw()`
  `plt.pause(.1)`
  * (if using Jupyter notebook, uncomment **`%matplotlib`** in the above to see the output correctly.)
- A linear combination **w = a u + b v** of two vectors **u** and **v** is an **affine** combination if **a + b = 1**. It is also called a **convex** combination if **a, b ≥ 0** in addition to **a + b = 1**. Are the vectors w created here are affine combinations of **u** and **v**? Are they also convex combinations?
- Change **`np.linspace(0,1,20)`** to **`np.linspace(-0.5,1.5,20)`**.
  - How does the plot change and why?
  - Are all the vectors **w** still affine combinations of **u** and **v**?
  - What about convex combinations?
- Add **`ax.cla()`** at the beginning of the **`for`** loop (cla stands for clear axis). What happens?

Shape models

A "shape" can be represented as an ordered or unordered set of points. Here, we represent a shape consisting of n points by an n by 2 matrix, each row of which represents a point. The following code creates a pair of 2D shapes and plots them:

`shape1.py`

```
import numpy as np
import matplotlib.pyplot as plt

n = 11

S1 = np.vstack((-np.cos(np.linspace(0, np.pi, n)),
                -.7+np.sin(np.linspace(0, np.pi, n)))).T


S2 = np.vstack((np.linspace(-1.2, 1.2, n),
                np.zeros(n))).T

print(S1.shape)
print(S2.shape)

plt.plot(S1[:, 0], S1[:, 1], 'bo-')
plt.plot(S2[:, 0], S2[:, 1], 'ro-')
```

```
plt.axis('equal')
plt.xlim(-2, 2)
plt.ylim(-2, 2)

plt.show()
```

- What are the shapes (dimensions) of S1 and S2?

Shapes, as defined above, form a vector space (can be scaled and added together). To look at matrices as vectors, you can **vectorize** them. That is to flatten an **n x 2** shape matrix to form a vector of size **2n**. Then perform addition, scaling, or linear combination:

```
s1 = S1.ravel()
s2 = S2.ravel()

s3 = a * s1 + b * s2
S3 = s3.reshape((n,2))
```

But, since matrices are added element-wise, you may simply write:

```
S3 = a * S1 + b * S2
```

- Plot the average shape S3 = 0.5 * S1 + 0.5 * S2.

Task 1 - Shape Morphing

Use what you learned in section "**Animating a plot**" (**`plt.draw, plt.pause, ax.cla`**) to animate the shape **S3** in the form of **`S3 = (1-alpha) * S1 + alpha * S2`**, by letting **`alpha`** range from 0 to 1 (convex combination). Use **`plt.cla()`** instead of **`ax.cla()`**.

- This is called **shape morphing**.
- Vary **`alpha`** from 0 to 1.5 (affine combinations). What happens?
- Try other ranges (e. g. -2 to 2). What's the output?
- Each shape has **2n** (here 22) entries. But all the shapes you see are in **span**(S1,S2), that is, they lie in a 2-dimensional subspace of a 22-dimensional vector space.
- (Point correspondence matter) Change **`-np.cos(np.linspace(0,np.pi,n))`** to **`np.cos(np.linspace(0,np.pi,n))`** when defining S1. What happens? Why?

Task 2 - Face Model

A face can be represented as a shape model consisting of a set of landmark points. The code below imports three faces Face1, Face2, and Face3 and plots Face1. Plotting a face is done using the function **`plot_face`** defined below. The file **`face_data.py`** has been provided to you.

`Task2a.py`

```
import matplotlib.pyplot as plt
import numpy as np

from face_data import Face1, Face2, Face3, edges

def plot_face(plt,X,edges,color='b'):
    "plots a face"

    plt.plot(X[:,0], X[:,1], 'o', color=color)

    i,j = edges[0]  # edge from node i to node j
    xi = X[i,0]
    yi = X[i,1]

    xj = X[j,0]
    yj = X[j,1]

    # draw a line between X[i] and X[j]
    plt.plot((xi,xj), (yi,yj), '-', color=color)

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

plot_face(plt, Face1, edges, color='b')

plt.show()
```

- The list **edge** contains a list of edges, each in the form of **`(i,j)`**. Print it to see how it looks.
- The function **`plot_face`** is supposed to plot the landmark points of the face, plus the edges between them. Currently, it only draws the first edge **`edge[0]`**. Change it to plot all the edges.

- Using the animation technique you learned above, morph a face shape from **Face1** to **Face2**, from **Face2** to **Face3**, and then from **Face3** back to **Face1**.
- Like before, try varying **alpha** from -.5 to 1.5 instead of 0 to 1.0 and see what happens.

For n vectors $v_1, v_2, \ldots, v_n$, a linear combination $a_1v_1 + a_2v_2 + \ldots + a_nv_n$ is called an affine combination if $a_1 + a_2 + \ldots + a_n = 1$. It is also a convex combination if all the scalars a_i are nonnegative. Here, we want to find linear combinations of **Face1, Face2**, and **Face3** to create **TargetFace1** and **TargetFace2**.

Task2b.py

```
import matplotlib.pyplot as plt
import numpy as np

from face_data import Face1, Face2, Face3, TargetFace1, TargetFace2,
edges

def plot_face(plt,X,edges,color='b'):
    "plots a face"

    plt.plot(X[:,0], X[:,1], 'o', color=color)

    i,j = edges[0]  # edge from node i to node j
    xi = X[i,0]
    yi = X[i,1]

    xj = X[j,0]
    yj = X[j,1]

    # draw a line between X[i] and X[j]
    plt.plot((xi,xj), (yi,yj), '-', color=color)

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


# make a guess
a = 1/3.
b = 1/3.
c = 1/3.

F = a * Face1 + b * Face2 + c * Face3
```

```
plot_face(plt, TargetFace1, edges, color='r')
plot_face(plt, F, edges, color='g')

# change a,b,c until the two plots align

plt.show()
```

- Find a convex combination of Face1, Face2, and Face3 to create TargetFace1. Keep tuning the scalars a, b, and c in the code until the blue and green plots align.
- Find a linear (not necessarily convex) combination to create TargetFace2. Assume a,b, and c are positive. Try to guess them yourself before reading the hint below:
    - a = 5.4 / 18 = ?.
- (Optional) Can you think of a way to find the scalars without trial and error?

Task 3 - Practice Vectorization

Consider an arbitrary matrix A and a vector u like the following

```
m, n = 20, 10
A = np.random.rand(m, n)
u = np.random.rand(n)
```

We perform the following operation on A and u to create the vector v.

```
v = np.zeros(m)
for i in range(n):
    v += A[:, i] * u[i]
```

- Write an equivalent program without loops in just a single line of code.

```
v = ...
```