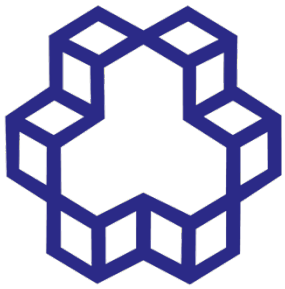

1928
K. N. Toosi University
of Technology

Department of electrical engineering

**Course**: Linear Algebra

**Professor**: Dr. Mahsan Tavakoli

**Lab Instructions - session 5**

**Linear Equations**

**Deadline**: -

Spring 2024

Prepared by
**Ramin Tavakoli**

# Lab Instructions - session 5

## Linear Equations

## Solving linear equations

Look at the code below.

`solve_eq.py`

```
import numpy as np
import timeit

N = 100
P = 1000

A = np.random.randn(N, N)
x = np.random.randn(N)
b = A @ x
# solve for x given A and b
x1 = np.linalg.solve(A, b)
x2 = np.linalg.inv(A) @ b

print("error1=", np.linalg.norm(x-x1))
print("error2=", np.linalg.norm(x-x2))

print("elapsed1=", timeit.timeit(lambda: np.linalg.solve(A, b), number=100))
print("elapsed2=", timeit.timeit(lambda: np.linalg.inv(A) @ b, number=100))
```

- What does the above do? Explain. (the function **`np.linalg.norm`** gives the length of a vector)
- Which method is more accurate? Using **`np.linalg.solve`** or using the inverse?
- Which method is faster?
- Set N to a larger number and look at the results.
- Set the true x to a matrix **`x = np.random.randn(N,P)`** with **`P=100`**, so that b becomes a matrix of the same size. Which method is faster? Choose a larger P. What happens?

# Task 1 – Purtub

In many real-world applications, we do not have access to the true vector of measurements **b = A x**, but rather a perturbed version of it (noisy measurements).

To simulate this scenario, we introduce noise by generating random perturbations and adding them to the calculated measurement vector:

```
noise = 0.001 * np.random.randn(N)
b = A @ x + noise
```

- solve the system of linear equations **A x = b** with the noisy version of **b**. Observe how the perturbation affects the error.
- Double the size of the perturbation and see how the error changes. To have a fair comparison you need to solve with **b= A@x + 2*noise** with the same values of **A, x**, and noise (do not recreate them).

# Task 2 - Singular case

Change the initial code (no noise added to b) such that **A** is singular. Here is one way to do it

```
A = np.random.randn(N,N)
A[:,N-1] = A[:,N-2]
```

- Why is **A** singular?
- What happens when you try to solve the equations?

# Task 3 - Near-singular case

In many real-world scenarios, the matrix **A** may not be exactly singular, but rather near singular. This means that the matrix is very close to a singular matrix (A singular matrix plus a small perturbation). Here we create such a scenario:

```
A = np.random.randn(N,N)
A[:,N-1] = A[:,N-2]
A += 0.00001 * np.random.randn(N,N)
```

- Compare the (order of) magnitude of the errors with the non-singular case. What happens as we bring A closer to a singular matrix (make 0.00001 smaller)?

## Task 4 - The effect of perturbation in near-singular case

Repeat task 1 but this time with a near-singular matrix **A**. How does a small perturbation affect the error in a near-singular case?

## Back to the Face Models

Remember the linear combination of faces from Lab 2 where you had to tune the scalars `a`, `b`, and `c` to reconstruct **TargetFace2** as a linear combination of **Face1**, **Face2**, and **Face3**:

`face1.py`

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

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

def plot_face(plt,X,edges,color='b'):
    "plots a face"
    plt.plot(X[:,0], X[:,1], 'o', color=color, markersize=3)
    for i,j in edges:
        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.

Face = a * Face1 + b * Face2 + c * Face3

plot_face(plt, TargetFace2, edges, color='r')
plot_face(plt, Face, edges, color='g')
# change a,b,c until the two plots align
plt.show()
```

In Lab 2 we found **a, b**, and **c** by trial and error. Now, we find them analytically. To do this, first, vectorize the 68 by 2 face matrices to obtain 136-dimensional vectors:

```
face1 = Face1.ravel()
face2 = Face2.ravel()
face3 = Face3.ravel()
t = TargetFace2.ravel()
```

Then, arrange the face vectors as the columns of a 136 by 3 matrix **F**:

```
F = np.stack((face1, face2, face3), axis=1)
```

Let **x = [a,b,c]T**. The relation **F x = t** gives a system of 136 equations and 3 unknowns. Solving for X gives the coefficients **a, b**, and **c**.

## Task 5

Choose the first 3 equations from the above to get a system of 3 equations and 3 unknowns. Solve the equations to find **a, b**, and **c**.

Task5.py

```
x =  # solve the equations
a, b, c = x
Face = a * Face1 + b * Face2 + c * Face3

plot_face(plt, TargetFace2, edges, color='r')
plot_face(plt, Face, edges, color='g')
plt.show()
```

## Task 6

Instead of choosing the first three equations from **F x = t**, choose three random equations. You may use **np.random.choice** to select 3 indices without replacement.

Task6.py

```
for i in range(5):
    inds = np.random.choice(range(136), 3, replace=False)
    # choose the equations
    a, b, c =  # solve the equations

    Face = a * Face1 + b * Face2 + c * Face3
    plot_face(plt, TargetFace2, edges, color='r')
    plot_face(plt, Face, edges, color='g')
```

```
    plt.show()
```

- Does choosing a different set of equations affect the result?

## Task 7 - Noisy measurements

Add a little Gaussian noise to the target face. This is similar to what we did in Task 1.

```
TargetFace2 += 3 * np.random.randn(*TargetFace2.shape)
```

Now, repeat Task 6. What happens?

- Do we get a face close to **TargetFace2** in most iterations?
- Does the quality of the result depend on the choice of the 3 equations?
- Why do you think this happens?
- (Optional) Can you think of a better way to find the coefficients **a,b,c**?