Cem Daloğlu
21602161 / 15.03.2021

EEE 361 Homework-2 Report

In this homework report, three different least square solvers are investigated and discussed. A, x and b are described as expected that is told in the homework statement.

- **Generalized Minimum Residual (GMRES)**

As stated in the lecture notes 2,

$$x_k = Q_k y_k \quad (eq.1)$$

Where $y_k = \underset{y_k}{min} \left\| H_{k+1,k} y_k - \|b\| e_1 \right\|$. To find y_k, QR decomposition can be applied since it is smaller least squares problem. The following figure shows the steps to find the y_k which minimizes the given equation.

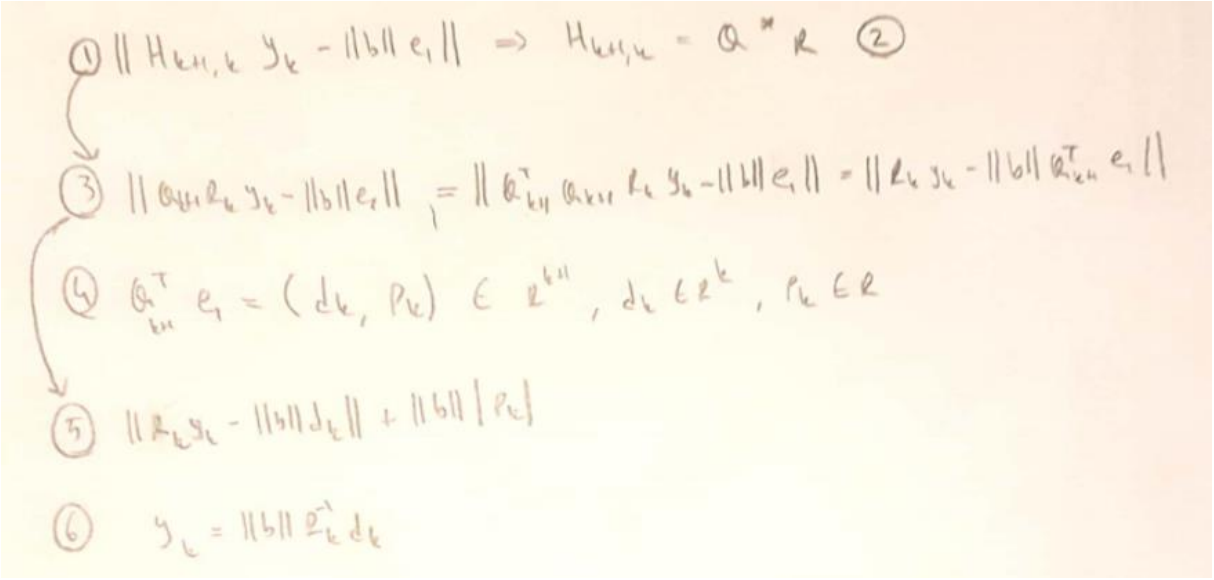

Figure 1: Step-by-step $y_k$computation

In the Figure 1, step 2 shows that QR decomposition is applied to $H_{k+1,k}$ and obtained Q and R where Q is a unitary matrix and R is an upper triangular matrix. R has form $R_k \in \mathcal{R}^{kxk}$. Step 3 shows the step 1's representation with QR decomposition. The final equation of the step 3's right hand side equation is partitioned in the step 4. Then, step 5 is obtained by combining last two steps. Finally,

$$y_k = \|b\| {R_k}^{-1} d_k$$

For the residual $\|b\| p_k$.

Since, Arnoldi method gives the Q_k (in the eq. 1) and y_k is found via QR decomposition method x_k can be computed. This is the GMRES algorithm, to summarize the steps of the algorithm;

- q_{k+1} and $h_{k+1,k}$ are computed by using Arnoldi's method where q_{k+1} and $h_{k+1,k}$ are the new columns of Q_k and $H_{k+1,k}$

- QR decomposition is applied to the new column of $H_{k+1,k}$ and does not change the previous columns.
- Partition is applied to the new column of Q since previous columns' partition is already done.
- Algorithm stops if the final iteration is applied or the error rate is significantly (can be changed by the user) small.

As a stopping criterion I checked if the k^{th} least square error ($\|b - Ax_k\|$) is sufficiently small (10^{-5} for instance) or if the iterations are over. If significantly smaller stop criteria is chosen and number of iterations are increased, the error will be lessen and the predicted x will be closer to the actual result.

For the Krylov subspace the algorithm generates Q_k vectors in the lecture notes. $Q_k \cong 0$ thus, the bases are orthogonal. However, I could not solve the problem in my GMRES function so I used built-in gmres function in scipy library.

The following Figures 2, 3 are obtained for this method.

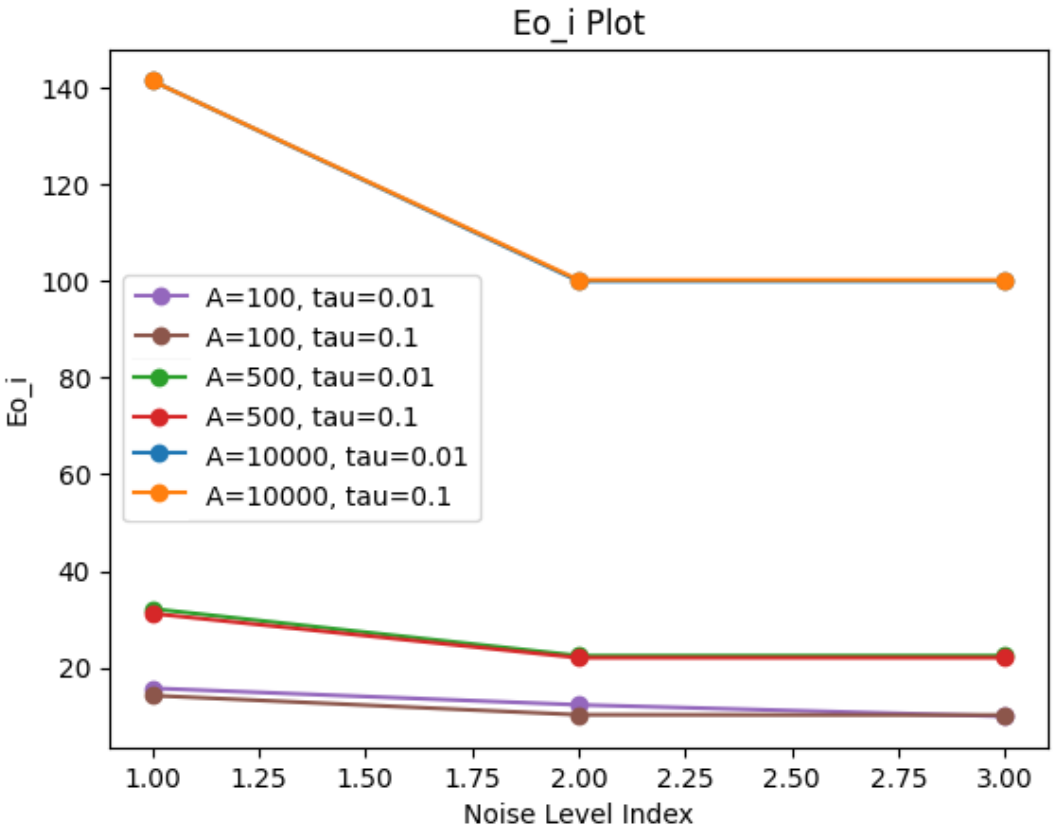

Figure 2: $E_{o,i}$ plot (there is a typo in the plot, all tau values are vice-versa)

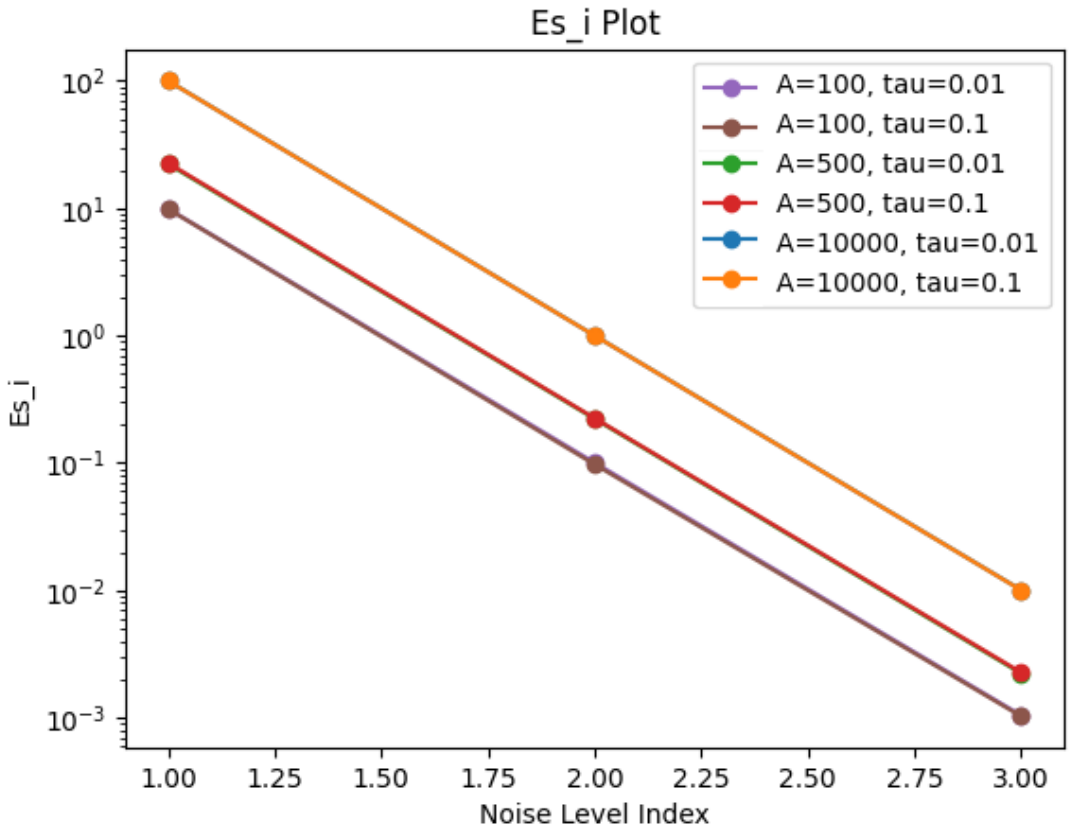

Figure 3: $E_{s,i}$ plot (there is a typo in the plot, all tau values are vice-versa)

For the Figures 2, 3, 4 and 5, $\sigma_w = 1, 0.01, 0.0001$ noise level indexes are 1, 2 and 3 respectively. Therefore, it can be said that smaller noise result as less error for both $E_{s,i}$ and $E_{o,i}$. Moreover, as the size of A increases, both errors decreases and the computed x gets closer to the actual x. The impact of τ can only be seen in Figure 2 and smaller τ has lower error.

- **Conjugate Gradient (CG)**

This method is applied exactly as in the second lecture notes (page 32). As a stopping criterion I checked if r_k is sufficiently small (10^{-5} for instance) or if the iterations are over. If significantly smaller stop criteria is chosen and number of iterations are increased, the error will be lessen and the predicted x will be closer to the actual result.

For the Krylov subspace the algorithm generates d_k vectors in the lecture notes. $d_k {d_k}^T \cong 0$ thus, the bases are orthogonal.

```
0.0030741781094622256
8.643087005468819e-08
5.390268368823885e-12
0.003100459068839958
8.689471570666074e-08
5.415418112873018e-12
```

Figure 4: $d_k {d_k}^T$ result of the last 6 operations for $\sigma_w = 1, 0.01, 0.0001$

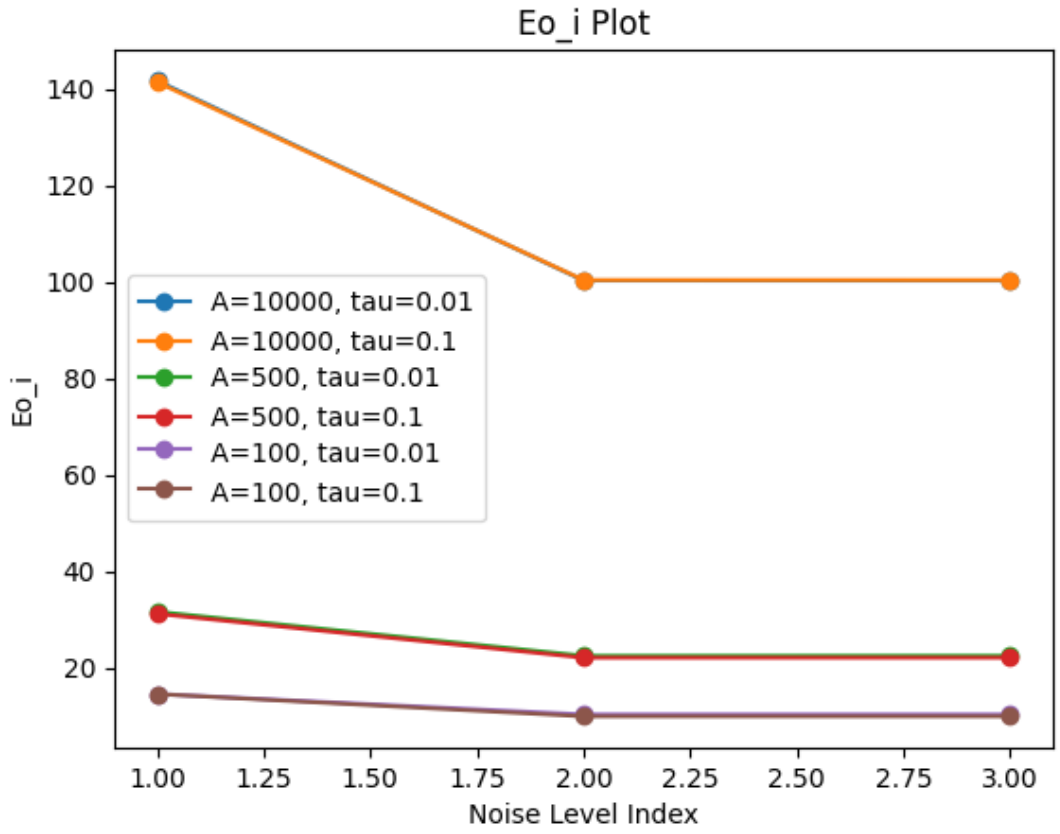

Figure 5: $E_{o,i}$ plot (there is a typo in the plot, all tau values are vice-versa)

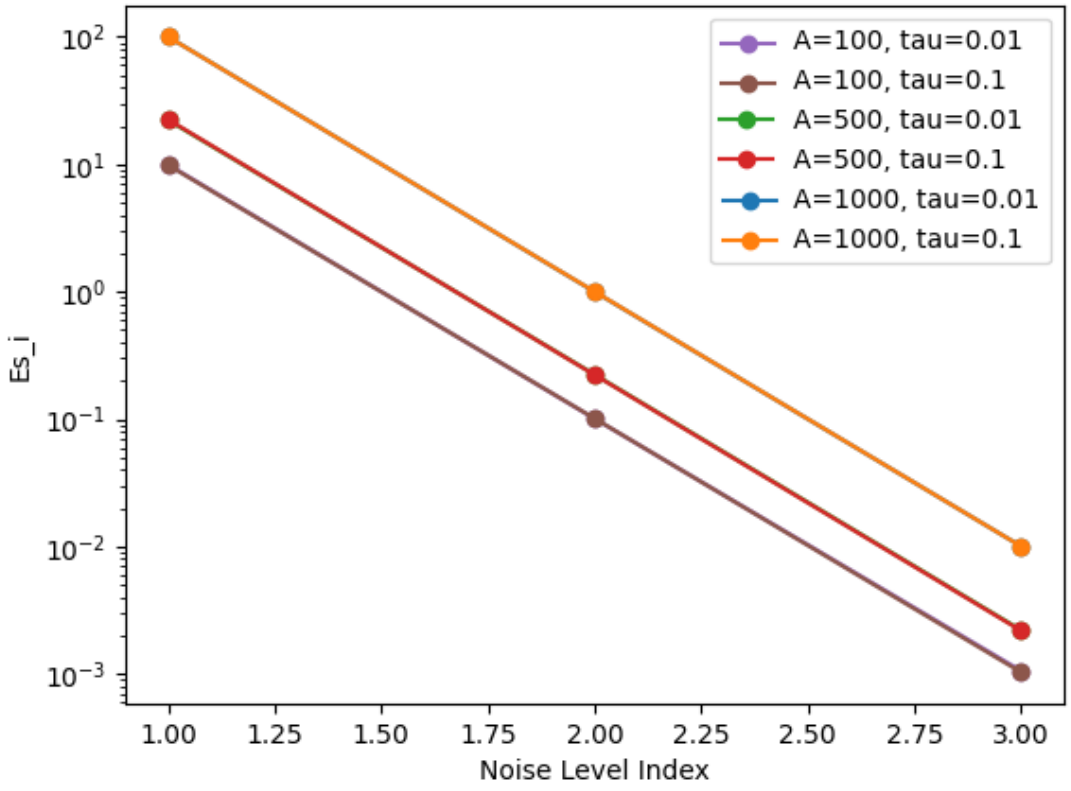

Figure 6: $E_{s,i}$ plot (there is a typo in the plot, all tau values are vice-versa)

By looking Figures 5 and 6, it can be said that smaller noise result as less error for both $E_{s,i}$ and $E_{o,i}$. Moreover, as the size of A increases, both errors decreases and the computed x gets closer to the actual x, which are the same results and observations of the previous method. Since similar stop criteria are used for GMRES and CG methods their plots are resulted as expected. The impact of τ cannot be seen. GMRES and CG methods gave similar error results for both $E_{s,i}$ and $E_{o,i}$.

- **Pseudo-inverse Method**

Since this method uses every data in the input matrix A, it takes the longest time so the maximum size A is chosen to be 1000.

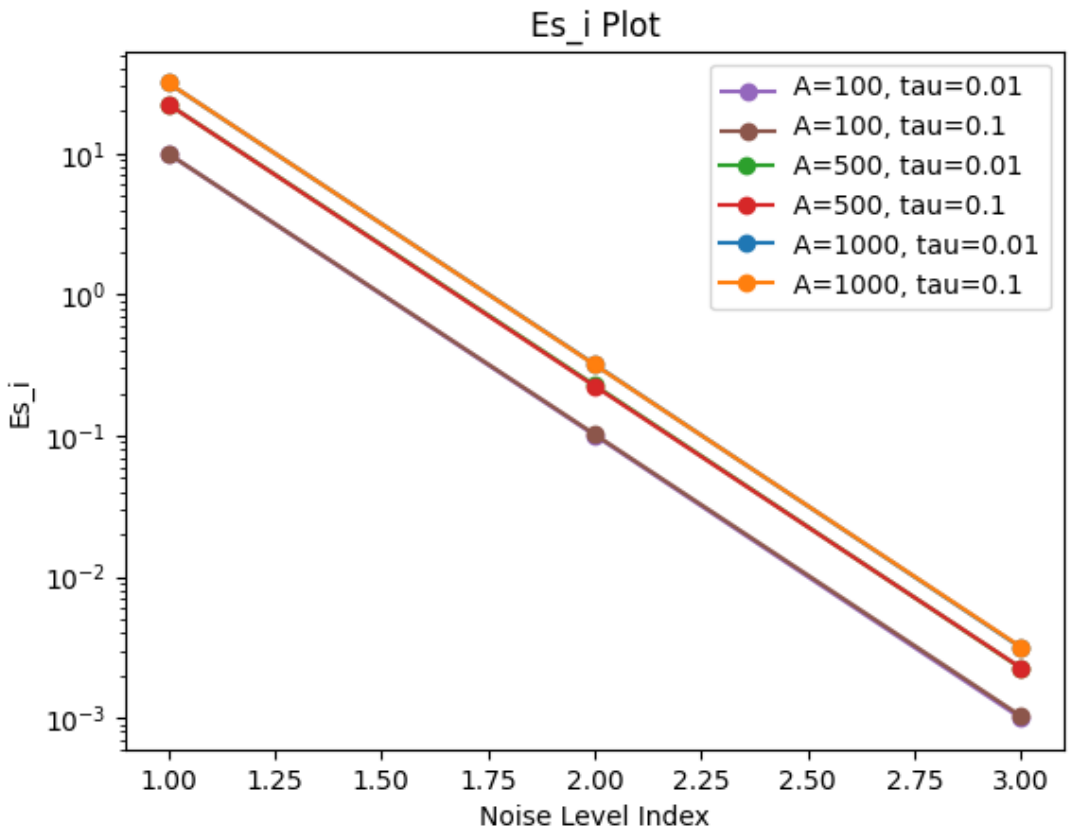

Figure 7: $E_{s,i}$ plot (there is a typo in the plot, all tau values are vice-versa)

Results are similar as the previous two results, so it can be said that similar performance values are obtained but GMRES and CG uses less data so their computation cost is less. Computation time for GMRES and CG are approximately 0.015 seconds for input size 500x500 and 0.15 for Pseudo-inverse method. For larger data pseudo-inverse method is problematic because of its time cost.

Appendix

```
import cv2 as cv
import matplotlib.pyplot as plt
import numpy as np
from scipy.sparse import linalg
import math
```

```
import copy
import time

size_A_array = [1000, 500, 100]
tau_array = [0.01, 0.1]
a_array = []

for size in range(3):
    size_A = size_A_array[size]

    a_diag = np.zeros((size_A, size_A), int)
    np.fill_diagonal(a_diag, 1)
    a_temp = np.zeros((size_A, size_A), float)
    for x in range(size_A):
        for y in range(size_A):
            if x != y:
                a_temp[x, y] = np.random.uniform(-1, 1, 1)
    a = a_diag + a_temp

    for tau_location in range(2):
        tau = tau_array[tau_location]

        for x in range(size_A):
            for y in range(size_A):
                if x != y:
                    if np.abs(a[x, y]) > tau:
                        a[x, y] = 0
        a_array.append(a)

def find_errors(a, b, e_ij, x):
    a = a
    b = b
    e_ij = e_ij
    x = x
    Es_i = np.zeros(3)
    Eo_i = np.zeros(3)

    for i in range(3):
        for j in range(10):
            Es_i[i] += np.square(np.linalg.norm(e_ij[i, j]))
        Es_i[i] = np.sqrt(1/10 * Es_i[i])

    for i in range(3):
        for j in range(10):
            Eo_i[i] += np.square(np.linalg.norm(b[i, j] - a @ x[i, j]))
        Eo_i[i] = np.sqrt(1/10 * Eo_i[i])

    return Es_i, Eo_i

def plot_error_estimated_sol(Es_i):
    Es_i = Es_i
    plt.plot([1, 2, 3],Es_i[:3], marker='o')
    plt.plot([1, 2, 3],Es_i[3:6], marker='o')
    plt.plot([1, 2, 3],Es_i[6:9], marker='o')
    plt.plot([1, 2, 3],Es_i[9:12], marker='o')
    plt.plot([1, 2, 3],Es_i[12:15], marker='o')
    plt.plot([1, 2, 3],Es_i[15:18], marker='o')
    plt.legend(["A=10000, tau=0.01", "A=10000, tau=0.1", "A=500, tau=0.01",
"A=500, tau=0.1", "A=100, tau=0.01",
                "A=100, tau=0.1"])
    plt.xlabel("Noise Level Index")
```

```
    plt.ylabel("Es_i")
    plt.yscale('log')
    plt.title("Es_i Plot")
    plt.show()

def plot_error_fit_observ(Eo_i):
    Eo_i = Eo_i
    plt.plot([1, 2, 3], Eo_i[0:3], marker='o')
    plt.plot([1, 2, 3], Eo_i[3:6], marker='o')
    plt.plot([1, 2, 3], Eo_i[6:9], marker='o')
    plt.plot([1, 2, 3], Eo_i[9:12], marker='o')
    plt.plot([1, 2, 3], Eo_i[12:15], marker='o')
    plt.plot([1, 2, 3], Eo_i[15:18], marker='o')
    plt.legend(["A=10000, tau=0.01", "A=10000, tau=0.1", "A=500, tau=0.01",
"A=500, tau=0.1", "A=100, tau=0.01", "A=100, tau=0.1"])
    plt.xlabel("Noise Level Index")
    plt.ylabel("Eo_i")
    plt.title("Eo_i Plot")
    # plt.yscale('log')
    plt.show()

def a_pseudo_method(a, b):
    a_pseudo = np.linalg.pinv(a)
    x = a_pseudo @ b
    return x


def conjugate_gradient(S, b, x0):
    x = copy.deepcopy(x0)
    r = b
    d = r
    r_k_norm = np.dot(r.T, r)
    for i in range(S[0].size):
        Sd = np.dot(S, d)
        alpha = r_k_norm / np.dot(d.T, Sd)
        x += alpha * d
        r += -1 * alpha * Sd
        r_kplus1_norm = np.dot(r.T, r)
        beta = r_kplus1_norm / r_k_norm
        r_k_norm = r_kplus1_norm
        if r_kplus1_norm < 1e-15:
            # print('Itr:', i)
            break
        d = r + beta * d
    return x

def gmres_method(S, b, x0):
    q = b / np.linalg.norm(b)
    x = []
    h = np.zeros((S[0].size + 1, S[0].size))
    v = np.zeros([S[0].size, 1])
    q = np.array(q).reshape([-1, 1])
    for i in range(S[0].size):
        if i == 0:
            v = np.outer(S[:, i], q[i])
        else:
            v = S[:, i] @ q[i]

        for j in range(i):
            h[j, i] = np.dot(q[j].T, v)
            v = v - h[j, i] * q[j]
```

```
        h[i + 1, i] = np.linalg.norm(v)

        if h[i + 1, i] != 0:
            q = v / h[i + 1, i]

        Q, R = QR_decomposition(h[i + 1, i])
    return x

# taken from
https://github.com/danbar/qr_decomposition/blob/master/qr_decomposition/qr_
decomposition.py
def QR_decomposition(A):
    """Gram-schmidt orthogonalization"""
    Q=np.zeros_like(A)
    for i in range(A.shape[1]):
        u = np.copy(A[:,i])
        for j in range(i):
            u -= np.dot(np.dot(Q[:, j].T, A[:,i]), Q[:, j])
        e = u / np.linalg.norm(u)
        Q[:, i] = e
    R = np.dot(Q.T, A)
    return (Q,R)


Es_i = np.zeros(3*6)
Eo_i = np.zeros(3*6)
r_arr = np.zeros(180)
for a_size in range(6):
    a = np.array(a_array[a_size])
    size_A = a[0].size
    sigma = [1, 0.01, 0.0001]
    x_0 = np.ndarray([10, size_A])
    x = np.ndarray([3, 10, size_A])
    x_intihal = np.ndarray([3, 10, size_A])
    e_ij = np.ndarray([3, 10, size_A])
    e_ij_intihal = np.ndarray([3, 10, size_A])
    b = np.ndarray([3, 10, size_A])
    for j in range(10):
        x_0[j, :] = np.random.randn(size_A)
        b_0 = a @ x_0[j]
        for i in range(3):
            w = sigma[i] * np.random.randn(size_A)
            b[i, j, :] = b_0 + w
            x[i, j, :] = a_pseudo_method(a, b[i, j, :])
            # check = linalg.cg(A=a, b=b[i, j, :], x0=np.zeros([size_A]))
// used for check purposes
            # check = linalg.gmres(A=a, b=b[i, j, :],
x0=np.zeros([size_A])) // used for plotting and check purposes
            # x[i, j, :] = conjugate_gradient(S=a, b=b[i, j, :],
x0=np.zeros([size_A]))
            # x[i, j, :] = gmres_method(S=a, b=b[i, j, :],
x0=np.zeros([size_A])) // does not work properly so i used scipy library
gmres function to plot
            # check = np.asarray(check[0])
            # x_intihal[i, j, :] = check
            e_ij[i, j, :] = x_0[j] - x[i, j]
            # e_ij_intihal[i, j, :] = x_0[j] - x_intihal[i, j]

    Es_i[3*a_size:3*a_size+3], Eo_i[3*a_size: 3*a_size+3] = find_errors(a,
b, e_ij_intihal, x)
```

```
# plot_error_estimated_sol(Es_i)
plot_error_fit_observ(Eo_i)
```