AMCS 211: Homework 4

Worked jointly by Eya Ben Amar and Michael Samet

Cross-Well Tomography

The Coefficient Matrix G

To evaluate whether the coefficient matrix G is ill-posed or not, we compute the condition number which is defined as

$$\kappa_G = \frac{\mid \lambda_{max} \mid}{\mid \lambda_{min} \mid} \tag{1}$$

where $\mid . \mid$ is the modulus of a complex number, $\lambda_{max}, \lambda_{min}$ are respectively the largest and smallest eigenvalues of the matrix G. Numerically, we obtained that $\kappa_G \approx 1.27 \times 10^{18}$, which is a very high condition number, and which implies that the matrix G is ill-conditionned, and hence cannot be inverted to find a solution to the linear system of equations $Gx = d$.

Linear Least Squares (LLS)

An alternative way to solve the problem is by formulating it as a LLS problem, and solve for x which satisfies the following.

$$x^* = \arg\min_x \mid\mid Gx - d \mid\mid \tag{2}$$

Finding x which satisfies Equation 4 is equivalent to finding x which satisfies the following system of linear equations

$$\left(G^\top G + \alpha I\right) x = G^\top d, \quad \alpha \in \mathbb{R}. \tag{3}$$

However, this formulation cannot recover the problem parameters because the condition number of the matrix $G^T G$, is $\kappa = 5.63 \times 10^{19}$, hence $log_{10}\kappa \approx 19.75$, which means that the solution to this system will lose about 19 digits of accuracy, and hence is unreliable. This motivates us to formulate the problem as a regularized linear least squares problem.

Regularized Linear Least Squares

Our aim now is to find x which satisfies the following

$$x^* = \arg\min_x \frac{1}{2} \mid\mid Gx - d \mid\mid^2 + \frac{\alpha}{2} \mid\mid x \mid\mid^2 \tag{4}$$

Which is equivalent to finding x that satisfies the following linear system of equations

$$\left(G^\top G + \alpha I\right) x = A^\top d \tag{5}$$

In order to choose the parameter α which controls the degree of regularization, we will solve the system given in (5), for different values of α, ranging between 10^{-6}, and 10^5, and observe its effect on the solution to the problem by visualising the heatmap of x at all the 16×16 grid points, as shown in Figure 2. Qualitatively, we can remark that for very big values of α, $\alpha \geq 1000$, and very small values of α, $\alpha \leq 10^{-5}$ the solution does not provide a valuable information at a visual level, compared to moderate values of α

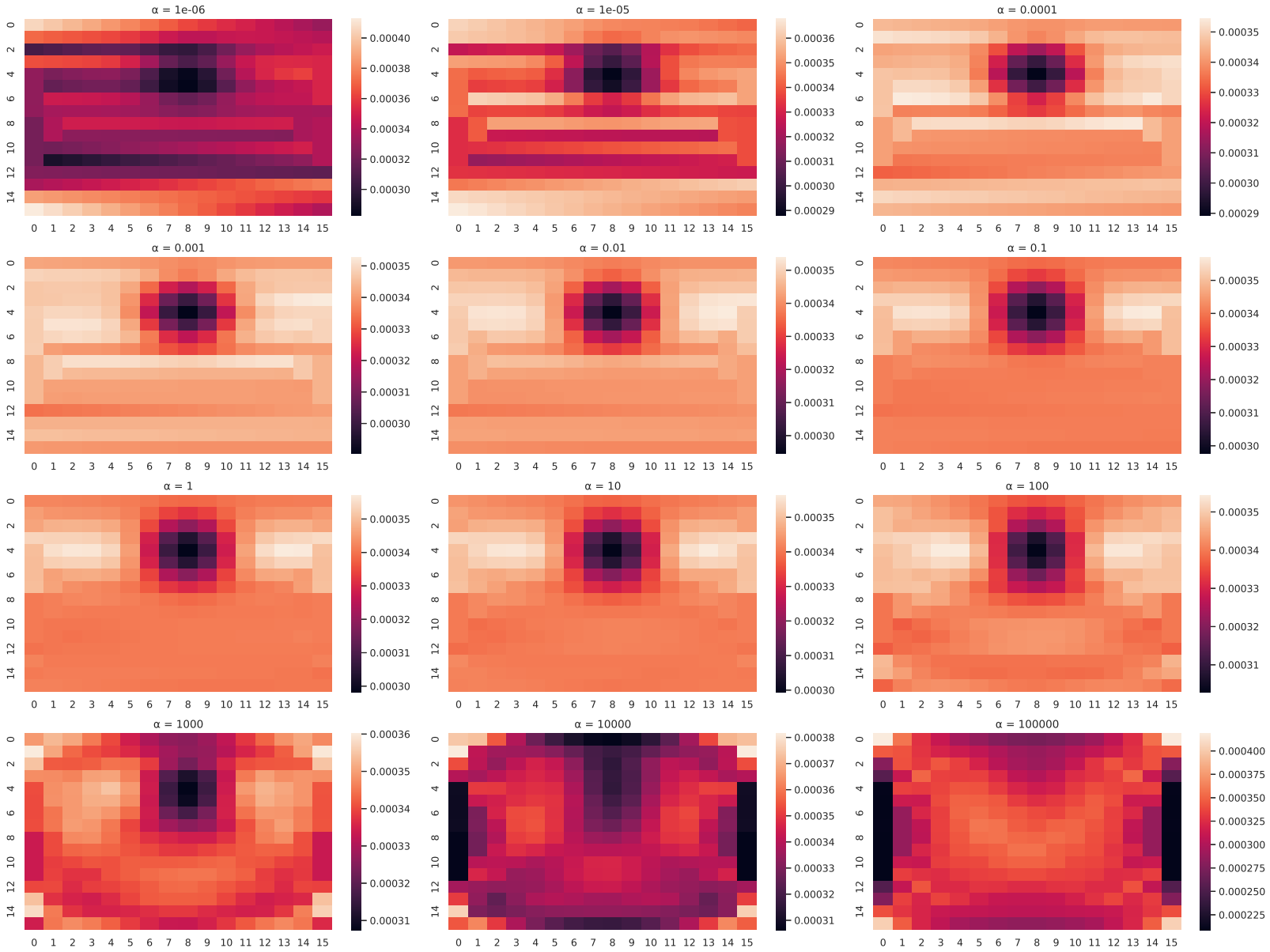


Figure 1: Reshaped vector x corresponding to solution of regularized linear least squares for different values of α.

Moreover, since some values of α lead to indistinguishable plots, a more numerical criteria might be useful, such as for instance, computing the mean squared error corresponding to each solution related to a used α

$$MSE_\alpha = \frac{1}{256}\sum_{i=1}^{256}((Gx_\alpha)_i - d_i)^2 \tag{6}$$

α	MSE_α
10^{-6}	3.073791e-07
10^{-5}	6.387229e-07
10^{-4}	1.116007e-06
10^{-3}	1.521944e-06
10^{-2}	3.687792e-06
10^{-1}	6.102643e-06
1	1.315154e-05
10	9.581193e-05
10^2	7.503911e-04
10^3	6.294515e-03
10^4	5.218317e-02
10^5	3.922552e-01

From a quantitative perspective, a good choice for α is the one which minimizes the error with respect to some metric, for instance, the mean-squared error MSE, which in this case corresponds to $\alpha = 10^{-6}$. This metric treats the value of the error equally at any point in the grid, whereas in reality, we would be more flexible with having errors far from the source of petroleum, as long as we are able to extract something meaningful more accurately. Hence, to choose the best α, we will consider the combination of the qualitative and quantitative perspectives, by picking the α which yields to the most plausible plot, in the sense that it captures a certain pattern, but also minimizes the MSE, which corresponds to $\alpha = 10^{-4}$, and hence the MSE is equal to $e = 1.116 \times 10^{-6}$

Singular Value Decomposition (SVD)

Now, we will use another regularization technique to solve the ill-conditioned least squares problem, namely we will use the SVD decomposition of the matrix G, into the product of the following matrices $G = U\Sigma V^T$, then we will compute the optimal x which minimizes the least squares, and which is given by

$$x^* = \sum_{i=1}^{r}\frac{u_i^t d}{\sigma_i}v_i \tag{7}$$

Where r corresponds to the number of singular values that we will keep and use in our approximation of the pseudo-inverse of G, one way to find r is actually to truncate the

possible values of the vector of singular values, σ_i, of G, such that they are bigger than certain threshold. To get a sense of what threshold to choose, we actually plotted the reshaped solution vector x for different choices of r, obtained by different thresholds, and kept the one which both yields an illustrative heatmap, and leads to reasonable error. The values of r that we tested correspond to the number of singular values that satisfy certain condition $\sigma_i \leq \gamma$, for $\gamma \in \{10^{-3}, 10^{-2}, 10^{-1}, 1, 10^{1}, 10^{2}\}$. Moreover, to reinforce the visual intuition by a numerical metric, we provide the error in norm 2 corresponding to each choice of r.

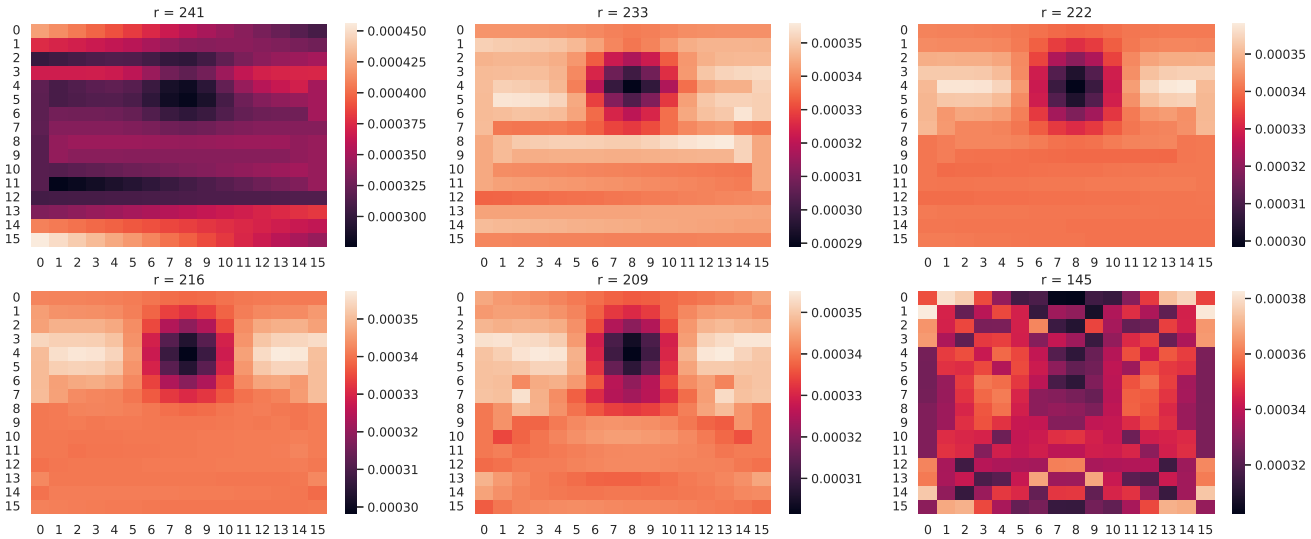


Figure 2: Reshaped vector x corresponding to the solution of the regularized LLS problem based on SVD with different values of r.

r	2-Norm Error
241.0	2.434085e-07
233.0	1.313737e-06
222.0	6.261107e-06
216.0	7.888358e-06
209.0	2.644042e-04
145.0	1.669668e-02
6.0	1.564555e+00

To conclude, we choose $r = 216$, which corresponds to $\sigma_i \geq 1, \forall i = 1, \ldots, 256$, and which yields 2-norm error of $e = 7.888 \times 10^{-6}$

Systems of Nonlinear Equations

Let

$$f(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ \\ f_2(\mathbf{x}) \end{bmatrix} \begin{bmatrix} \frac{2x_1+x_2}{(1+(2x_1+x_2)^2)^{\frac{1}{2}}} \\ \\ \frac{2x_1-x_2}{(1+(2x_1-x_2)^2)^{\frac{1}{2}}}. \end{bmatrix} \tag{8}$$

We want to find a solution to $f(\mathbf{x}) = 0$. we can use the basic Newton's method with unit step length. To do so, we need to compute the Jacobian of f.

$$\nabla f_1(\mathbf{x}) = \begin{bmatrix} 2\left(1+(2x_1+x_2)^2\right)^{-\frac{1}{2}} - 2(2x_1+x_2)^2\left(1+(2x_1+x_2)^2\right)^{-\frac{3}{2}} \\ \\ \left(1+(2x_1+x_2)^2\right)^{-\frac{1}{2}} - (2x_1+x_2)^2\left(1+(2x_1+x_2)^2\right)^{-\frac{3}{2}} \end{bmatrix} \tag{9}$$

$$\nabla f_2(\mathbf{x}) = \begin{bmatrix} 2\left(1+(2x_1-x_2)^2\right)^{-\frac{1}{2}} - 2(2x_1-x_2)^2\left(1+(2x_1-x_2)^2\right)^{-\frac{3}{2}} \\ \\ -\left(1+(2x_1-x_2)^2\right)^{-\frac{1}{2}} + (2x_1-x_2)^2\left(1+(2x_1-x_2)^2\right)^{-\frac{3}{2}} \end{bmatrix} \tag{10}$$

Then, we get

$$\mathrm{J(x)}= \begin{bmatrix} \nabla f_1(\mathbf{x})^T \\ \\ \nabla f_2(\mathbf{x})^T \end{bmatrix} \tag{11}$$

First, we start by the initial point $x_0 = [0.3, 0.3]^T$. We solve:

$$J^k \Delta x = -f^k$$
$$x^{k+1} = x^k + \Delta x$$

until we have $||f^k|| < 10^{-6}$. The algorithm converges to the minimizer
$x^* = [-1.90050844e-12, -3.80101689e-12]^T$ after $\boldsymbol{k = 6}$ iterations (iteration of x_0 included). We can confirm the quadratic convergence of the algorithm, both in the iterates and in the residuals, by plotting in Figure 3 and 4 $\mid f(\mathbf{x_k}) - f(\mathbf{x}^*) \mid$ and $\mid \mathbf{x_k} - \mathbf{x}^* \mid$, against the number of iterations k.

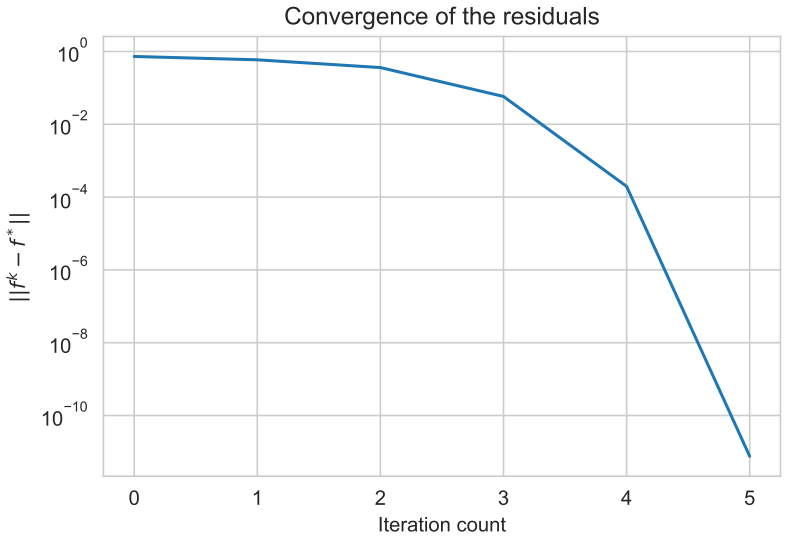


Figure 3: Error convergence of the residuals with the basic Newton method against the number of iterations.

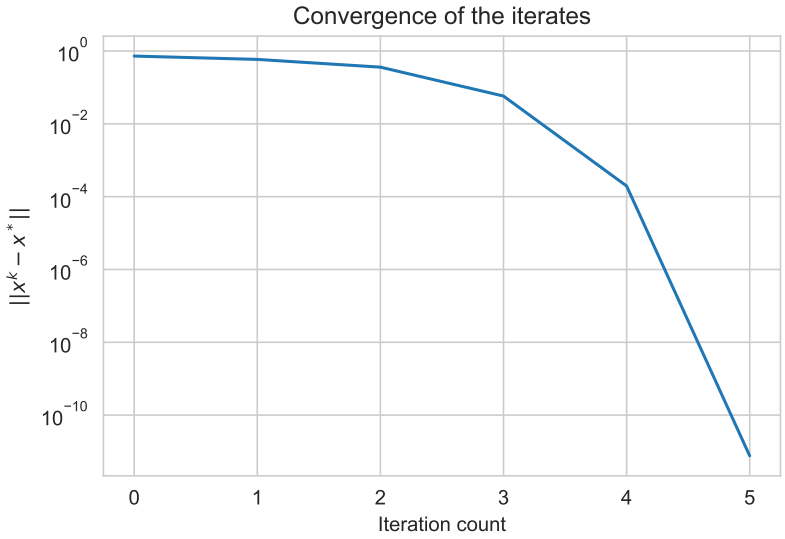


Figure 4: Error convergence of the iterates with the basic Newton method against the number of iterations.

Now, we will try to apply the basic Newton method with different starting point $x_0 = [0.5, 0.5]^T$ which is less to the solution. We notice that the algorithm diverges because the Jacobian matrix is singular in one of the iterates. We can say that the basic Newton method is sensitive to the starting point.

In order to solve this problem, we can globalize the Newton method ; i.e a line-search with an appropriate scalar merit function may be used at every iteration. In fact, finding the root of f is equivalent to minimizing the merit function $m(\mathbf{x}) = \frac{1}{2} f(\mathbf{x})^T f(\mathbf{x})$. We know that $\nabla m(\mathbf{x}) = J(\mathbf{x})^T f(\mathbf{x})$. Therefore, we can find the line search t in the direction of $\mathbf{p}^k$ with the objective function $m(\mathbf{x^k})$. We implement this strategy with the starting point $\mathbf{x_0} = [0.5, 0.5]^T$. We notice that the algorithm converges after $\boldsymbol{k = 7}$ iterations to the solution $\mathbf{x}^*$.
Figure 5 illustrates the convergence behaviour of the Globalize Newton's method and verify the quadratic convergence of this method.

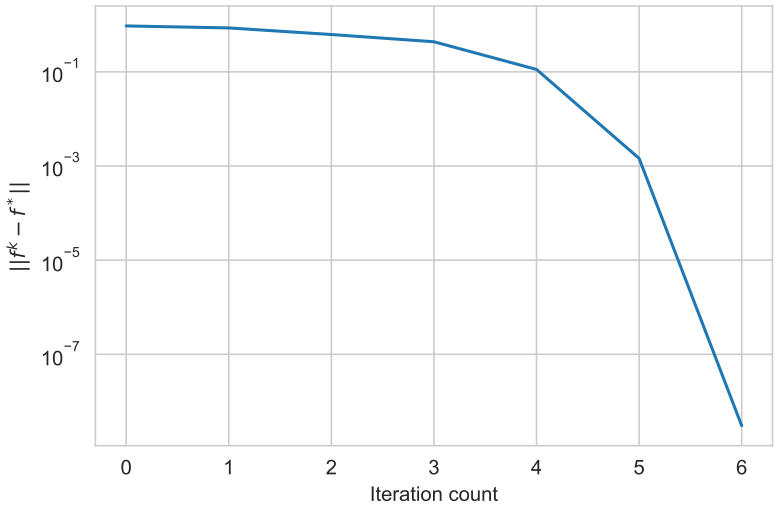


Figure 5: Error convergence of the Globalize Newton's method against the number of iterations.

```
import numpy as np
from numpy import linalg as la
import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
sns.set()
sns.set_style("whitegrid")
from google.colab import files
import warnings
import pandas as pd
warnings.filterwarnings("ignore") #warnings.filterwarnings(action='once')
```

Problem 1: Cross-Well Tomography

```
# importing the data
d = np.load('d.npy')
G = np.load('G.npy')
```

The matrix G is ill-conditionned, which can be shown by computing th e condition number in 2-norm.

```
la.cond(G)
```

```
1.2712687121978936e+18
```

If we formulate the problem as linear least squares problem, then we need to solve for $G^T G x = G^T d$, the condition number of the matrix $G^T G$ will determine the degree of accuracy of direct methods to solve this linear system of equations.

```
la.cond(G.T @ G)
```

```
5.637527291029533e+19
```

```
np.log10(la.cond(G.T @ G))
```

```
19.751088657292197
```

The $log_{10}(\kappa) = 19$, hence the solution to the system of linear equation using direct method will lose about 19 digits of accuracy, and hence is unreliable, which motivates the usage of regularization techniques to achieve higher numerical efficiency.

```
G.shape
```

```
(256, 256)
```

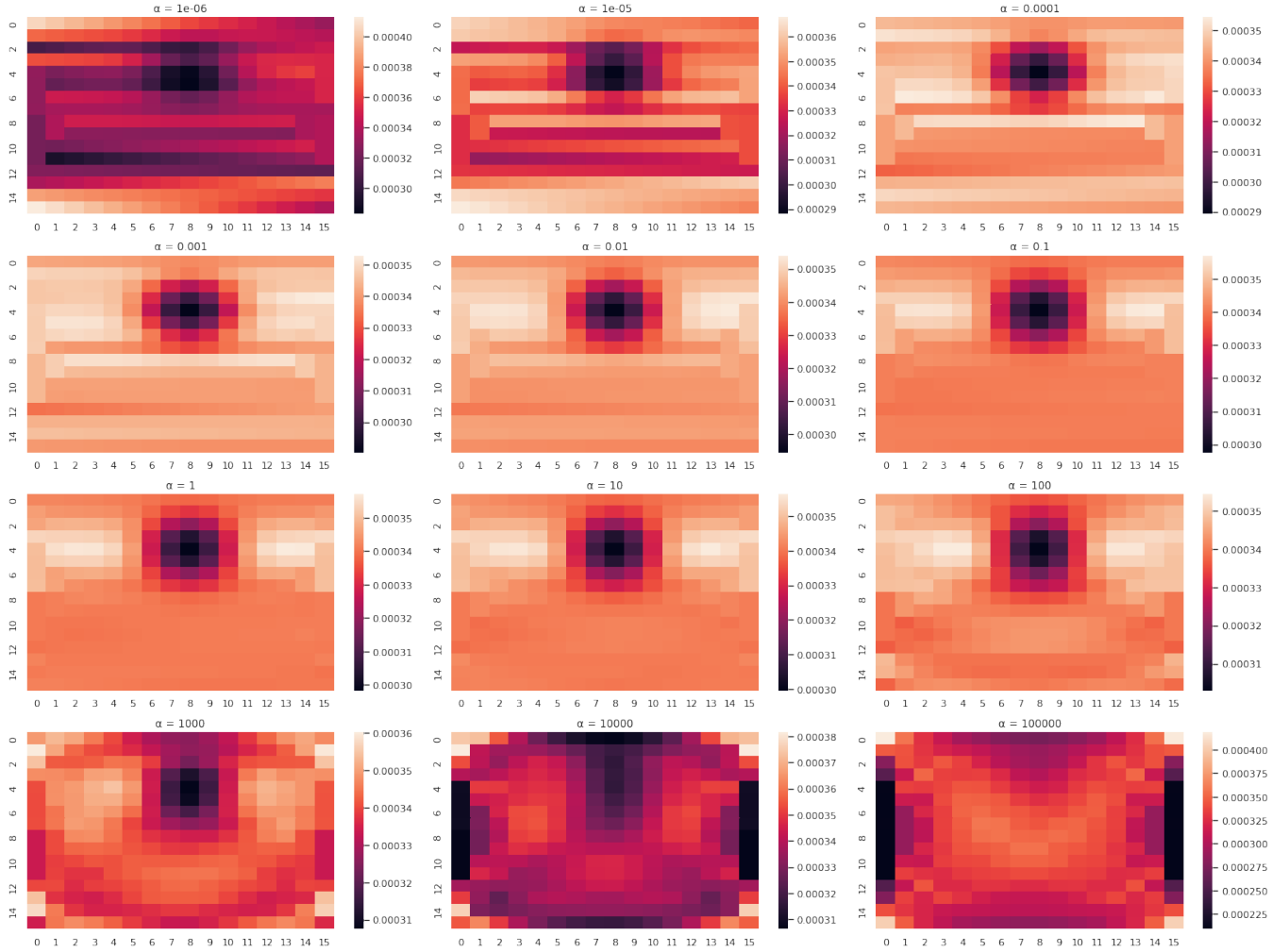
```
df = pd.DataFrame()
alpha_values = [10**i for i in range(-6,7)]
history = np.array([np.zeros(G.shape[0])])

for alpha in alpha_values:
  x = la.solve(G.T @ G + alpha*np.identity(G.shape[0]), G.T @ d)
  history = np.vstack( (history, x) )
  condition_number = la.cond(G.T @ G + alpha*np.identity(G.shape[0]))
  df2 = {'\u03B1': alpha, '\u039A':  condition_number}
  df = df.append(df2, ignore_index = True)

history = history[1:]
df
```

	α	K
0	0.000001	3.818102e+12
1	0.000010	3.818000e+11
2	0.000100	3.817991e+10
3	0.001000	3.817990e+09
4	0.010000	3.817990e+08
5	0.100000	3.817990e+07
6	1.000000	3.817991e+06
7	10.000000	3.818000e+05
8	100.000000	3.818090e+04
9	1000.000000	3.818990e+03
10	10000.000000	3.827990e+02
11	100000.000000	3.917990e+01
12	1000000.000000	4.817990e+00

```
plt.figure(figsize=(20,15))
```

```
plt.subplot(4,3,1)
x_reshaped = np.reshape(history[0],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[0])

plt.subplot(4,3,2)
x_reshaped = np.reshape(history[1],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[1])


plt.subplot(4,3,3)
x_reshaped = np.reshape(history[2],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[2])

plt.subplot(4,3,4)
x_reshaped = np.reshape(history[3],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[3])


plt.subplot(4,3,5)
x_reshaped = np.reshape(history[4],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[4])


plt.subplot(4,3,6)
x_reshaped = np.reshape(history[5],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[5])



plt.subplot(4,3,7)
x_reshaped = np.reshape(history[6],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[6])


plt.subplot(4,3,8)
x_reshaped = np.reshape(history[7],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[7])

plt.subplot(4,3,9)
x_reshaped = np.reshape(history[8],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[8])
```

```
plt.subplot(4,3,10)
x_reshaped = np.reshape(history[9],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[9])



plt.subplot(4,3,11)
x_reshaped = np.reshape(history[10],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B1 = %s"%alpha_values[10])



plt.subplot(4,3,12)
x_reshaped = np.reshape(history[11],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.tight_layout()
plt.title("\u03B1 = %s"%alpha_values[11])


pdf_name="heatmap"+".eps"
plt.savefig(pdf_name,format='eps', dpi=150)
files.download(pdf_name)
plt.show()
```



Which α to choose? What about alpha which miminizers the least squares the most?

```
errors = np.zeros(len(alpha_values))
df_error = pd.DataFrame()
for i in range(len(alpha_values)):
    x = history[i]
    errors[i] = la.norm(G@x - d)
    df2 = {'\u03B1': alpha_values[i], 'Error':  errors[i]}
    df_error = df_error.append(df2, ignore_index = True)
df_error
```

	α	Error
0	0.000001	3.073791e-07
1	0.000010	6.387229e-07
2	0.000100	1.116007e-06
3	0.001000	1.521944e-06
4	0.010000	3.687792e-06
5	0.100000	6.102643e-06
6	1.000000	1.315154e-05
7	10.000000	9.581193e-05
8	100.000000	7.503911e-04
9	1000.000000	6.294515e-03
10	10000.000000	5.218317e-02
11	100000.000000	3.922552e-01
12	1000000.000000	2.300743e+00

```
errors = np.zeros(len(alpha_values))
df_error = pd.DataFrame()
for i in range(len(alpha_values)):
    x = history[i]
    errors[i] = np.mean( (G@x - d)**2   )
    df2 = {'\u03B1': alpha_values[i], 'MSE':  errors[i]}
    df_error = df_error.append(df2, ignore_index = True)
df_error
```

	α	MSE
0	0.000001	3.690701e-16
1	0.000010	1.593621e-15
2	0.000100	4.865123e-15
3	0.001000	9.048105e-15
4	0.010000	5.312424e-14
5	0.100000	1.454775e-13
6	1.000000	6.756362e-13
7	10.000000	3.585909e-11
8	100.000000	2.199558e-09
9	1000.000000	1.547692e-07
10	10000.000000	1.063704e-05
11	100000.000000	6.010317e-04
12	1000000.000000	2.067742e-02

```
df_error.to_latex(index=False)
```

```
'\\begin{tabular}{rr}\n\\toprule\n             α &        Error \\\\\n\\mi
drule\n      0.000001 & 3.073791e-07 \\\\\n      0.000010 & 6.387229e-07 \
\\\\n      0.000100 & 1.116007e-06 \\\\\n      0.001000 & 1.521944e-06 \\\
\\n      0.010000 & 3.687792e-06 \\\\\n      0.100000 & 6.102643e-06 \\\\\
n      1.000000 & 1.315154e-05 \\\\\n     10.000000 & 9.581193e-05 \\\\\n
```

```
print(df_error[df_error.MSE == df_error.MSE.min()])
```

```
              α           MSE
    0  0.000001  3.690701e-16
```

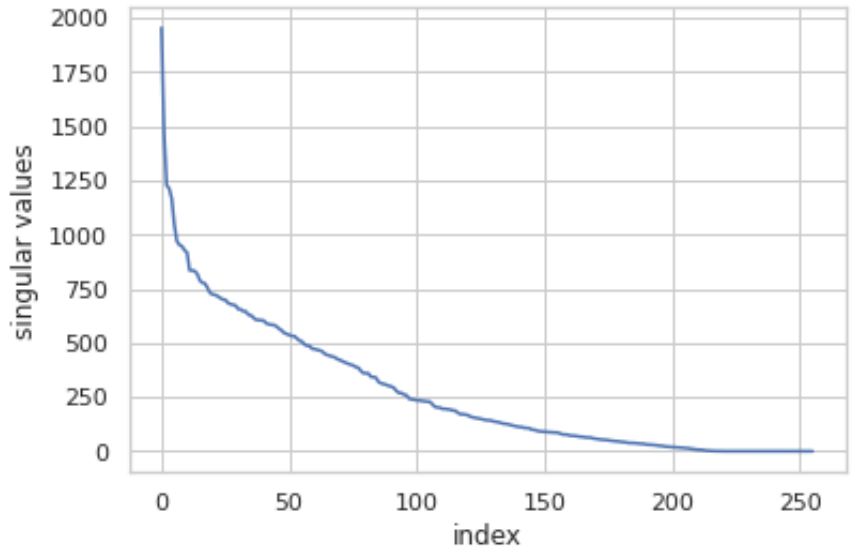
Regularization via Singular Value Decomposition (SVD)

```
U,sigma,VT = la.svd(G,full_matrices=False)
V = VT.T
n = G.shape[1]
#r = la.matrix_rank(G) # compute the rank of the matrix G.
r = len([sigma[i] for i in range(len(sigma)) if sigma[i] > 1])
sigma_inv = np.diag(np.hstack([1/sigma[:r], np.zeros(n-r)])) # constructing diag
G_plus = V @ sigma_inv @ U.T # G_plus is the Moore-Penrose pseudo-inverse matrix
x = G_plus @ d # computation of the solution
error = la.norm(G@x - d) #2-norm error
error
```

```
7.888357549430916e-06
```

```
plt.plot(sigma) # order of magnitude of singular values
plt.ylabel("singular values")
plt.xlabel("index")
```

```
Text(0.5, 0, 'index')
```



```
df = pd.DataFrame()
alpha_values = [10**i for i in range(-3,4)]
history = np.array([np.zeros(G.shape[0])])
r_values = []
for alpha in alpha_values:
  r = len([sigma[i] for i in range(len(sigma)) if sigma[i] > alpha])
  sigma_inv = np.diag(np.hstack([1/sigma[:r], np.zeros(n-r)])) # constructing di
  G_plus = V @ sigma_inv @ U.T # G_plus is the Moore-Penrose pseudo-inverse matr
  x = G_plus @ d # computation of the solution
  history = np.vstack( (history, x) )
  r_values.append(r)
  error = la.norm(G @ x - d)
  df2 = {'r': r, '2-Norm Error':  error}
  df = df.append(df2, ignore_index = True)

r_values = np.array(r_values)
history = history[1:]
df
```

	r	2-Norm Error
0	241.0	2.434085e-07
1	233.0	1.313737e-06
2	222.0	6.261107e-06
3	216.0	7.888358e-06
4	209.0	2.644042e-04
5	145.0	1.669668e-02
6	6.0	1.564555e+00

```
df.to_latex(index=False)
```

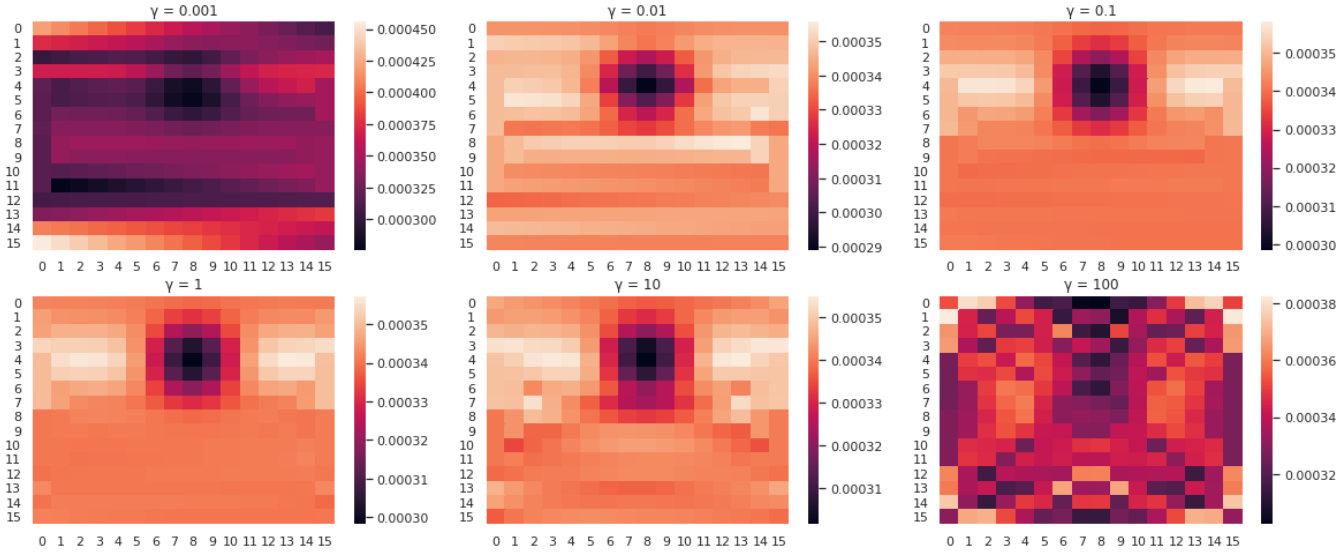
```
'\\begin{tabular}{rr}\n\\toprule\n    r &  2-Norm Error \\\\\n\\midrule\n2
41.0 &  2.434085e-07 \\\\\n233.0 &  1.313737e-06 \\\\\n222.0 &  6.261107e-
06 \\\\\n216.0 &  7.888358e-06 \\\\\n209.0 &  2.644042e-04 \\\\\n145.0 &
```

```
df2
```

```
{'2-Norm Error': 1.564554638462816, 'r': 6}
```

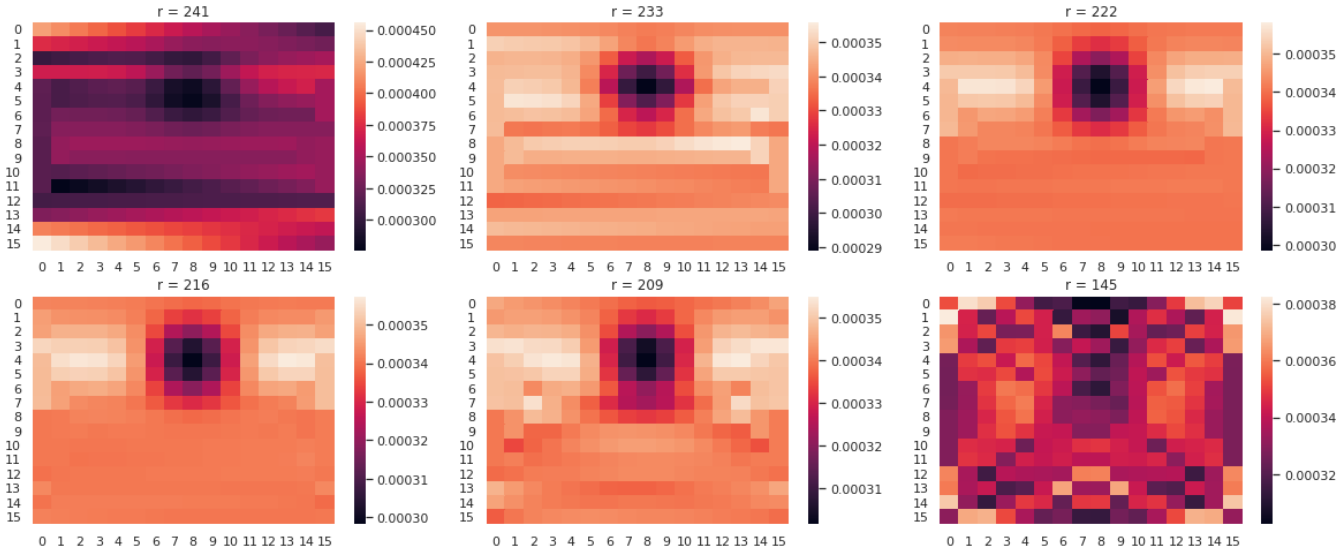
```
plt.figure(figsize=(20,8))

plt.subplot(2,3,1)
x_reshaped = np.reshape(history[0],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B3 = %s"%r_values[0])
```

```
plt.subplot(2,3,2)
x_reshaped = np.reshape(history[1],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B3 = %s"%alpha_values[1])


plt.subplot(2,3,3)
x_reshaped = np.reshape(history[2],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B3 = %s"%alpha_values[2])

plt.subplot(2,3,4)
x_reshaped = np.reshape(history[3],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B3 = %s"%alpha_values[3])

plt.subplot(2,3,5)
x_reshaped = np.reshape(history[4],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B3 = %s"%alpha_values[4])

plt.subplot(2,3,6)
x_reshaped = np.reshape(history[5],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("\u03B3 = %s"%alpha_values[5])

pdf_name="heatmap_svd"+".eps"
plt.savefig(pdf_name,format='eps', dpi=150)
files.download(pdf_name)
```




```
plt.figure(figsize=(20,8))

plt.subplot(2,3,1)
x_reshaped = np.reshape(history[0],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("r = %s"%r_values[0])

plt.subplot(2,3,2)
x_reshaped = np.reshape(history[1],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("r = %s"%r_values[1])


plt.subplot(2,3,3)
x_reshaped = np.reshape(history[2],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("r = %s"%r_values[2])

plt.subplot(2,3,4)
x_reshaped = np.reshape(history[3],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("r = %s"%r_values[3])

plt.subplot(2,3,5)
x_reshaped = np.reshape(history[4],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("r = %s"%r_values[4])
```

```
plt.subplot(2,3,6)
x_reshaped = np.reshape(history[5],(16,16))
sns.heatmap(x_reshaped,cbar=True)
plt.title("r = %s"%r_values[5])

pdf_name="heatmap_svd_r"+".eps"
plt.savefig(pdf_name,format='eps', dpi=150)
files.download(pdf_name)
```



```
import numpy as np
from numpy import linalg as la
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
def myf(x):
    f1=(2*x[0]+x[1])/((1+(2*x[0]+x[1])**2)**0.5)
    f2=(2*x[0]-x[1])/((1+(2*x[0]-x[1])**2)**0.5)
    return(np.array([f1,f2]))



def J(x):
    A=np.zeros((2,2))
    A[0][0]=2*((1+(2*x[0]+x[1])**2)**(-0.5))-2*(2*x[0]+x[1])**2*((1+(2*x[0]+x[1]
    A[0][1]=((1+(2*x[0]+x[1])**2)**(-0.5))-(2*x[0]+x[1])**2*((1+(2*x[0]+x[1])**2
    A[1][0]=2*((1+(2*x[0]-x[1])**2)**(-0.5))-2*(2*x[0]-x[1])**2*((1+(2*x[0]-x[1]
    A[1][1]=-((1+(2*x[0]-x[1])**2)**(-0.5))+(2*x[0]-x[1])**2*((1+(2*x[0]-x[1])**
    return A


x0=np.array([0.3,0.3])
J(x0)

     array([[ 0.8213195 ,  0.41065975],
            [ 1.75747942, -0.87873971]])


myf(x0)

     array([0.66896473, 0.28734789])


def newton(f, J, x0, tol = 1e-6):
    x = x0
    history = np.array( [x0] )
    while ( la.norm(f(x)) > tol ):
        p = la.solve(J(x), - f(x))
        x += p
        history = np.vstack( (history, x) )
    return x, history
```

$x_0 = [0.3, 0.3]^T$

```
xf,hist=newton(myf, J, x0, tol = 1e-6)
```

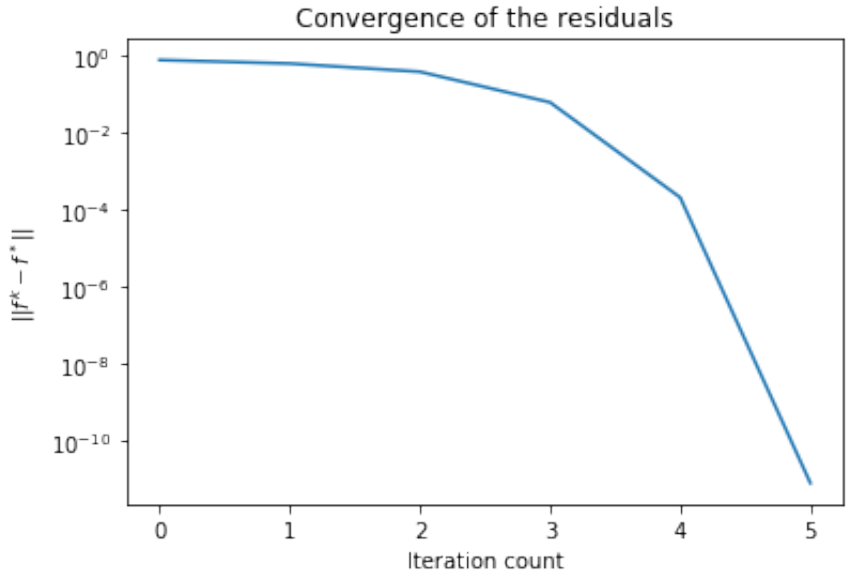
```
xf
```

```
    array([-1.90050844e-12, -3.80101689e-12])
```

```
hist
```

```
    array([[ 3.00000000e-01,  3.00000000e-01],
           [-1.89000000e-01, -3.51000000e-01],
           [ 9.68600430e-02,  1.93700403e-01],
           [-1.45374343e-02, -2.90748685e-02],
           [ 4.91567626e-05,  9.83135252e-05],
           [-1.90050844e-12, -3.80101689e-12]])
```

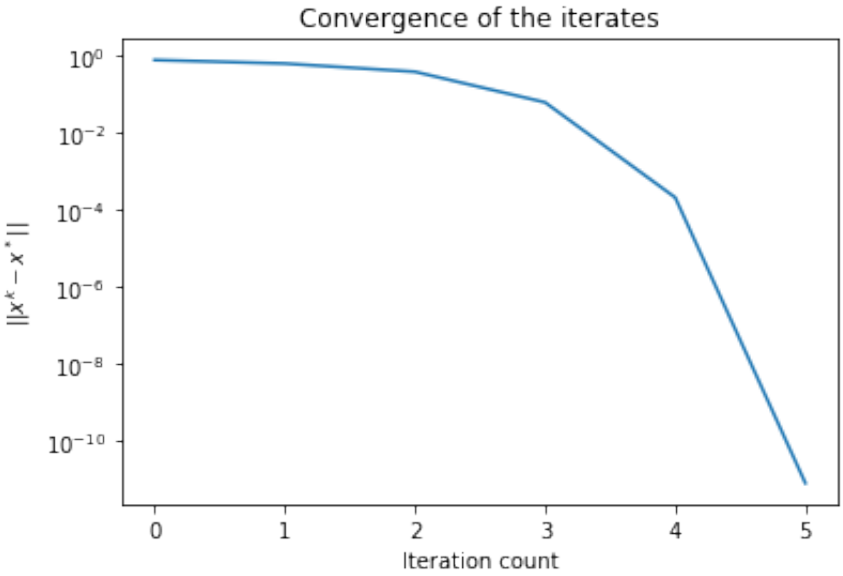
```
xstar=xf
nsteps = hist.shape[0]
fhist = []
for i in range(nsteps):
    fhist.append(la.norm(myf(hist[i,:])))
plt.figure('convergence')
plt.semilogy(np.arange(0, nsteps), (fhist))
plt.xlabel('Iteration count')
plt.ylabel(r'$||f^k - f^*||$')
plt.title('Convergence of the residuals')
plt.show()
```




```
fhist
```

```
    [0.7280677300184,
     0.5897022076342576,
     0.3612566054600514,
     0.058051672104732444,
     0.00019662704667453536,
     7.602033775698167e-12]
```

```
xstar=xf
nsteps = hist.shape[0]
normx = []
for i in range(nsteps):
    normx.append(la.norm((hist[i,:])))
plt.figure('convergence')
plt.semilogy(np.arange(0, nsteps), (fhist))
plt.xlabel('Iteration count')
plt.ylabel(r'$||x^k - x^*||$')
plt.title('Convergence of the iterates')
plt.show()
```



- $x_0 = [0.5, 0.5]^T$

```
x0=np.array([0.5,0.5])
xf,hist=newton(myf, J, x0, tol = 1e-6)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-12-38aab6bffbdf> in <module>
      1 x0=np.array([0.5,0.5])
----> 2 xf,hist=newton(myf, J, x0, tol = 1e-6)

<ipython-input-5-1a67cbf7e70c> in newton(f, J, x0, tol)
      3     history = np.array( [x0] )
      4     while ( la.norm(f(x)) > tol ):
----> 5         p = la.solve(J(x), - f(x))
      6         x += p
      7         history = np.vstack( (history, x) )

<__array_function__ internals> in solve(*args, **kwargs)

~\anaconda3\lib\site-packages\numpy\linalg\linalg.py in solve(a, b)
    391     signature = 'DD->D' if isComplexType(t) else 'dd->d'
    392     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 393     r = gufunc(a, b, signature=signature, extobj=extobj)
    394
    395     return wrap(r.astype(result_t, copy=False))

~\anaconda3\lib\site-packages\numpy\linalg\linalg.py in _raise_linalgerror_singular(err, flag)
     86
     87 def _raise_linalgerror_singular(err, flag):
---> 88     raise LinAlgError("Singular matrix")
     89
     90 def _raise_linalgerror_nonposdef(err, flag):
```

Globalize Newton's method

```
def m(x):
    return (0.5*np.dot(myf(x).T,myf(x)))
```

```
def backtrack_linesearch(f, gk, pk, xk, alpha = 0.1, beta = 0.8):
    t = 1
    while ( f(xk + t*pk) > f(xk) + alpha * t * gk @ pk ):
        t *= beta
    return t
```

```
def globalize_newton(f, J, x0, tol = 1e-6):
    x = x0
    history = np.array( [x0] )
    while ( la.norm(f(x)) > tol ):
        p = la.solve(J(x), - f(x))

        t = backtrack_linesearch(m, np.dot(J(x).T,myf(x)), p, x)
        x += t*p
        history = np.vstack( (history, x) )
    return x, history


x0=np.array([0.5,0.5])
xf,hist=globalize_newton(myf, J, x0, tol = 1e-6)


xf

    array([7.61010126e-10, 1.52202025e-09])


hist

    array([[ 5.00000000e-01,  5.00000000e-01],
           [-3.80000000e-01, -8.60000000e-01],
           [ 2.10902067e-01,  3.63173734e-01],
           [-1.20974314e-01, -2.41747085e-01],
           [ 2.82915488e-02,  5.65830975e-02],
           [-3.62318201e-04, -7.24636401e-04],
           [ 7.61010126e-10,  1.52202025e-09]])
```

```
xstar=xf
nsteps = hist.shape[0]
fhist = []
for i in range(nsteps):
    fhist.append(la.norm(myf(hist[i,:])))
plt.figure('convergence')
plt.semilogy(np.arange(0, nsteps), (fhist))
plt.xlabel('Iteration count')
plt.ylabel(r'$||f^k - f^*||$')
plt.show()
```