Numerical Optimization - Assignment 3

David Alvear(187594) - Nouf Farhoud(189731)

1. Gauss-Newton

1. Gauss-Newton. Nonlinear Least Squares (NLLS) problems arise in a large number of practical scientific and engineering contexts, and represent an important class of numerical optimization problems. NLLS problems commonly arise when trying to fit a model to a measured data set in a way that minimizes the discrepancy between model predictions and measured data. The simplest objective to minimize is the sum of the squares of the discrepancies at the measured data points.

Consider the NLLS problem of fitting a model with four parameters ($n = 4$) of the form

$$\phi(x;\theta) = \theta_1 e^{\theta_2 x}\cos(\theta_3 x + \theta_4)$$

to a given data set (x_i, y_i) of m points, such as the one shown below. Note that θ denotes the unknown vector here.

The Gauss-Newton method is an iterative optimization algorithm used to solve nonlinear least squares (NLLS) problems. The method approximates the Hessian matrix of the objective function using the Jacobian matrix, and then updates the parameter estimates using the Gauss-Newton update equation. The jacobian is the first derivative of a vector and the gradient is the first derivative of a scalar. Therefore, the jacobian of the gradient is the hessian matrix.

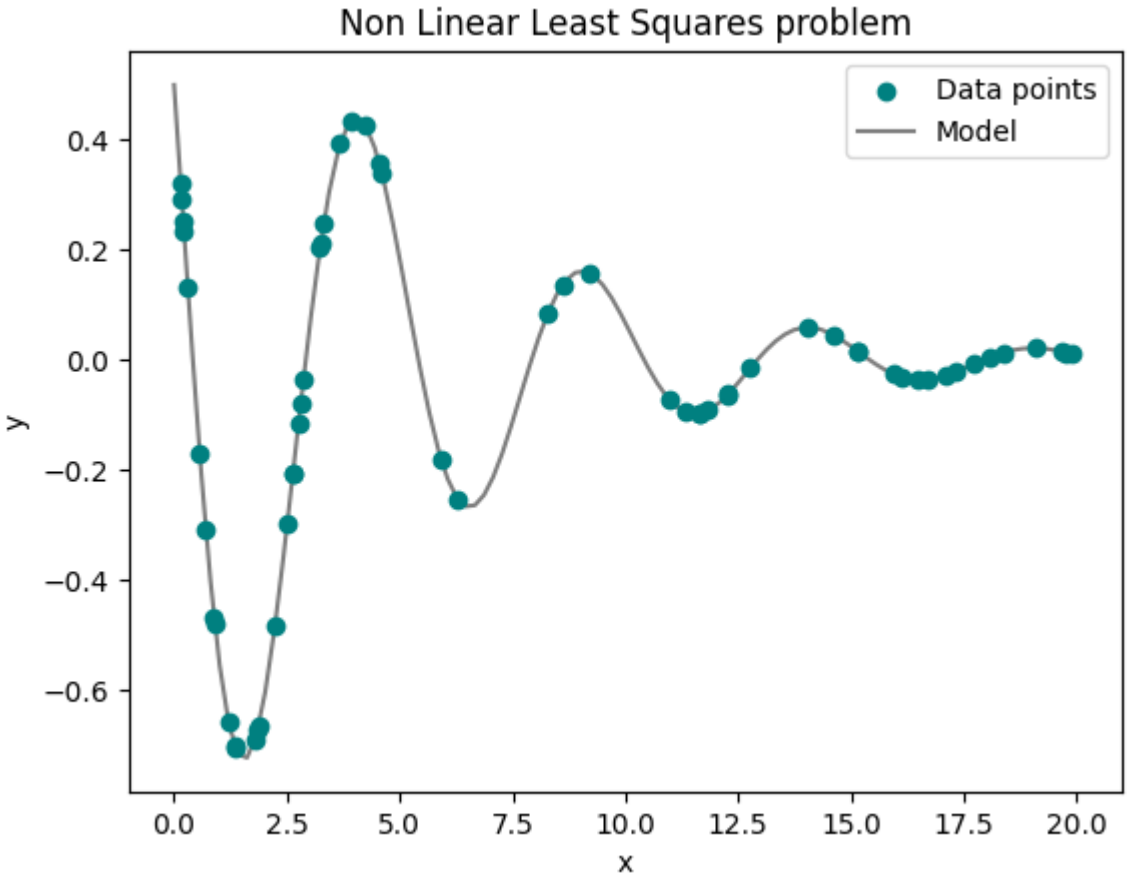
In [ ]:
```python
import numpy as np

def fit_data():
    # Load the data from the file
    data = np.load('data.npy')
    x = data[:, 0]  # Input data points
    y = data[:, 1]  # Output data points

    # Initial guess for the parameters
    theta0 = np.array([1.0, -0.1, 1.0, 1.0])

    # Gauss-Newton optimization
    max_iterations = 100  # Maximum number of iterations
    tolerance = 1e-6  # Tolerance for convergence

    theta = theta0
    iteration = 0
    converged = False

    while not converged and iteration < max_iterations:
        # Calculate the residual vector
        r = calculate_residual(theta, x, y)
```

```
        # Calculate the Jacobian matrix
        J = calculate_jacobian(theta, x)

        # Calculate the Gauss-Newton approximation of the Hessian matrix
        H = np.dot(J.T, J)

        # Calculate the gradient of the objective function
        gradient = np.dot(J.T, r)

        # Update the parameter estimates using the Gauss-Newton update
        delta_theta = -np.linalg.solve(H, gradient)
        theta = theta + delta_theta

        # Check for convergence
        if np.linalg.norm(delta_theta) < tolerance:
            converged = True

        iteration += 1

    # Calculate the final objective function value
    objective_value = calculate_objective(theta, x, y)

    # Display the results
    print('Optimization results:')
    print('Iterations:', iteration)
    print('Final parameter estimates:', theta)
    print('Final objective value:', objective_value)

    return theta, x, y

def calculate_residual(theta, x, y):
    m = len(x)
    r = np.zeros(m)

    for i in range(m):
        r[i] = theta[0] * np.exp(theta[1] * x[i]) * np.cos(theta[2] * x[i] +

    return r

def calculate_jacobian(theta, x):
    m = len(x)
    J = np.zeros((m, 4))

    for i in range(m):
        J[i, 0] = np.exp(theta[1] * x[i]) * np.cos(theta[2] * x[i] + theta[3
        J[i, 1] = theta[0] * x[i] * np.exp(theta[1] * x[i]) * np.cos(theta[2
        J[i, 2] = -theta[0] * x[i] * np.exp(theta[1] * x[i]) * np.sin(theta[
        J[i, 3] = -theta[0] * np.exp(theta[1] * x[i]) * np.sin(theta[2] * x[

    return J

def calculate_objective(theta, x, y):
    r = calculate_residual(theta, x, y)
    return 0.5 * np.dot(r, r)
```

```
# Call the fit_data function
theta, x, y = fit_data()
```

```
Optimization results:
Iterations: 7
Final parameter estimates: [ 1.         -0.2         1.25663706  1.04719755]
Final objective value: 1.0023119451853476e-26
```

In [ ]:
```
_x = np.linspace(0, 20, 100)
_y = theta[0]*np.exp(theta[1]*_x)*np.cos(theta[2]*_x+theta[3])

plt.scatter(x, y, zorder=8, c='teal',  label='Data points');
plt.plot(_x, _y, c='tab:gray', label='Model');
plt.title('Non Linear Least Squares problem');
plt.legend();
plt.xlabel('x');
plt.ylabel('y');
```

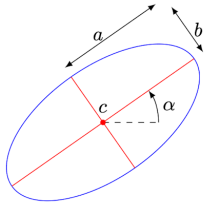


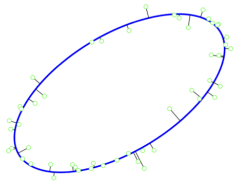
2. Ellipse fit

2. Ellipse fit. An ellipse in a plane can be described as the set of points

$$f(t,\theta) = \begin{bmatrix} c_1 + a\cos(\alpha)\cos(t) - b\sin(\alpha)\sin(t) \\ c_2 + a\sin(\alpha)\cos(t) + b\cos(\alpha)\sin(t) \end{bmatrix}$$

where t ranges from 0 to 2π. The vector $\theta = (c_1, c_2, a, b, \alpha)$ contains five parameters, with geometrical meanings illustrated in the figure below.





- We are interested in finding the distance of a point $x \in \mathbb{R}^2$ to an ellipse, which is defined as the distance to the nearest point on the ellipse. This can be written as the problem of minimizing the square distance $\|f(t,\theta) - x\|^2$ over the scalar t, with θ known. Formulate this as a nonlinear least squares problem and solve it to find the distance from the point $(2.5, 0)$ to the ellipse defined by the parameters $(0, 0, 2, 1, \pi/5)$.
- We consider the problem of fitting an ellipse to N points $x_1, \ldots, x_N$, in a plane, as shown in the figure on the right. The circles show the N points. The short lines connect each point to the nearest point on the ellipse. We will fit the ellipse by minimizing the sum of the squared distances of the N points to the ellipse. Formulate this as a nonlinear least squares problem. Give expressions for the Jacobian of the residual.
- Use a Gauss-Newton algorithm algorithm to fit an ellipse to the 10 points:

  $(0.5, 1.5)$, $(-0.3, 0.6)$, $(1.0, 1.8)$, $(-0.4, 0.2)$, $(0.2, 1.3)$, $(0.7, 0.1)$, $(2.3, 0.8)$, $(1.4, 0.5)$, $(0.0, 0.2)$, $(2.4, 1.7)$.

  To select a starting point, you can choose parameters θ^0 that describe a circle with radius one and center at the average of the data points, and initial values of t_i that minimize the objective function for these values of θ^0.

- Problem formulation to find the closest point from an ellipse to a determined x.

In [ ]:
```
import numpy as np
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
```

In [13]:
```
def model(t, theta):
  c1, c2, a, b, alpha = theta
  return np.array([c1 + a * np.cos(alpha) * np.cos(t) - b * np.sin(alpha) *
                   c2 + a * np.sin(alpha) * np.cos(t) + b * np.cos(alpha)

def objective_function(t, x, theta):
  return np.linalg.norm(model(t, theta) - x) # Distance to the ellips

def compute_jacobian_t(t, x, theta):
  c1, c2, a, b, alpha = theta
  return np.array([- a * np.cos(alpha) * np.sin(t) - b * np.sin(alpha) * np.
                   - a * np.sin(alpha) * np.sin(t) + b * np.cos(alpha) *

def gradient_function(jacobian, model, t, x, theta):
  J = jacobian(t, x, theta)
  residual = model(t, theta) - x
  return J.T * residual
```

```
def hessian_function(jacobian, t, x, theta):
  J = jacobian(t, x, theta)
  return J.T * J

def gauss_newton_solver_1D(t0, x, theta, tol=1e-5, max_iter=1000):
  # Optimize for t to get the distance to the ellipse
  history = [t0]
  t = t0
  iterations = 0
  p = 1
  while jnp.abs(p) > tol and iterations < max_iter:
    # 1. Determine update direction
    J = compute_jacobian_t(t, x, theta)
    residual = model(t, theta) - x
    p = - np.dot(J.T, residual) / np.dot(J.T, J)
    # 3. Update parameters
    t += p
    # logs
    iterations += 1
    history.append(t)

  # Return
  # print(f"Convergence in {iterations} iterations")
  return t, history
```

In [12]:
```
# Optimize for t
x = np.array([2.5, 0])
theta = (0, 0, 2, 1, np.pi/5)
t0 = 0.0
t, history = gauss_newton_solver_1D(t0, x, theta)
print(f"Distance to the ellipse: {t}")
```

```
Convergence in 22 iterations
Distance to the ellipse: -0.6989114364086619
```

- Formulate the problem to fit N points in an ellipse.

To formulate the distance minimization problem for a point (x) to an ellipse with known parameters (θ), we need to minimize the square distance ‖f(t, θ) - x‖^2 over the scalar t.

1) The distance from the point (x) to the ellipse can be defined as follows:

$$d = min \parallel f(t, \theta) - x \parallel^2$$

Substituting the equation of the ellipse f(t, θ) into the distance expression, we get:

$$d = min \parallel [c1 + acos(\alpha)cos(t) - bsin(\alpha)sin(t), c2 + asin(\alpha)cos(t) + bcos(\alpha)sin(t)$$

2) Expanding the square and simplifying, we have:

$$d = min[(c1 + acos(\alpha)cos(t) - bsin(\alpha)sin(t) - x1)^2 + (c2 + asin(\alpha)cos(t) + bcos($$

3) To solve this nonlinear least squares problem, we need to find the minimum of the objective function with respect to the parameter t. We can do this by taking the derivative of the objective function with respect to t, setting it equal to zero, and solving for t.

Differentiating the objective function with respect to t, we get:

$$d/dt[(c1 + acos(\alpha)cos(t) - bsin(\alpha)sin(t) - x1)^2 + (c2 + asin(\alpha)cos(t) + bcos(\alpha)si$$

4) To fit an ellipse to N points x1, x2, ..., xN by minimizing the sum of squared distances, we can define the objective function as follows:

$F(\theta) = \Sigma \parallel f(t, \theta) - xi\|^2$ θ represents the parameters of the ellipse xi represents each point in the set of N points

5) To calculate the Jacobian of the residual, we differentiate the objective function F(θ) with respect to the parameters θ. The Jacobian matrix J is given by:

$$J = (\partial F/\partial \theta)$$

6) For the Gauss-Newton algorithm, we start with an initial estimate θ0 and iteratively update it using the following update rule:

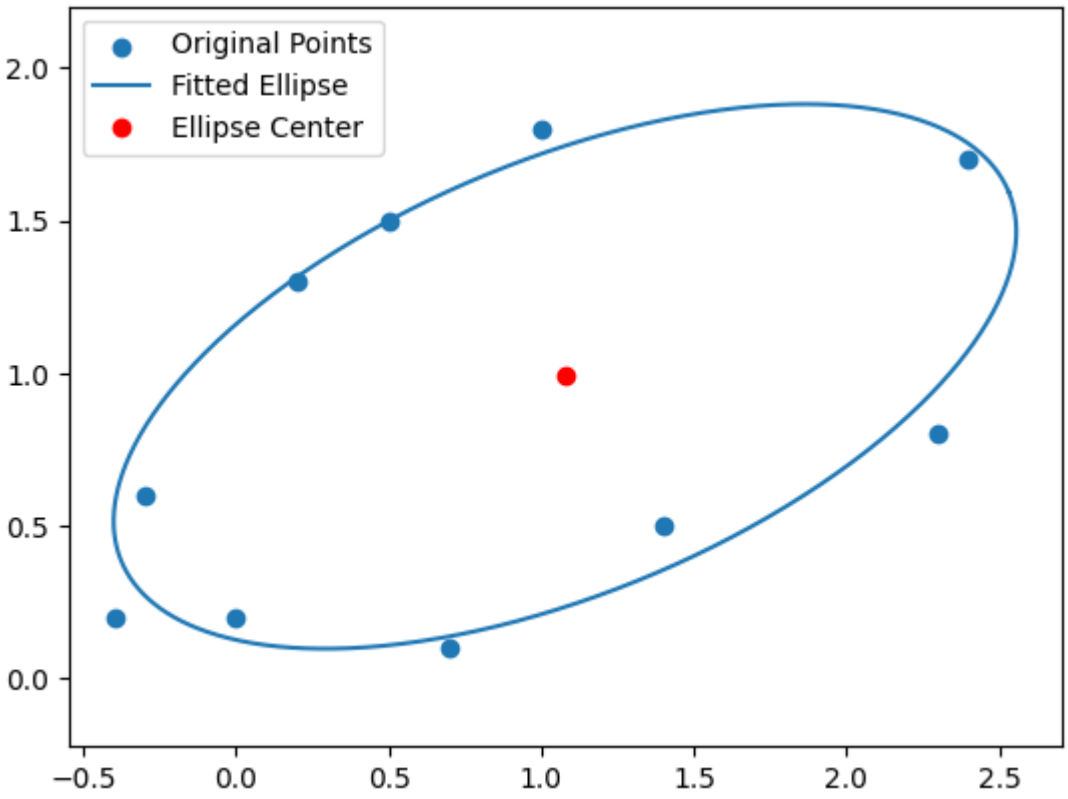
$$\theta(k+1) = \theta(k) - (J^T J)^{-1} J^T (F(\theta(k)))$$

In [14]:
```
def model(t, theta):
  c1, c2, a, b, alpha = theta
  return np.array([c1 + a * np.cos(alpha) * np.cos(t) - b * np.sin(alpha) *
                   c2 + a * np.sin(alpha) * np.cos(t) + b * np.cos(alpha)

def residuals_fn(x, theta):
  residuals = np.zeros((x.shape[0], 2))
  t_residuals = []
  for i, x_i in enumerate(x):
    t = np.linspace(0, 2*np.pi, 1000)
    distances = np.linalg.norm(model(t[:, np.newaxis], theta) - x_i, axis=1)
    t_min_index = np.argmin(distances)
    residuals[i] = model(t[t_min_index], theta) - x_i
    t_residuals.append(t[t_min_index])
  return residuals, t_residuals

def jacobian_ellipse(t, theta):
    c1, c2, a, b, alpha = theta
    J = np.array([[1, 0, np.cos(alpha) * np.cos(t), -np.sin(alpha) * np.sin(
                  [0, 1, np.sin(alpha) * np.cos(t), np.cos(alpha) * np.sin(t
    return J

def gauss_newton_solver(x, theta0, tol=1e-5, max_iter=1000):
  history = [theta0]
  theta = theta0
  iterations = 0
  t0 = 0
```

```
  p = np.ones_like(theta)
  while np.linalg.norm(p) > tol and iterations < max_iter:
    # Calulate the residual and the jacobian
    residual = np.zeros((x.shape[0], 2))
    jacobian = np.zeros((x.shape[0], 2, 5))
    grad_sum = np.zeros_like(theta0)
    hess_sum = np.zeros((len(theta0), len(theta0)))
    t_computed = np.zeros_like(x)
    for i in range(len(x)):
      # Find t for each point
      t, _ = gauss_newton_solver_1D(t0, x[i], theta)

      # Save the residual and the jacobian
      residual[i] = model(t, theta) - x[i]
      jacobian[i] = jacobian_ellipse(t, theta)
      t_computed[i] = t

      # Sum
      # Accumulate the gradient and Hessian
      grad_sum += jacobian[i].T @ residual[i]
      hess_sum += jacobian[i].T @ jacobian[i]

    # Compute gradient and Hessian
    r_reshaped = residual[:,:, np.newaxis]
    grad_val = np.matmul(jacobian.transpose(0,2,1), r_reshaped).squeeze(-1)
    hess_val = np.matmul(jacobian.transpose(0,2,1), jacobian) # Shape (10, 5

    # 1. Determine update direction
    # print(hess_val.shape)
    # print(grad_val.shape)
    p = np.linalg.solve(hess_sum, -grad_sum)

    # 3. Update parameters
    theta += p

    # logs
    iterations += 1
    history.append(theta)

  # Return
  print(f"Convergence in {iterations} iterations")
  return theta, t_computed, history


points = np.array([[0.5, 1.5], [-0.3, 0.6], [1.0, 1.8], [-0.4, 0.2], [0.2, 1
                   [0.7, 0.1], [2.3, 0.8], [1.4, 0.5], [0.0, 0.2], [2.4, 1.7

c1, c2 = np.mean(points, axis=0)
a = b = 2
alpha = 1
theta0 = np.array([c1, c2, a, b, alpha])

# Solve the problem with gauss-newton method
theta, t_vals, hist = gauss_newton_solver(points, theta0)
```

```
Convergence in 56 iterations
```

In [15]:
```
# Now, let's plot the result
t_values = np.linspace(0, 2 * np.pi, 1000)

ellipse_points = model(t_values, theta)

# Plot the original points
plt.scatter(points[:, 0], points[:, 1], label='Original Points')

# Plot the ellipse
plt.plot(ellipse_points[0], ellipse_points[1], label='Fitted Ellipse')

# Mark the center of the ellipse
plt.plot(theta[0], theta[1], 'ro', label='Ellipse Center')

plt.axis('equal')  # Equal scaling for both axes
plt.legend()
plt.show()
```



3. Cross-well tomography.

3. Cross-well tomography. Inverse problems are problems where we seek to measure the parameters of a physical system from measurements of its response to one or more inputs. This is in contrast to forward problems where, given a physical system, we seek to determine its response to a given input.

Consider, as an example, the following cross-well tomography problem used in petroleum exploration. Two vertical wells are located 1600 m apart. A seismic source is inserted in one of the wells at depths of $50, 150, \cdots, 1550$m. A string of receivers is inserted in the other well at depths of $50, 150, \cdots, 1550$m as shown in the figure below. For each source-receiver pair a travel time is recorded (such measurements, while accurate, have an error on the order of 0.5 ms). There are 256 ray paths and 256 corresponding measurements. We wish to determine the velocity structure in the two-dimensional plane between the two wells. Discretizing the problem into a 16×16 grid ($100m \times 100m$ blocks) gives 256 model parameters. The coefficient matrix and noisy measurement data for a problem are provided in data files.
Use `d = np.load('d.npy')` and `G = np.load('G.npy')` to load the d vector and G matrix respectively.

- Confirm that the coefficient matrix is ill-conditioned and hence cannot be inverted to find the solution of $Gx = d$.
- Formulate the problem as a (linear) least squares problem, and confirm that this formulation cannot recover the problem parameters.
- Formulate the problem as a regularized linear least squares problem, that minimizes the sum of the misfit and a regularization term

$$\underset{x}{\text{minimize}} \quad \tfrac{1}{2}\|Gx - d\|^2 + \tfrac{\alpha}{2}\|x\|^2$$

  for various values of alpha $10^{-6} \leq \alpha \leq 10^6$. What is the best α value to choose? Show the corresponding solution.
- Another regularization strategy used with ill-conditioned least squares problems relies on $A = U\Sigma V^t$, the singular value decomposition of the coefficient matrix to avoid computing $(A^tA)\backslash(A^tb)$. Derive the optimal solution as $x^* = \sum_{i=1}^{r} \frac{u_i^t b}{\sigma_i} v_i$, and use an appropriate r for solving the problem.

In [ ]:
```
import numpy as np

# Load the Coefficient matrix G and the noisy measurements vector d
d = np.load('d.npy')
G = np.load('G.npy')
```

- Determine that the coefficient matrix is ill-conditioned and hence cannot be inverted to find the solution $Gx = d$.

In [ ]:
```
condition_number = np.linalg.cond(G)
print(f"The condition number for G is: {condition_number}")
```

```
The condition number for G is: 8.439917434162691e+18
```

AS we can see, the matrix G is ill-conditioned so it cannot be inverted to find the solution Gx = d. So, performing computation with this matrix will be highly sensitive to errors and producing numerical innestability.

- Formulate the problem as a (linear) least squares problem, and confirm that this formulation cannot recover the problem parameters.

  Minimize $\frac{1}{2}\|Gx - d\|_2^2$

  Let $f(x) = \frac{1}{2}x^TG^TGx - d^TGx + \frac{1}{2}d^Td$

  Minimize $\nabla f(x) = 0$

  $\nabla f(x) = G^TGx - G^Td = 0$

$$G^T G x = G^T d$$

The solution is reduced to the result of the Linear System

$$\nabla^2 f(x) = G^T G$$

Solution if $\forall x, \quad x^T G^T G x > 0$

This is possible if the columns of G are linearly independent.

The issue with this problem formulation is that the matrix multiplication $G^T G$ square the condition number making the problem worse. The matrix G already has a large condition number.

In [ ]:
```
# Evaluate if the matrix G has linearly independent columns
rank_G = np.linalg.matrix_rank(G)
print(f"The rank of matrix G is: {rank_G}")
```

```
The rank of matrix G is: 243
```

Given that the rank of matrix G is 243 implies that there are some columns of G that are linearly dependent (G rank deficient). Then, the condition of $\forall x, \quad x^T G^T G x > 0$ for the hessian is not satisfied to find the solution of the least squares problem. The solution of the problem with this matrix could not have an unique solution or it could have infinity solutions because there are more unknowns than independent equations.

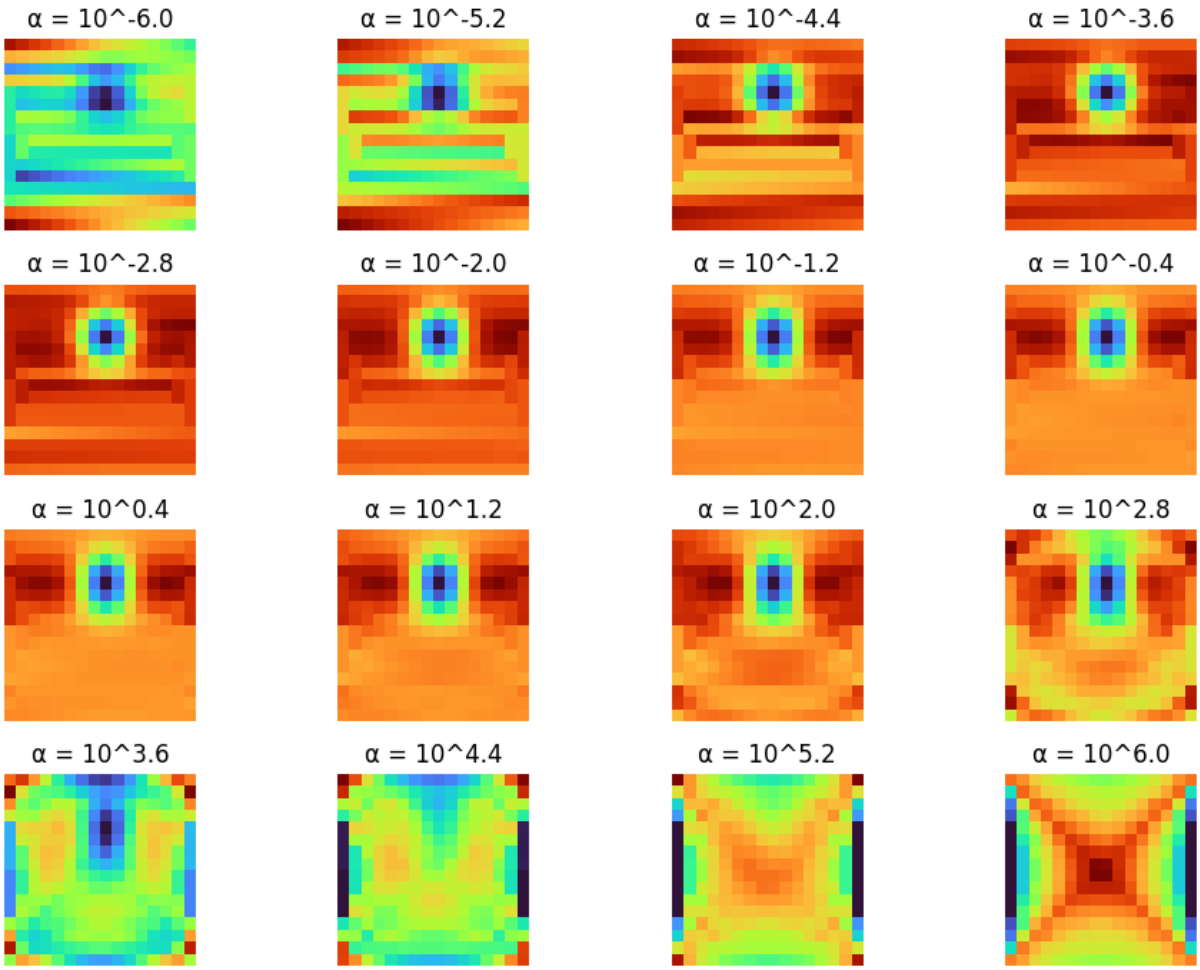
- Formulate the problem as a regularized linear least squares problem, that minimizes the sum of the misfit and a regularization term.

  $$Minimize_x \ \frac{1}{2}\|Gx - d\|_2^2 + \frac{\alpha}{2}\|x\|^2$$

  For this problem formulation we find the solution for the parameters using a regularization term that helps with the ill-conditined least squares problem.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

############################################################
#################### Objective Function ####################
def objective_fun_reguarization(G, d, alpha=0.001):
  # f = 0.5 * np.linalg.norm((np.dot(G, x) - d))**2 + np.linalg.norm(x)**2
  return np.linalg.solve((G.T @ G + 10**(alpha) * np.eye(G.shape[0])), np.dc

alphas = np.linspace(-6, 6, 16)

fig, axs = plt.subplots(4, 4, sharex=True, sharey=True, figsize=(10, 7))

for alpha, ax in zip(alphas, axs.flatten()):
  solution = objective_fun_reguarization(G, d, alpha).reshape((16, 16))
```

```
  ax.imshow(solution, cmap='turbo')
  ax.set_title(f'α = 10^{alpha:.3}')
  ax.grid(False)
  ax.axis('off')

plt.tight_layout()

# Show the plot
plt.show()
```



For the implemented solution of the least squares with the regularization we can see that the solution has a variety of solutions depending of the regularization hyperparameter. For this case we can conclude that the best alpha value for the solution is between the range $10^{-1.2} < \alpha < 10^{1.2}$.

- Another regularization strategy used with ill-conditioned least squares problems relies on $A = U\Sigma V^t$, the singular value decomposition of the coefficient matrix. Derive the optimal solution x^* and select the apropiate value of r.

Derive the optimal solution of SVD

We have $G = U\Sigma V^T$

We have that $G^T G x = G^T d$

Then: $V \Sigma U^T U \Sigma V^T x = V \Sigma U^T d$

$\Sigma^2 V^T x = \Sigma U^T d$

$\Rightarrow V^T x = \Sigma^{-1} U^T d$

$\Rightarrow V V^T x = V \Sigma^{-1} U^T d$

$\Rightarrow x^* = V \Sigma^{-1} U^T d = \sum_{i=1}^{n} \frac{u_i^T d}{\sigma_i} v_i$

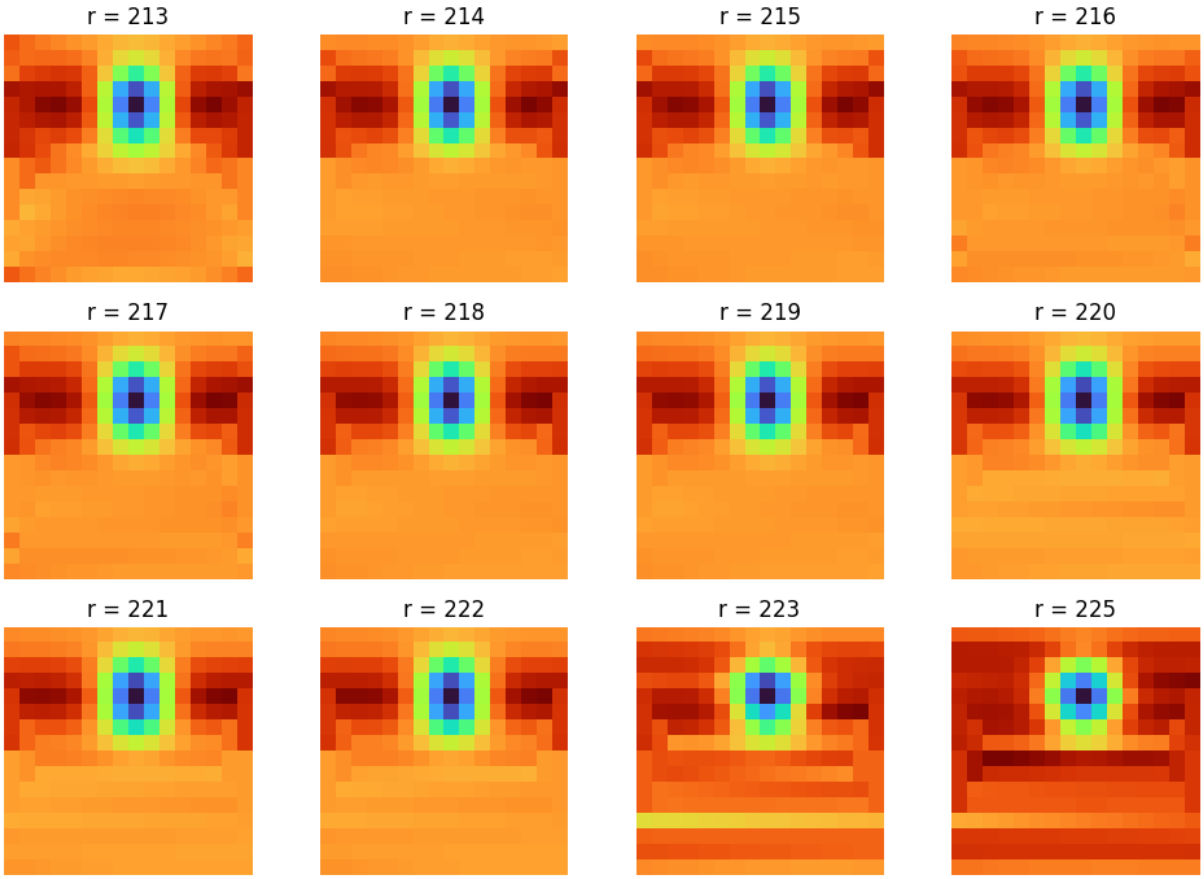
For this case we need to truncate the sum to avoid the small singular values near zero due to G is rank deficient.

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

In [ ]:
```python
# SVD of G
U, S, Vt = np.linalg.svd(G, full_matrices=False)

def truncate_SVD(U, S, Vt, tol=10e-5, points=12):
  # This function takes the S matrix and produce the values to truncate r
  max_tolerance = S.max() * tol
  rank = (S > max_tolerance).sum()
  r = np.linspace(rank-(points/2), rank + (points/2), points)
  return r

def solve_svd(d, U, S, Vt, truncate_rank):
  # This method solve the LLS using a truncate value with SVD
  U_trunc = U[:,:truncate_rank]
  Vt_trunc = Vt[:truncate_rank,:]
  S_inv = np.diag(1 / S[:truncate_rank])

  # Solve
  x = Vt_trunc.T @ S_inv @ U_trunc.T @ d
  return x

def solve_LLS_with_SVD(G, d, tol=10e-5, truncate=True, points=12):
  # This method solve to find G with multiple truncation values
  U, S, Vt = np.linalg.svd(G, full_matrices=False)
  r = [np.linalg.matrix_rank(G)]
  if truncate:
    r = truncate_SVD(U, S, Vt, tol=tol, points=points)

  solutions = []
  for i, r_val in enumerate(r):
    x = solve_svd(d, U, S, Vt, int(r_val))
    solutions.append(x)
  return solutions, r
```

In [ ]:
```python
# Create the optimal solution avoiding the small singular values
solutions, truncate_r = solve_LLS_with_SVD(G, d, points=12)
```

```
fig, axs = plt.subplots(3, 4, sharex=True, sharey=True, figsize=(10, 7))

for r, ax, solution in zip(truncate_r, axs.flatten(), solutions):
  ax.imshow(solution.reshape(16,16), cmap='turbo')# Plot the solution
  ax.set_title(f'r = {int(r)}')
  ax.grid(False)
  ax.axis('off')

plt.tight_layout()

# Show the plot
plt.show()
```



Using the Singular Value Decomposition we approximated to the optimal solution using different truncation values. This values were computed using the maximum singular value and maintaining the values that are close to this maximum. In the results we can see that we have a good approximation truncating the S, V, U matrices from the value 214 to 220.