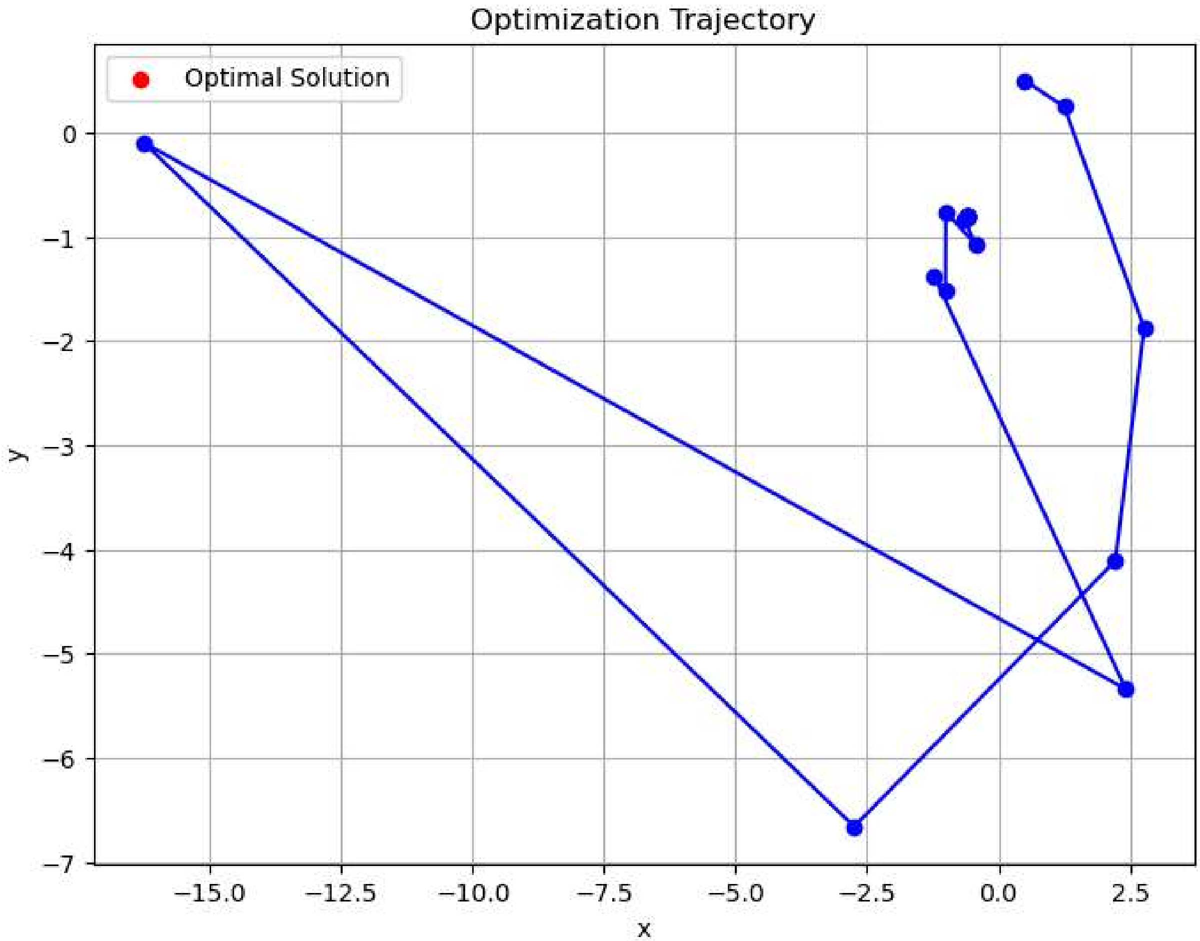
PROBLEM 1

PART 1

In [ ]:

In [37]:
```
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt

# Define the objective function
def objective_function(x):
    return 3*x[0] + 4*x[1]

# Define the equality constraint
def equality_constraint(x):
    return x[0]**2 + x[1]**2 - 1

# Initial guess for x
x0 = np.array([0.5, 0.5])

# Define the optimization problem
def optimization_callback(x):
    intermediate_solutions.append(x)

intermediate_solutions = [x0]  # Store the intermediate solutions
optimization_result = minimize(objective_function, x0, constraints={'type': 'eq', '

# Extract the optimal solution
x_opt = optimization_result.x

# Plot the trajectory of optimization
intermediate_solutions = np.array(intermediate_solutions)
plt.figure(figsize=(8, 6))
plt.plot(intermediate_solutions[:, 0], intermediate_solutions[:, 1], marker='o', li
plt.scatter(x_opt[0], x_opt[1], color='r', label='Optimal Solution')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Optimization Trajectory')
plt.legend()
plt.grid(True)
plt.show()
```



In [ ]:

PART 2

In [39]:

```
import numpy as np
from scipy.optimize import minimize

# Define the objective function
def objective_function(x):
    return x[1]**2 + x[1]**2 + x[2]**2

# Define the equality constraints
def equality_constraints(x):
    return [x[0]**2 + x[1]**2 - 1, x[0] + x[1] + x[2] - 1]

# Initial guess for x
x0 = np.array([0.5, 0.5, 0.5])

# Define the optimization problem
optimization_result = minimize(objective_function, x0, constraints={'type': 'eq', '

# Extract the optimal solution
x_opt = optimization_result.x

# Print the optimal solution
```

```
print("Optimal solution:")
print("x =", x_opt)
```

```
Optimal solution:
x = [ 1.00000000e+00  1.55756467e-09 -1.61056572e-09]
```

In [ ]:

PART 3

In [26]:

```
import numpy as np

# Define the objective function
def objective(x):
    return x[0]*x[1] + x[1]*x[2]

# Define the equality constraints
def constraint(x):
    return np.array([x[0]**2 + x[1]**2 - 2, x[0]**2 + x[2]**2 - 2])

# Define the gradient of the Lagrangian function
def lagrangian_gradient(x, lambd):
    grad_f = np.array([x[1], x[0] + x[2], x[1]])
    grad_h = np.array([[2*x[0], 2*x[1], 0], [2*x[0], 0, 2*x[2]]])
    return grad_f + np.dot(grad_h.T, lambd)

# Gradient method to solve the system of equations r(z) = 0
def gradient_method(initial_guess, tol=1e-6, max_iter=1000, lr=0.01):
    x = initial_guess[:3]
    lambd = np.zeros_like(constraint(x))  # Initialize lambda to zeros
    for _ in range(max_iter):
        # Compute the gradient of the residual function
        grad_r = lagrangian_gradient(x, lambd)

        # Update x using gradient descent
        x -= lr * grad_r[:3]

        # Update lambda using the method of multipliers
        lambd -= lr * constraint(x)

        # Check convergence
        if np.linalg.norm(constraint(x)) < tol:
            break

    return x, lambd

# Initial guess for x
initial_guess = np.array([1.0, 1.0, 1.0])

# Solve the optimization problem
solution, lambd = gradient_method(np.concatenate([initial_guess, np.zeros_like(cons

print("Optimized solution (x):", solution)
print("Optimized lambda:", lambd)
print("Objective value:", objective(solution))
```

```
Optimized solution (x): [ 1.52760135e-91 -6.05696384e-90  7.77288855e-90]
Optimized lambda: [19.46738547 19.20232668]
Objective value: -4.800536748824486e-179
```

In [ ]:

PART 4

In [40]:

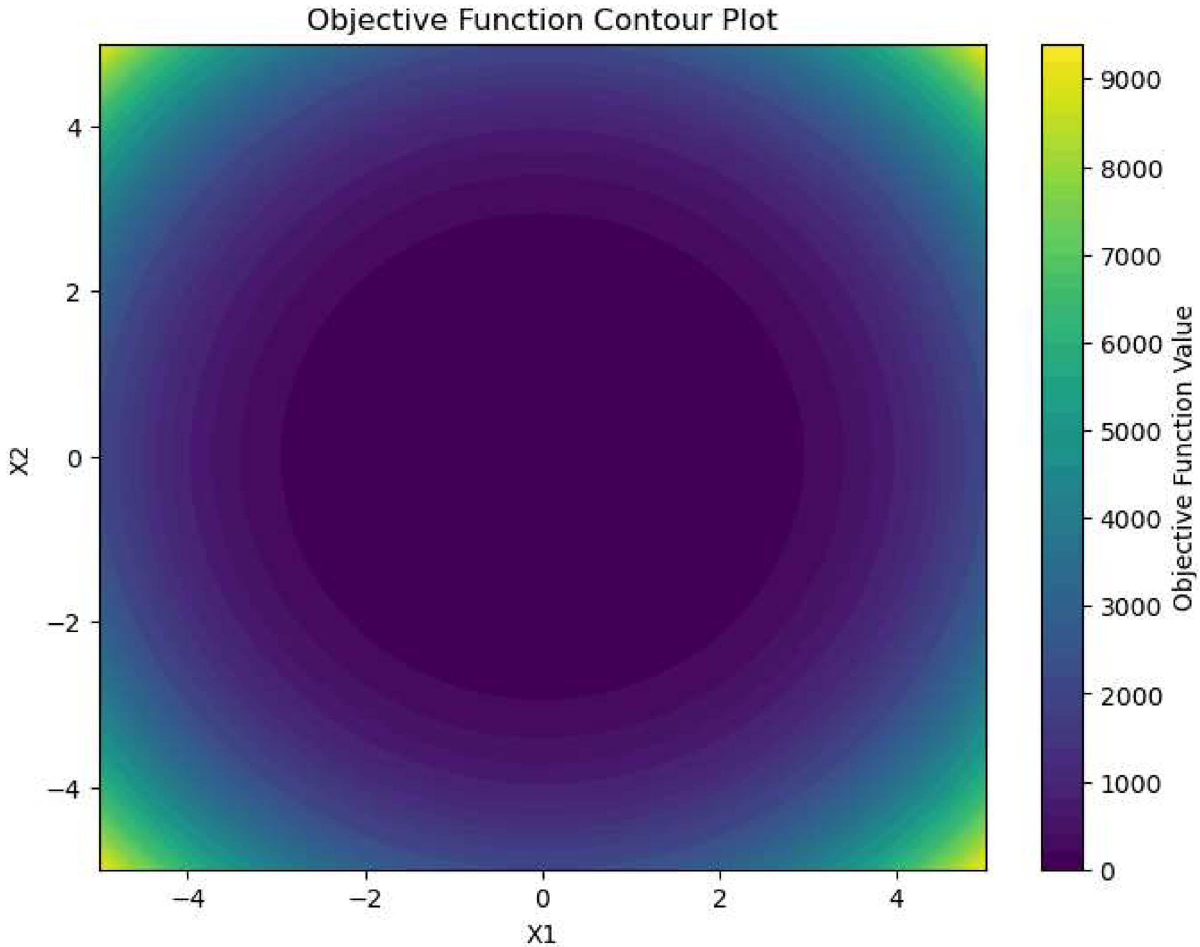
```
# Define the objective function
def objective(x, y):
    return x**2 + 2*y**2

# Define the constraint function
def constraint(x, y):
    return x + y - 2

# Calculate Lagrange multiplier
def lagrange_multiplier(x, y):
    return -(8/3)

# Calculate the solution
def solve_optimization_problem():
    # Calculate x and y using Lagrange multiplier
    x = 4/3
    y = 2/3

    # Verify constraint
    constraint_satisfied = constraint(x, y) == 0

    # Calculate Lagrange multiplier
    lambda_val = lagrange_multiplier(x, y)

    # Calculate objective function value
    obj_value = objective(x, y)

    return x, y, lambda_val, obj_value, constraint_satisfied

# Solve the optimization problem
x, y, lambda_val, obj_value, constraint_satisfied = solve_optimization_problem()

# Print the results
print("Optimized solution (x):", x)
print("Optimized solution (y):", y)
print("Lagrange multiplier:", lambda_val)
print("Objective value:", obj_value)
print("Constraint satisfied:", constraint_satisfied)
```

```
Optimized solution (x): 1.3333333333333333
Optimized solution (y): 0.6666666666666666
Lagrange multiplier: -2.6666666666666665
Objective value: 2.6666666666666665
Constraint satisfied: True
```

In [ ]:

PROBLEM 2

In [41]:
```
import numpy as np
import matplotlib.pyplot as plt

# Define the symmetric matrix A
A = np.array([[2, 1],
              [1, 3]])

# Define the objective function
def objective_function(X):
    return np.linalg.norm(A - np.dot(X, X.T)) ** 2

# Generate a grid of X values
n_points = 100
x_values = np.linspace(-5, 5, n_points)
y_values = np.linspace(-5, 5, n_points)
X1, X2 = np.meshgrid(x_values, y_values)
X = np.stack((X1, X2), axis=-1)

# Compute the objective function for each point in the grid
Z = np.zeros_like(X1)
for i in range(n_points):
    for j in range(n_points):
        Z[i, j] = objective_function(X[i, j])

# Plot the objective function
plt.figure(figsize=(8, 6))
plt.contourf(X1, X2, Z, levels=50, cmap='viridis')
plt.colorbar(label='Objective Function Value')
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Objective Function Contour Plot')
plt.show()
```



PART 3

In [42]:
```
import numpy as np

# Function to compute the objective function
def objective_function(X, A):
    B = A - np.dot(X, X.T)
    return np.trace(np.dot(B, B.T))

# Function to compute the analytical gradient
def analytical_gradient(X, A):
    B = A - np.dot(X, X.T)
    dB_dX = -2 * np.einsum('ij,kl->ijkl', X, X)  # Compute the Jacobian
    df_dX = np.einsum('ij,ijkl->kl', B, dB_dX)   # Compute the gradient
    return df_dX

# Function to compute the finite difference approximation of the gradient
def finite_difference_gradient(X, A, epsilon=1e-6):
    n = X.shape[0]
    df_dX_fd = np.zeros_like(X)
    for i in range(n):
        for j in range(n):
            X_plus = X.copy()
            X_plus[i, j] += epsilon
            X_minus = X.copy()
```

```
            X_minus[i, j] -= epsilon
            df_dX_fd[i, j] = (objective_function(X_plus, A) - objective_function(X_
    return df_dX_fd

# Generate a random symmetric positive definite matrix A
n = 3
A = np.random.rand(n, n)
A = np.dot(A, A.T)

# Generate a random initial guess for X
X = np.random.rand(n, n)

# Compute the analytical gradient
analytical_grad = analytical_gradient(X, A)

# Compute the finite difference approximation of the gradient
fd_grad = finite_difference_gradient(X, A)

# Compare the gradients
print("Analytical Gradient:\n", analytical_grad)
print("\nFinite Difference Gradient:\n", fd_grad)
print("\nDifference:\n", analytical_grad - fd_grad)
```

```
Analytical Gradient:
 [[-0.01558874 -0.52016468 -0.31487335]
 [-0.43152247 -0.57592002 -0.1451073 ]
 [-0.54948873 -0.12364644 -0.0268066 ]]

Finite Difference Gradient:
 [[-1.73702671 -1.83644105 -0.97731892]
 [ 1.02968921  1.43116578  0.37579261]
 [ 0.41051126 -1.54455477 -0.97081403]]

Difference:
 [[ 1.72143796  1.31627637  0.66244558]
 [-1.46121169 -2.0070858  -0.52089992]
 [-0.95999998  1.42090833  0.94400743]]
```

In [ ]:

PART 4

In [44]:

```
import numpy as np

def objective_function(X, A):
    B = A - np.dot(X, X.T)
    return np.trace(np.dot(B, B.T))

def gradient_descent(A, d, lr=0.01, max_iter=1000, tol=1e-6):
    n = A.shape[0]
    X = np.random.rand(n, d)
    iter_count = 0
    prev_loss = float('inf')

    while iter_count < max_iter:
```

```
        B = A - np.dot(X, X.T)
        grad = -4 * np.dot(B, X)
        X -= lr * grad
        loss = objective_function(X, A)
        if abs(prev_loss - loss) < tol:
            break
        prev_loss = loss
        iter_count += 1

    return X, loss, iter_count

# Example usage
n = 5  # Size of matrix A
d = 2  # Dimension of X
A = np.random.rand(n, n)
A = np.dot(A, A.T)  # Generating a symmetric positive definite matrix

# Solve using gradient descent
X_gd, loss_gd, iterations_gd = gradient_descent(A, d)

# Compare with spectral decomposition solution
eigvals, eigvecs = np.linalg.eigh(A)
X_spectral = eigvecs[:, -d:]

# Compare objective function values
loss_spectral = objective_function(X_spectral, A)

print("Gradient Descent Solution:")
print("X:\n", X_gd)
print("Objective Function Value:", loss_gd)
print("Iterations:", iterations_gd)
print("\nSpectral Decomposition Solution:")
print("X:\n", X_spectral)
print("Objective Function Value:", loss_spectral)
```

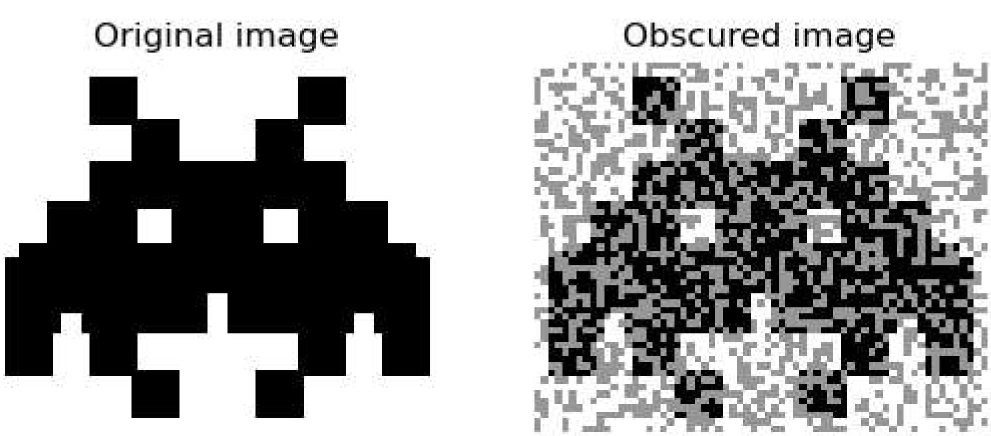
```
Gradient Descent Solution:
X:
 [[ 0.27584495  1.02800334]
 [ 1.15636933  0.37572636]
 [ 0.88870228 -0.00608153]
 [ 0.90558166  0.44780377]
 [ 0.42244643  0.36198595]]
Objective Function Value: 0.02547422782621354
Iterations: 188

Spectral Decomposition Solution:
X:
 [[-0.84252461 -0.37061696]
 [ 0.23989852 -0.60671329]
 [ 0.46343791 -0.39530181]
 [ 0.03688606 -0.51209854]
 [-0.12831873 -0.27574193]]
Objective Function Value: 8.384267051924887
```

In [ ]:

PROBLEM 3

PART 1

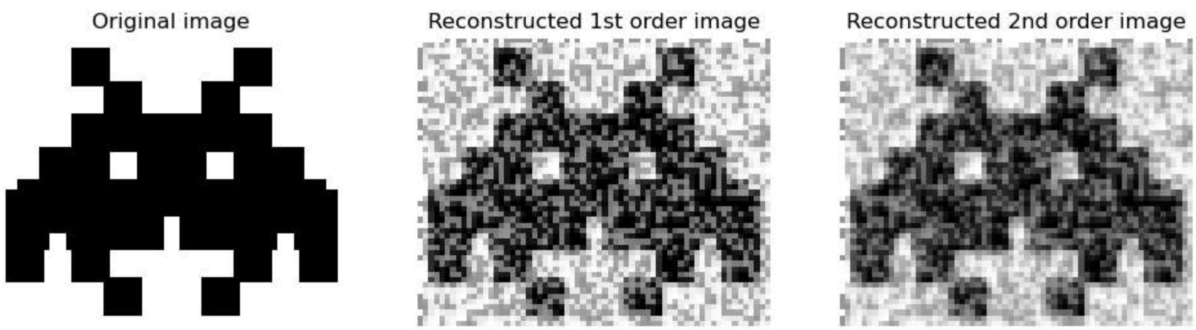
In [54]:
```
import numpy as np
import matplotlib.pyplot as plt

# Load the image
U0 = plt.imread('space.bmp').astype(float)

# Get the dimensions of the image
m, n = U0.shape

# Set the random seed for reproducibility
np.random.seed(666)

# Generate a random mask to obscure some pixels
unk = np.random.rand(m, n) < 0.4

# Apply the mask to obscure the pixels
U1 = U0 * (1 - unk) + 150 * unk

# Display the original and obscured images
plt.figure()
plt.subplot(1, 2, 1)
plt.imshow(U0, cmap='gray')
plt.title('Original image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(U1, cmap='gray')
plt.title('Obscured image')
plt.axis('off')
plt.show()
```



In [77]:
```
import numpy as np

def l2_loss(U):
    m, n = U.shape
```

```
    loss = 0
    for i in range(1, m):
        for j in range(1, n):
            loss += (U[i, j] - U[i-1, j])**2 + (U[i, j] - U[i, j-1])**2
    return loss

def gradient_l2_loss(U):
    m, n = U.shape
    gradient = np.zeros_like(U)
    for i in range(1, m):
        for j in range(1, n):
            gradient[i, j] += 2 * (U[i, j] - U[i-1, j]) + 2 * (U[i, j] - U[i, j-1])
    return gradient

def first_order_descent_l2(U, alpha, num_iterations):
    for _ in range(num_iterations):
        grad = gradient_l2_loss(U)
        U -= alpha * grad
    return U

def second_order_descent_l2(U, alpha, num_iterations):
    for _ in range(num_iterations):
        grad = gradient_l2_loss(U)
        U -= alpha * (grad + np.eye(U.shape[0], U.shape[1]) * 1)  # Approximating H
    return U
# Example usage:
# U1 is the obscured image matrix
# alpha is the learning rate
# num_iterations is the number of iterations for gradient descent
U0 = plt.imread('space.bmp').astype(float)
U1 = U0 * (1 - unk) + 150 * unk

# Convert U1 to double if necessary
U1 = U1.astype(float)

# Set hyperparameters
alpha = 0.001
num_iterations = 100

# Perform first-order descent with L2 loss
U_reconstructed = first_order_descent_l2(U1, alpha, num_iterations)
U_reconstructed_second_order = second_order_descent_l2(U1.copy(), alpha, num_iterat
# Print the reconstructed image
# print("Reconstructed image:")
# print(U_reconstructed)


# Display the original, first-order, and second-order reconstructed images
plt.figure(figsize=(12, 4))

# Original image
plt.subplot(1, 3, 1)
plt.imshow(U0, cmap='gray')
plt.title('Original image')
plt.axis('off')
```

```
# First-order reconstructed image
plt.subplot(1, 3, 2)
plt.imshow(U_reconstructed, cmap='gray')
plt.title('Reconstructed 1st order image')
plt.axis('off')

# Second-order reconstructed image
plt.subplot(1, 3, 3)
plt.imshow(U_reconstructed_second_order, cmap='gray')
plt.title('Reconstructed 2nd order image')
plt.axis('off')

plt.show()
```

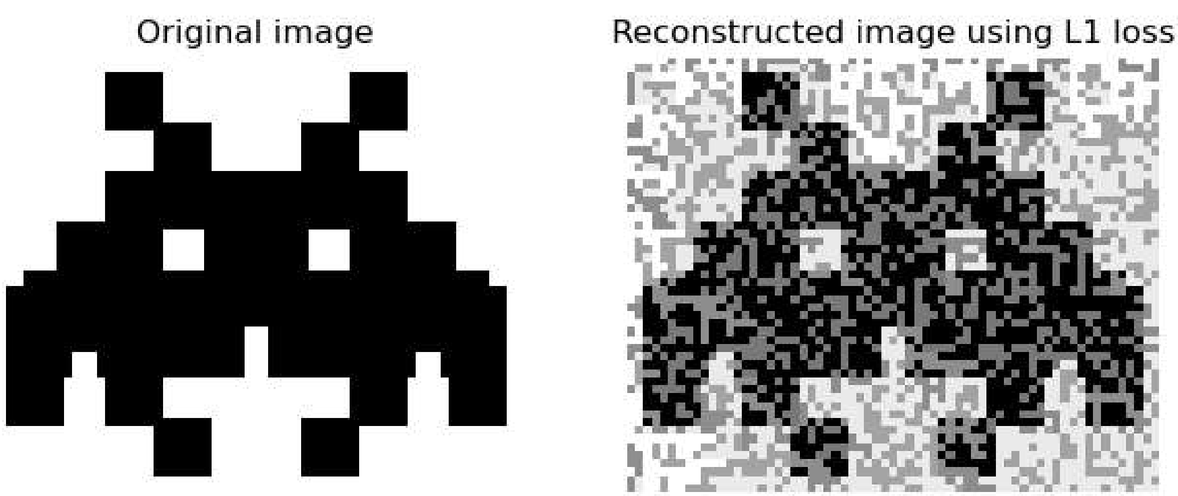


PART 2

In [80]:
```
def l1_loss(U):
    m, n = U.shape
    loss = 0
    for i in range(1, m):
        for j in range(1, n):
            loss += np.abs(U[i, j] - U[i-1, j]) + np.abs(U[i, j] - U[i, j-1])
    return loss

def gradient_l1_loss(U):
    m, n = U.shape
    gradient = np.zeros_like(U)
    for i in range(1, m):
        for j in range(1, n):
            gradient[i, j] += np.sign(U[i, j] - U[i-1, j]) + np.sign(U[i, j] - U[i,
    return gradient

def gradient_descent_l1(U, alpha, num_iterations):
    for _ in range(num_iterations):
        grad = gradient_l1_loss(U)
        U -= alpha * grad
    return U

# Example usage:
# U1 is the obscured image matrix
# alpha is the learning rate
# num_iterations is the number of iterations for gradient descent
U0 = plt.imread('space.bmp').astype(float)
U1 = U0 * (1 - unk) + 150 * unk
```

```
# Convert U1 to double if necessary
U1 = U1.astype(float)

# Set hyperparameters
alpha = 0.01
num_iterations = 1000

# Perform gradient descent with L1 loss
U_reconstructed_l1 = gradient_descent_l1(U1.copy(), alpha, num_iterations)
plt.figure(figsize=(12, 4))

# Print the reconstructed image
# Original image
plt.subplot(1, 3, 1)
plt.imshow(U0, cmap='gray')
plt.title('Original image')
plt.axis('off')

# First-order reconstructed image
plt.subplot(1, 3, 2)
plt.imshow(U_reconstructed_l1, cmap='gray')
plt.title('Reconstructed image using L1 loss')
plt.axis('off')
```

Out[80]: (-0.5, 64.5, 52.5, -0.5)

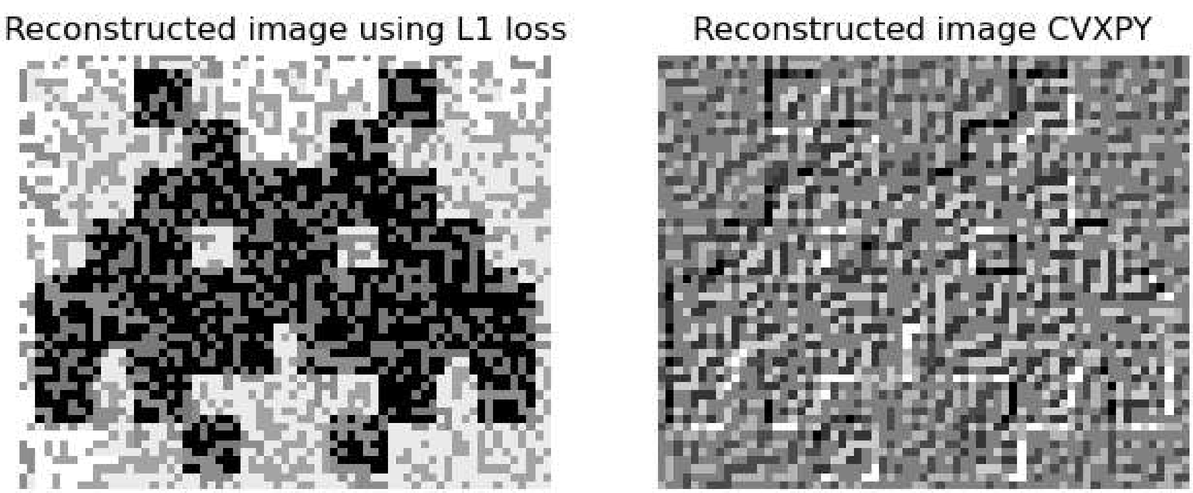


PART 3

In [87]:
```
import cvxpy as cp

# Define variables
grad_x = cp.Variable((U1.shape[0], U1.shape[1]-1))
grad_y = cp.Variable((U1.shape[0]-1, U1.shape[1]))

# Define objective function
obj = cp.sum(cp.abs(grad_x)) + cp.sum(cp.abs(grad_y))

# Define constraints
constraints = [
    grad_x == U1[:, 1:] - U1[:, :-1],
```

```
    grad_y == U1[1:, :] - U1[:-1, :]
]

# Define optimization problem
problem = cp.Problem(cp.Minimize(obj), constraints)

# Solve the problem
problem.solve()

# Reconstruct the image from gradients
U_reconstructed_cvx = np.zeros_like(U1)
U_reconstructed_cvx[:, :-1] += grad_x.value
U_reconstructed_cvx[1:, :] += grad_y.value

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
plt.imshow(U_reconstructed_l1, cmap='gray')
plt.title('Reconstructed image using L1 loss')
plt.axis('off')

# First-order reconstructed image
plt.subplot(1, 3, 2)
plt.imshow(U_reconstructed_cvx, cmap='gray')
plt.title('Reconstructed image CVXPY')
plt.axis('off')
```

Out[87]: (-0.5, 64.5, 52.5, -0.5)



In [ ]: