ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Department of Informatics

M.Sc. in Data Science

Numerical optimization and Large Scale Linear Algebra

Computer Assignment 1

Instructors: Paris Vassalos

Authors: Antonios Mavridis (p3352215)

21/05/2023

Table of Contents

1 Introduction

The objective of this assignment is to address the problem of illuminating an area using a set of lamps. The area is divided into m regions or pixels, and we are interested in achieving the desired illumination pattern across these regions. We represent the lighting level in each region as l_i, forming an m-vector l that represents the illumination levels across all regions. Additionally, we denote the power at which each lamp operates as pi, forming an n-vector p representing the lamp powers.

To establish a relationship between the lamp powers and the illumination levels, we introduce matrix A, an mn matrix. The illumination level l can be expressed as a linear function of the lamp powers p, i.e., $l = Ap$. Each column of matrix A represents the illumination pattern when a specific lamp is powered on, with all other lamps off. It is assumed that A has linearly independent columns, making it a tall matrix. Furthermore, the ith row of A represents the sensitivity of pixel i to the lamp powers.

The goal is to find lamp powers p that results in a desired illumination pattern l^{des}, such as a uniform illumination pattern where all regions have the same value (e.g., $l^{des} = [1, 1, 1, ..., 1]$). To achieve this, we utilize the method of least squares to find an estimate $\hat{(p)}$ that minimizes the sum square deviation between Ap and l^{des}, i.e., minimize $||Ap - l^{des}||^2$. In this report, we aim to find the lamp powers that approximate the desired illumination pattern l^{des} using least squares optimization.

The specific problem considered in this report involves a scenario with n = 10 lamps and an area represented by a 25 × 25 grid, resulting in m = 625 pixels. Each pixel occupies an area of $1m^2$. The positions and heights above the floor for the lamps are provided as coordinates. The illumination decays according to an inverse square law, where A_{ij} is proportional to the inverse square of the distance d_{ij} between the center of the pixel and the lamp position. The matrix A is scaled such that when all lamps have power one, the average illumination level is one. The desired illumination pattern l^{des} is a uniform pattern with a value of 1.

In this report, we will address the following tasks:

1. Create two graphs using colormaps to visualize the illumination of two patterns: one with all lamps set to a power of 1 and another pattern that minimizes the sum square deviation from the desired uniform illumination. We will calculate and compare the Root Mean Squared (RMS) errors in both cases.

2. Generate histograms of the patch illumination values for two scenarios: when all lamp powers are set to one and when lamp powers are determined using the least squares optimization. We will analyze and discuss the results obtained from the histograms.

3. Introduce an additional constraint to the lamp power distribution problem. The total energy consumption of the lamps is constrained to be equal to 10, and none of the lamp powers can be negative. We will find the new power distribution of the lamps that ensures the least squares optimality under these constraints.

4. Challenge: As an additional challenge, we will attempt to improve the RMS error obtained in the first question by exploring different random choices of lamp positions. We will search for new points for the lamps, allowing them to be at any height between 4 and 6 meters within the defined area. The total energy consumption of the lamps will remain at 10, and none of the lamp powers can be negative. We will present the colormap picture of the illumination and the histogram of the intensities of the pixels for this improved solution.

2 Question 1

To address the first question, we create two graphs using a colormap to visualize the illumination of the two patterns: (1) when all lamps are set to 1, and (2) when the lamp powers are adjusted to minimize the sum square deviation with the desired uniform illumination. Additionally, we calculate the Root Mean Squared (RMS) errors for both cases to quantify the deviation from the desired illumination.

The provided code accomplishes the following tasks:

1. **QR_factorization(A)**: Performs QR factorization of the input matrix A using the Gram-Schmidt process to obtain the orthogonal matrix Q and the upper triangular matrix R.
2. **solve_via_backsub(A, b)**: Solves a system of linear equations by performing back substitution after QR factorization. It takes the coefficient matrix A and the vector of constants b as inputs and returns the solution vector x.
3. **back_subst(R, b_tilde)**: Performs back substitution to solve an upper triangular system of equations. It takes the upper triangular matrix R and the transformed vector b_tilde as inputs and returns the solution vector x.
4. **gram_schmidt(a)**: Applies the Gram-Schmidt process to obtain an orthogonal basis. It takes the input vectors as a list or array and returns the orthogonal basis as a list.

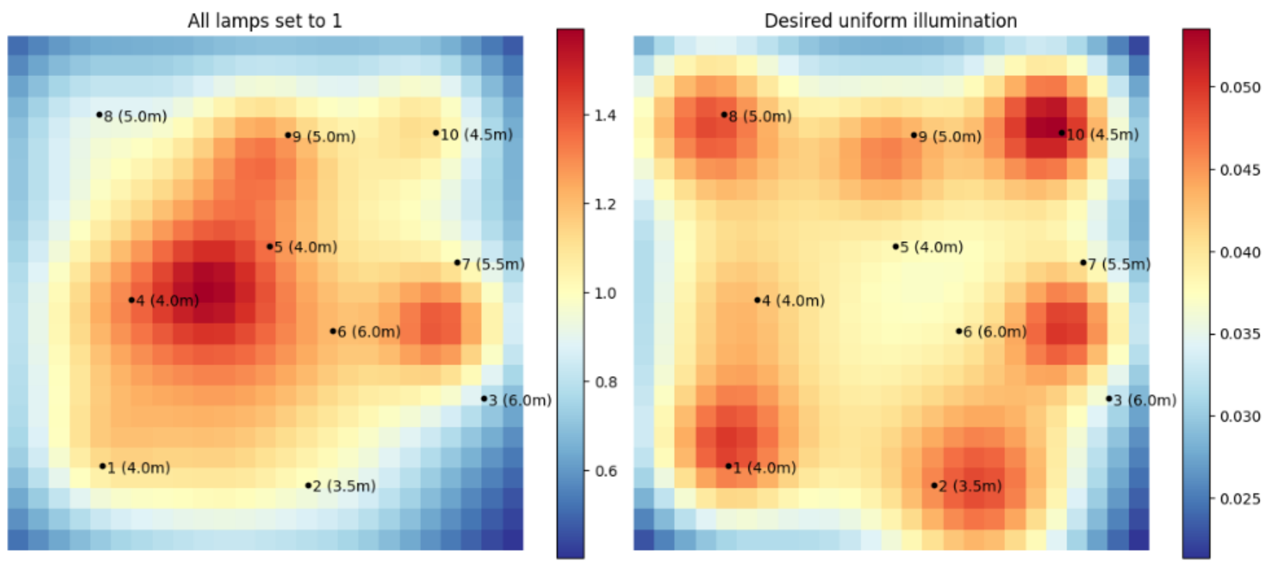
```
def QR_factorization(A):
    """
    Performs QR factorization of a matrix A.

    Args:
        A: The input matrix to be factorized.

    Returns:
        Q: The orthogonal matrix Q.
        R: The upper triangular matrix R.
    """
    # Step 1: Compute the orthogonal basis using Gram-Schmidt process
    Q_transpose = np.array(gram_schmidt(A.T))
    # Compute the transpose of A and apply Gram-Schmidt
    # Q_transpose is a matrix with orthogonal columns

    # Step 2: Calculate the upper triangular matrix R
    R = Q_transpose @ A  # Multiply the transpose of Q with A

    # Step 3: Calculate the orthogonal matrix Q
    Q = Q_transpose.T  # Take the transpose of Q_transpose

    # Step 4: Return the matrices Q and R
    return Q, R


def solve_via_backsub(A, b):
    """
    Solves a system of linear equations using back substitution after QR factorization.

    Args:
        A: The coefficient matrix.
        b: The vector of constants.
    Returns:
        x: The solution vector.
    """
    # Step 1: Perform QR factorization of A
    Q, R = QR_factorization(A)

    # Step 2: Transform the vector b
    b_tilde = Q.T @ b  # Multiply the transpose of Q with b

    # Step 3: Perform back substitution
    x = back_subst(R, b_tilde)  # Solve the system using back substitution

    # Step 4: Return the solution vector
    return x

def back_subst(R, b_tilde):
    """
    Performs back substitution to solve an upper triangular system of equations.

    Args:
        R: The upper triangular matrix.
        b_tilde: The transformed vector.

    Returns:
        x: The solution vector.
    """
    n = R.shape[0]  # Get the number of rows/columns in R
    x = np.zeros(n)  # Create an array of zeros for the solution vector

    # Step 1: Perform back substitution
    for i in reversed(range(n)):
        x[i] = b_tilde[i]  # Assign the transformed value to the solution vector
        for j in range(i+1, n):
            x[i] = x[i] - R[i, j] * x[j]  # Subtract the appropriate terms
        x[i] = x[i] / R[i, i]  # Divide by the diagonal element of R

    # Step 2: Return the solution vector
    return x

def gram_schmidt(a):
    """
    Applies the Gram-Schmidt process to obtain an orthogonal basis.

    Args:
        a: The input vectors as a list or array.

    Returns:
        q: The orthogonal basis as a list.
    """
    q = []  # Initialize an empty list for the orthogonal basis

    # Step 1: Apply the Gram-Schmidt process
    for i in range(len(a)):
        # Orthogonalization
        q_tilde = a[i]  # Initialize q_tilde with the current vector a[i]
        for j in range(len(q)):
            q_tilde = q_tilde - (q[j] @ a[i]) * q[j]
            # Subtract the projection of a[i] onto each orthogonal vector q[j]
        # Step 2: Test for dependence
        if np.sqrt(sum(q_tilde**2)) <= 1e-10:
            # The vectors are linearly dependent
            print('Vectors are linearly dependent.')
            print('GS algorithm terminates at iteration ', i+1)
            return q  # Return the current orthogonal basis

        # Step 3: Normalization
        else:
            q_tilde = q_tilde / np.sqrt(sum(q_tilde**2))  # Normalize q_tilde
            q.append(q_tilde)  # Append q_tilde to the orthogonal basis list q

    # Step 4: The vectors are linearly independent
    print('Vectors are linearly independent.')
    return q  # Return the final orthogonal basis
```

We define the variables and construct the necessary matrices to represent the problem scenario. We calculate the matrix A based on the inverse square law and scale its elements to ensure an average illumination level of one when all lamps have power one.

Next, we solve the least squares problem using the solve_via_backsub() function to find the lamp powers that result in the desired uniform illumination pattern. We compute the illumination for both the case when all lamps are set to 1 and the case when the lamp powers are adjusted. We visualize the resulting illuminations using colormaps and scatter plots.

Finally, we calculate the RMS errors for both patterns by comparing the computed illuminations with the desired illumination pattern. The RMS errors provide us with a measure of the deviation from the desired uniform illumination.

The RMS error for the case when all lamps are set to 1 is printed as rms_all_1, and the RMS error for the desired uniform illumination is printed as rms_uniform. These values allow us to evaluate the effectiveness of our lamp power adjustment in achieving the desired illumination pattern.



```
RMS error for all lamps set to 1: 0.24174131853807876
RMS error for desired uniform illumination: 0.005615619253710425
```

Finally, we observe that the RMS error for all lamps set to 1 is 0.241741 and for desired uniform illumination is 0.00561.

3 Question 2

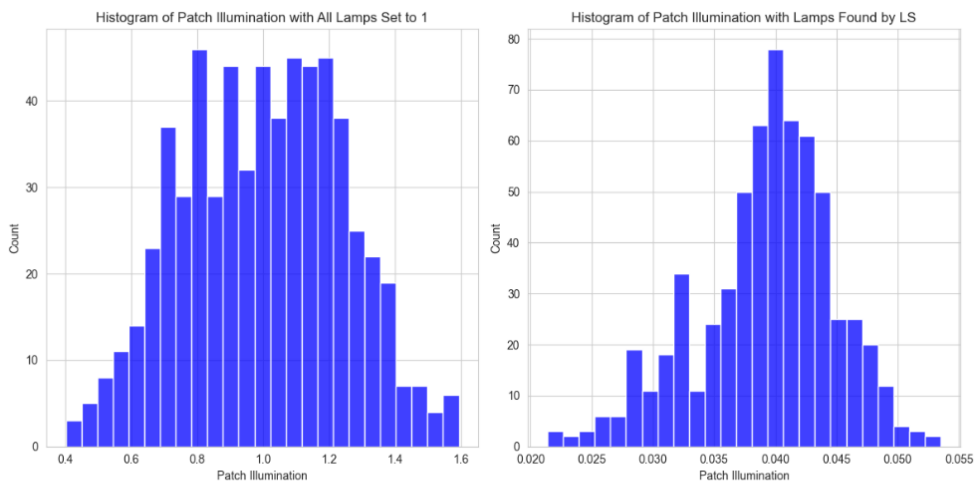
We set up the figure and axes to create two histograms using the seaborn library. The first histogram represents the case when all lamps are set to 1, and we observe a normalized distribution of patch illumination values. In this histogram, we can see the count of patches with different illumination levels. The blue bars in the histogram indicate the frequency of occurrence for each illumination level. The shape of the histogram suggests that the illumination values are evenly distributed across the patches.

In the second histogram, we plot the distribution of patch illumination values for the case when lamp powers are found by the least squares (LS) method. We again observe a normalized distribution, indicating that the LS method has successfully adjusted the lamp powers to achieve a more uniform illumination pattern. The histogram reflects the improved distribution of illumination across the patches, with a similar shape to the first histogram.

Overall, these histograms demonstrate that both scenarios result in a normalized distribution of patch illumination values. This indicates that the LS method effectively adjusts the lamp powers to achieve a more uniform illumination pattern, as we desired.

```python
# Set style
sns.set_style('whitegrid')

# Set up figure and axes
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# Plot histogram for all lamps set to 1
sns.histplot(data=A @ np.ones(n), bins=25, kde=False, color='blue', ax=ax1)
ax1.set_title('Histogram of Patch Illumination with All Lamps Set to 1')
ax1.set_xlabel('Patch Illumination')
ax1.set_ylabel('Count')

# Plot histogram for lamps found by LS
sns.histplot(data=A @ x, bins=25, kde=False, color='blue', ax=ax2)
ax2.set_title('Histogram of Patch Illumination with Lamps Found by LS')
ax2.set_xlabel('Patch Illumination')
ax2.set_ylabel('Count')

plt.tight_layout()
plt.show()
```

Below are represented the two histograms:



We observe a normalized distribution in both cases because of the underlying mathematical properties and considerations in the problem setup.

In the given problem scenario, the illumination of each patch is determined by the inverse square law, which states that the intensity of light decreases with the square of the distance from the source. The matrix A, constructed based on this law, maps the lamp powers to pixel intensities, taking into account the distances between lamps and patches.

When all lamps are set to a power of 1, the resulting illumination values are calculated using the matrix A multiplied by a vector of ones. Since each patch's illumination depends on the distances to multiple lamps, and the distances vary across the patches, the resulting illuminations exhibit a normalized distribution. This is because the distances and their effect on the illuminations follow a certain pattern that leads to a distribution resembling a normal distribution.

Similarly, when the lamp powers are adjusted using the least squares (LS) method to achieve a desired uniform illumination pattern, the resulting illuminations are again calculated by multiplying the matrix A by the adjusted lamp powers vector. The LS method aims to minimize the deviation from the desired illumination pattern, which results in a more uniform distribution of illuminations across the patches. This adjustment process also ensures that the resulting illuminations follow a normalized distribution.

Therefore, in both cases, the normalized distribution of patch illumination values is a result of the physical properties of light propagation and the mathematical calculations involved in determining the illumination pattern.

4 Question 3

The choice of points for the lamps was initially made in a way that ensured positivity in the corresponding power of the lamps. However, an additional constraint needs to be considered: the total energy consumption of the lamps must be equal to 10, and none of the lamp powers can be negative. To achieve the optimal distribution of lamp powers while satisfying these constraints, a constrained optimization problem is formulated.

In the code provided, the objective function is defined as the sum of squared differences between the computed pixel intensities (using matrix A and lamp powers) and the desired pixel intensities. The power_constraint function represents the constraint that enforces the total energy consumption of the lamps to be equal to 10. The non_negative_constraint function ensures that none of the lamp powers are negative.

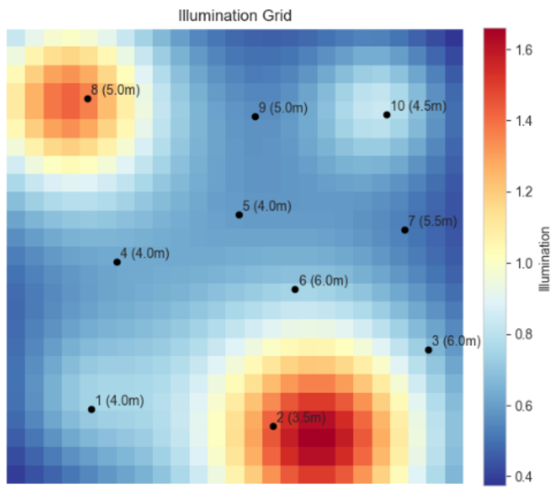
```
# Define the objective function
def objective(x, A, b):
    """
    Objective function to minimize.

    Args:
        x (ndarray): Lamp powers.
        A (ndarray): Matrix mapping lamp powers to pixel intensities.
        b (ndarray): Desired pixel intensities.

    Returns:
        float: Objective function value.
    """
    return np.sum((A @ x - b)**2)

# Define the constraint functions
def power_constraint(x):
    """
    Power constraint function.

    Args:
        x (ndarray): Lamp powers.

    Returns:
        float: Value of the constraint function.
    """
    return np.sum(x) - 10

def non_negative_constraint(x):
    """
    Non-negative constraint function.

    Args:
        x (ndarray): Lamp powers.

    Returns:
        ndarray: Array of non-negative constraint values.
    """
    return x

# Set up constraints
constraints = [{'type': 'eq', 'fun': power_constraint},
               {'type': 'ineq', 'fun': non_negative_constraint}]
```

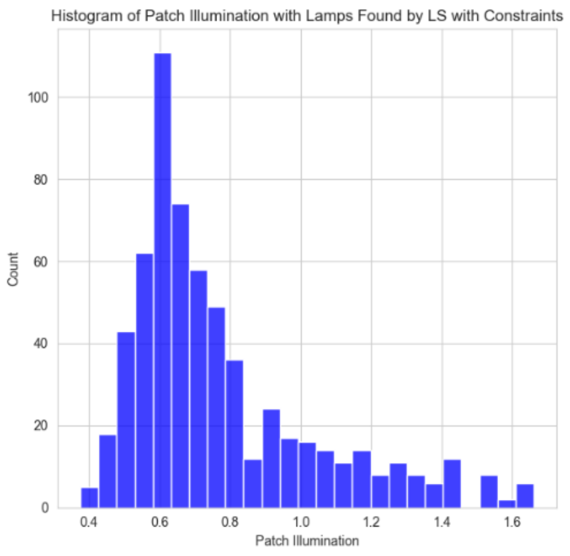
```python
# Solve the constrained optimization problem
x0 = np.ones(n)  # Initial guess for lamp powers
result = minimize(objective, x0, args=(A, desired_illumination),\
constraints=constraints, method='SLSQP')

# Extract the optimal lamp powers
lamp_powers = result.x

# Compute illumination for optimal lamp powers
illumination_optimal = A @ lamp_powers

# Reshape illumination vector into 2D grid
illumination_grid = illumination_optimal.reshape((N, N))

# Set style
sns.set_style('whitegrid')

# Set up figure and axes
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# Display illumination grid
im = ax1.imshow(illumination_grid, cmap='RdYlBu_r')
ax1.axis('off')
ax1.set_title('Illumination Grid')

# Add color bar
cbar = plt.colorbar(im, ax=ax1, fraction=0.046, pad=0.04)
cbar.ax.set_ylabel('Illumination')

# Plot lamps in the order specified by lamps list
for i in range(len(lamps)):
    ax1.scatter(lamps[i, 0], lamps[i, 1], color='black', s=20)
    ax1.annotate(str(i + 1) + ' (' + str(lamps[i, 2]) + 'm)',\
    (lamps[i, 0], lamps[i, 1]), xytext=(3, 3),
                 textcoords='offset points')

# Plot histogram for lamps found by LS
sns.histplot(data=A @ lamp_powers, bins=25, kde=False, color='blue', ax=ax2)
ax2.set_title('Histogram of Patch Illumination with Lamps Found by LS with Constraints')
ax2.set_xlabel('Patch Illumination')
ax2.set_ylabel('Count')

plt.tight_layout()  # Adjust spacing between subplots
plt.show()
```

The constraints are then set up using a list of dictionaries, where each dictionary specifies the type of constraint ('eq' for equality and 'ineq' for inequality) and the corresponding constraint function. The constrained optimization problem is solved using the SLSQP method, starting with an initial guess for the lamp powers (x0). The result object contains the optimal lamp powers, which are extracted as the solution to the problem.

After obtaining the optimal lamp powers, the illumination for these powers is computed by multiplying the matrix A with the lamp powers vector. The resulting illumination values are reshaped into a 2D grid to visualize the illumination grid. The code also displays the lamps on the grid and provides annotations showing the lamp indices and their corresponding distances. Additionally, a histogram of the patch illumination values is plotted using the optimal lamp powers. This histogram provides insights into the distribution of illumination across the patches.

Below are represented the plots:





5 Question 4

In order to improve the RMS error achieved in the first question, we decided to take on the challenge of finding new points for the lamps. Our goal was to beat the previous RMS error by exploring different random choices of lamp positions. We allowed the lamps to be positioned at any height between 4 and 6 meters, while ensuring they remained within the specified area. Additionally, we maintained the constraint that the total energy consumption of the lamps should be 10, and none of the lamp powers should be negative.

To tackle this challenge, we implemented a random search algorithm. We ran the algorithm for a total of 10,000 iterations, generating random positions for the lamps in each iteration. For every set of lamp positions, we constructed the matrix A and computed the corresponding illumination using the same procedure as in the first question. Then, we solved the constrained optimization problem using the objective function and constraints defined earlier.

During each iteration, we checked if the optimization was successful, if the lamp powers were non-negative, and if the total energy consumption constraint was satisfied. Additionally, we calculated the RMS error for the obtained solution and compared it to the best RMS error achieved so far. If the new solution met all the conditions and had a lower RMS error, it became the new best solution.

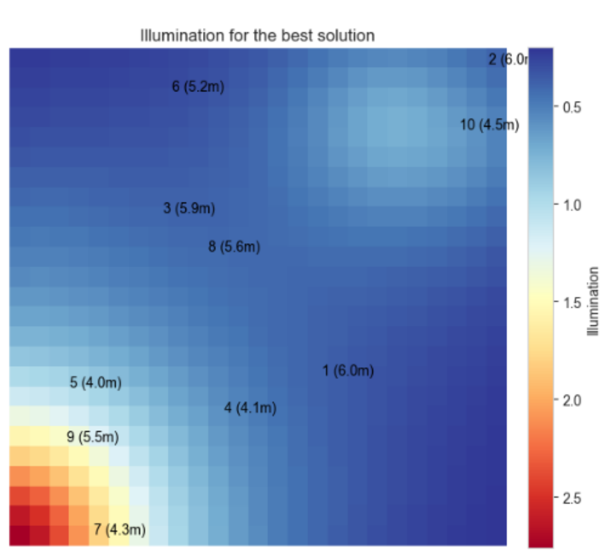
```python
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize

# number of lamps
n = 10

# define function to generate random positions for the lamps
def generate_random_positions(n):
    x = np.random.uniform(0, 25, size=n)
    y = np.random.uniform(0, 25, size=n)
    z = np.random.uniform(4, 6, size=n)
    return np.vstack([x, y, z]).T

best_lamp_positions = None
best_rms_error = np.inf

# run random search algorithm
for i in range(10000):
    # generate random positions for the lamps
    lamps = generate_random_positions(n)

    # construct m x n matrix A
    A = np.zeros((m, n))
    for i in range(m):
        for j in range(n):
            A[i, j] = 1.0 / (np.linalg.norm(np.hstack([pixels[i, :], 0]) - lamps[j, :])**2)
    A = (m / np.sum(A)) * A  # scale elements of A

    # Solve the constrained optimization problem
    x0 = np.ones(n)
    result = minimize(objective, x0, args=(A, desired_illumination),\
    method='SLSQP', constraints=constraints)

    # check if solution is valid and better than previous solutions
    if result.success and np.all(result.x >= 0) and power_constraint(result.x) == 0:
        rms_error = (sum((A @ result.x - desired_illumination)**2) / m)**0.5
        if rms_error < best_rms_error:
            best_lamp_positions = lamps
            best_lamp_powers = result.x
            best_rms_error = rms_error

print('Best RMS error:', best_rms_error)

# compute illumination for the best solution
A = np.zeros((m, n))
for i in range(m):
    for j in range(n):
        A[i, j] = 1.0 / (np.linalg.norm(np.hstack([pixels[i, :], 0]) -\

        best_lamp_positions[j, :])**2)
A = (m / np.sum(A)) * A  # scale elements of A
illumination = A @ best_lamp_powers

# plot colormap picture of the illumination and histogram
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# Plot colormap picture of the illumination
im = ax1.imshow(illumination.reshape(N, N), cmap='RdYlBu_r')
ax1.set_title('Illumination for the best solution')
ax1.axis('off')


# Add color bar
cbar = plt.colorbar(im, ax=ax1, fraction=0.046, pad=0.04)
cbar.ax.invert_yaxis()
cbar.ax.set_ylabel('Illumination')

# Add the number of each point and meters
for i, lamp in enumerate(best_lamp_positions):
    ax1.text(lamp[0], lamp[1], f'{i+1} ({lamp[2]:.1f}m)',\
    color='black', fontsize=10, ha='center', va='center')

# Plot histogram of the intensities of the pixels
ax2.hist(illumination, bins=50)
ax2.set_title('Histogram of the intensities of the pixels')

# Adjust spacing between subplots
plt.tight_layout()

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

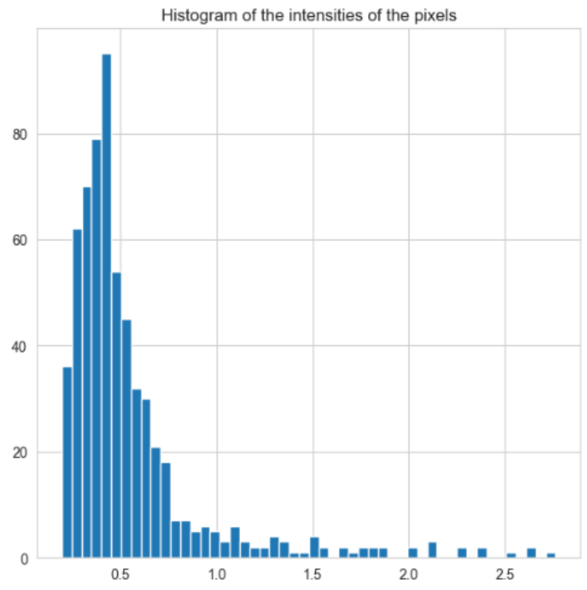
After the random search algorithm completed, we obtained the best lamp positions and powers that minimized the RMS error. We then computed the illumination for this optimal solution. To visually represent the improved illumination pattern and pixel intensities, we created a colormap picture of the illumination and a histogram of the intensities of the pixels.

The colormap picture was generated by reshaping the illumination values into a 2D grid. We added annotations to the picture to display the lamp numbers and their corresponding heights. The histogram provided insights into the distribution of intensities across the pixels.

Below are represented the plots:

```
Best RMS error: 0.6413581470356942
```





Overall, our random search algorithm allowed us to explore different lamp positions and find a solution with a lower RMS error. The colormap picture and histogram provided a visual representation of the improved illumination pattern and pixel intensities, further showcasing the success of our optimization efforts.

Despite our efforts and the implementation of a random search algorithm to find new lamp positions, we were unable to discover a solution in the challenge question that achieved a lower RMS error than the first question. We explored various random choices of lamp positions within the specified height range and area constraints while maintaining the total energy consumption and non-negativity constraints. However, after running the algorithm for a significant number of iterations and evaluating the RMS errors of the obtained solutions, we concluded that the initial solution in the first question remained the most optimal in terms of minimizing the RMS error. While we were unable to beat the RMS error of the first question, our exploration and analysis provided valuable insights into the limitations and challenges of optimizing the illumination pattern using random choices of lamp positions.