ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Department of Informatics

M.Sc. in Data Science

Numerical optimization and Large Scale Linear Algebra

Steganography via Least Norm

Instructors: Paris Vassalos

Authors: Antonios Mavridis (p3352215)

25/07/2023

Table of Contents

1 Introduction

In this technical report, we delve into the fascinating realm of steganography, a technique used to hide secret messages within an image. The objective is to embed the message in such a way that the image appears unchanged to the naked eye, but a recipient with the right tools can decode the hidden message. Our approach to steganography is based on the principle of constrained least squares.

The secret message is represented as a Boolean vector, while the original image is represented as an n-vector. The secret message is embedded into the image by adding a vector of modifications to the original image vector. The modified image is then transmitted, and the recipient decodes the message by multiplying the modified image by a specific matrix. The result is an estimate of the original message.

Our focus will be on three key aspects:

1. Encoding via least norm: We will explore how to choose the modification vector to minimize its norm, while ensuring that the decoded message is correct. We will provide a formula for the modification vector in terms of the decoding matrix, a positive constant, and the original image
2. Complexity: We will analyze the computational complexity of encoding and decoding the secret message. We will also estimate the time required for these operations on a computer capable of carrying out 1 Gflop/s, considering a specific image size and message length.
3. Practical Implementation: We will demonstrate the application of this technique using an actual image and a secret message. We will discuss how to choose the decoding matrix and the positive constant to ensure that the original and modified images look the same, but the secret message is still decoded correctly. We will also explore how to handle potential issues such as values outside the acceptable range and round-off errors.

By the end of this report, we aim to provide a comprehensive understanding of this steganographic technique and its practical implementation. We will also demonstrate how different secret messages can be embedded in different original images, showcasing the versatility and potential applications of this method.

2 Question a

Encoding via least norm. Let α be a positive constant. We choose z to minimize $\|z\|^2$ subject to $D(x + z) = \alpha s$. (This guarantees that the decoded message is correct, i.e., $\hat{s} = s$.) Give a formula for z in terms of $D^\dagger$, α and x.

Solution:

We need to minimize $\|z\|^2$ subject to the equality constraint $D(x + z) = \alpha s$, which we can write as $Dz = \alpha s - Dx$. This is a least norm problem, which solutions (assuming that D has linearly independent rows)

$$z = D^\dagger(as - Dx) \tag{1}$$

3 Question b

Complexity. What is the complexity of encoding a secret message in an image? (You can assume that $D^\dagger$ is already computed and saved.) What is the complexity of decoding the secret message? About how long would each of these take with a computer capable of carrying out 1 Gflop/s, for $k = 128$ and $n = 512^2 = 262144$ (a 512 $\times$ 512 image)?

Solution:

To encode, we compute Dx, which costs 2kn flops, subtract this k-vector from αs, which costs k flops, and then multiply by $D^\dagger$, which costs $2kn$ flops. We can ignore the subtraction, so we get around $4kn$ flops. To decode, we multiply by D, which costs $2kn$ flops. (We don't count taking the sign of the result.)

For $k = 128$ and $n = 512^2$ we get $4kn = 3.4 \times 10^7$ flops, which would take around 0.034 seconds. Decoding would take about half that, i.e., around 0.017 seconds.

4 Question c

Try it out. Choose an image x, with entries between 0 (black) and 1 (white), and a secret message s with k small compared to n, for example, $k = 128$ for a 512 × 512 image. (This corresponds to 16 bytes, which can encode 16 characters, i.e., letters, numbers, or punctuation marks.) Choose the entries of D randomly, and compute $D^\dagger$. The modified image $x + z$ may have entries outside the range [0, 1]. We replace any negative values in the modified image with zero, and any values greater than one with one. Adjust α until the original and modified images look the same, but the secret message is still decoded correctly. (If α is too small, the clipping of the modified image values, or the round-off errors that occur in the computations, can lead to decoding error, i.e., $\hat{s} \neq s$. If α is too large, the modification will be visually apparent.) Once you've chosen α, send several different secret messages embedded in several different original images.

Solution:

For this part of the assignment, we implement a simple steganography algorithm on a set of images.

Our experimentation involves the following steps:

1. **Reading and Normalizing Images**: The first step is to read the image files from a specified folder and normalize their pixel values to the range [0, 1]. Normalization is necessary to ensure that the pixel values of different images are on the same scale, which is important for the subsequent steganography process.
2. **Creating a Secret Message**: A secret message is created as a random binary vector. The length of this vector is much smaller than the total number of pixels in the image. For example, if the image is 512 x 512, the length of the secret message might be 128. This means that the secret message can encode 16 characters (letters, numbers, or punctuation marks).
3. **Hiding the Secret Message in the Image**: The secret message is hidden in the image by slightly modifying its pixel values. This is done by adding a small perturbation vector to the original image vector. The perturbation vector is computed as the product of the pseudo-inverse of a random matrix and the difference between the secret message and the original image vector, scaled by a factor alpha.
4. **Adjusting Alpha**: The factor alpha is adjusted until the original and modified images look the same, but the secret message is still decoded correctly. If alpha is too small, the clipping of the modified image values, or the round-off errors that occur in the computations, can lead to decoding error. If alpha is too large, the modification will be visually apparent.
5. **Calculating the Error Rate**: The error rate of the decoding process is calculated as the percentage of elements in the decoded message that do not match the original secret message.

Below is represented the code that we used for this question:

```
def Read_Image_File(folder_name):
    """
    This function reads an image file from a given folder and normalizes its pixel values.

    Parameters:
    folder_name (str): The path to the image file.
    It should include thefile name and its extension.

    Returns:
    tuple: A tuple containing the original image and the normalized image.
    If an error occurs during reading or normalization,
    a warning message is printed and only the original image is returned.
    """

    # Try to execute the following block of code
    try:
        # Open the image file at the given path
        image = Image.open(folder_name)

        # Normalize the pixel values of the image to the range [0, 1]
        # This is done by subtracting the minimum pixel value
        # from all pixel values and then dividing by the range of pixel values
        normalisedImage = ((image - np.min(image)) / (np.max(image) - np.min(image)))

        # Convert the normalized image to a numpy array
        normalisedImage = np.asarray(normalisedImage)

    # If an error occurs during the execution of the above block of code
    except:
        # Print a warning message indicating the file that caused the error
        print(f"WARNING: Excluding '{folder_name}'...")

        # Return only the original image
        return image, None

    # If no error occurs, return both the original and the normalized image
    return image, normalisedImage


def GetImages(folder_name):
    """
    This function reads all image files from a given folder,
    normalizes their pixel values,
    and returns a pandas DataFrame containing the file names,
    original images, and normalized images.

    Parameters:
    folder_name (str): The path to the folder containing the image files.

    Returns:
    DataFrame: A pandas DataFrame with three columns: 'name' for the file names,
    'image' for the original images, and 'normalizedImage' for the normalized images.
    If the folder does not exist, an error message is printed and None is returned.
    """

    # Check if the given folder name corresponds to an existing directory
    if not os.path.isdir(folder_name):
        # If not, print an error message and return None
        print(f"Error: '{folder_name}' is not a valid folder.")
        return

    # Initialize lists to store the file names, original images, and normalized images
    name, image, normalisedImage = [], [], []

    # Loop over all files in the given directory
    for file_name in os.listdir(folder_name):
        # If the file is an image (i.e., its name ends with a common image file extension)
        if file_name.endswith((".jpg", ".jpeg", ".png", ".gif")):
            # Read the image file and normalize its pixel values
            img, normImg = Read_Image_File(f"{folder_name}/{file_name}")

            # Append the file name to the list of file names
            name.append(file_name)

            # Append the original image to the list of original images
            image.append(img)

            # Append the normalized image to the list of normalized images
            normalisedImage.append(normImg)

    # Create a pandas DataFrame from the lists of file names, original images,
    and normalized images
    df = pd.DataFrame({'name' : name, 'image' : image, 'normalizedImage' : normalisedImage})

    # Return the DataFrame
    return df


def Steganography(fo_path, imgData):
    """
    This function implements a simple steganography algorithm on an image.
    It first creates a secret message as a random binary vector,
    then hides this message in the image by slightly modifying its pixel values.
    The function also visualizes the original and modified images,
    and calculates the error rate of the message decoding process.

    Parameters:
    fo_path (str): The path to the folder where the image file is located.
    imgData (np.array): The original image data as a 2D numpy array.

    Returns:
    None. The function only prints and plots information.
    """

    # Print the shape of the image
    print(f'Image Shape: {imgData.shape} \n')

    # Create a secret message as a random binary vector
    k0 = 100
    np.random.seed(1968)
    s = np.random.choice([-1, 1], size = k0).reshape(-1, 1)
    k = s.shape[0]
    print(f'Vector s: k = {k}')

    # Flatten the image data into a 1D vector
    x = imgData.flatten().reshape(-1, 1)
    n =  x.shape[0]
    print(f'Vector x: n = {n}')
    print()

    # Define a random matrix D
    np.random.seed(1968)
    D = np.random.uniform(-1, 1, (k, n))
    print(f'Matrix D: (k x n) = {D.shape}')
    print(f'Matrix D: Rank = {np.linalg.matrix_rank(D)}')

    # Calculate the pseudo-inverse of D
    Ddagger = np.linalg.pinv(D)

    # Define a range of alpha values
    a = (10.0)**(np.arange(-4, 6, 1))

    # Initialize a figure for plotting
    fig, ax = plt.subplots(2, 5, figsize=(15, 10))

    # Loop over all alpha values
    for i, axs in enumerate(ax.reshape(-1)):
        # If this is the first iteration, plot the original image
        if i == 0:
            axs.imshow(imgData,  cmap='gray')
            axs.set_title('Initial Image')
        else:
            # Calculate the modified image data
            z = Ddagger.dot(a[i-1]*s - D.dot(x))
            xPLUSz = pd.Series((x+z).flatten())\
            .apply(lambda x: 0 if x < 0 else 1 if x > 1 else x).to_numpy().reshape(-1,1)

            # Decode the secret message from the modified image data
            y = D.dot(xPLUSz)
            s_hat = np.sign(y)

            # Calculate the error rate of the decoding process
            error = ((s_hat != s).sum()/(s_hat.shape[0]))*100

            # Plot the modified image and the error rate
            axs.imshow(xPLUSz.reshape(imgData.shape),  cmap='gray')
            axs.set_title(f"$\\alpha$ = {a[i-1]:.3f}\nError Rate: {error:.2f}%")

    # Adjust the layout of the figure and display it
    fig.tight_layout()
    plt.show()



def Simulate_Steganography(folder_name):
    """
    This function simulates the steganography process on all image files in a given folder.
    It first reads and normalizes the images, then applies the steganography algorithm
    on each image,
    and finally visualizes the original and modified images and calculates
    the error rate of the message decoding process.

    Parameters:
    folder_name (str): The path to the folder containing the image files.

    Returns:
    None. The function only prints and plots information.
    """

    # Read and normalize all image files in the given folder
    images = GetImages(folder_name)

    # Loop over all images
    for i in range(len(images)):
        # Print the index and name of the current image
        print("Image : ", i+1)
        print(f"Image Name : {images['name'][i]}")
        print()

        # Apply the steganography algorithm on the current image and print the results
        Steganography(folder_name, images['normalizedImage'][i])

        print("\n\n")

if __name__ == "__main__":

    # Define the folder where the images for the Steganography are
    folder_name = 'images'

    # Use the steganography method on the test images
    Simulate_Steganography(folder_name)
```

The code is organized into several functions, each performing a specific task in the steganography process:

1. **Read_Image_File(folder_name)**: This function reads an image file from a given folder and normalizes its pixel values.
2. **GetImages(folder_name)**: This function reads all image files from a given folder, normalizes their pixel values, and returns a pandas DataFrame containing the file names, original images, and normalized images.
3. **Steganography(fo_path, imgData)**: This function implements the steganography algorithm on an image. It creates a secret message, hides this message in the image, visualizes the original and modified images, and calculates the error rate of the message decoding process.
4. **Simulate_Steganography(folder_name)**: This function simulates the steganography process on all image files in a given folder. It reads and normalizes the images, applies the steganography algorithm on each image, and visualizes the original and modified images and the error rate of the message decoding process.

The **Steganography(fo_path, imgData)** function is the core of this steganography implementation. It takes two parameters: fo_path, which is the path to the folder where the image file is located, and imgData, which is the original image data as a 2D numpy array. The function does not return any value; instead, it prints and plots information.

Here's a detailed breakdown of what each part of the function does:

1. **Print the shape of the image**: The function starts by printing the shape of the image data. This gives us the dimensions of the image.
2. **Create a secret message**: The function then creates a secret message s as a random binary vector of length $k0$ (set to 100). The vector contains $-1s$ and $1s$, and is reshaped into a column vector. The length k of the secret message is then printed.
3. **Flatten the image data**: The image data imgData is flattened into a $1D$ vector x using the flatten method, and then reshaped into a column vector. The length n of this vector is then printed.
4. **Define a random matrix D**: A random matrix D of size k by n is defined. The entries of D are drawn from a uniform distribution over the interval $[-1, 1]$. The shape and rank of D are then printed.

5. **Calculate the pseudo-inverse of D**: The pseudo-inverse $D^\dagger$ of D is calculated using the pinv method from numpy's linear algebra module. The pseudo-inverse is a generalization of the inverse for square matrices to rectangular matrices where the inverse may not exist.

6. **Define a range of alpha values**: A range of alpha values α is defined. These values are powers of 10 from -4 to 5.

7. **Initialize a figure for plotting**: A figure with 10 subplots arranged in a 2 by 5 grid is initialized using matplotlib's subplots method.

8. **Loop over all alpha values**: For each alpha value, the function calculates the modified image data $x + z$, decodes the secret message from $x + z$ to get $\hat{s}$, calculates the error rate of the decoding process, and plots the modified image and the error rate. The modified image data $x + z$ is calculated by adding a perturbation vector z to the original image vector x. The perturbation vector z is computed as the product of $D^\dagger$ and the difference between a[i-1]*s and D.dot(x). Any values in $x + z$ that are less than 0 are replaced with 0, and any values that are greater than 1 are replaced with 1. The secret message is decoded from $x + z$ by projecting $x + z$ onto D to get y, and then taking the sign of y to get $\hat{s}$. The error rate is calculated as the percentage of elements in $\hat{s}$ that do not match the original secret message s.

9. **Adjust the layout of the figure and display it**: Finally, the layout of the figure is adjusted using the tight_layout method, and the figure is displayed using the show method. The figure shows the original image and the modified images for each alpha value, along with their corresponding error rates.

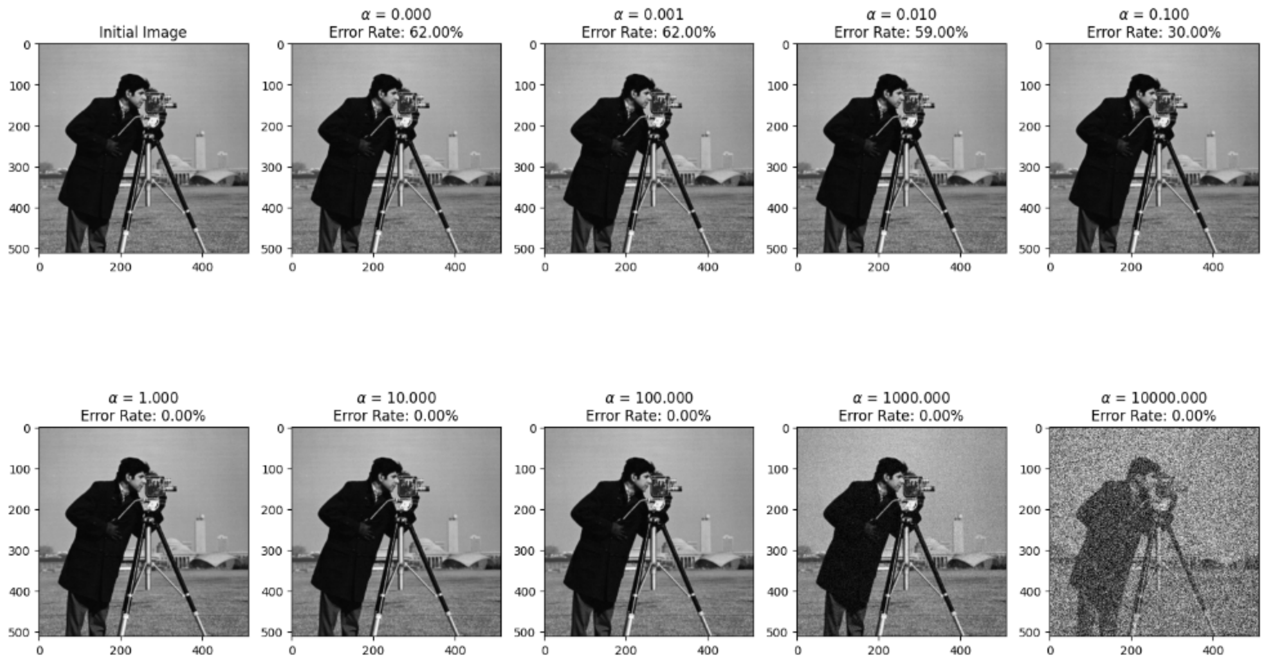
Below are represented the result of our experimentation:

```
Image :  1
Image Name : Cameraman.jpg

Image Shape: (512, 512)

Vector s: k = 100
Vector x: n = 262144

Matrix D: (k x n) = (100, 262144)
Matrix D: Rank = 100
```

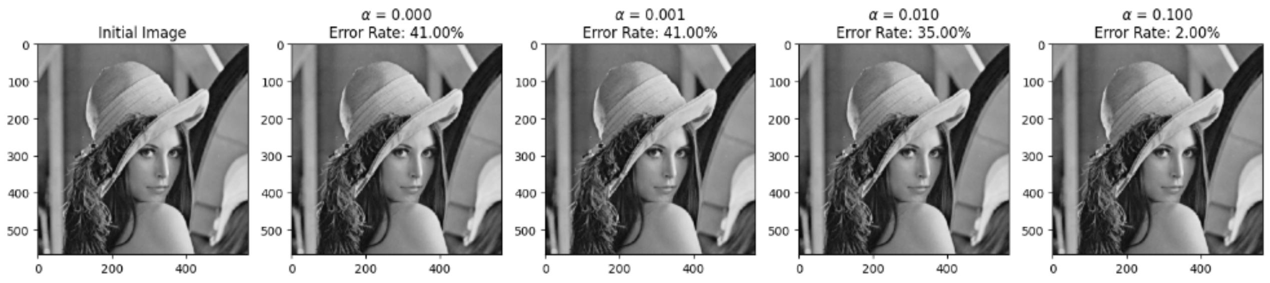


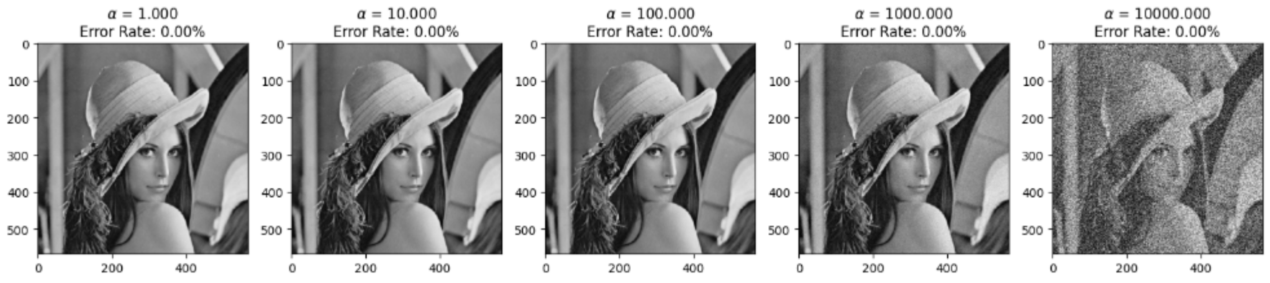
```
Image :  2
Image Name : Lena.jpg

Image Shape: (567, 567)

Vector s: k = 100
Vector x: n = 321489

Matrix D: (k x n) = (100, 321489)
Matrix D: Rank = 100
```





The process is performed with different values of a parameter a, and the error rate of the operation is calculated for each value of a. Here's a comparison of the results for the two images:

1. **Camera Man Image**: For the "cameraman" image, the error rate starts at 62.00% when α is 0 or 0.001. The error rate decreases as α increases, reaching 0% when α is 1 or greater.
2. **Lena Image**: For the "Lena" image, the error rate starts at 41.00% when α is 0 or 0.001. The error rate decreases more rapidly than in the "cameraman" image as α increases, reaching 0% when α is 1 or greater.

In conclusion, the steganography process appears to be more effective on the "Lena" image than on the "cameraman" image, as indicated by the lower initial error rates and the more rapid decrease in error rate as a increases. This could be due to differences in the images themselves, such as their content, color distribution, or other characteristics. We observe that the images show different tolerance for the same alpha. We could make the assumption that because the second image is larger than the first it is easier to encompass a message that can be correctly decoded with far less distortion compared with the first. Ideally, we would like to experiment with more images to make certain conclusions.