TSKS37 Lab 3
Image Processing and Monte Carlo Simulation

Fall 2024
Version of this document: January 17, 2025

Examination

This exercise is a set of tasks described further down in the document. We expect you to present the required figure(s). The lab assistant may also ask for an explanation of the written code.

There is no requirement on using jupyter for this lab, and thus you can use either a jupyter notebook or regular `.py` files.

Submission

After passing the labs, the code of Lab 2 and Lab 3 should be uploaded to the Lisam study room under Submissions. The course director will send the code to Ouriginal. The code from these two labs should be saved in one text document called coursecode_year_studentid1_studentid2.txt (e.g., TSKS37_2024_adaer110_evaer120.txt). In case you are using jupyter lab you can export your code through: `file > Export Notebook as > Executable Script`.

1 Edge Detection, Compression and Reconstruction of Image

In this exercise you will practice using Numpy. The goal is to detect edges in an image, giving a new image with edges. Then you will compress the edge image and finally reconstruct the edge image. You do not need to understand these concepts, the goal is to practice your Numpy skills going from equations to numerical evaluation. **We have provided you a code skeleton at the end of this exercise.**

Today several AI tools for writing code exist, we will not stop you from using them. However, we expect you to be able to explain every single line of your code. When you end up at a company after graduation you will surely use AI tools and it's very important that you understand what they do. If you (or the AI) use a function from an existing library, read the documentation for the function and be prepared to answer why you chose to use that function.

To pass this exercise you need to show and demonstrate the script and/or functions that you have created to solve the main tasks. Keep a balance between comments as well as good variable/function names in the code, well written code is easy to read and speaks for itself. Ensure that the plots are self-explaining.

1.1 Main Tasks

1. Convert a color image to gray scale

2. Detect edges in the gray scale image
3. Compress the edge image
4. Reconstruct the edge image from the compressed edge image
5. Run a monte carlo simulation to validate the reconstruction algorithm

1.2 Libraries

We will use the following Python libraries:

- Numpy
- PIL
- Matplotlib

There are many other libraries we could use that would make the lab easier (or more difficult). We encourage you to try those libraries on your own.

2 Mathematical Preliminaries

In this section we introduce some of the mathematics used in the lab. Do not get hung up on the math if you don't understand it, the math you have to implement with Numpy is laid out for you in the next section.

2.1 Image as a Matrix or Vector

Let our gray-scale image $\mathbf{X}$ be an $H \times W$ matrix: $\mathbf{X} \in \mathbb{R}^{H \times W}$, $H, W \in \mathbb{N}$. We can express the image as a vector $\mathbf{x} \in \mathbb{R}^{HW}$ where $\mathbf{x} = [x_{1,1}, x_{1,2}, \ldots, x_{1,W}, x_{2,1}, \ldots, x_{H,W}]^T$ which is just a flat version of $\mathbf{X}$. (lower-case x is an element, lower-case bold $\mathbf{x}$ is a vector, upper-case bold $\mathbf{X}$ is a matrix.)

2.2 Edge Detection

First we will apply edge detection to $\mathbf{X}$, which will result in an image with pixel values 0 and 1 only. We have 1 where our edge-detection algorithm has detected an edge. We denote the resulting binary image by $\mathbf{x}^e$. Example in Figure 1.

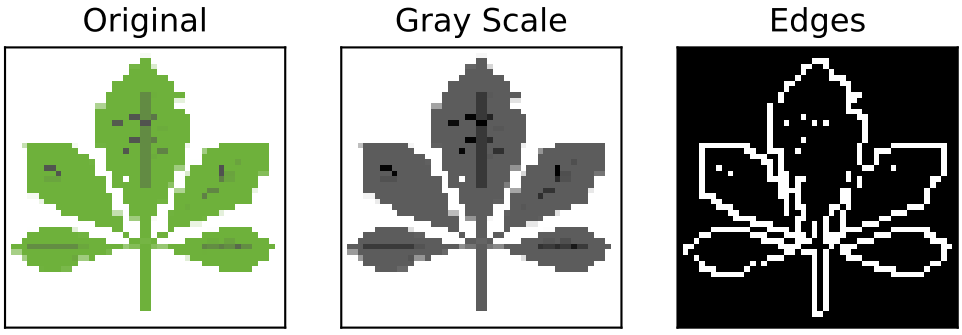


Figure 1: Example of image RGB (original), gray scale and edges only.

2.3 Compression

The type of compression used in this lab is linear compression, where we achieve compression by multiplying our image $\mathbf{x}^e$ with a compression matrix $\Phi \in \mathbb{R}^{M \times HW}$. Using Φ we define the compressed signal as $\mathbf{y}_0 = \Phi \mathbf{x}^e \in \mathbb{R}^M$. The *compression coefficient* r represents the ratio of compression, giving $M = \lfloor HW/r \rfloor$, where $\lfloor \cdot \rfloor$ means round down to nearest integer. While Φ is typically not invertible, we can still approximate $\mathbf{x}^e$ if it sparse enough, meaning that most elements in $\mathbf{x}^e$ are zero.

Since communication channels almost always introduce additive noise, we make the reconstruction more difficult by adding normally distributed noise: $\mathbf{n} \in \mathbb{R}^M, \mathbf{n} \sim \mathcal{N}(0, \sigma^2)$. This gives the noisy signal $\mathbf{y} = \mathbf{y}_0 + \mathbf{n}$. σ^2 is the variance of the noise, a higher variance gives a more severe noise.

This compression technique differs from the more common image compression techniques such as JPEG. However, there are similarities in that both methods transform the image and exploit image sparsity.

2.4 Reconstruction

Finally we want to reconstruct $\mathbf{x}^e$ from $\mathbf{y}$ where the reconstructed image is denoted by $\mathbf{x}' \in \mathbb{R}^{HW}$. This reconstruction is done by solving the following optimization problem:

$$\min_{\mathbf{x}'} f(\mathbf{x}'),$$

where

$$f(\mathbf{x}') = \frac{1}{2L}||\mathbf{y} - \Phi \mathbf{x}'||_2^2 + \alpha||\mathbf{x}'||_1 \in \mathbb{R}, L = HW,$$

and $||\cdot||_1$ is the 1-norm of a vector, which is the sum of the absolute value of each vector element. The 1-norm is used for "regularization" to reduce the impact of the noise on the reconstructed image $\mathbf{x}'$. Furthermore, $||\mathbf{x}||_2 = \sqrt{\mathbf{x}^T \mathbf{x}}$ is the 2-norm of a vector.

In TATA43, *Calculus in Several Variables*, you will learn about the gradient of a function: $\nabla f(\mathbf{x}') \in \mathbb{R}^{HW}$. The gradient $\nabla f(\mathbf{x}')$ is a vector pointing in the direction in which $f(\mathbf{x}')$ increases maximally in the point $\mathbf{x}'$. Similarly the negative gradient $-\nabla f(\mathbf{x}')$ points in the direction of maximum *decrease*. The gradient is simply the derivative of $f(\mathbf{x}')$ with respect to each element in $\mathbf{x}'$. We will minimize $f(\mathbf{x}')$ by taking repeated steps in the direction of maximum decrease, see Figure 2. This is a very common algorithm called *Gradient Descent* which you will learn about in TANA21 *Scientific Computing*. Gradient Descent is also used to train machine learning models such as neural networks.

2.5 Reconstruction Performance and Monte Carlo Simulation

To measure the performance of the reconstruction we introduce the *error rate* $E = ||\mathbf{x}' - \mathbf{x}^e||_0/(HW)$. Here we use the 0-norm $||\cdot||_0$, which is the number of non-zero elements.

To reliably compute E we need to perform a *monte carlo* (MC) simulation. A MC simulation is a method where you repeat an experiment several times, and compute the average result to account for an inherent randomness in the experiment. In this case, we want to repeat the experiment since $\mathbf{n}$ and Φ are random.

To describe the MC method, we consider the case of estimating a DC current I as seen in Figure 3. When we measure I, the instrument and environment will introduce thermal noise and interference, which we can model as a normally distributed random variable: $w \sim \mathcal{N}(0, 1)$ with mean 0 and variance 1. As a result, what we get from each measurement is $y = I + w$. To reduce the impact of w, we perform N measurements and get $y_n = I + w_n$ for each measurement $n \in \{1, 2, ..., N\}$. Finally, we

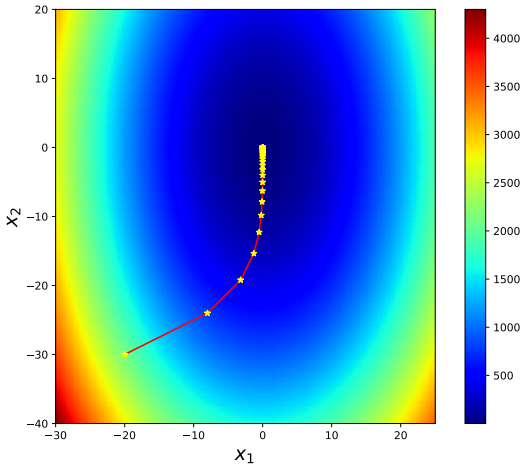


Figure 2: Visualization of Gradient Descent where $3x_1^2 + x_2^2$ is minimized by walking in the direction of maximum decrease. That is, walking from high values (red) to low values (blue).

say that our DC current estimate is $I \approx \hat{I} = \sum_{n=1}^{N} \frac{y_n}{N}$ with an error $E_I = I - \hat{I}$. This procedure is *one* experiment, but to tell if our estimator $\hat{I}$ is good we have to repeat the experiment several times to compute the average E_I, that repeated process is a MC simulation.

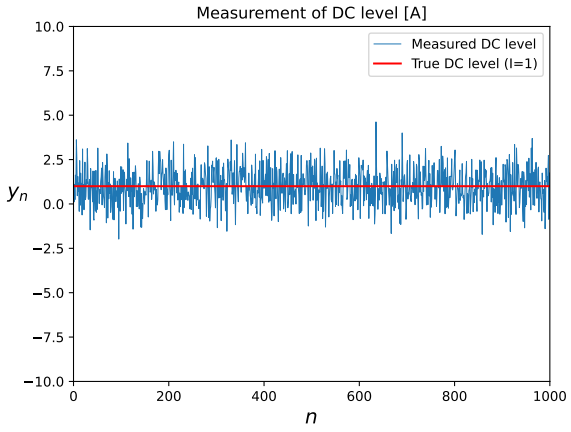


Figure 3: Example of measuring a DC level affected by normally distributed noise and interference.

In the same way, generating $\mathbf{n}$, Φ, compressing, reconstructing and finally computing E is *one* experiment. However, to know E reliably we need to repeat the experiment several times and compute the average E.

3 Edge Detection, Compression and Reconstruction of Image

You do not have to understand the concepts described above, you just have to follow the instructions below and implement the algorithms using Numpy. The only loops you should need are included in the code skeleton, but additional loops can be useful for plotting your results. Feel free to experiment with the different parameters, such as the thresholds T, sparse coefficient r and σ^2. Your final code

including all algorithms and all plots should run in about 1 minute. In Figure 4 you have a flowchart giving you an overview of what you will do in this assignment.

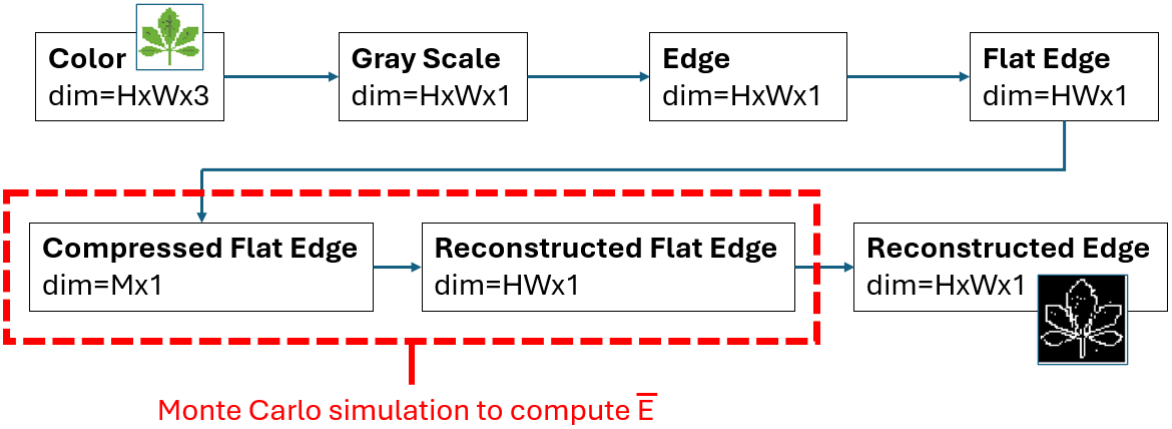


Figure 4: An overview of what you will do.

3.1 Load image, convert to gray scale

1. Load `leaf.png` using PIL, convert to a numpy array. Feel free to use another image at your own risk.
2. Plot the array using Matplotlib imshow().
3. What is the dimension of the array?
4. Convert the array to gray scale using Numpy. There is no function in Numpy for this, you have to weigh the RGB components into a single scalar. The value of each gray scale pixel is calculated as follows: $p = (R \cdot 0.299 + G \cdot 0.587 + B \cdot 0.114)/255$, where R, G, B are the Red, Green and Blue components of each pixel.
5. Plot the gray scale array using imshow(). Make sure your plot is in gray scale, you might have to add an extra argument to imshow().

Hints:

- Be very careful with the dimensions of your Numpy arrays. Vectors should always be column vectors (only 1 column). Double-check with np.shape().
- Apart from R,G,B, your loaded image might have a fourth dimension for transparency, ignore this.

3.2 Edge Detection

1. Create two copies of your gray scale array.
2. In the first array, subtract every pixel with its direct right neighbour. In the second array, subtract every pixel with its direct lower neighbour. Pixels without these neighbours can be left as they are.
3. Take the absolute value of each pixel in the resulting two arrays, then add the two arrays into a third array.

4. Let T be a threshold, for any pixel in the third array set it to 0 if it is smaller than T, otherwise set the pixel to 1. We recommend $T = 0.35$, but feel free to test different values to see the effect.
5. Replicate Figure 1.

3.3 Compression

1. Generate $\Phi \in \mathbb{R}^{M \times HW}$ by drawing $M \cdot HW$ numbers from the normal distribution $\mathcal{N}(0, 1)$. Put the generated numbers into into a $M \times HW$ array to create Φ. Generate 3 versions of Φ using $r \in \{1, 2, 3\}$.
2. Compute $\mathbf{y} \in \mathbb{R}^M$ using variances $\sigma^2 \in \{0, 10, 20\}$. If $\sigma^2 = 0$ we have no additive noise, which is the point. Remember to flatten the array into the vector $\mathbf{x}^e \in \mathbb{R}^{HW}$ first.
3. This should give you 9 instances of $\mathbf{y}$ using 3 different σ^2 and 3 different r. You can use loops to iterate over the different σ^2 and r.

Hint: Use `numpy.random.normal()`, set the argument *size* appropriately. Make sure you set the variance correctly, sometimes the expected argument is the standard deviation (read the documentation carefully). The standard deviation is the square root of the variance.

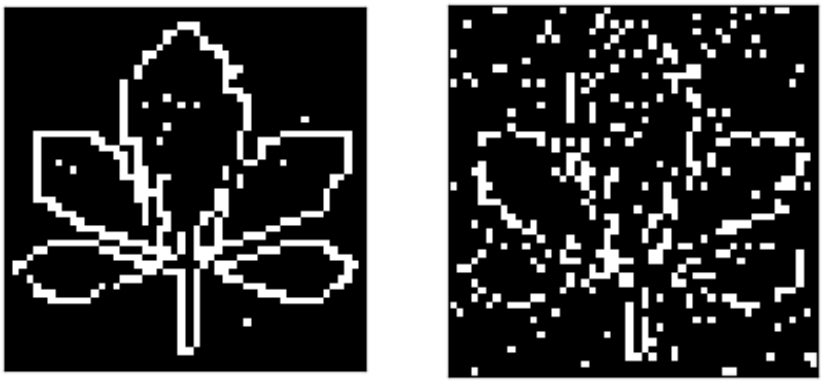
3.4 Reconstruction

Now we are going to reconstruct $\mathbf{x}^e$ from each of the 9 $\mathbf{y}$ where we denote the reconstruction by $\mathbf{x}'$. To do this we will solve $\min_{\mathbf{x}'} f(\mathbf{x}')$ using gradient descent. Loops are allowed for iterating over the gradient descent steps.

1. Set all elements in $\mathbf{x}'$ to 0.
2. Compute $\nabla f(\mathbf{x}') = -\frac{\Phi^T}{L}(\mathbf{y} - \Phi\mathbf{x}') + \alpha \text{sign}(\mathbf{x}')$, where $\alpha = 0.01$.
3. Compute $\mathbf{x}' = \mathbf{x}' - \gamma \nabla f(\mathbf{x}')$ where $\gamma = 0.5$. This updates $\mathbf{x}'$.
4. Compute $f(\mathbf{x}')$ and save it for step 7. Repeat from step 2, 25 times, then stop and go to step 5.
5. Set all values of $\mathbf{x}'$ smaller than T to 0, all values greater than T to 1 where T is a threshold. Let $T = 0.25$. The $\mathbf{x}'$ we have now is our reconstruction. Calculate E and save it.
6. Reshape $\mathbf{x}'$ into the array of size $H \times W$ and plot it using imshow(). Repeat for all 9 instances of $\mathbf{y}$ generated in *3.3. Compression.*
7. Plot $f(\mathbf{x}')$ for all 25 iterations for all 9 instances of $\mathbf{y}$ in a single plot. Let the iteration number be on the x-axis, $f(\mathbf{x}')$ on the y-axis, we will have 9 curves in the plot. Each curve should be labeled with its σ^2 and r which should be displayed in the plot legend. Make sure each curve is distinguishable. Save the plot. See Figure 6 for a sketch how how your plot should look.

Hints:

- Avoid computing every matrix multiplication in every iteration. Some of the multiplications, such as $\Phi^T\mathbf{y}$, only needs to be done once and not in every iteration. Expand $\nabla f(\mathbf{x}')$ from point (2.) and see for yourself what you can compute outside of the loop.
- The function *sign*() computes the sign of its argument, use `numpy.sign()`.
- As $\mathbf{x}'$ is updated $f(\mathbf{x}')$ should decrease.



(a) $r = 1, \sigma^2 = 0$ (b) $r = 3, \sigma^2 = 20$

Figure 5: What you can expect from the reconstruction.

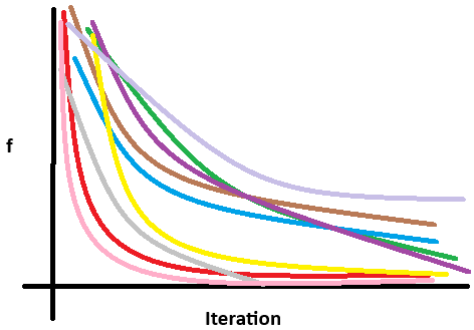


Figure 6: Sketch of how your plot of f in the 9 different cases should look like. It can be useful to let the y-axis be in log scale, and to include a plot legend. Make sure you can distinguish the 9 lines if the figure was printed in gray scale.

- Use Figure 5 as a reference for what your result should look like.

3.5 Monte Carlo Simulation

Since our experiment depends on the random variables Φ, $\mathbf{n}$, a Monte Carlo simulation is necessary for a more reliable measure of E. In addition to the 9 reconstructions you did in Section 3.4, you will repeat the 9 compressions and reconstructions $N = 10$ times (10 experiments) to compute the average $\overline{E}$ for each of the 9 instances, where

$$\overline{E} = \frac{1}{10} \sum_{n=1}^{10} E_n,$$

and E_n is E for each of the N experiments. $\overline{E}$ is a more reliable measure of the reconstruction performance. Here you do not have to save the actual reconstructed $\mathbf{x}^e$, only the error E_n for each of the 9 cases in each of the N experiments.

Note: Make sure to generate a *new* Φ and $\mathbf{n}$ for every experiment.

Hint: 10 experiments takes about 30 seconds in total for an image of size 50×50 pixels.

3.6 Plot the reconstructed edge images

Make a single figure with all 9 reconstructed edge images, meaning 9 subplots. Every subplot must have a title with σ^2, r and $\overline{E}$. You only have to include the 9 reconstructed edge images from the initial compression and reconstruction in Section 3.4. See Figure 7 for a reference of how your plot should look.

Note: Loops are allowed. Also, your $\mathbf{x}^e$ are column vectors and must be reshaped to 2D arrays before you can plot them as images.

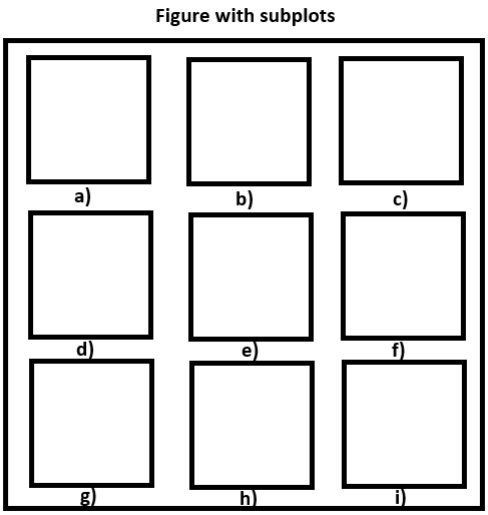


Figure 7: Sketch of how your plot with your 9 different reconstructions should look like. The captions of each sub-plot should specify the parameters for each reconstruction.

3.7 Code Skeleton

(This is also available in `/courses/TSKS37` and lisam)

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Given parameters
N_MC = 10
R = [1,2,3]
SIGMA2 = [0,10,20]
ALPHA = 0.01
GAMMA = 0.5
NUM_ITER = 25
T_RECONSTRUCT = 0.25
T_EDGE = 0.35

# ========================================================= #
# Load image here.
# --------------------------------------------------------- #

# ========================================================= #
# Convert to gray scale here.
# --------------------------------------------------------- #

# ========================================================= #
# Detect edges here.
# --------------------------------------------------------- #

# ========================================================= #
# Reproduce figure with image, gray scale and edges here.
# --------------------------------------------------------- #

# ========================================================= #
# Compress and reconstruct here, store the 9 reconstructed
# images.
# --------------------------------------------------------- #

# ========================================================= #
# Run Monte Carlo simulation here to compute the average
# error for each case.
for n in range(N_MC):
# EVERY RANDOM VARIABLE HAS TO BE CREATED INSIDE THIS LOOP
  for r in R:
    pass
    for sigma_2 in SIGMA2:
      print(r, sigma_2)
      # ===================================================== #
      # Compress here
      # ----------------------------------------------------- #

      # ===================================================== #
      # Reconstruct here
      for i in range(NUM_ITER):
        pass
      # ----------------------------------------------------- #
# --------------------------------------------------------- #

# ========================================================= #
# 3x3 sub plot here.
# --------------------------------------------------------- #
```