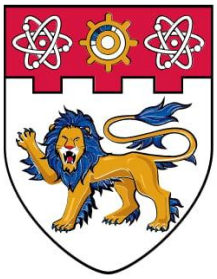


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

College of Computing and Data Science

AY 2025/2026

SC4061 Computer Vision

Lab 1 Report

Submitted By: Isaac Chun Jun Heng

Matriculation No.: U2221389B

Table of Contents

1. Introduction

The objective of this laboratory is to introduce image processing in the MATLAB context, although I have chosen to use Python as allowed in the brief. The following concepts are introduced in the laboratory, including experiments for:

1. Contrast Stretching
2. Histogram Equalization
3. Linear Spatial Filtering
4. Median Filtering
5. Suppressing Noise Interference Patterns
6. Undoing Perspective Distortion of Planar Surface
7. Perceptron Algorithms Implementation as per Lecture Slides

The following subsections includes the technical details to replicate the results, and the official source code can be found at: https://github.com/isaacchunn/computer-vision if any of the files seem to not load.

1.1 Environment Setup

The following table describes the environment that was used to run the experiments. Although the experiments are relatively simple and did not require any form of neural networks, extra information such as CUDA Toolkit and GPU specification were included just for completeness.

Hardware Specifications	
Central Processing Unit (CPU)	Ryzen 9800X3D 8-Core Processor
Graphics Processing Unit (GPU)	NVIDIA RTX 4070 SUPER 12GB GDDR6X VRAM
Random Access Memory (RAM)	32GB DDR5
Storage	1 TB Samsung 990 Pro SSD
Software Environment	
Operating System	Ubuntu WSL 2 Subsystem
Python	3.8.8
NVIDIA CUDA Toolkit	11.8
GPU Driver	NVIDIA Game Ready Driver 580.97
Development Tools	
IDE	Visual Studio Code w/ Jupyter Extension
Environment	Conda

To run the source code, detailed setup instructions can be found in the repository to replicate the results in the experiments below.

1.2 Helper Functions

Throughout the code, the following helper functions are implemented for reusability and consistency. Typically, these are functions related to displaying of images to keep the plots consistent and are used in the generation of diagrams included in this report.

1. load_image
    a. Reads an image using cv2 and changes the cvtColor if greyscale flag is provided.
2. display_image
    a. Displays an image automatically and plots it out on a plt.figure of size 10,10
3. display_images_side_by_side
    a. Displays two image side by side, typically for analysis. The function just creates plt subplots and plot both of the images side by side.

```python
# Helper function to load images, we are mainly working with greyscale images this lab
def load_image(path, is_greyscale = True):
    image = cv2.imread(path)
    if image is not None:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) if is_greyscale else cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    return image

# Helper function to display images, we will use this a lot
def display_image(image, figsize = (10, 10), title = "Image", vmin = None, vmax = None, grey = True):
    cmap_v = "gray" if grey else None
    if image is None:
        return
    plt.figure(figsize = figsize)
    plt.imshow(image, cmap = cmap_v, vmin = vmin, vmax = vmax)
    plt.title(title)
    plt.show()

# Helper function to display two images side by side for comparison
def display_images_side_by_side(image1, image2, title1="Image 1", title2="Image 2", figsize=(15, 6)):
    if image1 is None or image2 is None:
        print("One or both images are None")
        return

    fig, axes = plt.subplots(1, 2, figsize=figsize)

    # Display first image
    axes[0].imshow(image1, cmap='gray')
    axes[0].set_title(title1, fontsize=14)

    # Display second image
    axes[1].imshow(image2, cmap='gray')
    axes[1].set_title(title2, fontsize=14)

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

2. Experiments

In this section, thorough analysis of the experiments are done, starting with an overview of the key concepts involved in each experiment, followed by snippets of key source codes used to complete the experiment. Diagrams and analysis of those diagrams are also included as part of each experiment to display the results and insights drawn from these experiments in a clear and concise manner.

2.1 Contrast Stretching

This section provides the analysis for the 2.1 Contrast Stretching experiment listed in the Lab Manual.

2.1.1 What is Contrast Stretching?

Contrast stretching (also known as normalization or min-max scaling) is an image enhancement technique which aims to increase an image's contrast by expanding a narrow range of pixel intensity values found in an image to a wider desired range, such as the full 0-255 range for an 8-bit image. It aims to make details more visible by creating greater differences between dark and bright areas, using linear transformation functions. The rough process of this is highlighted by:

$$P_{\text{new}} = \frac{P_{\text{old}} - P_{\text{min}}}{P_{\text{max}} - P_{\text{min}}} \times 255 \quad (1)$$

where P_{new} denotes the intensity of the new pixel, P_{old} is the original intensity of the pixel. P_{min} and P_{max} denote the minimum and maximum intensity values throughout the image. 255 is a scaling factor for 8-bit images.

2.1.2 Part A and B – Loading of Image and Converting to greyscale

Figure 1: Contrast stretching – loading of pictures

From the figure, the loading of image was successful, and it was loaded as a grey scale image. The display also matches the dimensions of the image (320x443).

2.1.2 Part C – Minimum and Maximum Intensities

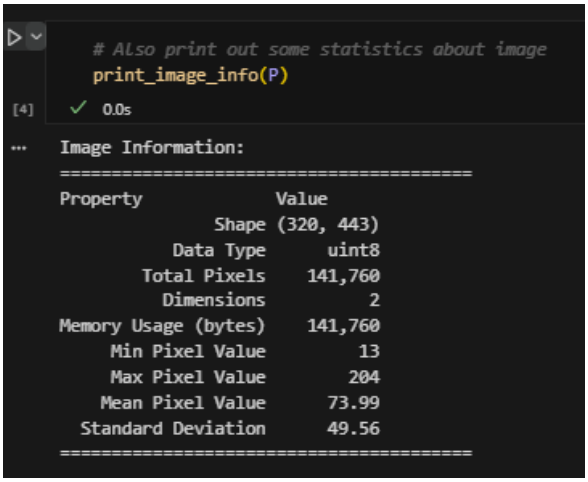


Figure 2: Minimum and Maximum Intensities of Image

Before applying contrast stretching, we need to identify the current range of pixel intensities in the image. In OpenCV, this is done using the **min()** and **max()** functions, that are encapsulated in my helper function. We can see that the min pixel intensity is **13**, and the max pixel intensity is **204**. The range is 204-13=**191** (out of possible 255). This indicates the image is **underutilizing** the available intensity range, which results in **reduced** contrast. We should/may apply contrast

stretching to redistribute these values across the full [0,255] range, which should increase the image's visual quality. Furthermore, the mean of **73.99** indicates that the average pixel intensity is closer to the darker end, and the standard deviation of **49.56** shows moderate variation in pixel intensities. In simple words, the image is really dark!

2.1.3 Part D & E – Implementing Contrast Stretching and Display

By applying the formula listed in part 2.1.1, to the image, we successfully executed contrast stretching onto the image.

Figure 3: Contrast stretched MRT image

To verify the results, we print the image information and realize that the pixel with least intensity is now **0**, while the max pixel intensity is now **255**. These pixel intensities now fully utilize the uint8 range [0,255], which has enhanced the image contrast. The image's mean pixel intensity has also increased to **81.43** from the original **73.99**, showing that the image seems a little bit brighter. Furthermore, there is more deviation between the pixels **66.17** compared to the old **49.56**, which is expected due to the pixels now utilizing the full range.

```
Contrast Stretched Image Information:
=====================================
Property             Value
               Shape (320, 443)
          Data Type    float64
        Total Pixels    141,760
          Dimensions          2
Memory Usage (bytes)  1,134,080
     Min Pixel Value        0.0
     Max Pixel Value      255.0
    Mean Pixel Value      81.43
  Standard Deviation      66.17
=====================================
```

Figure 4: Contrast stretched MRT image information

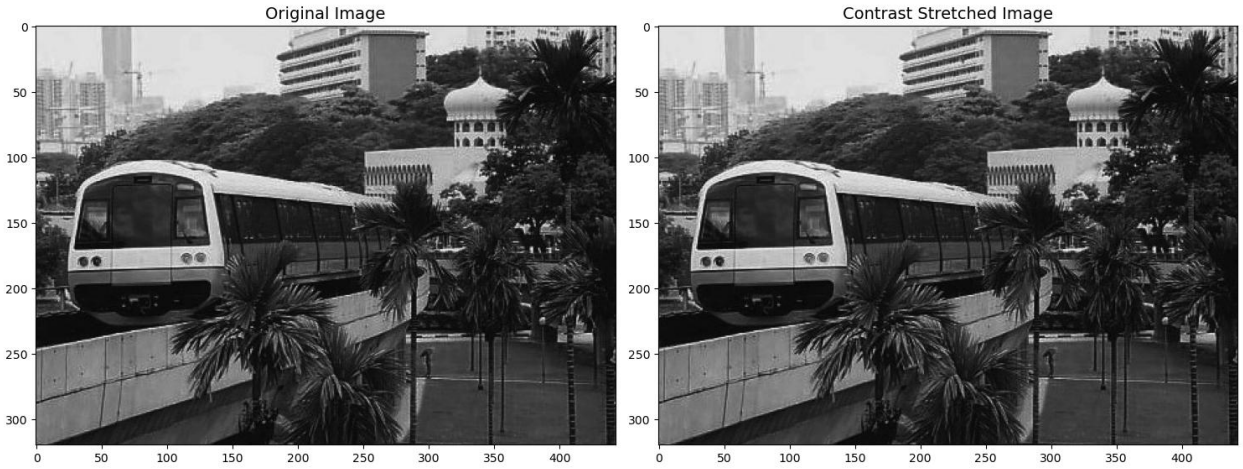


Figure 5: Original Image vs Contrast Stretched Image

2.2 Histogram Equalization

This section analyzes histogram equalization techniques and compare their effectiveness with different bin sizes.

2.2.1 What is Histogram Equalization?

Histogram equalization typically involves nonlinearly mapping some gray levels to other levels, in order to create a resultant histogram which is approximately uniform. In image processing, it typically improves an image's contrast by redistributing its pixel intensity values through analysis of its intensity histogram to create a transformation function that stretches the distribution to occupy the full possible range. This may or may not enhance detail and visibility in images with a narrow range of intensities.

2.2.2 Part A – Displaying of Image Intensity Histogram of Bin Size 10 and 256

```
def plot_intensity_histogram(image, bins, range = (0, 255), title = "Image"):
    # I think we can also use flatten over ravel - but flatten does occupy memory so we use ravel here for simplicity
    plt.hist(image.ravel(), bins = bins, range = range)
    plt.title(f"{title} ({bins} bins)")
    plt.xlabel("Pixel Intensity")
    plt.ylabel("Frequency")
    plt.show()
```

✓ 0.0s

```
# Plotting out intensity histogram with 10 bins
plot_intensity_histogram(P, 10, (0, 255), "Histogram of Original MRT Image")
# Plotting out intensity histogram with 256 bins
plot_intensity_histogram(P, 256, (0, 255), "Histogram of Original MRT Image")
```

✓ 0.1s

In this section, we use the plot_intensity_histogram helper function to plot out histograms of bin size **10** and **256**. From the histogram, we will see the distribution of the pixel intensities relative to their frequency.

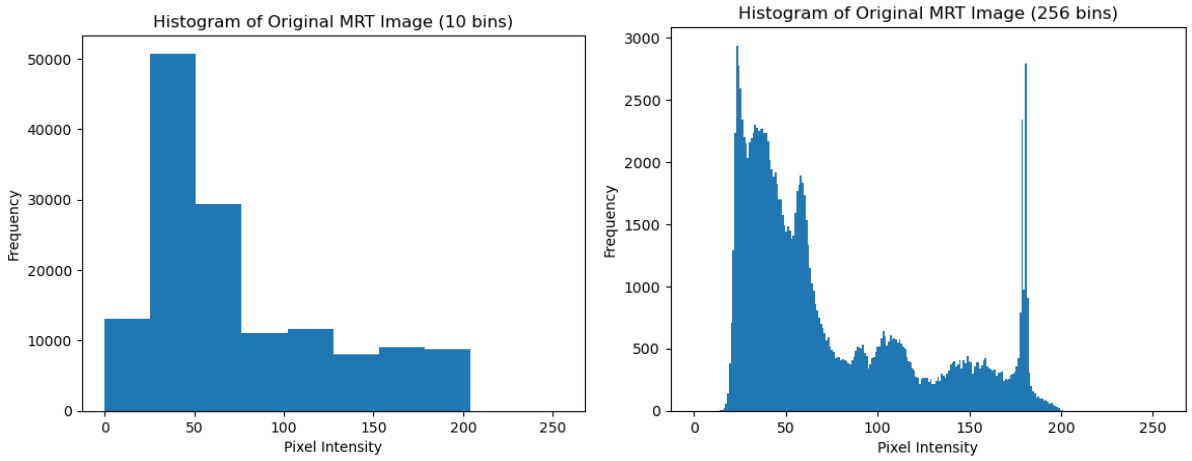


Figure 6: Histogram Plot (10 bins and 256 bins)

The key difference between both histograms is ultimately the granularity of detail.

In the **10** bins histogram: It groups 256 possible intensity values into just 10 broad categories (with each bin covering ~25-26 intensity levels), giving a simplified and high level view of the distribution. It allows us to make the inference that the image is predominantly dark with a gradual decrease towards brighter values.

In the **256** bins histogram: Each individual intensity value is shown separately, revealing fine-grained details that the 10-bin version misses. Specifically, we can see a bimodal distribution with a large peak around intensity 20-50 (the dark train/background) and a distinct spike around intensity **200** (likely the bright train lights or signage).

While the **10-bin** histogram is useful for quick analysis and identifying general trends, the **256-bin** histogram provides the complete picture which is necessary for precise analysis like histogram equalization, where we may need to know exactly how many pixels exist at specific intensity level in order for proper redistribution across the 0-255 range. Since the range is small, it is still okay to have a histogram for each bin at small computational cost.

2.2.3 Part B – Histogram Equalization on Image

Since we are not using MATLAB, as part of cv2's docs, it comes with histogram equalization: https://docs.opencv.org/4.x/d5/daf/tutorial_py_histogram_equalization.html

```
# Use cv2's histogram equalization
P3 = cv2.equalizeHist(P)

# Redisplay the histograms for P3 with 10 and 256 bins
plot_intensity_histogram(P3, 10, (0, 255), "Histogram of Equalied MRT Image")
plot_intensity_histogram(P3, 256, (0, 255), "Histogram of Equalized MRT Image")
```

✓ 0.2s Python

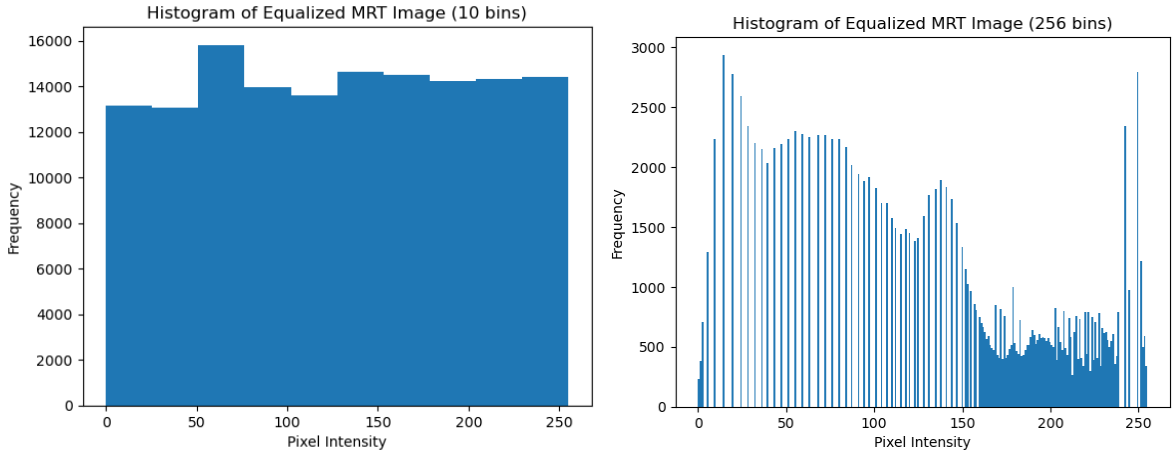


Figure 7: Histogram Plot after Equalization (10 bins and 256 bins)

In general, both histograms look more equalized than the original two image intensity histograms.

After equalization, the histograms show **partial equalization** rather than perfect uniformity. The **10-bin** histogram appears much more equalized, with frequencies relatively around **13,000** to **15,000** pixels per bin across most of the intensity range, which is close to the ideal uniform distribution. However, the **256-bin** histogram reveals that perfect equalization was not achieved. While the intensity values are now spread across the full 0-255 range, there is still an improvement compared to the original clustered distribution, there is significant variation with many spikes, gaps and bins with zero pixels.

Still, the key similarity is that both histograms show intensity values distributed across the entire range rather than concentrated in dark regions like the original intensity histogram. In the image, we can also see that there are lesser gaps in the brighter area of pixel intensities, causing the resulting image to look brighter than the original.

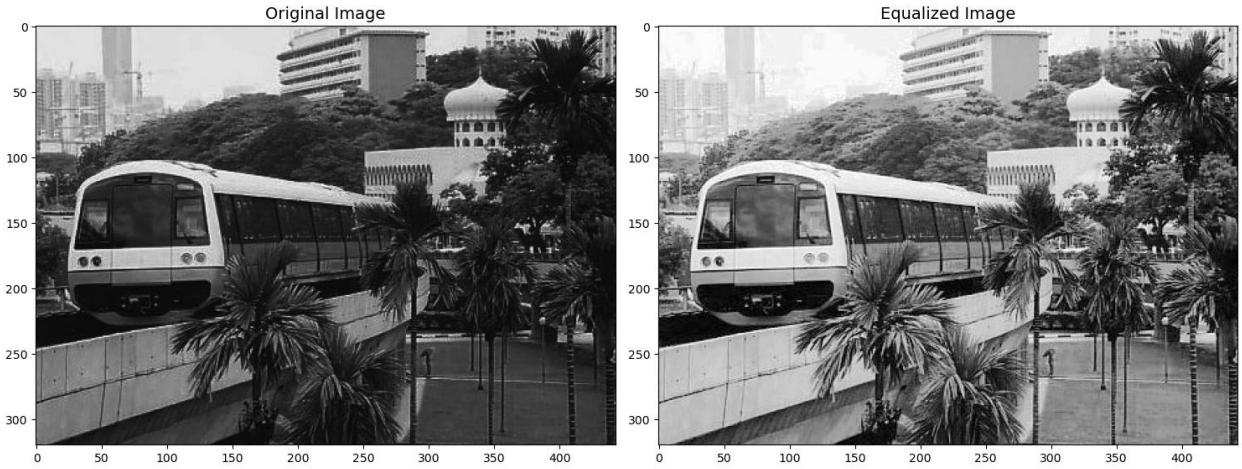


Figure 8: Side by side comparison (original image vs equalized image)

2.2.3 Part C – Rerunning of Histogram Equalization

In this section, we rerun the histogram equalization on the previous result obtained in Part B. **P3** was the result after histogram equalization.

```
# Re-equalize hist on the equalized image
P4 = cv2.equalizeHist(P3)
# Redisplay the histograms for P4 with 10 and 256 bins
plot_intensity_histogram(P4, 10, (0, 255), "Histogram of Equalied MRT Image")
plot_intensity_histogram(P4, 256, (0, 255), "Histogram of Equalized MRT Image")
```

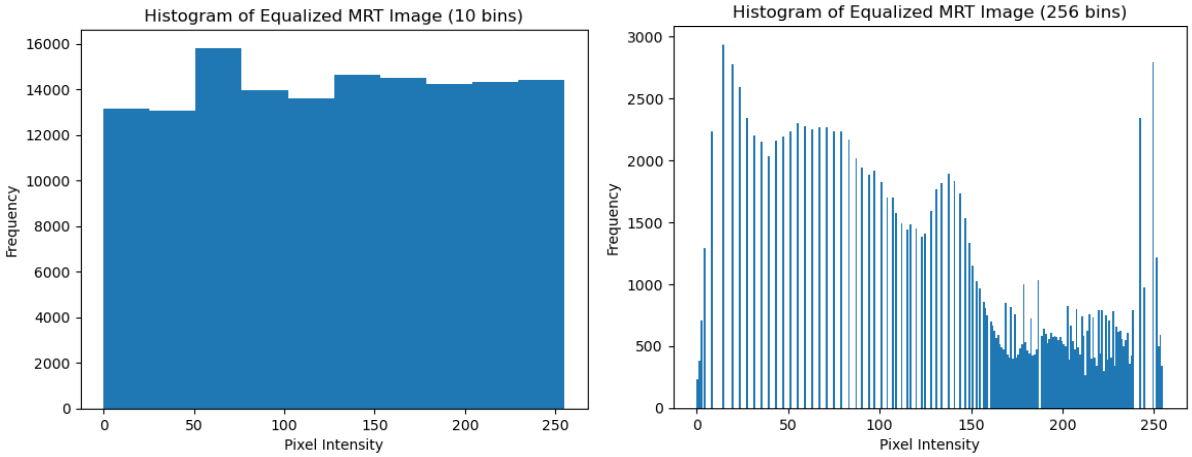


Figure 9: Histogram Equalization on already equalized images

From analysis, the histograms show virtually no change compared to the first equalizations, they look nearly identical. The histogram does not become more uniform after re-applying histogram. This occurs because histogram equalization is **idempotent** (applying it multiple time actually produces the same result as applying it once). Once pixel intensities have been redistributed to approximate a uniform distribution, the CDF of the equalized image is already approximately linear. If we reapply the transformation again, we are essentially applying a near-identity transformation that maps most pixels back to or close to their current intensity values. **This does not mean that there is no change to the distribution, there is in fact change. It is just minor changes.** This is partly due to the discrete nature of digital images, we cannot split bins that had multiple original intensity values merged, and we cannot fill empty bins where no pixels exist. Since the first equalization has already maximally spread the available intensities across the 0-255 range, subsequent equalizations cannot improve uniformity further. In the bottom Figure 10, we can also see virtually no change in the results, the images look close to identical to the naked eye.

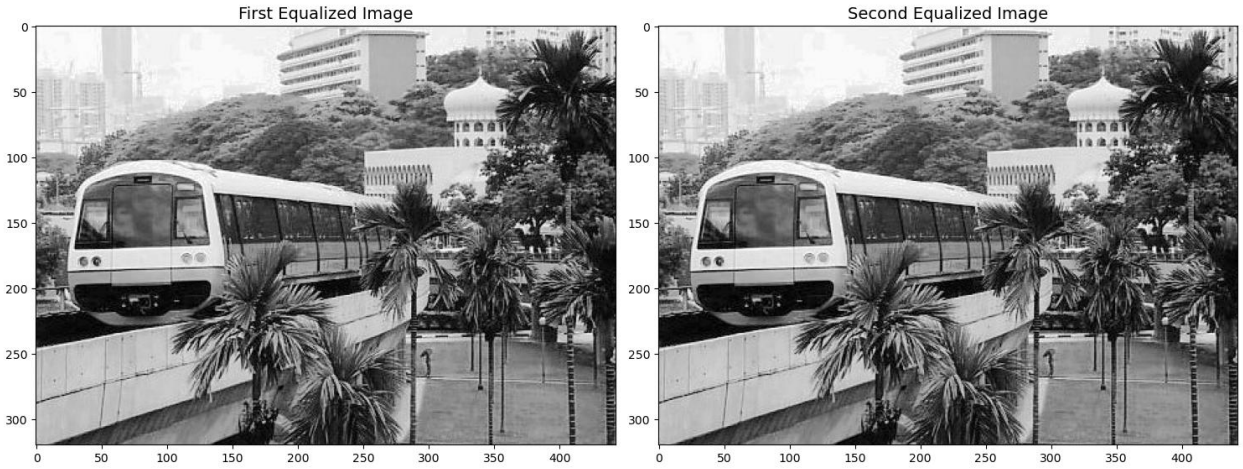


Figure 10: Side by side comparison (equalized image vs twice equalized image)

2.3 Linear Spatial Filtering

2.3.1 What is Linear Spatial Filtering?

In linear spatial filtering, typically an input image f(x, y) is convolved with a small spatial filter h(x,y) to create an output image g(x, y). The function h(x, y) can be considered to be the impulse response of the filter system. Another common name for h(x, y) is the point-spread function (PSF). This filtering operates in the spatial domain, altering the value of the current pixel based on the pixel's neighbors.

2.3.2 Part A – Generate Gaussian Filters with Normalization

```python
# Generate Gaussian filters using the formula: h(x,y) = (1/2πσ²) * e^(-(x²+y²)/2σ²)
# Size = dim of filter
# Sigma = how spread out the gaussian is , larger sigma = larger blur
def gaussian_filter(size, sigma):
    # Create coordinate grid centered at (0,0)
    # [-2, -1, 0, 1, 2] if size is 5
    axes = np.linspace(-(size//2), size // 2, size)
    # Create a mesh grid of the coordinates
    x, y = np.meshgrid(axes, axes)

    # formula: h(x,y) = (1/2πσ²) * e^(-(x²+y²)/2σ²)
    # First expression
    first_expn = 1.0 / (2.0 * np.pi * sigma**2.0)
    # Second expression
    second_expn = np.exp(-(x**2 + y**2) / (2.0 * sigma**2.0))
    # Combine
    gaussian = first_expn * second_expn
    # Normalize the gaussian
    return gaussian / gaussian.sum()

# Generate both filters
filter1 = gaussian_filter(5, 1.0)  # σ = 1.0, dim = 5
filter2 = gaussian_filter(5, 2.0)  # σ = 2.0, dim = 5

print("Filter 1 (σ=1.0):")
print(filter1, filter1.sum())
print("\nFilter 2 (σ=2.0):")
print(filter2, filter2.sum())
```

In the implementation above, we first create a coordinate grid centered at (0,0) using np.linspace based on the size of the kernel. We then create a mesh grid based on these axes to form the n by n matrix that is required of our Gaussian filter. Finally, following the formula for Gaussian represented by:

$$h(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (2)$$

We can then construct our filter and normalize it. After generation, the results are as follows:

```
Filter 1 (σ=1.0):
[[0.00296902 0.01330621 0.02193823 0.01330621 0.00296902]
 [0.01330621 0.0596343  0.09832033 0.0596343  0.01330621]
 [0.02193823 0.09832033 0.16210282 0.09832033 0.02193823]
 [0.01330621 0.0596343  0.09832033 0.0596343  0.01330621]
 [0.00296902 0.01330621 0.02193823 0.01330621 0.00296902]] 1.0

Filter 2 (σ=2.0):
[[0.02324684 0.03382395 0.03832756 0.03382395 0.02324684]
 [0.03382395 0.04921356 0.05576627 0.04921356 0.03382395]
 [0.03832756 0.05576627 0.06319146 0.05576627 0.03832756]
 [0.03382395 0.04921356 0.05576627 0.04921356 0.03382395]
 [0.02324684 0.03382395 0.03832756 0.03382395 0.02324684]] 1.0
```

Figure 11: Results of Gaussian Filters (σ = 1.0 and σ = 2.0)

A quick analysis shows that the filter is indeed working, with higher values in the middle of the mesh grid, are rotationally symmetric and follow the pattern. The sigma determines the weight on the center pixel, which is why Filter 1 has a larger value at the center than Filter 2. The sum of all values is also **1.0**, ensuring our implementation is accurate. The following code shows the implementation to plot the filters as meshes.

```
# Plotting of the gaussian filters as "3D meshes"
fig = plt.figure(figsize = (12, 6))
x,y = np.meshgrid(range(5), range(5))

# First subplot (σ=1.0)
ax1 = fig.add_subplot(1, 2, 1, projection = "3d")
ax1.set_title("3D Mesh for Gaussian Filter (σ=1.0)")
ax1.plot_surface(x, y, filter1, cmap = "plasma")

# Second subplot (σ=2.0)
ax2 = fig.add_subplot(1, 2, 2, projection = "3d")
ax2.set_title("3D Mesh forGaussian Filter (σ=2.0)")
ax2.plot_surface(x, y, filter2, cmap = "plasma")
plt.show()
```

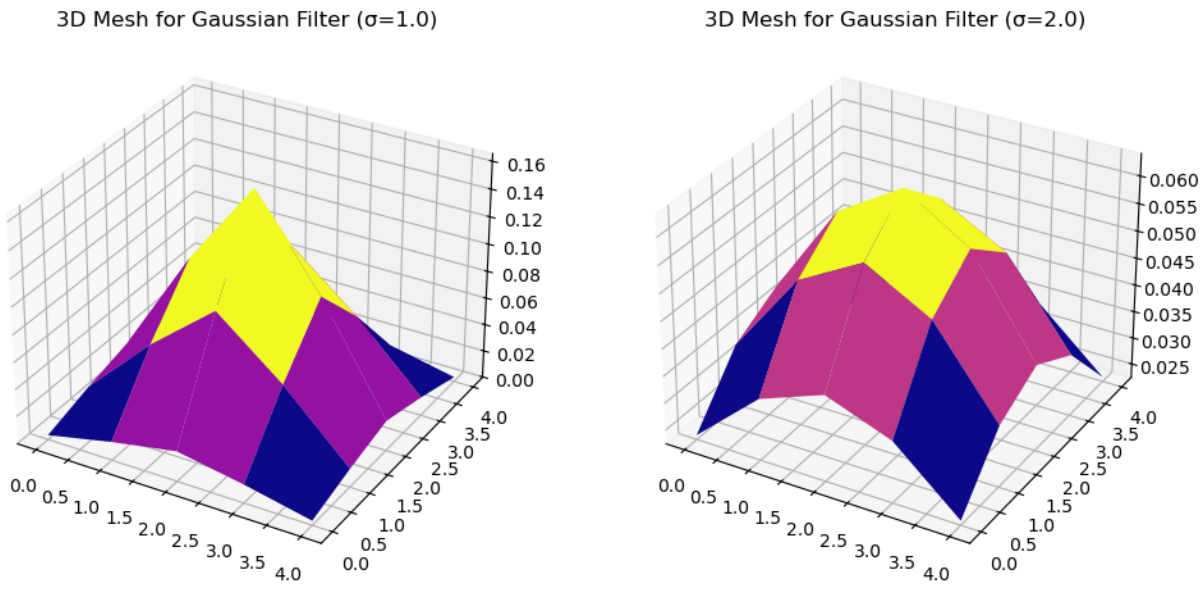


Figure 12: Gaussian Filters represented as meshes (σ = 1.0 and σ = 2.0)

The mesh for **σ=1.0** looks like a sharp mountain peak, forming a steep needle-like spike that shoots up to 0.16. It drops off rapidly as we move away from the center, where most of the mass is concentrated in the inner 3x3 region. In comparison for **σ=2.0**, we get a gentler hill forming a broad and more rounded dome with a lower peak, with much gradual slopes extending across the entire 5x5 area.

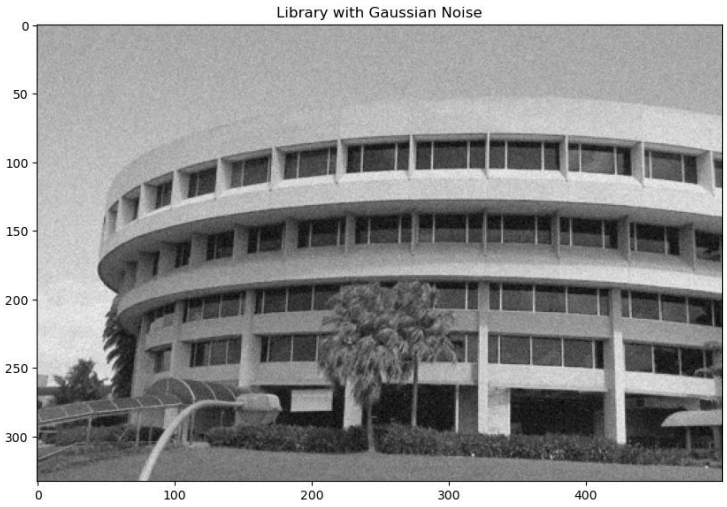
In image filtering, this means that after convolution,

For **σ=1.0**: The underlying output pixel is heavily influenced by the center pixel with 16% weight. It aims to keep the pixel mostly as is, just with some slight average with surrounding pixels.

For **σ=2.0**: The output pixels would just be a weighted average of the entire 5x5 region, with center pixel having just a tiny bit more weight. As a result, this creates stronger smoothing - mostly resulting in blurrier looking images.

2.3.2 Part B – Viewing of Image with Gaussian Noise

In this section, we reuse the helper functions described in previous sections to load and display the images.




```
Library with Gaussian Noise Information:
========================================
Property             Value
               Shape (333, 500)
           Data Type      uint8
         Total Pixels    166,500
           Dimensions          2
Memory Usage (bytes)    166,500
      Min Pixel Value          0
      Max Pixel Value        255
     Mean Pixel Value     131.83
   Standard Deviation      58.00
========================================
```

Figure 13: Library with Gaussian Noise and Information

In the original image, we observe it contains some noise (grainy textures, especially visible in the sky area), while fine details are preserved (window frames, tree leaves, etc.)

2.3.3 Part C – Image Filtering using Created Gaussian Filters

scipy provides functions for convolving 2d:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html

```
# Recreate the filters to ensure nothing was changed accidentally in previous cells
filter1 = gaussian_filter(5, 1.0)
filter2 = gaussian_filter(5, 2.0)

filtered_image_1 = scipy.signal.convolve2d(image_with_gaussian_noise, filter1, mode = "same")
filtered_image_2 = scipy.signal.convolve2d(image_with_gaussian_noise, filter2, mode = "same")

display_images_side_by_side(image_with_gaussian_noise, filtered_image_1, title1 = "Original Image", title2 = "Filtered Image (σ=1.0)")
display_images_side_by_side(image_with_gaussian_noise, filtered_image_2, title1 = "Original Image", title2 = "Filtered Image (σ=2.0)")
display_images_side_by_side(filtered_image_1, filtered_image_2, title1 = "Filtered Image (σ=1.0)", title2 = "Filtered Image (σ=2.0)")

print_image_info(filtered_image_1, "Filtered Image (σ=1.0)")
print_image_info(filtered_image_2, "Filtered Image (σ=2.0)")
```

The code just uses the previously mentioned function to generate two gaussian filters of **σ=1.0 and σ=2.0.** It then uses scipy's convolve2d to apply the filter onto the image, and then subsequently display the images side by side for comparison.

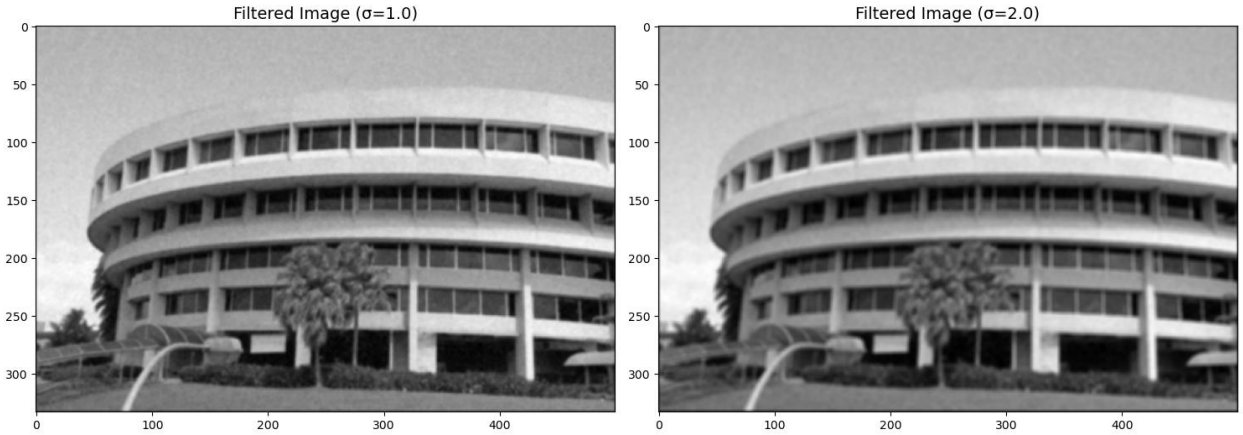


Figure 14: Filtered Image using Gaussian Filter (σ=1.0 and σ=2.0)

After filtering with the two filters:

σ = 1.0: A mild smoothing was applied to the image, reducing the noise while keeping most of the details and edges still visible. The image looks slightly softer, which is expected due to the averaging after the convolution. It is rather effective in preserving the details and should be used when prioritizing detail preservation over complete noise removal, at the cost of some blurriness.

σ = 2.0: A stronger smoothing was applied to the image, and more of the noise is observed to be removed. However, details in the scene like windows and tree-leaves are softer and blurrier. The entire image looks blurrier due to stronger averaging of the surrounding pixels with itself. This is effective when noise removal is a priority over detail preservation, at the cost of even more blurriness.

What are the trade-offs between using either of the two filters, or not filtering the image at all?

The filter with σ = 1.0 preserves more of the details of the images while not completing removing noise, while the filter with σ = 2.0 removes more noise at the cost of the loss of some details. If we do not filter, the noise is still present in the image, which reduces the visibility, but the original image's sharpness is preserved. Overall, it depends on our choice. If we want to:

1. **Remove more noise**: Use a **filter with higher σ** at the cost of blurrier images.
2. **Keep more details and remove some no**ise: Use a filter with **moderate σ like 1.0** to find the balance.

It is generally up to what the user wants to achieve with the filtering, and whether the prioritization is detail preservation or noise reduction.

2.3.3 Part D – Loading Image with Additive Speckle Noise

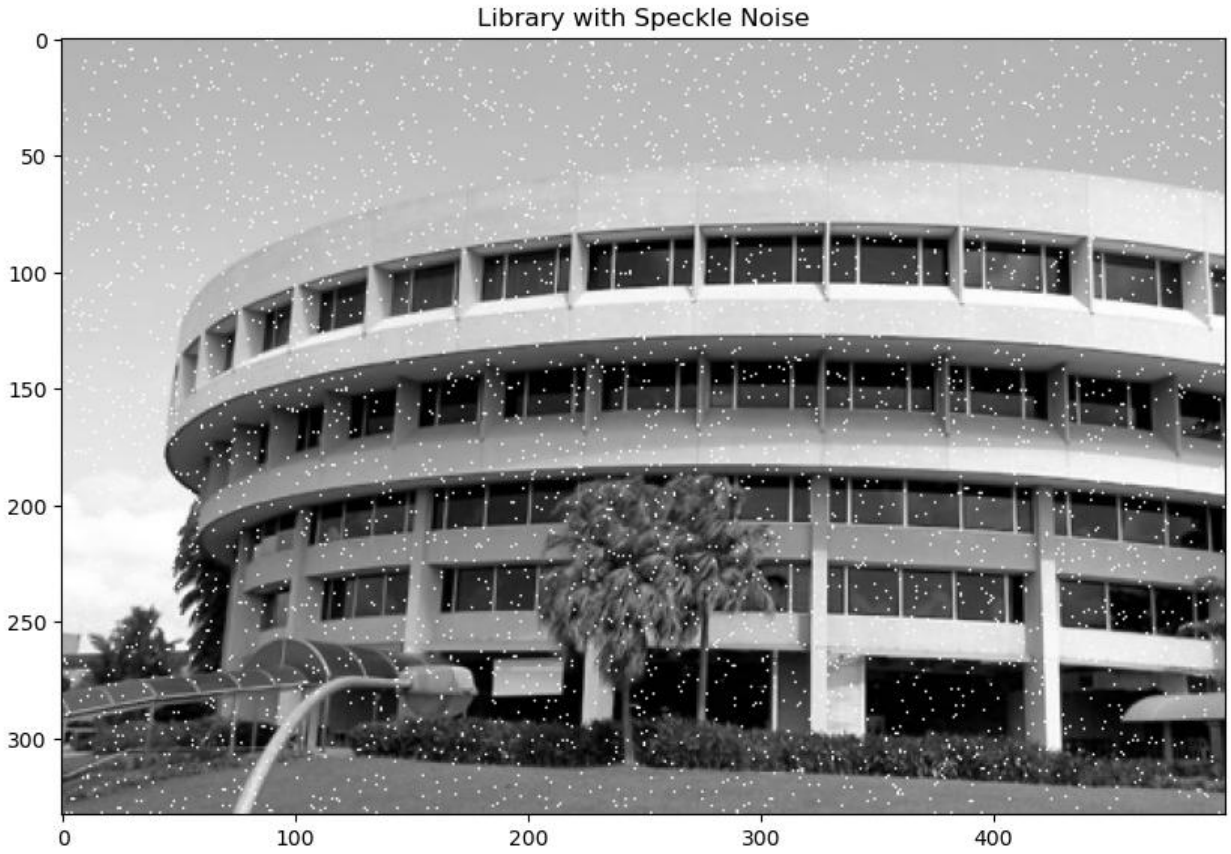


Figure 15: Image of library with speckle noise

From inspection, we see speckle noise is present in the image, which looks a little like salt and pepper noise. There are white dots all over the image. We utilize the same method as in Part C and apply the Gaussian filters onto this image with speckle noise.

2.3.3 Part E – Applying Gaussian Filter on Image with Additive Speckle Noise

We apply Gaussian filters with two different standard deviations (σ=1.0 and σ=2.0) to evaluate their effectiveness in removing speckle noise

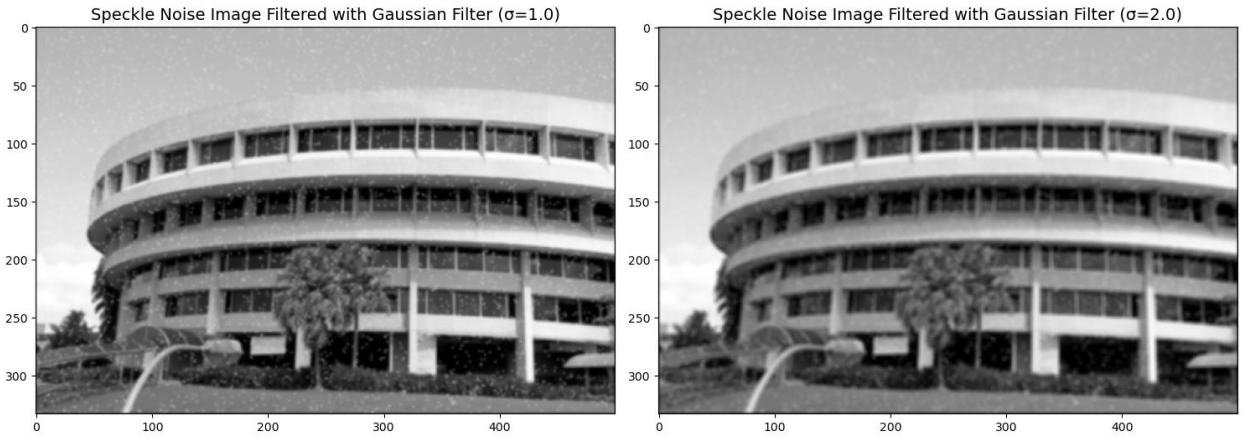


Figure 16: Speckle Noise Image Filtered with Gaussian Filters (σ=1.0 and σ=2.0)

After filtering with the two filters:

σ = 1.0: A mild smoothing was applied to the image, reducing the noise while keeping most of the details and edges still visible. The image looks slightly softer, which is expected due to the averaging after the convolution. The filter is **less effective in removing Speckle noise**, as Gaussian filters do not typically handle this kind of noise.

σ = 2.0: A stronger smoothing is applied, and more of the noise is observed to be removed. However, details in the scene like windows and tree-leaves are softer and blurrier. The entire image looks blurrier due to stronger averaging of the surrounding pixels with itself. It seems a bit more effective, but noise can still be seen, as it is not Gaussian noise.

Overall, Gaussian filters are not suitable for removing speckle noise due to the fundamental mismatch between the characteristics of the filter and the noise model. Speckle noise can appear at all frequency scales with random granular patterns, while Gaussian filters assumes and accounts for local pixel correlation, which may not be the case in speckle noise. In general, **median filters** are much more suited for Speckle noise.

2.4 Median Filtering

2.4.1 What is Median Filtering?

Median filtering is a special case of order-statistic filtering. For each pixel, the set of intensities of neighboring pixels (in a neighborhood of specified size) are ordered. Median filtering involves replacing the target pixel with the median pixel intensity.

2.4.2 Applying Median Filtering to Images with Interference (3x3 and 5x5)

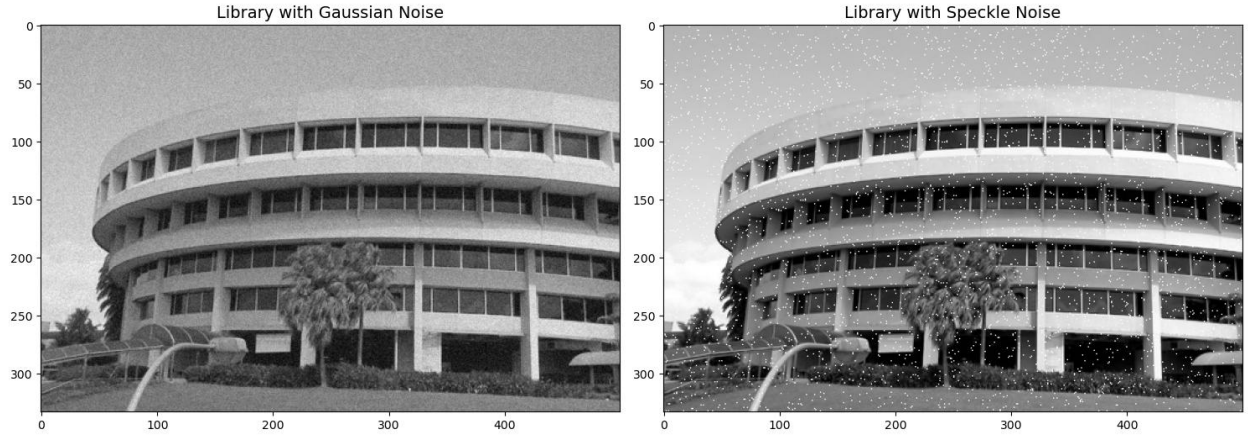


Figure 17: Original images with Gaussian Noise (left), Speckle Noise (right)

scipy also provides a median filter function, which we can apply to both our images with differing kernel sizes listed in the manual. (3x3, 5x5)

```
# Create median filters using scipy
median_filter_gaussian_3x3 = scipy.signal.medfilt2d(image_with_gaussian_noise, kernel_size = 3)
median_filter_gaussian_5x5 = scipy.signal.medfilt2d(image_with_gaussian_noise, kernel_size = 5)
median_filter_speckle_3x3 = scipy.signal.medfilt2d(image_with_speckle_noise, kernel_size = 3)
median_filter_speckle_5x5 = scipy.signal.medfilt2d(image_with_speckle_noise, kernel_size = 5)
```

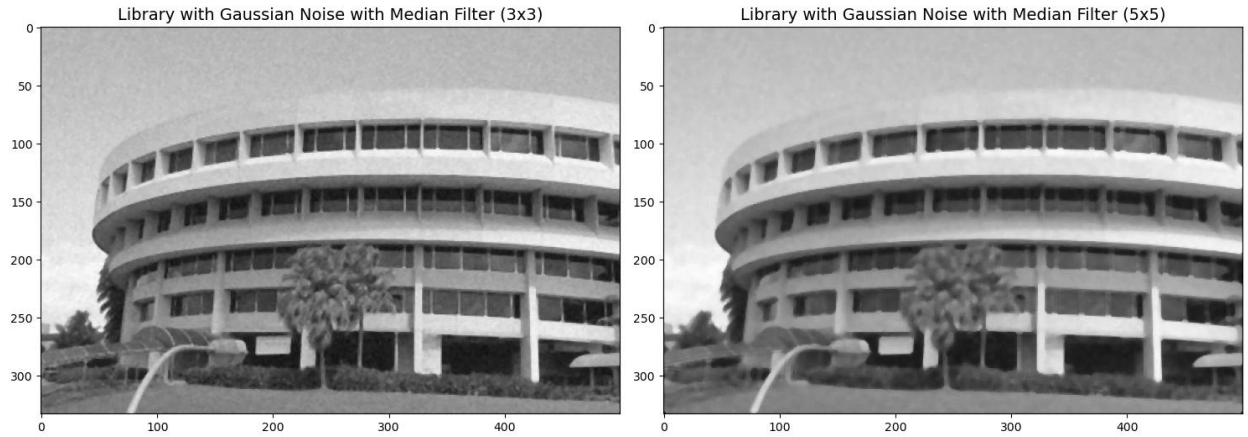


Figure 18: Image with Gaussian Noise with Median Filter (3x3 and 5x5)

On the image with Gaussian Noise, median filtering seems to be less effective, as median filters are excellent for salt-and-pepper type noise, rather than noise that follows a continuous distribution like Gaussian. Overall, Gaussian filters are still better at removing noise. On the plus side, median filters still preserves the edges and still provide a nice result, just it is not as effective as Gaussian filters for this type of noise.

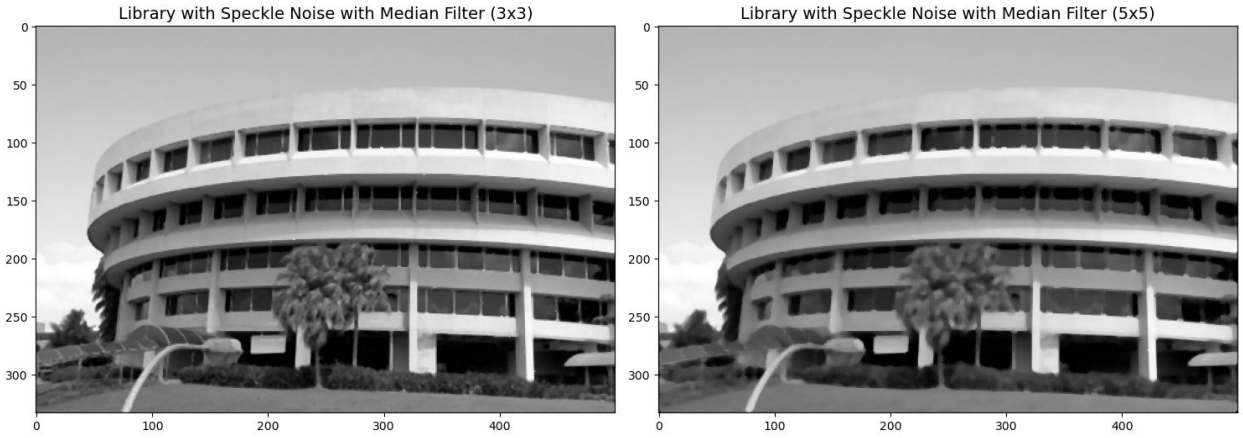


Figure 19: Image with Speckle Noise with Median Filter (3x3 and 5x5)

On the image with Speckle Noise, Median filtering is excellent at noise removal, and the crisp edges seem to be preserved while keeping structural details intact. By taking the median, we effectively reduce the impacts of isolated, high intensity pixels, making it great to deal with speckle noise. It is better than Gaussian filters in this sense.

What are the tradeoffs?

Median filtering is excellent for salt and pepper noise like speckles, while preserving the edges and structural details of the image. On the downside, it is less effective on common noise like Gaussian and provides an even larger blur effect than Gaussian with larger kernels.

Overall - isolated noise == median filtering, continuous noise throughout image == gaussian filtering.

2.5 Suppressing Noise Interference Patterns

2.5.1 What is Interference?

In images in videos, noise interference patterns like parallel, diagonal, horizontal and vertical lines can appear due to poor signal receptions such as in traditional television screens. As a result, such interference patterns heavily corrupt the image quality and cause several details in the image to be obscured, preventing legibility or human perception.

One technique to remove such interference is through Fourier Transforms, where alterations to the underlying image is done in its Fourier Transform representation in the frequency domain. In this frequency domain, each pixel value represents a particular amplitude of a specific frequency found in the image. This creates a sinusoidal wave pattern. This pixel value in the Fourier Transform also determines the contribution of that frequency in the resulting image, and the image is formed as the sum of these frequencies.

2.5.2 Part A – Loading of Image with Interference

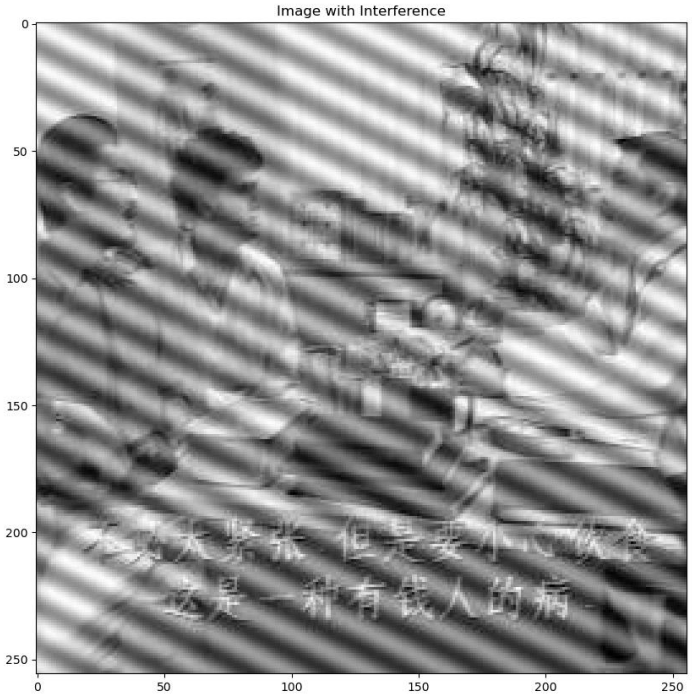


Figure 20: Image with interference, diagonal lines

In the image, there are multiple diagonal lines spread across the image, making it hard to conduct processing in the spatial domain to remove such noise. In this case, we must swap to the frequency domain and remove such noise.

2.5.3 Part B – Obtaining the Fourier Transform and Power Spectrum

numpy provides Fourier transform functions in its fft module. We can use that to conduct our analysis.

```
def show_power_spectrum(image, do_shifting = True):
    # Calculate the fourier transform of the image
    F = np.fft.fft2(image)
    # Compute the power spectrum
    S = np.abs(F)**2
    # Shift the zero frequency component to the center of the spectrum
    S_shifted = np.fft.fftshift(S ** 0.1) if do_shifting else S ** 0.1
    # Handle the plotting
    plt.figure(figsize = (10, 8))
    plt.imshow(S_shifted)
    plt.colorbar()
    plt.show()
```

In the function, we first calculate the Fourier transform of the image which returns complex values. We then compute the magnitude spectrum using np.abs(F) on the Fourier transform. To compute the power spectrum, we take the square of the magnitude. Optionally, we can also do shifting to shift the origin of the Fourier transform to the centre of the image.

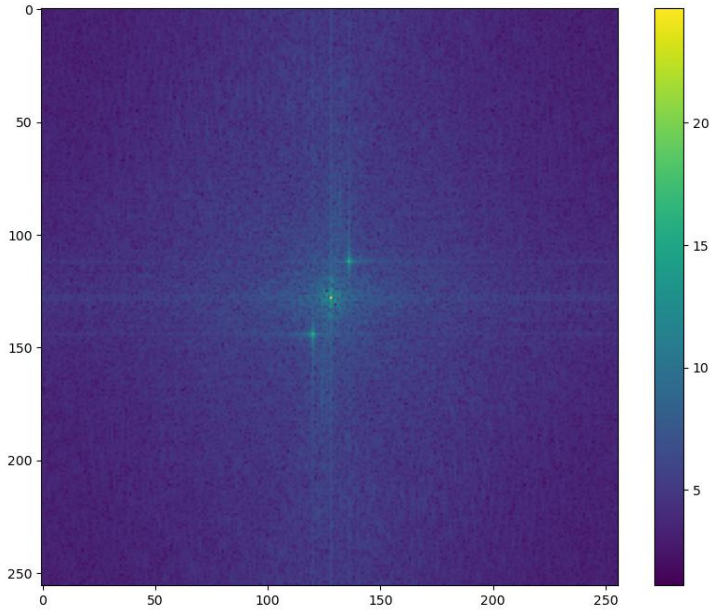


Figure 21: Power Spectrum of the Image with Interference

General Observations:

1. The center of the image is the DC component, shown by the bright green dot in the middle. It also represents the average brightness of the image and is always the brightest point in the power spectrum. Due to fftshift, it is in the middle of the image.
2. We also observe two symmetric peaks. and this tells us important details of the image:
    - The image contains a **periodic/repetitive** pattern
    - This pattern appears at some **specific frequency and direction**
    - It also symmetric due to the fundamental property that Fourier transforms of real images are conjugate symmetric. F(u,v) = F*(-u,-v).

Based on how far a point it is from the center, the further away it is from the center, the higher the frequency (causing fine and dense patterns), while close to center means low frequency (with coarse and sparse patterns). We observe the two peaks also form a diagonal line in the frequency domain, which indicates the presence of diagonal stripes at that specific frequency in the spatial domain. The positions of the peak tell us both the frequency (the spacing between stripes) and the orientation (diagonal) of the interference pattern.

2.5.3 Part C – Display Power Spectrum without fftshift

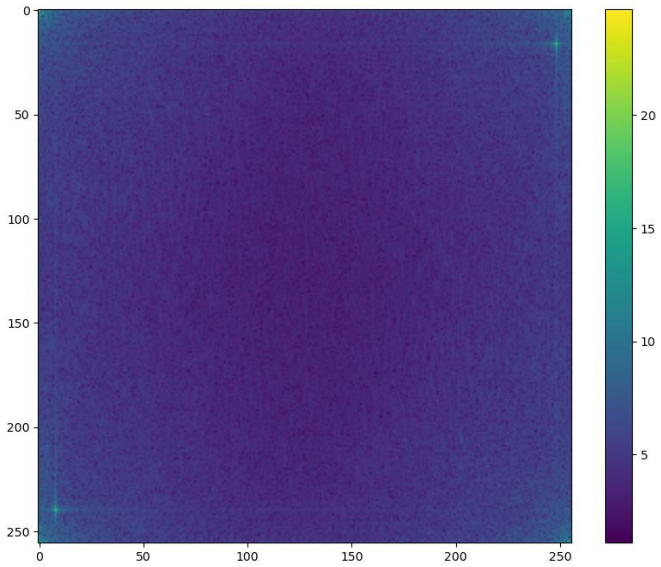


Figure 22: Power Spectrum of the Image with Interference (no fftshift)

To automate the process and automatically find peaks, a helper function listed below was used. In summary, it computes the Fourier transform, and the power spectrum, and then finds the top maximum brightness locations in the image using a maximum filter. We then save these values, depending on whether shifting was enabled. Since the DC is the brightest point, it will always be the first element in the array as the brightest point. However, in certain cases, we do not want to override the DC. Hence, we may choose to ignore it such that the image integrity is maintained, as DC is linked to the image's overall brightness.

```
def find_peaks_in_power_spectrum(image, num_peaks=3, do_shifting = False, ignore_dc = True):
    # Compute power spectrum
    F = np.fft.fft2(image)
    S = np.abs(F) ** 2
    S_shifted = np.fft.fftshift(S ** 0.1) if do_shifting else S ** 0.1

    # Find local maxima
    # Apply maximum filter to find local peaks
    local_max = scipy.ndimage.maximum_filter(S_shifted, size=20) == S_shifted

    # Get peak locations
    peak_coords = np.argwhere(local_max)
    peak_values = S_shifted[local_max]

    # Sort by intensity and get top N peaks
    # Ignore dc as its the brightest if enabled
    sorted_indices = np.argsort(peak_values)[::-1] if not ignore_dc else np.argsort(peak_values)[::-1][1:]
    top_peaks = peak_coords[sorted_indices[:num_peaks]]
```

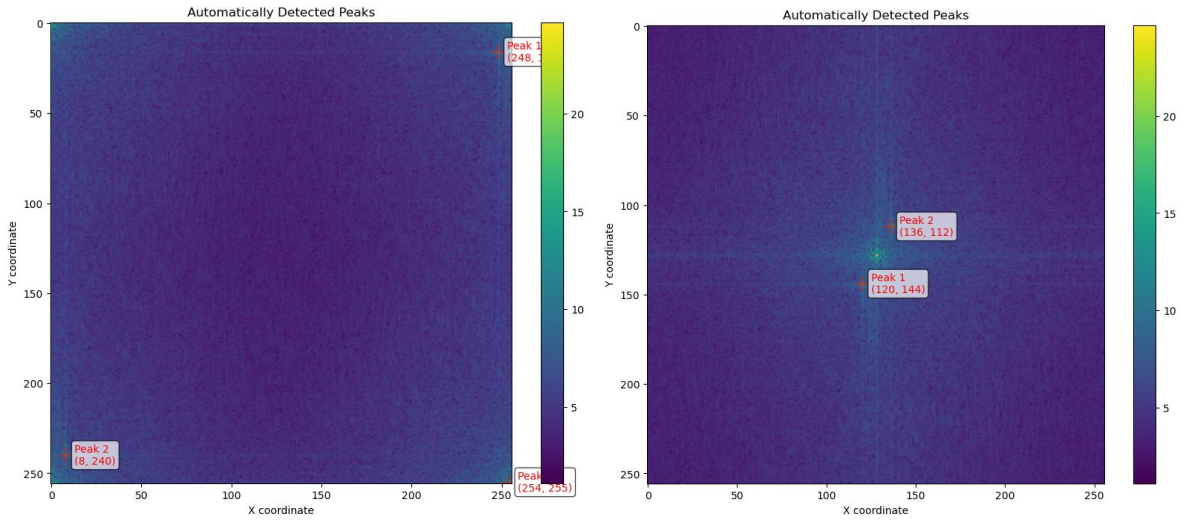


Figure 23: Top 2 Peaks in Power Spectrum (left – non shifted, right – shifted)

From the function, we are able to automatically measure the locations of peaks, and they are represented by the following result. We can then use these information to conduct image processing.

```
Detected Peak Coordinates:
Peak 1: x=120, y=144 | Distance from center: 17.89
Peak 2: x=136, y=112 | Distance from center: 17.89
```

2.5.4 Part D – Zeroing out Neighbor Elements at Identified Peaks

In this section, we zero out the 5x5 neighborhood elements at the locations corresponding to the peaks in the previously found Fourier Transform F. After zero-ing out at those locations, we acquire the following power spectrum visualizations. The “black” rectangles are the 5x5 that were set to 0.

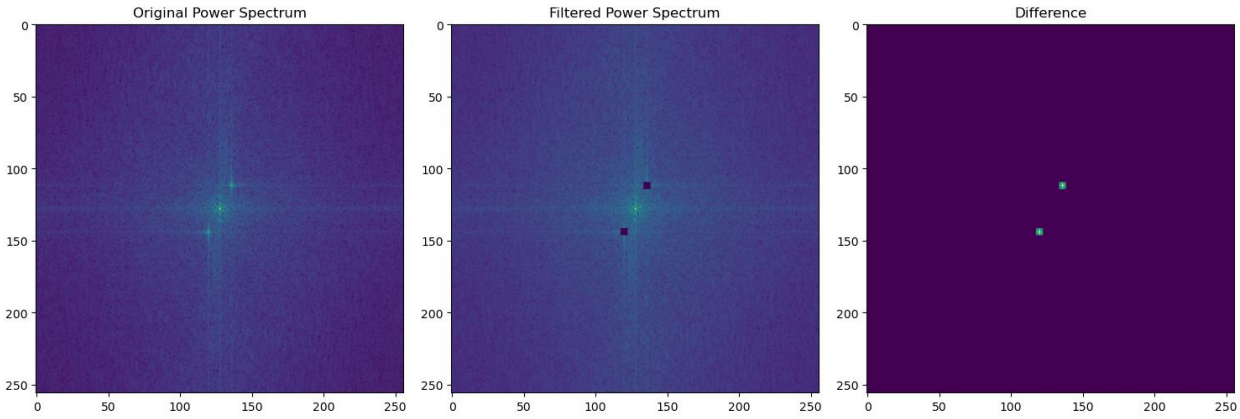


Figure 24: Original Power Spectrum, Filtered Power Spectrum, Difference between both (respectively)

2.5.5 Part E – Performing Inverse Fourier Transform to Reconstruct Image with Interference

In this section, we compute the inverse fourier transform using ifft2 and display the resultant image. We use a simple function using np's fft module, and then reconstruct the image by converting it back to uint8.

```
def inverse_fourier_transform(F):
    # Then use inverse fourier transform to reconstruct the image
    inverseft = np.fft.ifft2(F)
    # Convert back to uint8
    image_reconstructed = np.abs(inverseft).astype(np.uint8)
    return inverseft, image_reconstructed
✓ 0.0s
```

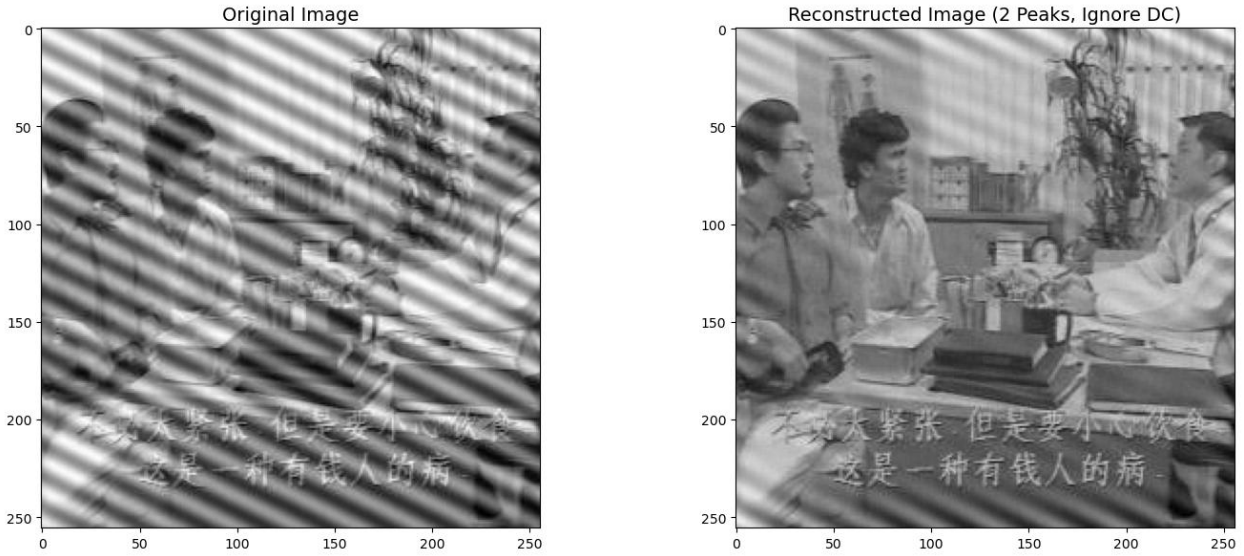


Figure 25: Original Image vs Reconstructed Image after 5x5 zeroing

Setting the neighboring pixels around the peaks to 0 helped to reduce a lot of the noise. It is clearer in the right view that there are people sitting at a table, with background details and the room structure. In this case, it may be acceptable for some to use this image as is for its intended purpose. The Chinese text is also now legible and clear. This shows how Fourier Transform is powerful, especially if the image to be recovered was for historical purposes. However, it may be possible that the image may appear less sharp than it would be without interference. Some fine details may be smoothed out as legitimate high-frequency content may have overlapped with the interference frequencies, which is unavoidable loss.

2.5.6 Part E(2) – Improving the Results

2.5.6.1 Increasing Neighbourhood Zeroing to 15x15

In the original power spectrum around the peaks, we observe that the surrounding area of the peaks is also bright, which may mean that the surrounding area also adds some contribution to the interference to the image. To test this theory, we increase the neighborhood zero-ing from **5x5** to **15x15.**

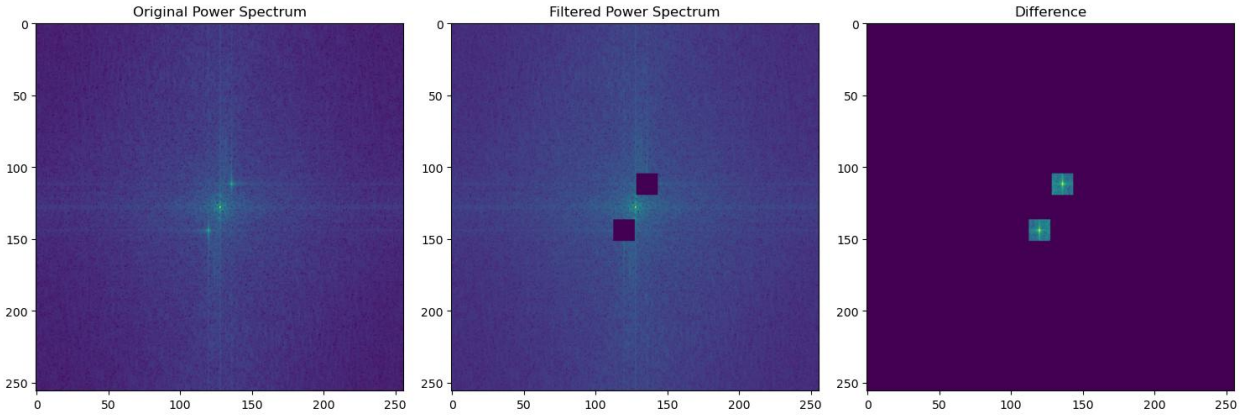


Figure 24: Original Power Spectrum, Filtered Power Spectrum **(15x15),** Difference

Figure 25: Original Image vs 5x5 zeroing vs 15x15 zeroing

From visual inspection, between the images of using 5x5 zeroing and 15x15 zeroing, we see way lesser noise in the 15x15 zeroing version. This means that it was semi-successful at removing noise in the picture, but we can still observe the presence of lines. This means that the surrounding area does contribute to the effect, but it is not all of it. We need to re-analyze the spectrum again.

2.5.6.2 Setting Entire Cross on Peak to Zero

After more detailed analysis of the power spectrum, there seems to be a vertical and horizontal line spreading across the entire x and y axis at each peak. This suggests that that may be contributing to the overall interference. To test the theory, we set the neighborhood pixels of 5x5

to be zero, as well as the cross through the entire power spectrum. The function for this can be found in the source, as it is too long to put it in the document.

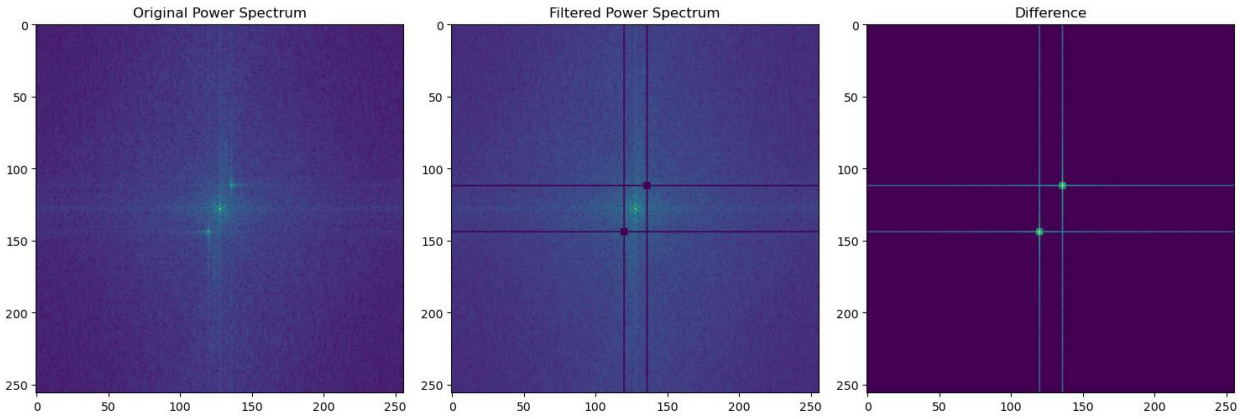


Figure 25: Original vs 5x5 + Cross zeroing vs Difference

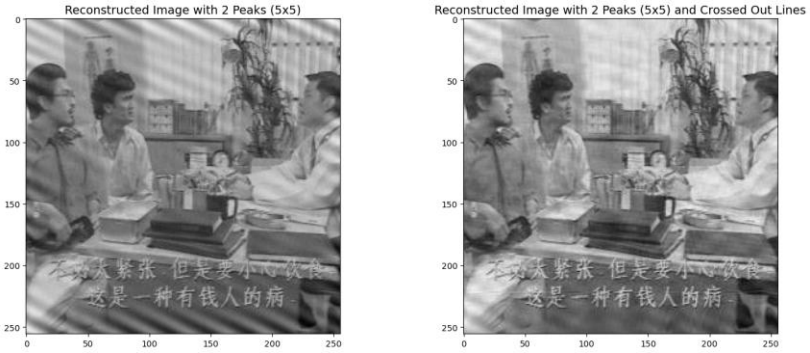


Figure 26: 5x5 zeroing vs 5x5 + Cross zeroing.

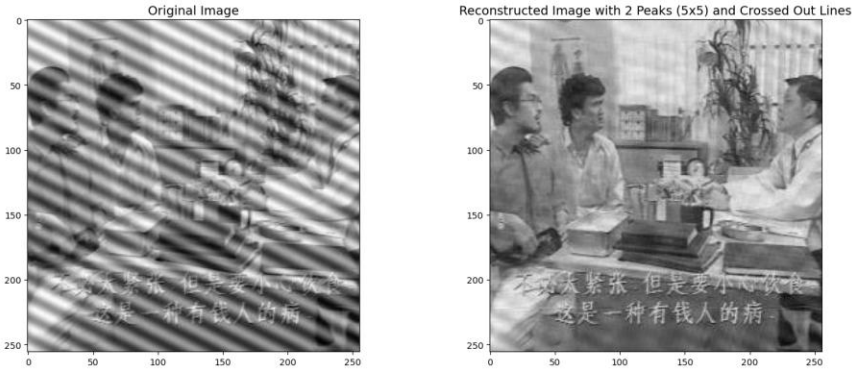


Figure 27: Original vs 5x5 + Cross zeroing.

From visual inspection, it seems that clearing the cross on top of the neighboring 5x5 pixels benefits the image more. This shows that our initial analysis of increasing the neighborhood size had only reduced the interference slightly but helped us gain the insight to cross out the entire lines instead. The resulting image compared to the original image shows a huge difference, highlighting our improvements were substantial.

2.5.7 Part F – Freeing the Primate by Filtering out the Fence

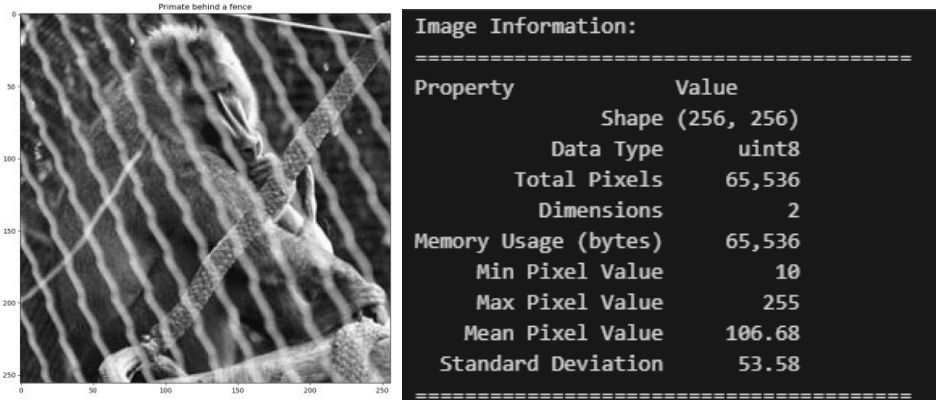


Figure 28: Image of Primate behind Fence and Information

In the picture, the fence can be seen as patterns among the image, and we need to access the image's fourier transform and power spectrum to see how we are able to remove the fence on the image. It would be hard to use processing in the spatial domain to handle this task.

2.5.7.1 Acquiring the Power Spectrum

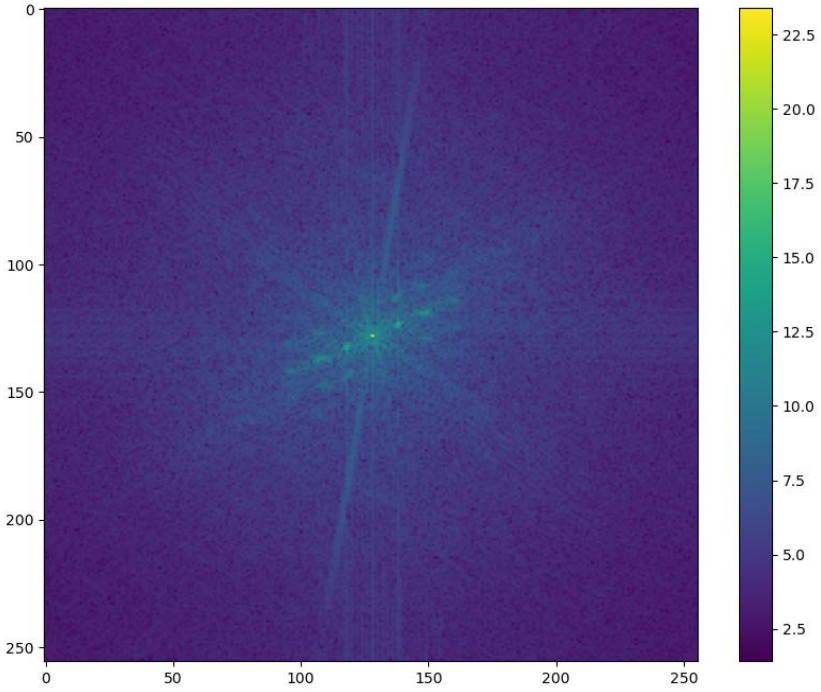
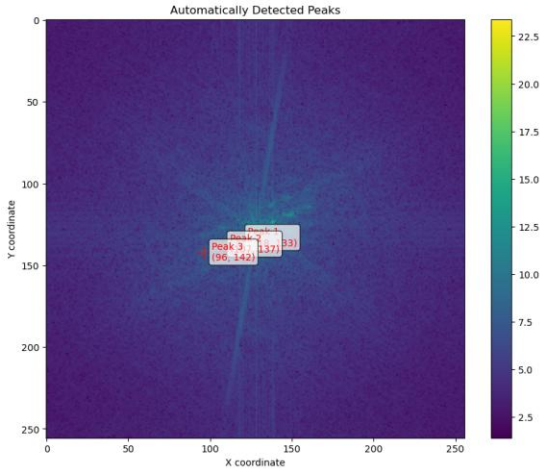


Figure 29: Power Spectrum of Primate Image

In the power spectrum, we notice a few pixels with high intensity that follow the diagonal pattern we see before. We can see patterns of it repeating down the y-axis, which highly indicates that these intensities belong to the fence itself. Knowing this information, we should be able to find some function to clear those lines in nature, setting them to zero along these lines would mean successful free-ing of the primate in essence.

2.5.7.2 Finding the Peaks in Power Spectrum

We reuse the peak function to find peaks in our image. Since the fence seems to be bright, it will definitely show in the power spectrum. In the original image, the fences also seem to go in a diagonal direction and are relatively bright colors.




```
Detected Peak Coordinates:
Peak 1: x=118, y=133 | Distance from center: 11.18
Peak 2: x=107, y=137 | Distance from center: 22.85
Peak 3: x=96, y=142 | Distance from center: 34.93
```

Figure 30: Identified peaks of Primate Image

After visualizing the peaks, we realize that these so-called bright lines sort of form a line, if we can find a line that roughly fits this in the Fourier transform, we can repeat those lines with some step in the Fourier transform space and clear the fence. This is a form of periodic interference.

2.5.7.3 Applying a Diagonal Notch Filter

In this section, a diagonal notch filter function was created, the source code can be found in the code as it is rather long. The main idea is – we create a notch filter that finds the direction of the line formed by these peaks. Then, along that line, we set all pixels to 0. Since the fence pattern is often more than one, the function also allows repeating of this line to hopefully get rid of the "noise" incurred by these fences.

```
F_filtered_primate, S_filtered_display_primate, S_original_display_primate = apply_diagonal_notch_filter(
    primate_image,
    peaks,
    line_thickness=5,  # Adjust to control thickness
    do_shifting=True,
    num_harmonics=0,
    dc_radius=0.5
)
```

The function takes in the thickness of the line, whether to shift the Fourier transform for visualization, as well as the "harmonics" to see how many times the line repeats. Protection of the DC is also included, as the lien might intersect the middle of the image, so we protect the dc with some radius 0.5.

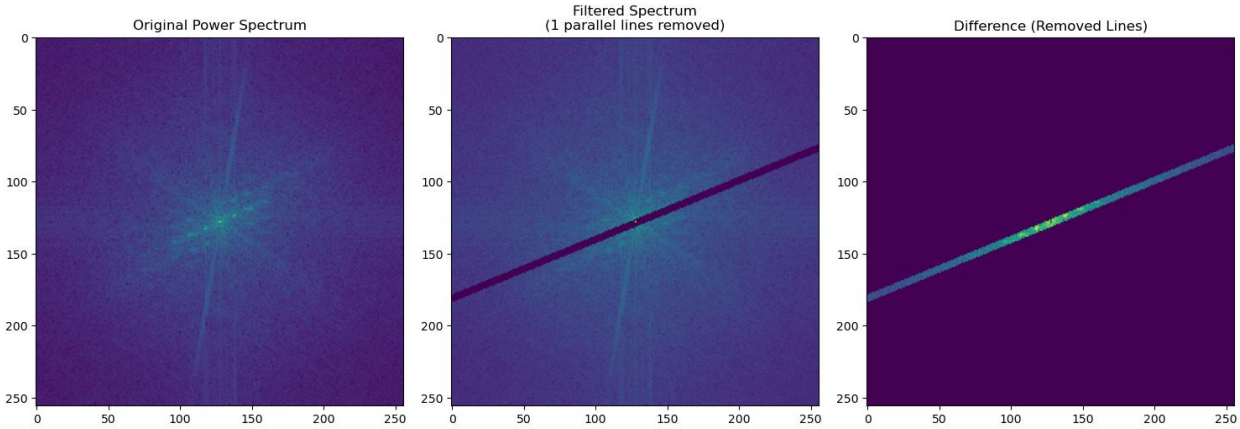


Figure 31: Removing Lines along the direction of peak (harmonics = 0, line thickness = 5, **DC radius = 0.5**)

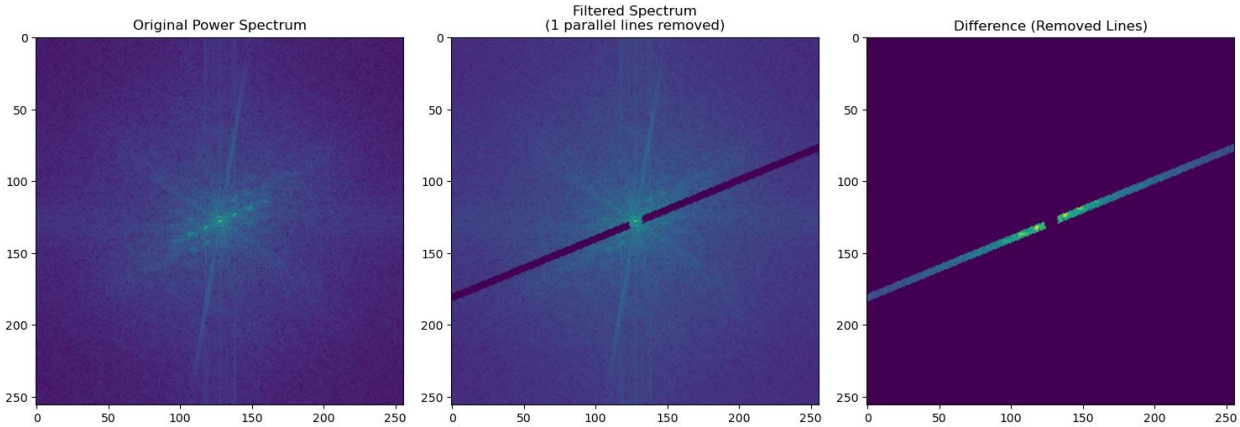


Figure 32: Removing Lines along the direction of peak (harmonics = 0, line thickness = 5, **DC radius = 5)**

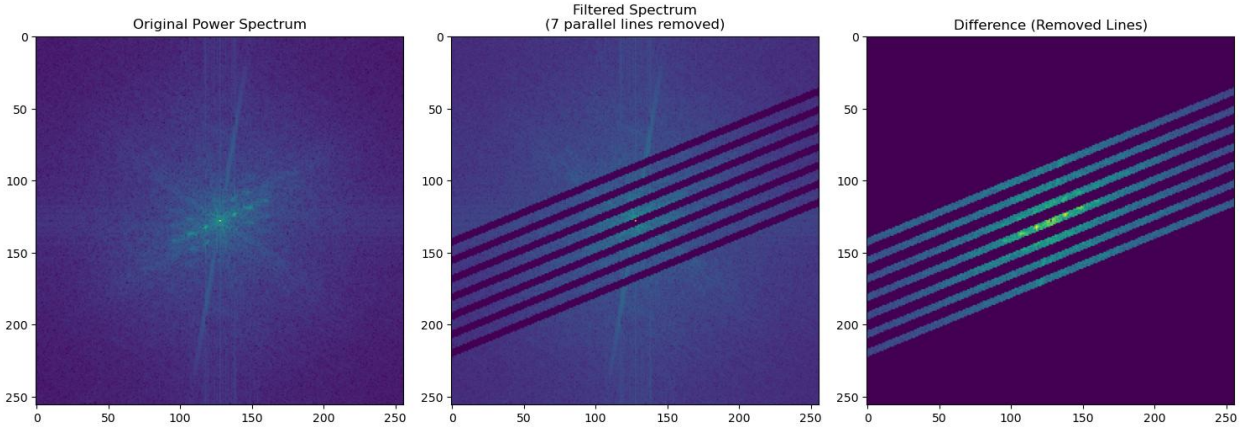


Figure 33: Removing Lines along the direction of peak (harmonics = 3, line thickness = 5, **DC radius = 0.5**)

From our filter, we can see that our filter has covered roughly the pattern of diagonals along the power spectrum, which should make the fence go away in general.

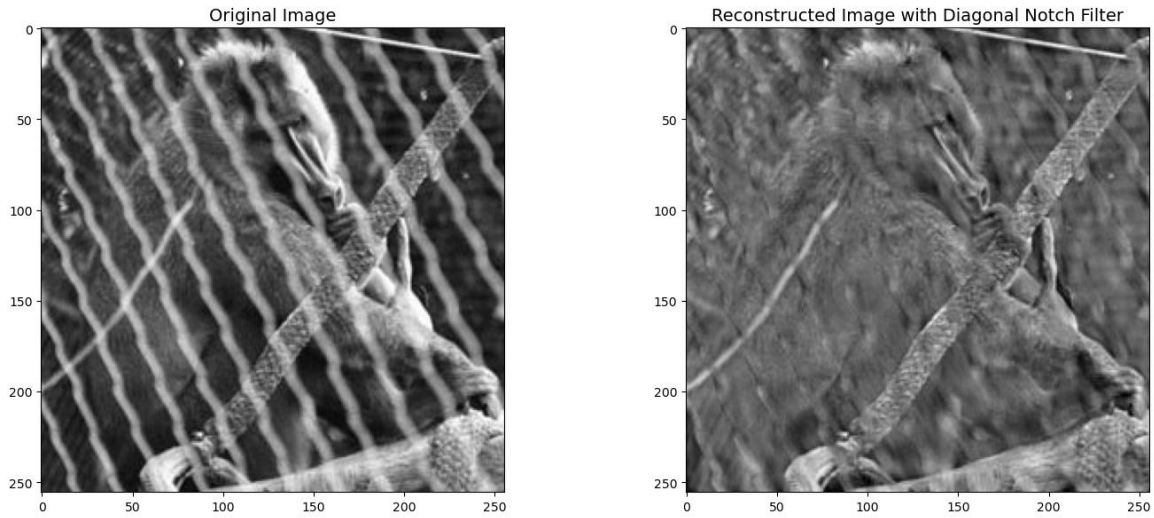


Figure 31: Reconstructed Image with Diagonal Notch Filter (harmonics = 3, line thickness = 5)

In this experiment, we can see that our filter is quite good at removing the fence while maintaining the structural integrity of the primate. The image looks blurrier, and contrast is lost. This result is expected as we are naively removing the entire diagonal along the image, and we may have removed other frequencies that were overlapping in the removed region, so loss is inevitable and expected. Perhaps, with other non-naïve methods, we will be able to remove the fence while keeping the colors intact.

2.5.7.4 Ablation – Test with Different Line Thickness

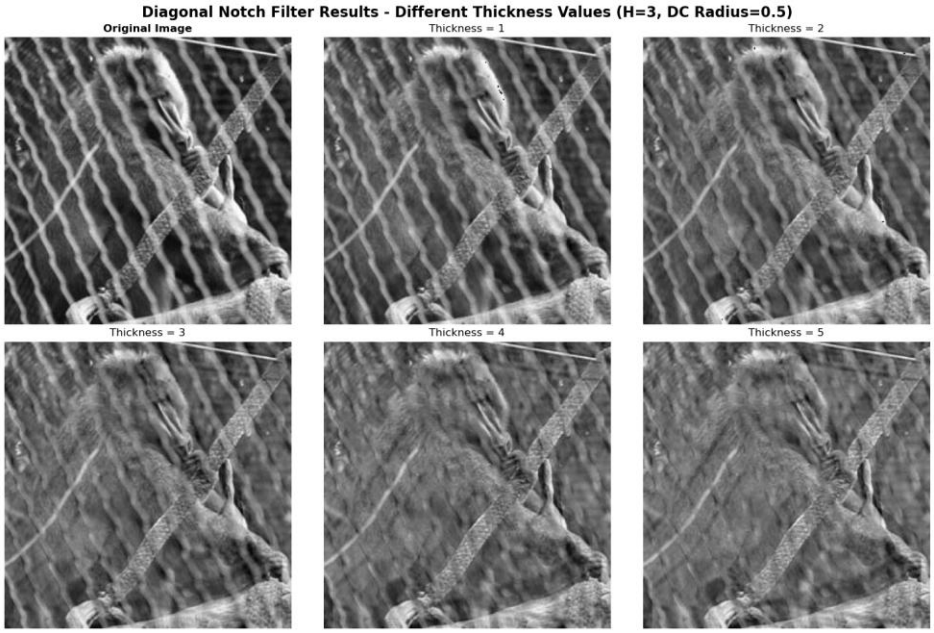


Figure 32: Ablation – Reconstruction with different diagonal line thickness in notch filter

From my proposed method, it seems that the best working thickness of 4, with 3 repeated harmonics and DC protection radius of 0.5 works the best. Although the image looks relatively greyish, one can still make the features out of an animal, and the fence is still visible, but not as visible as before. DC protection was needed to prevent the image from becoming too dark as setting it to 0 would cause the mean intensity of the image to be 0. Regardless, although the result was not clean, its the best we can do at this point of time with just simple notch filtering. It is hard to patch back parts that are obscured initially by the fence without some form of informed analysis of the image.

2.6 Undoing Perspective Distortion of Planar Surface

2.6.1 Part A - Displaying of Book Image


```
Image Information:
========================================
Property             Value
                Shape (240, 320, 3)
           Data Type         uint8
         Total Pixels       230,400
           Dimensions             3
Memory Usage (bytes)       230,400
      Min Pixel Value             0
      Max Pixel Value           255
     Mean Pixel Value        145.26
   Standard Deviation         48.46
========================================
```

Figure 33: Picture of Book Image and Information

2.6.2 Part B – Getting Corner Coordinates of Book

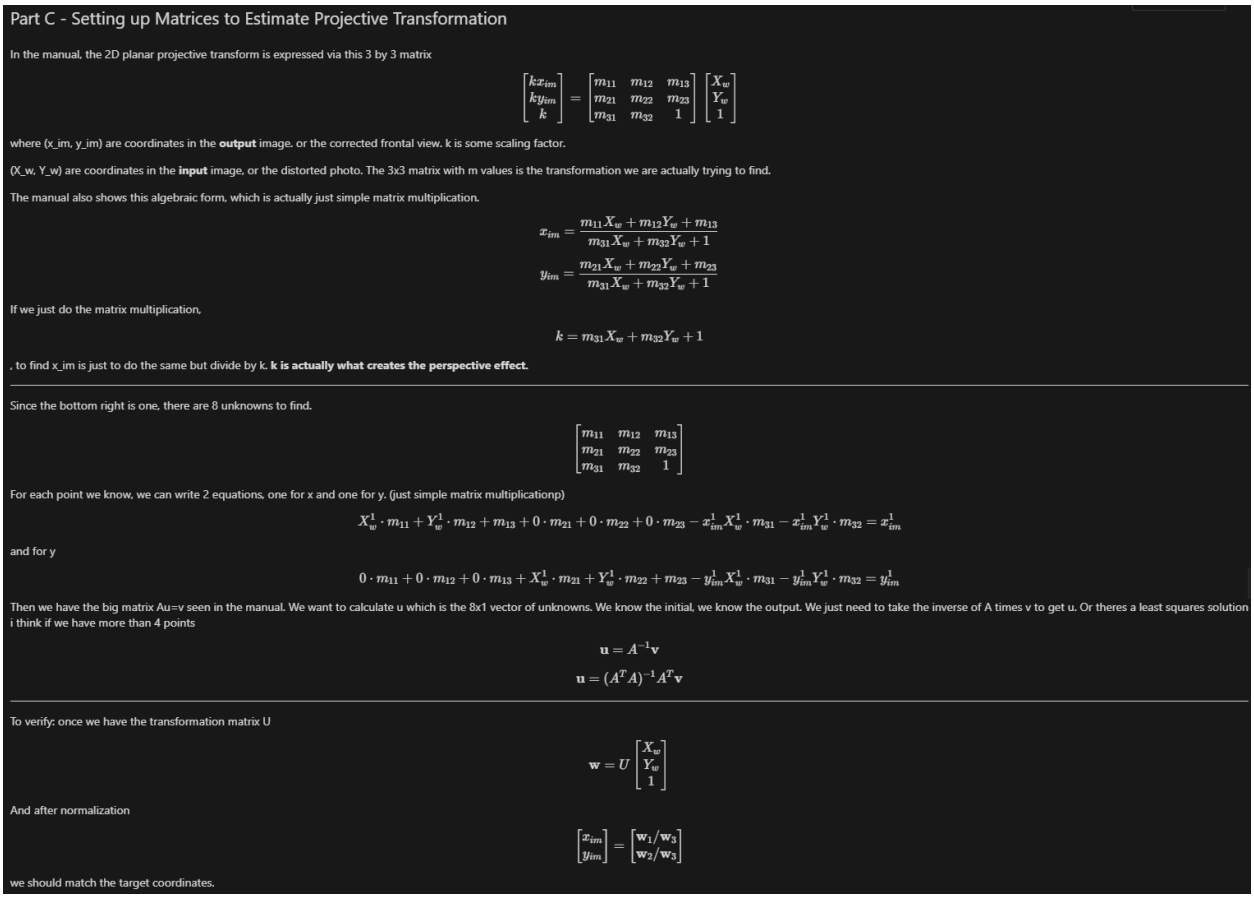
In this section, I did not use Python. I just used MATLAB to get the coordinates and import them as constant in this part. It was easier to just use matlab to get coordinates rather than implement an entire function in Python. The code is the same as the lab manual.

```
# Coordinates using ginput in MATLAB - (Top Left, Top Right, Bottom Right, Bottom Left)
x_coords = np.array([141.9, 308.7, 256.8, 4.23])
y_coords = np.array([29.1, 47.2, 215.1, 159.7])

# The A4 paper size is 210 x 297mm
x_a4 = np.array([0, 210, 210, 0])
y_a4 = np.array([0, 0, 297, 297])

# Make both into numpy arrays, but use vertical stack to merge coords, then transpose
source_coords = np.vstack([x_coords, y_coords]).T
target_coords = np.vstack([x_a4, y_a4]).T

print(f"Source Coordinates: {source_coords}")
print(f"Target Coordinates: {target_coords}")

✓ 0.0s
Source Coordinates: [[141.9   29.1 ]
 [308.7   47.2 ]
 [256.8  215.1 ]
 [  4.23 159.7 ]]
Target Coordinates: [[  0   0]
 [210   0]
 [210 297]
 [  0 297]]
```

Figure 34: Results of Coordinates and Target A4 Coordinates (210 x 297)

2.6.3 Part C – Setting up Matrices to Estimate Projective Transformation

In the notebook, I have included a detailed analysis of the function, which is represented by the following figure.



Part C - Setting up Matrices to Estimate Projective Transformation

In the manual, the 2D planar projective transform is expressed via this 3 by 3 matrix

$$\begin{bmatrix} kx_{im} \\ ky_{im} \\ k \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix}$$

where (x_im, y_im) are coordinates in the **output** image. or the corrected frontal view. k is some scaling factor.

(X_w, Y_w) are coordinates in the **input** image, or the distorted photo. The 3x3 matrix with m values is the transformation we are actually trying to find.

The manual also shows this algebraic form, which is actually just simple matrix multiplication.

$$x_{im} = \frac{m_{11}X_w + m_{12}Y_w + m_{13}}{m_{31}X_w + m_{32}Y_w + 1}$$

$$y_{im} = \frac{m_{21}X_w + m_{22}Y_w + m_{23}}{m_{31}X_w + m_{32}Y_w + 1}$$

If we just do the matrix multiplication,

$$k = m_{31}X_w + m_{32}Y_w + 1$$

, to find x_im is just to do the same but divide by k. **k is actually what creates the perspective effect.**

Since the bottom right is one, there are 8 unknowns to find.

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & 1 \end{bmatrix}$$

For each point we know, we can write 2 equations, one for x and one for y. (just simple matrix multiplicationp)

$$X_w^1 \cdot m_{11} + Y_w^1 \cdot m_{12} + m_{13} + 0 \cdot m_{21} + 0 \cdot m_{22} + 0 \cdot m_{23} - x_{im}^1 X_w^1 \cdot m_{31} - x_{im}^1 Y_w^1 \cdot m_{32} = x_{im}^1$$

and for y

$$0 \cdot m_{11} + 0 \cdot m_{12} + 0 \cdot m_{13} + X_w^1 \cdot m_{21} + Y_w^1 \cdot m_{22} + m_{23} - y_{im}^1 X_w^1 \cdot m_{31} - y_{im}^1 Y_w^1 \cdot m_{32} = y_{im}^1$$

Then we have the big matrix Au=v seen in the manual. We want to calculate u which is the 8x1 vector of unknowns. We know the initial, we know the output. We just need to take the inverse of A times v to get u. Or theres a least squares solution i think if we have more than 4 points

$$\mathbf{u} = A^{-1}\mathbf{v}$$

$$\mathbf{u} = (A^T A)^{-1} A^T \mathbf{v}$$

To verify: once we have the transformation matrix U

$$\mathbf{w} = U \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix}$$

And after normalization

$$\begin{bmatrix} x_{im} \\ y_{im} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1/\mathbf{w}_3 \\ \mathbf{w}_2/\mathbf{w}_3 \end{bmatrix}$$

we should match the target coordinates.

Figure 35: Information on Projective Transformation

From the information I have gathered, we can create the projection matrices using the following function. I did not use cv2.getPerspectiveTransform in the creation of this matrix, only for verification. Figure 36 shows the code that generates the projection matrices, and solves for the matrix U using np.linalg.solve(A,v).

```python
def setup_projection_matrices(source_coords, target_coords):
    """
    This function manually constructs the A matrix and v vector for projective transformation I guess

    For each point (Xw, Yw) to be projected onto (x_im, y_im), we get two equations by consulting the A matrix in the manual
    Row 1: = [Xw Yw 1 0 0 0 0 -x_im*Xw -x_im*Yw]
    Row 2: = [0 0 0 Xw Yw 1 -y_im*Xw -y_im*Yw]
    """
    n_points = len(source_coords)
    A = np.zeros((2 * n_points, 8))
    v = np.zeros(2 * n_points)

    for i in range(n_points):
        Xw, Yw = source_coords[i]
        x_im, y_im = target_coords[i]

        # First equation to get the xcoords
        A[2*i] = [Xw, Yw, 1, 0, 0, 0, -x_im*Xw, -x_im*Yw]
        v[2*i] = x_im

        # Second equation (for y coordinate)
        A[2*i + 1] = [0, 0, 0, Xw, Yw, 1, -y_im*Xw, -y_im*Yw]
        v[2*i + 1] = y_im

    return A, v

A, v = setup_projection_matrices(source_coords, target_coords)
print(f"A = {A}")
print(f"v = {v}")

# Solve for u, this is u = A inverse * v
u = np.linalg.solve(A, v)
print(f"Transformation parameters u = {u}")

# We also need to reshape u to be a 3x3 matrix as said in the lab manual
U = np.concatenate([u, [1]]).reshape(3, 3)
print("\nProjective Transformation Matrix U (3x3):")
print(U)
```

Figure 36: Acquiring the Projection Matrix

```
A = [[   141.9      29.1       1.        0.        0.        0.        0.
       0.  ]
 [     0.        0.        0.      141.9      29.1       1.        0.
       0.  ]
 [   308.7      47.2       1.        0.        0.        0.   -64827.
   -9912.  ]
 [     0.        0.        0.      308.7      47.2       1.        0.
       0.  ]
 [   256.8     215.1       1.        0.        0.        0.   -53928.
  -45171.  ]
 [     0.        0.        0.      256.8     215.1       1.   -76269.6
  -63884.7 ]
 [     4.23    159.7       1.        0.        0.        0.        0.
       0.  ]
 [     0.        0.        0.        4.23    159.7       1.    -1256.31
  -47430.9 ]]
v = [  0.   0. 210.   0. 210. 297.   0. 297.]
Transformation parameters u = [   1.49936359    1.58053129 -258.75315434   -0.41234874    3.79998732
  -52.06734419    0.00023532    0.00539207]

Projective Transformation Matrix U (3x3):
[[   1.49936359    1.58053129 -258.75315434]
 [  -0.41234874    3.79998732  -52.06734419]
 [   0.00023532    0.00539207    1.        ]]
```

Figure 37: Results of Transform Matrix U

2.6.3.1 Sub Part: Verifying our U Matrix against cv2's U Matrix

```
U_cv2 = cv2.getPerspectiveTransform(source_coords.astype(np.float32), target_coords.astype(np.float32))
print("="*60)
print("COMPARISON: Manual vs OpenCV")
print("="*60)
print("\nOur U matrix:")
print(U)
print("\nOpenCV's U matrix:")
print(U_cv2)
print("\nDifference (ours - cv2):")
print(U - U_cv2)
print(f"\nMax absolute error: {np.max(np.abs(U - U_cv2)):.6f}")
```

✓ 0.0s

```
============================================================
COMPARISON: Manual vs OpenCV
============================================================

Our U matrix:
[[   1.49936359    1.58053129 -258.75315434]
 [  -0.41234874    3.79998732  -52.06734419]
 [   0.00023532    0.00539207    1.        ]]

OpenCV's U matrix:
[[   1.49936356    1.58053122 -258.75313966]
 [  -0.41234869    3.79998713  -52.06735074]
 [   0.00023532    0.00539207    1.        ]]

Difference (ours - cv2):
[[ 0.00000003  0.00000006 -0.00001468]
 [-0.00000006  0.00000019  0.00000655]
 [-0.          0.          0.        ]]

Max absolute error: 0.000015
```

Figure 38: Our U matrix vs cv2's U matrix (Absolute error 0.000015)

From analysis, both matrices are about the **same**, the difference is due to small computational precision errors causing the inaccuracy. Our matrix computation is thus accurate, and we can move on to the next steps.

2.6.4 Part D and E – Warp Image and Display

In this section, we warp the image and display it to the screen, showing the full frontal view of the book. We use cv2's warp perspective on the image. Reference to the documentation:

https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html

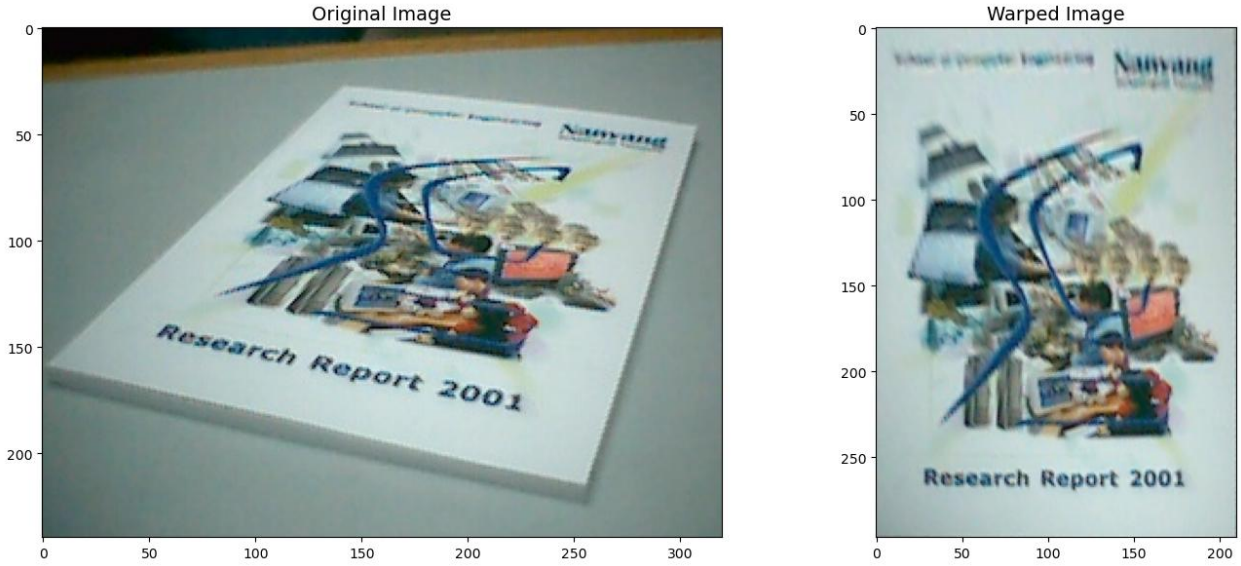


Figure 39: Original vs Warped Image

After warping, the perspective correction was successful. The book is now showing a frontal view with no more trapezoidal distortion or shape like the original image. The image dimensions are also correct, with the image having 210x297 pixels matching A4 proportions at 1 pixel = 1mm scale. The text is roughly readable, the Nanyang and Research Report 2001 words are readable.

However, we do face some blurriness and low-resolution, with loss of detail, some fine details are lost when compared to the original. Some possible causes can be that due to the mapping, this typically requires interpolation (estimation of pixel values at non integer positions.)

In warp perspective, it uses nearest neighbor or bilinear interpolation, which can cause blur. If we want higher quality, we need higher order interpolations. Furthermore, since the original photo seems to be taken at low resolution, after the "flattening" process, we are trying to reconstruct detail that wasn't captured well originally. Even on the book, the top text beside Nanyang takes less pixels and when warped to frontal view, these areas are "stretched" from limited data and causes even more blur.

2.6.4 Part F – Identify Big Rectangular Pink Area

In this section, we are tasked to identify the big rectangular pink area using any methods we wish. From visual inspection, the computer screen is full of pink pixels, and the rest of the image does not. Due to this, we can use color masking by isolating pixels within a specific color range to find a target object. For color detection, working in the HSV (Hue-Saturation-Value) space is better for color detection as the hue (color) is separated from the brightness of the image.

```python
def detect_screen(img):
    """
    Detect using refined color range
    """
    # Convert to HSV
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    # Based on the pink screen color range
    lower = np.array([104, 30, 50])
    upper = np.array([124, 255, 255])

    mask = cv2.inRange(hsv, lower, upper)

    # Find contours
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    result = img.copy()
    if contours:
        largest = max(contours, key=cv2.contourArea)
        x, y, w, h = cv2.boundingRect(largest)
        cv2.rectangle(result, (x, y), (x+w, y+h), (0, 255, 0), 2)
        print(f"Screen detected at: ({x}, {y}), size: {w}x{h}")

    return result

result = detect_screen(warped_image)
display_image(result, title="Manual Detection", grey=False)
```

Figure 40: Finding Screen Function

The function first converts the image color space to HSV, which is better for color detection. It also defines a lower and upper bound of HSV values to capture the pinkish area of the screen. This was done with an utility function to get the color ranges in the specified pixels and are input into the function as constants. We then create a binary mask and check every pixel's HSV value if its in the range. Finally, we find the contours of the white regions in the mask, and get the outer boundaries. This gives us the rough area of the area where we have a match. Later, we find the largest contour by area, as there may be many candidates in the image. By this, we assume that the screen is the largest object in the image. The last part of the function just draws a bounding rect.

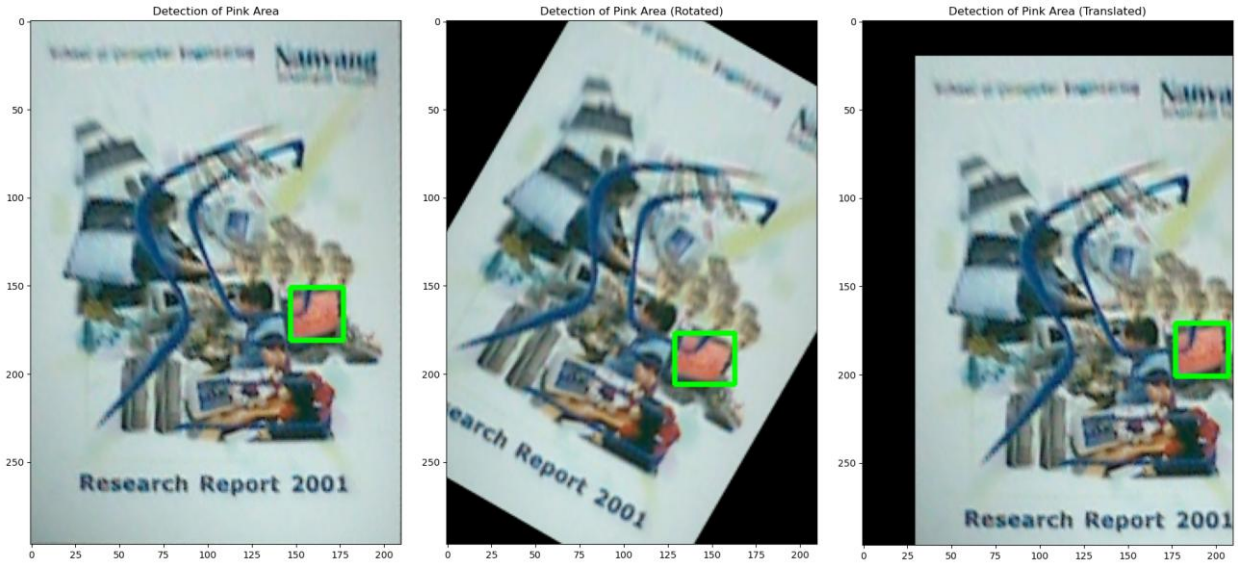


Figure 41: Detection of Pink Screen (Original vs Rotated 30 degrees vs Translated)

The detection is working as expected, even with transformations on the image, indicating that utilizing masks on the HSV color space can give great results in identifying certain parts of images.

2.7 Coding Two Perceptrons

The decision boundary can be represented in different ways, as seen in the slides. The summation form is just the linear combination expressed in short form, while the vector form is whats commonly known, W^Tx + W_n+1 (usually this is the bias). The separation rule uses 0 as the binary classifier. After the sum is passed into the activation function and we can get the actual output.

2.7.1 Part A – Simple Perceptron Training, Algorithm 1

From Algorithm 1, the weights are simply adjusted based on classification accuracy to find a linear decision boundary that accurately separates the two classes. On classification errors, weights are updated accordingly, and the boundary is shifted accordingly, while correct classifications make the weight remain unchanged. In the slides, the algorithm alternates between x1 and x2 on even and odd iterations to iteratively test between each point. The learning rate and bias are also kept at **1** throughout the experiment, with no updates. This is a simple learning algorithm to demonstrate how perceptron works.

```python
# Setup the inputs and parameters
x1 = np.array([3, 3, 1]) #belongs to c1
x2 = np.array([1, 1, 1]) #belongs to c2
weights = np.array([0, 0, 0]) # weights initially all

lr = 1 # learning rate set as 1 as per dlecture slides
# 100 should be enough to find a solution, especially since its just binary classification
# But if we want the best representing line, more iterations till convergence is better
def train(x1, x2, weights, lr, early_stop=False, max_epochs=100, early_stop_threshold=3):
    # Store original weights
    old_weights = np.copy(weights)
    weights = np.copy(weights)
    early_stop_count = 0
    epsilon = 1e-6
    # Copy vectors
    x1 = np.copy(x1)
    x2 = np.copy(x2)
    # Assume we start at epoch 1, the results will be different if we classify and optimize from x2 instead
    for epoch in range(1, max_epochs + 1):
        # From the slides example, it seems for odd iterations we try to classify x1, and then even for x2
        # For early stopping, we can just say that if we have classified both points correctly, we stop without finding the optimal line
        if epoch % 2 == 1:
            # Classify x1 sing np dot product
            classification = np.dot(weights, x1)
            if classification > 0:
                # If correct, no update
                pass
            else:
                # Incorrect, update weights based on lr * x
                weights += lr * x1
        else:
            # Classify x2 using np dot product
            classification = np.dot(weights, x2)
            if classification < 0:
                # If correct, no update
                pass
            else:
                # Incorrect, update weights based on - lr * x
                weights -= lr * x2

        # Check for early stopping based on weight diff to calculcate L2 norm
        weight_diff = np.linalg.norm(old_weights - weights)
        if weight_diff < epsilon:
            early_stop_count += 1
        else:
            # Update old weights
            old_weights = np.copy(weights)
            early_stop_count = 0

        # Check for early stopping
        if early_stop and early_stop_count == early_stop_threshold:
            return epoch, weights
    return epoch, weights
```

Figure 42: Simple Perceptron Implementation

In the code above, we do the classification for x1 and x2 in the lecture slides turn by turn, classifying x1 on odd epochs and x2 on even epochs. To test the classification, we use the dot product against each other. If there is a misclassification, we update the weights based on the update rules, weights += lr * x1 or weights -= lr * x2. As an addition to not run all the epochs, early stopping is also added, where if the weights have not changed over early_stop_threshold(3), the training is stopped. Another alternative would be to stop once the classifier can classify both points in succession. However, for scalability's sake, we check for convergence by weight differences for early stopping.

After running the algorithm, we see that the final weights remain the same with and without early stopping. This means that the algorithm/model is quite confident of its classifiers and no more weight updates are done. From the initial points, we can already infer that this not the best line to separate the two points.

```
Final weights: [ 1  1 -3]
Epochs taken: 13
Final weights: [ 1  1 -3]
Epochs taken: 100
```

2.7.1.1 Plotting of Decision Boundary

The following snippet shows the calculations I used to calculate the decision boundary in the notebook. It is for reference and is not an actual Figure.

To solve for x, we need in form of y as a function of x

1. Starting equation

$$w_1 * x s_1 + w_2 * x_2 + w_3 = 0$$

2. Isolate x_2 (y)

$$w_2 * x_2 = -w_1 * x_1 - w_3 = 0$$

3. Divide both sides by w_2

$$x_2 = (-w_1 * x_1 - w_3)/w_2$$

4. Rearrange to factor our negative

$$x_2 = (-w_3 - w_1 * x_1)/w_2$$

5. With calculated weights [1 1 -3]

$$x_2 = ((-3) - 1 * x_1)/1$$
$$x_2 = (3 - x_1)/1$$
$$x_2 = 3 - x_1$$

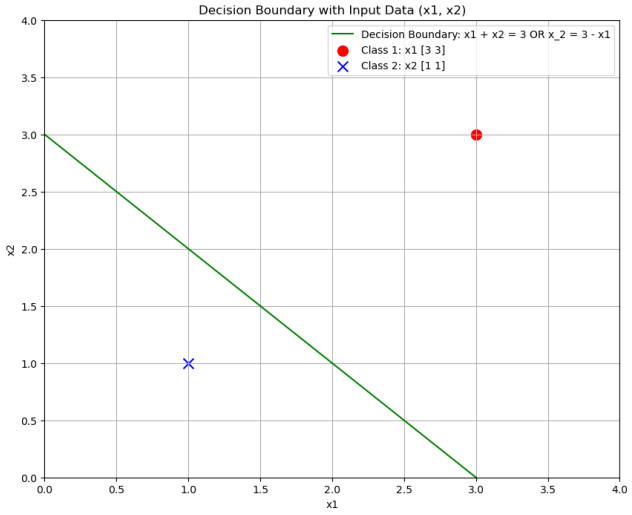


Figure 43: Decision Boundary Results (Algorithm 1)

As expected, from human inspection, the points are separated cleanly, the classification works as is. However, the problem with perceptrons is that it gives non unique solutions. When we calculate

$$x_1 + x_2 = 3$$

any line where $2 < (x1+x2) < 6$ should theoretically work. **The best line is the line that maximises the margin**, the distance to nearest points from both classes. **This "best" line** should have the following properties:

1. Pass through or near the midpoint of the two closest points of each class

2. Be perpendicular to the line connecting the points

We can plot out the best line using calculations like Support Vector Machines. The function below in Figure 44 uses the midpoint of the two points as the optimal position and calculates the direction between them. The margin of the line boundary of the perceptron is likewise the minimum of the distance from each point to the line boundary. We apply similar calculations to calculate the ideal line boundary at the midpoint.

```python
def plot_perceptron_vs_svm(x1, x2, perceptron_weights, figsize=(10, 8)):
    # Remove bias term if present
    x1_2d = x1[:2] if len(x1) == 3 else x1
    x2_2d = x2[:2] if len(x2) == 3 else x2

    # Calculate optimal line using midpoint
    midpoint = (x1_2d + x2_2d) / 2
    direction = x1_2d - x2_2d

    # Find optimal w for SVM
    optimal_w = np.array([
        direction[0],
        direction[1],
        -(direction[0]*midpoint[0] + direction[1]*midpoint[1])
    ])

    # Margin boundaries
    upper_margin_c = -(optimal_w[0]*x1_2d[0] + optimal_w[1]*x1_2d[1])
    lower_margin_c = -(optimal_w[0]*x2_2d[0] + optimal_w[1]*x2_2d[1])
    margin_distance = abs(upper_margin_c - lower_margin_c) / np.sqrt(optimal_w[0]**2 + optimal_w[1]**2)

    # Calculate point to line distance to find out margin
    def point_to_line_distance(point, w1, w2, w3):
        return abs(w1*point[0] + w2*point[1] + w3) / np.sqrt(w1**2 + w2**2)

    # Find margin against each x1 and x2 point.
    perceptron_margin_x1 = point_to_line_distance(x1_2d, perceptron_weights[0],
                                                  perceptron_weights[1], perceptron_weights[2])
    perceptron_margin_x2 = point_to_line_distance(x2_2d, perceptron_weights[0],
                                                  perceptron_weights[1], perceptron_weights[2])
    # The bottleneck is the point closest to the line, so we take the minimum of both
    perceptron_min_margin = min(perceptron_margin_x1, perceptron_margin_x2)


    boundary_x = np.linspace(min(x1_2d[0], x2_2d[0]) - 1, max(x1_2d[0], x2_2d[0]) + 1, 100)

    # Perceptron
    boundary_y_perceptron = (-perceptron_weights[2] - perceptron_weights[0] * boundary_x) / perceptron_weights[1]
    perceptron_norm = np.sqrt(perceptron_weights[0]**2 + perceptron_weights[1]**2)
    upper_perceptron_c = perceptron_weights[2] - perceptron_min_margin * perceptron_norm
    lower_perceptron_c = perceptron_weights[2] + perceptron_min_margin * perceptron_norm
    boundary_y_perceptron_upper = (-upper_perceptron_c - perceptron_weights[0] * boundary_x) / perceptron_weights[1]
    boundary_y_perceptron_lower = (-lower_perceptron_c - perceptron_weights[0] * boundary_x) / perceptron_weights[1]

    # SVM
    boundary_y_decision = (-optimal_w[2] - optimal_w[0] * boundary_x) / optimal_w[1]
    boundary_y_upper = (-upper_margin_c - optimal_w[0] * boundary_x) / optimal_w[1]
    boundary_y_lower = (-lower_margin_c - optimal_w[0] * boundary_x) / optimal_w[1]
```

Figure 44: Plotting of Margins of Perceptron vs Optimal Line Function

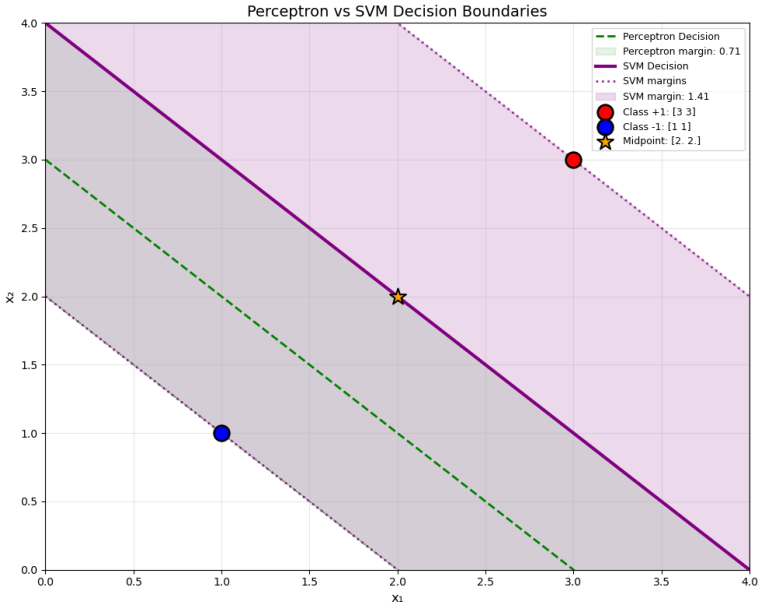


Figure 45: Plotting of Margins of Perceptron vs Optimal Line

From this, we can actually see the optimal line is the line that maximizes the margin. If we can acquire the line that is in between them which maximizes the margin of each side while staying perpendicular, we have a better separation. To achieve this, we need an algorithm that explicitly optimizes for margin (like SVM) rather than the perceptron which only ensures correct classification. It is a fundamentally different problem in that case.

2.7.2 Part B – Perceptron with Gradient Descent

Algorithm 2 represents a more sophisticated approach to perceptron learning by introducing a loss function to guide weight adjustments. Unlike Algorithm 1, where updates only happen when misclassification occurs, Algorithm 2 uses a least squares error function E(w) = 1/2(r-w^Tx)^2 to measure the difference between the predicted output (W^Tx) and the desired output r. This allows it to quantify numerically how far each prediction is. Since the loss function is differentiable, we can use gradient descent, where the derivative of the error function is calculated with respect to the weights, allowing us to create our update rule. In general, the weight adjustment is now scaled by the magnitude of the error, where larger mistakes trigger bigger corrections, and smaller errors result in finer adjustments, which creates a smoother convergence pattern. The learning rate is typically small, and we can train until the error across all training samples becomes sufficiently small. In Figure 46, the function for training with gradient descent is shown, and in Figure 47, the loss over epochs is shown.

```python
def train_gradient_descent(x1, x2, weights, lr, epochs, r1, r2, max_epochs= 100, normalize = True):
    # Step 1 - We should normalize the vectors to prevent larger vectors dominating weight updates
    if normalize:
        x1_norm = x1 / np.linalg.norm(x1)
        x2_norm = x2 / np.linalg.norm(x2)
    else:
        x1_norm = x1
        x2_norm = x2

    # Initialize the weights
    weights = np.copy(weights)

    # Track loss over iterations for our visualizations
    losses = []

    # Iterate for our epochs
    for epoch in range (1, max_epochs + 1):
        # Alternate between two input, like in algorith,m 1
        x = x1_norm if epoch % 2 == 1 else x2_norm
        r = r1 if epoch % 2 == 1 else r2

        pred = np.dot(weights, x)
        error = r - pred
        # Calculate the loss
        loss = 0.5 * error ** 2
        losses.append(loss)
        # Update weights using grad descent, thne dertivative of loss function is -(r-W^Tx) * x
        # w(k+1) = w(k) + a[r(k) - w^T(k)x(k)]x(k)
        weights = weights + lr * error * x

        # Print progress every 5 epochs
        if epoch % 5 == 0:
            print(f"Epoch {epoch}, Loss: {loss:.4f}, Weights: {weights}")

    # Return the final weights and losses
    return epoch, weights, losses
```

Figure 46: Training of Perceptron with Loss Function and Gradient Descent

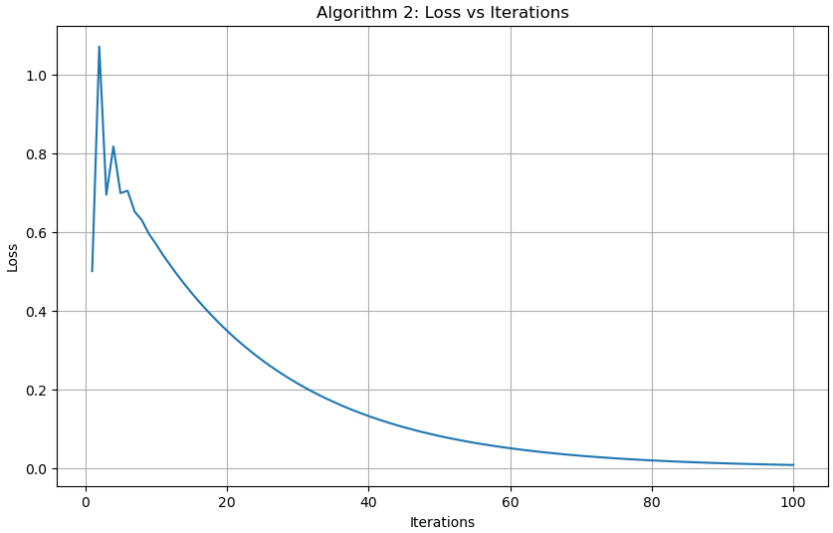


Figure 47: Loss vs Iterations Graph

At the start, the loss greatly increased, partly due to our large learning rate. Over the first few iterations, misclassifications can cause the loss to jump quickly as the error between the predictions is high. Over time, such jumps would slowly converge and form the nice downward smooth graph we are expecting after about iteration 7. Since the loss is decreasing, we can be quite confident that our algorithm is working well.

2.7.2.1 Plotting of Decision Boundary

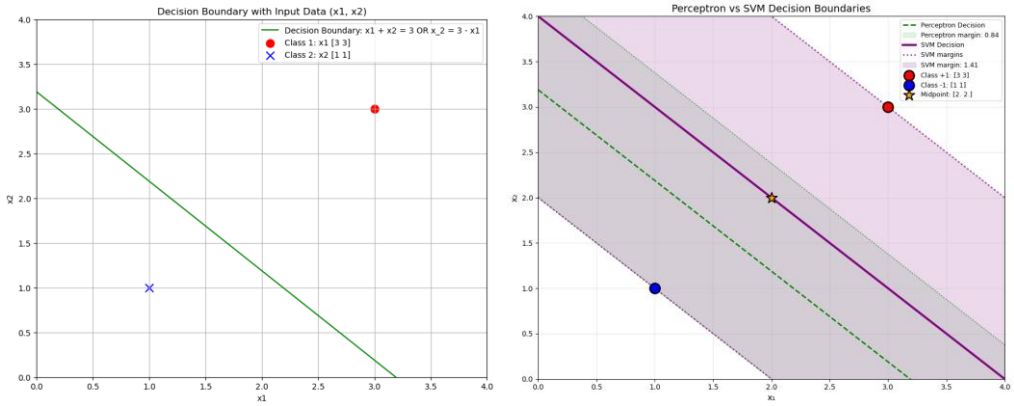


Figure 48: Decision Boundary of Algorithm 2 vs Optimal

In Algorithm 2, the perceptron margin (**0.84**) was larger than Algorithm 1 (**0.77**), showing a better boundary. This shows that incorporating a loss function helped in finding a more optimal decision boundary for this dataset, which is expected as weights update happens even on correct classifications. This is due to the behavior of Algorithm 2, where optimization happens on the differences between the ground truth and predictions.

2.7.3 Part C – Conclusion

For Algorithm 1:

1. Updates weights only on misclassification
2. Simple and efficient due to simple stopping criteria
3. No minimization of loss function, may not find the most optimal boundary as seen in the plots

For Algorithm 2:

1. Uses a loss function to adjust the weights to minimize overall error over time
2. Weights are updated based on an error function, the l2 of the difference between predicted and target output.
3. Smaller learning rate was used to ensure slower and smoother updates, leading to better convergence.
4. May take longer to train than Algorithm 1