SC4061 – Computer Vision
Laboratory 1

Cholakov Kristiyan Kamenov
U2123543B

College of Computing and Data Science

1 Table of Contents

2 Introduction 4

3 Experiments 4

3.1 Contrast Stretching 4

3.1.1 Concept 4

3.1.2 MATLAB Code Analysis and Results 4

3.1.2.1 Parts a) and b) 4

3.1.2.2 Part c) 6

3.1.2.3 Parts d) and e) 6

3.2 Histogram Equalization 7

3.2.1 Concept 7

3.2.2 MATLAB Code Analysis and Results 8

3.2.2.1 Part a) 8

3.2.2.2 Part b) 9

3.2.2.3 Part c) 11

3.3 Linear Spatial Filtering 13

3.3.1 Concept 13

3.3.2 MATLAB Code Analysis and Results 14

3.3.2.1 Part a) 14

3.3.2.2 Part b) 15

3.3.2.3 Part c) 15

3.3.2.4 Part d) 17

3.3.2.5 Part e) 17

3.4 Median Filtering 18

3.4.1 Concept 18

3.4.2 MATLAB Code Analysis and Results 18

3.4.2.1 Parts a), b) and c 18

3.4.2.2 Parts d) and e) 20

3.5 Suppressing Noise Interference Patterns 21

3.5.1 Concept 21

3.5.2 MATLAB Code Analysis and Results 21

3.5.2.1 Part a) 21

3.5.2.2 Part b) 22

3.5.2.3 Part c) 23

3.5.2.4 Part d) 24

3.5.2.5 Part e) 25

3.5.2.6 Part e) 26

3.6 Perspective Distortion of Planar Surface 31
3.6.1 Concept 31
3.6.2 MATLAB Code Analysis and Results 31
3.6.2.1 Part a) 31
3.6.2.2 Part b) 32
3.6.2.3 Part c) 33
3.6.2.4 Parts d) and e) 34
3.6.2.5 Part f) 35
3.7 Perceptron Algorithms 38
3.7.1 Concept 38
3.7.2 MATLAB Code Analysis and Results 38
3.7.2.1 Set-up 38
3.7.2.2 Part a) 39
3.7.2.3 Part b) 41
3.7.2.4 Part c) 43

2 Introduction

The objective of the laboratory is to introduce the following fundamental concepts in the Computer Vision field via a series of practical experiments:

- Contrast Stretching
- Histogram Equalization
- Linear Spatial Filtering
- Median Filtering
- Suppressing Noise Interference Patterns
- Undoing Perspective Distortion of Planar Surface
- Perceptron Algorithms

The experiments were conducted using MATLAB and its Image Processing Toolbox. MATLAB's robust environment allowed for hands-on experience applying these techniques and understanding their applications. The images used for the experiments were stored in a separate directory **img**, inside the working directory of the laboratory MATLAB file.

3 Experiments

In this section, I will examine each experiment in detail, starting with an overview of the key concepts involved, followed by a review of the MATLAB code used to implement the experiment, and concluding with an analysis of the results and insights drawn from the experiment.

3.1 Contrast Stretching

3.1.1 Concept

Contrast stretching is a global point processing technique (depending only on the currently processed pixel and the global minimum and maximum values) that is used to enhance the image by expanding the range of the intensity values. The low values are typically mapped to a predefined extreme minimum (often 0), while the high values are mapped to a predefined extreme maximum (often 255). This process can be described with the following mathematical formula:

$$P_{new} = \frac{P_{old} - P_{min}}{P_{max} - P_{min}} \times 255$$

P_{new} denotes the new value of the pixel's intensity (the result of the formula) and P_{old} is the original pixel intensity. P_{min} and P_{max} are the minimum and maximum intensity values in the image.

The purpose of contrast stretching is to make the details in the image more visible, particularly when the original image has poor contrast due to issues like insufficient lighting or low dynamic range.

3.1.2 MATLAB Code Analysis and Results

3.1.2.1 Parts a) and b)

MATLAB Code:

Figure 1 shows the code that covers parts a) and b) of the experiment. First, the image is loaded using the MATLAB function **imread**, the **whos** command is used to show the parameters of

the image. Then, it is visualized using **imshow** (I am defining a figure before visualizing to support multiple images displayed in separate figures) and a title is set using the **title** command. The image is then converted to grayscale with the help of the **rgb2gray** function, the parameters of the new image are checked again, and the image is plotted in a figure again.

```
% a) & b) Load and display the original image in one figure
Pc = imread('img/mrt-train.jpg');
whos Pc
figure;
imshow(Pc);
title('Original Image (RGB)');

% Convert to grayscale and display in another figure
P = rgb2gray(Pc);
whos P
figure;
imshow(P);
title('Grayscale Image');
```

Figure 1. Contrast stretching – the code for parts a) and b)

Results:

Figure 2 displays the difference in the parameters between the original image and the converted grayscale one. The size (width and length) of the image remains the same but after the conversion, there is only a single channel (representing the intensity), while previously each pixel was a combination of red, green, and blue colors. As a result of the conversion to grayscale, the size (in bytes) of the images is reduced 3 times.

```
Name        Size                Bytes  Class    Attributes

Pc        320x443x3            425280  uint8

Name        Size               Bytes  Class    Attributes

P         320x443            141760  uint8
```

Figure 2. Contrast stretching – the parameters of the images

However, no difference can be seen in Figure 3 between the original and the greyscale image because the original image is already a greyscale image.

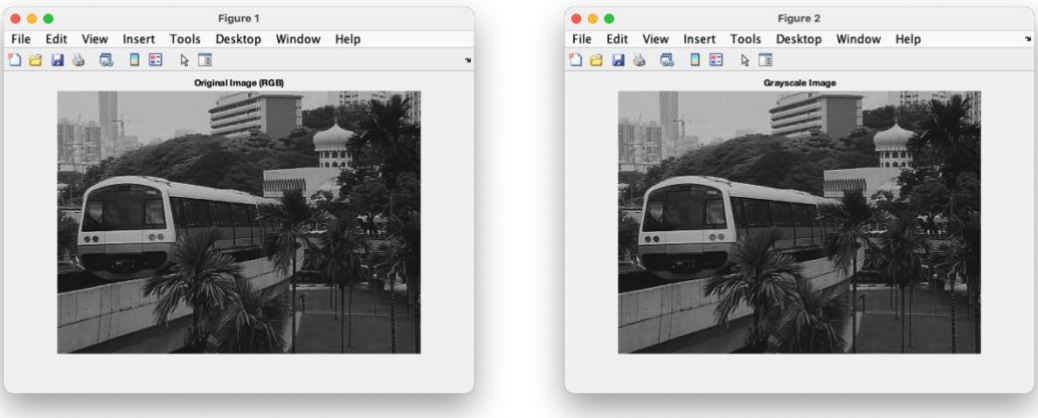


Figure 3. Contrast stretching – original and greyscale images

3.1.2.2 Part c)

MATLAB Code:

As seen in Figure 4, I am simply defining 2 new variables to store the minimum and maximum intensities of the converted image. I am getting the extrema values using the **min** and **max** functions and passing a vector instead of a matrix with the help of the **(:)** syntax. The minimum and maximum intensities are then printed in the console using the formatted print statement **fprintf** command.

```
% c) Check the minimum and maximum intensity values
min_intensity = min(P(:));
max_intensity = max(P(:));
fprintf('Minimum Intensity: %d\n', min_intensity);
fprintf('Maximum Intensity: %d\n', max_intensity);
```

Figure 4. Contrast stretching – the code for part c)

Results:

The result of the code can be seen in Figure 5, the original minimum and maximum intensities are 13 and 204.

```
Minimum Intensity: 13
Maximum Intensity: 204
```

Figure 5. Contrast stretching – original minimum and maximum intensities

3.1.2.3 Parts d) and e)

MATLAB Code:

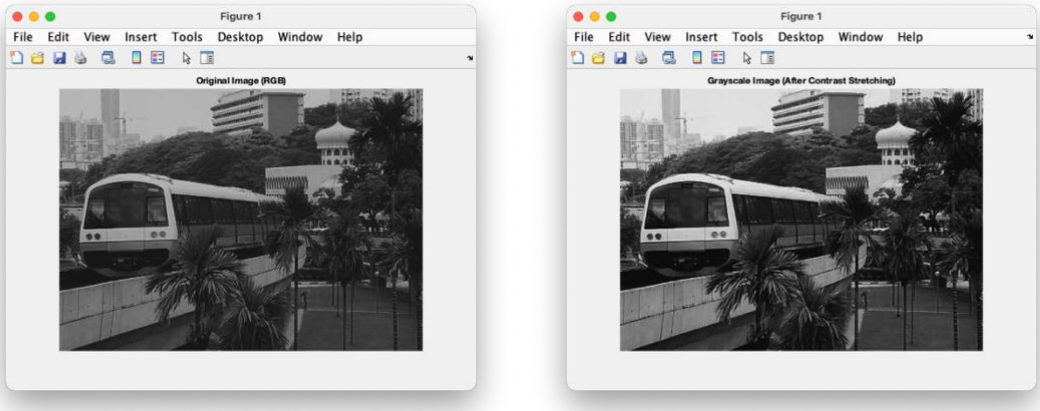
Figure 6 shows the code section for parts d) and e). First, I am creating new double instances of the image's intensities and the extrema values, they will be used for computing the new intensity values and are converted from **uint8** to **double** using the **double** function. The new matrix **P_stretched** is the result of my implementation of the described contrast stretching formula in the concept section of this experiment. However, this matrix is a result of multiplication and division with **double** values, thus it must be converted back to **uint8** using the **uint8** function. I am also getting the new extrema values in the same way but by passing a vector of the stretched matrix. Lastly, the new contrast-stretched image is plotted in a new figure.

Results:

Figure 7 presents the new extrema values and the comparison between the original image and the contrast-stretched one. As expected, the minimum and maximum values are now 0 and 255, covering the whole greyscale **uint8** intensity spectrum. Also, when comparing the original image with the enhanced one, it can be noticed that the details are easier to spot in the contrast-stretched image. The new image does not look dimmed like the original one because the intensities close to the original minimum and maximum are further expanded to the absolute spectrum extrema. The darker parts of the image become even darker, while the white/bright sectors are now even brighter.

```
% d) & e) Perform contrast stretching

% Convert the grayscale image to double for computation
P_double = double(P);

% Convert min and max intensities to double
min_intensity_double = double(min_intensity);
max_intensity_double = double(max_intensity);

% Apply contrast stretching
P_stretched = (P_double - min_intensity_double) * (255 / (max_intensity_double - min_intensity_double));

% Convert the result back to uint8 format
P2 = uint8(P_stretched);

% Verify that the minimum and maximum intensities are now 0 and 255
min_new = min(P2(:));
max_new = max(P2(:));
fprintf('New Minimum Intensity: %d\n', min_new);
fprintf('New Maximum Intensity: %d\n', max_new);

% Display the contrast-stretched image
figure;
imshow(P2);
title('Grayscale Image (After Contrast Stretching)');
```

Figure 6. Contrast stretching – the code for parts d) and e)




```
New Minimum Intensity: 0
New Maximum Intensity: 255
```

Figure 7. Contrast stretching – contrast-stretched image and new extrema

3.2 Histogram Equalization

3.2.1 Concept

Histogram equalization is another global point-processing technique that aims to enhance the contrast of an image by redistributing the image's intensities. However, it differs from contrast stretching because histogram equalization uses a nonlinear transformation to uniformly spread the intensity values of the image to the whole spectrum. As for the global parameters, the technique uses the global probabilities of the pixels rather than the extrema.

The histogram equalization technique depends on number of possible intensity levels L, the size of the image (number of pixels) $N \cdot M$. The transformation of the pixels to their new values can be described in the following steps.

Histogram Equalization Steps:

1. Compute the probability distribution function (PDF): $p_r(r_k) = \frac{n_k}{N \cdot M}$:
   - r_k is the k-th intensity level
   - n_k is the number of pixels with intensity r_k
2. Compute the cumulative distribution function (CDF): $c_r(r_k) = \sum_{j=0}^{k} p_r(r_j)$
3. Apply the transformation function: $s_k = \text{round}\big((L-1) \cdot c_r(r_k)\big)$
   - $L - 1$ is used because this is the maximum possible value (for example, if there are 256 values, the possible values are 0 – 255)

Simply, the whole formula can be rewritten as: $s_k = \text{round}\left((L-1) \cdot \sum_{j=0}^{k} p_r(r_j)\right)$

The purpose of histogram equalization is to uniformly redistribute the intensity values using a more sophisticated technique based on the distribution probabilities. This method is more robust in terms of outliers (minimum and maximum) because it does not purely rely on the extrema. As a result, the output of the transformation is an enhanced image with clearer details and better human perception.

3.2.2 MATLAB Code Analysis and Results

3.2.2.1 Part a)

MATLAB Code:

As seen in Figure 8, part a) consists of displaying 2 histogram plots for the given image: one with 10 bins and another with 256 bins. This is done using the **imhist** function part of the **Image Processing Toolbox**, the function takes 2 parameters (image and number of bins). Thus, to plot the 2 histograms, only the second parameter is altered.

```
% a) Display the intensity histogram of P using 10 and 256 bins

% Display the histogram with 10 bins using imhist
figure;
imhist(P, 10);
title('Histogram of Grayscale Image (10 bins)');

% Display the histogram with 256 bins using imhist
figure;
imhist(P, 256);
title('Histogram of Grayscale Image (256 bins)');
```

Figure 8. Histogram equalization – the code for part a)

Results:

In the figure below (Figure 9), the comparison between the 2 histograms with 10 and 256 bins can be observed. Both plots show the distribution of the intensities, however, the right one (the one with 256 bins) provides more detailed information about the distribution of the intensities. Having 256 bins for a grayscale image with **uint8** intensity values means that each bin corresponds to a single intensity value and the bin's y-axis value represents the number of pixels with exactly this intensity. On the other hand, the bins in the left plot correspond to a range of intensity values. Thus, their y-axis value is the sum of the number of pixels with intensity values in the range of the bin. This can be verified by comparing the y-axis value of

the bins, the ones in the left plot are in times higher than the ones in the figure with 256 bins. Typically, the bins' width is determined by a formula similar to this one:

$$\text{Bin Width} = \frac{\text{Max Value} - \text{Min Value}}{N_{\text{bins}}}$$

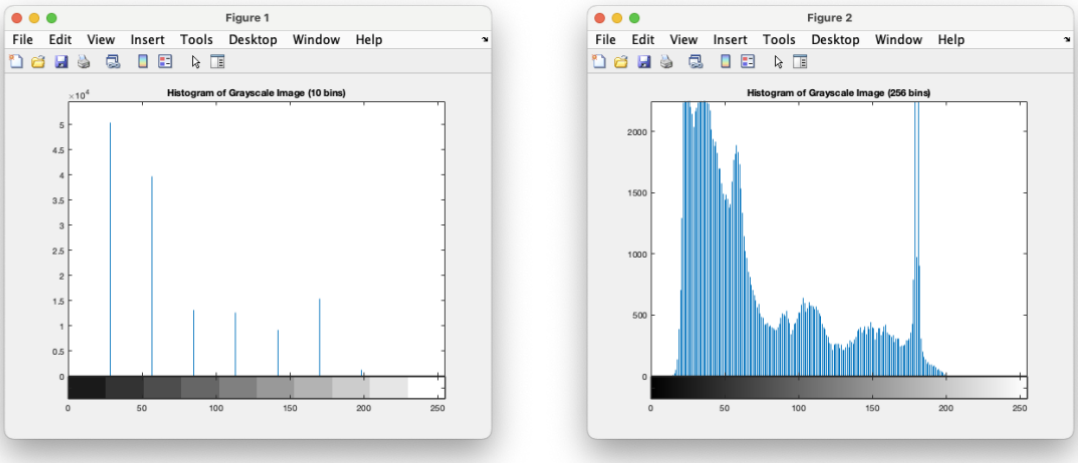


Figure 9. Histogram equalization – histograms of the image's intensities
(left – 10 bins; right – 256 bins)

3.2.2.2 Part b)

MATLAB Code:

The code for part b) (Figure 10) is again quite straightforward, I am simply using the **histeq** function and passing the image and the number of distinct levels. However, 255 bins are passed as the second parameter, although there are 256 distinct levels. This is due to the fact that in histogram equalization the maximum values will always map to themselves. After performing the histogram equalization, the histograms with 10 and 256 bins are visualized again with the help of **imhist** on the intensities of the new image. Then, the new image is also plotted in a figure.

```
% b) Perform histogram equalization and redisplay histograms

% Perform histogram equalization using histeq
P3 = histeq(P, 255);

% Display the histogram of the equalized image with 10 bins using imhist
figure;
imhist(P3, 10);
title('Histogram of Equalized Image (10 bins)');

% Display the histogram of the equalized image with 256 bins using imhist
figure;
imhist(P3, 256);
title('Histogram of Equalized Image (256 bins)');

% Display the equalized image
figure;
imshow(P3);
title('Equalized Grayscale Image');
```

Figure 10. Histogram equalization – the code for part b)

Results:

After running the code, the successful results are clearly seen in the histogram plots in Figure 11. Comparing them to the original ones from Figure 9, the effect of the equalization is clearly seen, especially in the histogram of the equalized image with 10 bins. On the other hand, there is a significant drop in the number of pixels with intensities in the bins in the range of 150-240 on the plot with 256 bins. However, the density in this region is much higher because most of the unique intensity values in the region are present, while in the beginning region, 0-150, many bins are not visible simply because they either contain too few instances or do not contain any pixels with such intensity values. This phenomenon is a result of the fact that histogram equalization does not guarantee that the number of unique intensity values will remain the same. Some values are simply gone because of the redistribution of the intensity values and the rounding of the results after the histogram equalization.

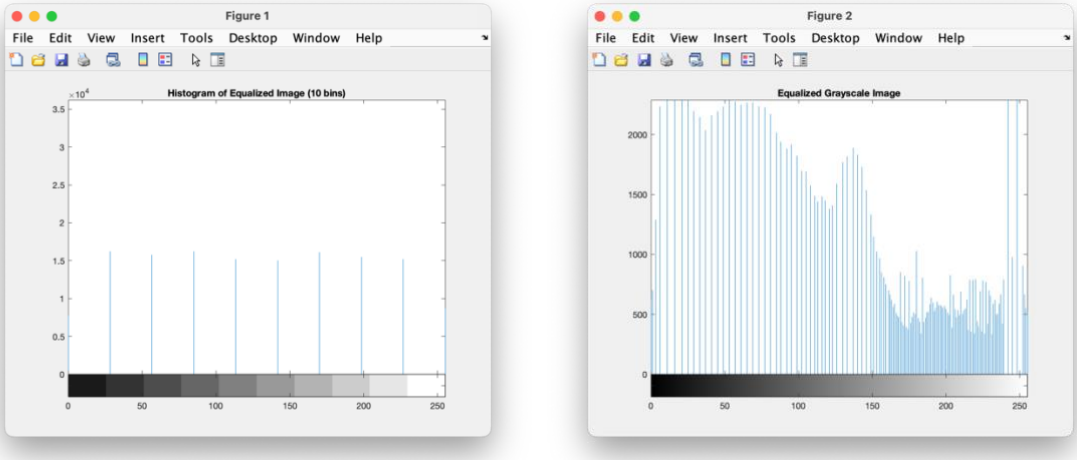


Figure 11. Histogram equalization – histograms of the image's intensities after equalization (left – 10 bins; right – 256 bins)

When the original image is compared with the equalized image, the histogram equalization has clearly enhanced the contrast range of the image. The new image is brighter, appears sharper than the original one, and the details are easier to see. In conclusion, the resulting image is more appealing to the human eye.

Figure 12. Histogram equalization – original image vs image after histogram equalization

3.2.2.3 Part c)

MATLAB Code:

The code for part c), presented in Figure 13, makes use of the same **histeq** and **imhist** functions to perform the histogram equalization again on the already equalized **P3** image, then plot its histogram with 256 bins and show the re-equalized image. Additionally, I am adding a comparison functionality between the 2 equalized images. First, I am performing a simple check with the **isequal** function, which will check if the pixel values of the 2 images are the same. Then, I subtract the 2 images and show a figure with the absolute value of the difference. This will make the pixels with no difference (0) black, and the different ones will appear as white dots.

```
% c) Rerun histogram equalization on P3

% Perform histogram equalization again on P3
P4 = histeq(P3, 255);

% Display the histogram of P4 with 256 bins using imhist
figure;
imhist(P4, 256);
title('Histogram of Re-Equalized Image (256 bins)');

% Display the re-equalized image
figure;
imshow(P4);
title('Re-Equalized Grayscale Image');

% Check if the equalized and re-equalized images are exactly the same
are_equal = isequal(P3, P4);

% Print the according response
if are_equal
    disp('P3 and P4 are exactly the same.');
else
    disp('P3 and P4 are different.');
end

% Subtract P3 from P4
difference = abs(double(P3) - double(P4));

% Display the difference
figure;
imshow(difference, []);
title('Differences Between P3 and P4');
```

Figure 13. Histogram equalization – the code for part c)

Results:

After performing the histogram equalization on the already equalized image and comparing the new histogram plot (the one on the right) in Figure 14 with the one from the first equalization, no difference can be clearly observed between the results of the same equalization on the original image and the re-equalization on the already equalized image. The same lack of difference is present in comparing the output images from the histogram equalization (Figure 15).

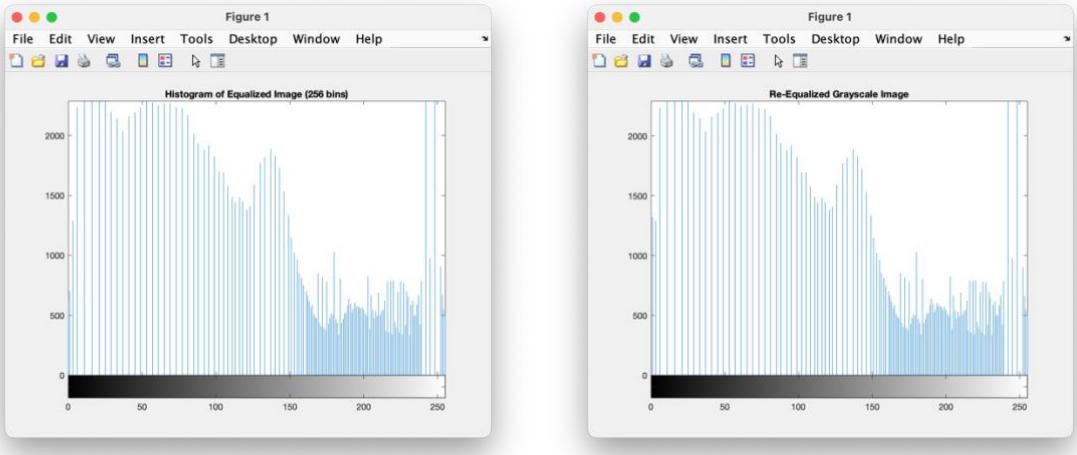
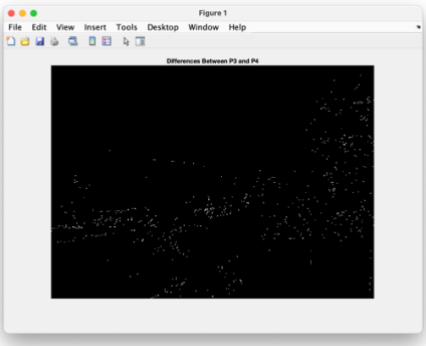


Figure 14. Histogram equalization – 256-bin histograms of the equalized and re-equalized images

Figure 15. Histogram equalization – equalized vs re-equalized images

However, despite not being able to visually identify any differences in the 2 images, there exist such differences, and this can be clearly seen from the simple check and the if-else statements in the code. The console output says that the images are different, and after visualizing them using the absolute difference functionality, the pixels with different intensities are highlighted in white.




```
P3 and P4 are different.
```

Figure 16. Histogram equalization – differences between the equalized and re-equalized images

In conclusion, re-applying histogram equalization on an already equalized image results in very minor differences that are not visible to the human eye in most cases. The re-equalization does not affect the image much because the pixel intensity values of the image are already close to the optimal uniform distribution. Also, continuing with re-applying the equalization on already equalized images will result in smaller and smaller differences until the equalization reaches the most uniform distribution. Typically, histogram equalization converges extremely fast. If the code from Figure 17 is used, the convergence of the equalization happens after 2 iterations (as seen in Figure 18) from starting with the original image.

```
% Check when the histogram equalization will converge

% Initial Image (P is the initial equalized image)
P_current = P;  % Start with the first equalized image
max_iterations = 100;  % Set a maximum number of iterations to avoid infinite loops
iterations = 0;  % Initialize iteration counter

% Loop to repeatedly equalize the image until convergence
while iterations < max_iterations
    P_next = histeq(P_current, 255);  % Re-equalize the current image

    % Check if the images are identical
    if isequal(P_current, P_next)
        disp(['Converged after ' num2str(iterations) ' iterations.']);
        break;  % Exit the loop if the images are the same
    end

    % Update for the next iteration
    P_current = P_next;
    iterations = iterations + 1;

    % If the loop reaches the maximum iterations, stop and report
    if iterations == max_iterations
        disp('Reached the maximum number of iterations without convergence.');
    end
end

% If it converged, display the number of iterations
if iterations < max_iterations
    disp(['Image equalization converged after ' num2str(iterations) ' iterations.']);
else
    disp('Image equalization did not converge within the max iterations.');
end
```

Figure 17. Histogram equalization – the code histogram equalization convergence

```
Converged after 2 iterations.
Image equalization converged after 2 iterations.
```

Figure 18. Histogram equalization – console output from running the code in Figure 17

3.3 Linear Spatial Filtering

3.3.1 Concept

Linear spatial filtering is a fundamental technique used to enhance an image by applying a filter (also known as kernel) to regions of pixel in the image. The filtering operates in the spatial domain, which means that it modifies a pixel based on its surroundings. One of the most core operations in linear spatial filtering is the convolution of an image with a filter (kernel).

The linear spatial filtering can be defined mathematically using the following equation:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} f(x-s, y-t)h(s,t) = f(x,y) \otimes h(s,t)$$

where:

- $f(x, y)$ is the input image
- $h(s, t)$ is the filter (kernel) or sometimes called point-spread function (PSF)
- $g(x, y)$ is the resulting output image
- a and b defining the size of the filter (kernel)

The filter (kernel) can be manually defined or be derived from another function. An example of a filter (kernel) defined by a function is the Gaussian averaging filter. They are defined by the Gaussian function:

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

where x and y are the dimension of the filter and σ (the standard deviation of a Gaussian distribution) is a free parameter.

3.3.2 MATLAB Code Analysis and Results

3.3.2.1 Part a)

MATLAB Code:

Figure 19 presents the code for part a), for each of the specified filters by the assignment, I am defining its sigma value (standard deviation) and then using the **fspecial** function to create a kernel by passing 3 arguments, the first argument is the type of the filer, the second one is the size, and the last one is the sigma parameter for the Gaussian formula. Then, I am normalizing the filters by dividing each kernel value by the sum of all values in the filter. Finally, the filters are visualized using the **mesh** function.

```
% a) Generate Gaussian filters with sigma = 1.0 and sigma = 2.0

% Create the Gaussian filter with sigma = 1.0
sigma1 = 1.0;
h1 = fspecial('gaussian', [5 5], sigma1);
h1 = h1 / sum(h1(:));

% Create the Gaussian filter with sigma = 2.0
sigma2 = 2.0;
h2 = fspecial('gaussian', [5 5], sigma2);
h2 = h2 / sum(h2(:));

% Display the filters as 3D graphs using mesh
figure;
mesh(h1);
title('Gaussian Filter with Sigma = 1.0');

figure;
mesh(h2);
title('Gaussian Filter with Sigma = 2.0');
```

Figure 19. Linear spatial filtering – the code for part a)

Results:

The visualized 5x5 kernels can be seen in Figure 20. As expected, the second filter is smoother because its standard deviation is 2 times higher. In contrast, the first kernel (the one on the left) has a standard deviation of 1, so its shape is sharper. Thus, the first filter will rely mostly on the center pixel value and will have lower weights for the neighbors compared to the kernel with a standard deviation of 2.

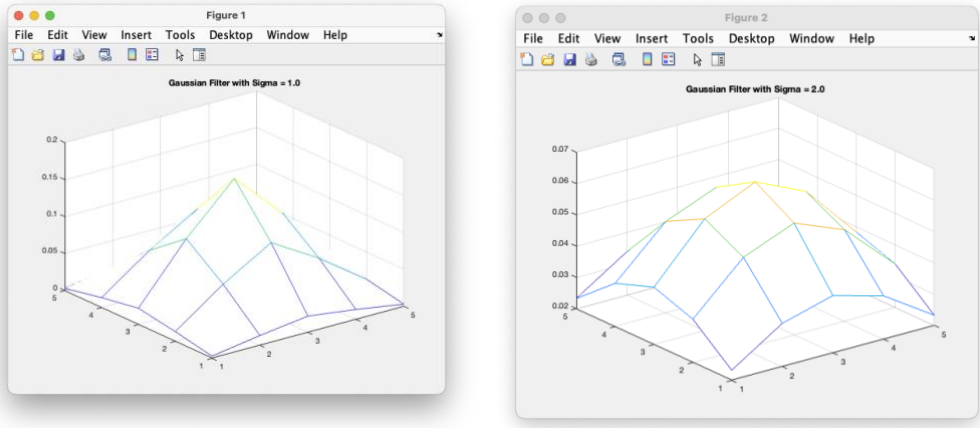


Figure 20. Linear spatial filtering – 5x5 kernels comparison for $\sigma = 1$ and $\sigma = 2$

3.3.2.2 Part b)

MATLAB Code:

The code snippet for this part (Figure 21) is straightforward, starting with reading the image via the **imread** command and then visualizing it using **imshow**.

```
% b) Load and view the noisy image (with Gaussian noise)
noisy_image_gaussian = imread('img/lib-gn.jpg');

% Display the noisy image
figure;
imshow(noisy_image_gaussian);
title('Image with Additive Gaussian Noise');
```

Figure 21. Linear spatial filtering – the code for part b)

Results:

As mentioned in the assignment, the image indeed has additive Gaussian noise. This type of noise is commonly used as a normal distribution of random pixel intensity variations.

Figure 22. Linear spatial filtering – image with additive Gaussian noise

3.3.2.3 Part c)

MATLAB Code:

As seen in Figure 23, I am applying the filters with the help of the **conv2** function, where I pass the image converted to **double** (for computation purposes), the kernel, and the output variant,

where specifying “same” will give the output with same dimensions as the original image (matrix with pixel values). After applying the convolution with the 2 filters, the image matrices must be converted back to **uint8**. Lastly, the filtered images are visualized in the same way as previously.

```
% c) Filter the noisy image using the Gaussian filters

% Apply the first filter (sigma = 1.0) to the noisy image
filtered_image1 = conv2(double(noisy_image_gaussian), h1, 'same');

% Apply the second filter (sigma = 2.0) to the noisy image
filtered_image2 = conv2(double(noisy_image_gaussian), h2, 'same');

% Convert the filtered images back to uint8 for display
filtered_image1 = uint8(filtered_image1);
filtered_image2 = uint8(filtered_image2);

% Display the filtered images
figure;
imshow(filtered_image1);
title('Filtered Image with Gaussian Filter (Sigma = 1.0)');

figure;
imshow(filtered_image2);
title('Filtered Image with Gaussian Filter (Sigma = 2.0)');
```

Figure 23. Linear spatial filtering – the code for part c)

Results:

The noise in both images (Figure 24) is decreased because of the applied Gaussian filters. The convolution with the kernels was able to decrease the additive Gaussian noise. However, both images lost some of the sharpness of the original image. This is normal as the filters are averaging, and thus, they are making the difference between the neighboring pixels smoother. In conclusion, both filters have an effect in clearing the noise, with the second filter removing more noise but with a cost.

There is a clear difference between the results of the 2 filters. The first image (where a kernel with $\sigma = 1$ is used) is sharper than the second one (where a kernel with $\sigma = 2$ is applied). The second image appears blurred because of the increased standard deviation. A higher standard deviation will result in increased weights for the neighboring pixels in the kernel. As a result, the transition between the pixels will become smoother, removing details and adding blur. However, high neighboring weights result in less noise if the center pixel is affected by the noise and the neighbors are not. There is a trade-off between noise removal and image blur.

Figure 24. Linear spatial filtering – Gaussian noise, Gaussian filter for $\sigma = 1$ and $\sigma = 2$

3.3.2.4 Part d)

MATLAB Code:

Figure 25 uses the same repetitive code structure for reading the image and displaying it in a figure.

```
% d) Load and view the noisy image (with speckle noise)
noisy_image_speckle = imread('img/lib-sp.jpg');

% Display the noisy image
figure;
imshow(noisy_image_speckle);
title('Image with Additive Speckle Noise');
```

Figure 25. Linear spatial filtering – the code for part d)

Results:

As mentioned in the assignment, the image indeed has additive speckle noise. This type of noise is also known as salt-and-paper noise and is very common in image processing.

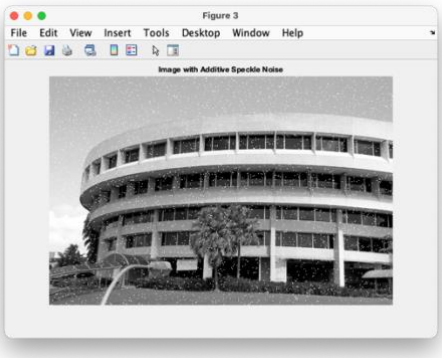


Figure 26. Linear spatial filtering – image with additive speckle noise

3.3.2.5 Part e)

MATLAB Code:

Similarly to part c), I am applying the filters to the image with speckle noise (Figure 27).

```
% e) Filter the speckle noisy image using the Gaussian filters

% Apply the first filter (sigma = 1.0) to the speckle noisy image
filtered_image_speckle1 = conv2(double(noisy_image_speckle), h1, 'same');

% Apply the second filter (sigma = 2.0) to the speckle noisy image
filtered_image_speckle2 = conv2(double(noisy_image_speckle), h2, 'same');

% Convert the filtered images back to uint8 for display
filtered_image_speckle1 = uint8(filtered_image_speckle1);
filtered_image_speckle2 = uint8(filtered_image_speckle2);

% Display the filtered images
figure;
imshow(filtered_image_speckle1);
title('Filtered Speckle Image with Gaussian Filter (Sigma = 1.0)');

figure;
imshow(filtered_image_speckle2);
title('Filtered Speckle Image with Gaussian Filter (Sigma = 2.0)');
```

Figure 27. Linear spatial filtering – the code for part e)

Results:

In contrast to the Gaussian noise, the speckle noise is quite challenging to the Gaussian filter, as seen in Figure 28. The noise here consists of extreme values that cannot be removed from the Gaussian kernel as the center pixel has the highest values, so the extreme brightness will be reflected in the sum of the weighted pixels in the kernel. Even the second filter with $\sigma = 2$ cannot completely remove the bright dots despite having larger coefficients (weights) for the neighboring pixel values.

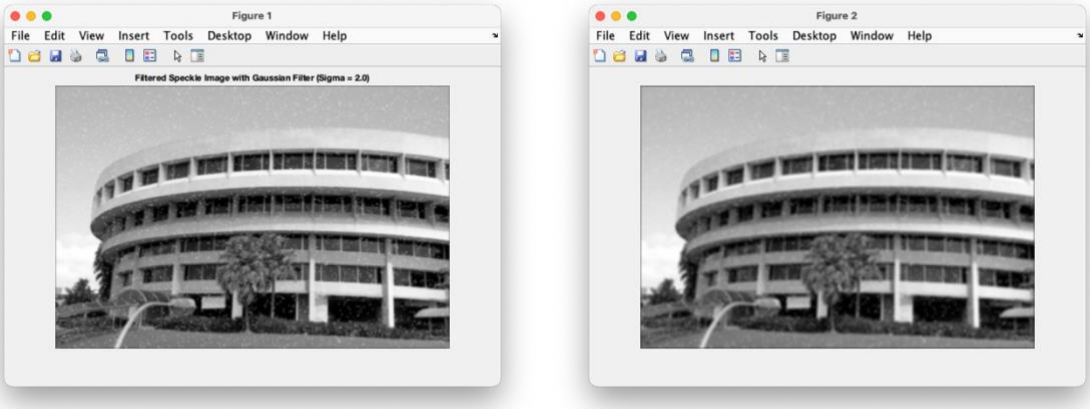


Figure 28. Linear spatial filtering – speckle noise, Gaussian filter for $\sigma = 1$ and $\sigma = 2$

3.4 Median Filtering

3.4.1 Concept

Median filtering is a non-linear filtering commonly used to reduce noise in images, particularly useful for speckle noise (also known as salt-and-pepper noise). It is a special case of order-statistic filtering, where pixel intensities in a local area are sorted, and the pixel at the center is replaced with the median value from this sorted set. Again, the filter is applied by iteration over the image.

3.4.2 MATLAB Code Analysis and Results

3.4.2.1 Parts a), b) and c

MATLAB Code:

The median filter is not a static one, so it cannot be represented and predefined as the Gaussian filter. It is computed at the moment during the iteration over the image. Thus, no code is needed for part a). The code in Figure 29 continues with loading the image with Gaussian noise, completing part b). The new difference from the Gaussian averaging filter code is in applying the median filter to the images. The new filter is not predefined, so the same **conv2** function cannot be used. To apply the median filter, the **medfilt2** function is used. This function takes only 2 parameters: the image matrix and the size of the filter (kernel). It should be noted that in this case, I am not converting the image to a **double** matrix, this is due to the fact that the median filter simply chooses the middle value in the sorted array (does not use any operations that may result in **double** values). Also, because the image is not converted to a **double** matrix, it does not require reverting back to **uint8**. The requirements in the assignment specify 2 filters with 2 different sizes: 3x3 and 5x5. To achieve this, I need to change only the second parameter. Lastly, the resulting images are visualized using the similar approaches from the previous parts.

```
% a) No need to define a new filter

% b) Load and view the noisy image (Gaussian noise)
noisy_image_gaussian = imread('img/lib-gn.jpg');

% Display the noisy image
figure;
imshow(noisy_image_gaussian);
title('Image with Additive Gaussian Noise');

% c) Apply median filtering to the noisy image using different neighborhood sizes

% Apply median filtering with 3x3 neighborhood
median_filtered_3x3_gaussian = medfilt2(noisy_image_gaussian, [3 3]);

% Apply median filtering with 5x5 neighborhood
median_filtered_5x5_gaussian = medfilt2(noisy_image_gaussian, [5 5]);

% Display the filtered images
figure;
imshow(median_filtered_3x3_gaussian);
title('Median Filtered Gaussian Image (3x3 Neighborhood)');

figure;
imshow(median_filtered_5x5_gaussian);
title('Median Filtered Gaussian Image (5x5 Neighborhood)');
```

Figure 29. Median filtering – the code for parts a), b) and c)

Results:

As seen in Figure 30, the median filter with size 3x3 is also relatively good for removing Gaussian noise and does not suffer from the smoothening of the edges caused by the Gaussian averaging filters. However, the median filter with size 5x5, which removes the noise even better, struggles with maintaining the realistic behavior of the image. Because of the median nature of the filter, the edges remain, but the textures are blurred as the most common intensity in the local region is chosen, resulting in animation-like images, not real-life looking.

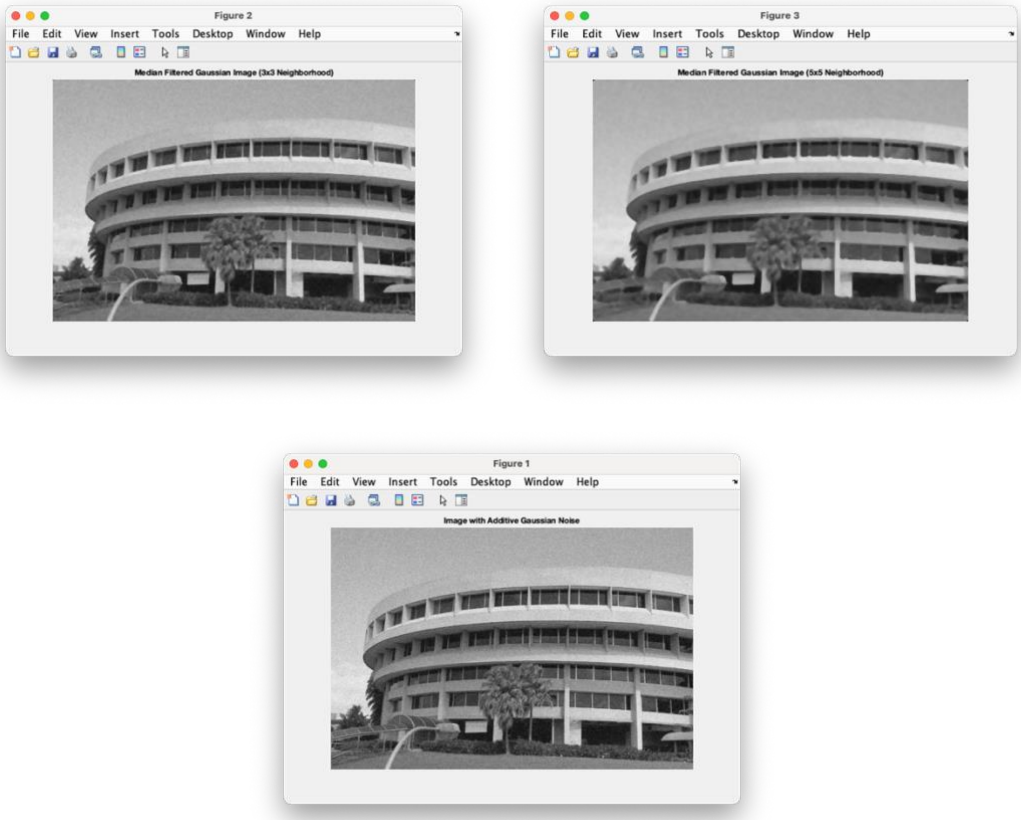


Figure 30. Median filtering – Gaussian noise, kernel sizes 3x3 and 5x5

3.4.2.2 Parts d) and e)

MATLAB Code:

The speckle image is loaded and visualized in the same way as before. As for applying the median filtering, the code structure is exactly the same as for part c). The only difference is the changed first **medfilt2** parameter, which now takes the speckle noise image.

```
% d) Load and view the noisy image (Speckle noise)
noisy_image_speckle = imread('img/lib-sp.jpg');

% Display the noisy image
figure;
imshow(noisy_image_speckle);
title('Image with Additive Speckle Noise');

% e) Apply median filtering to the speckle noisy image

% Apply median filtering with 3x3 neighborhood
median_filtered_3x3_speckle = medfilt2(noisy_image_speckle, [3 3]);

% Apply median filtering with 5x5 neighborhood
median_filtered_5x5_speckle = medfilt2(noisy_image_speckle, [5 5]);

% Display the filtered images
figure;
imshow(median_filtered_3x3_speckle);
title('Median Filtered Speckle Image (3x3 Neighborhood)');

figure;
imshow(median_filtered_5x5_speckle);
title('Median Filtered Speckle Image (5x5 Neighborhood)');
```

Figure 31. Median filtering – the code for parts d) and e)

Results:

Figure 32 shows the results of removing the speckle noise using a median filter. The results are much better than the ones using the Gaussian kernel in Figure 28. However, despite the median filter's ability to preserve the edges, increasing the kernel size will result in textures with fewer details. For example, the little details like the window frames and the leaves of the palm are starting to disappear because they are surrounded by many pixels with different intensities.

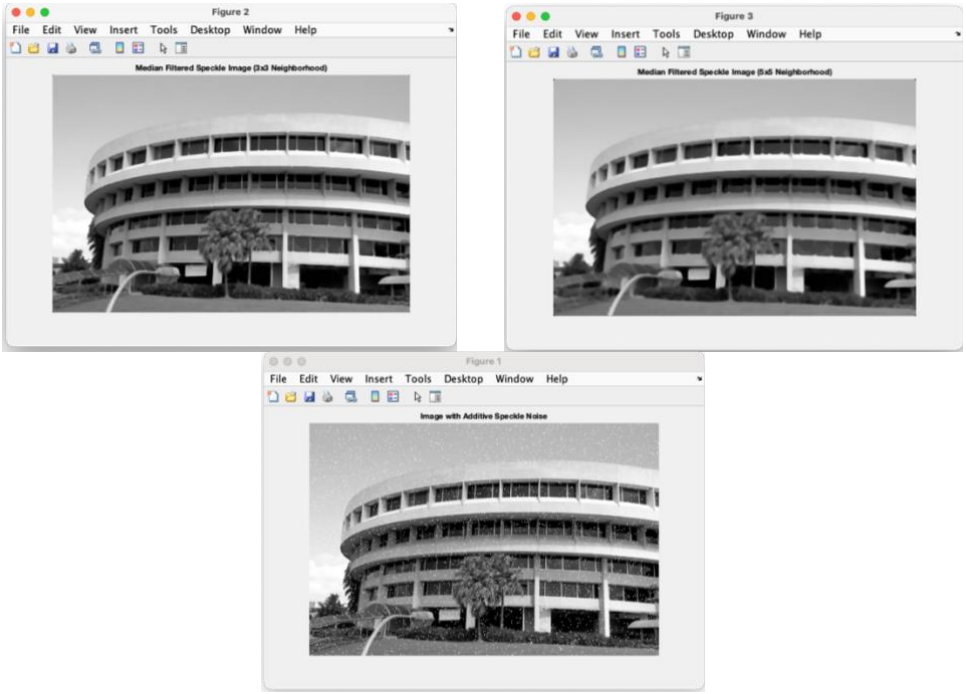


Figure 32. Median filtering – speckle noise, kernel sizes 3x3 and 5x5

3.5 Suppressing Noise Interference Patterns

3.5.1 Concept

Noise interference patterns, such as parallel lines (or stripes), can appear in images or videos due to poor signal reception or hardware malfunctions, such as in old television screens. These interference patterns are corrupting the image quality and human perception because the obstructions hide the details behind them and often make looking at the image harder. Fourier analysis is a powerful technique to spot these patterns and decrease their effect on the image. This section shows how bandpass filtering and Fourier transform can be used to reduce interference patterns effectively.

In the frequency domain, each "pixel" value (or point) represents the amplitude and phase of a specific frequency component in the image. This frequency component corresponds to a sinusoidal wave pattern (a base pattern) at a particular orientation and scale. The weight (or pixel value in the Fourier transform) determines the contribution of that frequency to the overall image. In essence, the image is represented as a sum of these base patterns, and the Fourier transform provides the magnitude and direction of these patterns.

In conclusion, after transforming an image with interference patterns, these patterns can be seen as peaks in the visualization of its transformation in the frequency domain. Thus, if these heights are correctly located and removed, the image quality will be greatly improved.

3.5.2 MATLAB Code Analysis and Results

3.5.2.1 Part a)

MATLAB Code:

The code for part a) (Figure 31) simply loads the new image specified by the assignment.

```
% a) Load and display the image with interference patterns
interference_image = imread('img/pck-int.jpg');

% Display the interference image
figure;
imshow(interference_image);
title('Original Image with Interference Patterns');
```

Figure 33. Suppressing noise interference patterns – the code for part a)

Results:

The original image with diagonal interference parallel stripes can be seen in Figure 34.

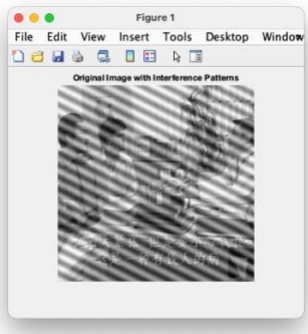


Figure 34. Suppressing noise interference patterns – the original image with interference

3.5.2.2 Part b)

MATLAB Code:

The code (Figure 35) starts directly with computing the Fourier transform of the image, and this is done using the **fft2** function, which accepts an image with **double** values for the pixels. The **fft2** function returns the 2-dimensional Fourier transform of the image. However, this result is a complex number and contains of 2 parts: a real and an imaginary part. The Fourier transform $F(u,v)$ for frequency (u,v) can be represented as:

$$F(u,v) = \mathrm{Re}\big(F(u,v)\big) + i \cdot \mathrm{Im}\big(F(u,v)\big)$$

By definition, the absolute value of a complex number $z = a + bi$ is $|z| = \sqrt{a^2 + b^2}$. Thus, the magnitude of the Fourier transform $F(u,v)$ can be formulated as:

$$|F(u,v)| = \sqrt{\mathrm{Re}\big(F(u,v)\big)^2 + \mathrm{Im}\big(F(u,v)\big)^2}$$

As for the power spectrum, in signal and image processing, power refers to how much energy is carried by each frequency component. Thus, the power of a frequency component is defined as the square of its amplitude (or magnitude), meaning that I can represent the power using the following expression:

$$\mathrm{Power}(u,v) = |F(u,v)|^2 = \mathrm{Re}\big(F(u,v)\big)^2 + \mathrm{Im}\big(F(u,v)\big)^2$$

In the code, I follow the same logic, and after obtaining the 2-dimensional Fourier transform of the image, I compute the power spectrum as the element-wise square (**.^2**) of the computed absolute values using **abs**.

This part of the experiment requires shifting the computed power spectrum and raising its elements to the power of 0.1. To achieve the shifting, the **fftshift** function is applied on the scaled power spectrum. Lastly, the spectrum is visualized in a figure.

```
% b) Compute the Fourier Transform and power spectrum

% Compute the 2D Fourier Transform of the image
F = fft2(double(interference_image));

% Compute the power spectrum
S = abs(F).^2;

% Shift the zero-frequency component to the center and display the power spectrum
figure;
imagesc(fftshift(S.^0.1));
colormap('default');
title('Power Spectrum of Interference Image (with fftshift)');
```

Figure 35. Suppressing noise interference patterns – the code for part b)

Results:

As seen in the shifted power spectrum in Figure 36, there are 2 peaks located diagonally apart from the center low-frequency peak region. The diagonal location of these 2 peaks, according to the center of the figure, describes the nature of the patterns in the image. By definition, moving along the x and y axis observes the vertical and horizontal stripes patterns. Moving diagonally gives us the patterns that appear diagonally on the image. In conclusion, it can be deduced that exactly these 2 points are causing the majority of the interference in the image. Also, the differences between the peaks and the low values are clearer as I am using the power spectrum (the values are squared, resulting in a larger difference).

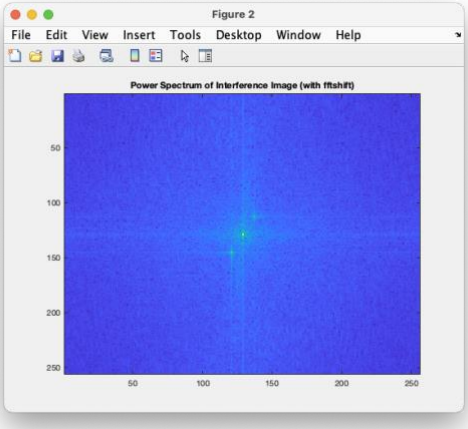


Figure 36. Suppressing noise interference patterns – shifted power spectrum visualization

3.5.2.3 Part c)

MATLAB Code:

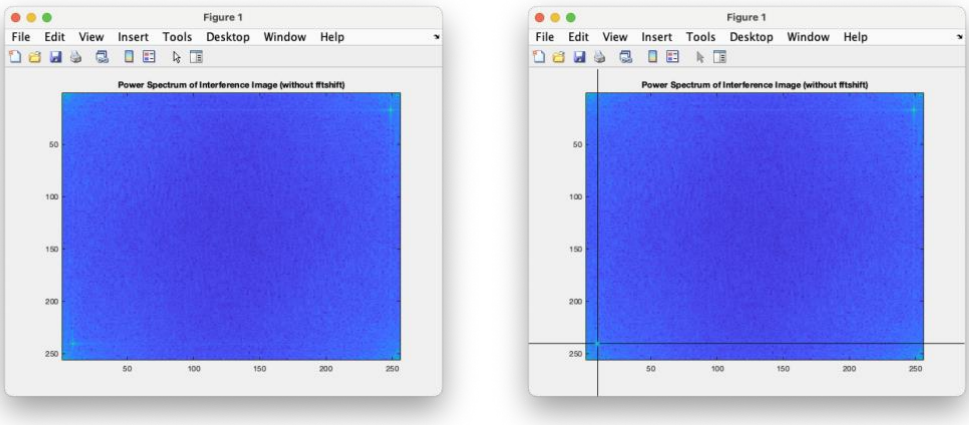
As seen in our code segment in Figure 37, I am repeating the visualization of the power spectrum but without using **fftshift** to meet the requirements of this part of the experiment. Continuing with the code, I use the **disp** command to display a message in the command window for the user to enter 2 coordinates (by clicking) on the visualized figure with the power spectrum. The actual coordinates are retrieved by the **ginput** command, where I specify that 2 inputs are needed. The function will listen for 2 clicks on the figure and store their x and y coordinates. Lastly, the coordinates are printed in the command window using a formatted string.

```
% c) Redisplay the power spectrum without fftshift and measure peaks

% Display the power spectrum without fftshift
figure;
imagesc(S.^0.1);
colormap('default');
title('Power Spectrum of Interference Image (without fftshift)');

% Use ginput to measure the coordinates of the peaks manually
disp('Use ginput to select the frequency peaks (2 peaks should be selected)');
[peak_x, peak_y] = ginput(2);

% Display the coordinates
fprintf('Selected peak coordinates: (%.1f, %.1f) and (%.1f, %.1f)\n', peak_x(1), peak_y(1), peak_x(2), peak_y(2));
```

Figure 37. Suppressing noise interference patterns – the code for part c)

Results:

The left plot in Figure 38 shows the non-shifted power spectrum, when no shifting is applied, the low-frequency patterns appear in the corners of the plot. The picture on the right shows the user interface (2 black lines intersecting with each other on the plot), allowing the user to select a point in the plot. The non-shifted spectrum holds the same information about the original image as the shifted one. The screenshot of the console below the 2 plots shows that the user was asked to choose 2 peak points, and their coordinates were successfully picked. The selection of the 2 points is independent of the shifting of the power spectrum, both the shifted and the untouched spectrums hold the same values. Thus, the key factor is to spot the exact points that hold the large weights corresponding to the diagonal stripes.




```
Use ginput to select the frequency peaks (2 peaks should be selected)
Selected peak coordinates: (8.9, 240.4) and (249.1, 17.0)
```

Figure 38. Suppressing noise interference patterns – non-shifted power spectrum and input coordinates for the diagonal peaks

3.5.2.4 Part d)

MATLAB Code:

The code for part d) (Figure 39) continues with setting the 5x5 regions around the selected peak points to 0. First, the window size is defined indirectly by setting the neighborhood size to 2, which means the affected pixels will be at most 2 units away (on each axis) from the center one. I am creating a copy of the Fourier transform where the selected regions will be set to 0. Also, it should be noted that the power spectrum is not altered, it is used only for visualization purposes. Then, the **for** loop in the next code lines iterates over the selected peaks (in this case, there are 2 selected peaks). In the inner part of the loop, I am rounding the x and y coordinates of the selected points because I will use them as indexes in the Fourier transform matrix, and they must be integers. For each selected peak, I am setting the peak region (from the selected center up to the pixels that are 2 units far on each axis) to 0. Then, the power spectrum is recomputed using the new Fourier transform with the 0 values, it is shifted and visualized in the same way as before.

```
% d) Set the 5x5 neighborhood around the peaks to zero in the Fourier domain

% Set a 5x5 neighborhood around each peak to zero in the Fourier transform F
neighborhood_size = 2;
F_filtered = F;

for i = 1:length(peak_x)
    x = round(peak_x(i));
    y = round(peak_y(i));

    % Zero out a 5x5 neighborhood around each peak
    F_filtered(y-neighborhood_size:y+neighborhood_size, x-neighborhood_size:x+neighborhood_size) = 0;
end

% Recompute the power spectrum after filtering
S_filtered = abs(F_filtered).^2;

% Display the filtered power spectrum
figure;
imagesc(fftshift(S_filtered.^0.1));
colormap('default');
title('Filtered Power Spectrum (after removing peaks)');
```

Figure 39. Suppressing noise interference patterns – the code for part d)

Results:

Figure 40 supports the statement that choosing the peak points is independent of the shifting of the spectrums when visualizing. Despite the fact that the regions were selected on the non-shifted spectrum, the setting to zero is correctly reflected in the shifted power spectrum. After the regions are set to zero, 2 darker rectangles can be observed on the plot, they appear so dark as the values they represent are simply 0.

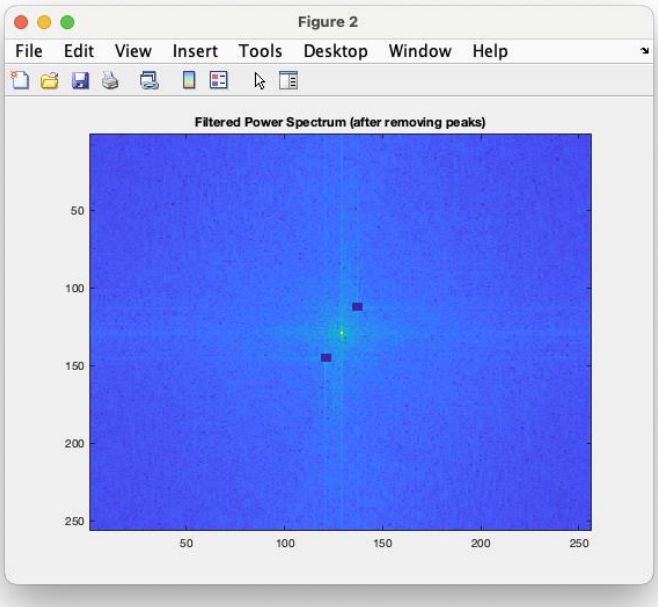


Figure 40. Suppressing noise interference patterns – the power spectrum of the Fourier transform after removing the 2 peaks 5x5 regions related to the diagonal stripes

3.5.2.5 Part e)

MATLAB Code:

As shown in Figure 41, the inversion of the Fourier transform is done using the **ifft2** function and passing the filtered Fourier transform with the 2 5x5 zero regions. After, I am getting the real part of the result (to avoid any warning when passing to the next function), and I am converting it to a **uint8** type so it can be successfully presented as an image using **imshow**.

```
% e) Compute the inverse Fourier transform and display the resultant image

% Compute the inverse Fourier transform to get the filtered image
filtered_image = ifft2(F_filtered);

% Take the real part of the result and convert it back to uint8
filtered_image = real(filtered_image);
filtered_image = uint8(filtered_image);

% Display the filtered image
figure;
imshow(filtered_image);
title('Filtered Image after Removing Interference');
```

Figure 41. Suppressing noise interference patterns – the code for part e)

Results:

Figure 42 shows the enhanced image after removing 2 5x5 regions corresponding to the diagonal stripes interfering with the original image. The new image is not as affected by the interference as the original, but there are still some patterns around the borders. After all, only 2 regions are removed, while the pattern is affecting other regions too. The spectrum in Figure 40 shows vertical and horizontal peak lines (perpendicular to each other), and starting from the selected peak points, these lines also correspond to the interference pattern. The result is good considering the simplicity of the approach, but for better results, a more complex methodology is needed.

Figure 42. Suppressing noise interference patterns – the enhanced image after removing the diagonal stripes patterns in the Fourier transform

3.5.2.6 Part e)

MATLAB Code:

In order to handle the more sophisticated nature of the interference in the provided image for part e), I have implemented a more complex approach, allowing the user to select multiple 5x5 windows. The code segment for this part is larger, so it will be discussed in the section.

The first section (Figure 43) loads the specific image for this part and performs a check on its size. By checking the third dimension, I am finding the number of channels in the image. If they are 3 (RGB), then the image is converted to a grayscale. Then, the grayscale image is visualized. Then, the 2D Fourier transform is computed in the same manner as in the previous parts of the experiment. Using the result from the **fft2** function, I am again computing the power spectrum by raising the elements of the Fourier transform (matrix) to the power of 2. However, this time, I am applying a different strategy for a better visualization, instead of again raising the elements to the power of 0.1 I am passing all elements through a logarithmic function and adding 1 to all of them to prevent zeros in the **log** function. In the end, I am plotting the shifted power spectrum (using **fftshift**), also, this time, I add a bar to the right of the plot to show the relation between color and the values in the plot. The **hold on** command at the end of the code segment is used to allow drawing multiple rectangles when choosing the peak points later.

```
% f) Filter the 'primate-caged.jpg' image to attempt removing the cage

% Load the primate-caged image
primate_image = imread('img/primate-caged.jpg');

% Check if the image is RGB, convert it to grayscale if necessary
if size(primate_image, 3) == 3
    primate_image = rgb2gray(primate_image);
end

% Display the original image
figure;
imshow(primate_image);
title('Original Primate Caged Image');

% Compute the 2D Fourier Transform of the grayscale image
F_primate = fft2(double(primate_image));

% Compute the power spectrum of the primate image
S_primate = abs(F_primate).^2;

% Apply logarithmic scaling to the power spectrum for better visibility
S_primate_log = log(1 + S_primate);

% Display the power spectrum using logarithmic scaling
figure;
imagesc(fftshift(S_primate_log));
colormap('default');
title('Logarithmic Power Spectrum of Primate Image (with fftshift)');
colorbar;
hold on;
```

Figure 43. Suppressing noise interference patterns – section 1 of the code for part f)

The second section of the code (Figure 44) deals with the user input. First, the array used for storing the selected regions is defined, and the region size is defined using the neighborhood size ($window\ size\ =\ 2\ \times\ neighborhood\ size\ +\ 1$). Then, a message is displayed telling the user to click on the points they want to zero out and press "Enter" when they are ready. In the inner body of the **while** loop, I am using the **ginput** function again, but this time I am getting all 3 outputs from it. The third output is defined as the mouse button or key on the keyboard used to select each point. The following **if-else** statement checks if the user clicked on the plot or pressed the "Enter" button. If the "Enter" is pressed, the loop is broken. Otherwise, the system continues to accept coordinates. For each new pair of coordinates, the array is extended with the new pair on top of the previously selected pairs. Then, each selected point is visualized using the **plot** and **rectangle** commands. The **plot** commands take the 2 coordinates of the selected point, the parameter **"rx"** which defines the **red** color of the mark, and the **cross** style of the mark, lastly, I am specifying the marker size and the line width. The **rectangle** command draws the window size of the region with a center – the selected point (based on the formula described above). The edge color is set to **green** ("g"), and line width is specified. Lastly, the **drawnow** command is used to reflect the changes immediately on the current plot so the user knows which regions have been selected already.

```
% Initialize arrays to store selected points and window sizes
selected_points = [];
neighborhood_size = 2;

% Loop until the user presses Enter
disp('Click on points to zero out. Press Enter when done.');
while true
    [x, y, button] = ginput(1);
    if isempty(button)
        break;
    end

    % Store the selected point
    selected_points = [selected_points; x, y];

    % Plot the selected point (cross) on the power spectrum
    plot(x, y, 'rx', 'MarkerSize', 10, 'LineWidth', 2);

    % Draw a rectangle around the selected point to indicate the window
    rectangle('Position', [x - neighborhood_size - 0.5, y - neighborhood_size - 0.5, ...
              2*neighborhood_size + 1, 2*neighborhood_size + 1], 'EdgeColor', 'g', 'LineWidth', 1.5);

    % Update the plot immediately
    drawnow;
end
```

Figure 44. Suppressing noise interference patterns – section 2 of the code for part f)

Figure 45 presents the last section of the code for part f). First, a copy of the Fourier transformer to hold the shifter transformer (used in the visualization – low frequencies in the center). This will allow immediate work on it using the coordinates of the selected points and then simply shift it again later. I am also getting the number of columns and rows in the Fourier transform because it will be useful later. Using the same logic from part d), I am setting the weights for the pixels in the windows to 0. However, this time, I have added 2 lines of code to ensure the stable behavior of the program with any user input. These lines of code handle the edge cases when the user has selected points outside of the image or near its edge when the 5x5 window exceeds the image limits. Using the defined columns and rows variables, 2 requirements are constructed for both x and y axis coordinates. The logic behind the 2 statements basically allows x and y values only in the range between 3 (neighborhood size plus 1 for the center) and the maximum value for the axis (the number of columns or rows) minus 2. This logic is aligned with the 1-based indexing in MATLAB. After altering the Fourier transform by setting the selected regions to 0, I am again getting its power spectrum in the same way as I have done before. The new power spectrum is again rescaled using a **log** function for better visualization. Then, it is plotted so the removed regions can be observed. In this section of the code, all calculations and actions were performed on the shifted Fourier transform because it was the shifter power spectrum that was visualized for the user's selection. Thus, the filtered image is retrieved using the inverse transform operation to the shifter Fourier transform after it is inverse shifted. **ifft2** can return a valid image only if it is passed a non-shifted transform matrix (low frequencies in the corners). However, this is not enough to retrieve the image because the **ifft2** function outputs complex numbers. First, only the real part of the complex numbers is considered (using **real** command), and then, the matrix values should be converted to integers with type **uint8**. I have also added one more step using **mat2gray** that assures the values are in the intensity range of 0 to 1, which can be converted to the desired **uint8** range by multiplying with the maximum value of 255. Having the image matrix in the wanted format, it can be displayed again using **imshow**.

```
% Initialize the filtered Fourier transform
F_primate_filtered = fftshift(F_primate);  % Work with shifted Fourier transform directly

% Get the number of rows and columns
[rows, cols] = size(F_primate_filtered);

% Zero out a 5x5 neighborhood around each selected point
for i = 1:size(selected_points, 1)
    x = round(selected_points(i, 1));
    y = round(selected_points(i, 2));

    % Ensure the indices are valid and within bounds
    x = max(neighborhood_size + 1, min(cols - neighborhood_size, x));
    y = max(neighborhood_size + 1, min(rows - neighborhood_size, y));

    % Zero out a 5x5 neighborhood around the selected point
    F_primate_filtered(y-neighborhood_size:y+neighborhood_size, x-neighborhood_size:x+neighborhood_size) = 0;
end

% Visualize the modified power spectrum after zeroing out the selected areas
S_primate_filtered = abs(F_primate_filtered).^2;  % Recompute the power spectrum after filtering
S_primate_filtered_log = log(1 + S_primate_filtered);  % Apply logarithmic scaling for better visualization

figure;
imagesc(S_primate_filtered_log);  % Display the filtered power spectrum
colormap('default');
title('Filtered Power Spectrum (after removing selected areas)');
colorbar;

% Compute the inverse Fourier transform to get the filtered primate image
primate_filtered_image = ifft2(ifftshift(F_primate_filtered));  % Shift back and compute inverse Fourier transform

% Take the real part and convert back to uint8
primate_filtered_image = real(primate_filtered_image);
primate_filtered_image = uint8(mat2gray(primate_filtered_image) * 255);

% Display the filtered primate image
figure;
imshow(primate_filtered_image);
title('Filtered Primate Image (after removing multiple selected areas)');
```

Figure 45. Suppressing noise interference patterns – section 3 of the code for part f)

Results:

Figure 46 shows the image that is specified by the assignment for part f), as it can be seen in the power spectrum, the "cage" interference pattern results in a more complex power spectrum visualization with more peak regions.

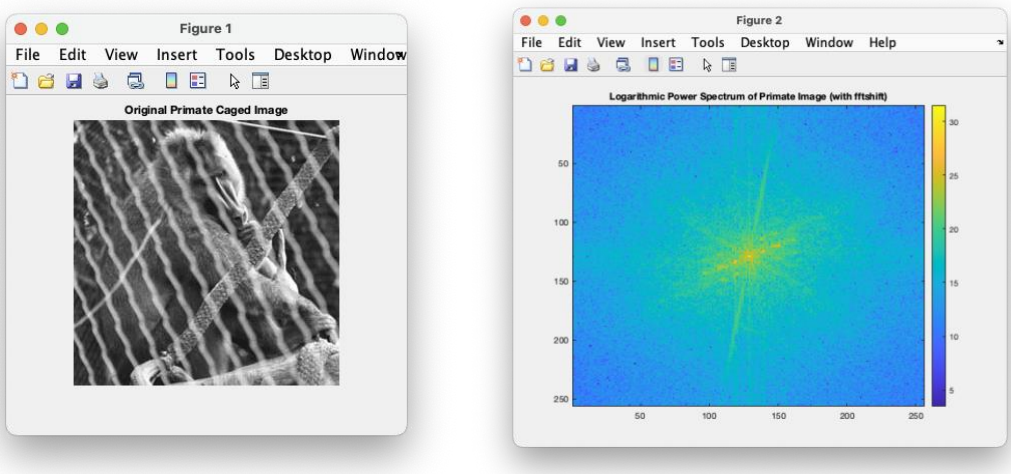


Figure 46. Suppressing noise interference patterns – the original image for part f) and its shifted power spectrum with logarithmic scale

Figure 47 presents the visualization of the user's inputs on the power spectrum plot. In order to maximize the enhancement of the image, the user should select the peaks around the center of the plot, they are more compared to the ones caused by the diagonal stripes. Thus, further analysis should be performed.

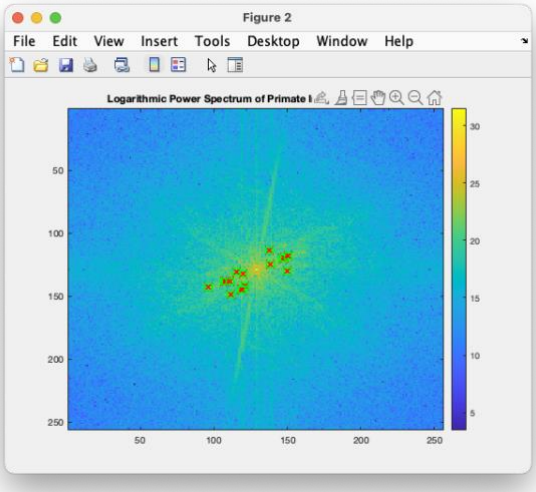


Figure 47. Suppressing noise interference patterns – user inputs (region selection) on the shifted power spectrum with logarithmic scale

As can be seen in Figure 48, the user's inputs have successfully set the regions to 0 (visible on the plot on the right), and the resulting image of applying inverse transform is indeed enhanced, the "cage" pattern's effect is minimized, but the image still has some leftovers from it. This can be further improved by either choosing to zero out more regions or using some more sophisticated masks or filters for the frequency domain.

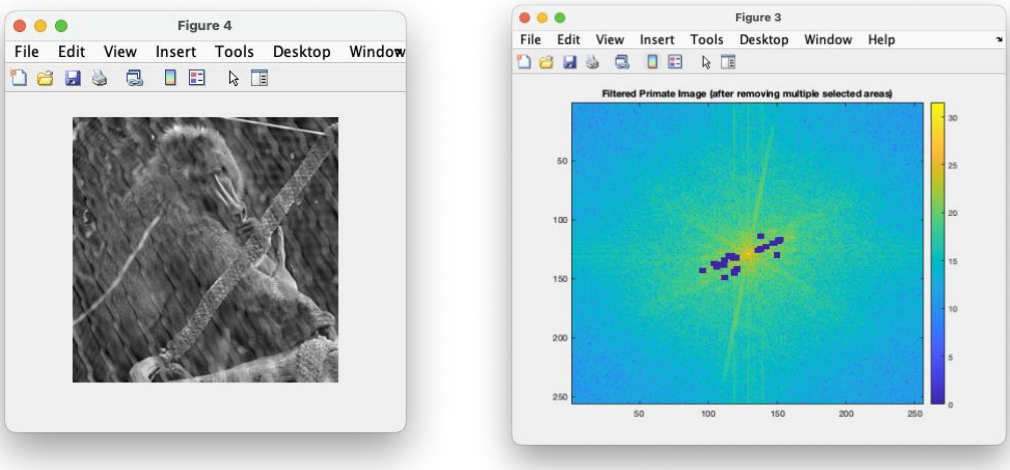


Figure 48. Suppressing noise interference patterns – filtered image and the filtered power spectrum with the selected regions set to 0

3.6 Perspective Distortion of Planar Surface

3.6.1 Concept

2D planar projective transformation is a mathematical technique that allows us to map points from one plane (the slanted view) to another (the front view) using a projective transformation matrix.

A projective transformation is defined by a 3x3 matrix that relates the coordinates of points in the original, distorted image (denoted as X_w and Y_w) to those in the rectified, or transformed image (denoted as x_{im} and y_{im}). This transformation accounts for both rotation and scaling effects in perspective, which are typically introduced when a flat object, like a book, is viewed from an angle. The transformation is represented as:

$$\begin{bmatrix} kx_i m \\ ky_i m \\ k \end{bmatrix} = \begin{bmatrix} m_{11} m_{12} m_{13} \\ m_{21} m_{22} m_{23} \\ m_{31} m_{32} \quad 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix}$$

Where k is a scaling factor, and the m-values are the unknown elements of the transformation matrix. These unknowns must be calculated based on corresponding points in the slanted image and the desired rectified image.

To calculate the projective transformation matrix, at least four corresponding points are needed between the slanted and rectified images. These points are selected manually from the distorted image. The desired coordinates are based on the actual dimensions of the object (for example, the A4 book's dimensions of 210 mm x 297 mm), and by solving a system of linear equations, the transformation matrix is obtained.

3.6.2 MATLAB Code Analysis and Results

3.6.2.1 Part a)

MATLAB Code:

The code for part a) (Figure 49) simply loads the specified by the assignment image for this experiment.

```
% a) Load and display the slanted image of the book
P = imread('img/book.jpg');  % Load the slanted image of the book
figure;
imshow(P);
title('Original Slanted Image of the Book');
```

Figure 49. Perspective Distortion of Planar Surface – the code for part a)

Results:

Figure 50 shows the visualized image used for this experiment. The image contains a slanted view of a book. The book is not rotated, so I am defining the order of selecting the corners as the top left, top right, bottom right, and bottom left.

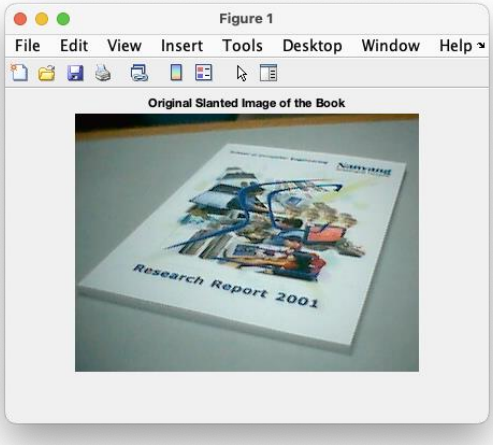


Figure 50. Perspective Distortion of Planar Surface – the original image for the experiment

3.6.2.2 Part b)

MATLAB Code:

As can be observed in Figure 51, apart from taking the user input for the four corners of the book, the code also defines the known coordinates of the desired output image. Specifically, after selecting the points in the distorted image using the **ginput** function, the code automatically sets the x and y coordinates of the known points in the corrected frontal view. The x and y vectors represent the coordinates of the corners of the A4 sheet in the output image, where the perspective distortion will be corrected. These coordinates are set according to the real-world dimensions of an A4 sheet, which measures 210 mm by 297 mm.

```
% b) Use ginput to manually select the four corners of the book
disp('Select the four corners of the book in the image');
[X, Y] = ginput(4);  % Manually select four corners of the book

% Define the desired coordinates for the corners of an A4 sheet (210 mm x 297 mm)
% X and Y represent the corners of the book in the original image
% x and y represent the corners of the book in the corrected (frontal) image
x = [0, 210, 210, 0];  % X-coordinates of the desired output
y = [0, 0, 297, 297];  % Y-coordinates of the desired output
```

Figure 51. Perspective Distortion of Planar Surface – the code for part b)

Results:

Figure 52 shows the user interface for choosing the coordinates of the corner. The known coordinates are set according to the real-world dimensions of an A4 sheet, which measures 210 mm by 297 mm. Using the corner order from the previous part, the following coordinates are known for the desired projective:

- top left corner: (0, 0)
- top right corner: (210, 0)
- bottom right corner: (210, 297)
- bottom left corner: (0, 297)

After receiving the 4 user input coordinates for 8 total, the calculation of the unknown **m** values is possible.

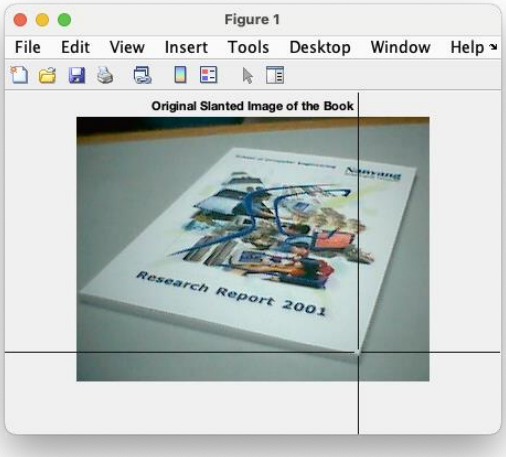


Figure 52. Perspective Distortion of Planar Surface – user corner selection

3.6.2.3 Part c)

MATLAB Code:

Figure 53 covers the code for setting the matrices needed for the equation ($Au = v$) and then solving for the unknown transformation **m** parameters (the u matrix). Using the structure of the matrices (described in the assignment):

$$\underbrace{\begin{bmatrix} X_w^1 & Y_w^1 & 1 & 0 & 0 & 0 & -x_{im}^1 X_w^1 & -x_{im}^1 Y_w^1 \\ 0 & 0 & 0 & X_w^1 & Y_w^1 & 1 & -y_{im}^1 X_w^1 & -y_{im}^1 Y_w^1 \\ X_w^2 & Y_w^2 & 1 & 0 & 0 & 0 & -x_{im}^2 X_w^2 & -x_{im}^2 Y_w^2 \\ 0 & 0 & 0 & X_w^2 & Y_w^2 & 1 & -y_{im}^2 X_w^2 & -y_{im}^2 Y_w^2 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ X_w^n & Y_w^n & 1 & 0 & 0 & 0 & -x_{im}^n X_w^n & -x_{im}^n Y_w^n \\ 0 & 0 & 0 & X_w^n & Y_w^n & 1 & -y_{im}^n X_w^n & -y_{im}^n Y_w^n \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{31} \\ m_{32} \end{bmatrix}}_{u} = \underbrace{\begin{bmatrix} x_{im}^1 \\ y_{im}^1 \\ x_{im}^2 \\ y_{im}^2 \\ . \\ . \\ x_{im}^n \\ y_{im}^n \end{bmatrix}}_{v}$$

First, I am initializing the empty matrix A and the empty vector v. Then a **for** loop is constructed for each pair of selected coordinates (X_w, Y_w) and their corresponding points in the desired output image (x_{im}, y_{im}), in the inner body of the loop 2 rows are added to the A matrix:

$$\begin{bmatrix} \mathrm{X_w} & \mathrm{Y_w} & 1 & 0 & 0 & 0 & -\mathrm{X_w} \cdot \mathrm{x_{im}} & -\mathrm{Y_w} \cdot \mathrm{x_{im}} \\ 0 & 0 & 0 & X_w & Y_w & 1 & -\mathrm{X_w} \cdot \mathrm{y_{im}} & -\mathrm{Y_w} \cdot \mathrm{y_{im}} \end{bmatrix}$$

and 2 rows are added to the vector v: $\begin{bmatrix} x_{im} \\ y_{im} \end{bmatrix}$.

After the loop, I use MATLAB's built-in linear equation solver to compute the transformation parameters by solving the system of equations $u = A \setminus v$, where A is the matrix of known points and v is the vector that contains the corresponding target coordinates.

After computing the transformation parameters vector u, the result is initially in the form of a single-column vector. To convert this into the standard 3x3 projective transformation matrix, I am using MATLAB's **reshape** function, which reorganizes the vector into matrix form by adding a 1 in the bottom-right corner (added as the last element in the vector, thus the last one

in the matrix). This reshaping is necessary because the projective transformation is expressed as a 3x3 matrix, which allows for a full 2D transformation, including translation, scaling, and perspective distortion.

Next, I am applying the transformation matrix U to the input points by multiplying them with the coordinates of the original points (the ones selected by the user). To bring the transformed points back from homogeneous coordinates to standard 2D coordinates, I am dividing the resulting points by their third coordinate. This ensures that the transformed points are correctly mapped to their new positions in the output image.

```
% c) Set up the matrices to estimate the projective transformation
A = [];
v = [];

for i = 1:4
    A = [A; X(i), Y(i), 1, 0, 0, 0, -x(i)*X(i), -x(i)*Y(i)];
    A = [A; 0, 0, 0, X(i), Y(i), 1, -y(i)*X(i), -y(i)*Y(i)];
    v = [v; x(i); y(i)];
end

% Solve for the transformation parameters (u)
u = A \ v;

% Convert the solution vector into the 3x3 projective transformation matrix U
U = reshape([u; 1], 3, 3)';

% Verify the transformation by transforming the original coordinates
w = U * [X'; Y'; ones(1, 4)];
w = w ./ (ones(3,1) * w(3,:));  % Normalize to get the transformed points

% Display the transformed coordinates
disp('Transformed coordinates:');
disp(w);
```

Figure 53. Perspective Distortion of Planar Surface – the code for part c)

Results:

As can be clearly seen in Figure 54, the transformation is working as it returns the correct transformed coordinates, equal to the predefined ones for the desired output image.

```
Transformed coordinates:
   -0.0000  210.0000  210.0000         0
         0         0  297.0000  297.0000
    1.0000    1.0000    1.0000    1.0000
```

Figure 54. Perspective Distortion of Planar Surface – the transformed coordinates

3.6.2.4 Parts d) and e)

MATLAB Code:

The code for these parts (Figure 55) follows the suggestions from the assignment. First, a projective transformation object is created by passing the transformation matrix U to the **maketform** function. Then, **imtransform** applies the projective transformation to the original image, where "XData", $[0 \quad 210]$ and "YData", $[0 \quad 297]$ arguments specify the desired dimensions of the output image in pixels. In this case, the output image will have a scale of 1 pixel corresponding to 1 mm, matching the physical dimensions of an A4-sized sheet (210 mm x 297 mm). Lastly, the output image from the **imtransform** operation is visualized in a figure.

```
% d) Apply the projective transformation to warp the image
T = maketform('projective', U');  % Create the projective transformation
P2 = imtransform(P, T, 'XData', [0 210], 'YData', [0 297]);  % Warp the image

% e) Display the result
figure;
imshow(P2);
title('Warped Image (Frontal View of the Book)');
```

Figure 55. Perspective Distortion of Planar Surface – the code for part d)

Results:

Figure 56 shows that the code is working, and the logic is right. The output transformed image is in the desired A4 size and has a correct orientation. The quality is good, and the content on the transformed image is properly aligned (both horizontally and vertically). The only problem is that some of the sections on the book cover are blurred. After a further inspection, these are the sections that were further from the camera when taking the original photo. This is also expected because they require the same resolution, however, because of the perspective and the further distance to the camera, these regions occupy fewer pixels than the closer ones. The blur is a result of the fact that the information in the original image about the further regions is simply less than the close ones.

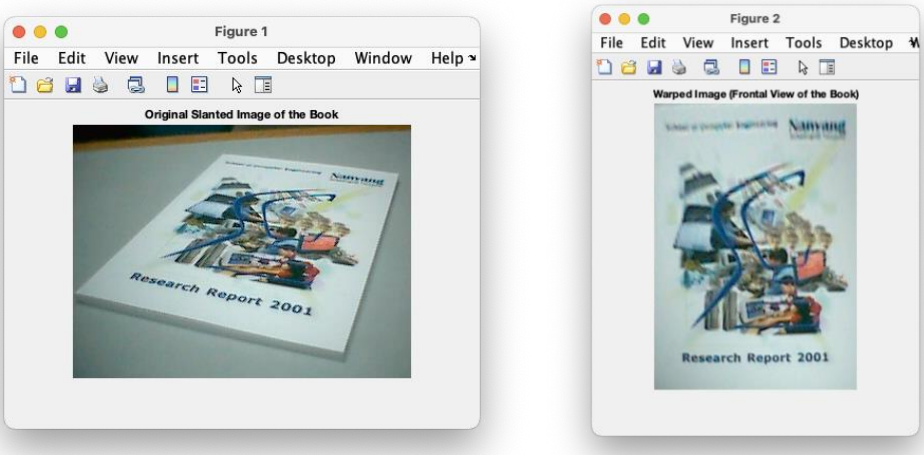


Figure 56. Perspective Distortion of Planar Surface – the comparison between the transformed image and the original one

3.6.2.5 Part f)

MATLAB Code:

The described object in the assignment has a unique color, so I will search for it using the HSV channels of the image. First, the code (Figure 57) starts with the **rgb2hsv** function, which I use to convert the image to HSV, this will enable searching for the wanted color. Then, the ranges for the hue, saturation, and value (brightness) of the wanted color are defined. I am using ranges because the searched object is not small and includes variations of similar colors. The searched color mask represents a binary matrix where each cell tells if the pixel HSV properties are in the specified color range (using the conditions for the minimum and maximum value for each HSV channel).

Then, I am applying morphological operations to clean up the binary mask, improving its accuracy by removing noise and refining the detected regions. The first operation, **imclose**, performs a morphological closing, which helps fill small gaps or holes within the detected regions by smoothing out the mask and connecting nearby regions. This is particularly useful for ensuring that the detected areas are solid and continuous (needed to connect the two parts of the screen divided by the blue curve). Afterward, **imopen** is used to remove small noise or irrelevant objects from the mask, eliminating tiny, isolated artifacts while preserving the important regions.

```
% f) Predefined color selection and matching in the warped image

% Convert the entire image to HSV for color detection
P2_hsv = rgb2hsv(P2);

% Define the updated target color HSV range for reddish, pink, and orange hues
hue_min = 0 / 360;   % Start from 0 degrees (red)
hue_max = 25 / 360;  % Extend up to 25 degrees (covering reddish-orange)
sat_min = 30 / 100;  % Allow lower saturation to capture light pinks and faded colors
sat_max = 100 / 100; % Max saturation to capture vibrant colors
val_min = 60 / 100;  % Allow darker values to include deeper reds
val_max = 100 / 100; % Max brightness to capture light shades like pink or orange

% Create a binary mask based on the updated color thresholds
color_mask = (P2_hsv(:,:,1) >= hue_min) & (P2_hsv(:,:,1) <= hue_max) & ...
             (P2_hsv(:,:,2) >= sat_min) & (P2_hsv(:,:,2) <= sat_max) & ...
             (P2_hsv(:,:,3) >= val_min) & (P2_hsv(:,:,3) <= val_max);

% Perform morphological operations to clean up the mask (remove noise, close gaps)
color_mask_cleaned = imclose(color_mask, strel('rectangle', [10, 10]));
color_mask_cleaned = imopen(color_mask_cleaned, strel('rectangle', [5, 5]));

% Label connected components in the mask
connected_components = bwconncomp(color_mask_cleaned);

% Measure properties of the connected components (such as area and bounding box)
region_stats = regionprops(connected_components, 'BoundingBox', 'Area');

% Check if any regions are found
if ~isempty(region_stats)
    % Find the largest component, assuming it corresponds to the target area
    [~, largest_component_idx] = max([region_stats.Area]);
    bounding_box = region_stats(largest_component_idx).BoundingBox;

    % Display the original warped image
    figure;
    imshow(P2);
    title('Warped Image with Detected Color Region');
    hold on;

    % Draw a rectangle around the largest detected color region
    rectangle('Position', bounding_box, 'EdgeColor', 'g', 'LineWidth', 2)

    red_channel = ones(size(P2, 1), size(P2, 2));    % Red channel (all ones for full red)
    green_channel = zeros(size(P2, 1), size(P2, 2));  % Green channel (all zeros)
    blue_channel = ones(size(P2, 1), size(P2, 2));   % Blue channel (all ones for full blue)

    % Combine the channels into the final colored mask (purple mask)
    colored_mask = cat(3, red_channel, green_channel, blue_channel);

    % Apply the mask to the detected region with transparency
    h_mask = imshow(colored_mask);
    set(h_mask, 'AlphaData', 0.3 * color_mask_cleaned);  % Set transparency based on the mask (30% opacity)

    hold off;
else
    disp('No matching color area detected in the image.');
end
```

Figure 57. Perspective Distortion of Planar Surface – the code for part f)

Continuing with the code, **bwconncomp** is used to label connected components within the cleaned binary mask, identifying distinct regions or objects based on pixel connectivity. This function segments the mask into separate connected areas. After labelling the components,

regionprops are applied to measure specific properties of these regions, such as their bounding boxes and areas. These measurements allow further processing or identification of the regions based on their size and shape, helping to isolate and track the desired areas within the image.

The next **if-else** statement is used to check if any region was found. If no regions are found, a corresponding message will be displayed. In the inner body of the logical statement, if there are many components, only the largest one will be selected. Then, its bounding box is retrieved and used for visualization. But first, the image is visualized, and the **hold on** command allows the overwriting of the existing image figure. Then, the rectangle representing the bounding box of the detected area is displayed. The next part deals with defining the color of the mask. I wanted it purple, so I am adding the 3 RGB channels with the green one set to only 0s. The channels are combined using the **cat** command. I am storing the image object of the mask returned from **imshow** (initially covering the whole transformed picture field). Using the **set** operation, I am setting the transparency of this top mask layer, so for the area of where the mask values are 1, the purple mask will be applied with a transparency of 0.3. The regions apart from the detected object will have mask values 0 (as nothing was detected there), resulting in a transparency of 0 (the purple mask will be invisible there). In conclusion, the purple mask should appear only on the detected areas matching the specified color HSV range.

Results:

Figure 58 presents the results of the detection. As can be seen, the rectangular area specified by the assignment requirements is found (the purple mask surrounded by the neon green rectangle). Also, there are no other purple regions, so the detection is fine-tuned for the object's properties. The result is good because the object area is detected with high accuracy, despite that the object is blocked by the blue curve crossing it.

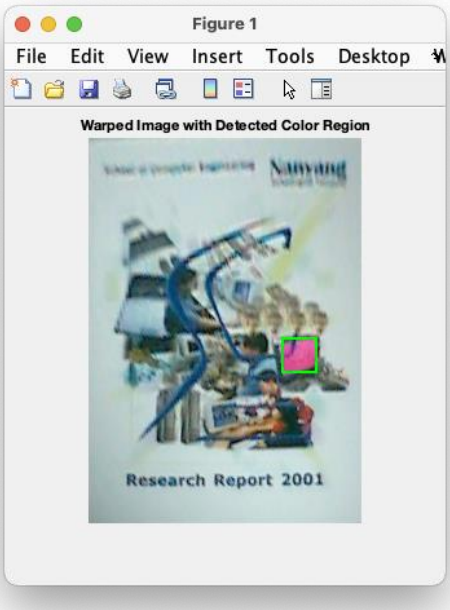


Figure 58. Perspective Distortion of Planar Surface – detected region with the specified color

3.7 Perceptron Algorithms

3.7.1 Concept

A perceptron is a type of artificial neuron used in machine learning, particularly for binary classification. The perceptron model takes input features, applies a linear transformation using weights, and produces an output that is thresholded to predict one of two possible classes. The goal is to iteratively adjust the weights to minimize the classification errors. This experiment focuses on two algorithms: Classical Perceptron and Gradient Descent-Based Perceptron. Typically, the elements of the mathematical formulation of the perceptions are:

- $w \in R^d$ be the weight vector
- $x_i \in R^d$ be the input vector (features) of sample i
- $y_i \in \{-1,1\}$ be the class label of sample i
- α be the learning rate

The classical perceptron algorithm is a basic form of supervised learning. It uses a threshold-based activation function and updates the weights whenever a misclassification occurs. The process continues iteratively until all samples are classified correctly or a maximum number of iterations is reached. The update rule adjusts the weights based on the input features and learning rate. The algorithm stops early when no misclassifications are detected in an iteration. This can be described in the following steps:

1. Initialize the weight vector w with small random values.
2. For each input vector x_i, compute the predicted output $\hat{y}_i = \text{sign}(w^T x_i)$, where $\hat{y}_i$ is the predicted label and the function $\text{sign}(\dots)$ returns 1if the value is positive and -1 if negative.
3. Update the weights when a misclassification occurs $w \leftarrow w + \alpha \cdot y_i \cdot x_i$ if $y_i \hat{y}_i \leq 0$
4. Repeat until convergence or a maximum number of iterations

The second algorithm is a variation that uses gradient descent to minimize the error function. Instead of only updating weights on misclassified samples, the gradient descent-based perceptron updates weights based on the error in the predicted output for each sample. The error is calculated as the difference between the actual class label and the predicted output. The weights are then adjusted to minimize the squared error. The algorithm is defined in the following steps:

1. Initialize the weight vector w with small random values.
2. For each input vector x_i, compute the predicted output $\hat{y}_i = w^T x_i$
3. Compute the error $\text{error} = y_i - \hat{y}_i$
4. Update the weights based on the gradient of the error function $w \leftarrow w + \alpha \cdot \text{error} \cdot x_i$
5. Repeat this process for each sample and iterate until the error is sufficiently small or a maximum number of iterations is reached.

3.7.2 MATLAB Code Analysis and Results

3.7.2.1 Set-up

MATLAB Code:

Before coding the algorithms, I am defining the input data and the desired classes for the data. The type of the inputs and their classes are similar to the ones used in the lectures. Each input data consists of 3 features (considering the third one a bias). I am also defining the minimum and maximum values for the x and y coordinates (used later for visualization purposes). Note that the x and y here refer to the first and second attributes of the input data x, not the class y.

```
% Define the input data and labels (two-class classification problem)
X = [-1 -1 1; -1 1 -1; 1 -1 -1; 1 1 1];  % Input data (4 samples, 3 features including bias)
y = [-1; -1; 1; 1];  % Desired class labels

% Find the dynamic limits for the plots
x_min = min(X(:, 1)) - 1;  % Add a margin to the minimum x value
x_max = max(X(:, 1)) + 1;  % Add a margin to the maximum x value
y_min = min(X(:, 2)) - 1;  % Add a margin to the minimum y value
y_max = max(X(:, 2)) + 1;  % Add a margin to the maximum y value
```

Figure 59. Perceptron Algorithms – the code for the set-up part

3.7.2.2 Part a)

MATLAB Code:

The code for this part (Figure 60) starts by initializing the weights to a random 3x1 vector using the **rand** function and setting the learning rate to 0.5. Then, I am setting the maximum number of iterations and defining an array to store the misclassifications for each iteration.

The training algorithm of the perceptron is defined in the next lines. It represents a **for** loop that will iterate for the maximum number of iterations if not stopped. Inside the loop, I first reset the counter for the misclassified points. I am also creating a second inner **for** loop structure that will perform the algorithm for each example (the x data points). Inside the inner loop, a check is performed to see if the output of the perceptron and the expected class for the data point have the same sign values (meaning they are the same). If they match, the algorithm continues with the next examples for the iteration. Otherwise, the weights are updated, and the number of misclassifications for the particular iteration is increased. After performing the algorithm for all training points for the iterations, the misclassifications counter is put in the array, monitoring the misclassifications for the iterations. Also, an early stopper is implemented by checking if the misclassifications are 0, and then the algorithm stops (not performing all maximum number of iterations).

After the completion of the algorithm, I am displaying the final weights in the console using the **disp** command. Then, I proceed with visualizing the number of misclassifications over the iterations in a line plot. If the algorithm has stopped earlier than the maximum number of iterations, I want to plot only up to 3 iterations after the one when the logic stopped.

The **min** function is used to safely set the end of the plot. It prevents exceeding the maximum number of iterations when adding the additional 3 iterations for visualization purposes. Then, the misclassifications are plotted over the number of executed iterations. The **hold on** and **hold off** commands are used to draw a vertical line over the chart to denote the moment of convergence. I am also adding a legend and setting titles for the axes.

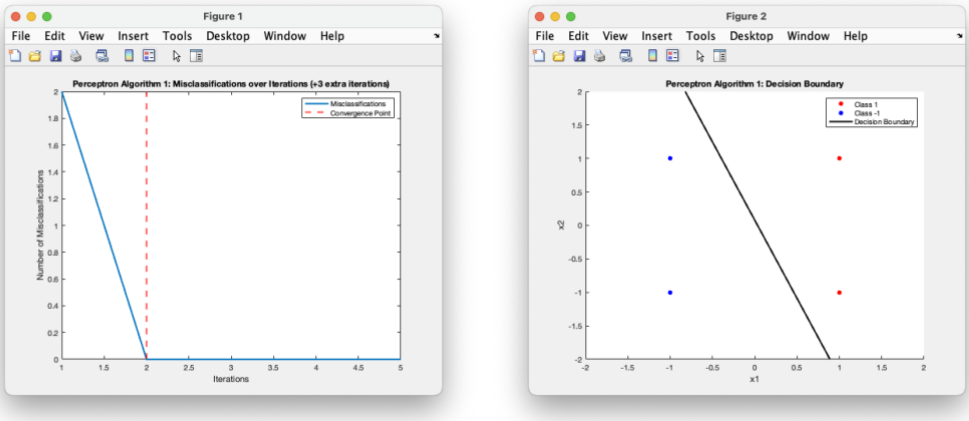
The second plot I am creating contains all training points 2D-visualized (using only the first 2 attributes, without the bias) on the figure and the decision boundary resulting from the trained weights. The decision boundary can be mathematically described as $y = -\frac{w_1 x + w_3}{w_2}$, where x and y are the coordinates on the plot. The decision boundary sets the rule for the classification of the training points. The points above are classified as class 1, and the ones below as class -1. Also, the points are colored based on their classes: red for class 1 and blue for class -1. The **plot** function is used for creating both plots. Lastly, a legend is added to the second plot, and the axes are titled.

```
% a) Perceptron Algorithm 1 (Classical Perceptron)

% Initialization
w1 = rand(3, 1);  % Initialize weights randomly to avoid zero initialization
alpha1 = 0.5;  % Learning rate
max_iter1 = 1000;  % Maximum number of iterations
misclassifications = zeros(max_iter1,  1);  % To store the number of misclassifications at each iteration

% Perceptron Algorithm 1 training
for iter1 = 1:max_iter1
    misclassified =  0;  % Count the number of misclassified samples
    for k = 1:size(X, 1)
        if y(k) * (w1' * X(k, :)') <= 0
            % Update rule for misclassified samples
            w1 = w1 + alpha1 * y(k) * X(k, :)';
            misclassified = misclassified +   1;
        end
    end
    misclassifications(iter1) = misclassified;    % Track the number of misclassified samples

    if misclassified ==  0
        break;  % Stop if all samples are classified correctly
    end
end

% Display final weights for Perceptron Algorithm 1
disp('Final weights for Perceptron Algorithm 1:');
disp(w1);

% Plotting the number of misclassifications over iterations (+3 iterations after convergence)
max_iter_plot1 = min(iter1 + 3, max_iter1);  % Ensure we don't exceed the array size
figure;
plot(1:max_iter_plot1, misclassifications( 1:max_iter_plot1), 'LineWidth', 2);
xlabel('Iterations');
ylabel('Number of Misclassifications' );
title('Perceptron Algorithm 1: Misclassifications over Iterations (+3 extra iterations)' );
hold on;
xline(iter1, '--r', 'LineWidth', 2);  % Vertical line at convergence iteration
legend('Misclassifications' , 'Convergence Point');
hold off;

% Plotting the decision boundary for Perceptron Algorithm 1
figure;
hold on;

% Plot the data points for both classes
scatter(X(y == 1, 1), X(y == 1, 2), 'r', 'filled' );  % Class 1
scatter(X(y == -1, 1), X(y == -1, 2), 'b', 'filled' );  % Class -1

% Calculate and plot the decision boundary (only if w(2) is non-zero)
x_vals = linspace(x_min, x_max, 100);  % Generate x values for the boundary
if w1(2) ~= 0
    y_vals = -(w1(1) * x_vals + w1(3)) / w1(2);  % Decision boundary equation
    plot(x_vals, y_vals, 'k-', 'LineWidth', 2);  % Decision boundary
    title('Perceptron Algorithm 1: Decision Boundary');
else
    disp('No valid decision boundary. Weights may need more training.');
end

% Formatting the plot
xlabel('x1');
ylabel('x2');
legend('Class 1', 'Class -1', 'Decision Boundary');
xlim([x_min x_max]);  % Set dynamic x-axis limits
ylim([y_min y_max]);  % Set dynamic y-axis limits
hold off;
```

Figure 60. Perceptron Algorithms – the code for part a)

Results:

Figure 61 shows the results from running the code for Algorithm 1 (Classical Perceptron). First, it should be noted that the results will vary every time because the initial weights are initialized randomly. Thus, the time taken for converging greatly depends on the initial weights, it can even be the case that the initial weights correctly classify the training points. In addition, the decision boundary also depends on the initialized weights, different initial weights will result in different decision boundaries. Now, let's continue with the analysis, the plot on the left shows that it took 2 iterations for the algorithm to achieve the best accuracy. This means that after altering the weights in the first iteration, the algorithm correctly predicted the classes in the second one. However, as seen in the figure on the right, the final decision boundary is not the most optimal one. The most optimal in this case will be a vertical line located exactly in the middle between the x-axis of the example points. The Classical Perceptron algorithm cannot guarantee the best result because it stops when the points are correctly classified, it does not consider any error metrics.




```
Final weights for Perceptron Algorithm 1:
    1.2785
    0.5469
   -0.0425
```

Figure 61. Perceptron Algorithms – the misclassification plot, the decision boundary for Algorithm 1, and the final weights

3.7.2.3 Part b)

MATLAB Code:

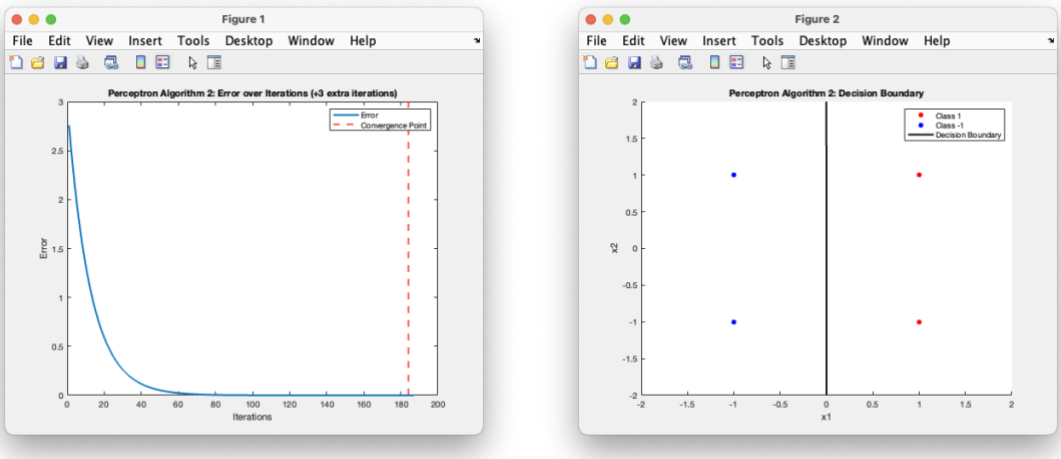
To avoid the problem of the first algorithm not giving the most optimal solution, the second algorithm (Gradient Descent-Based Perceptron) is implemented. The structure of the code (Figure 62) is very similar to the one for the previous algorithm, so the repetitive elements will not be described in detail again. Again, the code starts with initializing the variables needed for the algorithms to start training. The structure of the nested **for** loops is similar, with 2 main differences. The first one is the replacing of the correct classification **if** statement with computing the error for each training data for each iteration. I am using the error function from the lecture slides: $E(w) = \frac{1}{2}(r - w^T x)^2$. Thus, its gradient is $\frac{\partial E(w)}{\partial w} = (r - w^T x)(-x)$. These formulas are implemented in the code, as seen in Figure 62. The second biggest difference is the condition for stopping the algorithms. In this case, I am defining a threshold for the requirement of an optimal solution. The optimal solution is defined as one with a total error less than the specified threshold. Then, the code continues in the same way as for Algorithm 1, but instead of plotting the misclassifications, the total error is plotted for each iteration.

```
% Initialization
w2 = rand(3, 1);  % Initialize weights randomly
alpha2 = 0.01;  % Learning rate
max_iter2 = 1000;  % Maximum number of iterations
errors = zeros(max_iter2, 1);  % To store the error at each iteration

% Perceptron Algorithm 2 training using gradient descent
for iter2 = 1:max_iter2
    total_error = 0;
    for k = 1:size(X, 1)
        % Compute the output and error
        r = y(k);  % Desired output
        output = w2' * X(k, :)';  % Predicted output
        error = r - output;  % Error
        total_error = total_error + 0.5 * error^2;  % Accumulate squared error

        % Weight update using gradient descent
        w2 = w2 + alpha2 * error * X(k, :)';
    end
    errors(iter2) = total_error;

    % Stop if the error is sufficiently small
    if total_error < 1e-6
        break;
    end
end

% Display final weights for Perceptron Algorithm 2
disp('Final weights for Perceptron Algorithm 2:');
disp(w2);

% Plotting the error over iterations (+3 iterations after convergence)
max_iter_plot2 = min(iter2 + 3, max_iter2);  % Ensure we don't exceed the array size
figure;
plot(1:max_iter_plot2, errors(1:max_iter_plot2), 'LineWidth', 2);
xlabel('Iterations');
ylabel('Error');
title('Perceptron Algorithm 2: Error over Iterations (+3 extra iterations)');
hold on;
xline(iter2, '--r', 'LineWidth', 2);  % Vertical line at convergence iteration
legend('Error', 'Convergence Point');
hold off;

% Plotting the decision boundary for Perceptron Algorithm 2
figure;
hold on;

% Plot the data points for both classes
scatter(X(y == 1, 1), X(y == 1, 2), 'r', 'filled' );  % Class 1
scatter(X(y == -1, 1), X(y == -1, 2), 'b', 'filled' );  % Class -1

% Calculate and plot the decision boundary
x_vals = linspace(x_min, x_max, 100);  % Generate x values for the boundary
if w2(2) ~= 0
    y_vals = -(w2(1) * x_vals + w2(3)) / w2(2);  % Decision boundary equation
    plot(x_vals, y_vals, 'k-', 'LineWidth', 2);  % Decision boundary
    title('Perceptron Algorithm 2: Decision Boundary');
else
    disp('No valid decision boundary. Weights may need more training.');
end

% Formatting the plot
xlabel('x1');
ylabel('x2');
legend('Class 1', 'Class -1', 'Decision Boundary');
xlim([x_min x_max]);  % Set dynamic x-axis limits
ylim([y_min y_max]);  % Set dynamic y-axis limits
hold off;
```

Figure 62. Perceptron Algorithms – the code for part b)

Results:

As clearly seen in Figure 63, the result of the Gradient Descent-Based Perceptron algorithm is much better than the one previously derived from the Classical Perceptron algorithm. Now, the decision boundary appears perfectly in the middle between the 2 classes of training points. Also, the final weights from the algorithm are much closer than the most optimal ones for the training dataset: [1 0 0] (corresponding to the desired vertical line). As seen in the left plot, the error becomes close to 0 in the first 50 iterations. The range from 50 to the point of convergence shows that the total error is remarkably close to 0, but the algorithm still continues as it has to meet the threshold condition for a very small total error. Tuning the threshold can result in faster convergence with the tradeoff of larger error. For example, choosing a threshold around 20 will result in fast early stopping, but the error will be closer to 1 than to 0. However, choosing a threshold around 80 will result in 2 times faster stopping of the algorithm and still an error close to 0. It should be noted that choosing extremely low thresholds for the error may lead to slower algorithm execution with minimal effect on the already good-enough error. In conclusion, the second algorithm achieves much better results, and the learning (training) can be monitored better as there is a defined error function. Also, this better definition of the algorithm's performance helps find the number of required iterations for a sufficiently low error.




```
Final weights for Perceptron Algorithm 2:
    0.9999
    0.0005
    0.0004
```

Figure 63. Perceptron Algorithms – the total error plot, the decision boundary for Algorithm 2, and the final weights

3.7.2.4 Part c)

MATLAB Code:

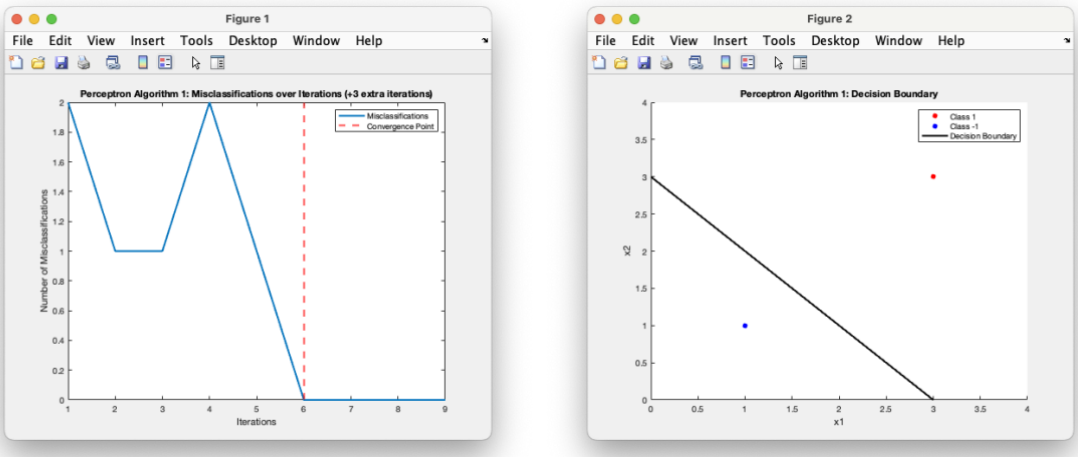
In this part the experiments will be conducted again but using the data points form the lecture slides. Figure 64 shows the initialization of the required points, and the parameters used for the algorithms. The new points are [3 3 1] and [1 1 1] with classes 1 and -1 respectfully. For both algorithms the weighs are initialized as [0 0 0]. The first algorithm uses a learning rate of 1 (as in the lecture slides), while I have chosen 0.1 as the learning rate for the second algorithm.

```
X = [3 3 1; 1 1 1];  % Lecture slides input data
y = [1; -1];  % Lecture slides input data classes (c1=1, c2=-1)

% Algorithm 1 parameters
w1 = [0; 0; 0];  % Weights as in the lecture slides
alpha1 = 1;  % Learning rate as in the lecture slides

% Algorithm 2 parameters
w2 = [0; 0; 0];  % Weights as in the lecture slides
alpha2 = 0.1;  % Learning rate as in the lecture slides
```

Figure 64. Perceptron Algorithms – parameters for part c)

Results:

Figure 65 presents the results of running Algorithm 1 for with the parameters and the data points from Figure 64. As it is seen in the figure, the algorithm updates the weights at first but then the misclassifications return back to 2. This is due to the high learning rate which results in too big changes in the weights. Although, the perceptron finds the correct direction after the 4th iteration and successfully achieves 0 misclassification in iteration 6. The algorithm successfully classifies the 2 points, however, as shown in the plot with the decision boundary, the boundary (result of the algorithm) is not the most optimal one because the line is much closer to one of the points and is located further from the other one. This is further proved by the final weights [1 1 -3] do not represent the ideal diagonal line passing through the middle of the distance between the 2 input points.

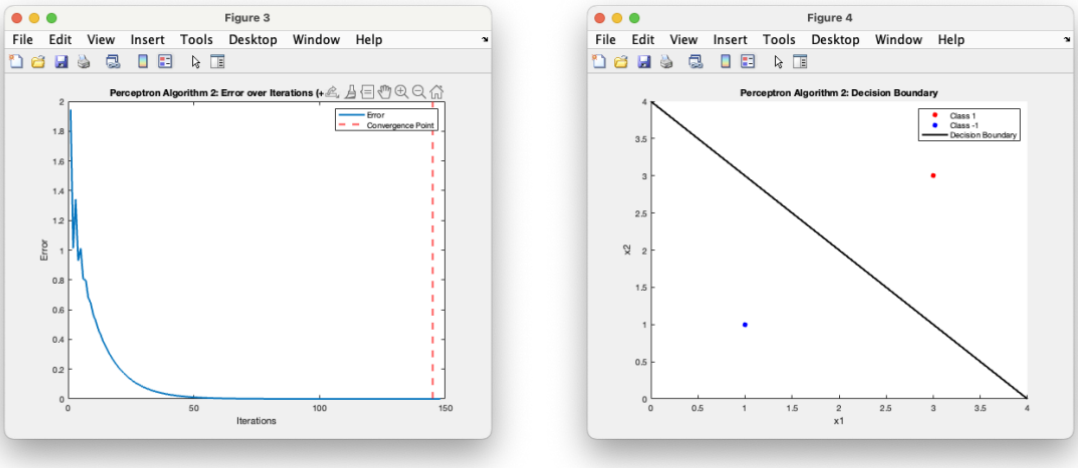



```
Final weights for Perceptron Algorithm 1:
     1
     1
    -3
```

Figure 65. Perceptron Algorithms – the misclassification plot, the decision boundary for Algorithm 1, and the final weights (for the lecture slides data points)

As for the second algorithm, as seen in Figure 66, the perceptron reaches convergence with the specified error requirement in around 140 iterations. However, the error is very close to 0, even in the early iterations around iteration 50. The Gradient Descent-Based Perceptron results in different weights than the Classic Perceptron. The new weights represent a line going through the middle between the distance of the 2 input points. The lecture slides provide only 2 data points, thus, there exist countless lines passing at equal distances from the points. The decision

boundary in the figure is very close to one of them but is not exactly ideal as the weights are not rounded to the perfect integer weights of [0.5 0.5 2]. This is due to the nature of the Gradient Descent-Based Perceptron and the floating-point results of the weights updates.




```
Final weights for Perceptron Algorithm 2:
    0.4997
    0.4997
   -1.9984
```

Figure 66. Perceptron Algorithms – the total error plot, the decision boundary for Algorithm 2, and the final weights (for the lecture slides data points)

In conclusion, both algorithms successfully classify the input points from the lecture slides. Although, there is no single ideal solution in the lecture example because that there infinitely many lines passing at equal distances from 2 points.