ENGI 9804 Lab 01

Chang Wan
202381126

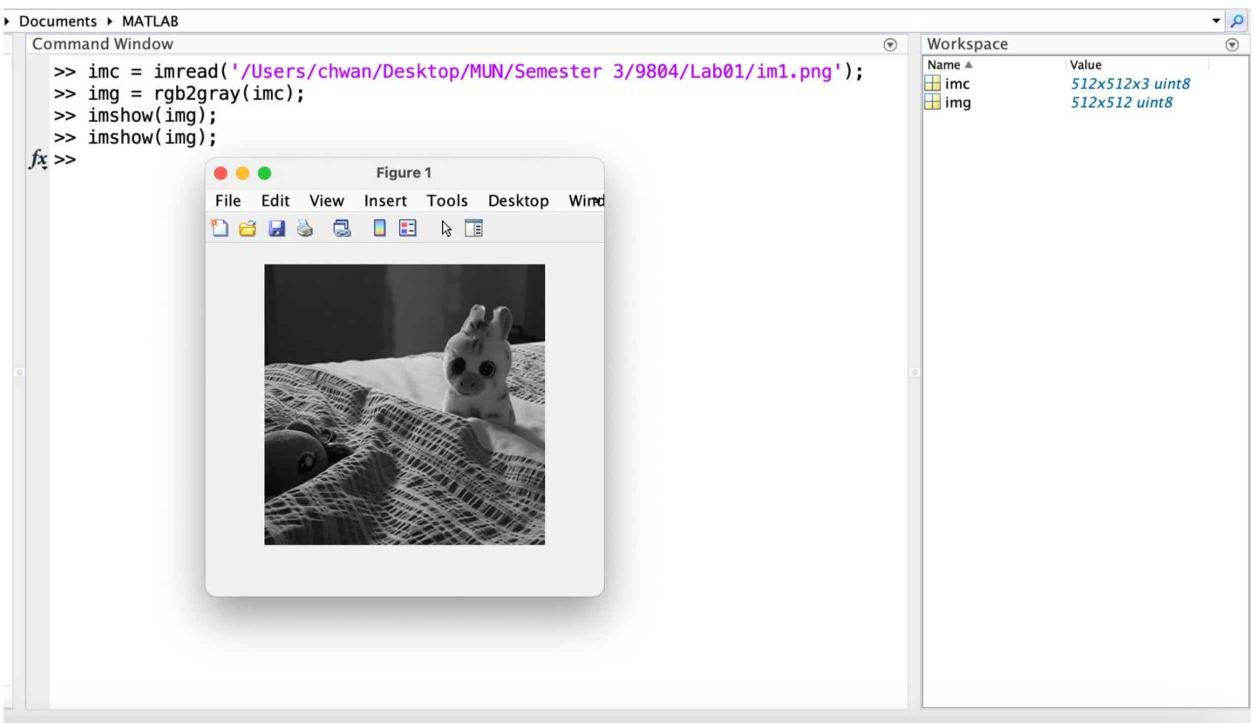
Ziyang Zhou
202193950

Geometric Transformation [46 points]

1. [1] Download the test image (im1.png) from Brightspace under *Lab 01*. Read the image, convert the image to a grayscale image, and display the image.

```
imc = imread('im1.png');% Read the image
img = rgb2gray(imc);     % Convert to grayscale. The supplied image is
    a grayscale, so you don't need to use this step.
imshow(img);             % View image
```

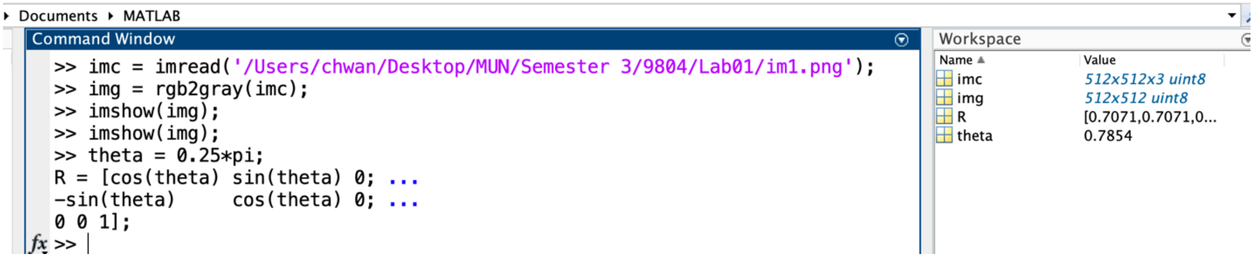
Answer:



2. (a) Define the transformation matrix with a rotation angle of 45 degrees.

```
theta = 0.25*pi;
R = [cos(theta) sin(theta) 0; ...
-sin(theta)     cos(theta) 0; ...
0 0 1];
```

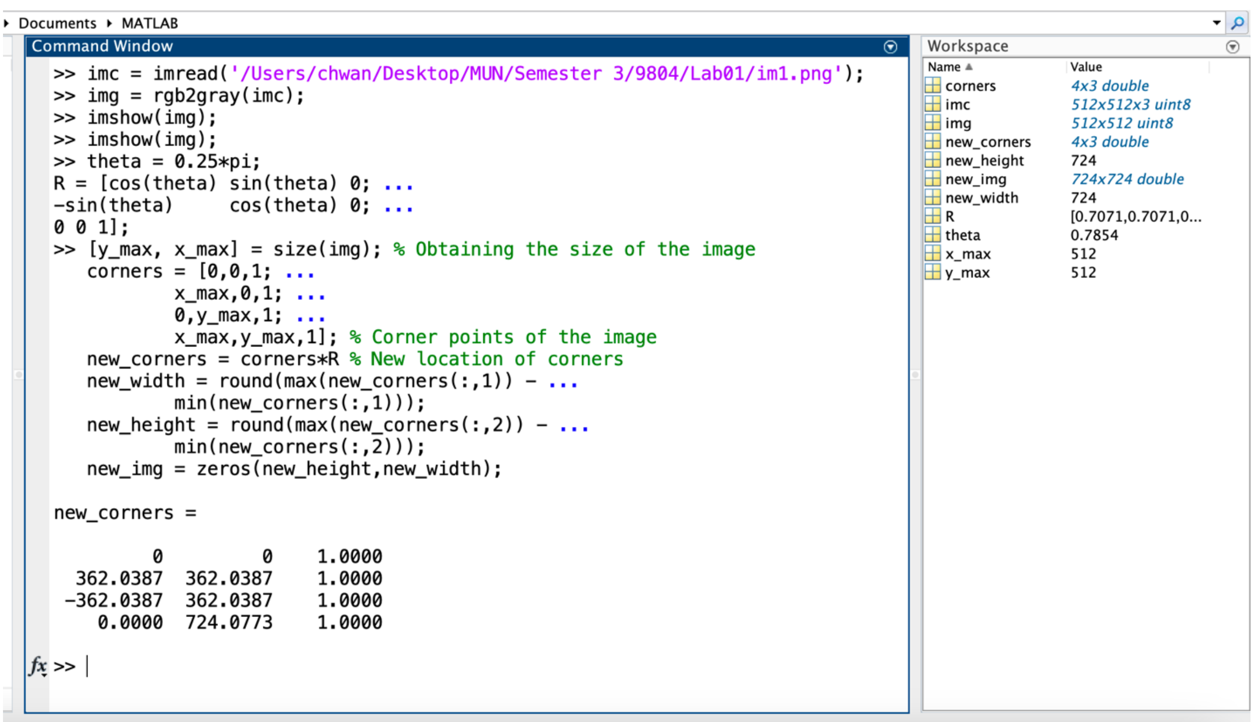
Answer:



(b) Calculate the size of the transformed image and generate an empty image with the calculated size.

```
[y_max, x_max] = size(img); % Obtaining the size of the image
corners = [0,0,1; ...
          x_max,0,1; ...
          0,y_max,1; ...
          x_max,y_max,1]; % Corner points of the image
new_corners = corners*R % New location of corners
new_width = round(max(new_corners(:,1)) - ...
          min(new_corners(:,1)));
new_height = round(max(new_corners(:,2)) - ...
          min(new_corners(:,2)));
new_img = zeros(new_height,new_width);
```

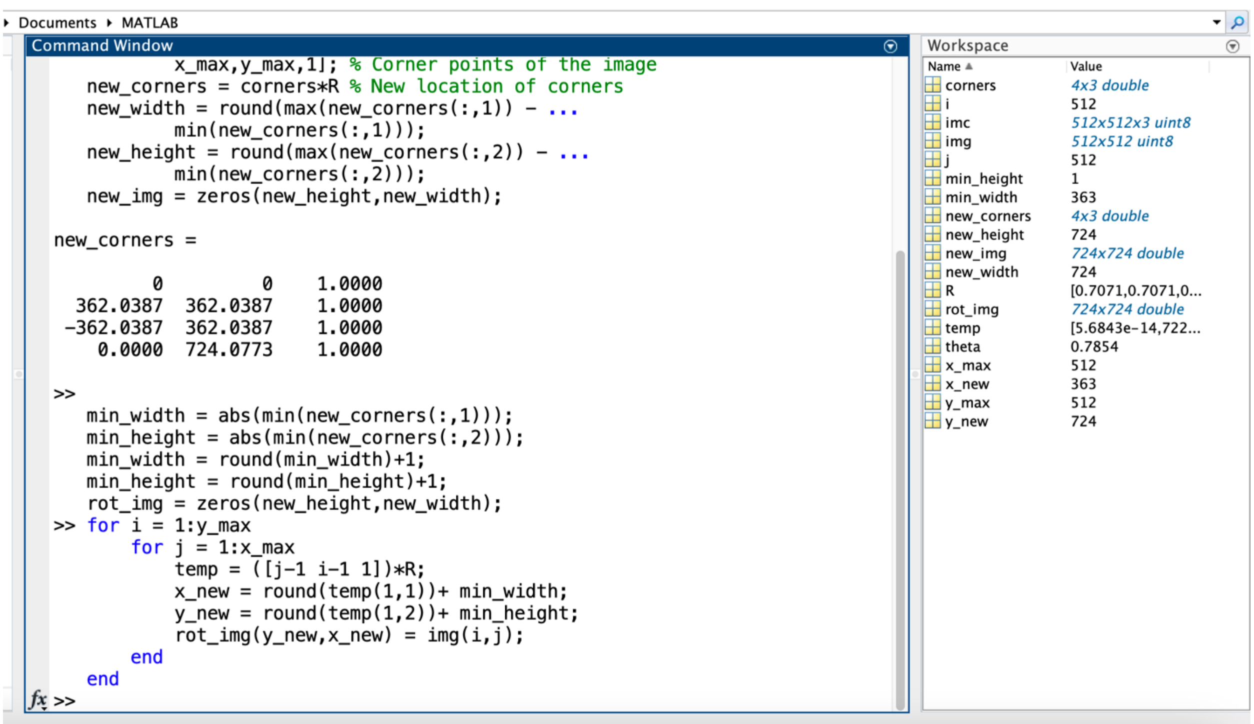
Answer:



(c) Transform each pixel of the original image with the transformation matrix and assign the intensity to the new location.

```
min_width = abs(min(new_corners(:,1)));
min_height = abs(min(new_corners(:,2)));

min_width = round(min_width)+1;
min_height = round(min_height)+1;

rot_img = zeros(new_height,new_width);

for i = 1:y_max
        for j = 1:x_max
                temp = ([j-1 i-1 1])*R;
                x_new = round(temp(1,1))+ min_width;
                y_new = round(temp(1,2))+ min_height;
                rot_img(y_new,x_new) = img(i,j);
        end
end
```

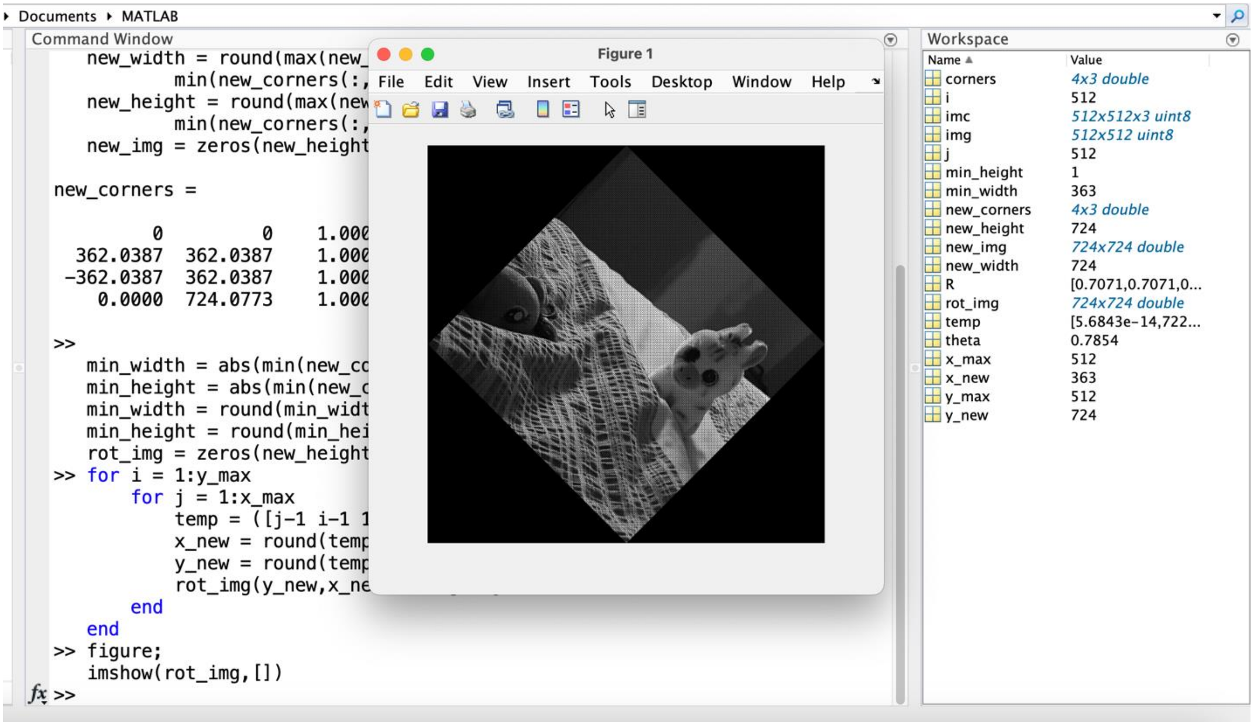
Answer:



(d) [5] View the transformed image

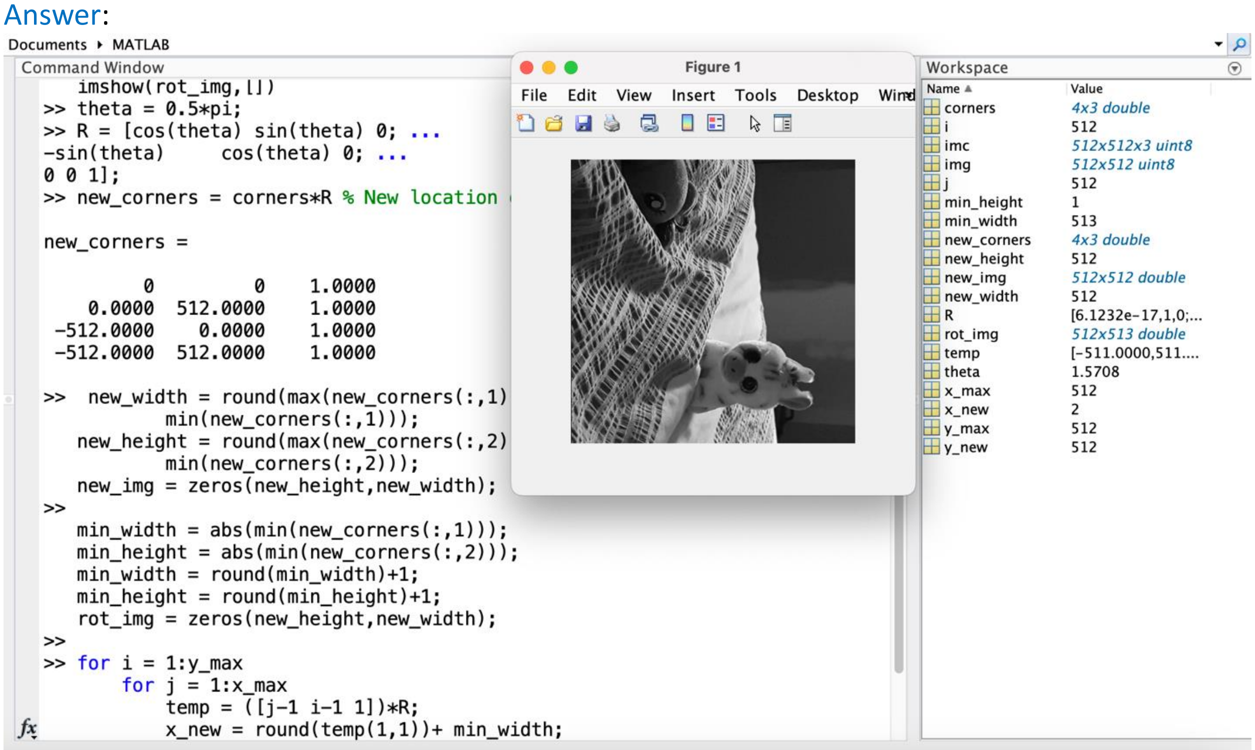
```
figure;
imshow(rot_img, [])
```

Answer:



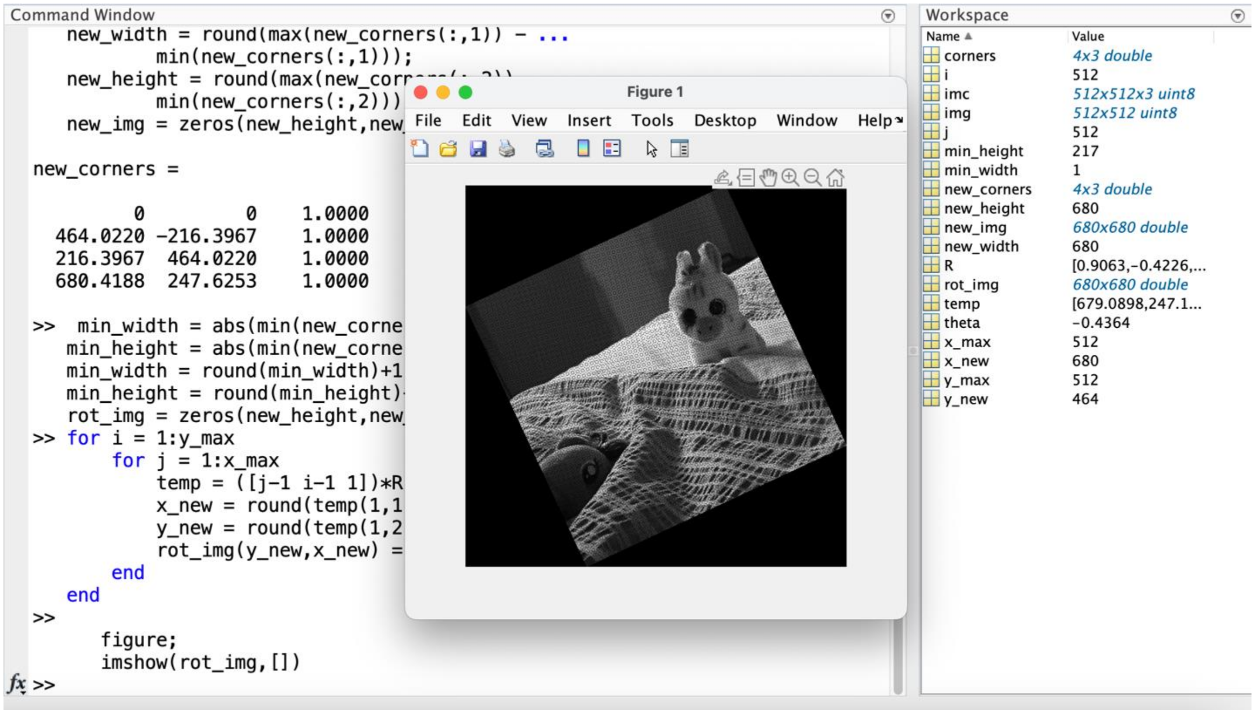
3. [5] Complete the steps above (Q2.a - 2.d) for angle $\theta = 90^0$.

Answer:



4. [5] Complete the steps above (Q2.a - 2.d) for angle $\theta = -25^0$.

Answer:



5. [5] It can be seen that in some cases, when the image is transformed the output image has 'empty' black pixels. Explain why this happens in only some images, but not others.

Answer:

Because there is an empty image created as the size of original image. After the rotate of original image, when it cannot fill all the empty image, the empty image background will might show.

6. In the previous questions, we defined the size of the output image and mapped each input pixel to the corresponding output pixel. We can improve the output image, ensuring no 'black' or 'empty' pixels by first calculating the size of the transformed image, generating an empty image of the correct output size, and then calculate the *inverse* of the forward transformation matrix. For each pixel of the transformed output image, we use the inverse transformation to search backwards in the input image for the correct pixel. In this way, every pixel in the output image will find an input pixel. If the corresponding pixel in the input image is outside image boundary, we can simply set the value of the output pixel to zero.

   (a) Calculate the inverse transformation matrix.

Answer:

For rotation:

$$R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R^{-1} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

For Scaling:

$$S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad S^{-1} = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

For Translation:

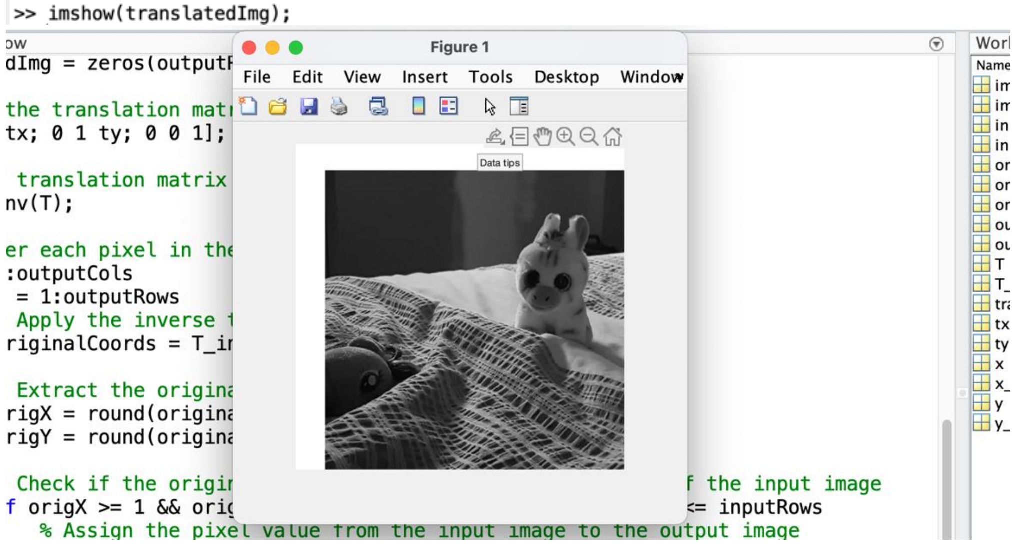
$$T = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \qquad T^{-1} = \begin{pmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{pmatrix}$$

(b) Modify the code in Q2 to calculate the following inverse transformation, and use it to iterate through each output image pixel to search the input image for the following transformations. Visualize your results.

i [5] Translation ($t_x = 50, t_y = 45$)

```
>> % Read the image
imc = imread('/Users/chwan/Desktop/MUN/Semester 3/9804/Lab01/im1.png');

% Convert to grayscale
img = rgb2gray(imc);

% Translation parameters
tx = 50;
ty = 45;

% Get the size of the input image
[inputRows, inputCols] = size(img);

% Calculate the size of the output image
outputRows = inputRows + abs(ty);
outputCols = inputCols + abs(tx);

% Calculate the necessary shift for translation
if tx > 0
    x_shift = tx;
else
    x_shift = 0;
end

if ty > 0
    y_shift = ty;
else
    y_shift = 0;
end

% Initialize the output image with zeros
translatedImg = zeros(outputRows, outputCols, 'uint8');

% Define the translation matrix
T = [1 0 tx; 0 1 ty; 0 0 1];

% Inverse translation matrix
T_inv = inv(T);
```

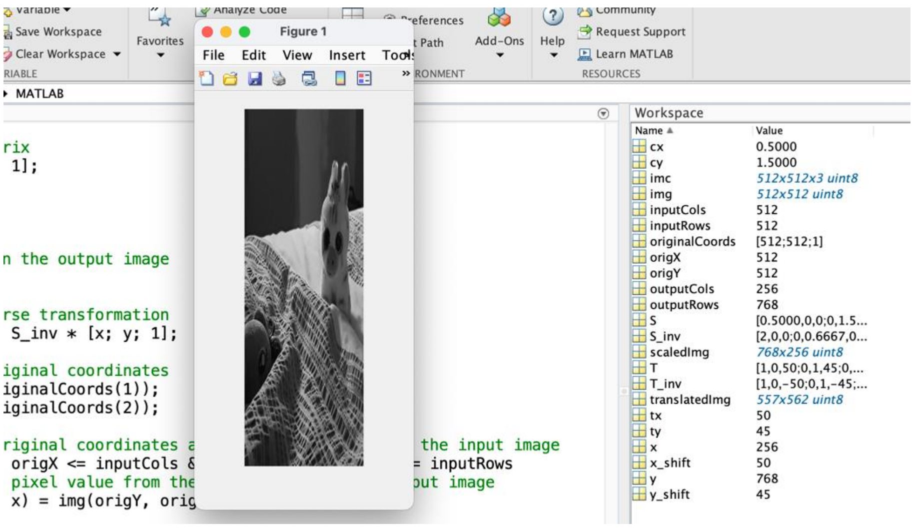
```
% Loop over each pixel in the output image
for x = 1:outputCols
    for y = 1:outputRows
        % Apply the inverse transformation
        originalCoords = T_inv * [x; y; 1];

        % Extract the original coordinates
        origX = round(originalCoords(1));
        origY = round(originalCoords(2));

        % Check if the original coordinates are within the bounds of the input image
        if origX >= 1 && origX <= inputCols && origY >= 1 && origY <= inputRows
            % Assign the pixel value from the input image to the output image
            translatedImg(y, x) = img(origY, origX);
        else
            % Set the value of the output pixel to zero
            translatedImg(y, x) = 255;
        end
    end
end
>> imshow(translatedImg);
```



ii [5] Scale $(c_x = 0.5, c_y = 1.5)$

```
>> % Read the image
imc = imread('/Users/chwan/Desktop/MUN/Semester 3/9804/Lab01/im1.png');

% Convert to grayscale
img = rgb2gray(imc);

% Scaling parameters
cx = 0.5;
cy = 1.5;

% Get the size of the input image
[inputRows, inputCols] = size(img);

% Calculate the size of the output image based on scaling factors
outputRows = round(inputRows * cy);
outputCols = round(inputCols * cx);

% Initialize the output image with zeros
scaledImg = zeros(outputRows, outputCols, 'uint8');

% Define the scaling matrix
S = [cx 0 0; 0 cy 0; 0 0 1];

% Inverse scaling matrix
S_inv = inv(S);
```

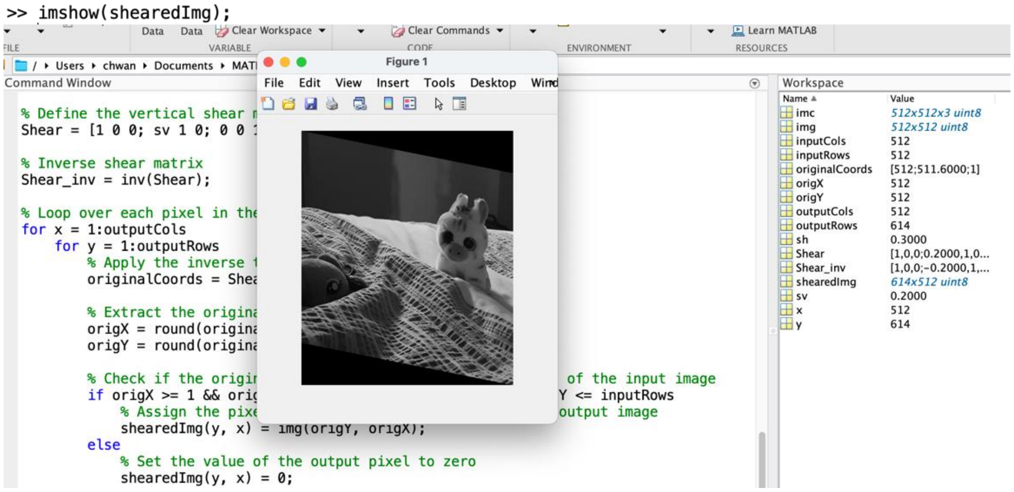
```
% Loop over each pixel in the output image
for x = 1:outputCols
    for y = 1:outputRows
        % Apply the inverse transformation
        originalCoords = S_inv * [x; y; 1];

        % Extract the original coordinates
        origX = round(originalCoords(1));
        origY = round(originalCoords(2));

        % Check if the original coordinates are within the bounds of the input image
        if origX >= 1 && origX <= inputCols && origY >= 1 && origY <= inputRows
            % Assign the pixel value from the input image to the output image
            scaledImg(y, x) = img(origY, origX);
        else
            % Set the value of the output pixel to zero
            scaledImg(y, x) = 0;
        end
    end
end
>> imshow(scaledImg);
```



iii [5] Shear (vertical) ($s_v = 0.2$)

```
>> % Read the image
imc = imread('/Users/chwan/Desktop/MUN/Semester 3/9804/Lab01/im1.png');

% Convert to grayscale
img = rgb2gray(imc);

% Vertical shear parameters
sv = 0.2; % vertical shear factor

% Get the size of the input image
[inputRows, inputCols] = size(img);

% Calculate the size of the output image based on the shear factor
outputRows = inputRows + abs(round(sv * inputCols));
outputCols = inputCols;

% Initialize the output image with zeros
shearedImg = zeros(outputRows, outputCols, 'uint8');

% Define the vertical shear matrix
Shear = [1 0 0; sv 1 0; 0 0 1];

% Inverse shear matrix
Shear_inv = inv(Shear);
```

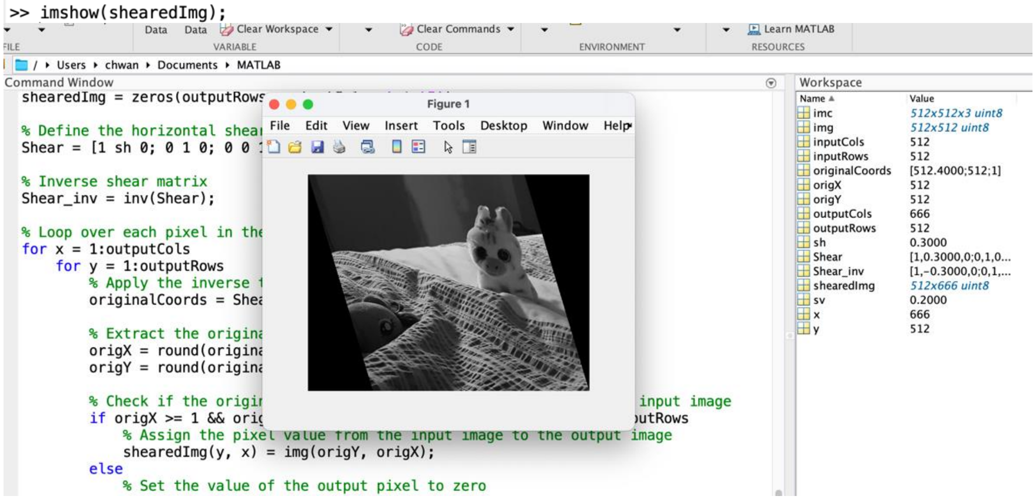
```
% Loop over each pixel in the output image
for x = 1:outputCols
    for y = 1:outputRows
        % Apply the inverse transformation
        originalCoords = Shear_inv * [x; y; 1];

        % Extract the original coordinates
        origX = round(originalCoords(1));
        origY = round(originalCoords(2));

        % Check if the original coordinates are within the bounds of the input image
        if origX >= 1 && origX <= inputCols && origY >= 1 && origY <= inputRows
            % Assign the pixel value from the input image to the output image
            shearedImg(y, x) = img(origY, origX);
        else
            % Set the value of the output pixel to zero
            shearedImg(y, x) = 0;
        end
    end
end
>> imshow(shearedImg);
```



iv [5] Shear (horizontal) ($s_v = 0.3$)

```
>> % Read the image
imc = imread('/Users/chwan/Desktop/MUN/Semester 3/9804/Lab01/im1.png');

% Convert to grayscale
img = rgb2gray(imc);

% Horizontal shear parameters
sh = 0.3; % horizontal shear factor

% Get the size of the input image
[inputRows, inputCols] = size(img);

% Calculate the size of the output image based on the shear factor
outputRows = inputRows;
outputCols = inputCols + abs(round(sh * inputRows));

% Initialize the output image with zeros
shearedImg = zeros(outputRows, outputCols, 'uint8');

% Define the horizontal shear matrix
Shear = [1 sh 0; 0 1 0; 0 0 1];

% Inverse shear matrix
Shear_inv = inv(Shear);
```

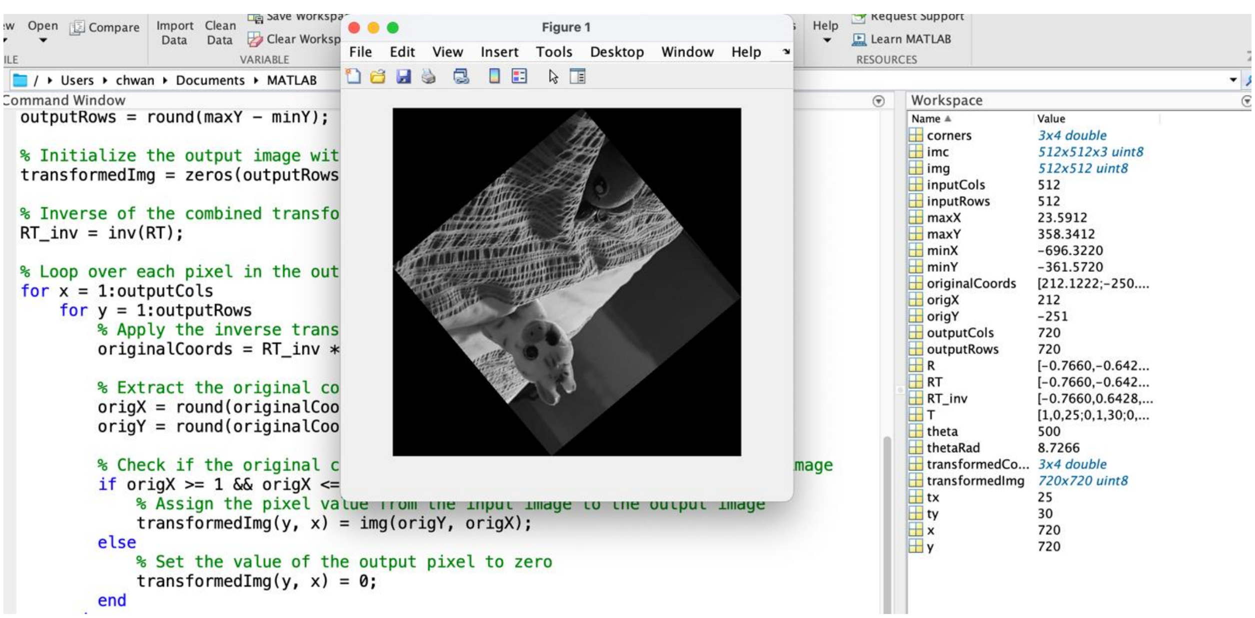
```
% Loop over each pixel in the output image
for x = 1:outputCols
    for y = 1:outputRows
        % Apply the inverse transformation
        originalCoords = Shear_inv * [x; y; 1];

        % Extract the original coordinates
        origX = round(originalCoords(1));
        origY = round(originalCoords(2));

        % Check if the original coordinates are within the bounds of the input image
        if origX >= 1 && origX <= inputCols && origY >= 1 && origY <= inputRows
            % Assign the pixel value from the input image to the output image
            shearedImg(y, x) = img(origY, origX);
        else
            % Set the value of the output pixel to zero
            shearedImg(y, x) = 0;
        end
    end
end
>> imshow(shearedImg);
```



v [5] Rotation ($\theta = 50^0$) followed by translation (($t_x = 25, t_y = 30$))

```
>> % Read the image
imc = imread('/Users/chwan/Desktop/MUN/Semester 3/9804/Lab01/im1.png');

% Convert to grayscale
img = rgb2gray(imc);

% Rotation and translation parameters
theta = 500; % Rotation angle in degrees
tx = 25; % Translation in x direction
ty = 30; % Translation in y direction

% Get the size of the input image
[inputRows, inputCols] = size(img);

% Convert angle to radians
thetaRad = deg2rad(theta);

% Calculate the rotation matrix
R = [cos(thetaRad) -sin(thetaRad) 0; sin(thetaRad) cos(thetaRad) 0; 0 0 1];

% Calculate the translation matrix
T = [1 0 tx; 0 1 ty; 0 0 1];

% Combined transformation matrix
RT = T * R;
```

```
% Find the corners of the input image to calculate the output size
corners = [1 1 1; inputCols 1 1; 1 inputRows 1; inputCols inputRows 1]';
transformedCorners = RT * corners;

% Calculate the bounds of the transformed image
minX = min(transformedCorners(1,:));
maxX = max(transformedCorners(1,:));
minY = min(transformedCorners(2,:));
maxY = max(transformedCorners(2,:));

% Calculate the size of the output image
outputCols = round(maxX - minX);
outputRows = round(maxY - minY);

% Initialize the output image with zeros
transformedImg = zeros(outputRows, outputCols, 'uint8');

% Inverse of the combined transformation matrix
RT_inv = inv(RT);

% Loop over each pixel in the output image
for x = 1:outputCols
    for y = 1:outputRows
        % Apply the inverse transformation
        originalCoords = RT_inv * [x + minX; y + minY; 1];

        % Extract the original coordinates
        origX = round(originalCoords(1));
        origY = round(originalCoords(2));

        % Check if the original coordinates are within the bounds of the input image
        if origX >= 1 && origX <= inputCols && origY >= 1 && origY <= inputRows
            % Assign the pixel value from the input image to the output image
            transformedImg(y, x) = img(origY, origX);
        else
            % Set the value of the output pixel to zero
            transformedImg(y, x) = 0;
        end
    end
end
>> imshow(transformedImg);
```



Performance evaluation:

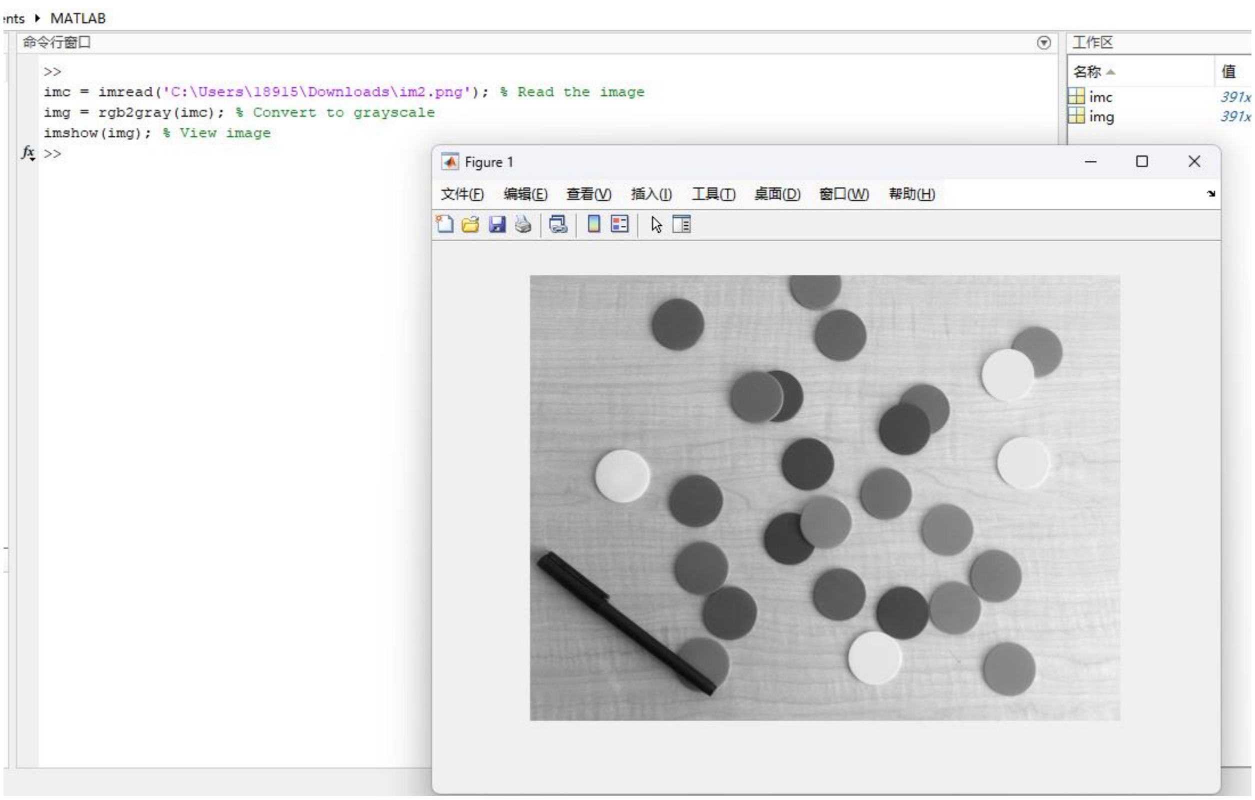
The built-in functions in MATLAB are highly optimized for performance, while custom implementations, especially those involving loops and manual calculations, tend to be slower.

Histogram Equalization [24 points]

1. [1] Download the test image (im2.png) from Brightspace under *Lab 01*. Read the image, convert the image to a grayscale image, and visualize it.

```
imc = imread('im2.png');% Read the image
img = rgb2gray(imc);     % Convert to grayscale
imshow(img);             % View image
```

Answer:



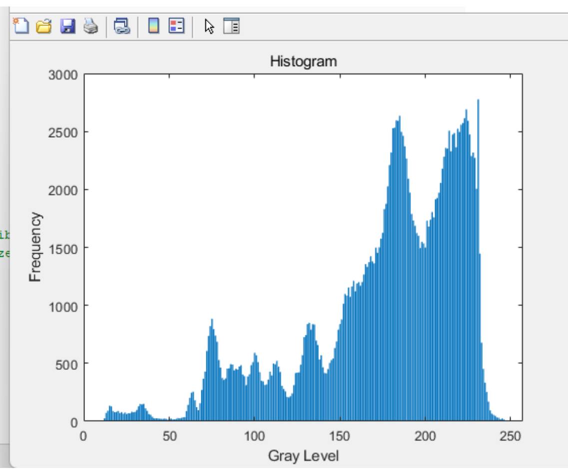
2. [5] Grayscale images have 256 gray levels. Therefore create an empty matrix having dimensions of 1×256, generate the histogram and display it.

```
H=size(img,1); % Read the height of the image
W=size(img,2); % Read the width of the image

Hist_arr=zeros(1,256); % Array for holding the (original) histogram
Hist_eq_arr=zeros(1,256); % Array for holding the (equalized)
    histogram

CDF_array=zeros(1,length(Hist_arr)); % array to hold cumulative
    distribution function (CDF)
hist_eq_img=uint8(zeros(H,W)); % A 2D array for keeping histogram
    equalized image (intensity in 8 bit integers)
for i=1:H,
        for j=1:W,
                Hist_arr(1,img(i,j)+1)=Hist_arr (1,img(i,j)+1)+1 ;
        end
end
```

Answer:

```
>>
>> H=size(img,1); % Read the height of the image
W=size(img,2); % Read the width of the image
>> Hist_arr=zeros(1,256); % Array for holding the (original) histogram
Hist_eq_arr=zeros(1,256); % Array for holding the (equalized)
histogram
>> CDF_array=zeros(1,length(Hist_arr)); % array to hold cumulative distribution function (CDF)
hist_eq_img=uint8(zeros(H,W)); % A 2D array for keeping histogram equalized image (intensity in 8 bit integers)
for i=1:H,
    for j=1:W,
        Hist_arr(1,img(i,j)+1)=Hist_arr (1,img(i,j)+1)+1 ;
    end
end
>> bar(Hist_arr); %display the Histogram as a bar chart
title('Histogram'); %set the title as Histogram
xlabel('Gray Level'); %set the x-axis as gray level
ylabel('Frequency'); %set the y-axis as frequency
>>
```

```
>>
>>
>>
>>
>>
>>
>>
>> H=size(img,1); % Read the height of the image
W=size(img,2); % Read the width of the image
>> Hist_arr=zeros(1,256); % Array for holding the (original) histogram
Hist_eq_arr=zeros(1,256); % Array for holding the (equalized)
histogram
>> CDF_array=zeros(1,length(Hist_arr)); % array to hold cumulative distrib
hist_eq_img=uint8(zeros(H,W)); % A 2D array for keeping histogram equalize
for i=1:H,
    for j=1:W,
        Hist_arr(1,img(i,j)+1)=Hist_arr (1,img(i,j)+1)+1 ;
    end
end
>> bar(Hist_arr); %display the Histogram as a bar chart
title('Histogram'); %set the title as Histogram
xlabel('Gray Level'); %set the x-axis as gray level
ylabel('Frequency'); %set the y-axis as frequency
>>
```



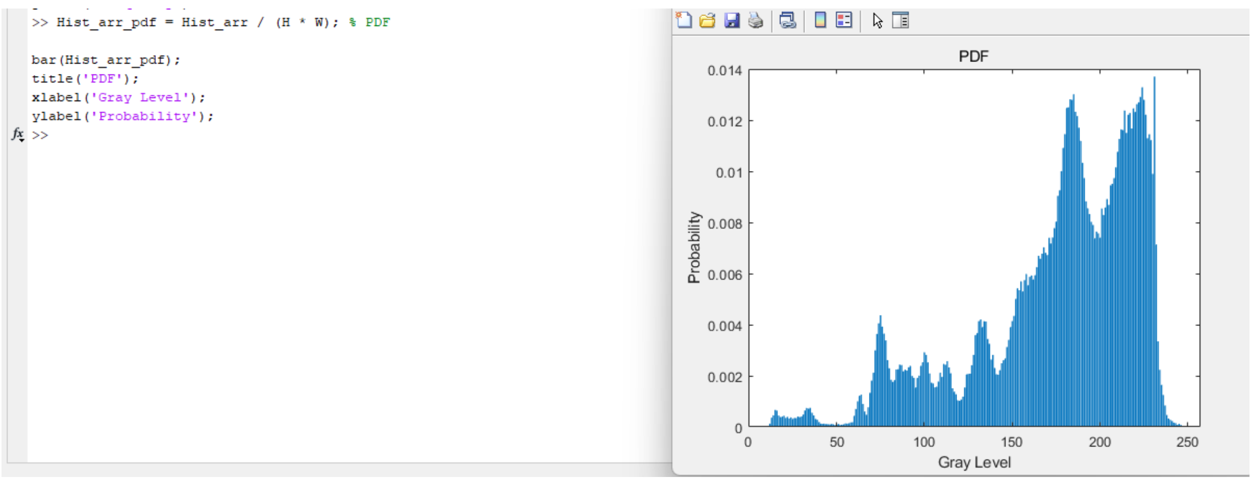
Explanation:

*The formal code part has been divided into 3 parts. The first part is reading the height and width of the image, save them as H and W. Then it initialized a 1*256 array with all elements initialized as 0. This array is used to store the number of pixels for each gray level.*
*Then using a double loop to iterate each pixel in the image. img(i, j) + 1 calculate the gray level of the current iterated pixel. Once the gray level of this pixel is calculated, the count of pixels for this gray level in the array we intialized before,which is Hist_arr , will be added 1 count. After iterating all the pixels in the image, the Hist_arr will store a 1*256 array, each element represents the number of pixels for each gray level.*
Then we use a bar chart to display the Histogram, name it as Histogram and set the x-axis as gray level and y-axis as Frequency.

3. [5] Calculate $p_r(r_k)$ for the image and display it. (Total number of pixels $= y_{max} \times x_{max}$).

```
Hist_arr_pdf=Hist_arr/(H*W); % PDF
```

Answer:



Explanation:

Based on the PDF's formula, we know that:
pr(rk)=nk/(M×N)

*Since we already have Hist_arr that stores all the number of pixels for each gray level, we need to divide them by the number of all the pixels, which is equal to H*W.*

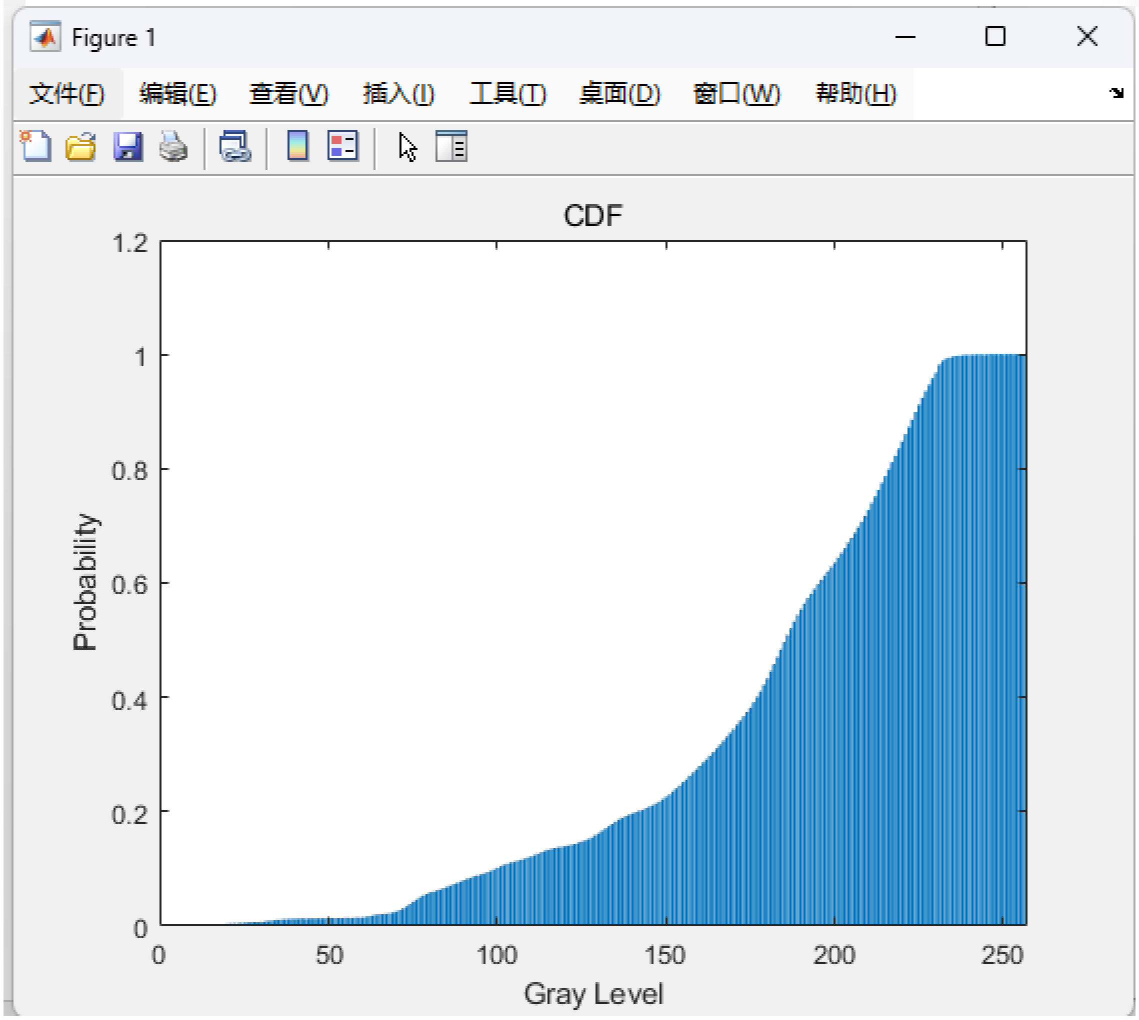
Then we display the PDF in a bar chart.

4. [5] Calculate the CDF from the PDF, and display it.

```
dummy1=0; % A dummy variable to hold the summation results
for k=1:length(Hist_arr); % Generating the CDF from PDF
        dummy1=dummy1+Hist_arr_pdf(k);
        CDF_array(k)= dummy1;
end
```

Answer:

```
>> dummy1 = 0; % A dummy variable to hold the summation results
for k = 1:length(Hist_arr)
    dummy1 = dummy1 + Hist_arr_pdf(k);
    CDF_array(k) = dummy1;
end
>> bar(CDF_array);
title('CDF');
xlabel('Gray Level');
ylabel('Probability');
>>
```



Explanation:

Since we know that CDF is the for gray level rk is equal to the sum of all the PDF for the gray level that is below rk

$$\mathrm{CDF}(r_k) = \sum_{i=0}^{k} p_r(r_i)$$

**This screenshot is from Chatgpt.*

To get the CDF for Hist_arr, we use a dummy variable (initialized as 0) as the count for the sum of the PDF, then use a for loop to iterate the Hist_arr, the first gray level's CDF is equal to the value of its PDF, then the next gray level's CDF will be the sum of the previous sum of the PDF plus the current gray level's PDF. We go through all the elements in the Hist_arr and get all the CDF for each gray level. Then we display it in a bar chart.

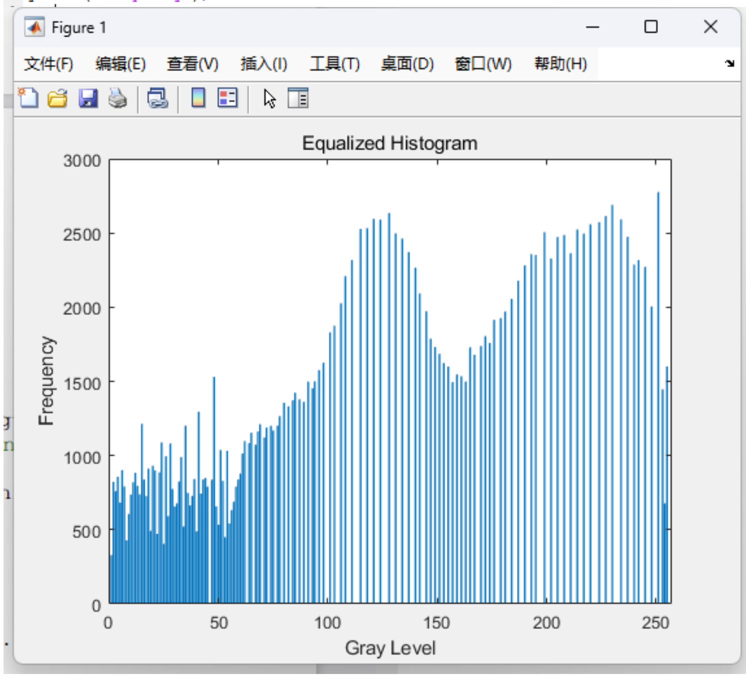
5. [3] Calculate the equalized histogram using equation 1 and map the original gray levels to the new gray levels. Display the original and new values.


```
for l=1:H, % Histogram equalization
        for m=1:W,
         hist_eq_img(l,m)= round(CDF_array(img(l,m)+1)*(length(
            Hist_arr_pdf)-1)); % scale to 255 and round to nearest
            integer
         Hist_eq_arr(l, hist_eq_img(l,m)+1)=Hist_eq_arr (l,hist_eq_img
            (l,m)+1)+1 ;  % Its histogram
        end
end
```

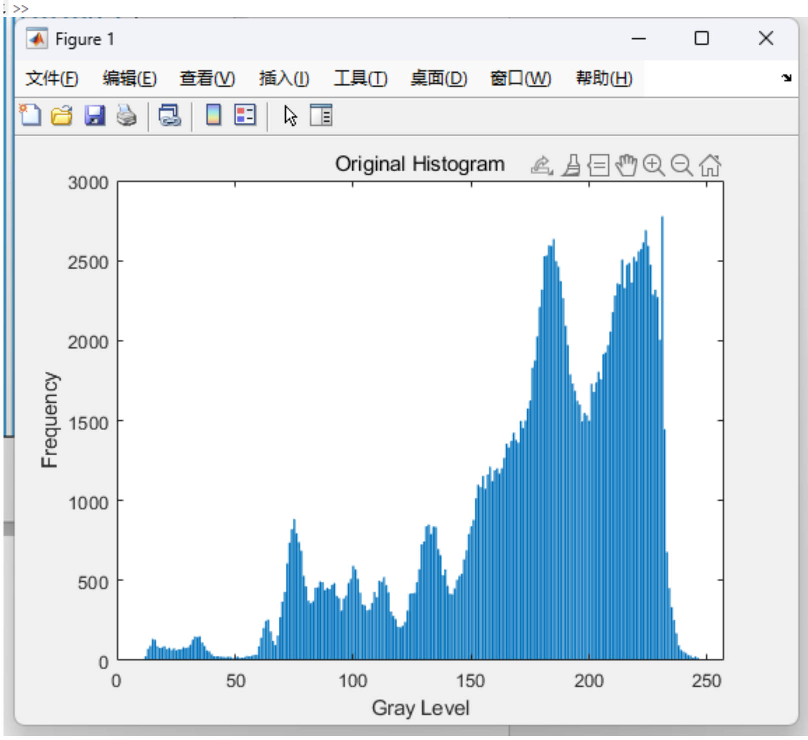
Answer:

```
>>
for l=1:H, % Histogram equalization
    for m=1:W,
        hist_eq_img(l,m)= round(CDF_array(img(l,m)+1)*(length(Hist_arr_pdf)-1)); % scale to 255 and round to nearest integer
        Hist_eq_arr(l, hist_eq_img(l,m)+1)=Hist_eq_arr (l,hist_eq_img(l,m)+1)+1 ; % Its histogram
    end
end
```

```
>> bar(Hist_eq_arr);
title('Equalized Histogram');
xlabel('Gray Level');
ylabel('Frequency');
```




```
>> bar(Hist_arr);
title('Original Histogram');
xlabel('Gray Level');
ylabel('Frequency');
>>
```



Explanation:

Based on the Histogram equalization equation discussed in our lecture, we have the formula:

Histogram equalization equation for images

$$s_k = T(r_k) = (L-1)\underbrace{\sum_{j=0}^{k} p_r(r_j)}_{\text{CDF}}$$

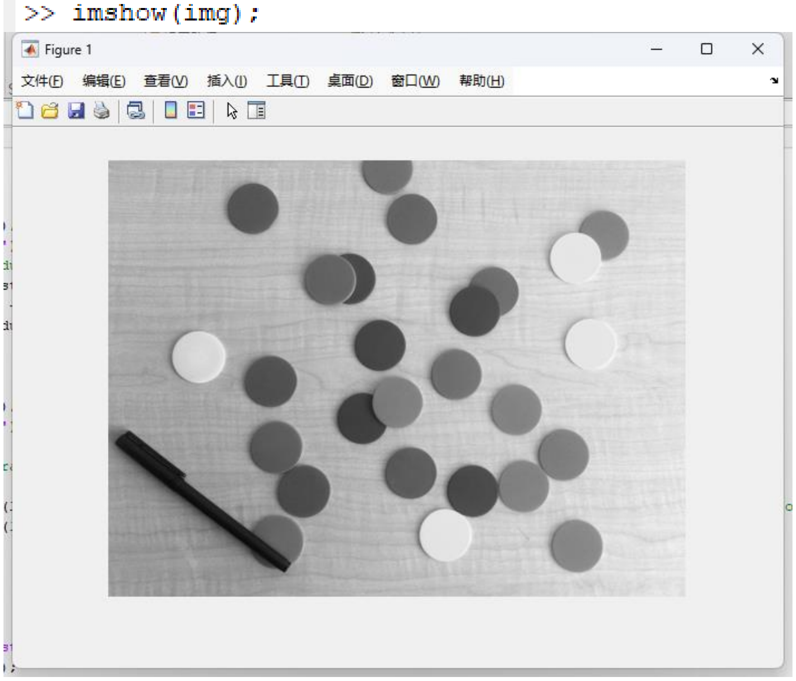
**This screenshot is from the lecture note.*

We use a double loop to iterate each pixel in the image. For each pixel:
L in our problem is equal to the length of the Hist_arr_pdf, which represents the total number of the gray level.
Then we use (L-1) to multiply this pixel's CDF, then round it to the nearest integer to get T(rk).
Then we save it into the new histogram equalized image hist_eq_img.
Then in the new equalized histogram, we will update the count of each gray level.

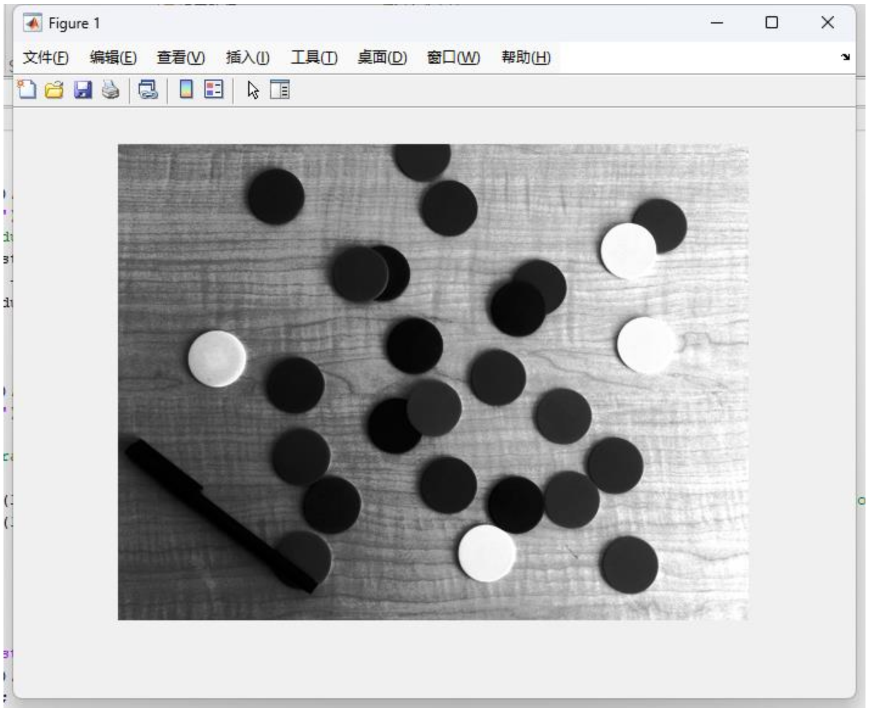
6. [5] Display both images, before (original) and after histogram equalization.

Answer:

```
>> imshow(img);
```




```
>> imshow(hist_eq_img)
```



Performance Evaluation

The result of our histogram equalization process cannot be called perfect. Though after the equalization, the contrast of the gray level in this image is improved, we still have a lot of circle objects in the original image. Even after the equalization, the gray level range of this circle objects will still show more frequently than the gray level ranges of the background.