ECSE – 436 Lab 3
Group 1

Muhammad Taha- 260505597
Chrouk Kasem 260512917

Question 1

a) Sub-sampling the image changes the image resolution, since for each sub-sampling ratio, we skip pixels in both dimensions: x and y direction, based on the ratio. Figures 1 to 4 show the results obtained for each sub-sampling ratio. In case of subsampling in 2:1, the x-dimension is sub-sampled by 2 :1 ratio, while y-dimension hasn't been sub-sampled, giving the image the stretched look as shown in Figure 1. For case of subsampling 4:1, both dimensions were subsample by 2:1 ratio giving result shown in Figure 2. For case of sub-sampling the image to a ratio of 8:1, the x-dimension is subsampled by 4:1 ratio, while y-dimension was sub-sampled by 2:1 ratio giving image shown in Figure 3. Finally, for case of sub-sampling image by 16:1 ratio, both dimensions are sub-sampled by 4:1 to 1 ratio, greatly decreasing image resolution, as shown in Figure 4. The MATLAB code to achieve sub-sampling is shown in Figure 30 in Appendix.

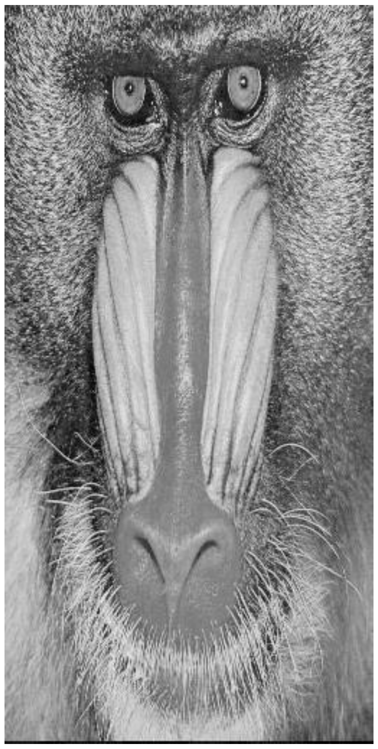


Figure 1: Sub-sampling 2:1

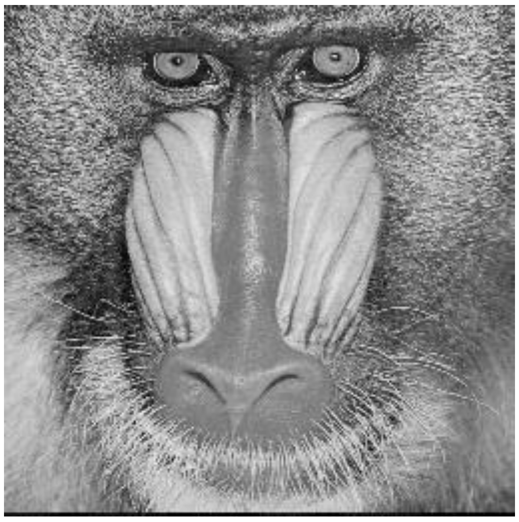


Figure 2: Sub-sampling 4:1

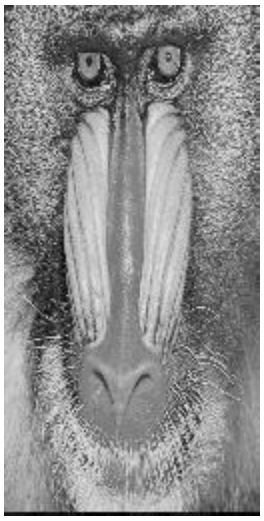


Figure 3: Sub-sampling 8:1

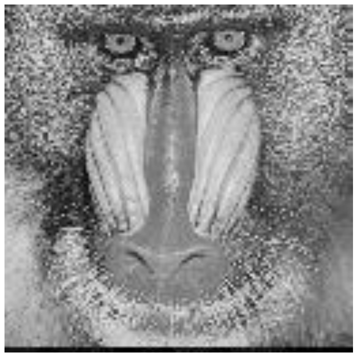


Figure 4: Sub-sampling 16:1

b) Figures 5 to 8 show the results obtained via quantization. As can be seen, 1 and 2 bit quantization gives false contours, because intensity of image is just represented by 2 values instead of 255 values in 1 bit quantization, and 4 values instead of 255 values in 2 bit quantization. The script used to achieve quantization can be seen in Figure 31 in Appendix.

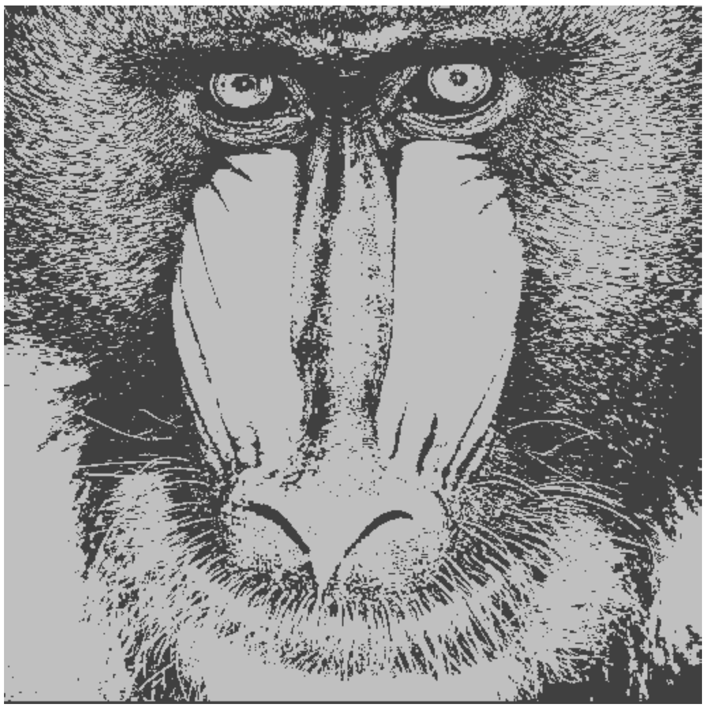


Figure 5: 1 bit quantization

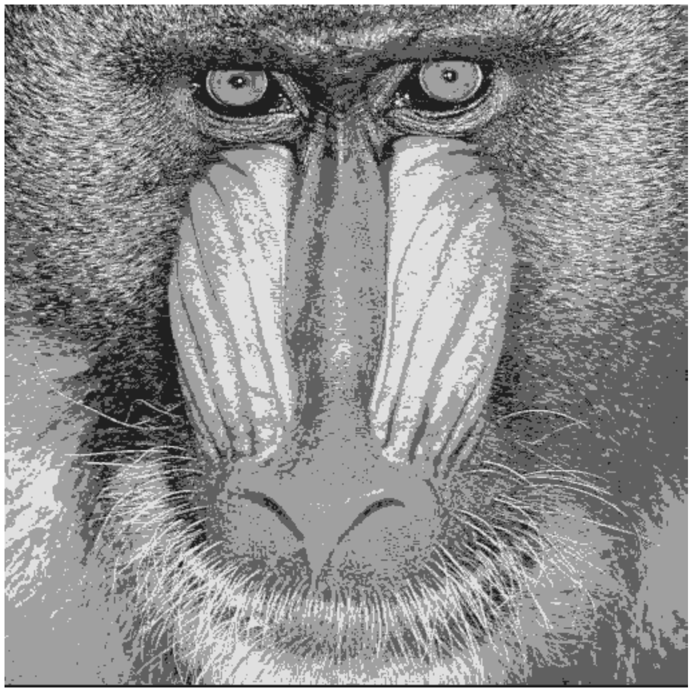


Figure 6: 2 bit quantization

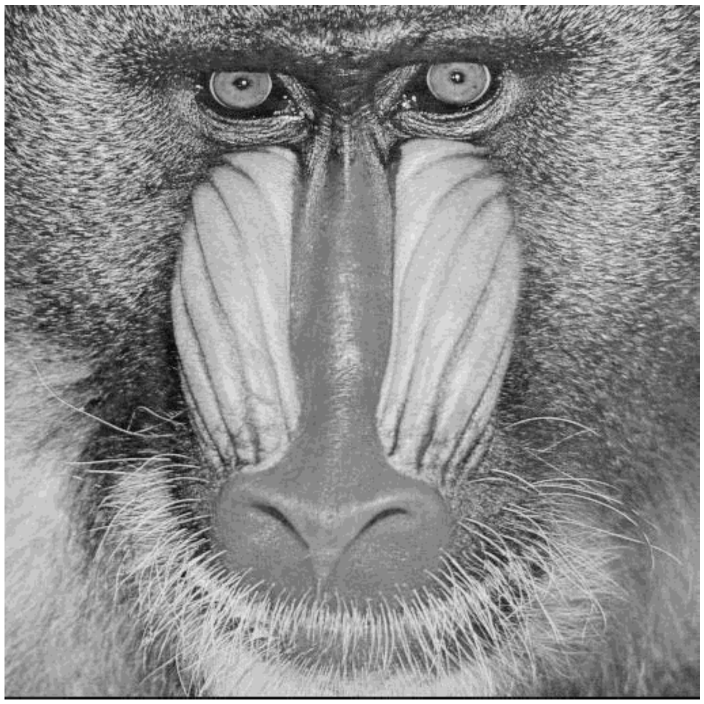


Figure 7: 4 bit quantization

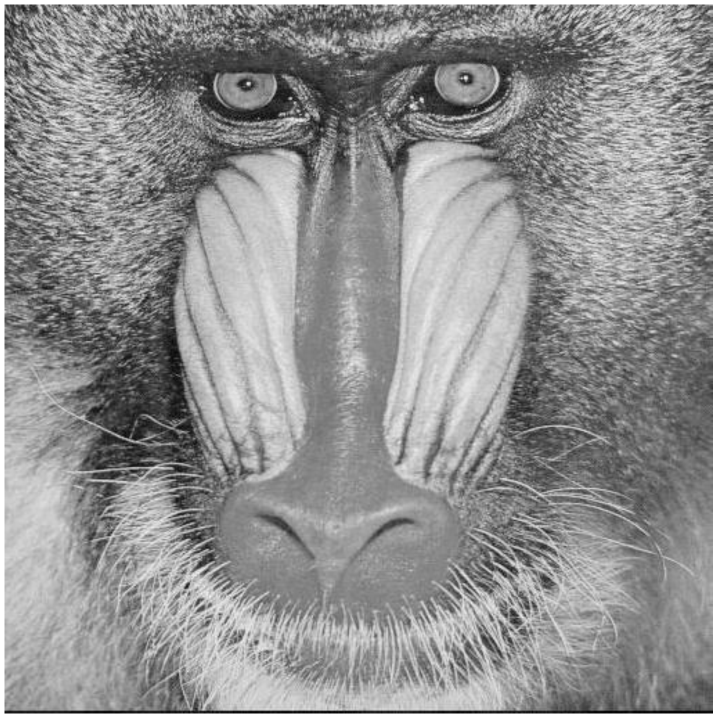


Figure 8: 6 bit quantization

c) It was observed that the perceived image quality differed greatly when having SNR as -30, -10, 0, 10 and 30 dB. Larger SNR's such as 40dB, 30dB, 25dB, 20dB and 10dB result in much better perceived image quality, while lower SNR values decrease perceived image quality as noise is equal, in case of 0dB, l if not more than the signal, in cases of -10dB and -30dB. The resulting image in these cases are shown in figures 9, 10, 11, 12 and 13 respectively. The script used to add Gaussian noise can be seen in Figure 32 in Appendix.

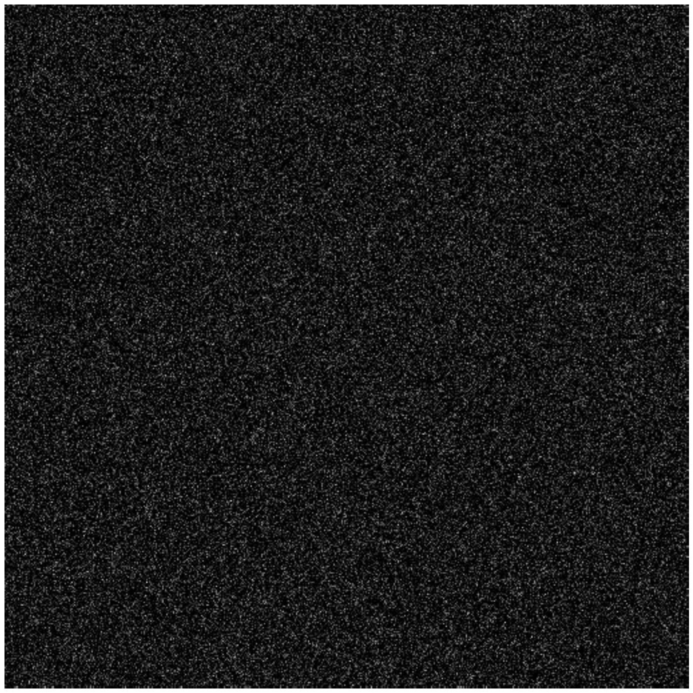


Figure 9: Gaussian noise with SNR = -30dB

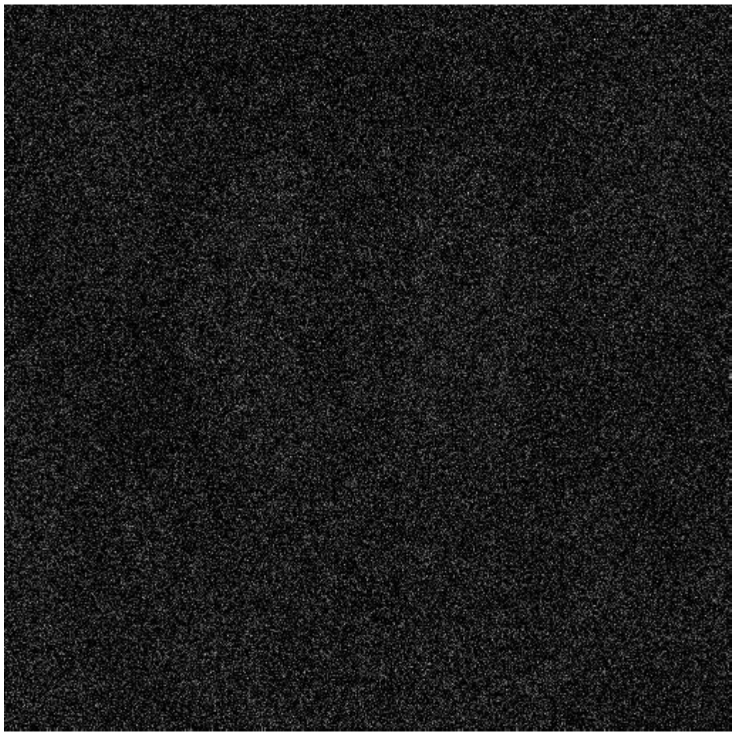


Figure 10: Gaussian noise with SNR = -10dB

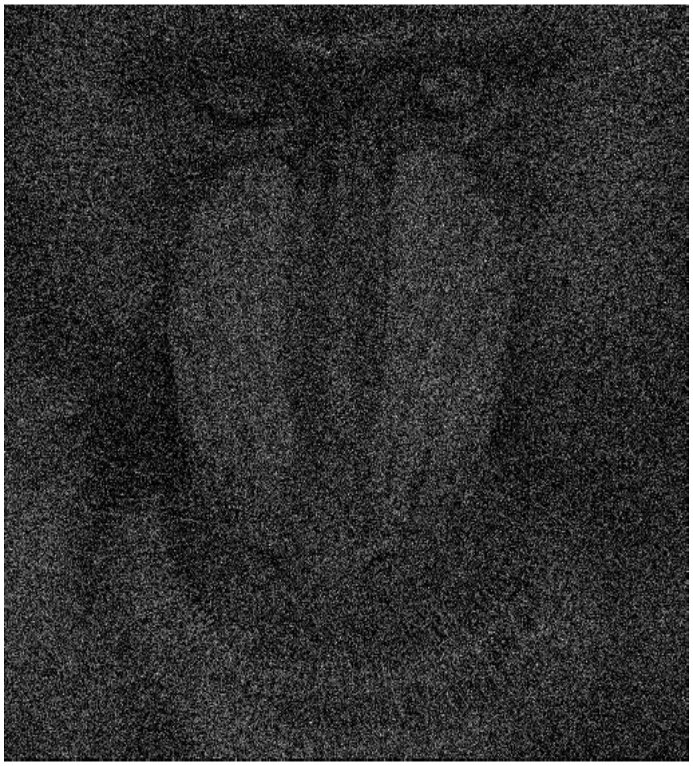


Figure 11: Gaussian noise with SNR = 0dB

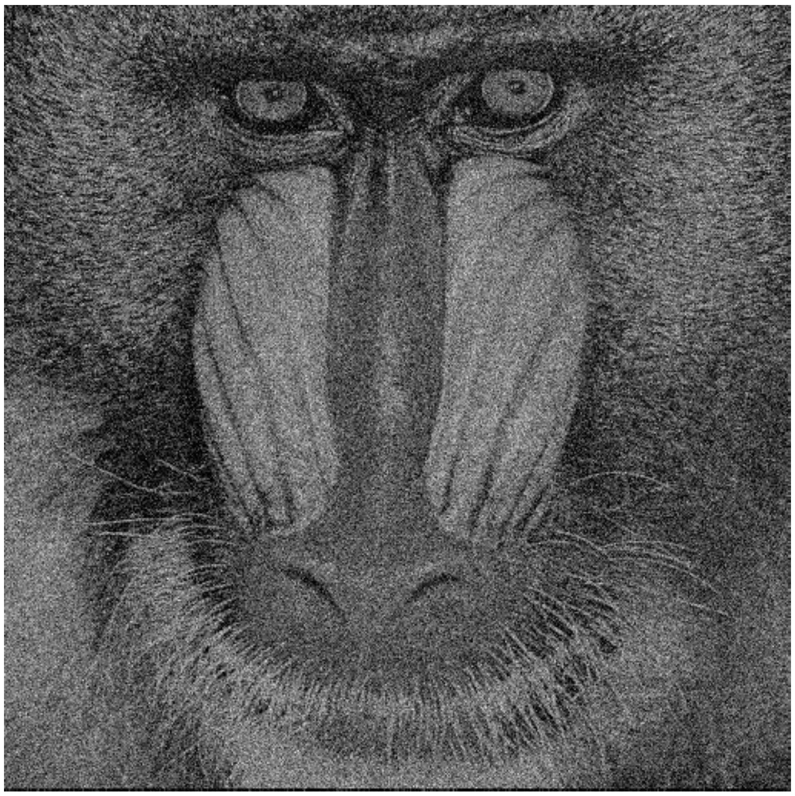


Figure 12: Gaussian noise with SNR = 10dB

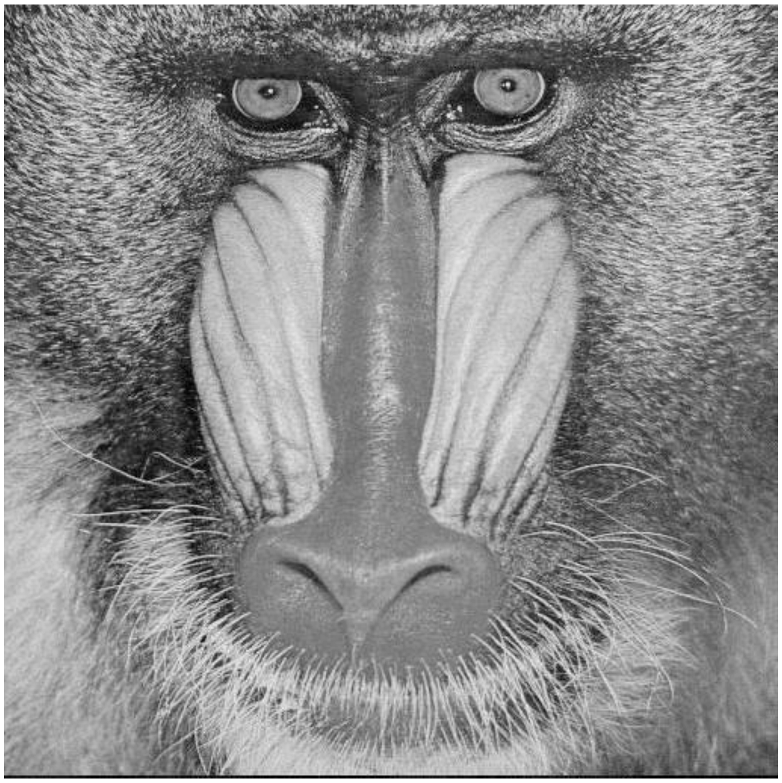


Figure 13: Gaussian noise with SNR = 30dB

d) The resulting image obtained by corrupting the pixels with i.i.d noise for impulse probability 0.1, 1, 5 and 10 percent is shown from figures 14 to 18 respectively. To calculate the SNR, the P_{noise} was calculated by squaring each entry in the noise matrix and summing the result. P_{image} was calculated in a similar manner, and the two powers divided by each other to arrive at the SNR. Log10 of this result was taken and then multiplied by 10 to arrive at SNR in dB. For 0.1% probability, the SNR = 30.81 dB. For 1% probability, SNR = 21.07 dB. For 5% probability, SNR = 14.17 dB. For 10% probability, SNR = 11.16 dB. Figure 33 in appendix shows the script used to add impulsive noise.

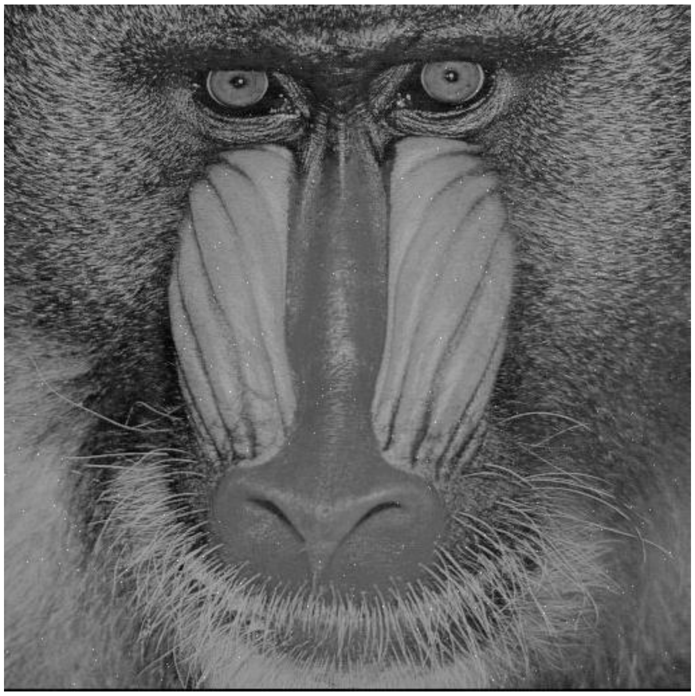


Figure 14: probability iid = 0.1%

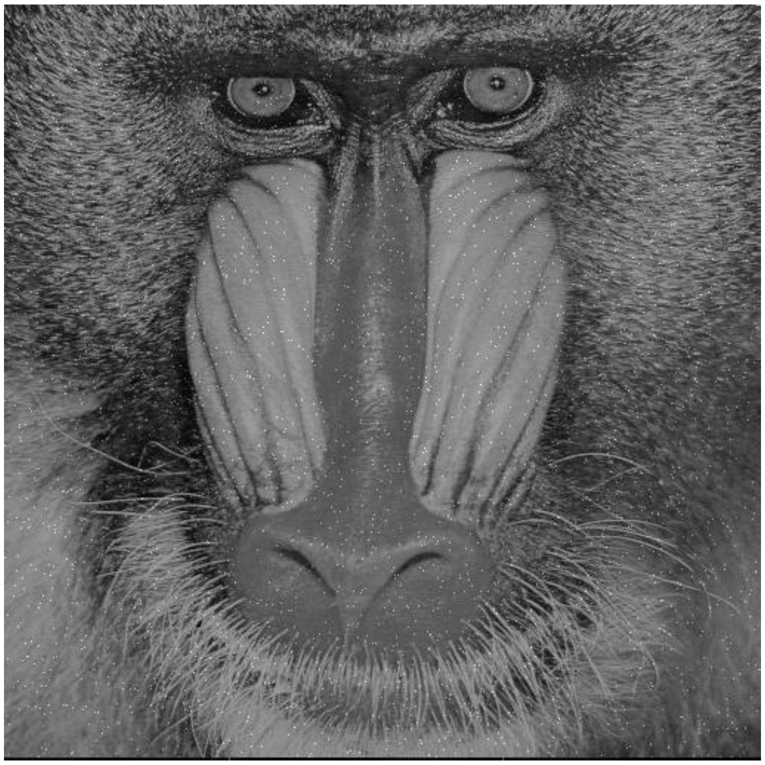


Figure 15: probability iid = 1%

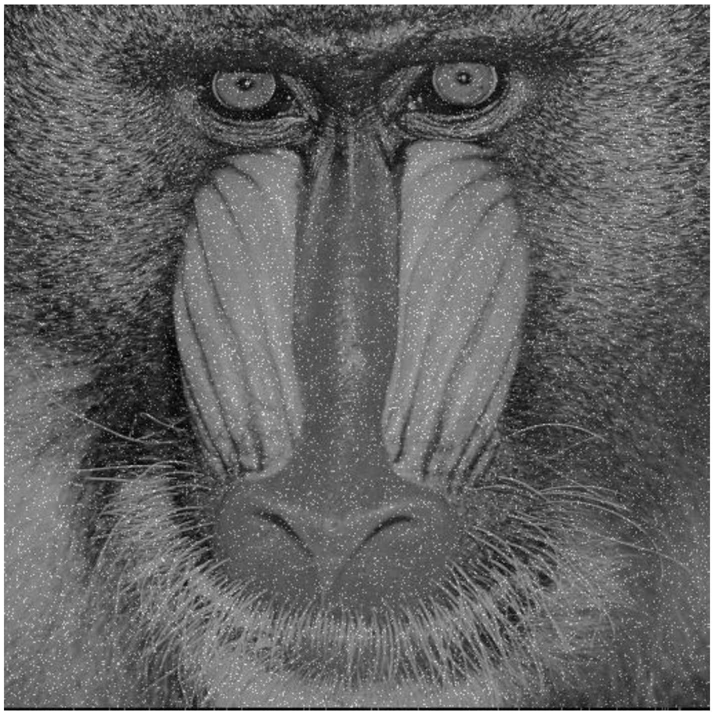


Figure 16: probability iid = 5%

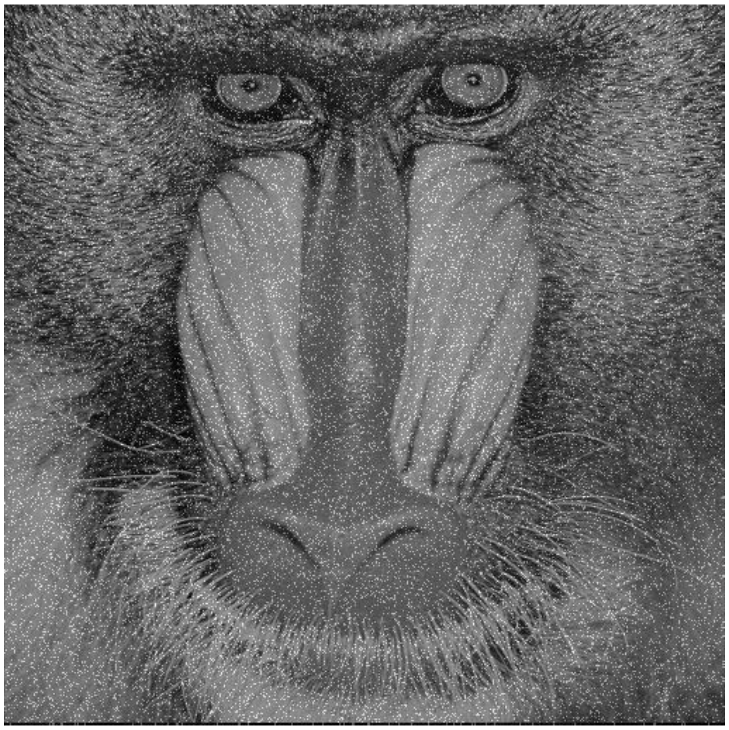


Figure 17: probability iid = 10%

e) The Gaussian noise seems to add black pixels as noise in the image, whereas the i.i.d. one seems to add white impulsive pixels.

Question 2

a) It was observed that with higher values of N, the blurring intensity increases. Figure 18 shows the image blurred with N = 20 and figure 19 shows the image blur with N = 5. Figure 34 in Appendix shows how the images were blurred.

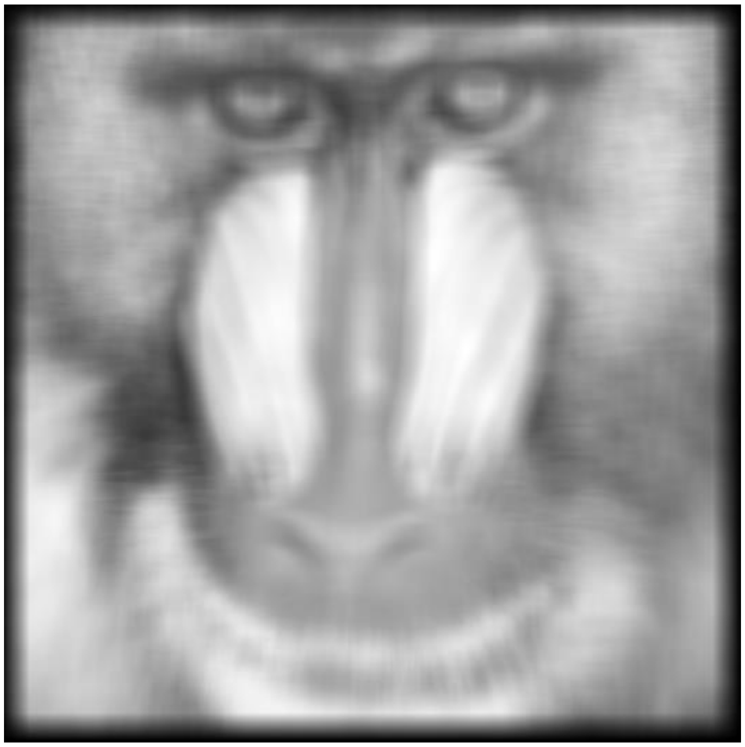


Figure 18: blur image with N = 20

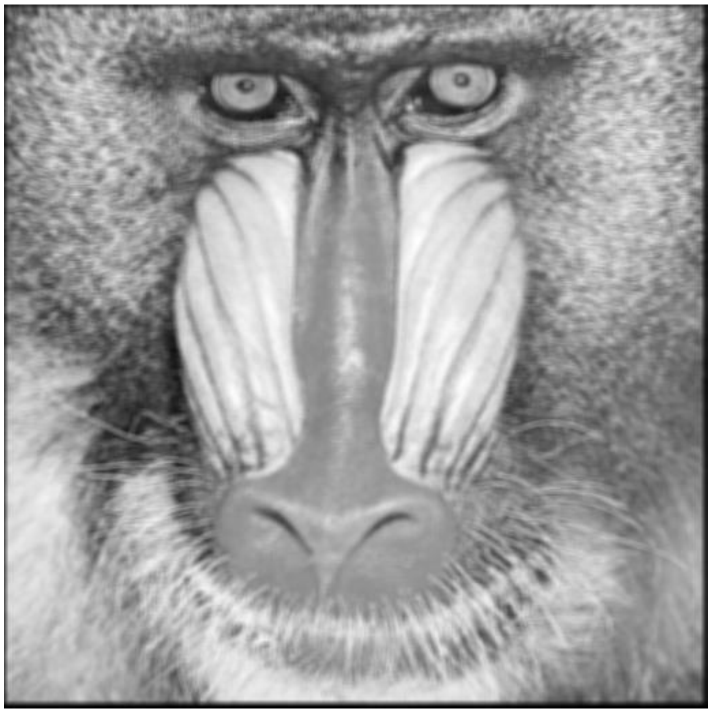


Figure 19: blur image with N = 5

b) For this test, the SNR for Gaussian blur was set as 10dB and the resulting image is given in Figure 12. Figure 20 shows the result obtained when applying a 3 by 3 low pass filter on the noisy image. Figure 21 shows the result obtained by applying 5 by 5 low pass filter on the image. The filter was applied in frequency domain by taking FFT of the original image as well as the filter and convolving them. The results are obtained by then taking the inverse FFT of the convolution. Figure 35 shows the script used to reduce Gaussian noise.

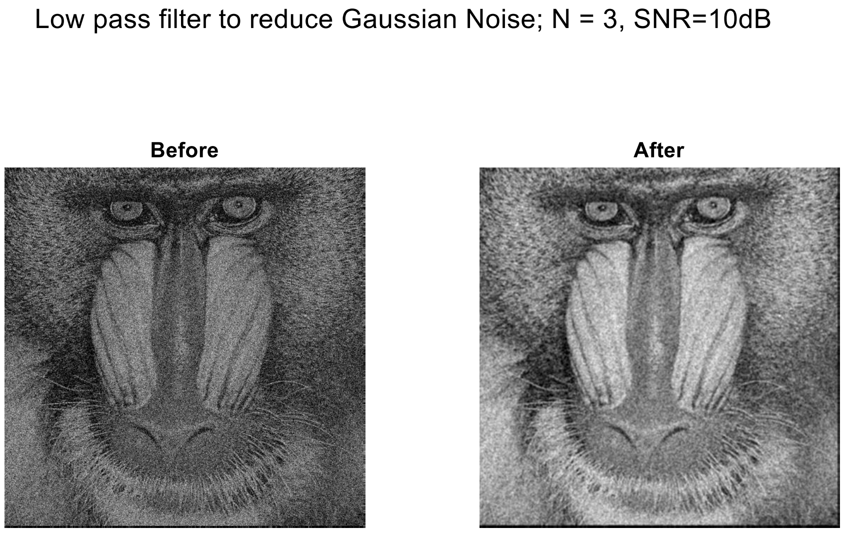


Figure 20: 3x3 low pass filter on image with Gaussian noise. SNR = 10dB

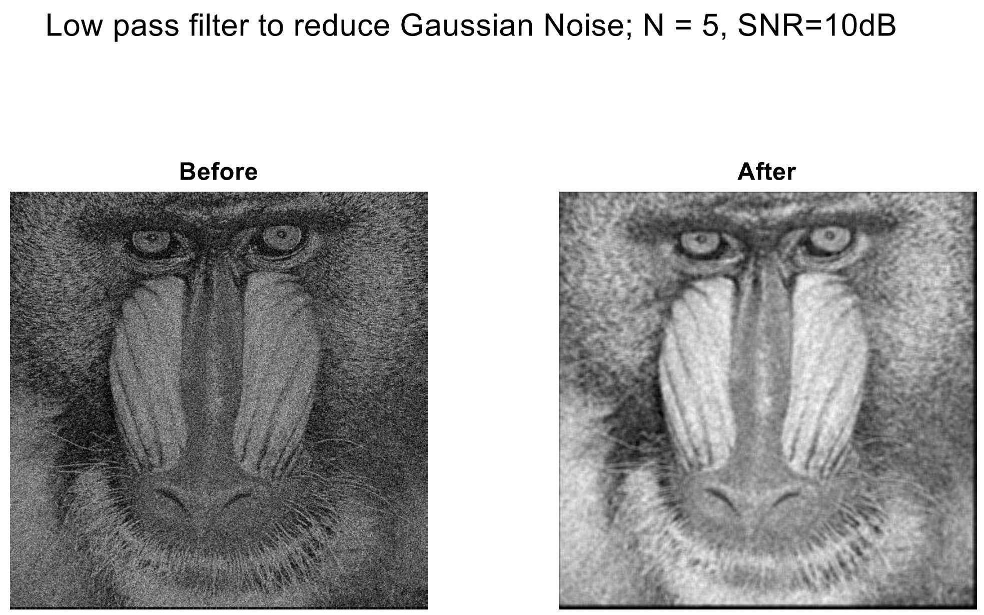


Figure 21: 5x5 low pass filter on image with Gaussian noise. SNR = 10dB

c) For this experiment, the probability of i.i.d. impulse was set as 10%, where the resulting noisy image is shown in figure 17. The result obtained by applying 3 by 3 low pass filter on this noisy image is shown in figure 22 and the result of 5 by 5 filter is shown in figure 23. This approach works because it blurs out the i.i.d. impulse created on the image, therefore fixing the noise. Figure 36 shows the script used to reduce impulsive noise.

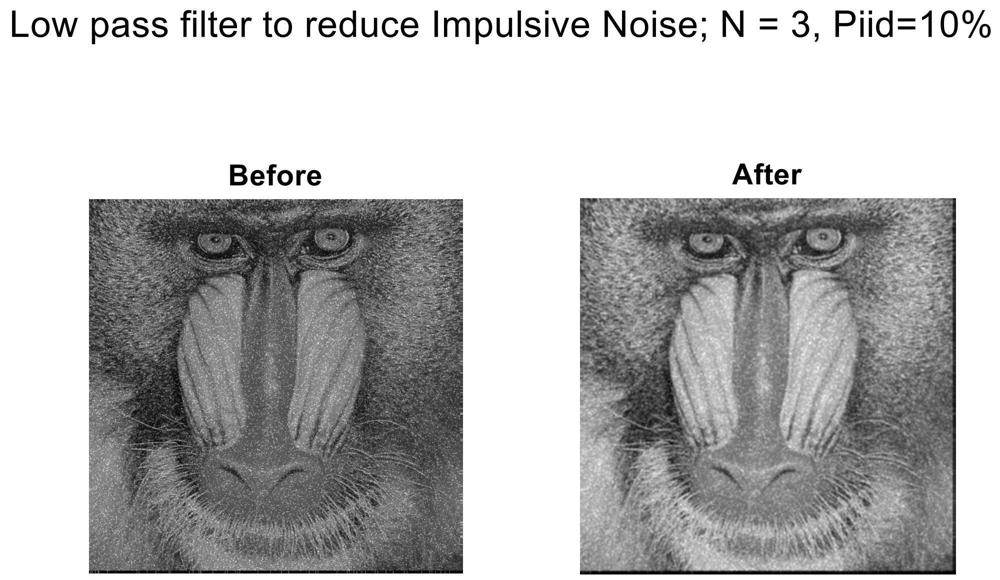


Figure 22: 3x3 low pass filter on image with iid noise. Piid = 10%

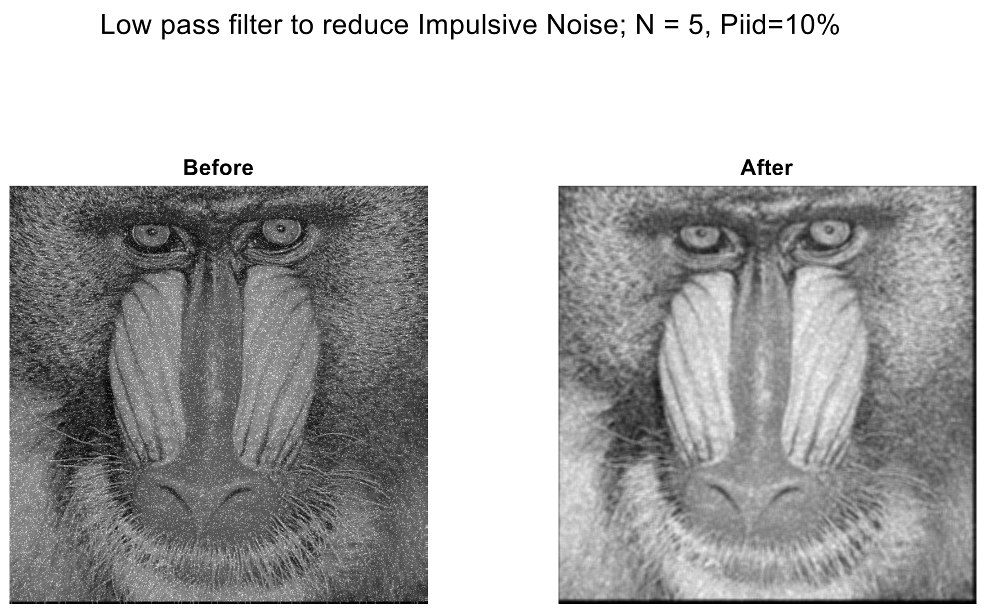


Figure 23: 5x5 low pass filter on image with iid noise. Piid = 10%

d) The result obtained by applying the 3 by 3 median filter is shown in figure 24, while figure 26 shows the result of 5 by 5 median filter. This filter takes the median of every 3-by-3 subsection of the image and therefore, it would average out the i.i.d. impulse encountered in most cases. Thus, median filter works in removing most of the impulsive noise. Figure 37 shows the script used to reduce impulsive noise using Median filter.

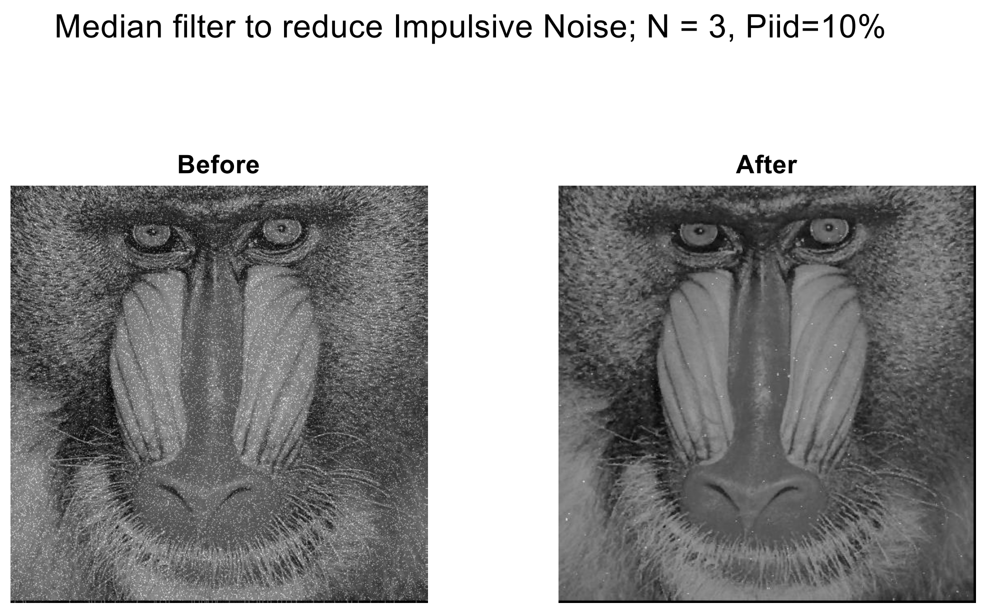


Figure 24: 3x3 Median Filter

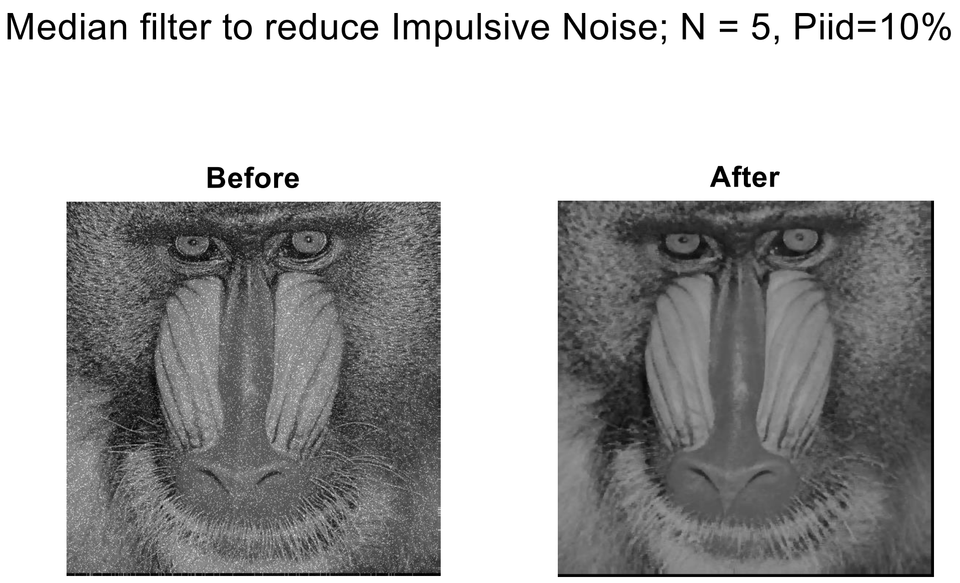


Figure 25: 5x5 median filter

e) Applying the high pass filter on the image gives the resulting image of Figure 26. This seems to have passed the high intensity pixels in the image, therefore giving a highlight effect. Figure 38 shows the script used to pass the image through high pass filter

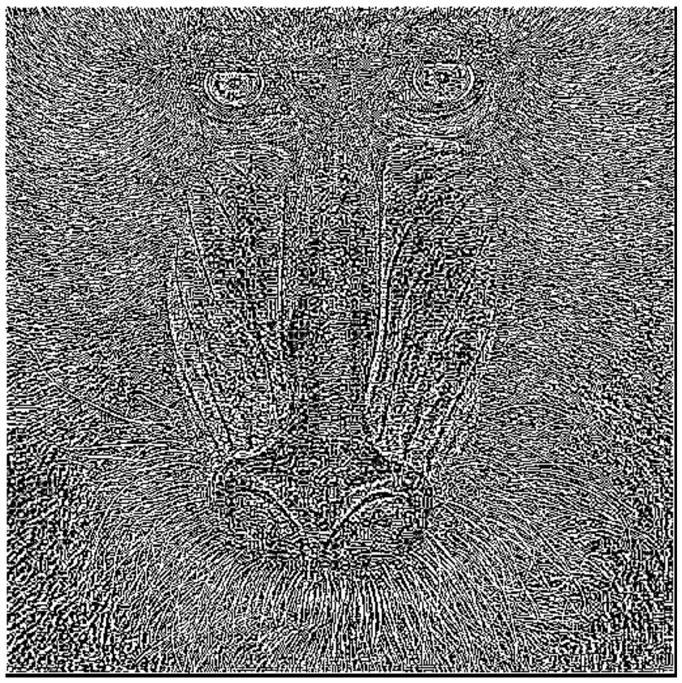


Figure 26: High pass filter result

Question 3

a) Figure 27 shows the 2-D fft plot result on the original image. The image contains some bright as well as dark spots, thus indicating a range of frequencies varying from high frequencies to lower frequencies.

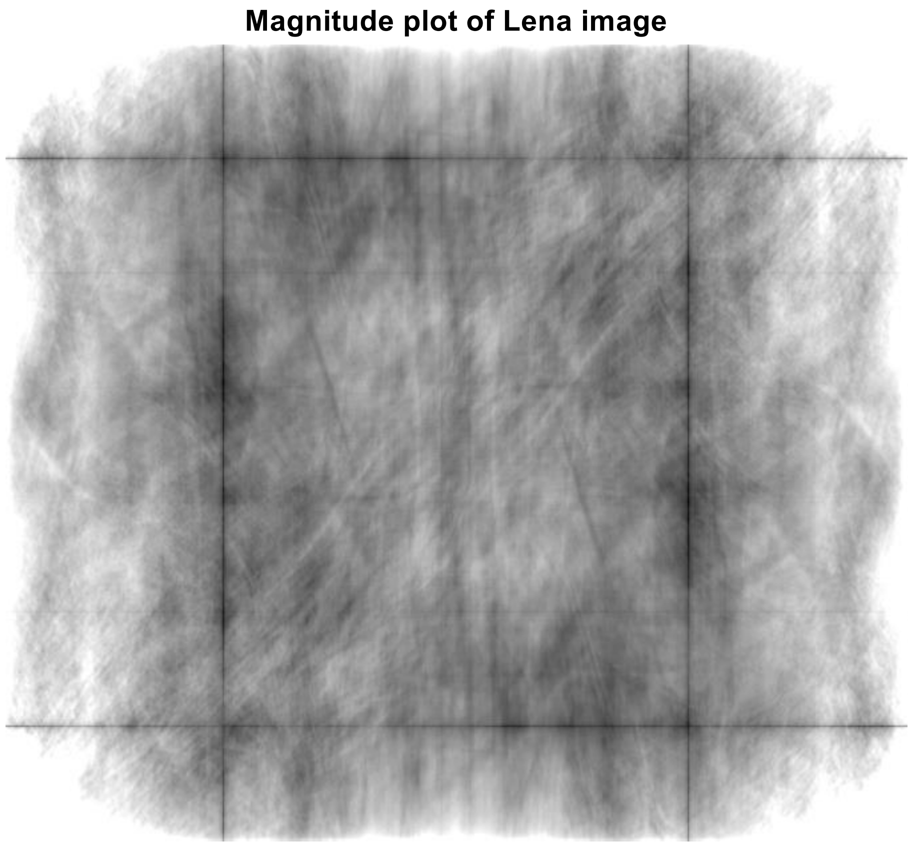


Figure 27: 2D FFT Magnitude plot of Lena Image

b) Figure 28 below shows the magnitude plot for the Japanese flag. Figure 29 shows the plot for a single impulse signal and Figure 30 shows the magnitude plot for 2D harmonic spatial signal.

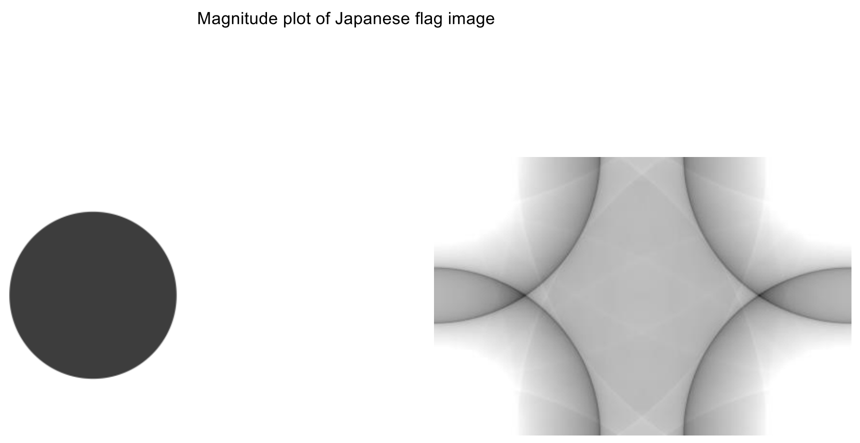


Figure 28: 2D FFT magnitude plot of Japanese flag image

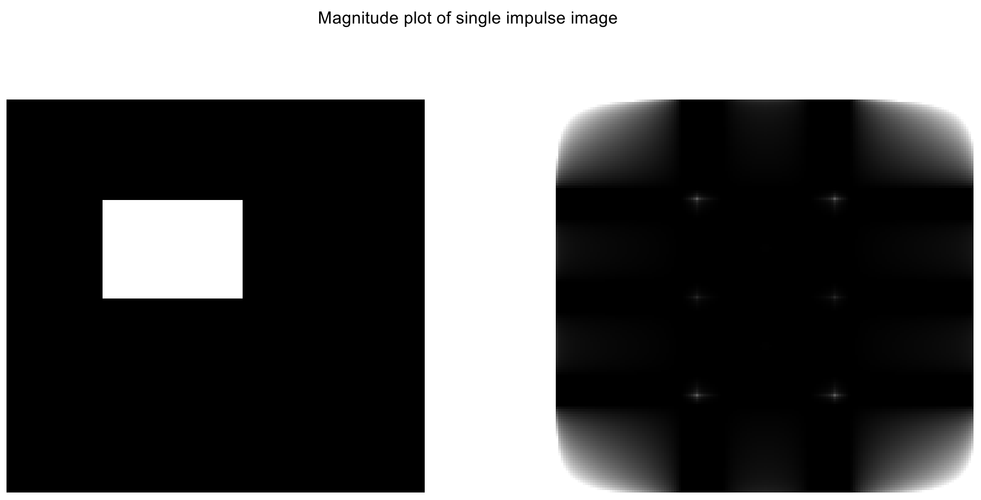


Figure 29: 2D FFT magnitude plot of impulse image

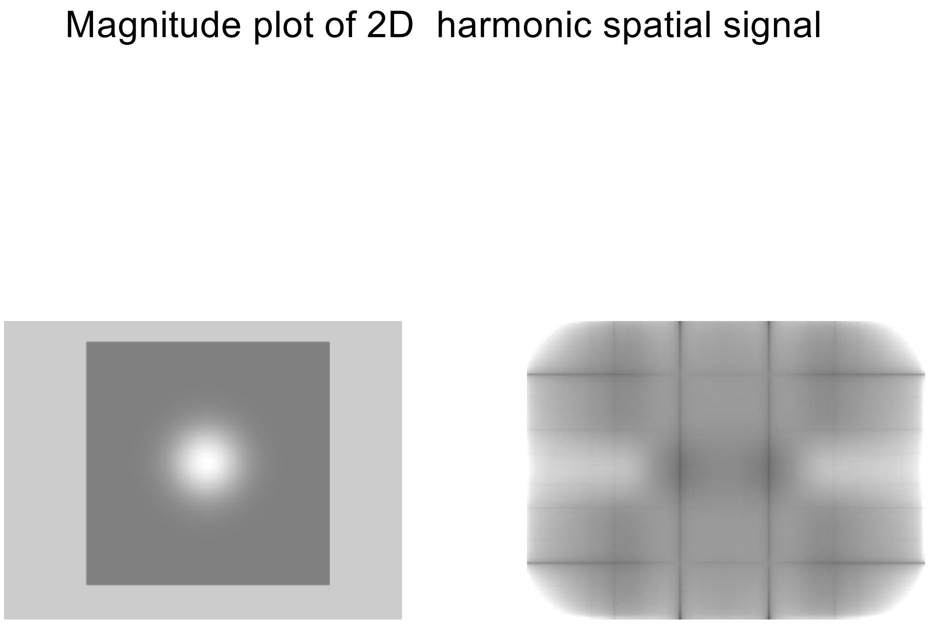


Figure 30: 2D FFT magnitude plot of spatial harmonic signal

c) Figure 31 shows the 2D FFT magnitude plot of the low pass filter used to blur the image as used in 2(a). The plot is obtained by convolving the filter with Baboon's image and taking the 2D FFT of the result, taking its magnitude and then taking inverse FFT and plotting the image. Figure 32 repeats the same experiment with the high pass filter, as used in 2(e). The same experiment was repeated for the median filter used in 2(d), and the results plotted and shown as in Figure 33. These plots reflect what was observed in each of these filters, where Figure 31 shows the intensities as blurred, Figure 32 shows high intensity and sharp magnitudes and Figure 33 shows the same magnitude plot as the magnitude plot observed with the Baboon, except that magnitudes are slightly darker.

Magnitude plot of the low pass filter convoluted with Baboon image

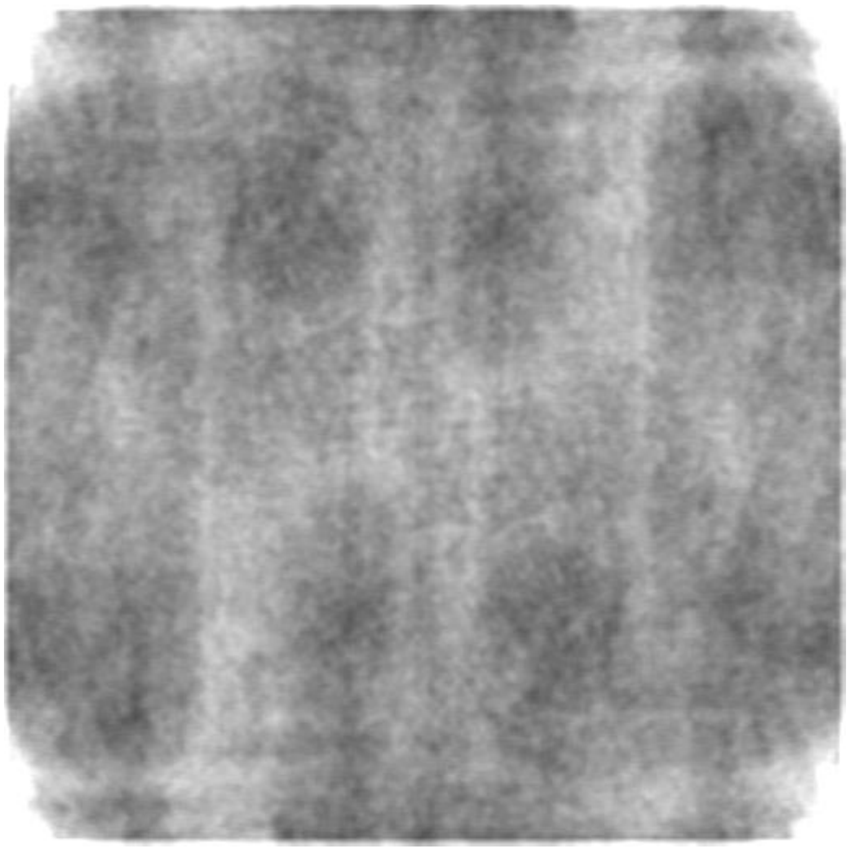


Figure 31: Low pass filter magnitude plot, used to blur an image

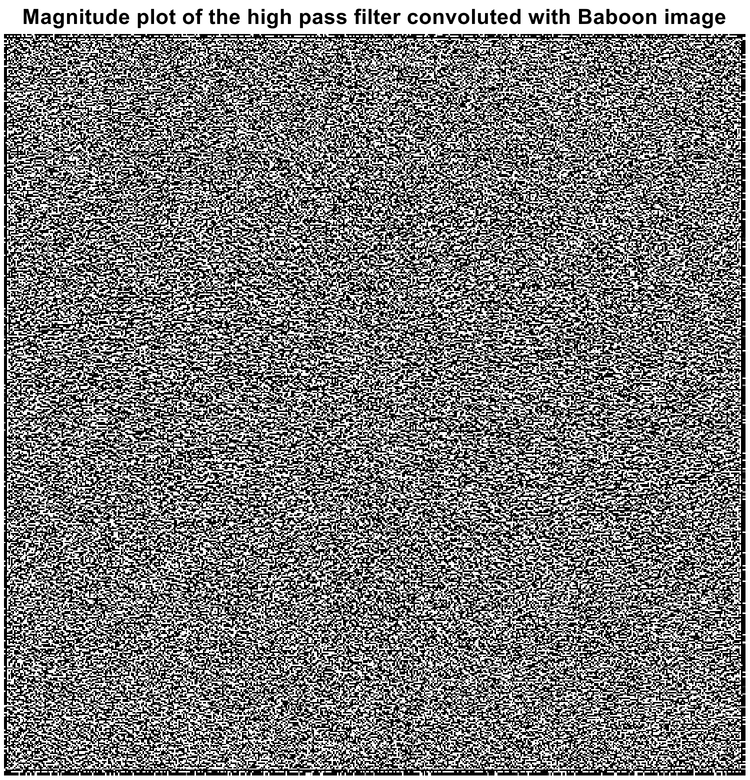


Figure 32: Magnitude plot of the high pass filter

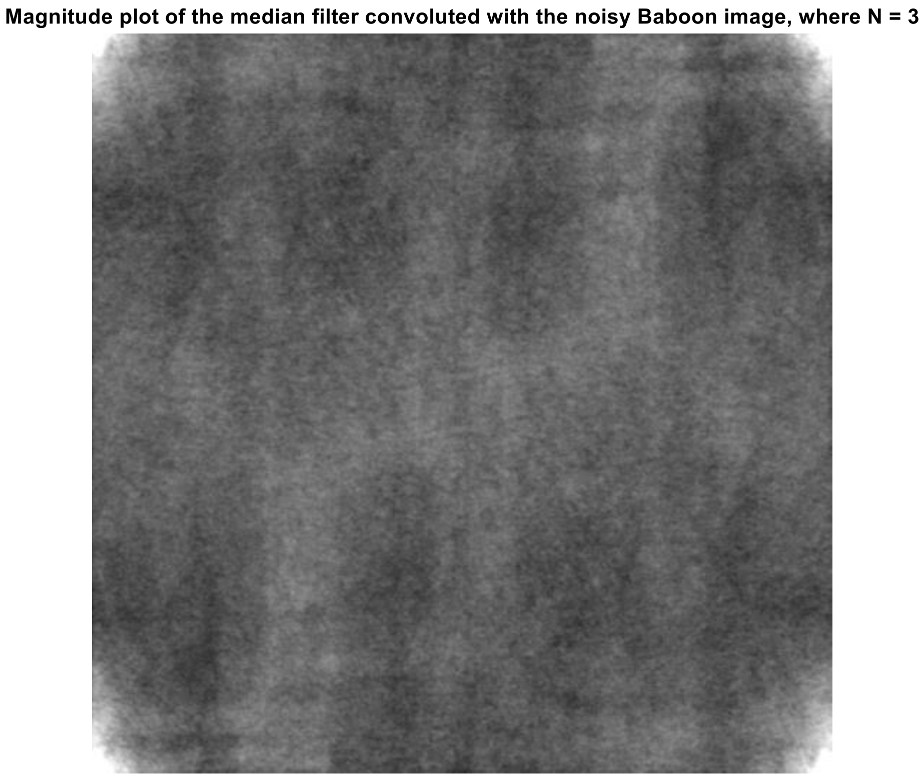


Figure 33: Magnitude plot of the median filter

d) The magnitude of the image was taken by first taking the FFT of the mandrill image and then taking the absolute of the result, as shown in Figure 28. This shows how many different frequencies or intensities are present in the image.

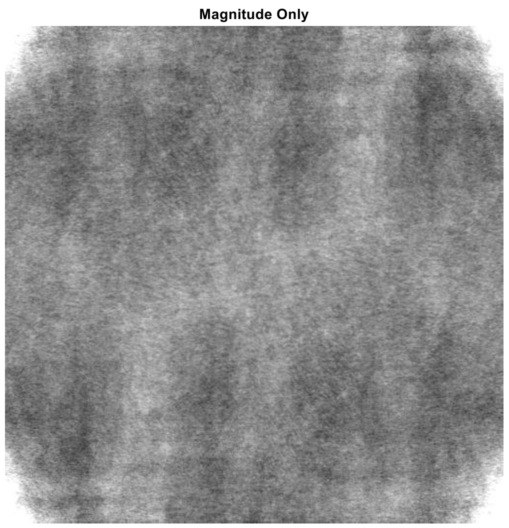


Figure 34: Magnitude of 2D Fourier transform of Mandrill image

e) The frequency of the image was taken by first computing the FFT of the Lena image and then dividing the result with the magnitude to set it to one. This gave the resulting image as shown in Figure 29. This shows the relative positioning of each intensity at the image.

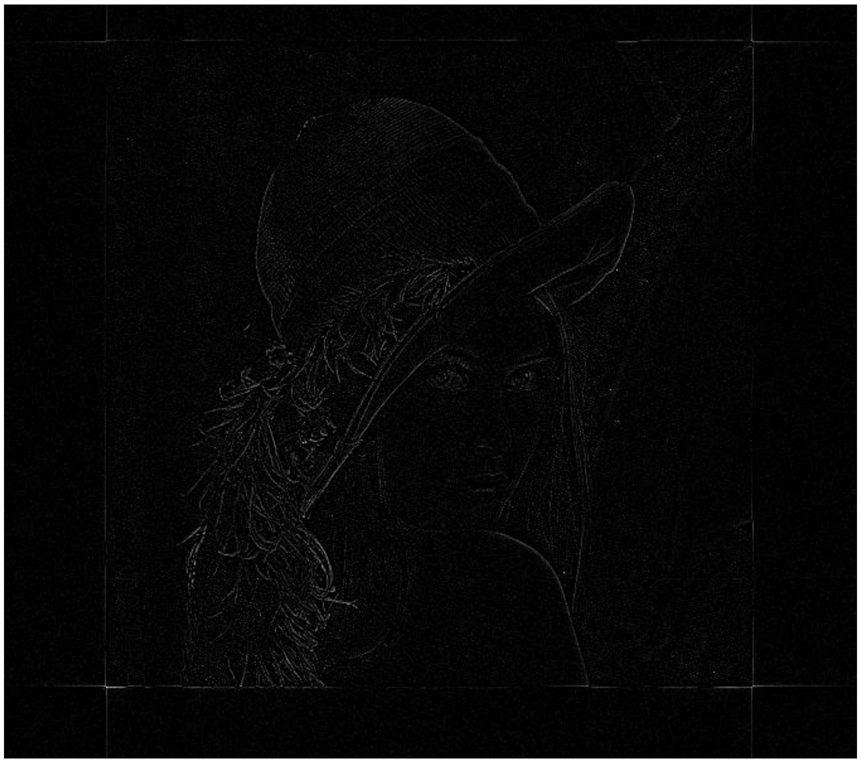


Figure 35: Phase 2D Fourier transform of Lena image

f) The constructed image in part (e) is better because there's still a magnitude information, although, set to one, when reconstructing image in phase only-information. This contains more information than having an image reconstructed with magnitude only where the phase is set to zero. The entropy of image in part (e) is larger than part (d).

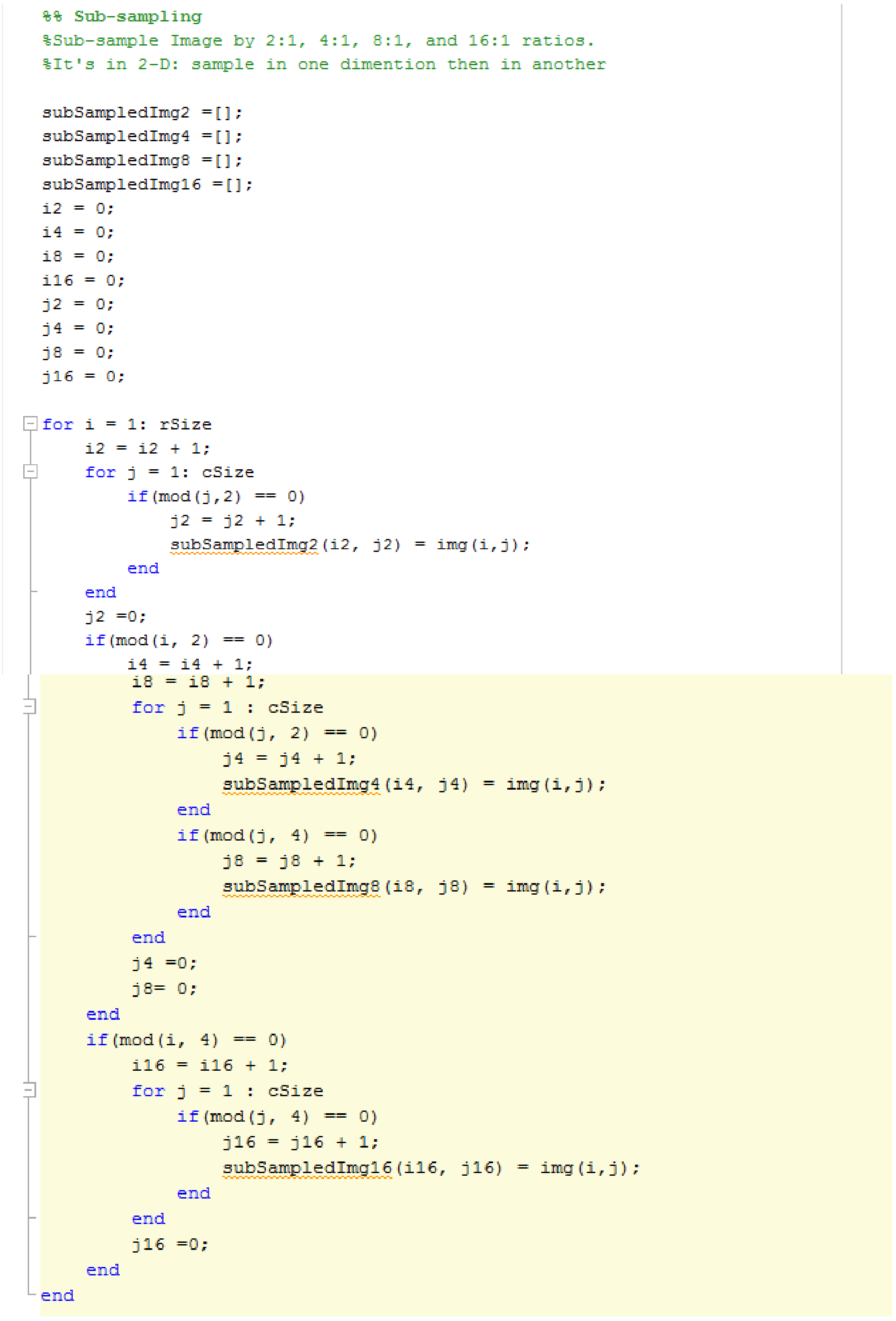
Question 4:

a) The artificial image of the square is generated on each X,Y where X and Y are the dimensions of the image. The background is set to white by making the 4-bit intensities of R (red color intensity), G (green color intensity) and B (blue color intensity) to 1111. The square is generated using an if statement at X and Y. For a square of side of 30 pixels the intensities at X = 100 to X=130, and at Y= 100 and Y = 140, since VGA supports up to 640x480 Y side has to be multiplied by 1.33 to have a square, are changed to 0000 for R,G, and B for a black square. Using a switch case statements where cases represent switches on board, different sizes of the square are changed by changing X and Y if statement pixel parameters. The Verilog code is shown in Figure 45 in Appendix. The square module is then connected to input/output ports in Video_generator.bdf as shown in Figure 46.

b) To move square across the screen, the square produced is moved one pixel in X dimension and one pixel in Y dimension on each positive edge clock cycle. Different speeds are achieved by moving square by more or less pixels in each dimension on each clock cycle. Also, switch case statements are used for different speeds controlled by switched on FPGA board. The Verilog code is shown in Figures 47 and 48 in Appendix. This module is also connected to input/output ports in Video_generator.bdf as shown in Figure 46. The FSMCLK is connected to VGA_VS in DE2_Default Verilog module.

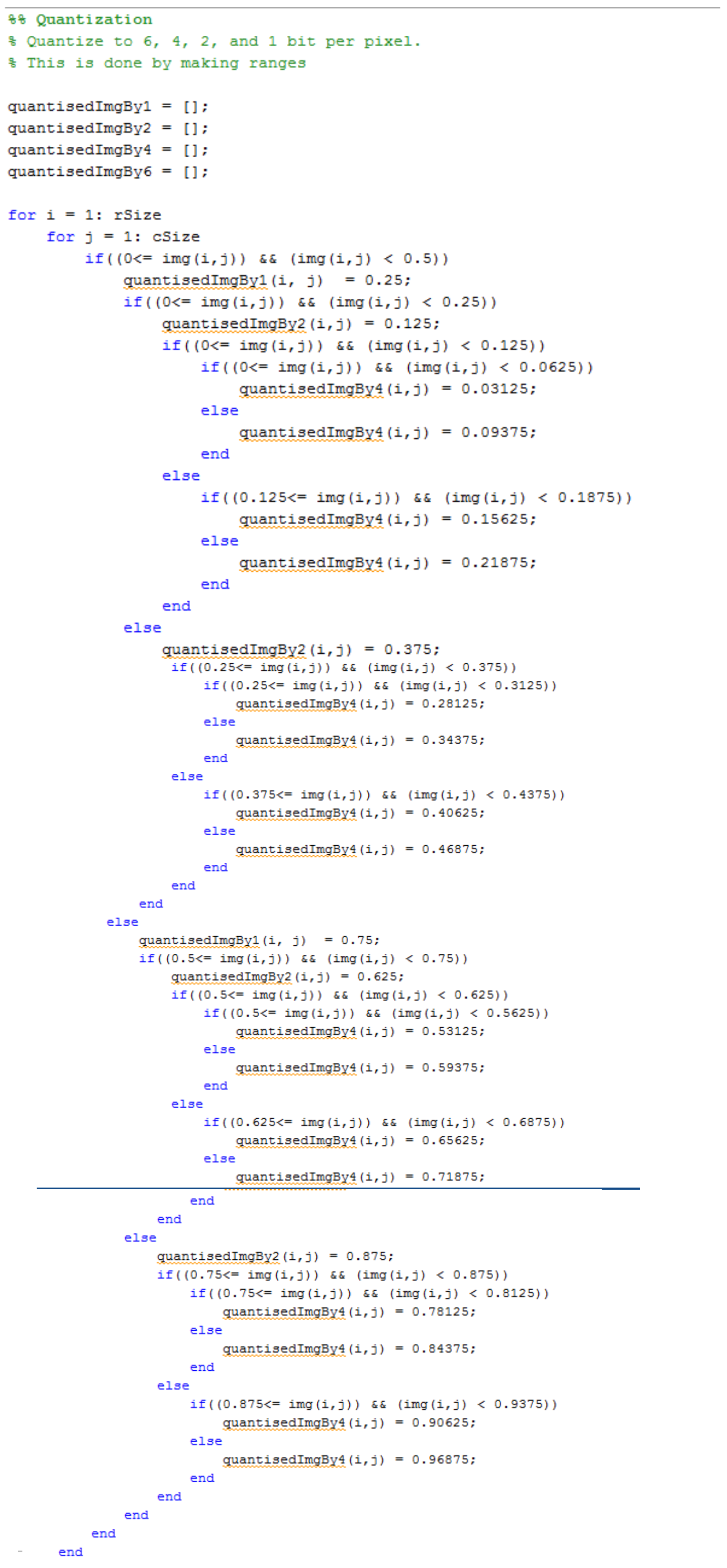
c) The moving square in part (b) is made to change color in each second by having a counter that increments on each clock cycle. Since the clock has a frequency of 60 Hz, the counter has to reach 60, equivalent to one second, for the color of square to be changed. For different colors of the rainbow, the intensities R, G and B were changed accordingly. The Verilog code for this part is shown in **Figure** in Appendix.

Appendix



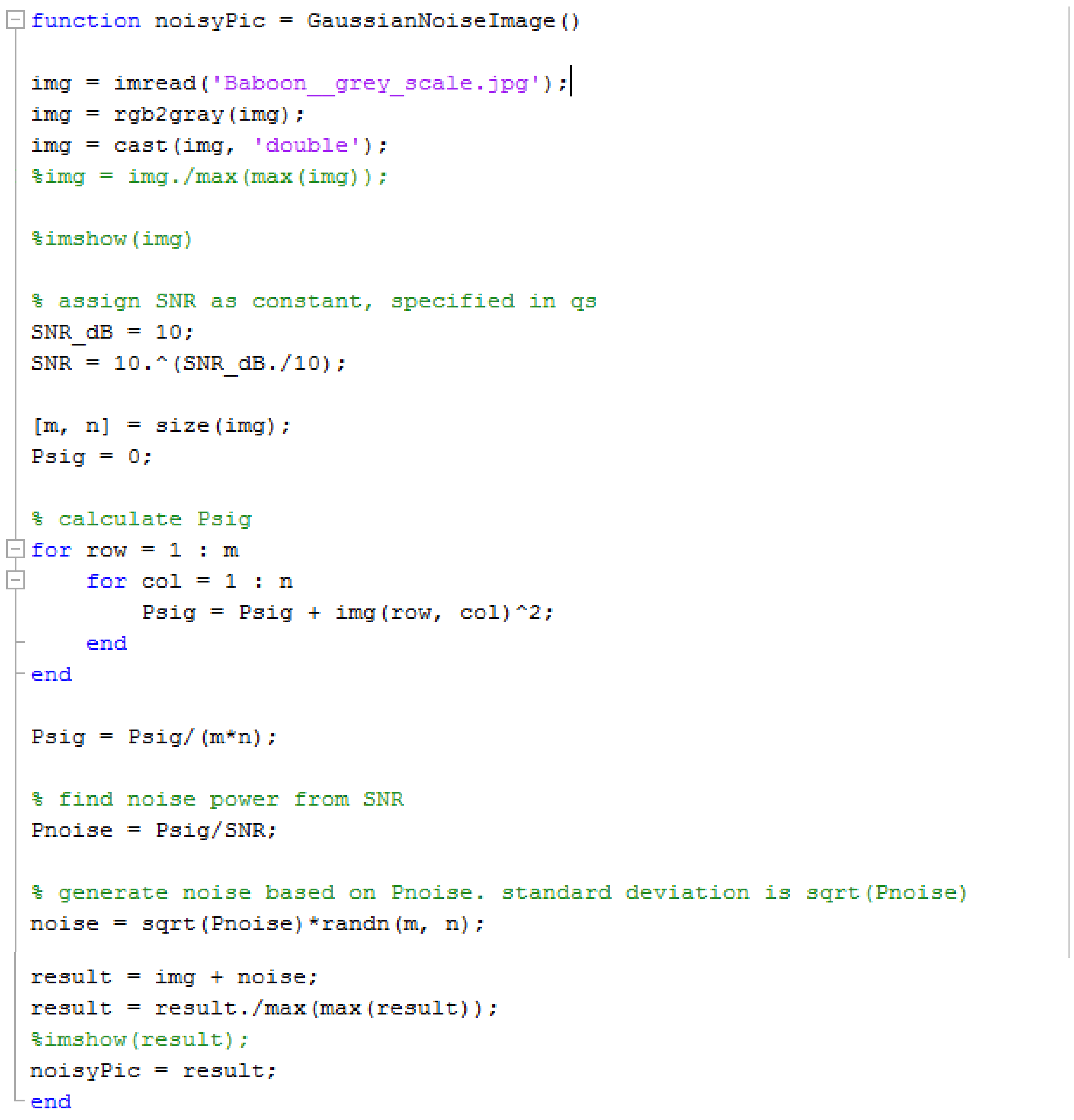
```
%% Sub-sampling
%Sub-sample Image by 2:1, 4:1, 8:1, and 16:1 ratios.
%It's in 2-D: sample in one dimention then in another

subSampledImg2 =[];
subSampledImg4 =[];
subSampledImg8 =[];
subSampledImg16 =[];
i2 = 0;
i4 = 0;
i8 = 0;
i16 = 0;
j2 = 0;
j4 = 0;
j8 = 0;
j16 = 0;

for i = 1: rSize
    i2 = i2 + 1;
    for j = 1: cSize
        if(mod(j,2) == 0)
            j2 = j2 + 1;
            subSampledImg2(i2, j2) = img(i,j);
        end
    end
    j2 =0;
    if(mod(i, 2) == 0)
        i4 = i4 + 1;
        i8 = i8 + 1;
        for j = 1 : cSize
            if(mod(j, 2) == 0)
                j4 = j4 + 1;
                subSampledImg4(i4, j4) = img(i,j);
            end
            if(mod(j, 4) == 0)
                j8 = j8 + 1;
                subSampledImg8(i8, j8) = img(i,j);
            end
        end
        j4 =0;
        j8= 0;
    end
    if(mod(i, 4) == 0)
        i16 = i16 + 1;
        for j = 1 : cSize
            if(mod(j, 4) == 0)
                j16 = j16 + 1;
                subSampledImg16(i16, j16) = img(i,j);
            end
        end
        j16 =0;
    end
end
```

Figure 36: Sub-sampling algorithm



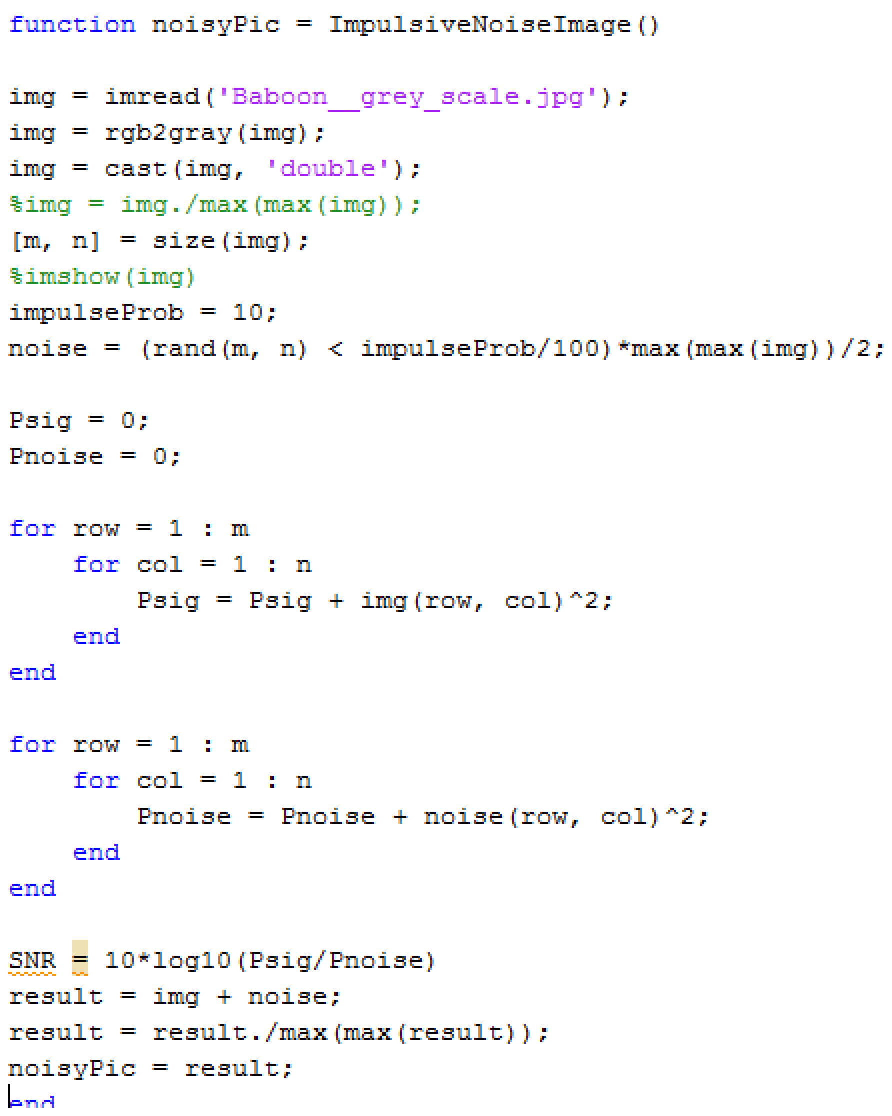
```
%% Quantization
% Quantize to 6, 4, 2, and 1 bit per pixel.
% This is done by making ranges

quantisedImgBy1 = [];
quantisedImgBy2 = [];
quantisedImgBy4 = [];
quantisedImgBy6 = [];

for i = 1: rSize
    for j = 1: cSize
        if((0<= img(i,j)) && (img(i,j) < 0.5))
            quantisedImgBy1(i, j)  = 0.25;
            if((0<= img(i,j)) && (img(i,j) < 0.25))
                quantisedImgBy2(i,j) = 0.125;
                if((0<= img(i,j)) && (img(i,j) < 0.125))
                    if((0<= img(i,j)) && (img(i,j) < 0.0625))
                        quantisedImgBy4(i,j) = 0.03125;
                    else
                        quantisedImgBy4(i,j) = 0.09375;
                    end
                else
                    if((0.125<= img(i,j)) && (img(i,j) < 0.1875))
                        quantisedImgBy4(i,j) = 0.15625;
                    else
                        quantisedImgBy4(i,j) = 0.21875;
                    end
                end
            else
                quantisedImgBy2(i,j) = 0.375;
                if((0.25<= img(i,j)) && (img(i,j) < 0.375))
                    if((0.25<= img(i,j)) && (img(i,j) < 0.3125))
                        quantisedImgBy4(i,j) = 0.28125;
                    else
                        quantisedImgBy4(i,j) = 0.34375;
                    end
                else
                    if((0.375<= img(i,j)) && (img(i,j) < 0.4375))
                        quantisedImgBy4(i,j) = 0.40625;
                    else
                        quantisedImgBy4(i,j) = 0.46875;
                    end
                end
            end
        else
            quantisedImgBy1(i, j)  = 0.75;
            if((0.5<= img(i,j)) && (img(i,j) < 0.75))
                quantisedImgBy2(i,j) = 0.625;
                if((0.5<= img(i,j)) && (img(i,j) < 0.625))
                    if((0.5<= img(i,j)) && (img(i,j) < 0.5625))
                        quantisedImgBy4(i,j) = 0.53125;
                    else
                        quantisedImgBy4(i,j) = 0.59375;
                    end
                else
                    if((0.625<= img(i,j)) && (img(i,j) < 0.6875))
                        quantisedImgBy4(i,j) = 0.65625;
                    else
                        quantisedImgBy4(i,j) = 0.71875;
                    end
                end
            else
                quantisedImgBy2(i,j) = 0.875;
                if((0.75<= img(i,j)) && (img(i,j) < 0.875))
                    if((0.75<= img(i,j)) && (img(i,j) < 0.8125))
                        quantisedImgBy4(i,j) = 0.78125;
                    else
                        quantisedImgBy4(i,j) = 0.84375;
                    end
                else
                    if((0.875<= img(i,j)) && (img(i,j) < 0.9375))
                        quantisedImgBy4(i,j) = 0.90625;
                    else
                        quantisedImgBy4(i,j) = 0.96875;
                    end
                end
            end
        end
    end
```

Figure 37: Quantization of image



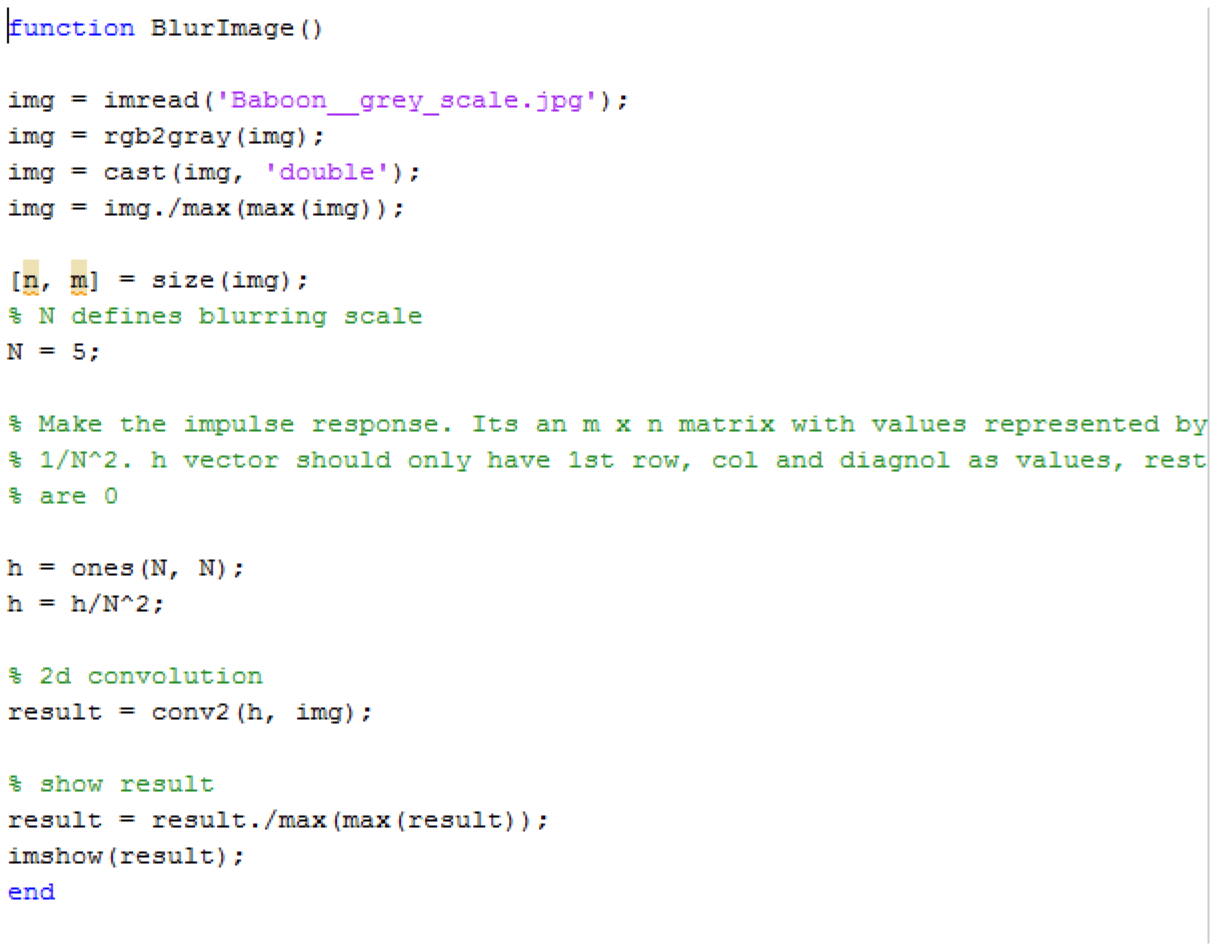
```matlab
function noisyPic = GaussianNoiseImage()

img = imread('Baboon__grey_scale.jpg');
img = rgb2gray(img);
img = cast(img, 'double');
%img = img./max(max(img));

%imshow(img)

% assign SNR as constant, specified in qs
SNR_dB = 10;
SNR = 10.^(SNR_dB./10);

[m, n] = size(img);
Psig = 0;

% calculate Psig
for row = 1 : m
    for col = 1 : n
        Psig = Psig + img(row, col)^2;
    end
end

Psig = Psig/(m*n);

% find noise power from SNR
Pnoise = Psig/SNR;

% generate noise based on Pnoise. standard deviation is sqrt(Pnoise)
noise = sqrt(Pnoise)*randn(m, n);

result = img + noise;
result = result./max(max(result));
%imshow(result);
noisyPic = result;
end
```

Figure 38: Gaussian Noise corruption



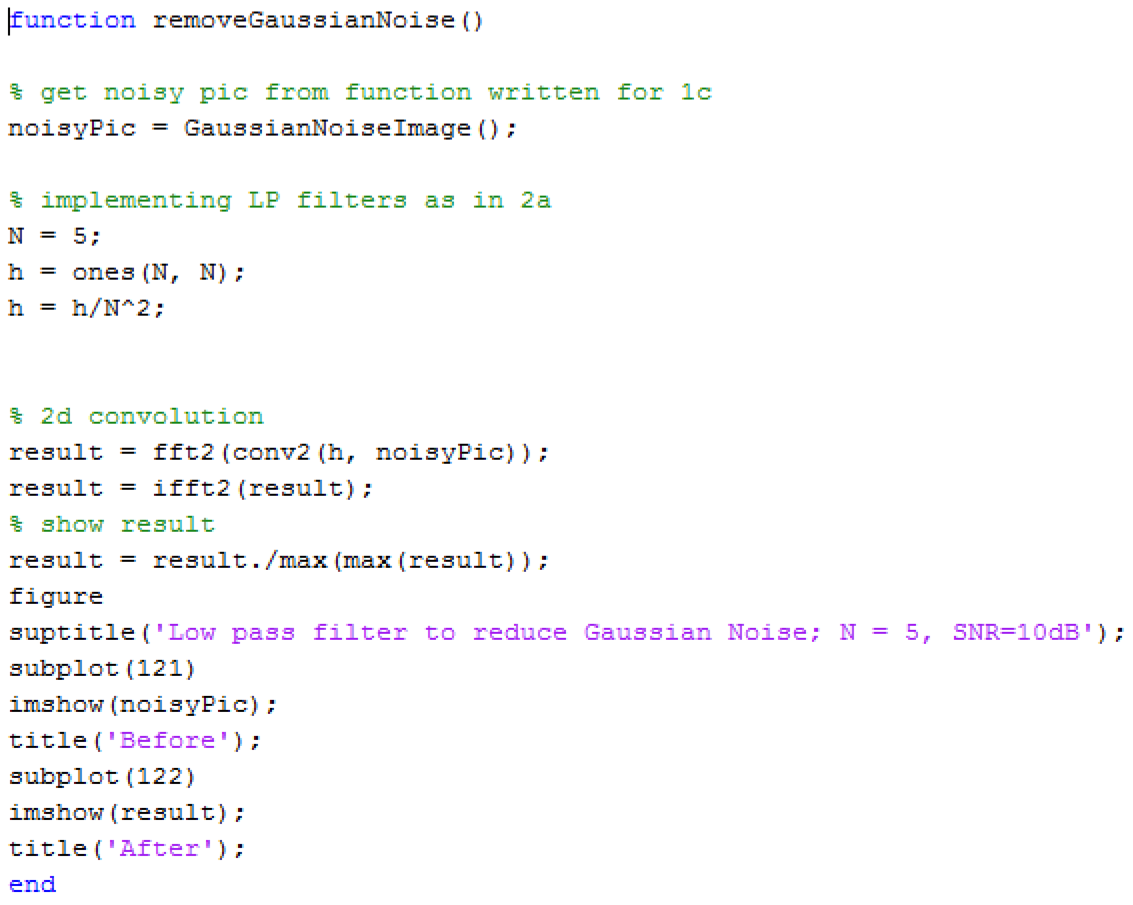
```matlab
function noisyPic = ImpulsiveNoiseImage()

img = imread('Baboon__grey_scale.jpg');
img = rgb2gray(img);
img = cast(img, 'double');
%img = img./max(max(img));
[m, n] = size(img);
%imshow(img)
impulseProb = 10;
noise = (rand(m, n) < impulseProb/100)*max(max(img))/2;

Psig = 0;
Pnoise = 0;

for row = 1 : m
    for col = 1 : n
        Psig = Psig + img(row, col)^2;
    end
end

for row = 1 : m
    for col = 1 : n
        Pnoise = Pnoise + noise(row, col)^2;
    end
end

SNR = 10*log10(Psig/Pnoise)
result = img + noise;
result = result./max(max(result));
noisyPic = result;
end
```

Figure 39: Impulsive noise corruption



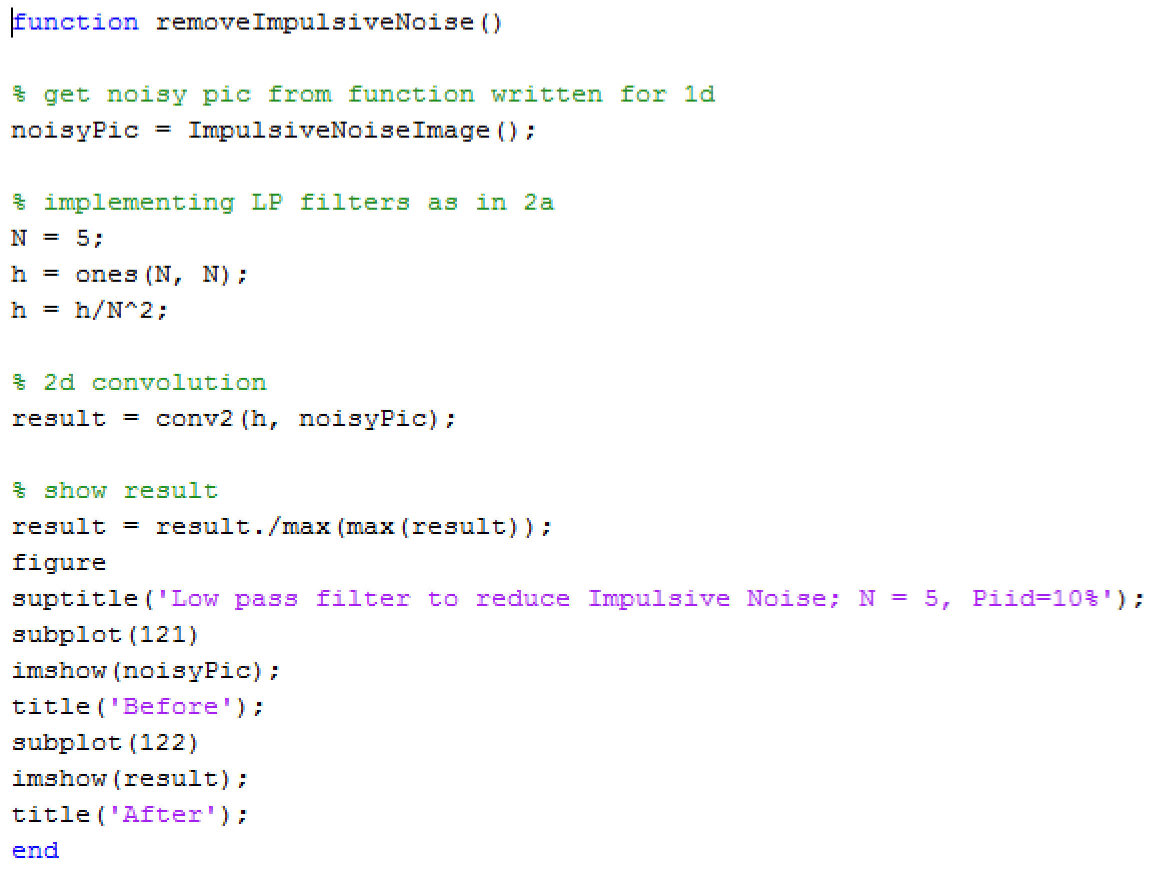
```
function BlurImage()

img = imread('Baboon__grey_scale.jpg');
img = rgb2gray(img);
img = cast(img, 'double');
img = img./max(max(img));

[n, m] = size(img);
% N defines blurring scale
N = 5;

% Make the impulse response. Its an m x n matrix with values represented by
% 1/N^2. h vector should only have 1st row, col and diagnol as values, rest
% are 0

h = ones(N, N);
h = h/N^2;

% 2d convolution
result = conv2(h, img);

% show result
result = result./max(max(result));
imshow(result);
end
```

Figure 40: Blur Image



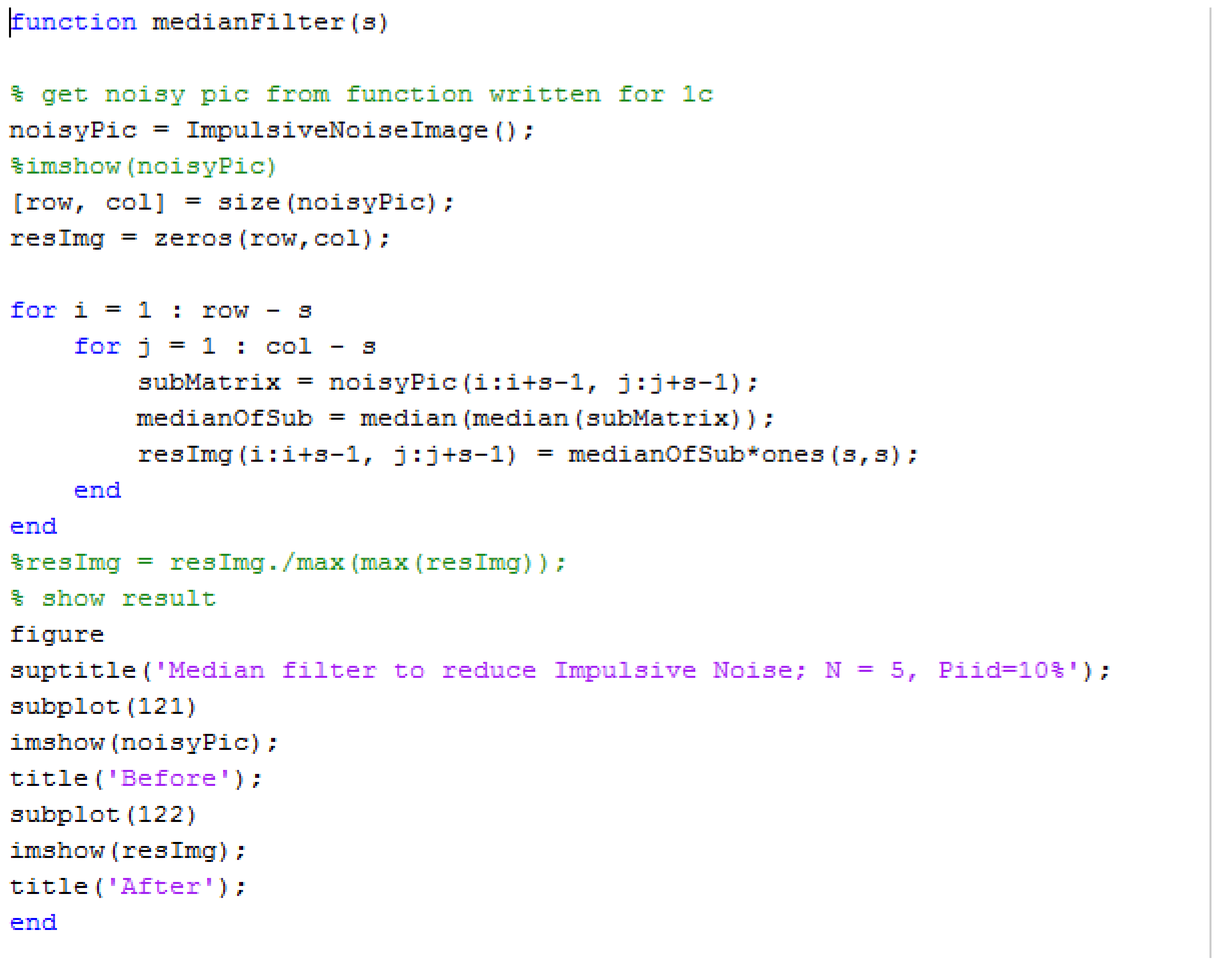
```matlab
function removeGaussianNoise()

% get noisy pic from function written for 1c
noisyPic = GaussianNoiseImage();

% implementing LP filters as in 2a
N = 5;
h = ones(N, N);
h = h/N^2;


% 2d convolution
result = fft2(conv2(h, noisyPic));
result = ifft2(result);
% show result
result = result./max(max(result));
figure
suptitle('Low pass filter to reduce Gaussian Noise; N = 5, SNR=10dB');
subplot(121)
imshow(noisyPic);
title('Before');
subplot(122)
imshow(result);
title('After');
end
```

Figure 41: Remove Gaussian noise from image using low pass filter



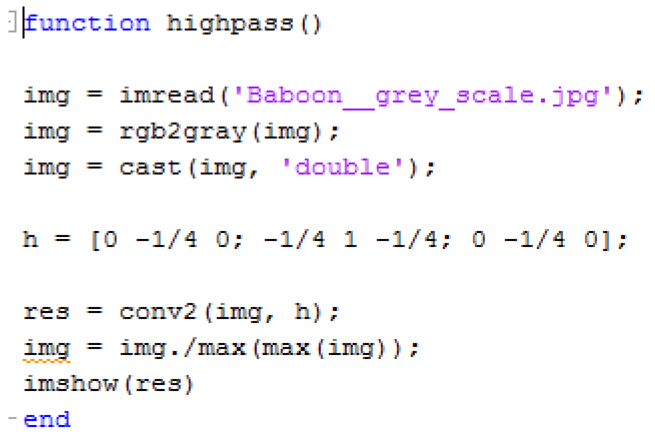
```
function removeImpulsiveNoise()

% get noisy pic from function written for 1d
noisyPic = ImpulsiveNoiseImage();

% implementing LP filters as in 2a
N = 5;
h = ones(N, N);
h = h/N^2;

% 2d convolution
result = conv2(h, noisyPic);

% show result
result = result./max(max(result));
figure
suptitle('Low pass filter to reduce Impulsive Noise; N = 5, Piid=10%');
subplot(121)
imshow(noisyPic);
title('Before');
subplot(122)
imshow(result);
title('After');
end
```

Figure 42: Remove impulsive noise from image using low pass filter



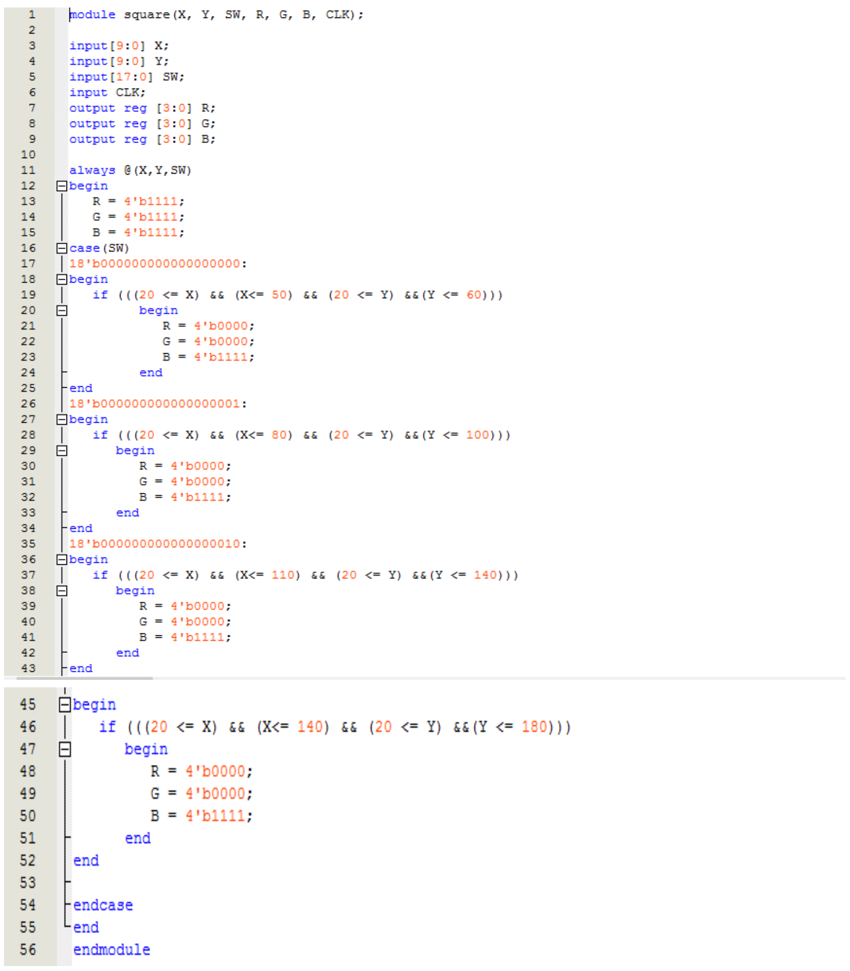
```
function medianFilter(s)

% get noisy pic from function written for 1c
noisyPic = ImpulsiveNoiseImage();
%imshow(noisyPic)
[row, col] = size(noisyPic);
resImg = zeros(row,col);

for i = 1 : row - s
    for j = 1 : col - s
        subMatrix = noisyPic(i:i+s-1, j:j+s-1);
        medianOfSub = median(median(subMatrix));
        resImg(i:i+s-1, j:j+s-1) = medianOfSub*ones(s,s);
    end
end
%resImg = resImg./max(max(resImg));
% show result
figure
suptitle('Median filter to reduce Impulsive Noise; N = 5, Piid=10%');
subplot(121)
imshow(noisyPic);
title('Before');
subplot(122)
imshow(resImg);
title('After');
end
```

Figure 43: Remove Impulsive noise using median pass



```
function highpass()

img = imread('Baboon__grey_scale.jpg');
img = rgb2gray(img);
img = cast(img, 'double');

h = [0 -1/4 0; -1/4 1 -1/4; 0 -1/4 0];

res = conv2(img, h);
img = img./max(max(img));
imshow(res)
end
```

Figure 44: High pass on image



```
module square(X, Y, SW, R, G, B, CLK);

input[9:0] X;
input[9:0] Y;
input[17:0] SW;
input CLK;
output reg [3:0] R;
output reg [3:0] G;
output reg [3:0] B;

always @(X,Y,SW)
begin
   R = 4'b1111;
   G = 4'b1111;
   B = 4'b1111;
case(SW)
18'b000000000000000000:
begin
   if (((20 <= X) && (X<= 50) && (20 <= Y) &&(Y <= 60)))
         begin
            R = 4'b0000;
            G = 4'b0000;
            B = 4'b1111;
         end
end
18'b000000000000000001:
begin
   if (((20 <= X) && (X<= 80) && (20 <= Y) &&(Y <= 100)))
      begin
         R = 4'b0000;
         G = 4'b0000;
         B = 4'b1111;
      end
end
18'b000000000000000010:
begin
   if (((20 <= X) && (X<= 110) && (20 <= Y) &&(Y <= 140)))
      begin
         R = 4'b0000;
         G = 4'b0000;
         B = 4'b1111;
      end
end
```

```
begin
   if (((20 <= X) && (X<= 140) && (20 <= Y) &&(Y <= 180)))
      begin
         R = 4'b0000;
         G = 4'b0000;
         B = 4'b1111;
      end
end

endcase
end
endmodule
```

Figure 45: Verilog script to produce square

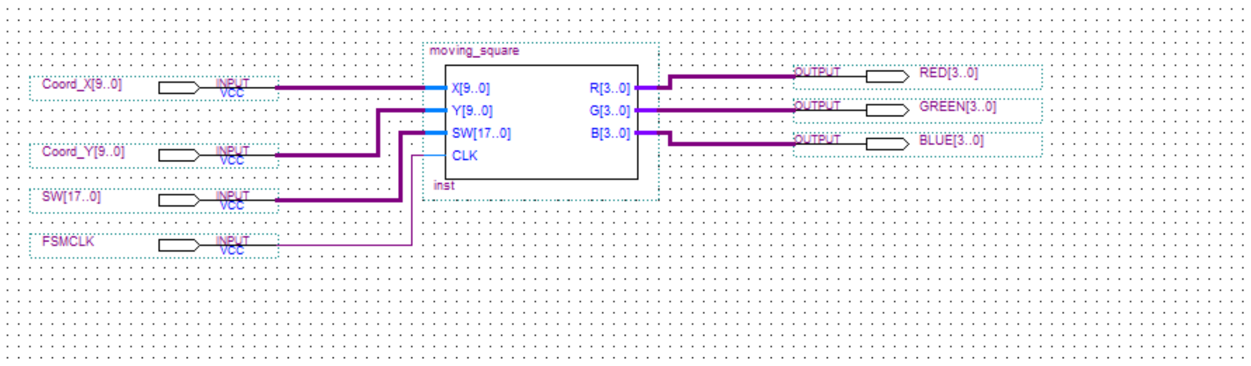
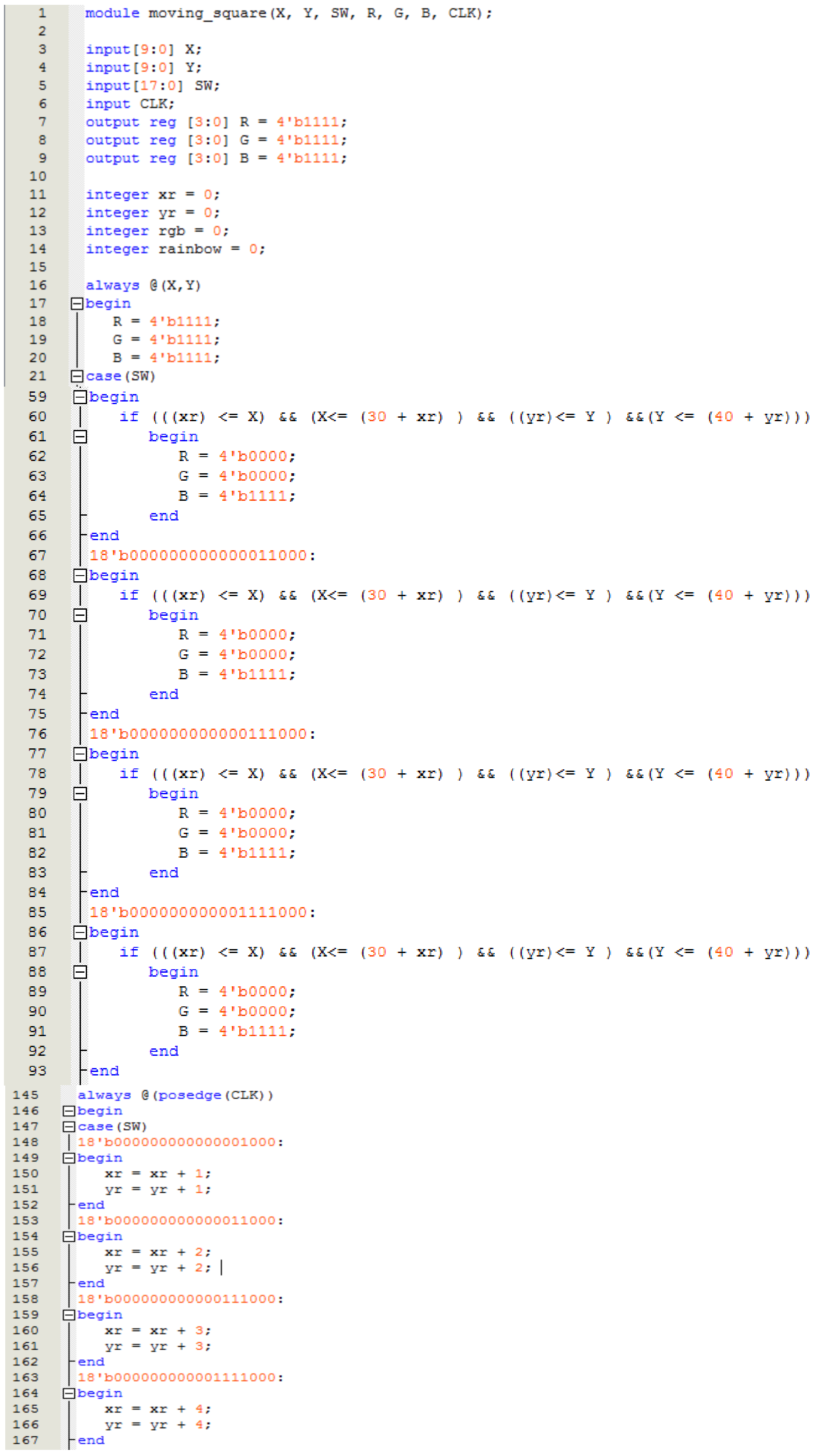
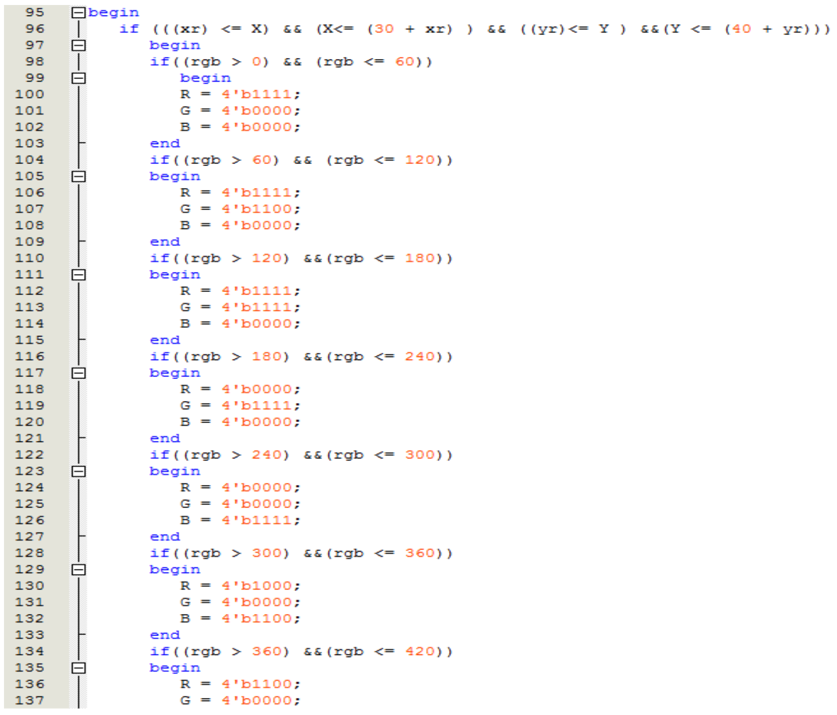


Figure 46: Block Diagram for the moving_square module

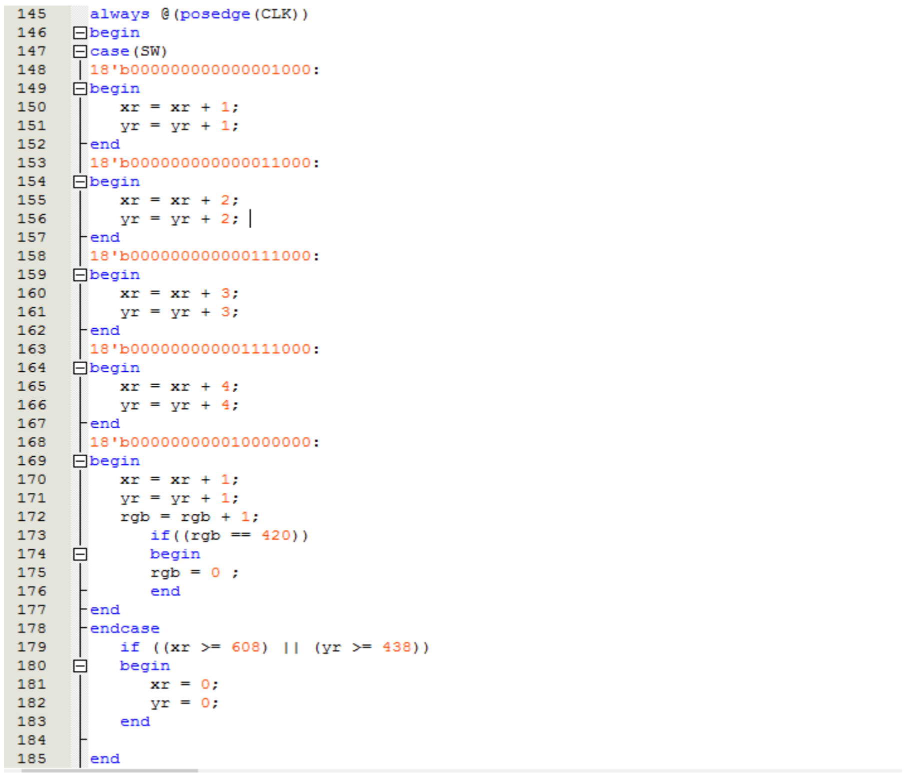


```
module moving_square(X, Y, SW, R, G, B, CLK);

input[9:0] X;
input[9:0] Y;
input[17:0] SW;
input CLK;
output reg [3:0] R = 4'b1111;
output reg [3:0] G = 4'b1111;
output reg [3:0] B = 4'b1111;

integer xr = 0;
integer yr = 0;
integer rgb = 0;
integer rainbow = 0;

always @(X,Y)
begin
   R = 4'b1111;
   G = 4'b1111;
   B = 4'b1111;
case(SW)
begin
    if (((xr) <= X) && (X<= (30 + xr) ) && ((yr)<= Y ) &&(Y <= (40 + yr)))
        begin
            R = 4'b0000;
            G = 4'b0000;
            B = 4'b1111;
        end
end
18'b000000000000011000:
begin
    if (((xr) <= X) && (X<= (30 + xr) ) && ((yr)<= Y ) &&(Y <= (40 + yr)))
        begin
            R = 4'b0000;
            G = 4'b0000;
            B = 4'b1111;
        end
end
18'b000000000000111000:
begin
    if (((xr) <= X) && (X<= (30 + xr) ) && ((yr)<= Y ) &&(Y <= (40 + yr)))
        begin
            R = 4'b0000;
            G = 4'b0000;
            B = 4'b1111;
        end
end
18'b000000000001111000:
begin
    if (((xr) <= X) && (X<= (30 + xr) ) && ((yr)<= Y ) &&(Y <= (40 + yr)))
        begin
            R = 4'b0000;
            G = 4'b0000;
            B = 4'b1111;
        end
end
always @(posedge(CLK))
begin
case(SW)
18'b000000000000001000:
begin
   xr = xr + 1;
   yr = yr + 1;
end
18'b000000000000011000:
begin
   xr = xr + 2;
   yr = yr + 2;
end
18'b000000000000111000:
begin
   xr = xr + 3;
   yr = yr + 3;
end
18'b000000000001111000:
begin
   xr = xr + 4;
   yr = yr + 4;
end
```

Figure 47: Verilog code implementation for moving square - 1



```
begin
    if (((xr) <= X) && (X<= (30 + xr) ) && ((yr)<= Y ) &&(Y <= (40 + yr)))
        begin
        if((rgb > 0) && (rgb <= 60))
            begin
            R = 4'b1111;
            G = 4'b0000;
            B = 4'b0000;
        end
        if((rgb > 60) && (rgb <= 120))
        begin
            R = 4'b1111;
            G = 4'b1100;
            B = 4'b0000;
        end
        if((rgb > 120) &&(rgb <= 180))
        begin
            R = 4'b1111;
            G = 4'b1111;
            B = 4'b0000;
        end
        if((rgb > 180) &&(rgb <= 240))
        begin
            R = 4'b0000;
            G = 4'b1111;
            B = 4'b0000;
        end
        if((rgb > 240) &&(rgb <= 300))
        begin
            R = 4'b0000;
            G = 4'b0000;
            B = 4'b1111;
        end
        if((rgb > 300) &&(rgb <= 360))
        begin
            R = 4'b1000;
            G = 4'b0000;
            B = 4'b1100;
        end
        if((rgb > 360) &&(rgb <= 420))
        begin
            R = 4'b1100;
            G = 4'b0000;
```



```
always @(posedge(CLK))
begin
case(SW)
18'b000000000000001000:
begin
    xr = xr + 1;
    yr = yr + 1;
end
18'b000000000000011000:
begin
    xr = xr + 2;
    yr = yr + 2;
end
18'b000000000000111000:
begin
    xr = xr + 3;
    yr = yr + 3;
end
18'b000000000001111000:
begin
    xr = xr + 4;
    yr = yr + 4;
end
18'b000000000010000000:
begin
    xr = xr + 1;
    yr = yr + 1;
    rgb = rgb + 1;
        if((rgb == 420))
        begin
        rgb = 0 ;
        end
end
endcase
    if ((xr >= 608) || (yr >= 438))
    begin
        xr = 0;
        yr = 0;
    end

end
```

Figure 48: Verilog code implementation for moving square - 2