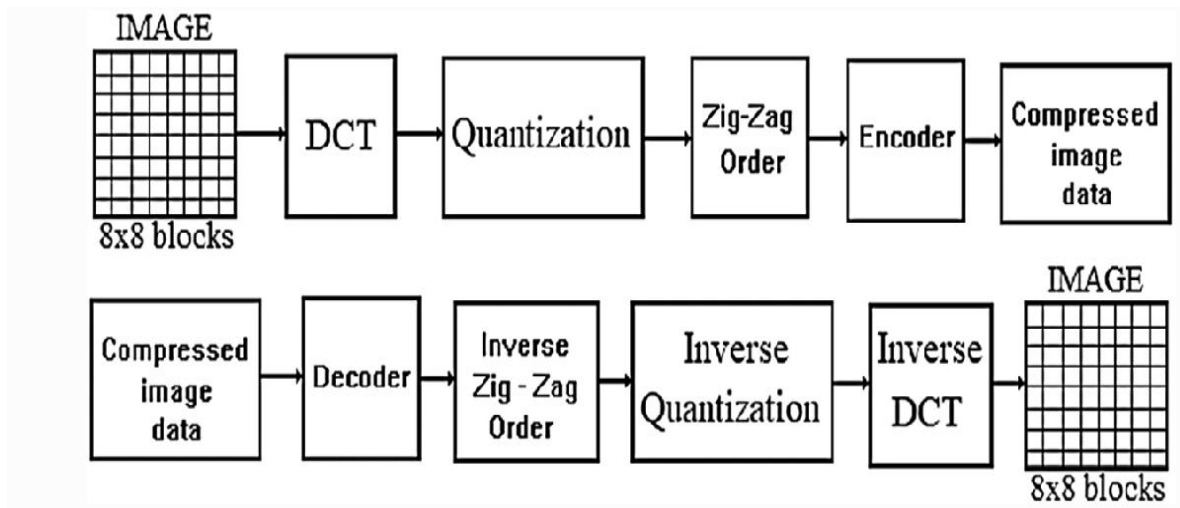


JPEG Compression

JPEG stands for Joint Photographic Experts Group. It is the first international standard for image compression. JPEG can support 2^{24} colors. It can do various level of compression which make it ideal for the web graphics. That's why it is hugely used in digital camera. JPEG is lossy compression technique. Although it is lossy technique if an image file format in JPEG, the resolution of the image remains high. This is the major advantages of JPEG. JPEG compression technique involves many steps. The Major Steps in JPEG Coding involve:

- An RGB to YCbCr color space conversion;
- Actual image is segmented into blocks of 8 x 8 size.
- The pixel values within every block ranging from [-128 to 127] but pixel values of a black and white image range from [0-255] that's why, every block is shifted from [0-255] to [-128 to 127].
- The DCT works from left side to right, top to bottom thereby it is implemented over every block.
- Quantization will compress every block.
- Quantized matrix is encoded.
- Reverse process will help in reconstruction image. This process uses the IDCT (Inverse Discrete Cosine Transform) .

The below images describe the whole procedure:



First of all, read the original RGB image and then Plot it.

```
img = imread('ronaldo.jpg');
r=size(img,1);
c=size(img,2);
% Displaying the image
figure(1);
imshow(img);
title('Original RGB image')
```

Original RGB image



Converting the RGB to YCbCr color space.

JPEG never uses RGB. It converts the RGB image to a different color space, YCbCr, to separate the luminance channel (Y) from the chrominance channels (CbCr).

```
%Convert the image into YCbCr color space
YCBCR=rgb2ycbcr(img);
figure(2)
imshow(YCBCR);
title('Image in YCbCr Color Space');
```

Image in YCbCr Color Space

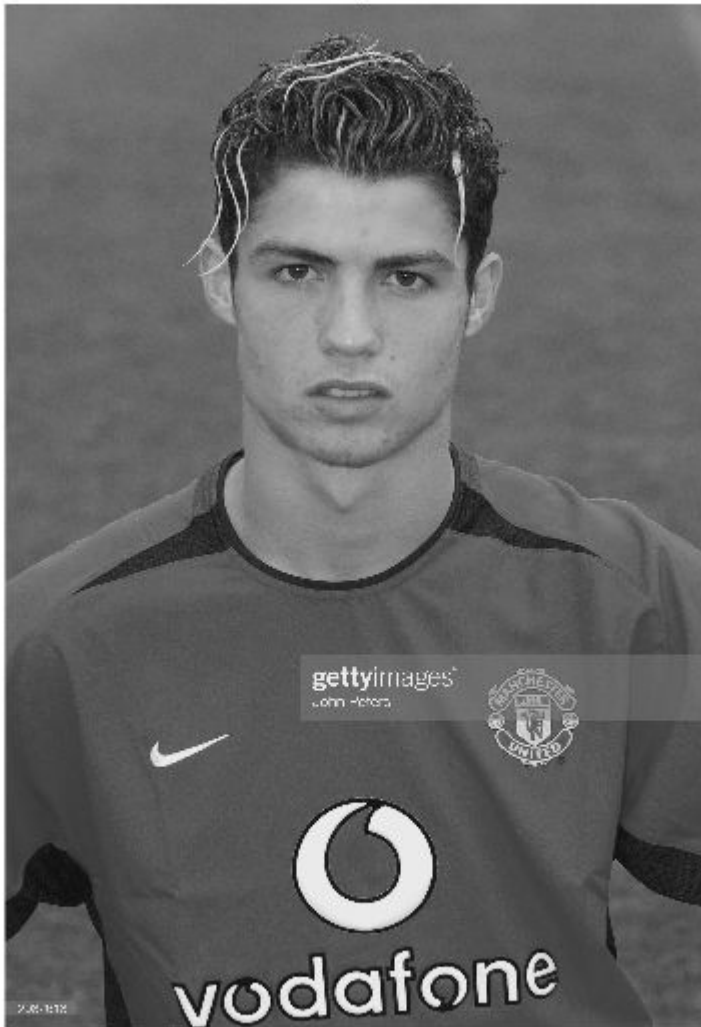


Display Y, Cb and Cr components separately.

```
Y_component=ycbcr(:,:,1); % to get Y component  
Cb_component=ycbcr(:,:,2); % to get Cb component  
Cr_component=ycbcr(:,:,3); % to get Cr component
```

```
figure(3);  
imshow(Y_component);  
title('Y component');
```

Y component



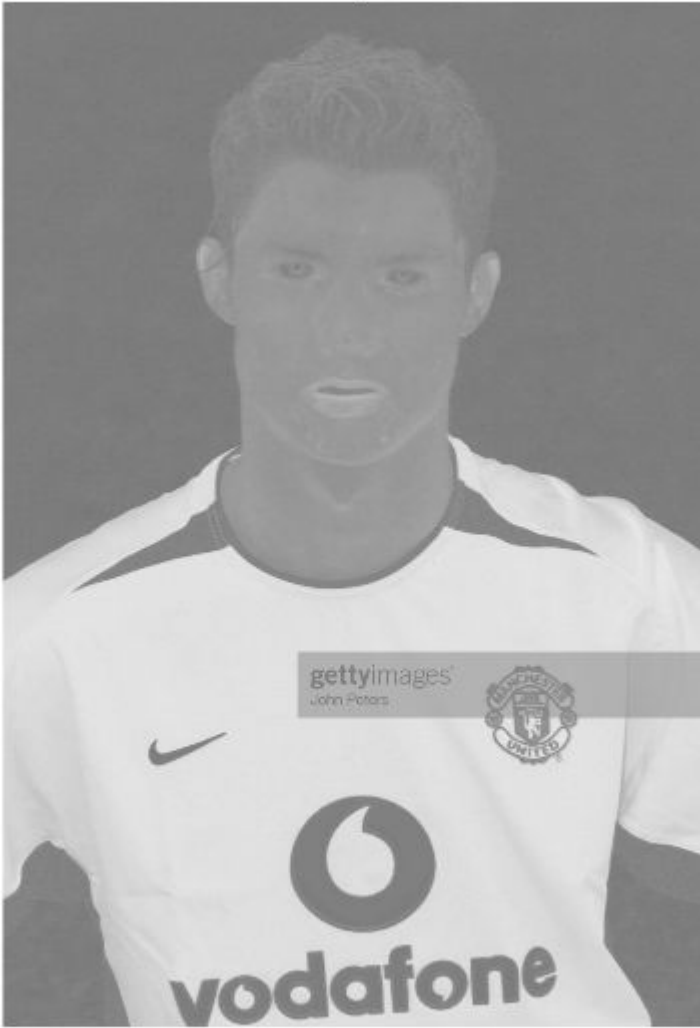
```
figure(4);  
imshow(Cb_component);  
title('Cb component');
```

Cb component



```
figure(5);  
imshow(Cr_component);  
title('Cr component');
```

Cr component



Chroma Subsampling

Human eye is more sensitive in brightness than color, that's why we can safely down-sample chroma components. This strategy is called "Chroma subsampling". We can take advantage of an important property of the human visual system. We are more sensitive to the luminance component than the chrominance component. That is, the human eye is more sensitive to brightness than colour. So, it makes sense that we can drop some of the chrominance components without affecting the quality of perception.

The subsampling format is expressed as a ratio $a:b:c$. The first part represents the luminance component of the frame (Y) and the other two represent the chrominance components (Cb and Cr). So, a subsampling format of 4:4:4 means that for every 4 luminance components, we have 4 Cb and 4 Cr components. This means that basically, there is no compression taking place. This is not really useful. I have used 4:2:0 process for better result. 4:2:0 means we transmit a Y value for each pixel but we downsample the Cr and Cb by 2.

```
%4:2:0 subsample Cb and Cr component  
Cb_subsample = Cb_component(1:2:r,1:2:c);  
Cr_subsample = Cr_component(1:2:r,1:2:c);
```

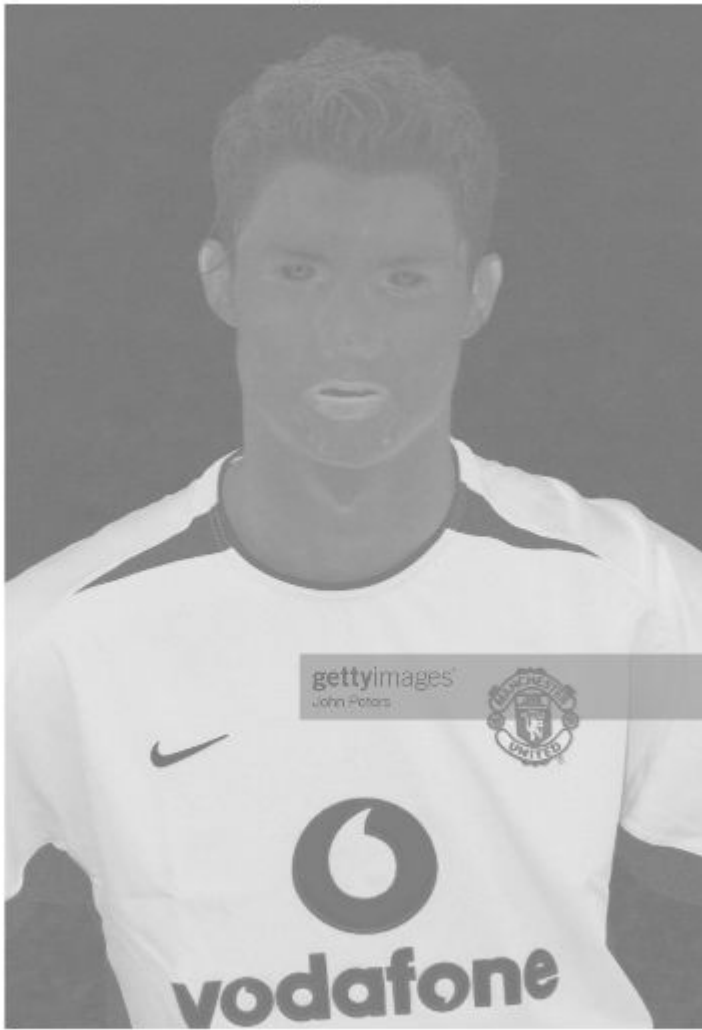
```
figure(6);  
imshow(Cb_subsample);  
title('subsampled Cb element'); % subsampled Cb element is displayed
```

subsampled Cb element



```
figure(7);  
imshow(Cr_subsample);  
title('subsampled Cr element'); % subsampled Cr element is displayed
```

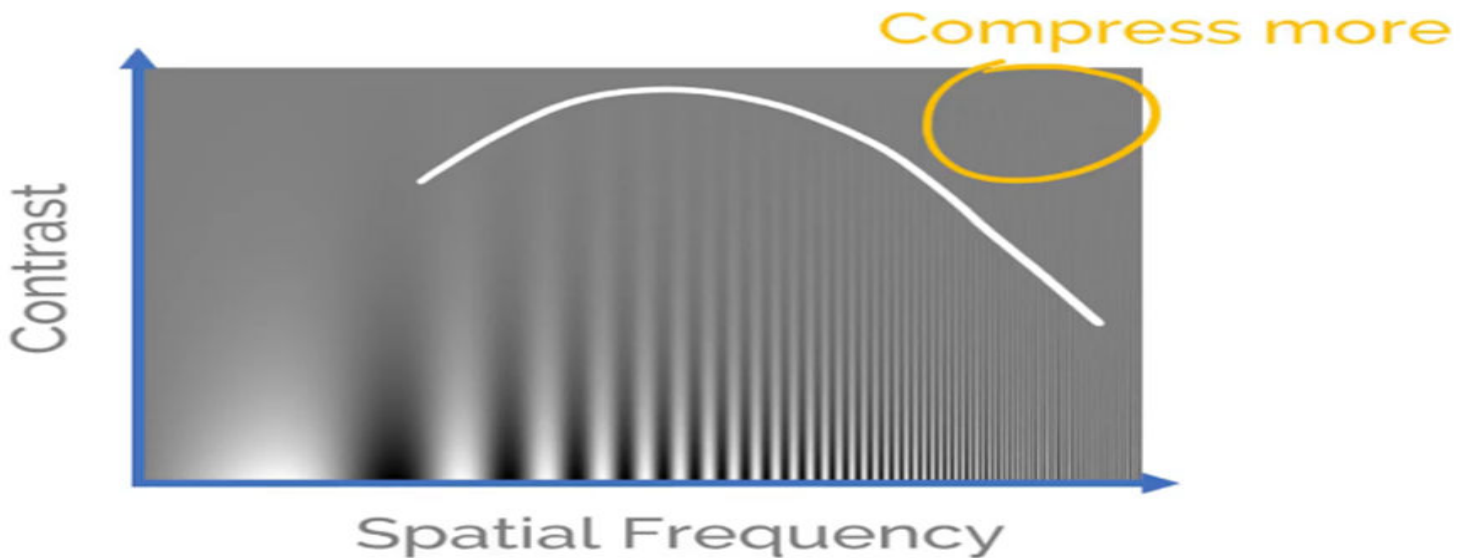
subsampling Cr element



Block Based DCT

Let's take a look at another aspect of the human eye: "Frequency-dependent contrast sensitivity." The bars under the curve are more noticeable than the rest in the diagram below. This allows for slight compression in

the upper right corner's that less noticeable high frequencies.



DCT is a method that expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. Because of a variation in their boundary behavior (The behavior of a wave upon reaching the end of a medium is referred to as boundary behavior. When one medium ends, another medium begins, the interface of the two media is referred to as the boundary and the behavior of a wave at that boundary is described as its boundary behavior.), cosine functions are utilized instead of sine functions for compression. The brightness of an image, unlike sine, does not require zero values on the boundary. As a result, a linear combination of sines is difficult to mimic such a signal.

The entire image we want to compress is divided into sub-images each of which comprises of 8x8 pixels. The question is that, why JPEG compression processes image by 8x8 blocks instead of applying Discrete Cosine Transform to the whole image? There are more than one answer for this question:

1. Any matrices of sizes greater than 8 X 8 are harder to do mathematical operations or not supported by hardware or take longer time.
2. Any matrices of sizes less than 8 X 8 don't have enough information to continue along with the pipeline. It results in bad quality of the compressed image.

so, 8x8 block of each component (Y, Cb, Cr) is converted to a frequency-domain representation. Before computing the DCT of the 8x8 block, its values are shifted from a positive range to one centered around zero. For an 8-bit image, each entry in the original block falls in the range [0, 255]. The midpoint of the range that is 128 is subtracted from each entry to produce a data range that is centered around zero, so that the modified range is [-128, 127]. We call this **shifted-block**.

This step reduces the dynamic range in the DCT processing. The question is what is dynamic range? Answer: DYNAMIC RANGE is the range of tonal difference between the lightest light and darkest dark of an image. The higher the dynamic range, the more potential shades can be represented in the image. For high-performance JPEG image compression, we have to use the low dynamic range discrete cosine transform.

We can describe the whole above thing by picture representation:

Original Sub-image

64	60	57	56	48	47	47	43
61	58	53	52	48	49	52	53
67	60	53	53	49	47	48	54
68	61	63	63	62	65	65	64
71	61	70	63	69	74	88	88
83	94	102	105	107	111	110	115
95	108	108	124	122	130	128	128
107	118	125	134	137	142	141	137



Shifted sub-image

-64	-68	-71	-72	-80	-81	-81	-85
-67	-70	-75	-76	-80	-79	-76	-75
-61	-68	-75	-75	-79	-81	-80	-74
-60	-67	-65	-65	-66	-63	-63	-64
-57	-67	-58	-65	-59	-54	-40	-40
-45	-36	-26	-23	-21	-17	-18	-13
-33	-20	-20	-4	-6	2	0	0
-21	-10	-3	6	9	14	13	9



After DCT

-376	-23	1	-2.5	-0.3	4	0.2	-2.6
-224	53	20	3.4	5	3	0.6	2.3
68	3.3	-14	-0.3	-2.8	-1.9	-4.7	-6.2
2.3	-8.9	-1.5	-3.8	-2.5	1.2	1.4	1.9
-8.4	1.2	1.9	3.3	-2.1	5	1.8	5.3
4.5	7.3	-7.4	1.9	1.3	-0.7	-1.5	-6
6.4	6.8	-3.2	-2.6	1.3	-2.1	1.7	1
-16	0.1	9	0.8	1.8	1.7	-1	1

The last table is the 8x8 table of coefficients. In this frequency representation we can easily compress the frequencies which are less visible to us that is upper left corner by dividing these numbers with some constants and then quantizing them (next step).

```
DCT_y = @dct2; % to obtain DCT coefficients for Y component
DCT_coef_Y = blkproc(Y_component, [8 8], DCT_y); % block processing for 8*8blocks where
                                                    % the DCT function is performed
DCT_coef_Y = fix(DCT_coef_Y); % each element is rounded to the nearest integer towards zero
```

```
figure(8);
imshow(DCT_coef_Y);
title('DCT of Y element'); % subsampled Cb element is displayed
```

DCT of Y element



```
DCT_Cb = @dct2; %dct2 command is used for doing DCT for Cb element
DCT_coef_Cb = blkproc(Cb_subsample, [8 8], DCT_Cb); % block processing for
                                                    % 8*8blocks the DCT function is performed
DCT_coef_Cb = fix(DCT_coef_Cb);%each element is rounded to the nearest integer towards zero
```

```
figure(9);
imshow(DCT_coef_Cb);
title('DCT of Cb element'); % subsampled Cb element is displayed
```

DCT of Cb element



```
DCT_Cr = @dct2; %performing the DCT using the command dct2 for Cr element
DCT_coef_Cr = blkproc (Cr_subsample, [8 8], DCT_Cr); %performing block processing operation
                                                    % for 8*8blocks where in interior the DCT
                                                    %function is performed
DCT_coef_Cr = fix(DCT_coef_Cr);%rounds each element to nearest intezer toward zero
```

```
figure(10);
imshow(DCT_coef_Cr);
title('DCT of Cr element'); % subsampled Cr element is displayed
```

DCT of Cr element



```
% DCT coefficient matrix as well as image of the DCT transformed image blocks  
% of the 1st and 2nd block in the 5th row from top for the luminance element.  
Y_DCT_block1_row5 = zeros(8,8); %to extract the 5th row first block 8*8matrix  
Y_DCT_block1_row5(1:8,1:8) = DCT_coef_Y(33:40,9:16);
```

```
Y_DCT_block2_row5 = zeros(8,8); %to extract the second block 8*8 matrix  
Y_DCT_block2_row5=DCT_coef_Y(33:40,1:8);
```

Quantization

We will now quantize the coefficient table we obtained using DCT. This is the real lossy part of the process. We know that high frequency part can be eliminated without much loss in the look of the image. So we now prepare a 8x8 quantization table. This table will have have very small values at the top-left part and very high values

towards the bottom-right part. Every value in the coefficient table is divided by the corresponding value in the quantization table and rounded to the nearest integer. Now, because of the high divisor in the bottom-right part, the divided values here become zero - thus eliminating the high frequency component. This Quantization table is up to the encoder, and therefore the table is kept in the image header so the image can be later decoded.

We can describe the above talk by using the figure:

Quantization Table								After Quantization							
16	11	10	16	24	40	51	61	-24	-23	0	0	0	0	0	0
12	12	14	19	26	58	60	55	-19	4	1	0	0	0	0	0
14	13	16	24	40	57	69	56	5	0	1	0	0	0	0	0
14	17	22	29	51	87	80	62	0	0	0	0	0	0	0	0
18	22	37	56	68	109	103	77	0	0	0	0	0	0	0	0
24	35	55	64	81	104	113	92	0	0	0	0	0	0	0	0
49	64	78	87	103	121	120	101	0	0	0	0	0	0	0	0
72	92	95	98	112	100	103	99	0	0	0	0	0	0	0	0

The left one is standard JPEG Quantization table. The right one is our sub-image after quantization which I got by dividing each value in our coefficient table with the corresponding value in the quantization table. Notice that in the quantized output table, all values except the top-left 3x3 block are all zeroes. These are the high frequency data we eliminated. JPEG's claim to fame is that with just these 9 values we can get almost the same image back.

```
Y_Quant_matrix = [16 11 10 16 24 40 51 61; 12 12 14 19 26 58 60 55; 14 13 16 24 40 57 69 56; 14 17 22 29 51 87 80 62; 18 22 37 56 68 109 103 77; 24 35 55 64 81 104 113 92; 49 64 78 87 103 121 120 101; 72 92 95 98 112 100 103 99];
```

```
% Quantization is done for the luminance element (Y) by using the quantization matrix
quant_y = @(DCT_Yelement) round(DCT_Yelement ./ Y_Quant_matrix);
%performing block processing operation for 8*8blocks
quantized_Y = blkproc(DCT_coef_Y,[8 8],quant_y);
```

```
Cr_Quant_matrix = [17 18 24 47 99 99 99 99; 18 21 26 66 99 99 99 99; 24 26 56 99 99 99 99 99; 47 66 99 99 99 99 99 99; 99 99 99 99 99 99 99 99; 99 99 99 99 99 99 99 99; 99 99 99 99 99 99 99 99; 99 99 99 99 99 99 99 99];
```

```
%Quantization is done for the chrominance element (Cb)by using the quantization matrix
quant_Cb = @(DCT_Cbelement) round(DCT_Cbelement ./ Cr_Quant_matrix);
%performing block processing operation for 8*8blocks
quantized_Cb = blkproc(DCT_coef_Cb,[8 8],quant_Cb);
```

```
%Quantization is done for the chrominance element (Cr) by using the quantization matrix
quant_Cr = @(DCT_Crelement) round(DCT_Crelement ./ Cr_Quant_matrix);
%performing block processing operation for 8*8blocks
```


```
quantized_Cr = blkproc(DCT_coef_Cr,[8 8],quant_Cr);
```

```
%extracting the fifth row 8*8 block of quantized matrix
Y_DCT_block1_row5 = zeros(8,8);
Y_DCT_block1_row5(1:8,1:8) = quantized_Y(33:40,1:8)
```

```
Y_DCT_block1_row5 = 8×8
61      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
```

```
figure(11);
imshow(Y_DCT_block1_row5);
title('block 1 row 5');
```

block 1 row 5



```
%extracting the fifth 8*8 block of quantized matrix
Y_DCT_block2_row5 = zeros(8,8);
Y_DCT_block2_row5(1:8,1:8) = quantized_Y(25:32,33:40)
```

```
Y_DCT_block2_row5 = 8×8
62      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
```

```
figure(12);
imshow(Y_DCT_block2_row5);
title('block 2 row 5');
```

block 2 row 5



```
% (1,1) element of forth block quantized matrix
DCcoeff_block1_row5 = Y_DCT_block1_row5(1,1);
% (1,1) element of fifth block quantized matrix
DC_coeff_block2_row5 = Y_DCT_block2_row5(1,1);
```

```
%displaying DC coefficient
```

```
message1 = sprintf('DCT DC coefficient of the first block in the 5th Row of the quantized image is %f',
```

```
message1 =
    'DCT DC coefficient of the first block in the 5th Row of the quantized image is 61.00
    '
```

```
%displaying DC coefficient
```

```
message2 = sprintf('DCT DC coefficient for the second block in the 5th Row of the quantized image is %f',
```

```
message2 =
    'DCT DC coefficient for the second block in the 5th Row of the quantized image is 62.00
    '
```

ZigZag Scanning:

We have now got the compressed output as a 2D array. We also know that a lot of them are zeroes. So, we will find a better way to store the sub-image than store its as a 2D array.

We will store the values in a zigzag order. So the data will be:

-24, -23, 19, 5, 4, 0, 0, 1, 0, 0, 0, 0, 1 followed by 53 zeroes.

After Quantization

-24	-23	0	0	0	0	0	0
-19	4	1	0	0	0	0	0
5	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

As we saw above, after quantization, we end up with lot of zeros in high frequencies. We can store this information more efficiently by rearranging these in zig-zag order from top-left to bottom-right. This way we can group the zeros together.

```
% zigzag scan for first block in the fifth row
% use four variables, one to switch between rows and columns , other to
% move across the diagonals and the rest two to get the x and y positions

temp1=0;
size_block1_row5=size(Y_DCT_block1_row5);
%Size of the first block computed by multiplying columns and rows
sum_block1=size_block1_row5(2)*size_block1_row5(1);
for i1=2:sum_block1
    j1=rem(i1,2); % to check whether the value is even or odd
    for counti1=1:size_block1_row5(1)
        for countj1=1:size_block1_row5(2)
            if((counti1+countj1)==i1)
                temp1=temp1+1;
                if(j1==0)
                    zigzagblock5_1(temp1)= Y_DCT_block1_row5(countj1,i1-countj1);
                else
                    zigzagblock5_1(temp1)= Y_DCT_block1_row5(i1-countj1,countj1);
                end
            end
        end
    end
end
Y_DCT_block1_row5;
disp(zigzagblock5_1);
```

Columns 1 through 29

61 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Columns 30 through 58

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Columns 59 through 64

0 0 0 0 0 0

```
% zigzagscan for second block in the fifth row
```

```
temp2=0;
size_block2_row5=size(Y_DCT_block2_row5);
sum_block2=size_block2_row5(2)*size_block2_row5(1); % M*N value is calculated
for i2=2:sum_block2
    j2=rem(i2,2); % to check whether the value is even or odd
    for count_2i=1:size_block2_row5(1)
        for count_2j=1:size_block2_row5(2)
            if((count_2i+count_2j)==i2)
                temp2=temp2+1;
                if(j2==0)
                    zigzagblock5_2(temp2)=Y_DCT_block2_row5(count_2j,i2-count_2j);
                else
                    zigzagblock5_2(temp2)=Y_DCT_block2_row5(i2-count_2j,count_2j);
                end
            end
        end
    end
end
Y_DCT_block2_row5;
disp(zigzagblock5_2);
```

Columns 1 through 29

62 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Columns 30 through 58

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Columns 59 through 64

0 0 0 0 0 0

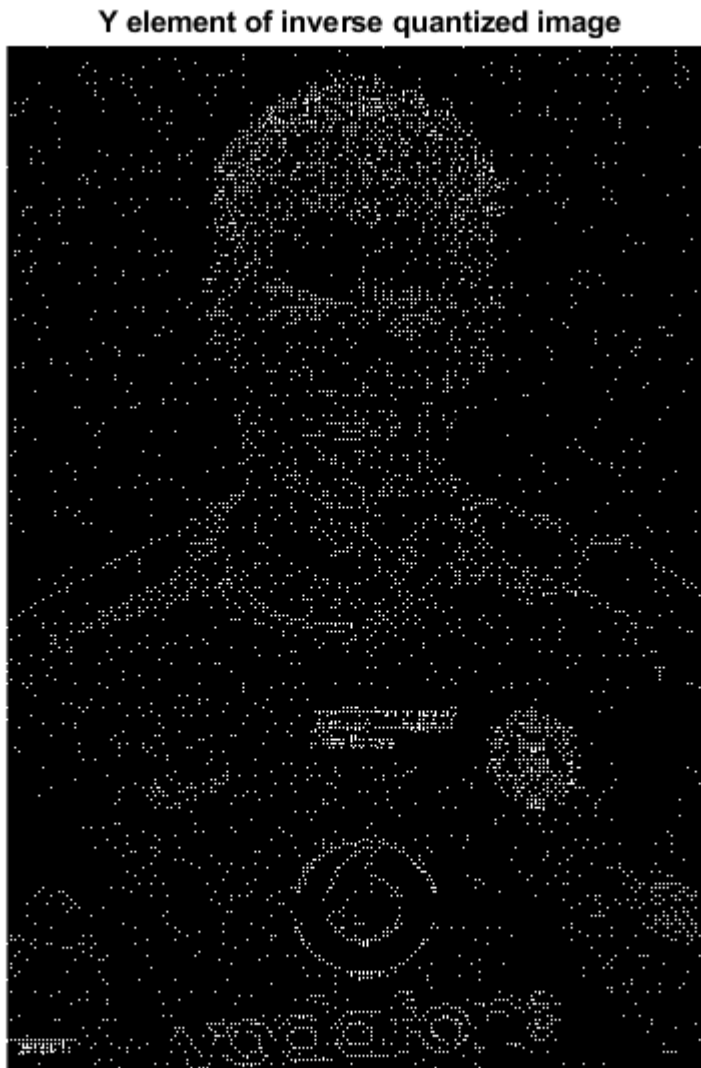
DECODER

Compute the inverse Quantized images obtained in Previous Steps.

```
% Inverse quantization for luminance element
Inv_Quant_y = @(quantized_Y) round(quantized_Y .* Y_Quant_matrix);
% block processing operation
```

```
Inverse_Quantized_Y = blkproc(quantized_Y,[8 8],Inv_Quant_y);
```

```
figure(13);  
imshow(Inverse_Quantized_Y);  
title('Y element of inverse quantized image')
```



```
% inverse quantization for Cb element  
InverseQuant_Cbelement = @(quantized_Cb) round(quantized_Cb .* Cr_Quant_matrix);  
% block processing operation  
Inverse_Quantized_Cb = blkproc(quantized_Cb,[8 8],InverseQuant_Cbelement);
```

```
figure(14);  
imshow(Inverse_Quantized_Cb);  
title('Cb element of inverse quantized image');
```

Cb element of inverse quantized image



```
figure(15);  
imshow(Inverse_Quantized_Cr),  
title('Cr element of inverse quantized image')
```

Cr element of inverse quantized image



```
% inverse quantization for Cr element
InverseQuant_Crelement = @(quantized_Crelement) round(quantized_Crelement .* Cr_Quant_matrix);
%block processing operation
Inverse_Quantized_Cr = blkproc(quantized_Cr,[8 8],InverseQuant_Crelement);
```

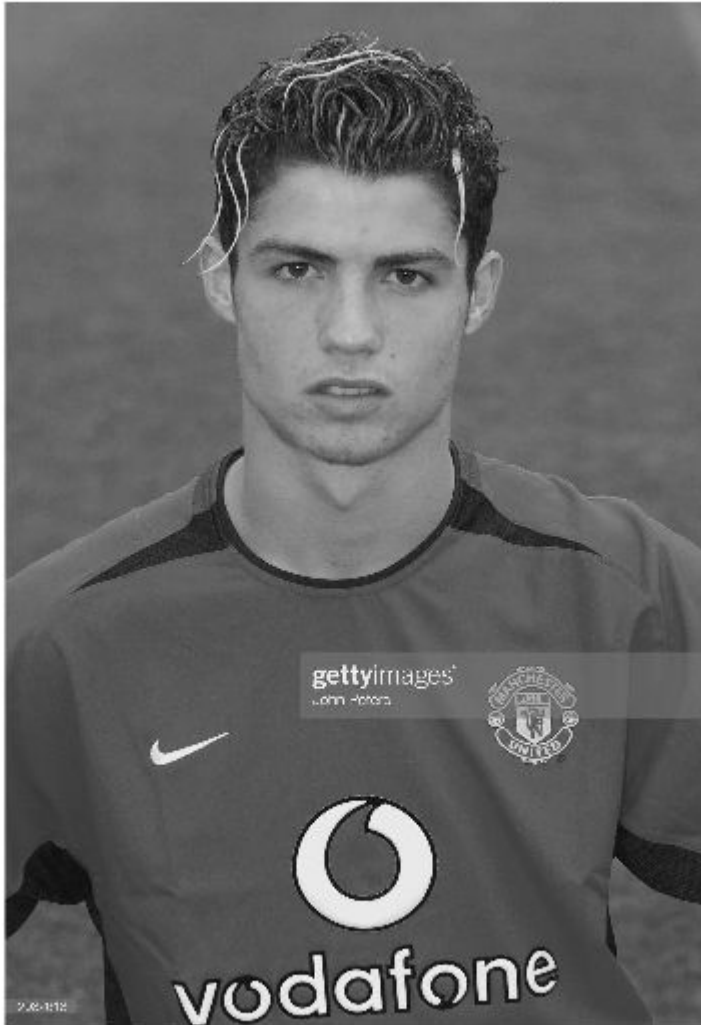
Reconstruct the image by computing Inverse DCT coefficients.

```
a_DCT = @idct2; % inverse DCT operation
% block processing operation on 8*8 Y block
IDCT_Y = blkproc(Inverse_Quantized_Y, [8 8], a_DCT);
IDCT_Y = uint8(fix(IDCT_Y));
```

```
figure(16);
imshow(IDCT_Y);
```

```
title('Y element of inverse DCT image');
```

Y element of inverse DCT image



```
b_DCT = @idct2; % inverse DCT operation  
% block processing operation on 8*8 Cb block  
IDCT_Cb = blkproc(Inverse_Quantized_Cb, [8 8], b_DCT);  
IDCT_Cb = uint8(fix(IDCT_Cb));
```

```
figure(17);  
imshow(IDCT_Cb);  
title('Cb element of inverse DCT image');
```

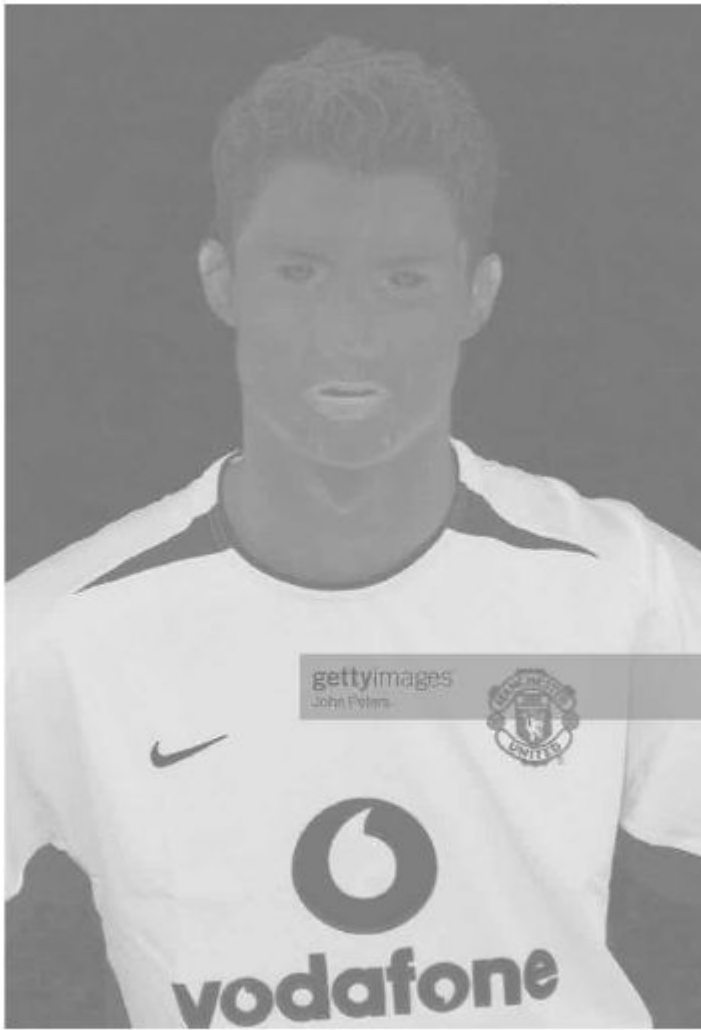
Cb element of inverse DCT image



```
c_DCT = @idct2; % inverse DCT operation
% block processing operation on 8*8 Cr block
IDCT_Cr = blkproc(Inverse_Quantized_Cr, [8 8], c_DCT);
IDCT_Cr = uint8(fix(IDCT_Cr));
```

```
figure(18);
imshow(IDCT_Cr);
title('Cr element of inverse DCT image');
```

Cr element of inverse DCT image



Upsampling using Row- Column Replication

For odd-numbered rows, the missing pixels are filled up by copying the pixels from the previous columns. Then, in order to complete the even-numbered rows, the odd-numbered rows are added to the subsequent even-numbered rows. In this process, the numbers of rows are doubled.

```
% Temporary 3 dimensional matrix with element zeros  
temp2=zeros(r,c,3);
```

```
% Cb subsampled matrix elements to the temporary matrix  
temp2(1:2:r,1:2:c,2)=IDCT_Cb(:,:);  
% Cr subsampled matrix elements to the temporary matrix  
temp2(1:2:r,1:2:c,3)=IDCT_Cr(:,:);
```



```
Cb_upsample=temp2(:,:,2);  
Cr_upsample=temp2(:,:,3);
```

```
Cb_upsample=uint16(Cb_upsample);  
Cr_upsample=uint16(Cr_upsample);
```

```
% performing upsampling using rowcolumn replication for odd number of i;  
for i=1:2:r  
    for j=2:2:c  
        Cb_upsample(i,j)=Cb_upsample(i,j-1);  
        Cr_upsample(i,j)=Cr_upsample(i,j-1);  
    end  
end
```

```
% performing upsampling using rowcolumn replication for even number of i  
% for Cb,Cr ie., assign previous row element to respective row element  
  
for i=2:2:r  
    for j=1:c  
        Cb_upsample(i,j)=Cb_upsample(i-1,j);  
        Cr_upsample(i,j)=Cr_upsample(i-1,j);  
    end  
end
```

```
Cb_upsample=uint8(Cb_upsample);  
Cb_upsample=uint8(Cb_upsample);
```

```
ycbcr__upsample(:,:,1)=IDCT_Y; % assigning y element  
ycbcr__upsample(:,:,2)=Cb_upsample; % assigning the Cb element  
ycbcr__upsample(:,:,3)=Cr_upsample; %assigning the Cr element
```

```
%converting ycbcrimage obtained by row column upsampling to RGB image  
rgb__upsample=ycbcr2rgb(ycbcr__upsample);  
figure(19);  
imshow(rgb__upsample);  
title('Reconstructed RGB image');
```

Reconstructed RGB image



```
%Subplotting the original rgb image and rgbimg  
% obtained from rowcolumn replication upsampling  
figure(20);  
imshow(img),  
title('Original image');
```

Original image



```
figure(21)
imshow(rgb__upsample);
title('Reconstructed image');
```

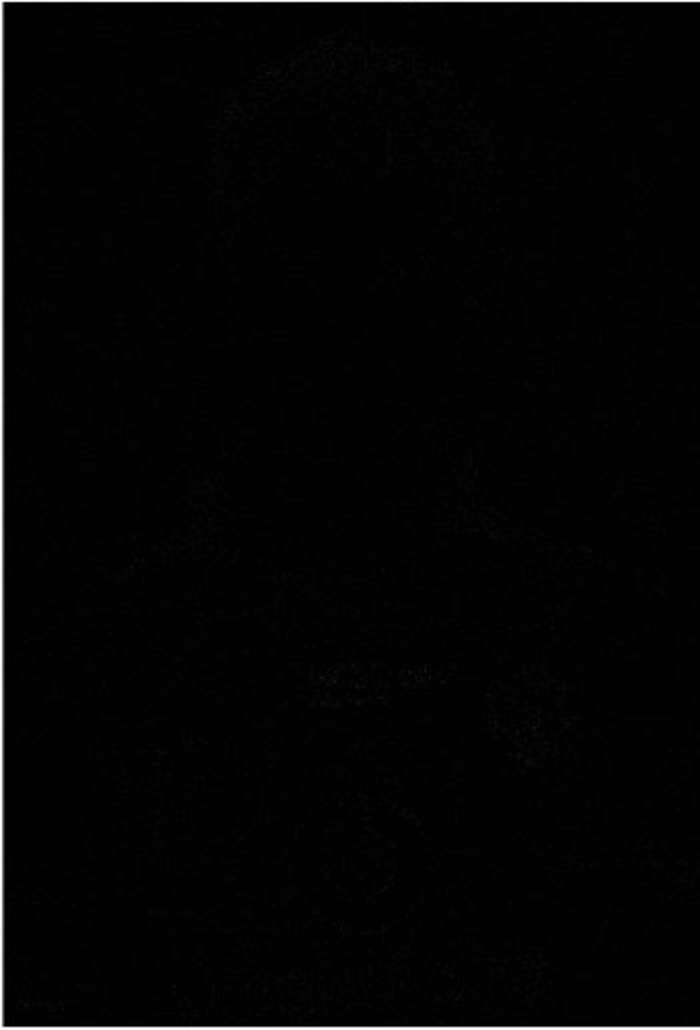
Reconstructed image



Display the Error Image (by subtracting the reconstructed image from the original) for the luminance image.

```
% difference between initial y element and inverse DCT of Y element
Error_luminance = Y_component - IDCT_Y;
figure(21);
imshow(Error_luminance);
title('error image');% error image of luminance
```

error image



Mean Squared Error(MSE) and PSNR(Peak Signal to Noise Ratio) in Image Compression:

PSNR and MSE are the error metrics used to compare image compression quality.

MSE represents the cumulative squared error between the compressed and the original image.

- The lower the value of MSE, the lower the error.

```
% MEAN SQUARE ERROR OF Y element  
Difference = (Y_component - IDCT_Y).^2;  
S = sum(sum(Difference));
```

```
MSE = S/(r*c)
```

```
MSE = 6.3205
```

```
Mse=sprintf ('The Mean Square Error for luminance component is %.2f\n',MSE)
```

```
Mse =  
'The Mean Square Error for luminance component is 6.32  
,
```

PSNR stands for Peak Signal to Noise Ratio. This ratio is often used as a quality measurement between the original and a compressed image.

- The higher the PSNR, the better the quality of the compressed, or reconstructed image.

```
% PSNR OF luminance(y)  
PSNR_Y1=10*(log10(((255)^2)/MSE)); % Calculation of PSNR  
psnr_Y = sprintf('PSNR for luminance component = %.2f dB\n',PSNR_Y1)
```

```
psnr_Y =  
'PSNR for luminance component = 40.12 dB  
,
```