

Questão 1

Gabriel Tavares - 10773801

► PlotlyBackend()

```
. begin
.   using Pkg
.   using PlutoUI
.   using Images
.   using DSP
.   using FileIO
.   using FFTW
.   using ImageShow
.   using Statistics
.   using ImageCore
.   using Plots
.   plotly()
. end
```



```
• begin  
•     imagem_original = load("cat.png")[1:end-1,:]  
• end
```

Conversão RGB -> YCbCr

Aqui os canais de cor da imagem são convertidos de RGB para YCbCr seguindo a relação

$$Y = \alpha_r R + \alpha_b B + \alpha_g G$$

$$Cb = \frac{1}{2(1 - \alpha_b)}(B - Y)$$

$$Cr = \frac{1}{2(1 - \alpha_r)}(R - Y)$$

```
• begin
•   Y_original = get_Y.(imagem_original)
•   Cb_original = get_Cb.(imagem_original)
•   Cr_original = get_Cr.(imagem_original)
•   noprint
• end
```

Subamostragem Cb e Cr

Nos novos canais de cor, iremos subamostrar pela metade os canais de crominância. Para isso, iremos pular linhas e colunas da imagem nesses canais

```
• begin
•   Cr_sub = Cr_original[1:2:end, 1:2:end]
•   Cb_sub = Cb_original[1:2:end, 1:2:end]
•   noprint
• end
```

DCT blocos

Agora em cada canal iremos dividir a imagem em bloco de 4x4, 8x8 e 16x16. Em cada bloco iremos aplicar a transformada discreta de cossenos.

Para isso, iremos primeiro subtrair os blocos por 0.5 para a imagem variar entre **-0.5** até **0.5** e multiplicar os blocos por **255** para podermos quantizar esses valores no item seguinte. A quantização considera valores de pixels que variam de **0** até **255**, e não **0.0** até **1.0** como é nessa implementação de imagens.

Para poder dividir a imagem em blocos sem perda, vamos adicionar linhas e colunas na borda da imagem para termos um número inteiro de blocos.

```

• begin
•     #Tamanhos finais da imagem
•     linhas_original = size(imagem_original)[1]
•     colunas_original = size(imagem_original)[2]
•
•     #Tamanhos finais da imagem
•     linhas_original_Y = size(Y_original)[1]
•     colunas_original_Y = size(Y_original)[2]
•
•     linhas_original_C = size(Cb_sub)[1]
•     colunas_original_C = size(Cb_sub)[2]
•     noprint
• end

```

Blocos DCT - 4x4

```

• begin
•     #ADIÇÃO DE LINHAS PRA TER BLOCOS COMPLETOS DE 8X8
•     linhas_add_Y_4 = linhas_original_Y % 4 == 0 ? 0 : 4 - linhas_original_Y % 4
•     colunas_add_Y_4 = colunas_original_Y % 4 == 0 ? 0 : 4 - colunas_original_Y % 4
•
•     linhas_expand_Y_4 = linhas_original_Y + linhas_add_Y_4
•     colunas_expand_Y_4 = colunas_original_Y + colunas_add_Y_4
•
•
•     linhas_add_C_4 = linhas_original_C % 4
•     colunas_add_C_4 = colunas_original_C % 4
•
•     linhas_expand_C_4 = linhas_original_C + linhas_add_C_4
•     colunas_expand_C_4 = colunas_original_C + colunas_add_C_4
•
•
•     Y_expand_4= my_padarray(Y_original, linhas_add_Y_4, colunas_add_Y_4) .- 0.5
•     Cb_expand_4 = my_padarray(Cb_sub, linhas_add_C_4, colunas_add_C_4) .- 0.5
•     Cr_expand_4 = my_padarray(Cr_sub, linhas_add_C_4, colunas_add_C_4) .- 0.5
•
•     noprint
• end

```

```

• begin
•     #DCT por bloco
•
•     #Luminancia=====
•     numx_blocos_Y_4 = size(Y_expand_4)[1]÷4
•     numy_blocos_Y_4 = size(Y_expand_4)[2]÷4
•     Y_dct_4 = zeros(size(Y_expand_4))
•
•     for i in 1:numx_blocos_Y_4
•         for j in 1:numy_blocos_Y_4
•             bloco = Y_expand_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•             Y_dct_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = dct(bloco*255)
•         end
•     end
•
•     #Cromância=====
•     numx_blocos_C_4 = size(Cb_expand_4)[1]÷4
•     numy_blocos_C_4 = size(Cb_expand_4)[2]÷4
•     Cb_dct_4 = zeros(size(Cb_expand_4))
•
•     for i in 1:numx_blocos_C_4
•         for j in 1:numy_blocos_C_4
•             bloco = Cb_expand_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•             Cb_dct_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = dct(bloco*255)
•         end
•     end
•
•     Cr_dct_4 = zeros(size(Cr_expand_4))
•
•     for i in 1:numx_blocos_C_4
•         for j in 1:numy_blocos_C_4
•             bloco = Cr_expand_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•             Cr_dct_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = dct(bloco*255)
•         end
•     end
• end

```

Blocos DCT - 8x8

```

• begin
•     #ADIÇÃO DE LINHAS PRA TER BLOCOS COMPLETOS DE 8x8
•     linhas_add_Y_8 = linhas_original_Y % 8 == 0 ? 0 : 8 - linhas_original_Y % 8
•     colunas_add_Y_8 = colunas_original_Y % 8 == 0 ? 0 : 8 - colunas_original_Y % 8
•
•     linhas_expand_Y_8 = linhas_original_Y + linhas_add_Y_8
•     colunas_expand_Y_8 = colunas_original_Y + colunas_add_Y_8
•
•
•     linhas_add_C_8 = linhas_original_C % 8
•     colunas_add_C_8 = colunas_original_C % 8
•
•     linhas_expand_C_8 = linhas_original_C + linhas_add_C_8
•     colunas_expand_C_8 = colunas_original_C + colunas_add_C_8
•
•
•     Y_expand_8= my_padarray(Y_original, linhas_add_Y_8, colunas_add_Y_8) .- 0.5
•     Cb_expand_8 = my_padarray(Cb_sub, linhas_add_C_8, colunas_add_C_8) .- 0.5
•     Cr_expand_8 = my_padarray(Cr_sub, linhas_add_C_8, colunas_add_C_8).- 0.5
•
•     noprnt
• end

```

```

• begin
•   #DCT por bloco
•   #Luminancia=====
•   numx_blocos_Y_8 = size(Y_expand_8)[1]÷8
•   numy_blocos_Y_8 = size(Y_expand_8)[2]÷8
•   Y_dct_8 = zeros(size(Y_expand_8))
•
•   for i in 1:numx_blocos_Y_8
•     for j in 1:numy_blocos_Y_8
•       bloco = Y_expand_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•       Y_dct_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = dct(bloco*255)
•     end
•   end
•
•   numx_blocos_C_8 = size(Cb_expand_8)[1]÷8
•   numy_blocos_C_8 = size(Cb_expand_8)[2]÷8
•   Cb_dct_8 = zeros(size(Cb_expand_8))
•
•   for i in 1:numx_blocos_C_8
•     for j in 1:numy_blocos_C_8
•       bloco = Cb_expand_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•       Cb_dct_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = dct(bloco*255)
•     end
•   end
•
•   Cr_dct_8 = zeros(size(Cr_expand_8))
•
•   for i in 1:numx_blocos_C_8
•     for j in 1:numy_blocos_C_8
•       bloco = Cr_expand_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•       Cr_dct_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = dct(bloco*255)
•     end
•   end
• end

```

Blocos DCT - 16x16

```

• begin
•   #ADIÇÃO DE LINHAS PRA TER BLOCOS COMPLETOS DE 8x8
•   linhas_add_Y_16 = linhas_original_Y % 16 == 0 ? 0 : 16 - linhas_original_Y % 16
•   colunas_add_Y_16 = colunas_original_Y % 16 == 0 ? 0 : 16 - colunas_original_Y %
16
•
•   linhas_expand_Y_16 = linhas_original_Y + linhas_add_Y_16
•   colunas_expand_Y_16 = colunas_original_Y + colunas_add_Y_16
•
•
•   linhas_add_C_16 = linhas_original_C % 16
•   colunas_add_C_16 = colunas_original_C % 16
•
•   linhas_expand_C_16 = linhas_original_C + linhas_add_C_16
•   colunas_expand_C_16 = colunas_original_C + colunas_add_C_16
•
•
•   Y_expand_16= my_paddarray(Y_original, linhas_add_Y_16, colunas_add_Y_16) .- 0.5
•   Cb_expand_16 = my_paddarray(Cb_sub, linhas_add_C_16, colunas_add_C_16) .- 0.5
•   Cr_expand_16 = my_paddarray(Cr_sub, linhas_add_C_16, colunas_add_C_16).- 0.5
•
•   noprint
• end

```

```
• begin
•   #DCT por bloco
•   #Luminancia=====
•   numx_blocos_Y_16 = size(Y_expand_16)[1]÷16
•   numy_blocos_Y_16 = size(Y_expand_16)[2]÷16
•   Y_dct_16 = zeros(size(Y_expand_16))
•
•   for i in 1:numx_blocos_Y_16
•       for j in 1:numy_blocos_Y_16
•           bloco = Y_expand_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•           Y_dct_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = dct(bloco*255)
•       end
•   end
•
•   numx_blocos_C_16 = size(Cb_expand_16)[1]÷16
•   numy_blocos_C_16 = size(Cb_expand_16)[2]÷16
•   Cb_dct_16 = zeros(size(Cb_expand_16))
•
•   for i in 1:numx_blocos_C_16
•       for j in 1:numy_blocos_C_16
•           bloco = Cb_expand_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•           Cb_dct_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = dct(bloco*255)
•       end
•   end
•
•   Cr_dct_16 = zeros(size(Cr_expand_16))
•
•   for i in 1:numx_blocos_C_16
•       for j in 1:numy_blocos_C_16
•           bloco = Cr_expand_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•           Cr_dct_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = dct(bloco*255)
•       end
•   end
• end
```


Quantização

Com os canais divididos em blocos de DCTs, iremos quantizar os coeficientes de cada bloco para eliminar as informações menos importantes. Usaremos a matriz de quantização:

$$Q = k \begin{bmatrix} 8 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Em cada bloco iremos realizar uma divisão inteira entre o bloco e a matriz para ter a quantização.

Essa é uma matriz 8x8, portanto terá que ser adaptada pros blocos 4x4 e 16x16.

No bloco **4x4** iremos apenas subamostrar a matriz, pulando linhas e colunas.

No bloco **16x16** iremos superamostrar a matriz colocando zeros entre as linhas e colunas e depois passar por um filtro de interpolação linear. A última linha e coluna serão apenas replicadas das anteriores por causa do efeito de borda da interpolação.

Além disso a matriz **Q_n** é multiplicada por um fator **k = 3**, para termos diferentes níveis de qualidade na quantização.



```
@bind k Slider(1:7, default = 3, show_value = true)
```

```
begin
    Q =
    [8 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]

    Q = Q*k

    noprint
end
```

Quantização - 4x4


```

• begin
•   Q_4 = Q[1:2:end, 1:2:end]
•   #Luminancia=====
•   Y_dct_q_4 = zeros(size(Y_expand_4))
•
•   for i in 1:numx_blocos_Y_4
•     for j in 1:numy_blocos_Y_4
•       bloco = Y_dct_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•       Y_dct_q_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = (bloco).÷Q_4
•     end
•   end
•
•   Cb_dct_q_4 = zeros(size(Cb_expand_4))
•
•   for i in 1:numx_blocos_C_4
•     for j in 1:numy_blocos_C_4
•       bloco = Cb_dct_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•       Cb_dct_q_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = (bloco).÷Q_4
•     end
•   end
•
•   Cr_dct_q_4 = zeros(size(Cr_expand_4))
•
•   for i in 1:numx_blocos_C_4
•     for j in 1:numy_blocos_C_4
•       bloco = Cr_dct_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•       Cr_dct_q_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = (bloco).÷Q_4
•     end
•   end
• end

```

Quantização - 8x8

```

• begin
•   Q_8 = Q
•
•   #Luminancia=====
•   Y_dct_q_8 = zeros(size(Y_expand_8))
•
•   for i in 1:numx_blocos_Y_8
•     for j in 1:numy_blocos_Y_8
•       bloco = Y_dct_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•       Y_dct_q_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = (bloco).÷Q_8
•     end
•   end
•
•   Cb_dct_q_8 = zeros(size(Cb_expand_8))
•
•   for i in 1:numx_blocos_C_8
•     for j in 1:numy_blocos_C_8
•       bloco = Cb_dct_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•       Cb_dct_q_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = (bloco).÷Q_8
•     end
•   end
•
•   Cr_dct_q_8 = zeros(size(Cr_expand_8))
•
•   for i in 1:numx_blocos_C_8
•     for j in 1:numy_blocos_C_8
•       bloco = Cr_dct_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•       Cr_dct_q_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = (bloco).÷Q_8
•     end
•   end
• end

```

Quantização - 16x16

```

• begin
•     Q_16 = zeros(16,16)
•     Q_16[1:2:end, 1:2:end] = Q
•     Q_16 = imfilter(Q_16, int_filter, Fill(0)) #interpolação das frequencias altas
•     Q_16[end-1, end] = Q_16[end-1, end - 1]
•     Q_16[end, end-1] = Q_16[end-1, end-1]
•     Q_16[:, end] = Q_16[:, end-1]
•     Q_16[end, :] = Q_16[end-1, :]
•
•
•
•     #Luminancia=====
•     Y_dct_q_16 = zeros(size(Y_expand_16))
•
•     for i in 1:numx_blocos_Y_16
•         for j in 1:numy_blocos_Y_16
•             bloco = Y_dct_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•             Y_dct_q_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = (bloco).÷Q_16
•         end
•     end
•
•     Cb_dct_q_16 = zeros(size(Cb_expand_16))
•
•     for i in 1:numx_blocos_C_16
•         for j in 1:numy_blocos_C_16
•             bloco = Cb_dct_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•             Cb_dct_q_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = (bloco).÷Q_16
•         end
•     end
•
•     Cr_dct_q_16 = zeros(size(Cr_expand_16))
•
•     for i in 1:numx_blocos_C_16
•         for j in 1:numy_blocos_C_16
•             bloco = Cr_dct_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•             Cr_dct_q_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = (bloco).÷Q_16
•         end
•     end
• end

```

Reconstrução

Para reconstruir a imagens, iremos fazer o inverso dos itens anteriores.

- Em cada bloco iremos multiplicar o bloco pela matriz de quantização para recuperar os valores quantizados.
- Depois faremos a Transformada Discreta de Cossenos Inversa para ter a imagem no domínio do espaço
- Dividimos essa imagem por **255** para recuperar a forma que julia interpreta as imagens (0.0 até 1.0)
- Por fim somamos **0.5** a todos os pixels da imagem

Com isso temos os 3 canais da imagem recuperados no domínio espacial e na escala correta.

Por fim, é necessário superamostrar os canais de cromância que estão subamostrados. Para isso, preenchemos as linhas e colunas com zeros e fazemos uma interpolação linear usando um filtro interpolador.

Reconstrução - 4x4

```
• begin
•   #Luminancia=====
•   Y_rec_4 = zeros(size(Y_expand_4))
•
•   for i in 1:numx_blocos_Y_4
•       for j in 1:numy_blocos_Y_4
•           bloco = Y_dct_q_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•           bloco = bloco.*Q_4
•           bloco = idct(bloco)./255
•           Y_rec_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = bloco .+ 0.5
•       end
•   end
•   Y_rec_4 = Y_rec_4[1:linhas_original, 1:colunas_original]
•
•   Cb_rec_sub_4 = zeros(size(Cb_expand_4))
•
•   for i in 1:numx_blocos_C_4
•       for j in 1:numy_blocos_C_4
•           bloco = Cb_dct_q_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•           bloco = bloco.*Q_4
•           bloco = idct(bloco)./255
•           Cb_rec_sub_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = bloco .+ 0.5
•       end
•   end
•
•   Cb_rec_4 = zeros(size(Cb_rec_sub_4).*2)
•   Cb_rec_4[1:2:end, 1:2:end] = Cb_rec_sub_4
•   Cb_rec_4 = imfilter(Cb_rec_4, int_filter)
•   Cb_rec_4 = Cb_rec_4[1:linhas_original, 1:colunas_original]
•
•
•   Cr_rec_sub_4 = zeros(size(Cr_expand_4))
•
•   for i in 1:numx_blocos_C_4
•       for j in 1:numy_blocos_C_4
•           bloco = Cr_dct_q_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4]
•           bloco = bloco.*Q_4
•           bloco = idct(bloco)./255
•           Cr_rec_sub_4[(i-1)*4 + 1:i*4, (j-1)*4+1:j*4] = bloco .+ 0.5
•       end
•   end
•
•   Cr_rec_4 = zeros(size(Cr_rec_sub_4).*2)
•   Cr_rec_4[1:2:end, 1:2:end] = Cr_rec_sub_4
•   Cr_rec_4 = imfilter(Cr_rec_4, int_filter)
•   Cr_rec_4 = Cr_rec_4[1:linhas_original, 1:colunas_original]
•   noprint
•
• end
```

Reconstrução - 8x8

```

• begin
•   #Luminancia=====
•   Y_rec_8 = zeros(size(Y_expand_8))
•
•   for i in 1:numx_blocos_Y_8
•       for j in 1:numy_blocos_Y_8
•           bloco = Y_dct_q_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•           bloco = bloco.*Q_8
•           bloco = idct(bloco)./255
•           Y_rec_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = bloco .+ 0.5
•       end
•   end
•   Y_rec_8 = Y_rec_8[1:linhas_original, 1:colunas_original]
•
•   Cb_rec_sub_8 = zeros(size(Cb_expand_8))
•
•   for i in 1:numx_blocos_C_8
•       for j in 1:numy_blocos_C_8
•           bloco = Cb_dct_q_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•           bloco = bloco.*Q_8
•           bloco = idct(bloco)./255
•           Cb_rec_sub_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = bloco .+ 0.5
•       end
•   end
•
•   Cb_rec_8 = zeros(size(Cb_rec_sub_8).*2)
•   Cb_rec_8[1:2:end, 1:2:end] = Cb_rec_sub_8
•   Cb_rec_8 = imfilter(Cb_rec_8, int_filter)
•   Cb_rec_8 = Cb_rec_8[1:linhas_original, 1:colunas_original]
•
•   #---
•
•   Cr_rec_sub_8 = zeros(size(Cr_expand_8))
•
•   for i in 1:numx_blocos_C_8
•       for j in 1:numy_blocos_C_8
•           bloco = Cr_dct_q_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8]
•           bloco = bloco.*Q_8
•           bloco = idct(bloco)./255
•           Cr_rec_sub_8[(i-1)*8 + 1:i*8, (j-1)*8+1:j*8] = bloco .+ 0.5
•       end
•   end
•
•   Cr_rec_8 = zeros(size(Cr_rec_sub_8).*2)
•   Cr_rec_8[1:2:end, 1:2:end] = Cr_rec_sub_8
•   Cr_rec_8 = imfilter(Cr_rec_8, int_filter)
•   Cr_rec_8 = Cr_rec_8[1:linhas_original, 1:colunas_original]
•   noprint
•
• end

```

Reconstrução - 16x16

```

• begin
•   #Luminancia=====
•   Y_rec_16 = zeros(size(Y_expand_16))
•
•   for i in 1:numx_blocos_Y_16
•       for j in 1:numy_blocos_Y_16
•           bloco = Y_dct_q_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•           bloco = bloco.*Q_16
•           bloco = idct(bloco)./255
•           Y_rec_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = bloco .+ 0.5
•       end
•   end
•   Y_rec_16 = Y_rec_16[1:linhas_original, 1:colunas_original]
•
•   Cb_rec_sub_16 = zeros(size(Cb_expand_16))
•
•   for i in 1:numx_blocos_C_16
•       for j in 1:numy_blocos_C_16
•           bloco = Cb_dct_q_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•           bloco = bloco.*Q_16
•           bloco = idct(bloco)./255
•           Cb_rec_sub_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = bloco .+ 0.5
•       end
•   end
•
•   Cb_rec_16 = zeros(size(Cb_rec_sub_16).*2)
•   Cb_rec_16[1:2:end, 1:2:end] = Cb_rec_sub_16
•   Cb_rec_16 = imfilter(Cb_rec_16, int_filter)
•   Cb_rec_16 = Cb_rec_16[1:linhas_original, 1:colunas_original]
•
•   #---
•
•   Cr_rec_sub_16 = zeros(size(Cr_expand_16))
•
•   for i in 1:numx_blocos_C_16
•       for j in 1:numy_blocos_C_16
•           bloco = Cr_dct_q_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16]
•           bloco = bloco.*Q_16
•           bloco = idct(bloco)./255
•           Cr_rec_sub_16[(i-1)*16 + 1:i*16, (j-1)*16+1:j*16] = bloco .+ 0.5
•       end
•   end
•
•   Cr_rec_16 = zeros(size(Cr_rec_sub_16).*2)
•   Cr_rec_16[1:2:end, 1:2:end] = Cr_rec_sub_16
•   Cr_rec_16 = imfilter(Cr_rec_16, int_filter)
•   Cr_rec_16 = Cr_rec_16[1:linhas_original, 1:colunas_original]
•   noprint
•
• end

```

Análise

Aqui iremos analisar a qualidade da imagem de forma qualitativa e de forma analítica

```
• begin
•   #visualização das imagens
•   imagem_recuperada_4 = get_RGB.(Y_rec_4, Cb_rec_4, Cr_rec_4)
•   imagem_recuperada_8 = get_RGB.(Y_rec_8, Cb_rec_8, Cr_rec_8)
•   imagem_recuperada_16 = get_RGB.(Y_rec_16, Cb_rec_16, Cr_rec_16)
•
•   mosaico = hcat(imagem_recuperada_4, imagem_recuperada_8, imagem_recuperada_16)
•   noprint
• end
```

Qualitativamente



Qualitativamente, usando $k = 3$ temos

- **16x16** : a coloração está correta, mas vemos grandes blocos se formando, o que deixa a imagem bastante pixelada. Isso ocorre por causa da divisão das DCTs em grandes blocos
- **8x8** : a coloração está correta, e vemos a formação de blocos, mas esses blocos são menores e mais discretos.
- **4x4** : a coloração começa a ter erro, e puxar um pouco para o vermelho. Além disso, é possível ver os blocos se formando, mas por serem pequenos, a sensação de pixelamento da imagem é menor

PSNR

Aqui vamos calcular o **PSNR** de cada imagem usando as diferentes janelas. Esse cálculo vê a PSNR de cada canal e faz a média entre os 3 canais para ter o valor final.

```
• begin
•   PSNR_4 = PSNR(imagem_original, imagem_recuperada_4)
•   PSNR_8 = PSNR(imagem_original, imagem_recuperada_8)
•   PSNR_16 = PSNR(imagem_original, imagem_recuperada_16)
•   noprint
• end
```


Com **k = 3** os diferentes padrões de JPEG tem as seguintes PSNRs

PSNR:

- **4x4** : 26.54dB
- **8x8** : 27.09dB
- **16x16** : 27.15dB

Taxa de números diferentes de zero

Aqui calculamos a quantidade de valores diferentes de zero para analisar a quantidade de coeficientes que carregam informações que ainda temos.

```
• begin
•     num_zeros_4 = sum(Y_dct_q_4 .!= 0) + sum(Cb_dct_q_4 .!= 0) + sum(Cr_dct_q_4 .!=
•     0)
•     not_zeros_ratio_4 =
•     num_zeros_4/(n_element(Y_dct_q_4)+n_element(Cb_dct_q_4)+n_element(Cr_dct_q_4))
•
•     num_zeros_8 = sum(Y_dct_q_8 .!= 0) + sum(Cb_dct_q_8 .!= 0) + sum(Cr_dct_q_8 .!=
•     0)
•     not_zeros_ratio_8 =
•     num_zeros_8/(n_element(Y_dct_q_8)+n_element(Cb_dct_q_8)+n_element(Cr_dct_q_8))
•
•     num_zeros_16 = sum(Y_dct_q_8 .!= 0) + sum(Cb_dct_q_8 .!= 0) + sum(Cr_dct_q_8 .!=
•     0)
•     not_zeros_ratio_16 =
•     num_zeros_16/(n_element(Y_dct_q_16)+n_element(Cb_dct_q_16)+n_element(Cr_dct_q_16))
•     noprint
• end
```

Com **k = 3** os diferentes padrões de JPEG tem as seguintes taxas de números diferentes de zero

Taxa de numeros:

- **4x4** : 8.53%
- **8x8** : 4.65%
- **16x16** : 4.61%

Conclusão

Com essa simulação, vemos a importância da escolha de uma janela ideal no padrão JPEG.

A janela de **4x4** é uma janela que tem menor compressão da imagem (mais valores diferentes de zero) e traz algumas distorções na imagem final (canais de cor principalmente), mas não tem muito efeito do pixelamento visual.

A janela **16x16** apresenta maior compressão e maior PSNR, mas ao vermos a imagem, fica bastante visível a formação de grandes blocos que deixa a imagem visualmente pior.

Já a janela **8x8** acaba equilibrando essas relações. Ela tem uma PSNR alta, uma taxa de compressão de dados alta e o efeito de pixelamento é menos notável na imagem final.

Functions

```
noprint =
```

```
• noprint = md"""
```

```
n_element (generic function with 1 method)
```

```
• function n_element(matriz)
•     return size(matriz)[1]*size(matriz)[2]
• end
```

Gray matrix images functions

```
• md" ### Gray matrix images functions"
```

```
mat_to_image (generic function with 1 method)
```

```
• function mat_to_image(mat)
•     return RGB.(mat,mat,mat)
• end
```

```
image (generic function with 1 method)
```

```
• function image(mat)
•     return RGB.(mat,mat,mat)
• end
```

RGB and YCbCr functions

```
• md" ### RGB and YCbCr functions
•
• "
```

```
• struct YCbCr_pixel
•     Y
•     Cb
•     Cr
• end
```

luminancia (generic function with 1 method)

```
• function luminancia(pixel::YCbCr_pixel)
•     return pixel.Y
• end
```

croma_blue (generic function with 1 method)

```
• function croma_blue(pixel::YCbCr_pixel)
•     return pixel.Cb
• end
```

croma_red (generic function with 1 method)

```
• function croma_red(pixel::YCbCr_pixel)
•     return pixel.Cr
• end
```

get_Y (generic function with 1 method)

```
• function get_Y(pixel::RGB)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     Y = αr*red(pixel) + αg*green(pixel) + αb*blue(pixel)
•
•     return Y
• end
```

get_Cb (generic function with 1 method)

```
• function get_Cb(pixel::RGB)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     Y = get_Y(pixel)
•     Cb = 1/(2*(1-αb))*(blue(pixel)-Y)
• end
```

get_Cr (generic function with 1 method)

```
• function get_Cr(pixel::RGB)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     Y = get_Y(pixel)
•     Cr = 1/(2*(1-αr))*(red(pixel)-Y)
• end
```

get_YCbCr (generic function with 1 method)

```
• function get_YCbCr(pixel::RGB)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     Y = get_Y(pixel)
•     Cb = get_Cb(pixel)
•     Cr = get_Cr(pixel)
•     return Y, Cb, Cr
• end
```

```
get_YCbCr (generic function with 2 methods)
```

```
• function get_YCbCr(R,G,B)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     pixel = RGB(R,G,B)
•     Y = get_Y(pixel)
•     Cb = get_Cb(pixel)
•     Cr = get_Cr(pixel)
•     return Y, Cb, Cr
• end
```

```
get_R (generic function with 1 method)
```

```
• function get_R(Y, Cb, Cr)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     return Y + (2-2*αr)*Cr
• end
```

```
get_G (generic function with 1 method)
```

```
• function get_G(Y, Cb, Cr)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     return Y - αb/αg*(2-2*αb)*Cb - αr/αg*(2-2*αr)*Cr
• end
```

```
get_B (generic function with 1 method)
```

```
• function get_B(Y, Cb, Cr)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     R = get_R(Y,Cb,Cr)
•     G = get_G(Y, Cb, Cr)
•
•     return Y + (2-2*αb)*Cb
• end
```

```
get_RGB (generic function with 1 method)
```

```
• function get_RGB(Y, Cb, Cr)
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•
•     R = get_R(Y,Cb,Cr)
•     G = get_G(Y, Cb, Cr)
•     B = get_B(Y, Cb, Cr)
•
•     return RGB(R,G,B)
• end
```

PSNR

```
• md" ## PSNR"
```

MSE (generic function with 1 method)

```

• function MSE(imagem_exata, imagem)
•     MSE = 0
•     linhas, colunas = size(imagem)
•
•     for lin in 1:linhas
•         for col in 1:colunas
•             MSE += (imagem_exata[lin, col] - imagem[lin, col])^2
•         end
•     end
•
•     MSE = MSE/(linhas*colunas)
•
•     return MSE
• end

```

PSNR (generic function with 1 method)

```

• function PSNR(imagem_exata, imagem)
•     PSNR = 0
•     maxi = 1.0
•     MSE_ = MSE(imagem_exata, imagem)
•     PSNR = 10*log10(maxi/ MSE_)
•     return PSNR
• end

```

PSNR (generic function with 2 methods)

```

• function PSNR(imagem_exata::Matrix{RGB{N0f8}}, imagem::Matrix{RGB{Float64}})
•     PSNR_r = PSNR(red.(imagem_exata), red.(imagem))
•     PSNR_g = PSNR(green.(imagem_exata), green.(imagem))
•     PSNR_b = PSNR(blue.(imagem_exata), blue.(imagem))
•     PSNR_ = mean([PSNR_r, PSNR_g, PSNR_b])
•
•
•     return PSNR_
•
• end

```

Interpolador

```

• md" ## Interpolador"

```

```

int_filter = 3×3 Matrix{Float64}:
 0.25  0.5  0.25
 0.5   1.0  0.5
 0.25  0.5  0.25

```

```

• int_filter = [0.25  0.5    0.25;
•              0.5    1.0    0.5 ;
•              0.25  0.5    0.25]

```

Padarray

```

• md" ## Padarray"

```

my_padarray (generic function with 1 method)

```

• #Adiciona linhas e colunas no canto direito e linhas na parte inferior da matrix
• # Só faz o espelhamento da imagem
•
• #exemplo
• #A = [ 1 2 3
• #      4 5 6
• #      7 8 9]
• #
• #padarray(A, 2, 2)
• #[ 1 2 3 3 2
• #   4 5 6 6 5
• #   7 8 9 9 8
• #   7 8 9 9 6
• #   4 5 6 8 5
•
• function my_padarray(imagem, linhas_add, colunas_add)
•     linhas_original = size(imagem)[1]
•     colunas_original = size(imagem)[2]
•
•     imagem_out = zeros(typeof(imagem[1]), linhas_original + linhas_add,
•         colunas_original+colunas_add)
•
•     imagem_out[1:linhas_original, 1:colunas_original] = imagem
•
•
•     linhas_preenche = imagem[ end:-1:(end-linhas_add+1) , :]
•     colunas_preenche = imagem[ :, end:-1:(end-colunas_add+1)]
•     quadrado_preenche = imagem[ end:-1:(end-linhas_add+1), end:-1:(end-
•         colunas_add+1)]
•
•
•     imagem_out[linhas_original + 1: end, 1:colunas_original] = linhas_preenche
•     imagem_out[1:linhas_original, colunas_original + 1 : end] = colunas_preenche
•     imagem_out[linhas_original+1:end, colunas_original+1:end] = quadrado_preenche
•
•
•     return imagem_out
• end
•

```