

# JPEG

Gabriel Tavares 10773801

Guilherme Reis 10773700

## ► PlotlyBackend()

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

# Leitura da imagem



```
• begin
•     imagem_original = load("cat.png")
• end
```

## Subamostragem de Cb e Cr

A imagem será convertida do espaço RGB para o espaço YCbCr e depois os canais Cb e Cr serão subamostrados.

```

• begin
•     αr = 0.299
•     αg = 0.587
•     αb = 0.114
•     noprint
• end

```

```

• begin
•     R = red.(imagem_original)
•     G = green.(imagem_original)
•     B = blue.(imagem_original)
•     noprint
• end

```

```

• begin
•     Y = get_Y.(imagem_original)
•     Cb = get_Cb.(imagem_original)
•     Cr = get_Cr.(imagem_original)
•
•     Cbsub = Cb[1:2:end, 1:2:end] # Subamostragem de Cb
•     Crsub = Cr[1:2:end, 1:2:end] # Subamostragem de Cr
•     noprint
• end

```

## Taxa de compressão com a subamostragem

Até agora diminuimos os canais de cores 4 vezes e mantivemos o canal de luminancia, portanto reduzimos pela **metade o tamanho da imagem**

0.5

```

• begin
•     tamanho_original_imagem = n_element(imagem_original) * 3
•     tamanho_int = n_element(Y) + n_element(Cbsub) + n_element(Crsub)
•     compressao_int = round(tamanho_int/tamanho_original_imagem, digits = 2)
• end

```

## Reconstrução da imagem

Com os canais Cb e Cr subamostrados, iremos reconstruir a imagem usando um interpolador de média de 2 pontos na imagem para recriar os canais de cor no tamanho original.

```

int_filter = 3x3 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]

```



```
• begin
•     Cb_int = zeros(size(Y))
•     Cb_int[1:2:end, 1:2:end] = Cbsub
•     Cb_int = imfilter(Cb_int, int_filter)
•
•     Cr_int = zeros(size(Y))
•     Cr_int[1:2:end, 1:2:end] = Crsub
•     Cr_int = imfilter(Cr_int, int_filter)
•
•     RGB_int = get_RGB.(Y, Cb_int, Cr_int)
•
•     imagem_reconstruida = RGB_int
• end
```

```

• begin
•   PSNR_red = MSE(red.(imagem_original), red.(imagem_reconstruida))
•   PSNR_green = MSE(green.(imagem_original), green.(imagem_reconstruida))
•   PSNR_blue = MSE(blue.(imagem_original), blue.(imagem_reconstruida))
•   PSNR_medio = mean([PSNR_red, PSNR_green, PSNR_blue])
•   noprint
• end

```

## Análise da reconstrução

Qualitativamente vemos que a qualidade da imagem permanece muito próxima da original, já que o olho humano é menos sensível a cor do que a luz.

Analiticamente o PSNR médio dos 3 canais da imagem reconstruída : **3.0e-5dB**, o que explica a alta qualidade da imagem em comparação com a original

## DCT

---

### Expansão da imagem

Aqui as imagens devem ter um tamanho múltiplo de 8 para definirmos os blocos de DCT na imagem

```

• begin
•   linhas_original = size(imagem_original)[1]
•   colunas_original = size(imagem_original)[2]
•
•   linhas_add = 8 - (linhas_original - (linhas_original ÷ 8)*8)
•   colunas_add = 8 - (colunas_original - (colunas_original ÷ 8)*8)
•
•   linhas_expand = linhas_original + linhas_add
•   colunas_expand = colunas_original + colunas_add
•   noprint
• end

```

```

• begin
•   Y_expand=padarray(Y,Pad(:symmetric, linhas_add, colunas_add))[1:end, 1:end] .-
0.5
•   Cb_expand = padarray(Cb, Pad(:symmetric, linhas_add, colunas_add))[1:end, 1:end]
•   Cr_expand = padarray(Cr, Pad(:symmetric, linhas_add, colunas_add))[1:end, 1:end]
•
•   Cb_sub = Cb_expand[1:2:end, 1:2:end] .- 0.5 #expandido e subamostrado
•   Cr_sub = Cr_expand[1:2:end, 1:2:end] .- 0.5 #expandido e subamostrado
•
•   noprint
• end

```

## DCT do blocos

A imagem é dividida em blocos de 8x8 e em cada bloco é feita a DCT desse bloco.

```

• begin
•     numx_blocos_Y =size(Y_expand)[1]÷8
•     numy_blocos_Y =size(Y_expand)[2]÷8
•     Y_dct = zeros(size(Y_expand))

•     for i in 1:numx_blocos_Y
•         for j in 1:numy_blocos_Y
•             bloco = Y_expand[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]
•             Y_dct[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] = dct(bloco)
•         end
•     end

•     numx_blocos_Cb =size(Cb_sub)[1]÷8
•     numy_blocos_Cb =size(Cb_sub)[2]÷8
•     Cb_dct = zeros(size(Cb_sub))

•     for i in 1:numx_blocos_Cb
•         for j in 1:numy_blocos_Cb
•             bloco = Cb_sub[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]
•             Cb_dct[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] = dct(bloco)
•         end
•     end

•     numx_blocos_Cr =size(Cr_sub)[1]÷8
•     numy_blocos_Cr =size(Cr_sub)[2]÷8
•     Cr_dct = zeros(size(Cr_sub))

•     for i in 1:numx_blocos_Cr
•         for j in 1:numy_blocos_Cr
•             bloco = Cr_sub[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]
•             Cr_dct[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] = dct(bloco)
•         end
•     end
• end

```

## Quantização das DCTs

Depois de ter o blocos transformados, iremos quatizar os coeficientes da DCT seguindo a tabela de quantização :

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

```

• 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];
•
•   k = 3
•
•   noprint
• end

```

```

• begin
•   #Quantização de Y
•
•   Y_dct_q = zeros(size(Y_expand))
•
•   for i in 1:numx_blocos_Y
•     for j in 1:numy_blocos_Y
•       bloco = Y_dct[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]*255
•       Y_dct_q[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] = round.(bloco./(k*Q))
•     end
•   end
•
•   Cb_dct_q = zeros(size(Cb_sub))
•
•   for i in 1:numx_blocos_Cb
•     for j in 1:numy_blocos_Cb
•       bloco = Cb_dct[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]*255
•       Cb_dct_q[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] = round.(bloco./(k*Q))
•     end
•   end
•
•   Cr_dct_q = zeros(size(Cb_sub))
•
•   for i in 1:numx_blocos_Cr
•     for j in 1:numy_blocos_Cr
•       bloco = Cr_dct[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]*255
•       Cr_dct_q[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] = round.(bloco./(k*Q))
•     end
•   end
•
• end

```

## Reconstrução

Com os coeficientes quantizados, iremos reconstruir a imagem multiplicando os coeficientes quantizados pela matriz de quantização novamente e analizar a diferença entre as imagens.

```

• begin
  •   #DCT inversa
  •
  •   Y_rec_expand = zeros(size(Y_expand))
  •
  •   for i in 1:numx_blocos_Y
  •     for j in 1:numy_blocos_Y
  •       bloco = Y_dct_q[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]
  •       bloco = bloco .* (k*Q)
  •       bloco = bloco./255
  •       bloco = idct(bloco)
  •       bloco = bloco .+ 0.5
  •       Y_rec_expand[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] .= bloco
  •     end
  •   end
  •
  •   Cb_rec_expand = zeros(size(Cb_sub))
  •
  •   for i in 1:numx_blocos_Cb
  •     for j in 1:numy_blocos_Cb
  •       bloco = Cb_dct_q[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]
  •       bloco = bloco .* (k*Q)
  •       bloco = bloco./255
  •       bloco = idct(bloco)
  •       bloco = bloco .+ 0.5
  •       Cb_rec_expand[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] .= bloco
  •     end
  •   end
  •
  •   Cr_rec_expand = zeros(size(Cr_sub))
  •
  •   for i in 1:numx_blocos_Cr
  •     for j in 1:numy_blocos_Cr
  •       bloco = Cr_dct_q[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8]
  •       bloco = bloco .* (k*Q)
  •       bloco = bloco./255
  •       bloco = idct(bloco)
  •       bloco = bloco .+ 0.5
  •       Cr_rec_expand[(i-1)*8 + 1:(i)*8, (j-1)*8+1:(j)*8] .= bloco
  •     end
  •   end
  •
  • end

```

```

• begin
  •   #Interpolação de Cb e Cr
  •   Cb_rec_int_expand = zeros(size(Cb_expand))
  •   Cb_rec_int_expand[1:2:end, 1:2:end] = Cb_rec_expand
  •   Cb_rec_int_expand = imfilter(Cb_rec_int_expand, int_filter)
  •
  •   Cr_rec_int_expand = zeros(size(Cr_expand))
  •   Cr_rec_int_expand[1:2:end, 1:2:end] = Cr_rec_expand
  •   Cr_rec_int_expand = imfilter(Cr_rec_int_expand, int_filter)
  •   noprint
  • end

```

```

• begin
  •   #Recuperação do tamanho original da imagem
  •   Y_rec = Y_rec_expand[1:linhas_original, 1:colunas_original]
  •   Cb_rec = Cb_rec_int_expand[1:linhas_original, 1:colunas_original]
  •   Cr_rec = Cr_rec_int_expand[1:linhas_original, 1:colunas_original]
  •   noprint
  •
  • end

```

Abaixo temos a imagem original à esquerda e a imagem reconstruída à direita



```

• begin
•     imagem_rec = get_RGB.(Y_rec,Cb_rec, Cr_rec)
•
•     PSNR_red_ = MSE(red.(imagem_original), red.(imagem_rec))
•     PSNR_green_ = MSE(green.(imagem_original), green.(imagem_rec))
•     PSNR_blue_ = MSE(blue.(imagem_original), blue.(imagem_rec))
•     PSNR_medio_ = mean([PSNR_red_, PSNR_green_, PSNR_blue_])
•     noprint
• end

```

## Análise

Qualitativamente observamos que as imagens são relativamente parecidas. É possível ver a formação de blocos quadrados mais no fundo da imagem, mas o conteúdo principal ainda é bom. Essa qualidade é bastante dependente do valor de **k**.

- $k = 1$  : a imagem permanece com uma qualidade muito boa e os erros são pouco perceptíveis.
- $k = 3$  : começamos a reparar os blocos de na imagem, principalmente ao fundo
- $k = 5$  : os blocos ficam muito visíveis no fundo e começam a aparecer no gato

Analiticamente o PSNR médio dos 3 canais da imagem reconstruída com **k = 3** é: **0.00124dB**

## Número de coeficientes nulos

```

• begin
•     nulos_Cb = sum(Cb_dct_q .== 0)
•     nulos_Cr = sum(Cr_dct_q .== 0)
•     nulos_Y  = sum(Y_dct_q .== 0)
•
•     razao_nulos = (sum([nulos_Cb, nulos_Cr, nulos_Y]))/
•                     sum([n_element(Cb_dct_q),n_element(Cr_dct_q),n_element(Y_dct_q)])
•     noprint
• end

```

Com isso vemos que **92.86%** dos coeficientes são nulos, o que permite uma codificação dos valores que diminua muito o tamanho da imagem.

## Entropia

---

### Coeficientes DC

```

• begin
•     Y_dct_DC = Y_dct_q[1:8:end, 1:8:end]
•     Cb_dct_DC = Cb_dct_q[1:8:end, 1:8:end]
•     Cr_dct_DC = Cr_dct_q[1:8:end, 1:8:end]
•
•     noprint
• end

```

```

• begin
•     filt_descorrelacao =
•         [0.25    0.5    0.25
•          0.5    1      0.5
•          0.25   0.5    0.25]
•     noprint
• end

```

```

• begin
•     Y_dct_DC_desc = round.(imfilter(Y_dct_DC, filt_descorrelacao))
•     Cb_dct_DC_desc = round.(imfilter(Cb_dct_DC, filt_descorrelacao))
•     Cr_dct_DC_desc = round.(imfilter(Cr_dct_DC, filt_descorrelacao))
•     noprint
• end

```

```

• begin
•     probabilidades_Y_DC = zeros(2047*2 +1)
•     for i in 1:length(probabilidades_Y_DC)
•         probabilidades_Y_DC[i] = sum(Y_dct_DC_desc .== i-2047)
•     end
•     probabilidades_Y_DC = probabilidades_Y_DC./n_element(Y_dct_DC_desc)
•
•
•     probabilidades_Cb_DC = zeros(2047*2 +1)
•     for i in 1:length(probabilidades_Cb_DC)
•         probabilidades_Cb_DC[i] = sum(Cb_dct_DC_desc .== i-2047)
•     end
•     probabilidades_Cb_DC = probabilidades_Cb_DC./n_element(Cb_dct_DC_desc)
•
•     probabilidades_Cr_DC = zeros(2047*2 +1)
•     for i in 1:length(probabilidades_Cr_DC)
•         probabilidades_Cr_DC[i] = sum(Cr_dct_DC_desc .== i-2047)
•     end
•     probabilidades_Cr_DC = probabilidades_Cr_DC./n_element(Cr_dct_DC_desc)
•     noprint
• end

```

```

• begin
•     H_DC_Y = 0
•     for k in 1: length(probabilidades_Y_DC)
•         pk = probabilidades_Y_DC[k]
•         H_DC_Y += pk!=0 ? - log2(pk) * pk : 0
•     end
•
•     H_DC_Cb = 0
•     for k in 1: length(probabilidades_Cb_DC)
•         pk = probabilidades_Cb_DC[k]
•         H_DC_Cb += pk!=0 ? - log2(pk) * pk : 0
•     end
•
•     H_DC_Cr = 0
•     for k in 1: length(probabilidades_Cr_DC)
•         pk = probabilidades_Cr_DC[k]
•         H_DC_Cr += pk!=0 ? - log2(pk) * pk : 0
•     end
• end

```

Comprimento médio é em torno de 5.5, portanto o número de bits necessário é b

## Análise de correlação dos coeficientes DC

Na imagem abaixo estamos plotando os coeficientes DC da transformada de luminância para observarmos que eles praticamente formam a imagem de volta subamostrada. Isso é um indicativo da correlação existente entre trechos da imagem.



# AC

```

• begin
•     Y_dct_AC = Y_dct_q[1:8:end, 1:8:end]
•     Cb_dct_AC = Cb_dct_q[1:8:end, 1:8:end]
•     Cr_dct_AC = Cr_dct_q[1:8:end, 1:8:end]
•
•     nowrap
• end

• begin
•     probabilidades_Y_AC = zeros(2047*2 +1)
•     for i in 1:length(probabilidades_Y_DC)
•         probabilidades_Y_DC[i] = sum(Y_dct_DC_desc .== i-2047)
•     end
•     # probabilidades_Y_DC = probabilidades_Y_DC./n_element(Y_dct_DC_desc)
•
•
•     # probabilidades_Cb_DC = zeros(2047*2 +1)
•     # for i in 1:length(probabilidades_Cb_DC)
•     #     probabilidades_Cb_DC[i] = sum(Cb_dct_DC_desc .== i-2047)
•     # end
•     # probabilidades_Cb_DC = probabilidades_Cb_DC./n_element(Cb_dct_DC_desc)
•
•     # probabilidades_Cr_DC = zeros(2047*2 +1)
•     # for i in 1:length(probabilidades_Cr_DC)
•     #     probabilidades_Cr_DC[i] = sum(Cr_dct_DC_desc .== i-2047)
•     # end
•     # probabilidades_Cr_DC = probabilidades_Cr_DC./n_element(Cr_dct_DC_desc)
•     nowrap
• end

```

# Functions

---

```

nowrap =
• nowrap = md"""

```

n\_element (generic function with 1 method)

```

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

```

## 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)
•    $\alpha_r = 0.299$ 
•    $\alpha_g = 0.587$ 
•    $\alpha_b = 0.114$ 
•
•   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)
•    $\alpha_r = 0.299$ 
•    $\alpha_g = 0.587$ 
•    $\alpha_b = 0.114$ 
•
•   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)
•    $\alpha_r = 0.299$ 
•    $\alpha_g = 0.587$ 
•    $\alpha_b = 0.114$ 
•
•   return Y + (2-2* $\alpha_r$ )*Cr
• end

```

## get\_G (generic function with 1 method)

```

• function get_G(Y, Cb, Cr)
•    $\alpha_r = 0.299$ 
•    $\alpha_g = 0.587$ 
•    $\alpha_b = 0.114$ 
•
•   return Y -  $\alpha_b/\alpha_g * (2-2*\alpha_b) * Cb - \alpha_r/\alpha_g * (2-2*\alpha_r) * Cr$ 
• end

```

## get\_B (generic function with 1 method)

```

• function get_B(Y, Cb, Cr)
•    $\alpha_r = 0.299$ 
•    $\alpha_g = 0.587$ 
•    $\alpha_b = 0.114$ 
•
•   R = get_R(Y,Cb,Cr)
•   G = get_G(Y, Cb, Cr)
•
•   return Y + (2-2* $\alpha_b$ )*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 = 2^8 -1
•     MSE_ = MSE(imagem_exata, imagem)
•     PSNR = 10*log10(maxi/ MSE_)
•     return PSNR
• end

```