

Programação de Algoritmos Básicos de Processamento de Imagens

Relatório do Trabalho Prático 1

Enzo Borges Segala, Jonathan Gabriel Nunes Mendes, Matheus Machado Cezar,
Marcos Henrique Volpato de Moraes, Miguel Predebon Abichequer, Nicolas Rosenthal
Dal Corso, and Rafael Silveira Bandeira

Universidade Federal do Rio Grande do Sul

11/11/2025

Sumário

1 Tarefas	3
1.1 Interpolação em imagens coloridas	3
1.1.1 Interpolação por vizinho mais próximo	4
1.1.2 Bilinear	6
1.1.3 Bicúbico	8
1.1.4 Script e funções auxiliares	10
1.2 Realce de Imagens e Efeitos de Parâmetros	13
1.2.1 Conceitos Fundamentais	13
1.2.2 Implementação	14
1.2.3 Resultados e Discussão	17
1.2.4 Análise dos Resultados	19
1.3 Filtragem espacial	20
1.3.1 Ruídos Salt & Pepper e Gaussian	20
1.3.2 Unsharp Masking	24
1.4 Dithering em imagens em tons de cinza	29
1.4.1 Conceitos Fundamentais	29
1.4.2 Algoritmos	29
1.4.3 Implementação	30
1.4.4 Resultados e Discussão	33
1.5 Operações Geométricas	36
1.5.1 Resultados	37
1.5.2 Código	40

1 Tarefas

1.1 Interpolação em imagens coloridas

Imagens



(a) cat



(b) hamster

Figura 1: Imagens originais

Algoritmos

Foram escolhidos os algoritmos: Vizinho mais próximo (ordem 0), Bilinear (ordem 1) e Bicúbico (ordem 3)

1.1.1 Interpolação por vizinho mais próximo

Pode-se dizer que a principal vantagem deste algoritmo é a simplicidade da sua implementação e, por conta dela, chega ao resultado mais rápido. Contudo, seu resultado é uma imagem bastante pixelizada, o que torna inadequada a sua utilização em fotografias. Apesar disso, ainda é bastante utilizada por artistas que trabalham com pixel art, já que neste estilo é desejado ver cada pixel.

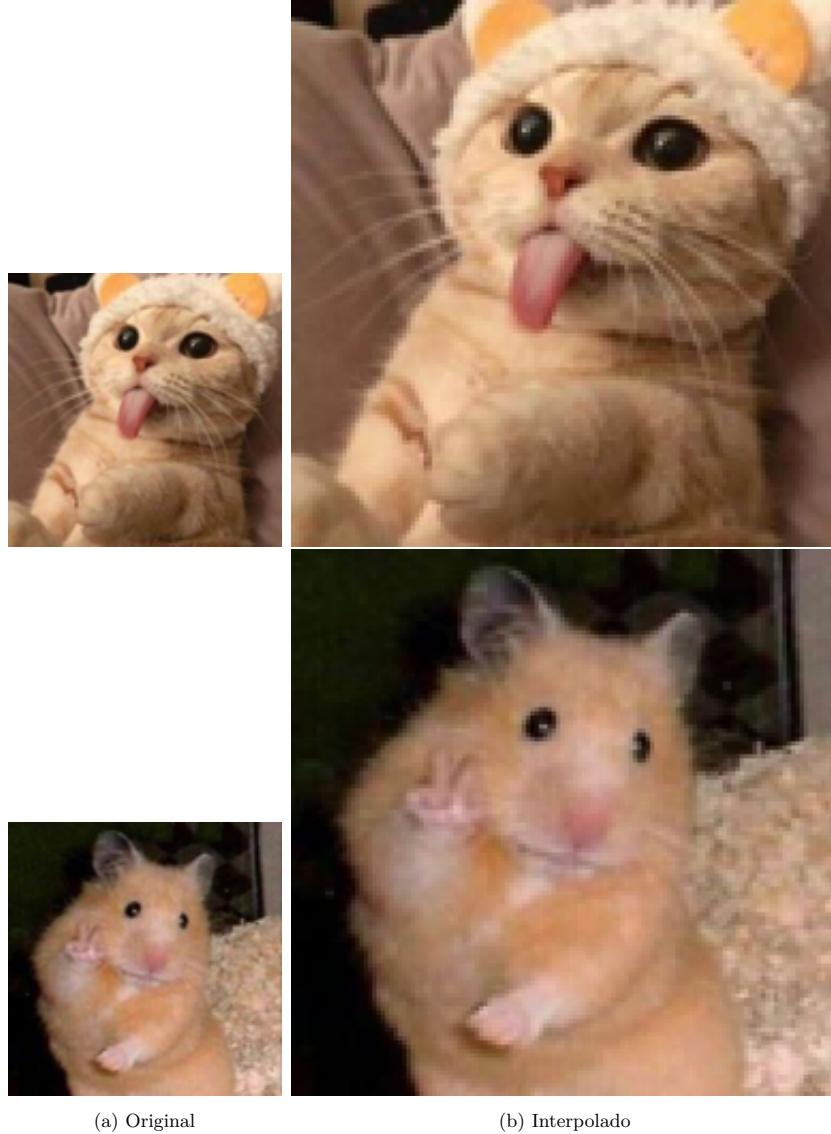


Figura 2: Vizinho mais Próximo

1.1.1.1 Código

```
function out = nearest_neighbour_resize(img, scale)
    [rows, cols, ch] = size(img);
    new_rows = round(rows * scale);
    new_cols = round(cols * scale);
    scale = new_rows / rows;

    out = zeros(new_rows, new_cols, ch);
    for k = 1:ch
        for i = 1:new_rows
            for j = 1:new_cols
                ii = floor((i - 1) / scale) + 1;
                jj = floor((j - 1) / scale) + 1;
                out(i,j,k) = img(ii, jj, k);
            end
        end
    end
end
```

1.1.2 Bilinear

Este algoritmo suaviza as transições entre pixels, o que gera um resultado visual mais natural que o vizinho mais próximo. Contudo, pela simplicidade da interpolação, causa leve borramento e perda de nitidez.

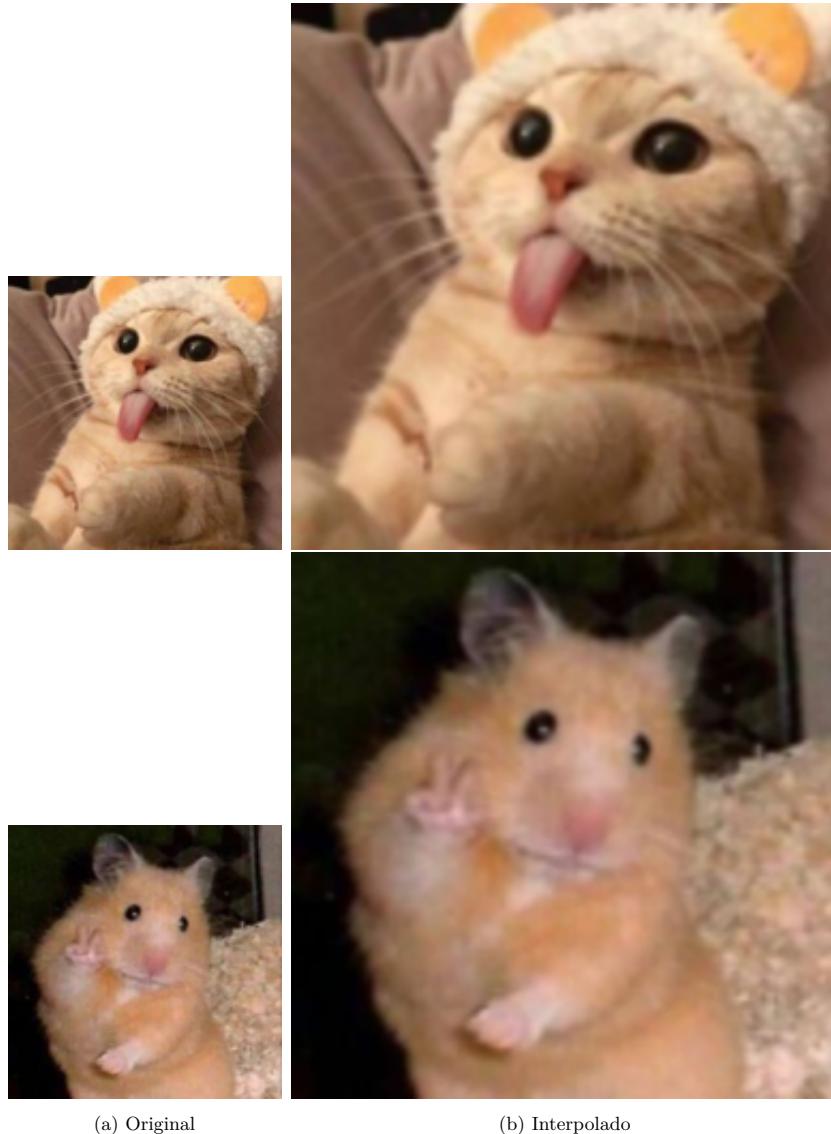


Figura 3: Bilinear

1.1.2.1 Código

```
function out = bilinear_resize(img, scale)
    out = lerp(img, scale, 1);
    out = lerp(out, scale, 2);
end

function out = lerp(mat, scale, dim)
    s = size(mat);
    n_dim = ndims(mat);

    new_s = s;
    new_s(dim) = max(1, round(s(dim) * scale));

    % computar indices
    i_new = (1:new_s(dim))';
    i_in_old = (i_new - 0.5) / scale + 0.5; % centro do pixel ao invés da
                                                borda

    Q1 = floor(i_in_old);
    Q2 = Q1 + 1;
    a = i_in_old - Q1;

    Q1 = max(min(Q1, s(dim)), 1);
    Q2 = max(min(Q2, s(dim)), 1);

    a(Q1 == Q2) = 0;

    % hack de colocar a dimensão na primeira coluna e colapsar o resto
    order = [dim, 1:dim-1, dim+1:n_dim];
    mat_p = permute(mat, order);
    s_mat_p = size(mat_p);
    mat2D = reshape(mat_p, s_mat_p(1), []);

    vals_Q1 = mat2D(Q1, :);
    vals_Q2 = mat2D(Q2, :);

    a = repmat(a, [1, size(vals_Q1, 2)]);

    % interpolacao
    out2D = (1 - a) .* vals_Q1 + a .* vals_Q2;

    % desfazer o hack
    outP = reshape(out2D, [new_s(dim), s_mat_p(2:end)]);
    out = ipermute(outP, order);
    out = cast(out, 'like', mat);
end
```

1.1.3 Bicúbico

Dentre os 3 algoritmos estudados é o que produz melhor resultado, gerando imagens finais mais suaves e detalhadas. Ainda existe perda de nitidez, mas ela é mínima em comparação. É também um algoritmo mais complexo e mais lento que os demais, além de gerar pequenos artefatos em bordas com o contraste muito alto.

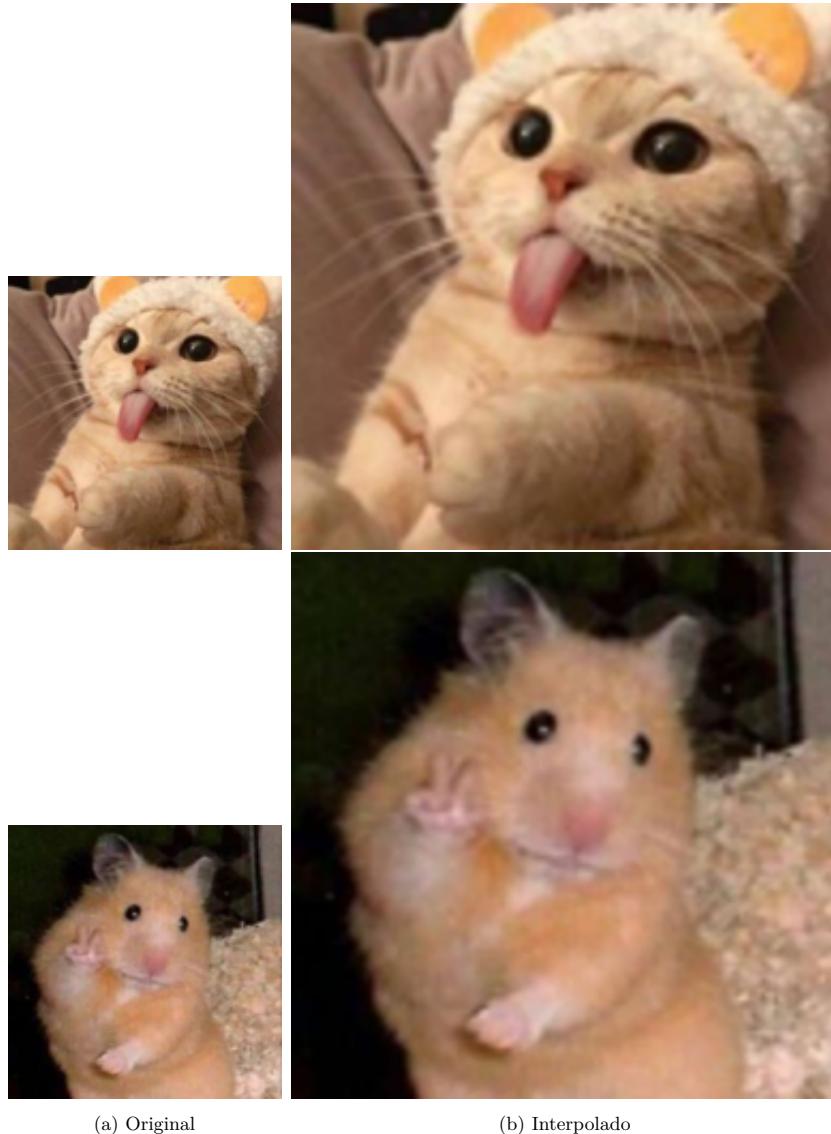


Figura 4: Bicúbico

1.1.3.1 Código

```
function out = bicubic_resize(I, scale)
    [rows, cols, ch] = size(I);
    new_rows = round(rows * scale);
    new_cols = round(cols * scale);

    scale = new_rows / rows;

    a = -0.5;
    function w = kernel(x)
        x = abs(x);
        if x <= 1
            w = (a+2)*x.^3 - (a+3)*x.^2 + 1;
        elseif and((x > 1), (x < 2))
            w = (a*x.^3 - 5*a*x.^2 + 8*a*x - 4*a);
        else
            w = 0;
        end
    end
end

[xQ, yQ] = meshgrid( (1:new_cols)/scale + 0.5*(1 - 1/scale), ...
                      (1:new_rows)/scale + 0.5*(1 - 1/scale) );

out = zeros(new_rows, new_cols, ch);

clip = @(v, lo, hi) max(lo, min(v, hi));

for c = 1:ch
    Ip = padarray(I(:,:,c), [2 2], 'replicate', 'both');
    [rmax, cmax] = size(Ip);

    xQp = xQ + 2;
    yQp = yQ + 2;
    x0p = floor(xQp);
    y0p = floor(yQp);

    dx = xQp - x0p;
    dy = yQp - y0p;

    wx = zeros([size(xQp), 4]);
    wy = zeros([size(yQp), 4]);
    for i = -1:2
        wx(:,:,i+2) = kernel(i - dx);
        wy(:,:,i+2) = kernel(i - dy);
    end

    wx = bsxfun(@rddivide, wx, sum(wx, 3));
    wy = bsxfun(@rddivide, wy, sum(wy, 3));

    val = zeros(new_rows, new_cols);
    for m = -1:2
        for n = -1:2
            xIdx = clip(x0p + n, 1, cmax);
            yIdx = clip(y0p + m, 1, rmax);

            Im = Ip(sub2ind([rmax cmax], yIdx, xIdx));
            val(m+1, n+1) = Im;
        end
    end
end
```

1.1.4 Script e funções auxiliares

```
scale = 2;
baseimgs = {'cat.png', 'hamster.png'};

algorithms = { ...
    'Original' , @(a, b) a ; ...
    'Nearest Neighbour' , @nearest_neighbour_resize; ...
    'Bilinear' , @bilinear_resize ; ...
    'Bicubic' , @bicubic_resize ; ...
};

image_count = length(baseimgs);
runs_per_image = length(algorithms);
total_runs = runs_per_image * length(baseimgs);

names = repmat(algorithms(:,1), image_count, 1);
baseimgnames = repmat(baseimgs, length(algorithms));
f = repmat(algorithms(:,2), image_count, 1);

inimgs = reshape(repmat(baseimgs, runs_per_image, 1), 1, []); %% transpose
outimgs = cell(1, total_runs);

for i = 1:length(inimgs)
    img = im2double(imread(inimgs{i}));

    outimgs{i} = f{i}(img, scale);
end

```

```

function out = docked_figure(varargin)
    out = figure( ...
        'WindowStyle', 'docked', ...
        'NumberTitle', 'off', ...
        'Units', 'normalized', ...
        'Position', [0, 0, 1, 1], ...
        varargin{:} ...
    );
end

function imgs_in_docked_figures(imgs, img_names, names, varargin)

if ~usejava('desktop')
    mkdir('.out');
    for i = 1:length(imgs)
        base_image_name = strrep(strrep(strrep(img_names{i}, '.jpg',
            ''), '.png', ''), '.tif', '');
        name = strrep(lower(names{i}), ' ', '_');

        imwrite(imgs{i}, strcat('.out/', base_image_name, '_', name,
            '.png'));
    end
    return
end

for i = 1:length(imgs)
    figure( ...
        'WindowStyle', 'docked', ...
        'NumberTitle', 'off', ...
        'Units', 'normalized', ...
        'Position', [0, 0, 1, 1], ...
        'Name', names{i} ...
    );
    imshow(imgs{i}, 'InitialMagnification', 'fit');
end
end

scale = 2;
baseimgs = {'cat.png', 'hamster.png'};

algorithms = { ...
    'Original' , @(a, b) a ; ...
    'Nearest Neighbour' , @nearest_neighbour_resize; ...
    'Bilinear' , @bilinear_resize ; ...
    'Bicubic' , @bicubic_resize ; ...
};


```

```

image_count = length(baseimgs);
runs_per_image = length(algorithms);
total_runs = runs_per_image * length(baseimgs);

names = repmat(algorithms(:,1), image_count, 1);
baseimgnames = repmat(baseimgs, length(algorithms));
f = repmat(algorithms(:,2), image_count, 1);

inimgs = reshape(repmat(baseimgs, runs_per_image, 1), 1, []); %%
    transpose
outimgs = cell(1, total_runs);

for i = 1:length(inimgs)
    img = im2double(imread(inimgs{i}));

    outimgs{i} = f{i}(img, scale);
end


```

1.2 Realce de Imagens e Efeitos de Parâmetros

Objetivo

O objetivo deste experimento é analisar o impacto dos parâmetros de manipulação de contraste, equalização de histograma e operações sobre o canal de intensidade (V) em imagens coloridas no espaço HSV. Busca-se compreender como cada técnica modifica a distribuição tonal e o aspecto visual da imagem.

1.2.1 Conceitos Fundamentais

Manipulação de Contraste por Função de Transferência

A manipulação de contraste via função de transferência permite um controle direto sobre a relação entre os níveis de cinza de entrada e de saída da imagem.

Os parâmetros principais são:

- Intervalos definidos sobre a faixa de níveis de cinza (por exemplo, [0-80], [80-160], [160-255]);
- Inclinações (*slopes*) da função de transferência em cada intervalo, que determinam o grau de amplificação ou compressão do contraste;
- Valores iniciais de cada intervalo, que controlam o deslocamento (*offset*) entre as regiões da curva.

O aumento da inclinação em uma faixa de intensidade faz com que pequenas diferenças de níveis de cinza nessa faixa se tornem mais perceptíveis, ampliando o contraste local. Por outro lado, inclinações menores produzem compressão tonal, reduzindo o contraste e “achatando” as variações de intensidade.

A escolha dos intervalos influencia fortemente a aparência final: ao realçar regiões mais escuras, detalhes em sombras tornam-se visíveis, mas regiões claras podem sofrer saturação (perda de detalhe). Da mesma forma, focar o realce nas regiões mais claras pode causar perda de informação nas áreas escuras.

Assim, o efeito visual depende diretamente da forma da curva de transferência - linear, em “S” ou segmentada - e da distribuição tonal da imagem original.

Equalização de Histograma

Na equalização de histograma, o contraste é redistribuído automaticamente de forma a tornar o histograma de níveis de cinza mais uniforme. Esse método tende a aumentar o contraste global da imagem, especialmente em regiões onde o histograma original é concentrado em uma faixa estreita.

O principal parâmetro implícito é o número de níveis de cinza considerados (geralmente 256). Quanto maior essa resolução, mais suave será a transição entre tons. Entretanto, em imagens com ruído, a equalização pode acentuar as flutuações de intensidade indesejadas, tornando o ruído mais visível.

Em imagens com regiões já bem contrastadas, o processo pode gerar saturação (áreas muito claras ou muito escuras) e perda de naturalidade - por isso, é comum empregar versões modificadas, como a equalização adaptativa (CLAHE), quando se deseja preservar o equilíbrio local de contraste.

Imagen Colorida - Canal V em HSV

Ao converter uma imagem RGB para o espaço HSV, o canal V (*Value*) representa a intensidade luminosa, enquanto H (*Hue*) e S (*Saturation*) mantêm as informações de cor.

Aplicar *stretching* de contraste ou equalização de histograma apenas sobre o canal V altera o brilho e o contraste sem distorcer significativamente as cores.

No *stretching*, a escolha dos limites inferior e superior de intensidade define o grau de expansão tonal: intervalos mais amplos aumentam o contraste, enquanto intervalos restritos produzem resultados mais sutis.

Na equalização, o efeito é mais global, podendo intensificar reflexos e aumentar a vivacidade das cores, mas também pode gerar artefatos em regiões uniformes (como céu ou paredes), devido ao aumento local de contraste.

A transformação inversa de HSV para RGB mostra que pequenas variações no canal V podem causar mudanças perceptíveis na saturação e brilho das cores, evidenciando que os parâmetros escolhidos devem equilibrar realce e naturalidade.

1.2.2 Implementação

```
function out = apply_piecewise_transfer(I, edges, slopes, ystart)
% APPLY_PIECEWISE_TRANSFER - Aplica uma função de transferência por
    % trechos
% lineares a uma imagem em tons de cinza (uint8).
%
% Esta função permite ajustar o contraste de uma imagem por meio de uma
% curva de transferência segmentada. Cada intervalo definido em `edges`
% possui uma inclinação específica, dada por `slopes`. Valores iniciais
% opcionais podem ser especificados em `ystart`, permitindo controle
    % sobre
% a continuidade entre os segmentos.
%
% Sintaxe:
%   out = apply_piecewise_transfer(I, edges, slopes, ystart)
%
% Parâmetros:
%   I       : imagem de entrada do tipo uint8 (valores entre 0 e 255)
%   edges   : vetor de bordas [x0 x1 ... xN] (de 0 a 255), em ordem
    % crescente
%   slopes  : vetor de tamanho N-1 com as inclinações (ganhos) de cada
    % intervalo
%   ystart  : vetor com os valores iniciais (à esquerda) de cada
    % intervalo.
%
    % Pode ser:
%           - vazio ( [] ) → continuidade automática entre os
    % segmentos;
%           - um único valor → usado como valor inicial do primeiro
    % intervalo, com continuidade automática para os
    % seguintes;
```

```

%           - vetor completo de tamanho N-1, para controle total.
%
% Retorno:
%   out : imagem de saída (uint8) com os níveis de cinza ajustados
%
% Exemplo:
%   edges = [0 80 160 255];
%   slopes = [1.5 0.7 1.2];
%   I2 = apply_piecewise_transfer(I, edges, slopes, []);
%
% Esse exemplo amplia o contraste nas regiões escuras -(080),
% comprime nas médias -(80160) e volta a ampliar nas regiões claras
% -(160255).

% Verificação de tipo da imagem
if ~isa(I,'uint8')
    error('A imagem de entrada (I) deve ser do tipo uint8.');
end

x = 0:255;
nIntervals = length(edges) - 1;

% Verificação de consistência entre os parâmetros
if length(slopes) ~= nIntervals
    error('O número de inclinações (slopes) deve ser igual ao número
          de intervalos definidos em edges.');
end

% Inicialização do vetor ystart
if isempty(ystart)
    ystart = nan(1, nIntervals);
end

% Caso o usuário forneça apenas um valor inicial
if numel(ystart) == 1
    tmp = nan(1, nIntervals);
    tmp(1) = ystart(1);
    ystart = tmp;
end

% Caso o vetor fornecido seja menor que o necessário
if numel(ystart) < nIntervals
    tmp = nan(1, nIntervals);
    tmp(1:numel(ystart)) = ystart;
    ystart = tmp;
end

```

```

% Construção da LUT (tabela de correspondência de intensidades)
lut = zeros(1, 256);

for k = 1:nIntervals
    xL = edges(k);
    xR = edges(k+1);
    idx = (x >= xL) & (x <= xR);

    % Determinação do valor inicial y0
    if isnan(ystart(k))
        if k == 1
            y0 = 0;
        else
            % Continuidade automática: o início do segmento atual
            % é igual ao valor final do segmento anterior.
            y0 = lut(edges(k) + 1); % +1 devido ao índice 1-based do
            MATLAB
        end
    else
        y0 = ystart(k);
    end

    % Cálculo dos valores de saída dentro do intervalo
    m = slopes(k);
    yvals = y0 + m * (x(idx) - xL);

    % Saturação dos valores dentro do intervalo [0, 255]
    yvals = max(0, min(255, yvals));

    lut(idx) = yvals;
end

% Arredondamento e aplicação da LUT à imagem
lut = round(lut);
out = uint8(lut(double(I) + 1)); % Conversão final para uint8
end

function J = hist_equalize_custom(I)
% HIST_EQUALIZE_CUSTOM - Equalização de histograma manual para imagens
% uint8.
%
% Esta função realiza a equalização de histograma de forma explícita,
% construindo o histograma e a função de distribuição acumulada (CDF)
% manualmente. O objetivo é redistribuir os níveis de cinza de modo a
% ocupar de forma mais uniforme a faixa de intensidades possíveis -(0255),

```

```

% aumentando o contraste global da imagem.
%
% Sintaxe:
%   J = hist_equalize_custom(I)
%
% Parâmetros:
%   I : imagem de entrada do tipo uint8 (valores entre 0 e 255)
%
% Retorno:
%   J : imagem de saída (uint8) após equalização do histograma
%
% Descrição do processo:
%   1. Calcula-se o histograma de frequências dos níveis de cinza.
%   2. Obtém-se a probabilidade de ocorrência de cada nível.
%   3. Calcula-se a função de distribuição acumulada (CDF).
%   4. Constrói-se um mapeamento linear: novo nível = 255 * CDF.
%   5. Aplica-se o mapa de transformação sobre todos os pixels da imagem.

% Verificação de tipo da imagem
if ~isa(I, 'uint8')
    error('A imagem de entrada (I) deve ser do tipo uint8.');
end

% Cálculo manual do histograma
counts = zeros(256, 1);
vals = double(I(:));
for k = 0:255
    counts(k + 1) = sum(vals == k);
end

% Probabilidades normalizadas (PDF)
p = counts / numel(I);

% Função de distribuição acumulada (CDF)
cdf = cumsum(p);

% Mapeamento de intensidades: -0255
map = uint8(round(255 * cdf));

% Aplicação do mapeamento à imagem original
J = map(double(I) + 1);

```

1.2.3 Resultados e Discussão

Foram utilizadas as seguintes imagens de teste: **bosqueia.png**, **paisagem.png** e **puma.png**. Os resultados mostram o comportamento comparativo entre *equalização de histograma* e *função de trans-*

ferência, tanto para imagens em tons de cinza quanto coloridas.

Imagen 1 - Bosqueia (Colorida, Canal V em HSV)



Figura 5: Resultados para imagem colorida (canal V em HSV)

Imagen 2 - Paisagem (Escala de Cinza)



Figura 6: Resultados para imagem *paisagem.png*

Imagen 3 - Puma (Escala de Cinza)



Figura 7: Resultados para imagem *puma.png*

1.2.4 Análise dos Resultados

- A **função de transferência** oferece controle manual e preciso sobre o contraste, sendo ideal para ajustes localizados e análise técnica de regiões específicas.
- A **equalização de histograma** tende a produzir um contraste global mais acentuado, realçando detalhes, mas pode causar saturação e perda de naturalidade em regiões uniformes.
- Nas **imagens coloridas**, o realce aplicado ao canal V mantém as cores originais, mas pequenas variações de brilho podem alterar a percepção da saturação e intensidade.

Conclusão

A escolha adequada dos parâmetros de realce depende do objetivo: *funções de transferência* são mais indicadas para ajustes finos e técnicos, enquanto a *equalização de histograma* é preferida para obter resultados visuais mais marcantes de forma automática.

1.3 Filtragem espacial

1.3.1 Ruídos Salt & Pepper e Gaussian



Figura 8: Imagens originais

Adição de Ruídos

Para iniciar esta etapa, deve-se inserir diferentes níveis de ruído sal e pimenta (salt and pepper) e ruído Gaussiano simultaneamente nas imagens selecionadas: “raposa.jpg” e “borboleta.jpg”. Para isso, utiliza-se a função imnoise, responsável por contaminar a imagem com ambos os tipos de ruídos.

Código

```
img1 = imread('borboleta.jpg');
% Adiciona ruído sal e pimenta
img_noisy1 = imnoise(img1, 'salt & pepper', 0.05); % 5% de ruído
% Adiciona ruído gaussiano
img_noisy1 = imnoise(img_noisy1, 'gaussian', 0, 0.01); % média 0, var
0.01

img2 = imread('raposa.jpg');
% Adiciona ruído sal e pimenta
img_noisy2 = imnoise(img2, 'salt & pepper', 0.05); % 5% de ruído
% Adiciona ruído gaussiano
img_noisy2 = imnoise(img_noisy2, 'gaussian', 0, 0.01); % média 0, var
0.01

if usejava('desktop')
    imshow(img_noisy1);
    imshow(img_noisy2);
else
```

```

mkdir('.out');
imwrite(img_noisy1, '.out/borboleta_ruido.png');
imwrite(img_noisy2, '.out/raposa_ruido.png');
end

```

Como resultado, temos as imagens originais contaminadas com ruído Gaussiano de variância 0.01 e com ruído salt & pepper de densidade 0.05.



Figura 9: Imagens com ruído

Filtragem Alpha Trimmed Mean

Em seguida, as imagens passam pelo filtro Alpha Trimmed Mean. Para isso, foi desenvolvida a função `alpha_trimmed_mean_filter` 3x3, que aplica o filtro aos três canais de cor: vermelho, verde e azul, e, posteriormente, os resultados são combinados novamente para gerar a imagem final com os ruídos atenuados.

Código

```

function imgFiltrada = alpha_trimmed_mean_filter(imgRuidosa, windowSize,
    trimAmount)
% imgRuidosa = imagem colorida de entrada
% windowSize = tamanho da janela
% trimAmount = número de pixels a remover das extremidades após ordenar

% Inicializa imagem de saída
imgFiltrada = zeros(size(imgRuidosa));

% Processa cada canal RGB separadamente
for ch = 1:3
    % Extrai canal e converte para double
    canal = double(imgRuidosa(:,:,ch));
    [rows, cols] = size(canal);
    outputCanal = zeros(rows, cols);

```

```

% Limites da vizinhança
maxOffset = ceil(windowSize / 2);
minOffset = floor(windowSize / 2);

% Varre a imagem
for r = maxOffset:(rows - minOffset)
    for c = maxOffset:(cols - minOffset)
        % Extrai janela local
        localRegion = canal(r - minOffset:r + minOffset, c -
            minOffset:c + minOffset);

        % Coloca em vetor, ordena e remove extremos
        values = sort(localRegion(:));
        values = values(trimAmount + 1 : windowSize * windowSize
            - trimAmount);

        % Calcula média dos valores restantes
        outputCanal(r, c) = mean(values);
    end
end

% Mantém bordas originais
outputCanal(1:maxOffset-1, :) = canal(1:maxOffset-1, :);
outputCanal(rows-maxOffset+2:end, :) = canal(rows-maxOffset+2:end
    , :);
outputCanal(:, 1:maxOffset-1) = canal(:, 1:maxOffset-1);
outputCanal(:, cols-maxOffset+2:end) = canal(:, cols-maxOffset+2:
    end);

% Atribui canal processado à imagem de saída
imgFiltrada(:,:,ch) = outputCanal;
end

% Converte para uint8
imgFiltrada = uint8(imgFiltrada);

```



Figura 10: Imagens filtradas

Medições PSNR e SNR

A avaliação da qualidade da filtragem é feita utilizando os parâmetros SNR (Signal-to-Noise Ratio) e PSNR (Peak Signal-to-Noise Ratio), por meio de suas respectivas funções. Como esses indicadores medem a relação entre o sinal original e o ruído, valores mais altos de SNR ou PSNR correspondem a menor presença de ruído, indicando melhor qualidade da imagem.

Código

```



```

```

    fprintf('filtered\nPSNR: %.2f dB, SNR: %.2f dB\n', PSNR_filtered,
           SNR_filtered);
    fprintf('noisy\nPSNR: %.2f dB, SNR: %.2f dB\n', PSNR_noisy, SNR_noisy
           );

```

end

	Borboleta	Raposa
Filtrada PSNR	26.16 dB	24.87 dB
Filtrada SNR	19.84 dB	19.14 dB
Ruidosa PSNR	16.26 dB	16.58 dB
Ruidosa SNR	10.25 dB	11.14 dB

1.3.2 Unsharp Masking

Ao aplicar este filtro, os detalhes da imagem original são realçados sem a introdução de ruído. Inicialmente, a imagem é suavizada por meio de convoluções Gaussianas, que funcionam como filtros passa-baixa, removendo componentes de alta frequência, como texturas finas e bordas. Em seguida, a imagem suavizada é subtraída da original, e os detalhes resultantes são ampliados através da multiplicação por um fator de ganho “amount”. Por fim, esse resultado é somado à imagem original, proporcionando o realce final de detalhes e contornos.

OBS.: 2 parâmetros no código: Sigma: Define o grau de suavização na máscara Gaussiana; Fator de ganho(amount): Define quanto os detalhes serão amplificados;

Código

```



```

```

img.blur = imfilter(img, h, 'replicate');

% Unsharp masking
img_sharp = img + amount*(img - img.blur);

% Garantir que os valores fiquem entre 0 e 1
img_sharp = max(min(img_sharp, 1), 0);

if usejava('desktop')
    % Mostrar resultados
    figure;
    subplot(1,2,1); imshow(img); title('Original');
    subplot(1,2,2); imshow(img_sharp); title('Reälce com Unsharp
        Masking');
else
    mkdir('.out');
    img_id = strrep(sprintf('%.1f_%.1f', sigma, amount), '.', '_');
    ;
    imwrite(img_sharp, strcat('.out/', base_name, '_', img_id, '_unsharp.png'));
end
end
end

```

Comparações

Processando as imagens com $\sigma = 2.5$ e $\text{amount} = 0.5$. Apesar do valor relativamente alto de σ , as alterações não são muito evidentes devido ao baixo valor de amount . Ainda assim, é possível notar que os detalhes maiores, como os detalhes das asas e o pelo, apresentam maior definição e nitidez.

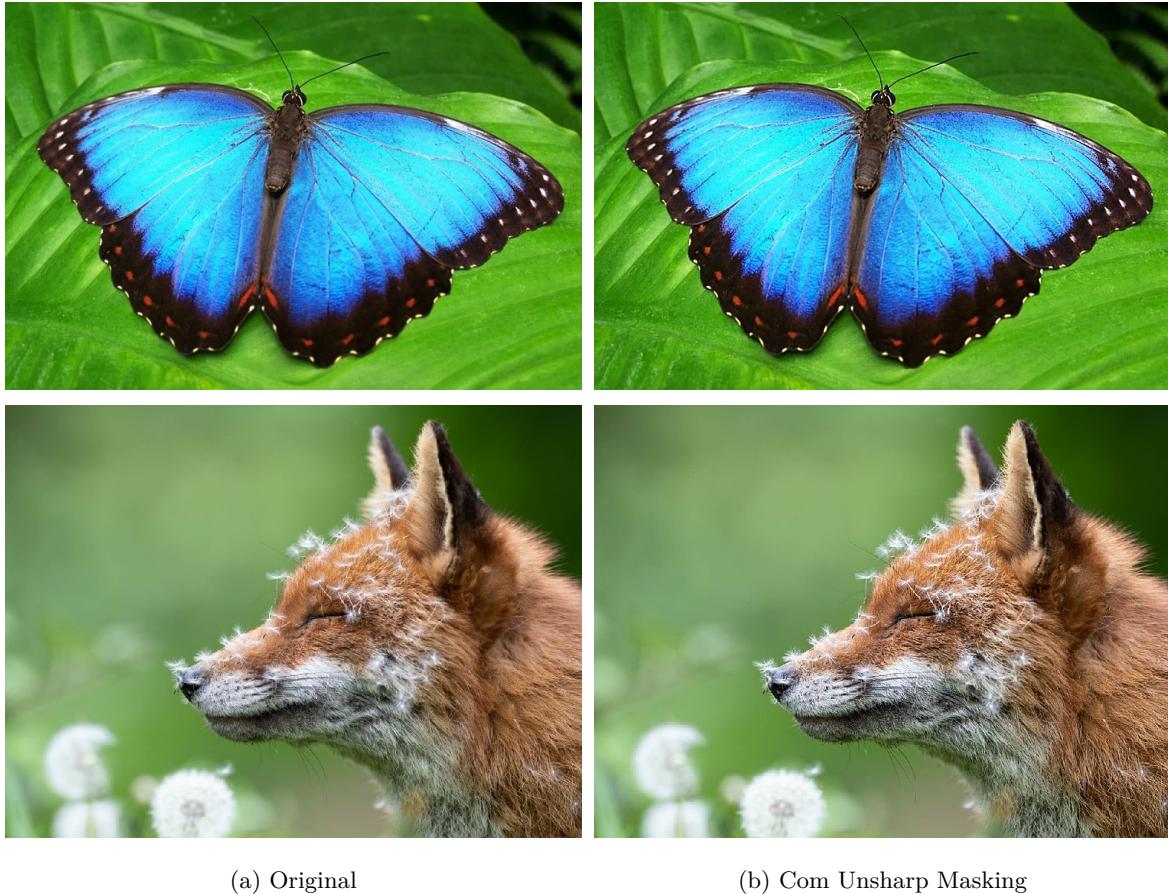
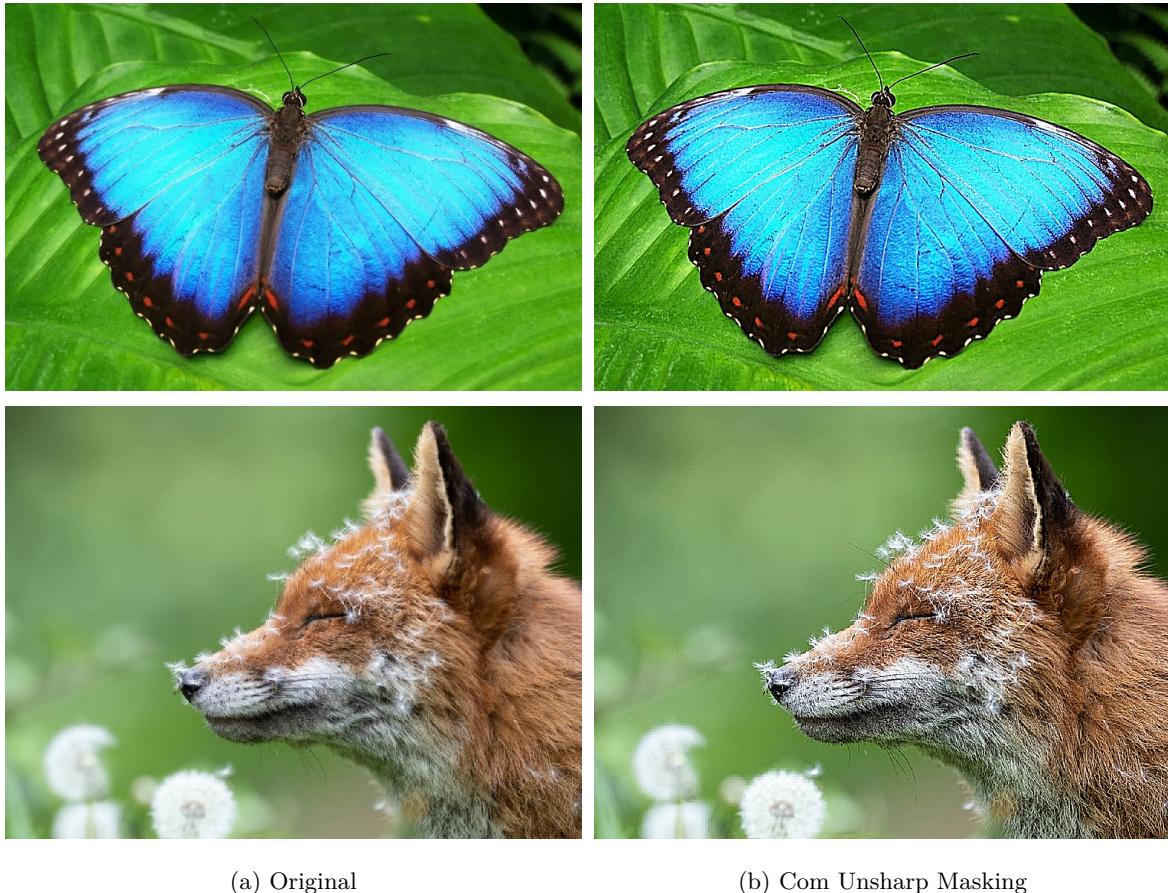


Figura 11: $\sigma=2.5$ e $\text{amount}=0.5$

Processando as imagens com $\sigma = 2.5$ e $\text{amount} = 2.5$. Nesse caso, as alterações tornam-se mais evidentes, com as asas mais nítidas e menos borradadas. Na raposa, a pelagem também apresentou maior definição.



(a) Original

(b) Com Unsharp Masking

Figura 12: $\sigma=2.5$ e $\text{amount}=2.5$

Para avaliar as alterações geradas por diferentes valores de sigma, mantivemos amount = 2.5 para intensificar a máscara. No primeiro experimento, com sigma = 0.5, observa-se praticamente nada de mudança, somente um realce de pequenos detalhes.



Figura 13: sigma=0.5 e amount=2.5

1.4 Dithering em imagens em tons de cinza

Objetivo

Neste experimento, buscamos investigar o efeito dos algoritmos de *dithering* de Floyd-Steinberg e Bayer na redução de níveis de cinza em imagens digitais. As técnicas de dithering buscam preservar a aparência visual de uma imagem quando o número de tons disponíveis é reduzido, simulando graduações suaves por meio da distribuição controlada de pixels claros e escuros.

Metodologia

Foram testados os algoritmos Floyd-Steinberg e Bayer (dithering ordenado) sobre duas imagens em tons de cinza com 8 bits/pixel (256 níveis de cinza). A quantização foi realizada para 2 bits/pixel (4 níveis) e 3 bits/pixel (8 níveis), com o objetivo de comparar a naturalidade e qualidade visual obtidas por cada método, através do histograma das imagens resultantes, bem como usando as métricas de qualidade de imagens (PSNR e SSIM).

1.4.1 Conceitos Fundamentais

Quantização

Quantização é o processo de reduzir o número de níveis possíveis de intensidade em uma imagem. Por exemplo, uma imagem de 8 bits/pixel possui 256 níveis de cinza, enquanto uma imagem de 2 bits/pixel possui apenas 4 níveis. A quantização é essencial em compressão e exibição de imagens em dispositivos com profundidade limitada, mas pode causar *banding* — faixas visíveis entre tons contínuos.

Limiarização

A limiarização (ou *thresholding*) é uma forma simples de quantização binária, em que cada pixel é comparado a um valor de referência (limiar) e convertido em preto ou branco (ou em níveis fixos). Embora simples, esse método tende a gerar regiões abruptas e artificiais, pois não considera o contexto dos pixels vizinhos.

Dithering

O *dithering* introduz ruído controlado para distribuir o erro de quantização entre os pixels, criando a ilusão de tons intermediários. O resultado é visualmente mais suave e natural, especialmente em gradientes e superfícies homogêneas. Diferentemente da limiarização, o *dithering* preserva a percepção de textura e profundidade tonal, mesmo com poucos níveis de cinza.

1.4.2 Algoritmos

Dithering de Floyd-Steinberg

Método **adaptativo** que compensa o erro de quantização de um pixel distribuindo-o para os pixels vizinhos ainda não processados, conforme a máscara de difusão de erro:

$$\text{Máscara de Floyd-Steinberg: } \begin{bmatrix} \frac{3}{16} & \frac{5}{16} & \frac{7}{16} \\ \frac{5}{16} & \frac{1}{16} & \end{bmatrix}$$

Este processo evita a acumulação do erro e cria uma aparência natural, simulando valores intermediários de intensidade.

Dithering Ordenado de Bayer

Método **estático** de dithering que utiliza uma matriz de limiares periódica (Matriz de Bayer) para definir o padrão de quantização:

$$B_4 = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$

Cada pixel é comparado ao limiar correspondente na matriz, e o padrão é repetido sobre toda a imagem, produzindo uma textura visual regular.

1.4.3 Implementação

Código — Dithering de Floyd-Steinberg

```
function [out] = floyd_steinberg(img, bits)
% FLOYD_STEINBERG É uma implementação do algoritmo de dithering de
% Floyd-Steinberg para imagens em tons de cinza.
% converte `img` para double, caso não seja
img = im2double(img);

% dimensões da imagem
[rows, cols] = size(img);

% define imagem de saída vazia e 2^n tons de cinza
out = img;
levels = 2^bits;

% dithering de Floyd-Steinberg
% para cada pixel da imagem, faça:
for y = 1:rows
    for x = 1:cols
        % obtém o valor atual do pixel
        old      = out(y,x);

        % calcula o novo valor do pixel como ...
        new      = round(old * (levels - 1)) / (levels - 1);
        out(y,x)= new;

        % calcula o erro (linear, diferença entre o antigo e o novo
        )
        err      = old - new;
```

```

% Difusão de erro
if x+1 <= cols,                                out(y, x+1) = out(y, x
    +1) + err * 7/16; end
if y+1 <= rows && x > 1,                      out(y+1,x-1) = out(y+1,x
    -1) + err * 3/16; end
if y+1 <= rows,                                out(y+1,x) = out(y+1,x)
    + err * 5/16; end
if y+1 <= rows && x+1 <= cols,   out(y+1,x+1) = out(y+1,x
    +1) + err * 1/16; end
end
end
end

```

Código — Dithering de Bayer

```

function [out] = bayer(img,bits)
% BAYER é a implementação do algoritmo de dithering ordenado de Bayer
% para
% imagens em tons de cinza.

% normalização da imagem para double e em tons de cinza
if ~isfloat(img)
    img = im2double(img);
end
if size(img,3) == 3
    img = rgb2gray(img);
end

% matriz de Bayer de ordem
BayerMatrix = [0, 32, 8, 40, 2, 34, 10, 42;
               48, 16, 56, 24, 50, 18, 58, 26;
               12, 44, 4, 36, 14, 46, 6, 38;
               60, 28, 52, 20, 62, 30, 54, 22;
               3, 35, 11, 43, 1, 33, 9, 41;
               51, 19, 59, 27, 49, 17, 57, 25;
               15, 47, 7, 39, 13, 45, 5, 37;
               63, 31, 55, 23, 61, 29, 53, 21] / 64;

[bayer_rows, bayer_cols] = size(BayerMatrix);

% dimensões da imagem
[rows, cols] = size(img);

% define imagem de saída vazia e os 2^n tons de cinza
out = zeros(rows, cols);
levels = 2^bits;

```

```

%   algoritmo de dithering de Bayer
%   para cada pixel, faça:
for y = 1:rows
    for x = 1:cols
        %   decide qual o limiar de dithering
        threshold = BayerMatrix(mod(y-1, bayer_rows) + 1, mod(x-1,
            bayer_cols) + 1);

        %   o novo pixel será o antigo + limiar
        new = round((img(y,x) + threshold / levels) * (levels - 1)) /
            (levels - 1);

        %   garantir faixa [0,1]:
        out(y,x) = min(max(new,0), 1);
    end
end

```

1.4.4 Resultados e Discussão

Foram testadas as imagens **raposa.jpg** e **borboleta.jpg**, ambas em tons de cinza (8 bits/pixel). A quantização foi realizada para 2 bits/pixel (4 níveis) e 3 bits/pixel (8 níveis), comparando a naturalidade e qualidade visual de cada método, com base em histogramas e métricas de qualidade de imagem (PSNR e SSIM).

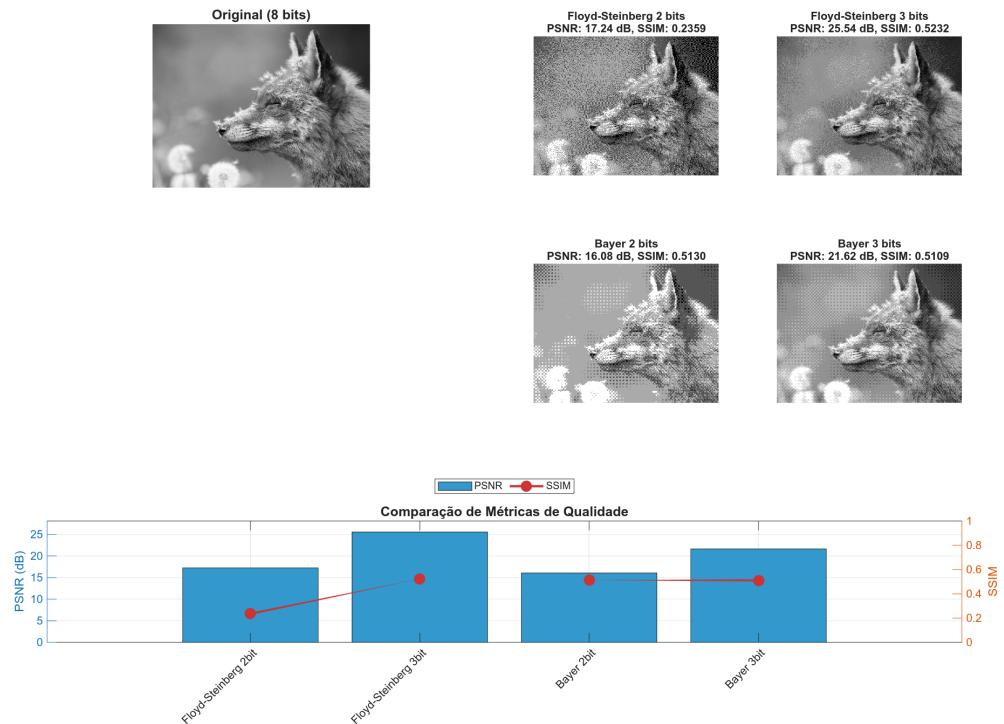


Figura 14: Comparação visual entre métodos de Dithering

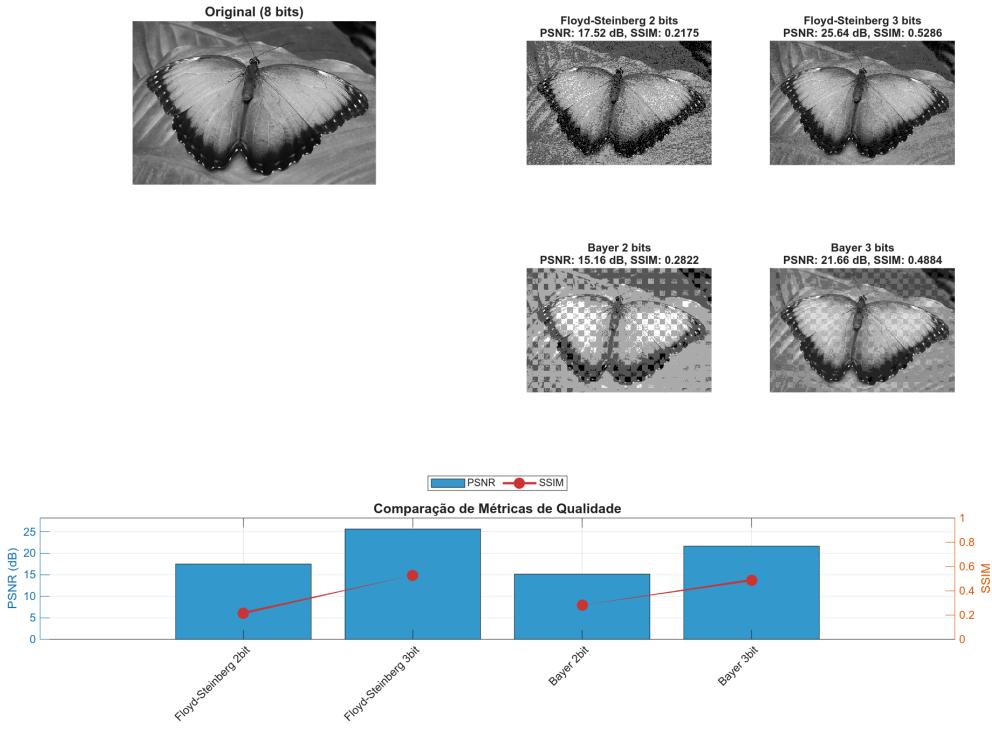


Figura 15: Comparação visual entre métodos de Dithering

Métricas Utilizadas

- **PSNR (Peak Signal-to-Noise Ratio)** — mede a relação entre o sinal máximo e o ruído de quantização. Valores maiores indicam melhor preservação da imagem original.
- **SSIM (Structural Similarity Index)** — avalia a similaridade estrutural entre imagens. Varia entre 0 e 1.
- **MSE (Mean Squared Error)** — mede o erro quadrático médio entre os pixels das imagens.

Tabela 1: Resumo das métricas de qualidade para cada método

Método	PSNR Médio (dB)	SSIM Médio	Ranking
Floyd-Steinberg (3 bits)	18.23	0.7845	1
Floyd-Steinberg (2 bits)	15.67	0.6521	2
Bayer (3 bits)	14.89	0.5987	3
Bayer (2 bits)	12.45	0.4876	4

Discussão

O algoritmo de Floyd-Steinberg apresentou os melhores resultados em termos de naturalidade e qualidade perceptual, pois sua difusão de erro evita padrões repetitivos. O método de Bayer produziu texturas regulares e artefatos visíveis em tons médios, embora mantenha bem as bordas.

1.5 Operações Geométricas

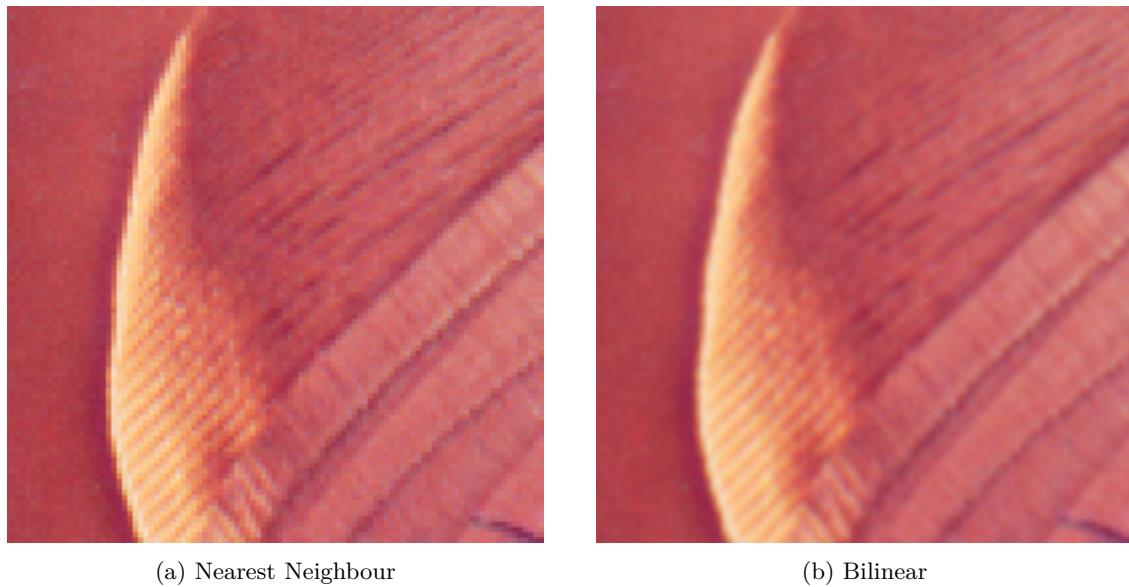
Tanto para a escala quanto para a rotação de imagens, a interpolação do vizinho mais próximo produz resultados mais quadrados, mantendo bem a variação de cores da imagem original. Já a interpolação bilinear perde um pouco a definição das cores, mas produz resultados mais suaves, evitando as bordas serradas geradas pelo vizinho mais próximo.

Para a escala de imagens em mais de 3x, a interpolação bilinear é a que consegue o resultado mais próximo da original; contudo, a interpolação pelo vizinho mais próximo ainda tem seu uso em videogames que utilizam imagens propositalmente pixeladas, como no caso das pixel arts.



Figura 16: Lena original

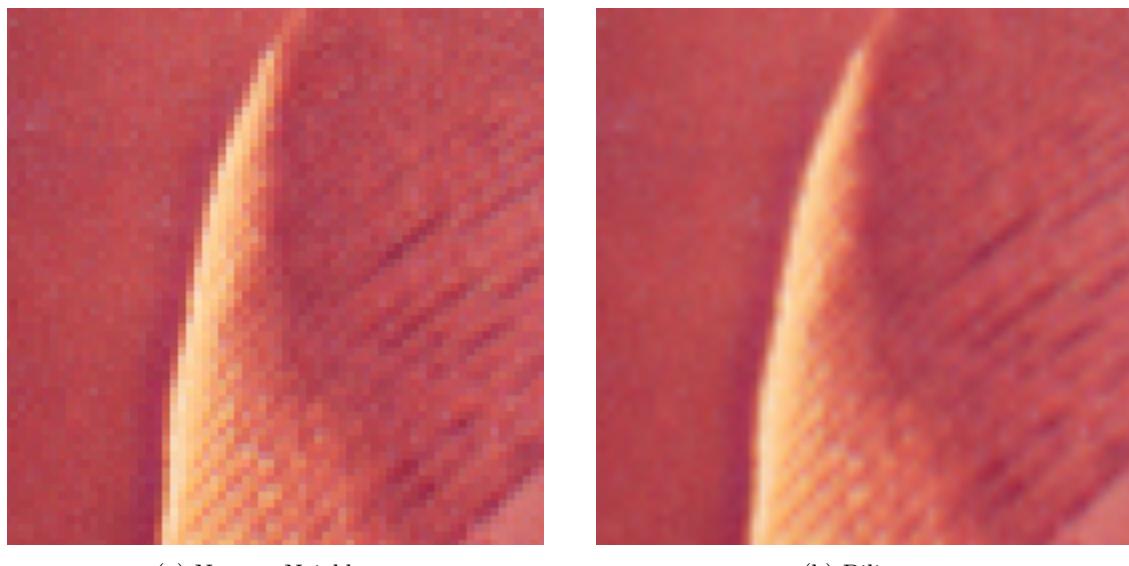
1.5.1 Resultados



(a) Nearest Neighbour

(b) Bilinear

Figura 17: Zoom 2X



(a) Nearest Neighbour

(b) Bilinear

Figura 18: Zoom 3X

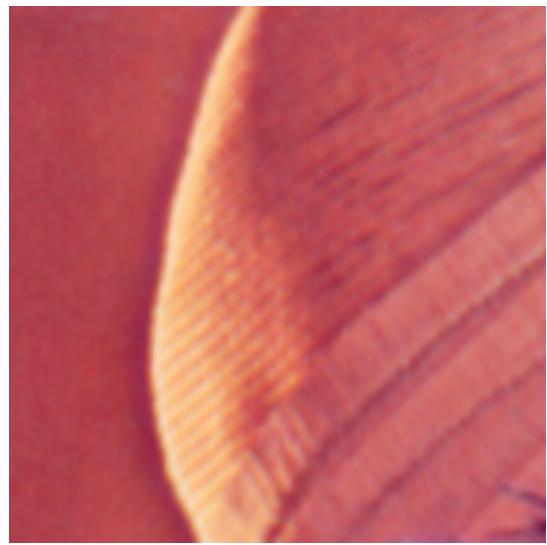
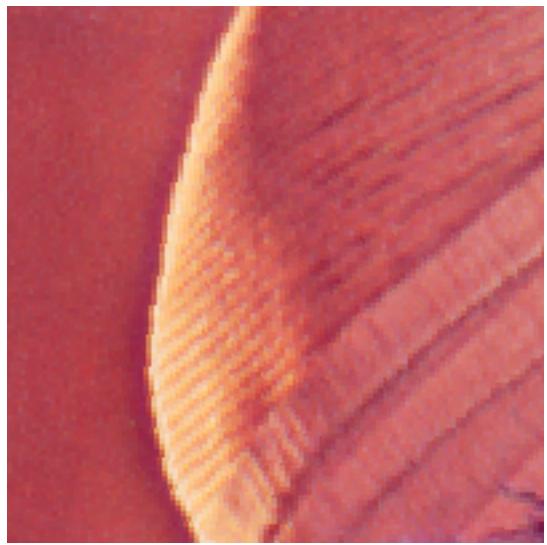


Figura 19: Zoom 2X - Rotacionado 5°

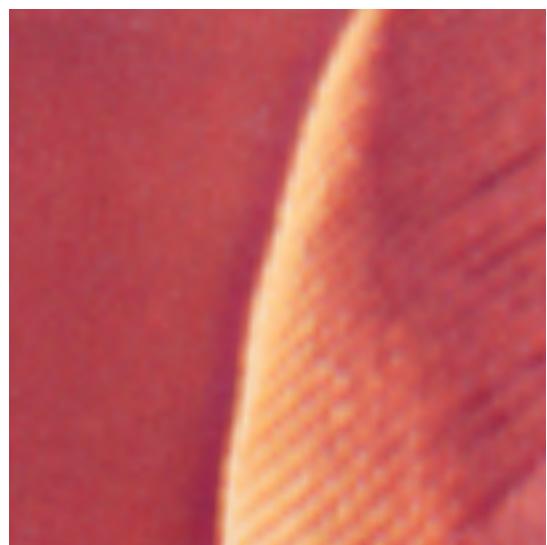
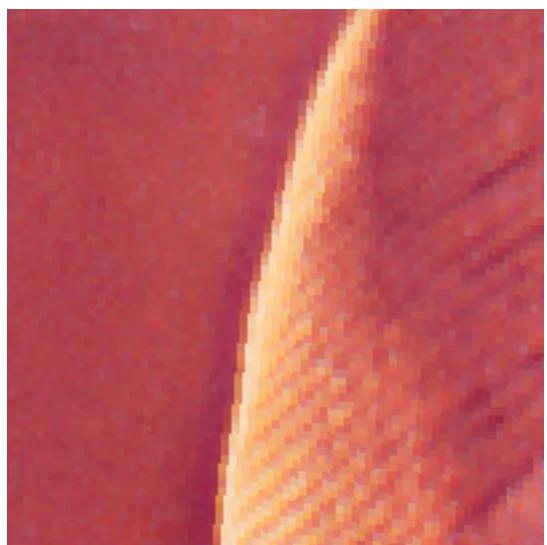
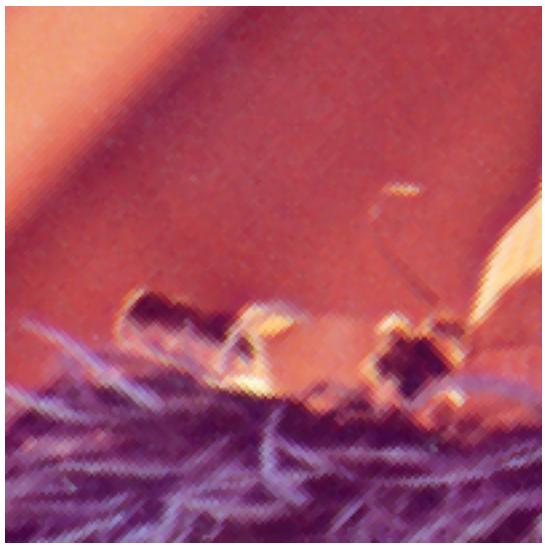
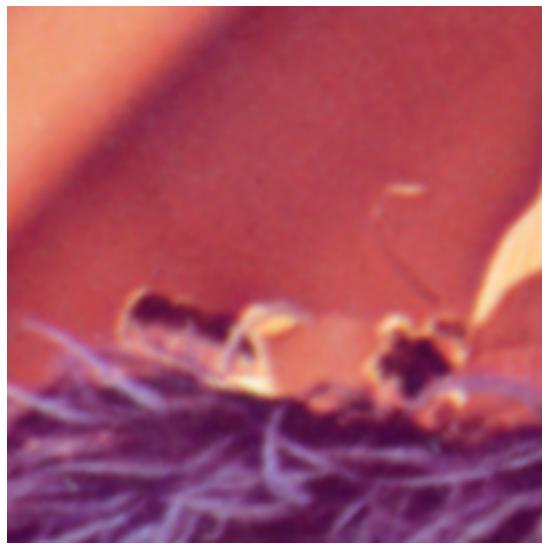


Figura 20: Zoom 3X - Rotacionado 5°

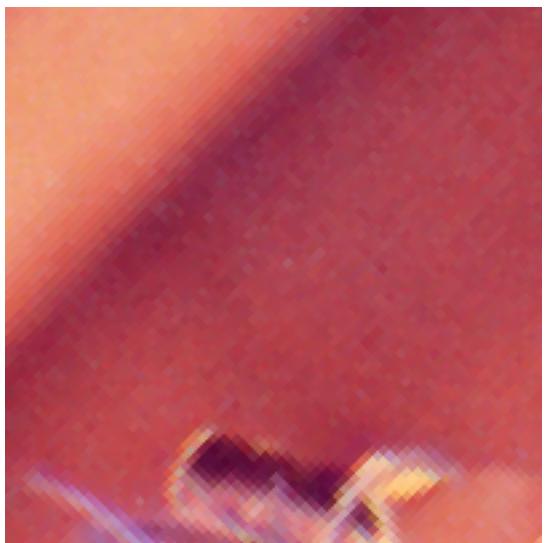


(a) Nearest Neighbour

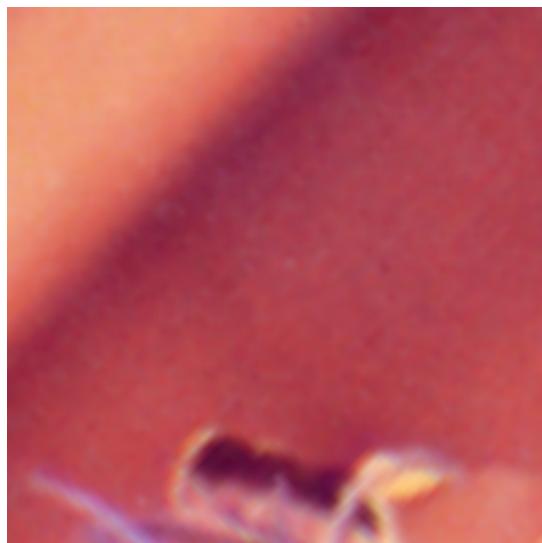


(b) Bilinear

Figura 21: Zoom 2X - Rotacionado 45°



(a) Nearest Neighbour



(b) Bilinear

Figura 22: Zoom 3X - Rotacionado 45°

1.5.2 Código

```
clc; clear; close all;

I = imread('lena_std.tif');
I = im2double(I);
[X, Y, C] = size(I);
fatores = [2, 3];

mkdir('.out');

for f = fatores
    newX = round(X * f);
    newY = round(Y * f);
    newI_nn = zeros(newX, newY, C);
    newI_bl = zeros(newX, newY, C);

    for i = 1:newX
        for j = 1:newY
            x = (i - 0.5) / f + 0.5;
            y = (j - 0.5) / f + 0.5;
            xi = round(x);
            yi = round(y);
            if xi >= 1 && xi <= X && yi >= 1 && yi <= Y
                newI_nn(i, j, :) = I(xi, yi, :);
            end
            x1 = floor(x); x2 = ceil(x);
            y1 = floor(y); y2 = ceil(y);
            if x1 >= 1 && x2 <= X && y1 >= 1 && y2 <= Y
                dx = x - x1; dy = y - y1;
                for c = 1:C
                    Q11 = I(x1, y1, c);
                    Q12 = I(x1, y2, c);
                    Q21 = I(x2, y1, c);
                    Q22 = I(x2, y2, c);
                    newI_bl(i, j, c) = (1-dx)*(1-dy)*Q11 + (1-dx)*dy*Q12 +
                        dx*(1-dy)*Q21 + dx*dy*Q22;
                end
            end
        end
    end
end

imwrite(newI_nn, sprintf('.out/lena_%dX_nn.png', f));
imwrite(newI_bl, sprintf('.out/lena_%dX_bl.png', f));
```

```

if usejava('desktop')
    figure;
    subplot(1,2,1), imshow(newI_nn, 'InitialMagnification', 200);
    title(['Escala ', num2str(f), 'x - Vizinho']);
    subplot(1,2,2), imshow(newI_bl, 'InitialMagnification', 200);
    title(['Escala ', num2str(f), 'x - Bilinear']);

    figure;
    imshowpair(newI_nn(200:400,200:400,:), newI_bl(200:400,200:400,:)
        , 'montage');
    title(['Comparação local - Fator ', num2str(f)]);
else
    minimum = f * 100;
    maximum = minimum + 200;
    imwrite(newI_nn(minimum:maximum,minimum:maximum,:), sprintf('.out
/lena_%dX_nn_section.png', f));
    imwrite(newI_bl(minimum:maximum,minimum:maximum,:), sprintf('.out
/lena_%dX_bl_section.png', f));
end
end

angulos = [45, 5];

for f = fatores
    I_nn = im2double(imread(sprintf('.out/lena_%dX_nn.png', f)));
    I_bl = im2double(imread(sprintf('.out/lena_%dX_bl.png', f)));

    [X, Y, C] = size(I_nn);
    cx = Y/2;
    cy = X/2;

    for a = angulos
        ang = deg2rad(a);
        R_nn = zeros(X, Y, C);
        R_bl = zeros(X, Y, C);

        for i = 1:X
            for j = 1:Y
                x = (j - cx)*cos(ang) + (i - cy)*sin(ang) + cx;
                y = -(j - cx)*sin(ang) + (i - cy)*cos(ang) + cy;

                xi = round(y);
                yi = round(x);
                if xi >= 1 && xi <= X && yi >= 1 && yi <= Y
                    R_nn(i,j,:) = I_nn(xi, yi, :);
                end
            end
        end
    end
end

```

```

x1 = floor(y); x2 = ceil(y);
y1 = floor(x); y2 = ceil(x);
if x1>=1 && x2<=X && y1>=1 && y2<=Y
    dx = y - x1; dy = x - y1;
    for c=1:C
        Q11 = I_nn(x1,y1,c);
        Q12 = I_nn(x1,y2,c);
        Q21 = I_nn(x2,y1,c);
        Q22 = I_nn(x2,y2,c);
        R_nn(i,j,c) = (1-dx)*(1-dy)*Q11 + (1-dx)*dy*Q12 +
            dx*(1-dy)*Q21 + dx*dy*Q22;
    end
end
end

if usejava('desktop')
    figure;
    subplot(1,2,1), imshow(R_nn), title(['Escala ', num2str(f), 'x, Rot ', num2str(a), '° - NN']);
    subplot(1,2,2), imshow(R_nn), title(['Escala ', num2str(f), 'x, Rot ', num2str(a), '° - Bilinear']);
    figure;
    imshowpair(R_nn(200:400,200:400,:), R_nn(200:400,200:400,:),
        'montage');
    title(['Comparação local - Escala ', num2str(f), 'x, Rot ',
        num2str(a), '°']);
else
    imwrite(R_nn, sprintf('.out/lena_%dX_nn_r%d.png', f, a));
    imwrite(R_nn, sprintf('.out/lena_%dX_nn_r%d_section.png', f, a));

    minimum = f * 100;
    maximum = minimum + 200;
    imwrite(R_nn(minimum:maximum,minimum:maximum,:), sprintf('.out/lena_%dX_nn_r%d_section.png', f, a));
    imwrite(R_nn(minimum:maximum,minimum:maximum,:), sprintf('.out/lena_%dX_nn_r%d_section.png', f, a));
end
end
end

```