

Processamento de imagens

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

11/11/2025

Sumário

| | | |
|----------|---|----------|
| 1 | Introdução | 3 |
| 2 | Tarefas | 3 |
| 2.1 | Interpolação em imagens coloridas | 3 |
| 2.1.1 | Interpolação por vizinho mais próximo | 4 |
| 2.1.2 | Bilinear | 7 |
| 2.1.3 | Bicúbico | 10 |
| 2.1.4 | Funções auxiliares | 13 |
| 2.2 | Realce de imagens no domínio espacial (da imagem) | 15 |
| 2.2.1 | Arquivo principal | 15 |
| 2.3 | Filtragem espacial | 16 |
| 2.3.1 | Ruídos Salt & Pepper e Gaussian | 16 |
| 2.3.2 | Unsharp Masking | 21 |
| 2.4 | Operações Geométricas | 25 |
| 2.4.1 | Resultados | 26 |
| 2.4.2 | Código | 29 |

1 Introdução

...

2 Tarefas

2.1 Interpolação em imagens coloridas

```
%% Interpolação em imagens coloridas
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);

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

parfor i = 1:length(inimags)
    img = im2double(imread(inimags{i}));

    outimags{i} = f{i}(img, scale);
end
imgs_in_docked_figures(outimags, baseimgnames, names);
```



(a) cat



(b) hamster

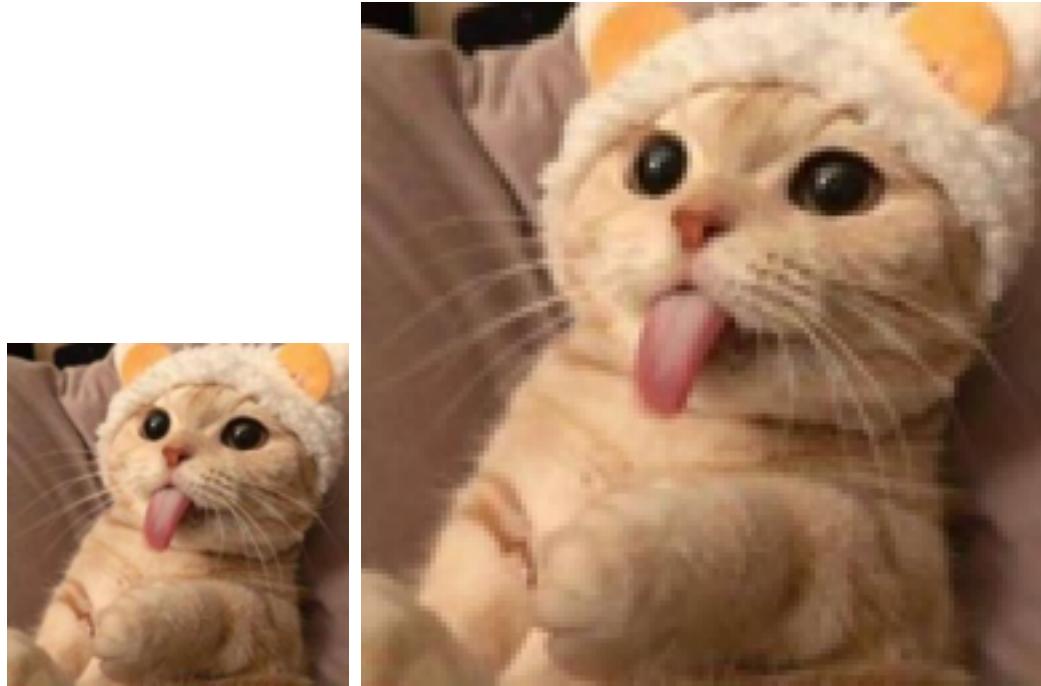
Figura 1: Imagens originais

Algoritmos

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

2.1.1 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.



(a) Original - cat

(b) Interpolado - cat

Figura 2: Vizinho mais Próximo



(a) Original - hamster

(b) Interpolado - hamster

Figura 3: Vizinho mais Próximo

2.1.1.1 Código

```

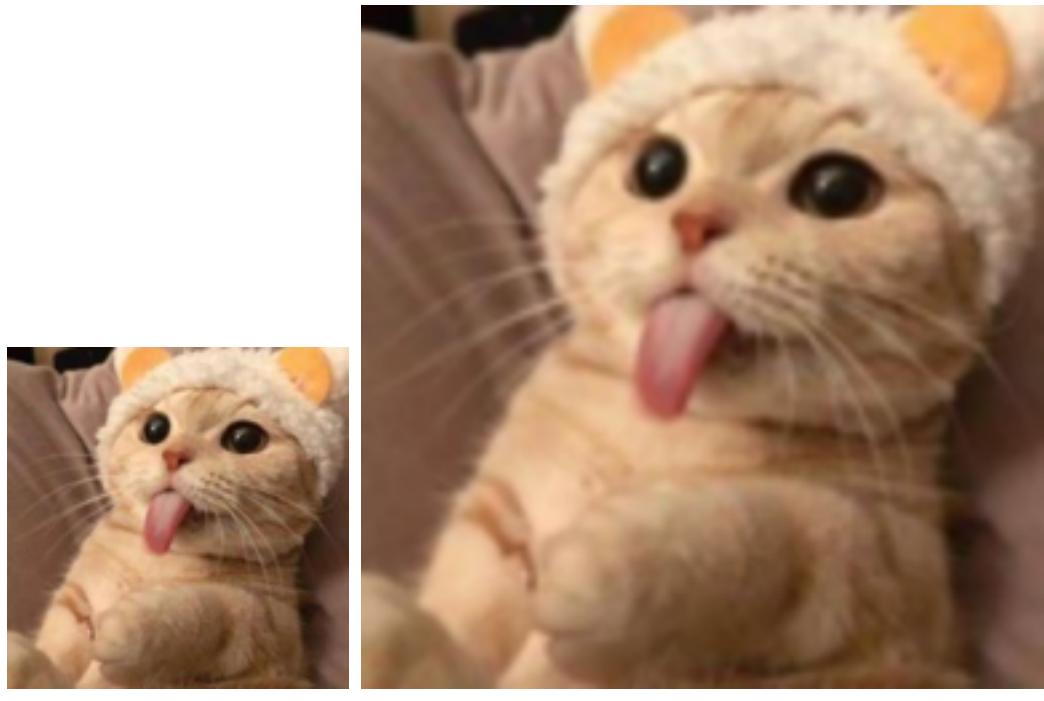
function out = nearest_neighbour_resize(img, scale)
[rows, cols, ch] = size(img);
new_rows = round(rows * scale);
new_cols = round(cols * scale);
scale = new_rows / rows;

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

```

2.1.2 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.



(a) Original - cat

(b) Interpolado - cat

Figura 4: Bilinear



(a) Original - hamster



(b) Interpolado - hamster

Figura 5: Bilinear

2.1.2.1 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, :);

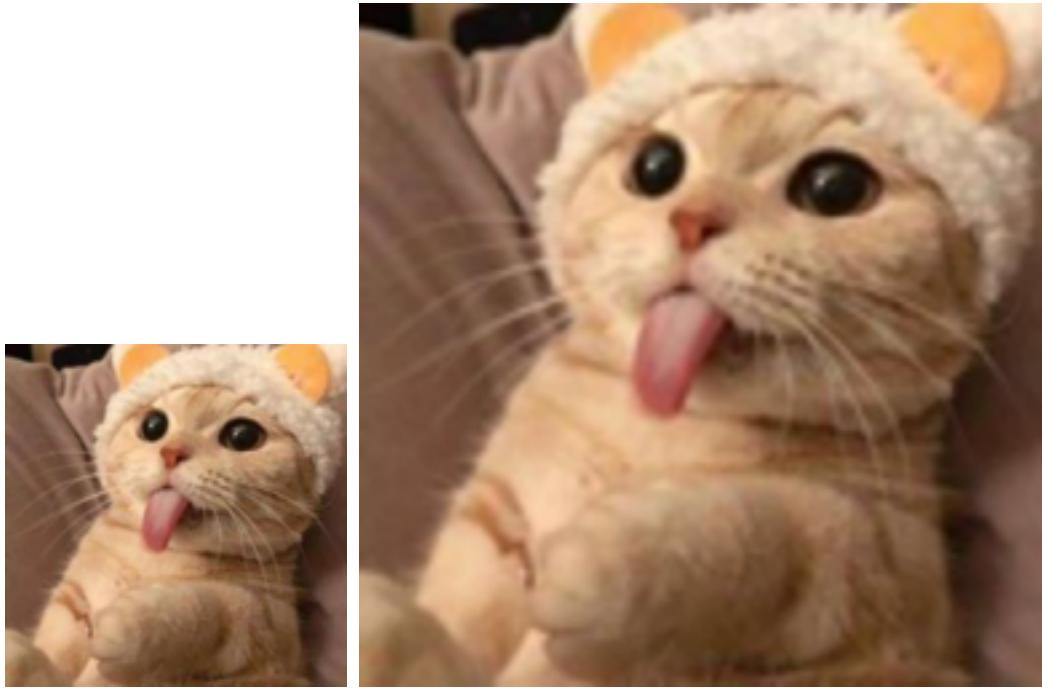
    a = repmat(a, [1, size(vals_Q1, 2)]);

    % 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
```

2.1.3 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.



(a) Original - cat

(b) Interpolado - cat

Figura 6: Bicúbico



(a) Original - hamster



(b) Interpolado - hamster

Figura 7: Bicúbico

2.1.3.1 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

    [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(@ordivide, wx, sum(wx, 3));
        wy = bsxfun(@ordivide, 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
end
```

2.1.4 Funções auxiliares

```
function out = at_dim(n_dims, dim, i)
    out = repmat({':'}, 1, n_dims);
    out{dim} = i;
end

function out = get_at_dim(src, dim, i)
    at = repmat({':'}, 1, ndims(src));
    at{dim} = i;
    out = subsref(src, substruct('()', at));
end

function out = set_at_dim(dst, dim, i, src)
    at = repmat({':'}, 1, ndims(dst));
    at{dim} = i;
    out = subsasgn(dst, substruct('()', at), src);
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

```

```

%% Interpolação em imagens coloridas
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);

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

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

```

2.2 Realce de imagens no domínio espacial (da imagem)

2.2.1 Arquivo principal

```
% Realce de imagens no domínio espacial (da imagem)
```

2.3 Filtragem espacial

2.3.1 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.

```

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.

```

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);
end

```



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.

```



```

| | 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 |

2.3.2 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

```
imgs = {'borboleta.jpg', 'raposa.jpg'};
sigmas = [2.5, 2.5, 0.5];
amounts = [0.5, 2.5, 2.5];
for i = 1:length(imgs)
    % Leitura da imagem original (colorida)

    name = imgs{i};
    base_name = strrep(name, '.jpg', '');
    img = im2double(imread(name));

    for j = 1:length(sigmas)
        %% Parâmetros do filtro Gaussiano
        % desvio padrão da Gaussiana (controla suavização)
        sigma = sigmas(j);
        % fator de realce
        amount = amounts(j);

        %% Criar imagem suavizada
        % filtro Gaussiano 5x5
        h = fspecial('gaussian', [5 5], sigma);
        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('Realce com Unsharp Masking');
        else
            mkdir('.out');
            img_id = strrep(sprintf('%.1f %.1f', sigma, amount), '.', '_');
            img_id = img_id + '_';
            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.

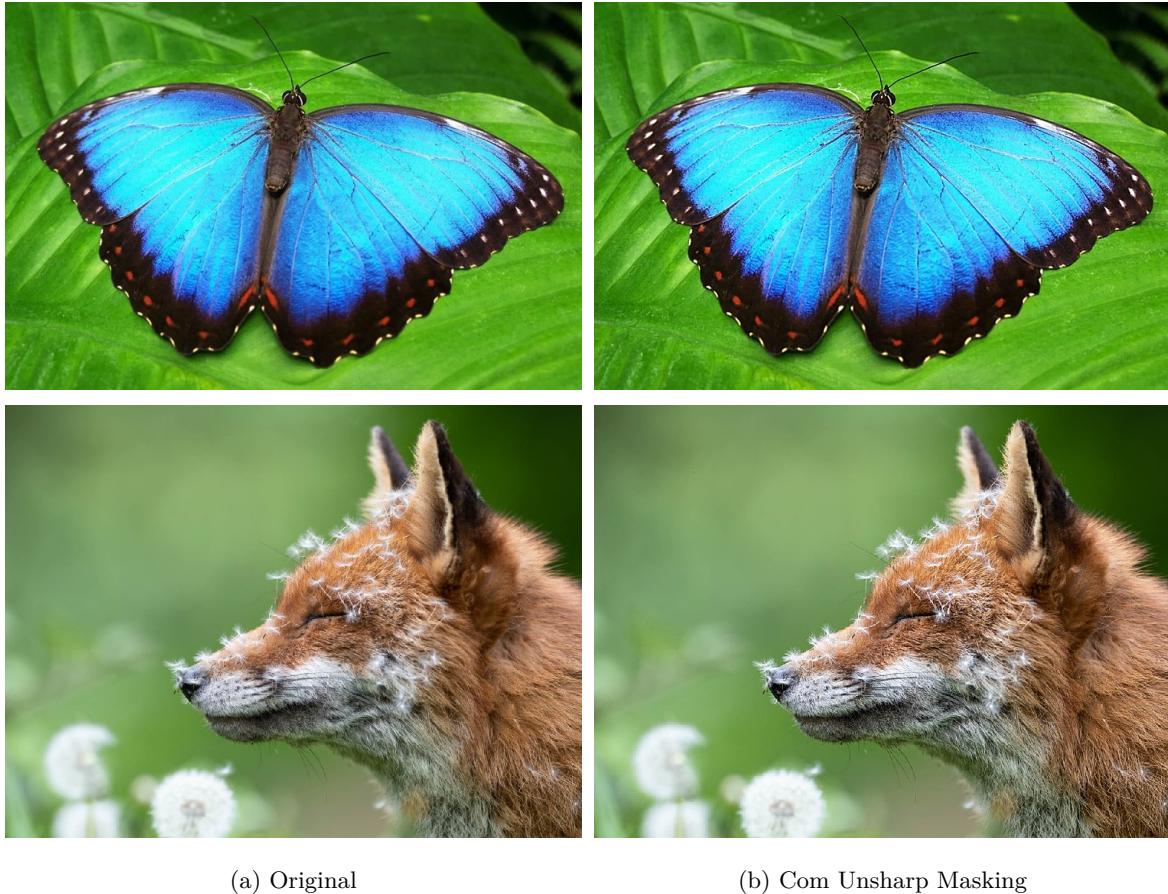


Figura 11: $\sigma=2.5$ e $\text{amount}=0.5$

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: $\text{sigma}=2.5$ 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.

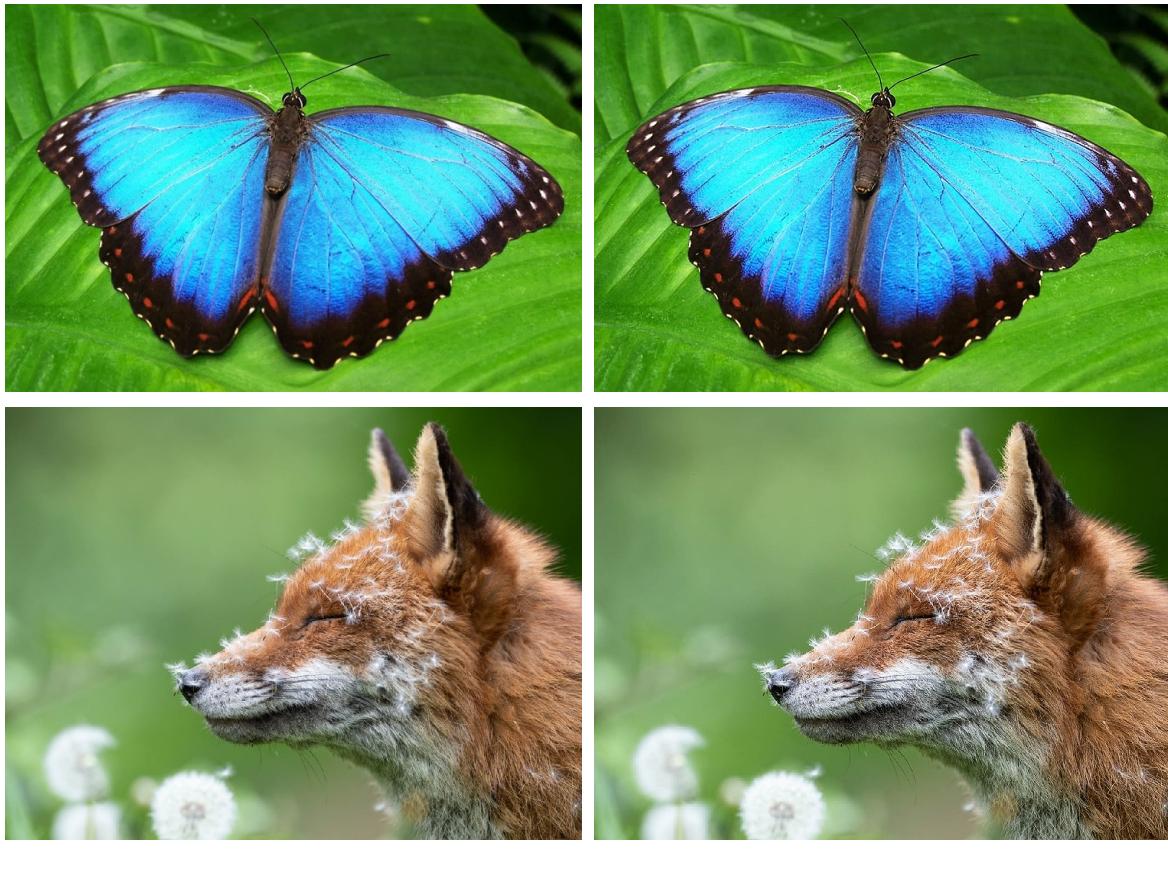


Figura 13: sigma=0.5 e amount=2.5

2.4 Operações Geométricas

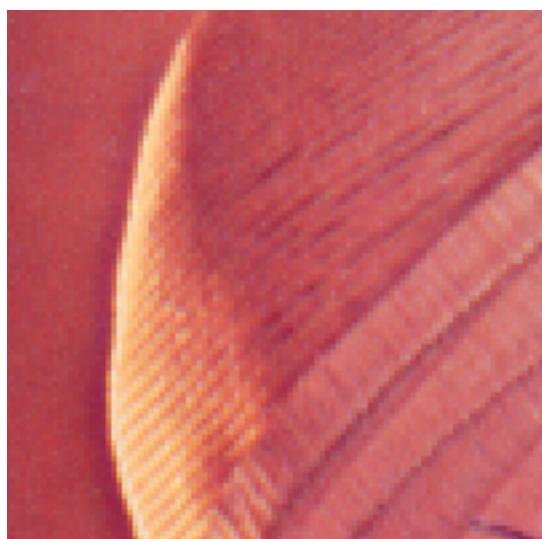
Tanto para a escala quanto para a rotação de imagens, a interpolação do vizinho mais próximo produz resultados mais quadrados, mantendo bem a variação de cores da imagem original. Já a interpolação bilinear perde um pouco a definição das cores, mas produz resultados mais suaves, evitando as bordas serradas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

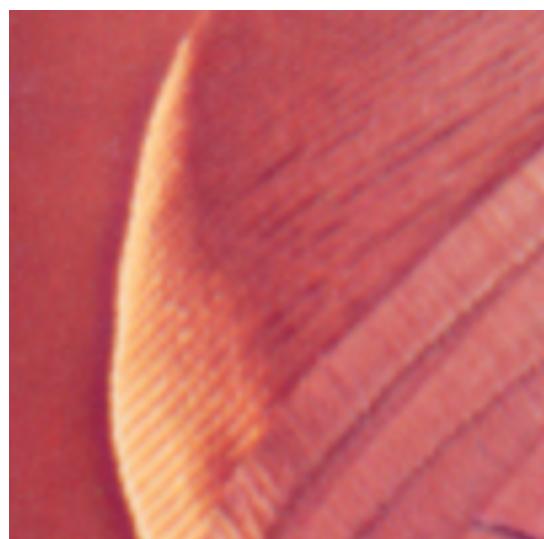
2.4.1 Resultados



Figura 14: Lena original

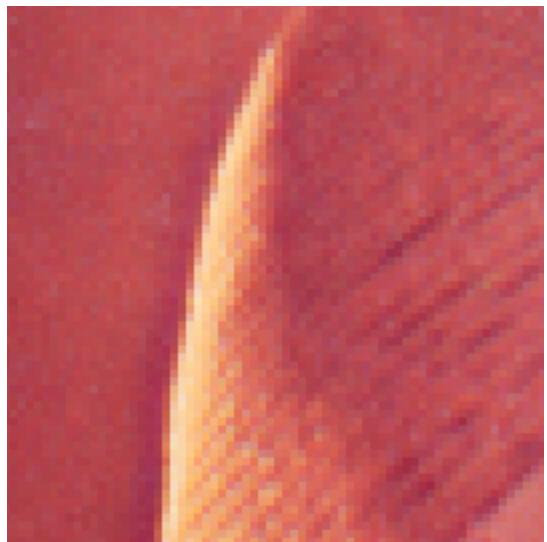


(a) Nearest Neighbour

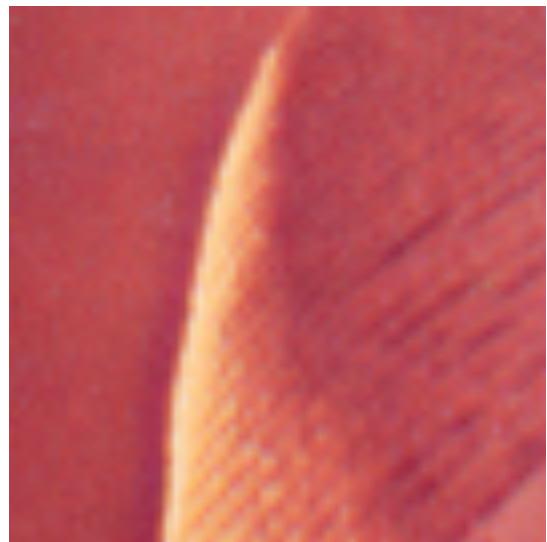


(b) Bilinear

Figura 15: Zoom 2X

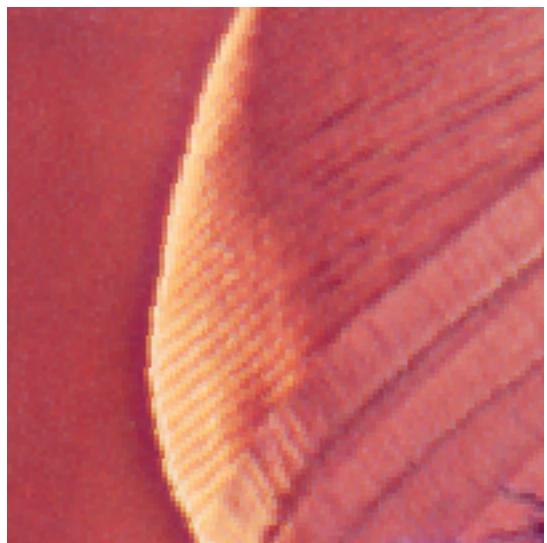


(a) Nearest Neighbour

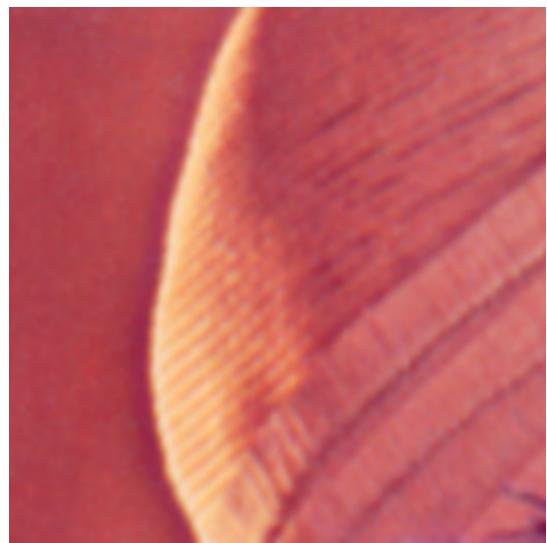


(b) Bilinear

Figura 16: Zoom 3X

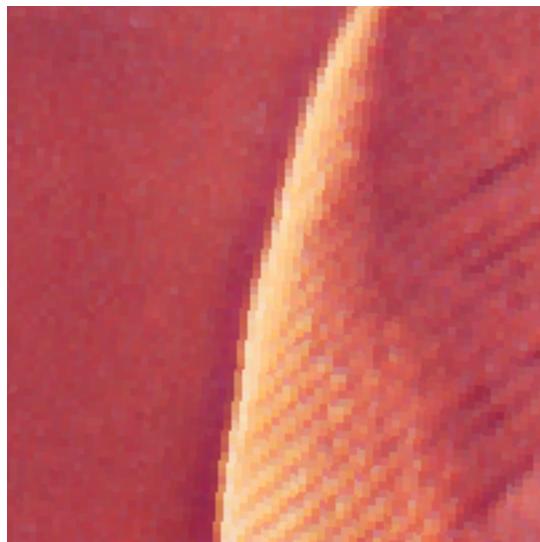


(a) Nearest Neighbour

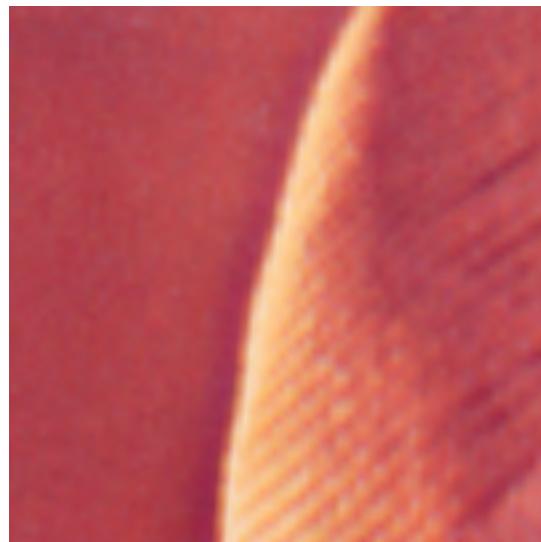


(b) Bilinear

Figura 17: Zoom 2X - Rotacionado 5°



(a) Nearest Neighbour

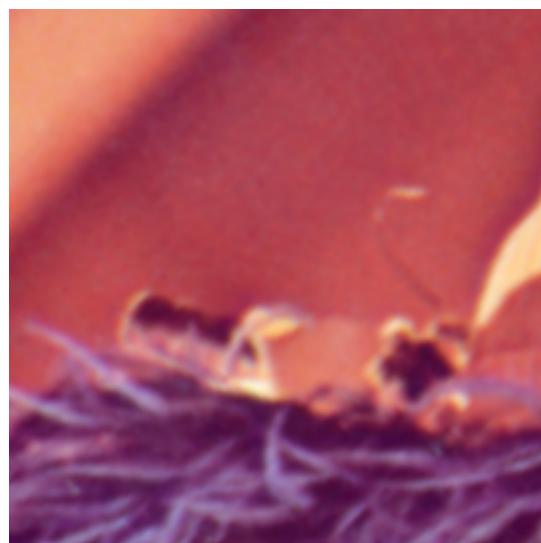


(b) Bilinear

Figura 18: Zoom 3X - Rotacionado 5°

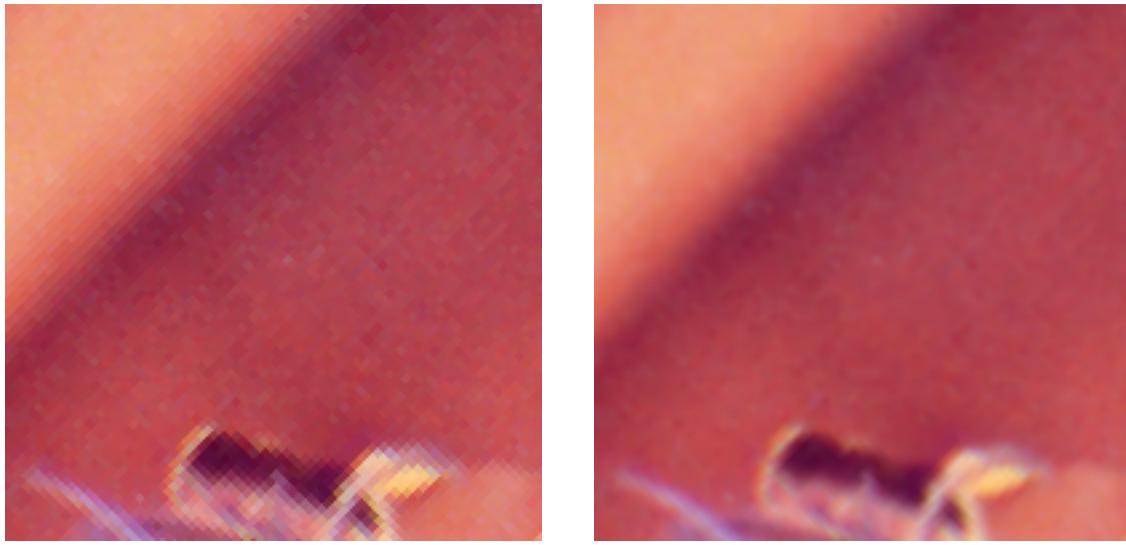


(a) Nearest Neighbour



(b) Bilinear

Figura 19: Zoom 2X - Rotacionado 45°



(a) Nearest Neighbour

(b) Bilinear

Figura 20: Zoom 3X - Rotacionado 45°

2.4.2 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
        end
    end
end

```

```

        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_nn(i,j,c) = (1-dx)*(1-dy)*Q11 + (1-dx)*dy*Q12 +
                    dx*(1-dy)*Q21 + dx*dy*Q22;
            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_b1 = im2double(imread(sprintf('.out/lena_%dX_b1.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_b1 = 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

            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_b1(x1,y1,c);
                    Q12 = I_b1(x1,y2,c);
                    Q21 = I_b1(x2,y1,c);
                    Q22 = I_b1(x2,y2,c);
                    R_b1(i,j,c) = (1-dx)*(1-dy)*Q11 + (1-dx)*dy*Q12 +
                        dx*(1-dy)*Q21 + dx*dy*Q22;
                end
            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_b1), title(['Escala ', num2str(f), ' x, Rot ', num2str(a), ' ° - Bilinear']);
end

```

```

figure;
imshowpair(R_nn(200:400,200:400,:), R_bl(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_bl, sprintf('.out/lena_%dX_bl_r%d.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_bl(minimum:maximum,minimum:maximum,:), sprintf('.%
out/lena_%dX_bl_r%d_section.png', f, a));
end
end
end

```