

Universidade Estadual de Campinas
MC970 - Introdução ao Processamento de Imagem Digital
Prof. Dr. Hélio Pedrini

Trabalho 4

Mateus de Lima Almeida
242827

13 de junho de 2025

1 Introdução

O presente trabalho aborda a implementação, realização e exploração de algoritmos para transformações e análises de imagens, com foco tanto em operações geométricas quanto na extração de propriedades de regiões. Com o objetivo de introduzir, solidificar e explorar o conhecimento aos conceitos de técnicas de transformação de imagens por meio da utilização de interpolações como vizinho mais próximo, bilinear, bicúbica e Lagrange, e de análise de objetos, com base em medidas como área, perímetro, excentricidade e solidez. Neste presente relatório, imagens foram rotacionadas, escaladas, segmentadas, rotuladas, e seus resultados foram analisados.

Este relatório detalha a metodologia adotada na execução dos exercícios sugeridos e também realiza a discussão de seus resultados, bem como a exploração de hipóteses surgidas no processo, junto também com a discussão de seus testes e resultados.

2 Materiais e Métodos

Para todas as tarefas a seguir, os materiais utilizados foram a linguagem Python 3.9 junto com as bibliotecas OpenCV2, Scikit-Image, Numpy, Pillow e Matplotlib. Para utilização das bibliotecas e execução dos códigos, também foi utilizado um ambiente virtual (venv), onde foram instaladas as bibliotecas citadas.

2.1 Organização de Arquivos

A pasta do projeto foi organizada de modo a facilitar a visualização dos resultados de cada exercício de maneira independente. Dentro da pasta estão:

- Arquivo Python da parte 1 *classifica.py*
- Arquivo Python da parte 2 *transforma.py*
- Arquivo *requirements.txt* com as dependências necessárias do projeto.
- Pasta *images* que contém:
 - Imagens PNG que serão os Inputs dos exercícios
 - Pasta *outputs* que contém 2 pastas (parte1 e parte2) contendo as imagens-saída de cada modo.

2.2 Versão Python e Bibliotecas

A versão de Python utilizada, junto com as bibliotecas e suas versões foram as seguintes:

- Python 3.9.1
- Pillow 11.1.0
- Opencv-python 4.11.0.86
- Numpy 2.0.2
- Matplotlib 3.9.4

2.3 Execução de classifica.py

Os códigos feitos para realizar as operações e transformações sugeridas neste trabalho foram todos executados dentro de um ambiente virtual *venv*, então é recomendável que para a reprodução seja feito o mesmo. É necessário que as dependências de *requirements.txt* sejam instaladas. Para executar o código, basta utilizar o seguinte comando no terminal, dentro de um ambiente virtual com as dependências instaladas:

```
python classifica.py
```

Ao ser executado, o programa solicitará ao usuário que informe o nome da imagem de entrada (sem a extensão *t.png*). A imagem deve estar localizada na pasta *images*. As imagens disponíveis para análise são:

- *objetos1.png*
- *objetos2.png*
- *objetos3.png*

As saídas geradas são salvas automaticamente na pasta *images/outputs/parte1/* com os seguintes nomes:

- *img_gray.png* — imagem em tons de cinza
- *img_bin.png* — imagem binarizada
- *img_contornos.png* — imagem com contornos desenhados
- *img_rotulada.png* — imagem original com rótulos das regiões
- *hist_area.png* — histograma da área das regiões

Além das imagens, o programa imprime no terminal o número total de regiões detectadas, bem como suas propriedades geométricas (área, perímetro, excentricidade e solidez), e a contagem de regiões pequenas, médias e grandes. Caso novas imagens sejam inseridas na pasta *images*, elas poderão ser utilizadas normalmente na análise, bastando informar o nome ao executar o script.

2.4 Execução de transforma.py

Os códigos feitos para realizar as operações e transformações sugeridas neste trabalho foram todos executados dentro de um ambiente virtual *venv*, então é recomendável que para a reprodução seja feito o mesmo. É necessário que as dependências de *requirements.txt* sejam instaladas.

Para executar o programa, basta inserir o comando básico para rodar um arquivo *.py* no terminal (certifique-se de estar na pasta correta). O comando básico é o seguinte:

```
python transforma.py -i nome_entrada -o nome_saida -a ANGULO -m METODO
```

ou, no caso de transformação por escala:

```
python transforma.py -i nome_entrada -o nome_saida -e ESCALA -m METODO
```

Os nomes de entrada (`-i`) e saída (`-o`) devem ser passados sem a extensão `.png`, pois ela é adicionada automaticamente pelo código. A imagem de entrada deve estar localizada na pasta `images`, e a imagem de saída será salva na pasta `images/outputs/parte2/`. Abaixo estão listadas as imagens disponíveis atualmente na pasta `images`, que podem ser usadas como entrada:

- `house.png`
- `baboon_monocromatica.png`
- `seagull.png`

Os métodos de interpolação disponíveis para o argumento `-m` são:

- `vizinho` (interpolação por vizinho mais próximo)
- `bilinear` (interpolação bilinear)
- `bicubica` (interpolação bicúbica)
- `lagrange` (interpolação de Lagrange)

Para a rotação, o argumento `-a` deve ser fornecido em graus (ex: 30, -45, etc.). Para escala, o argumento `-e` deve ser um fator real positivo (ex: 1.5 para aumentar, 0.5 para reduzir).

Exemplos de uso:

```
python transforma.py -i house -o house_rot30 -a 30 -m bicubica
python transforma.py -i seagull -o seagull_escala2 -e 2.0 -m lagrange
```

Por fim, caso novas imagens sejam adicionadas à pasta `images`, elas podem ser usadas diretamente como entrada, sem necessidade de alterar o código. O nome definido em `-o` será utilizado como base para salvar o arquivo de saída, sempre com a extensão `.png`.

3 Resultados, testes e discussão

3.1 Medição e classificação de objetos

O objetivo da primeira parte é realizar a extração de medidas geométricas de objetos presentes em uma imagem, como área, perímetro, excentricidade e solidez, além de classificá-los com base na área. Para isso, uma série de transformações e análises são realizadas, conforme descrito a seguir.

A primeira imagem usada para testar o método é a `objetos1.png` (Figura 1).

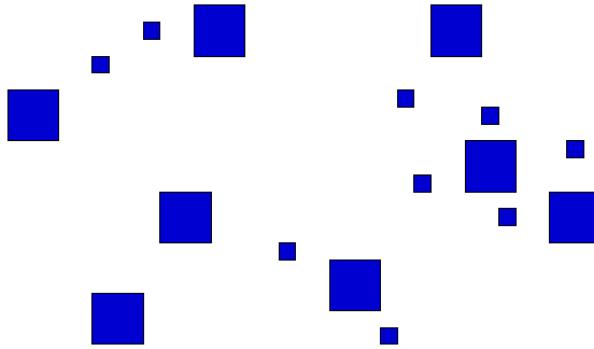


Figura 1: objetos1.png

O primeiro passo do processo consiste na leitura da imagem colorida e posterior conversão para tons de cinza. Esta etapa é necessária pois as operações de segmentação e extração de propriedades são mais simples em imagens monocromáticas. A conversão para tons de cinza foi realizada utilizando a função `cvtColor` da biblioteca OpenCV com o código `cv2.COLOR_RGB2GRAY`. O resultado pode ser observado na Figura 2.

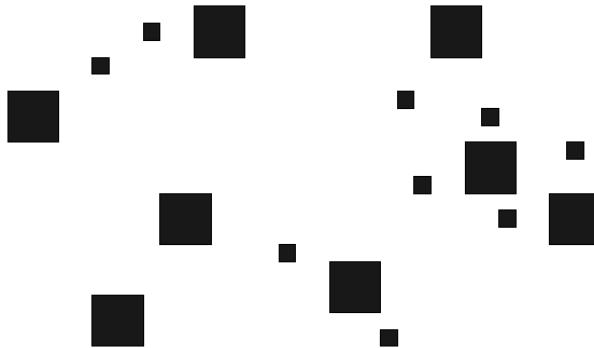


Figura 2: Imagem convertida para tons de cinza

Em seguida, a imagem em tons de cinza foi binarizada utilizando um limiar fixo de 250. Essa binarização transforma os pixels claros (brancos) em fundo e os pixels escuros (os objetos) em primeiro plano. Como a imagem original possui fundo branco, a inversão é necessária para que os objetos sejam destacados como regiões ativas. A função `threshold` do OpenCV foi utilizada com o modo `cv2.THRESH_BINARY_INV`.



Figura 3: Imagem binarizada com fundo invertido

Com a imagem binarizada, foi possível detectar os contornos dos objetos presentes na cena. Para isso, foi utilizada a função `findContours` da biblioteca OpenCV. Os contornos foram desenhados em uma nova imagem com fundo branco, utilizando a função `drawContours`. O resultado pode ser visto na Figura 4.

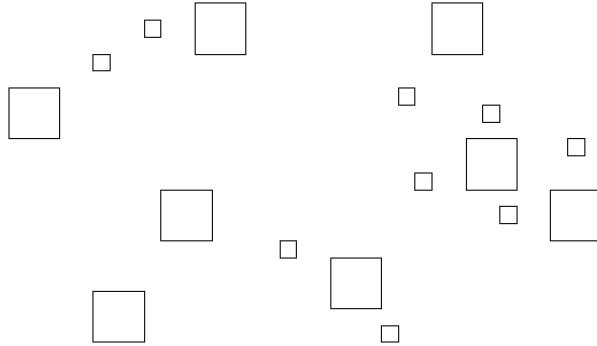


Figura 4: Contornos dos objetos

Após detectar os contornos, foi feita uma limpeza da imagem para remover pequenos componentes que não representam objetos relevantes. Essa etapa foi feita com a função `remove_small_objects` da biblioteca `skimage.morphology`, considerando objetos com menos de 20 pixels como ruído.

As regiões restantes foram então rotuladas com a função `measure.label`, que atribui um identificador numérico para cada componente conectado. As propriedades geométricas dos objetos, como área, perímetro, excentricidade e solidez, foram extraídas utilizando `regionprops`. As medidas foram impressas no terminal, conforme solicitado.

Para facilitar a visualização das regiões identificadas, foi criada uma imagem colorida a partir da imagem original, com os números dos rótulos desenhados sobre o centróide de cada objeto. Isso foi feito com a função `putText` da OpenCV. A imagem resultante pode ser vista na Figura 9.

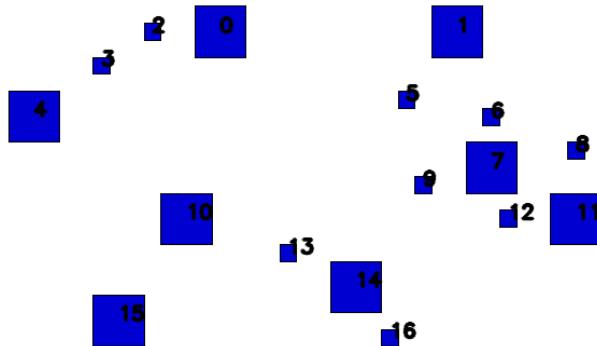


Figura 5: Objetos rotulados com seus respectivos índices

Além da extração de propriedades, os objetos também foram classificados com base em sua área. Os critérios de classificação são:

- **Pequeno:** área < 1500 pixels
- **Médio:** $1500 \leq$ área < 3000 pixels
- **Grande:** área ≥ 3000 pixels

As quantidades de objetos em cada faixa foram impressas no terminal. Por fim, foi gerado um histograma com as áreas dos objetos detectados, mostrado na Figura 6.

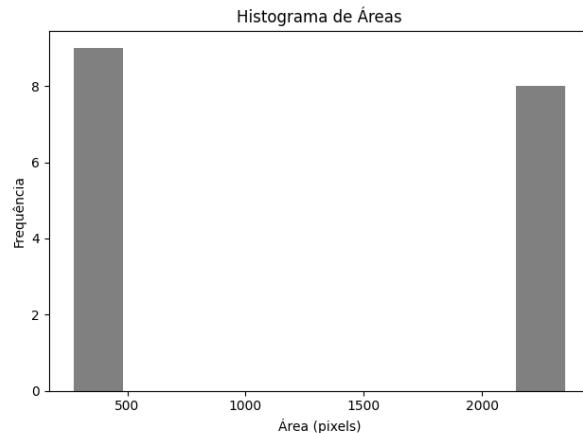


Figura 6: Histograma das áreas dos objetos detectados

Além das imagens produzidas pelo código, também há a saída em formato de texto que realiza a contagem dos objetos/regiões na imagem. Para *objetos1.png*, o código apresentou as seguintes saídas:

Categoría	Número de Regiões
Pequenas (área < 1500)	9
Médias (1500 ≤ área < 3000)	8
Grandes (área ≥ 3000)	0
Total	17

Tabela 1: Classificação das regiões com base na área

Região	Área	Perímetro	Excentricidade	Solidez
0	2352.0	190.000000	0.201039	1.000000
1	2352.0	190.000000	0.201039	1.000000
2	272.0	62.000000	0.338502	1.000000
3	272.0	62.000000	0.338502	1.000000
4	2304.0	188.000000	0.000000	1.000000
5	272.0	62.000000	0.338502	1.000000
6	289.0	64.000000	0.000000	1.000000
7	2352.0	190.000000	0.201039	1.000000
8	289.0	64.000000	0.000000	1.000000
9	289.0	64.000000	0.000000	1.000000
10	2352.0	190.000000	0.201039	1.000000
11	2352.0	190.000000	0.201039	1.000000
12	289.0	64.000000	0.000000	1.000000
13	272.0	62.000000	0.338502	1.000000
14	2304.0	188.000000	0.000000	1.000000
15	2352.0	190.000000	0.201039	1.000000
16	272.0	62.000000	0.338502	1.000000

Tabela 2: Propriedades geométricas das 17 regiões detectadas

As propriedades geométricas das regiões rotuladas foram obtidas utilizando a função `regionprops`, da biblioteca `skimage.measure`. A Tabela 6 apresenta os valores obtidos para cada uma das 17 regiões detectadas, enquanto a Tabela 5 resume a quantidade de regiões em cada faixa de classificação de área (pequena, média e grande).

A área corresponde ao número de pixels pertencentes a uma determinada região e é obtida diretamente pelo atributo `area` de cada objeto retornado por `regionprops`. O perímetro representa a soma das distâncias entre os pixels que compõem a borda da região e é fornecido pelo atributo `perimeter`, que calcula esse valor considerando a forma contínua do contorno. A excentricidade, extraída por meio do atributo `eccentricity`, indica o alongamento da região e é definida como a excentricidade da elipse com os mesmos momentos centrais que a região analisada. Seu valor varia entre 0 (forma perfeitamente circular) e 1 (forma completamente alongada). Por fim, a solidez é uma medida da compacidade da região e representa a razão entre sua área e a área do seu casco convexo. Essa medida é obtida com o atributo `solidity` e apresenta valor máximo igual a 1 quando a região é perfeitamente convexa, decrescendo conforme a forma da região se torna mais irregular ou fragmentada.

As próximas imagens testadas serão `objetos2.png` e `objetos3.png` (Figuras 7 e 8).

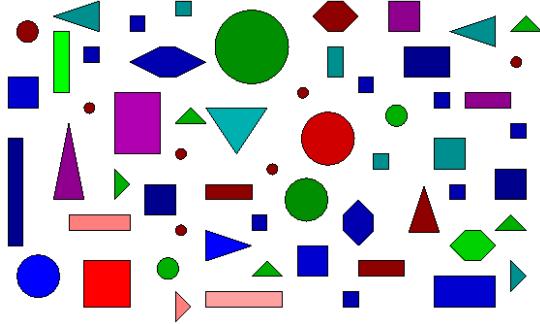


Figura 7: objetos2.png



Figura 8: objetos3.png

Para poder observar o comportamento da implementação em mais instâncias e validar seu funcionamento, é importante utilizar imagens de teste com diferentes características. A imagem *objetos1.png* contém apenas quadrados azuis de tamanhos variados, sendo útil para verificar se o código consegue distinguir corretamente as regiões com base exclusivamente na área. No entanto, ela não permite avaliar a extração de propriedades mais complexas, como excentricidade ou solidez, já que todos os objetos possuem a mesma forma. Por outro lado, as imagens *objetos2.png* e *objetos3.png* contêm objetos com formas, tamanhos e cores variadas, permitindo explorar melhor as capacidades do código, como o cálculo de excentricidade (para objetos alongados ou circulares) e a solidez (para regiões côncavas ou irregulares).

A seguir, é possível ver os resultados da aplicação na imagem *objetos2.png* (as imagens intermediárias de contorno, binarização e outras não serão mostradas por uma questão de espaço e para evitar redundâncias, já que os resultados são idênticos aos da imagem passada, mas com as novas formas).

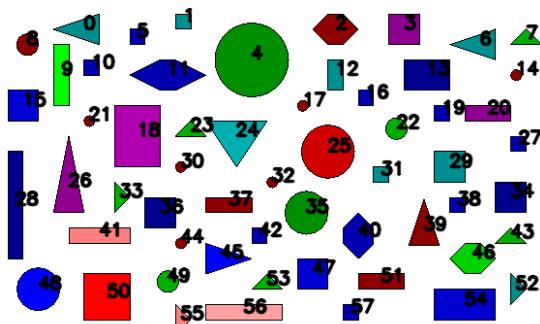


Figura 9: Objetos rotulados com seus respectivos índices

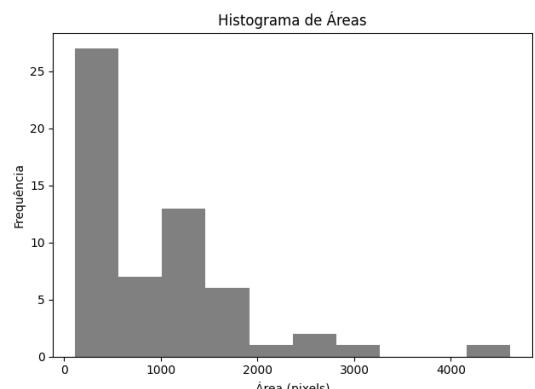


Figura 10: Histograma das áreas dos objetos detectados

Para *objetos2.png*, o código apresentou as seguintes saídas:

Categoría	Número de Regiões
Pequenas (área < 1500)	47
Médias ($1500 \leq$ área < 3000)	9
Grandes (área ≥ 3000)	2
Total	58

Tabela 3: Classificação das 58 regiões detectadas com base na área

Por uma questão de espaço (a tabela original de regiões de *objetos2.png* possui mais de 50 linhas), a 4 se trata apenas das 20 primeiras linhas, ou seja, das 20 primeiras regiões da imagem.

Região	Área	Perímetro	Excentricidade	Solidez
0	816.0	138.012	0.816	0.965
1	272.0	62.000	0.339	1.000
2	1024.0	118.853	0.632	1.000
3	1056.0	126.000	0.244	1.000
4	4616.0	251.279	0.066	0.987
5	272.0	62.000	0.339	1.000
6	785.0	136.219	0.816	0.953
7	289.0	77.255	0.816	1.000
8	419.0	73.941	0.116	0.970
9	1088.0	158.000	0.964	1.000
10	289.0	64.000	0.000	1.000
11	1552.0	181.439	0.892	0.963
12	544.0	94.000	0.848	1.000
13	1536.0	156.000	0.746	1.000
14	113.0	36.971	0.154	0.950
15	1056.0	126.000	0.244	1.000
16	272.0	62.000	0.339	1.000
17	113.0	36.385	0.293	0.974
18	3072.0	220.000	0.662	1.000
19	289.0	64.000	0.000	1.000

Tabela 4: Propriedades geométricas de exemplo para 20 das 58 regiões detectadas

Observevando os resultados acima, é possível observar que, diferentemente da primeira imagem, que só possuía formas semelhantes a quadrados (e com uma variação de tamanhos bem limitada), resultando apenas em valores específicos de excentricidade e solidez, a análise dessa imagem com mais variações resulta (assim como esperado) em valores mais variados dessas duas medidas.

A seguir, o mesmo será feito com *objetos3.png*.



Figura 11: Objetos rotulados com seus respectivos índices

Para *objetos3.png*, o código apresentou as seguintes saídas:

Categoría	Número de Regiões
Pequenas (área < 1500)	5
Médias (1500 ≤ área < 3000)	1
Grandes (área ≥ 3000)	3
Total	9

Tabela 5: Classificação das 9 regiões com base na área

Região	Área	Perímetro	Excentricidade	Solidez
0	3969.0	313.765	0.816	0.747
1	791.0	119.983	0.741	0.896
2	3584.0	259.463	0.898	0.976
3	540.0	99.255	0.890	0.902
4	438.0	88.770	0.856	0.909
5	1684.0	174.125	0.868	0.967
6	642.0	103.012	0.890	0.963
7	3934.0	305.421	0.911	0.773
8	675.0	96.326	0.620	0.970

Tabela 6: Propriedades geométricas das 9 regiões detectadas

É interessante notar também, que os resultados obtidos nesse último experimento com a imagem *objetos3.png* estão consistentes com os números apresentados no documento inicial proposto a atividade. Isso mostra não só que a implementação apresenta resultado satisfatório, mas que ela também funciona corretamente com formas menos convencionais (a região não convexa de número 0, por exemplo).

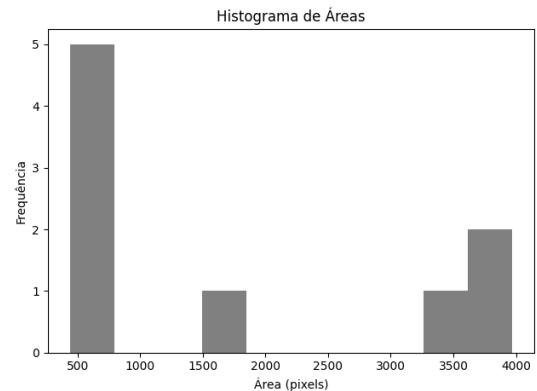


Figura 12: Histograma das áreas dos objetos detectados

3.2 Transformações geométricas com interpolação

A segunda parte do trabalho consistiu na aplicação de transformações geométricas (rotação e escala) utilizando diferentes métodos de interpolação. Essas transformações foram aplicadas sobre imagens em tons de cinza.

O código desenvolvido recebe como parâmetros principais: a imagem de entrada (**-i**), a imagem de saída (**-o**), o ângulo de rotação (**-a**), o fator de escala (**-e**) e o método de interpolação (**-m**).

O argumento **-m** define o método de interpolação a ser utilizado durante a transformação. Foram implementados quatro métodos distintos: vizinho mais próximo, interpolação bilinear, interpolação bicúbica e interpolação de Lagrange.

A imagem escolhida para realizar os testes e comparação entre os diferentes tipos de interpolação foi a imagem *baboon_monocromatica.png* (Figura 13).

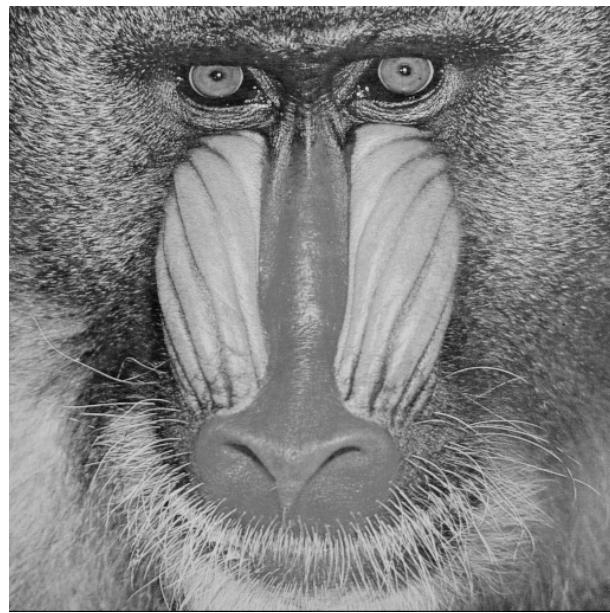


Figura 13: *baboon_monocromatica.png*

A seguir, os métodos de interpolação serão explicados, e os resultados serão mostrados utilizando a imagem.

3.2.1 Vizinho mais próximo

```
def interpolacao_vizinho(img, x, y):
    xi, yi = int(round(x)), int(round(y))
    if 0 <= xi < w and 0 <= yi < h:
        return img[yi, xi]
    return 0
```

Esse tipo de interpolação simplesmente arredonda as coordenadas reais (x, y) para os inteiros mais próximos (x_i, y_i) , retornando o valor do pixel nessa posição. É o tipo mais simples e rápido de interpolação, mas tende a gerar bordas serrilhadas e uma imagem de menor qualidade geral.

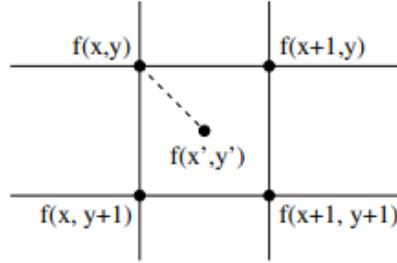


Figura 14: Exemplo de transformação utilizando interpolação por vizinho mais próximo

A seguir, está o resultado da aplicação de uma operação de escala com valor de escala maior que 1 (ampliação). O comando utilizado no código foi:

```
python transforma.py -i baboon_monocromatica -o baboon_viz_2 -e 2.0 -m vizinho
```

Como é possível ver no comando, o valor utilizado para a escala foi de 2.0. Isso significa que as dimensões da imagem serão dobradas. A imagem *baboon_monocromatica.png* original possui dimensões 512x512, então, nossa imagem resultado esperada deve ter 1024x1024.

A imagem resultado *baboon_viz_2.png* (Figura 15) e a comprovação de suas novas dimensões podem ser vistas a seguir.

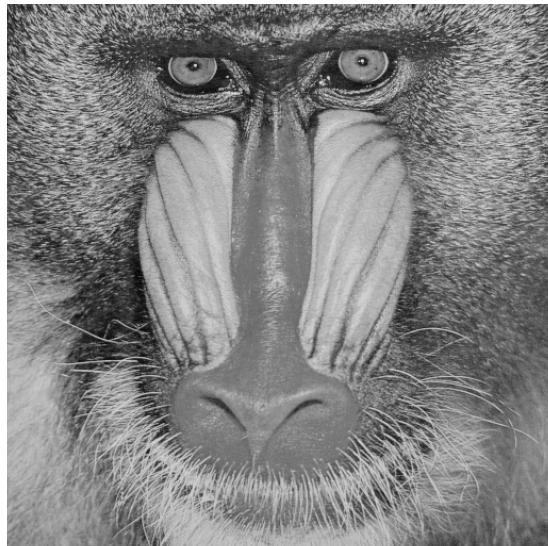


Imagen	
Dimensões	1024 x 1024
Largura	1024 pixels
Altura	1024 pixels
Intensidade de bits	8
Arquivo	
Nome	<i>baboon_viz_2.png</i>
Tipo de item	Arquivo PNG

Figura 16: dimensoes de *baboon_viz_2.png*

Figura 15: *baboon_viz_2.png*

Como é possível notar, a imagem resultante possui dimensões duas vezes maiores que a original.



Figura 17: zoom em baboon_monocromatica.png



Figura 18: zoom em baboon_viz_2.png

Ao comparar as imagens de forma mais cuidadosa, é possível notar pequenas diferenças na nitidez das duas imagens, onde a imagem original apresenta nitidez maior em alguns pontos, principalmente perto do olho. Importante também notar que, em questão de tamanho de pixels, a imagem com zoom em *baboon_viz_2.png* possui o dobro de altura e largura comparada ao zoom na imagem original.

3.2.2 Interpolação bilinear

```
def interpolacao_bilinear(img, x, y):
    x0, y0 = int(x), int(y)
    dx, dy = x - x0, y - y0

    val = (
        (1 - dx) * (1 - dy) * img[y0, x0] +
        dx * (1 - dy) * img[y0, x0 + 1] +
        (1 - dx) * dy * img[y0 + 1, x0] +
        dx * dy * img[y0 + 1, x0 + 1]
    )
    return int(val)
```

A interpolação bilinear realiza uma média ponderada dos quatro pixels vizinhos com base nas distâncias relativas dx e dy até a posição real (x, y) . Essa abordagem proporciona transições suaves entre os tons da imagem.

A fórmula exata utilizada é:

$$I(x, y) = (1-dx)(1-dy) \cdot I(x_0, y_0) + dx(1-dy) \cdot I(x_0+1, y_0) + (1-dx)dy \cdot I(x_0, y_0+1) + dxdy \cdot I(x_0+1, y_0+1)$$

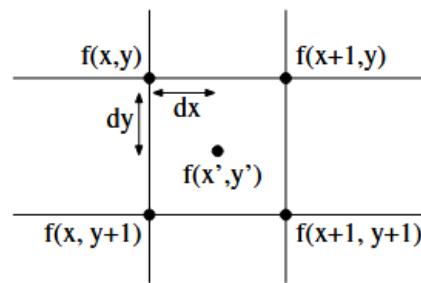


Figura 19: Exemplo de transformação utilizando interpolação bilinear

A mesma operação realizada no exemplo anterior será realizada aqui. Os resultados podem ser vistos a seguir:

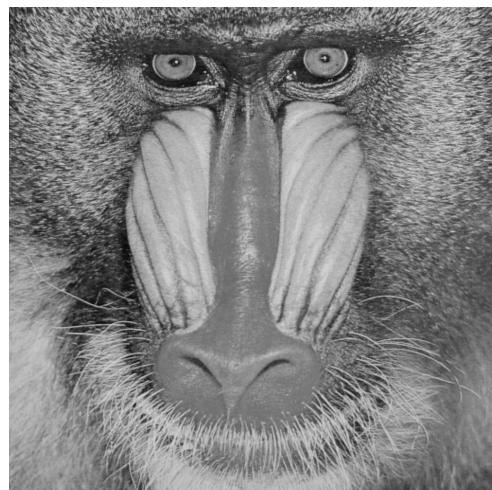


Figura 20: baboon_bil_2.png



Figura 21: zoom em baboon_monocromatica.png



Figura 22: zoom em baboon_bil_2.png

Esse método oferece um bom equilíbrio entre qualidade visual e custo computacional. Mas ainda assim é possível ver que, comparando com o método anterior, o teste não evidenciou uma diferença muito grande entre os resultados. Ao comparar com a imagem original, ainda é possível notar uma diferença, mas mais uma vez, isso se deve principalmente ao fato de existirem mais pixels para representar uma mesma área.

3.2.3 Interpolação bicúbica

```
def interpolacao_bicubica(img, x, y):
    x0, y0 = int(x), int(y)
    result = 0
    for m in range(-1, 3):
        for n in range(-1, 3):
            xm = np.clip(x0 + m, 0, w - 1)
            yn = np.clip(y0 + n, 0, h - 1)
            dx = x - (x0 + m)
            dy = y - (y0 + n)
            result += img[yn, xm] * R(dx) * R(dy)
    return int(np.clip(result, 0, 255))
```

Neste método, utilizamos uma janela 4×4 ao redor da coordenada real (x, y) , aplicando uma função B-spline cúbica para calcular os pesos de interpolação. A função $R(s)$ utilizada é definida como:

$$R(s) = \frac{1}{6} [P(s+2)^3 - 4P(s+1)^3 + 6P(s)^3 - 4P(s-1)^3] \quad \text{com } P(t) = \max(0, t)$$

Essa versão da função $R(s)$ proporciona suavidade contínua e deriva de uma base B-spline cúbica uniforme. O resultado é uma interpolação mais natural e suave, com excelente preservação de bordas curvas e gradientes suaves, especialmente útil em rotações e escalamentos não inteiros.

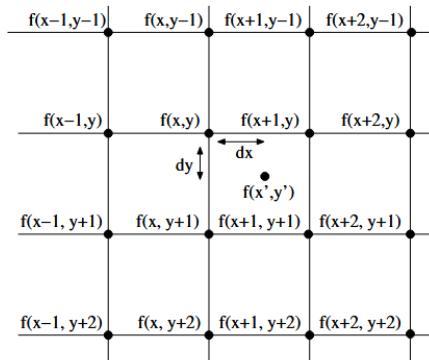


Figura 23: Exemplo de transformação utilizando interpolação bicúbica com B-spline

Visualmente, espera-se que essa interpolação minimize artefatos de aliasing e proporcione maior qualidade na reconstrução da imagem.

Os resultados podem ser vistos abaixo:

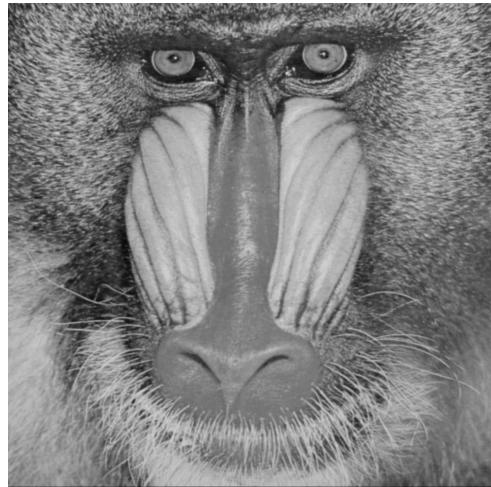


Figura 24: baboon_bic_2.png



Figura 25: zoom em baboon_monocromatica.png



Figura 26: zoom em baboon_bic_2.png

Como pode ser visto acima, o resultado da imagem ampliada pelo método da interpolação bicúbica gerou uma imagem um pouco mais "suave" que os resultados anteriores, suavização que provém da robustez do método e de sua capacidade de eliminar/atenuar artefatos e outros elementos indesejados na imagem. Algo digno de nota sobre o processo, mas que não convém análise profunda aqui, é o fato de a execução desse método demorar bem mais (em medições despretensiosas, cerca de 7 a 10 vezes mais) que o método anterior. A demora pode ser fruto da implementação do método, que utiliza o cálculo de R repetidamente, e, para imagens grandes (como a 512x512 original), muito mais operações são realizadas.

3.2.4 Interpolação de Lagrange

```
def interpolacao_lagrange(img, x, y):
```

```

x0, y0 = int(math.floor(x)), int(math.floor(y))
dx, dy = x - x0, y - y0
result = 0
for m in range(-1, 3):
    for n in range(-1, 3):
        xm = np.clip(x0 + m, 0, w - 1)
        yn = np.clip(y0 + n, 0, h - 1)
        result += img[yn, xm] * LagrangeAux(m, dx) * LagrangeAux(n, dy)
return int(np.clip(result, 0, 255))

```

A interpolação de Lagrange também utiliza uma vizinhança 4×4 em torno da posição real (x, y) , e se baseia em polinômios interpoladores de Lagrange de grau 3. Cada valor do pixel vizinho é ponderado por dois coeficientes independentes: um para a direção horizontal e outro para a vertical.

Espera-se que esse método produza reconstruções suaves e precisas, preservando melhor os contornos e transições contínuas, mas com maior custo computacional em relação aos demais métodos.

Os resultados podem ser vistos abaixo:

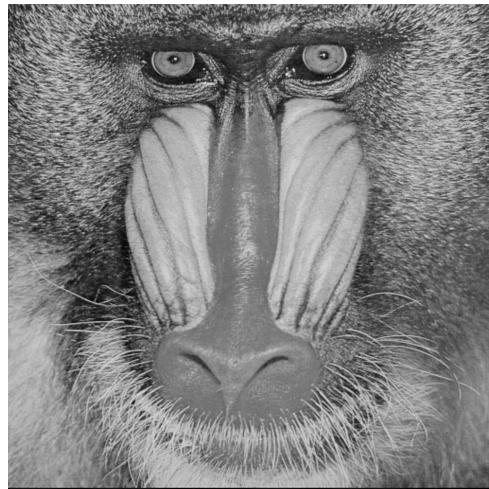


Figura 27: baboon_lag_2.png



Figura 28: zoom em baboon_monocromatica.png



Figura 29: zoom em baboon_lag_2.png

Como é possível ver acima, o resultado, assim como esperado, é uma imagem com reconstruções mais precisas, sem aquela suavização excessiva do método anterior, capturando as nuances da imagem original de maneira bem mais fidedigna. Porém, assim com o método anterior, o método de Lagrange apresentou um tempo de execução consideravelmente maior que os dois primeiros métodos.

É importante salientar que, embora todas as operações mostradas tenham sido realizadas com escala 2.0, escalas em $[0, 1]$ e maiores que 2.0 funcionam perfeitamente, onde numeros no intervalo $(0, 1)$, em vez de ampliar, reduzem a imagem. Também é importante ressaltar que, quanto maior o número escolhido, maior será o tempo de execução do código, naturalmente.

3.2.5 Rotação de imagens

A transformação de rotação foi implementada considerando o centro da imagem como ponto fixo. O ângulo de rotação θ (em graus) é convertido para radianos, e os coeficientes do cosseno e seno são computados:

```
ang_rad = math.radians(angulo)
cos_a, sin_a = math.cos(ang_rad), math.sin(ang_rad)
```

Para garantir que a imagem resultante acomode completamente a área rotacionada, calcula-se o novo tamanho da imagem com base na projeção dos quatro cantos originais:

```
corners = [(-cx, -cy), (w - cx, -cy), (-cx, h - cy), (w - cx, h - cy)]
x_coords = [cos_a * x - sin_a * y for x, y in corners]
y_coords = [sin_a * x + cos_a * y for x, y in corners]
```

Essas coordenadas são usadas para determinar a largura e altura da imagem de saída, garantindo que nenhum conteúdo rotacionado seja cortado.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x - c_x \\ y - c_y \end{bmatrix}$$

Para cada pixel na imagem de saída, aplica-se a transformação inversa para localizar a posição correspondente na imagem original. A interpolação é então aplicada nesta posição:

```
x_in = cos_a * x + sin_a * y + cx
y_in = -sin_a * x + cos_a * y + cy
```

Esse processo assegura que a imagem resultante seja visualmente consistente com a rotação desejada. As diferentes técnicas de interpolação controlam a suavidade e a fidelidade dos contornos e detalhes na imagem rotacionada.

A seguir, será utilizado como exemplo a imagem *house.png* (Figura 30) para demonstrar o funcionamento das rotações.

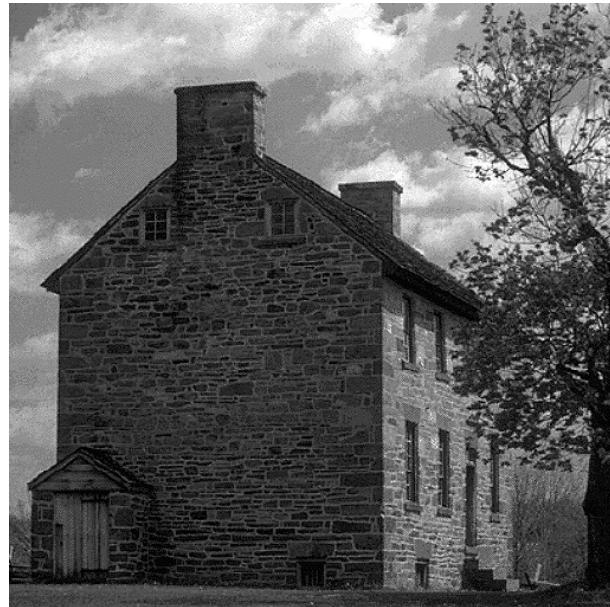


Figura 30: *house.png*

Primeiramente, uma rotação de 90 graus será realizada, utilizando a interpolação por vizinho mais próximo. O comando utilizado é:

```
python transforma.py -i house -o house_rot_viz -a 90 -m vizinho
```



Figura 31: house.png rotacionada em 90 graus pelo método dos vizinhos

Durante o processo de rotação, é comum que certas regiões da imagem de saída não sejam cobertas por nenhum pixel da imagem original, especialmente quando o ângulo não é múltiplo de 90° . No caso das interpolações por vizinhos e bilinear, essas lacunas surgem nos cantos da nova imagem e correspondem a áreas que não possuem mapeamento direto após a transformação inversa. Como a imagem de saída é inicializada com todos os pixels iguais a zero, essas regiões permanecem com valor 0, resultando visualmente em áreas pretas ao redor da imagem rotacionada. Esse comportamento é útil para destacar os limites da transformação, mas pode ser modificado caso se deseje outro preenchimento, como branco ou cinza.

Para mostrar esse comportamento, a mesma imagem será rotacionada em 45 graus.



Figura 32: house.png rotacionada em 45 graus pelo método dos vizinhos

Agora é possível notar nitidamente o comportamento mencionado, onde as regiões não cobertas por pixels da imagem original são preenchidas (implicitamente) com pixels pretos, devido à inicialização com `np.zeros()`.

Porém, as rotações utilizando as interpolações bicúbica e lagrange se comportam de forma diferente. A seguir, a mesma imagem rotacionada 45 graus, porém utilizando Lagrange como interpolador.



Figura 33: `house.png` rotacionada em 45 graus pelo método de Lagrange

Ao contrário das interpolações do tipo vizinho mais próximo e bilinear, os métodos bicúbico e de Lagrange não produzem bordas pretas nítidas quando a imagem é rotacionada. Isso ocorre devido à forma como esses interpoladores lidam com coordenadas próximas das bordas da imagem original. Ambos utilizam uma vizinhança 4×4 ao redor da coordenada real para realizar a interpolação. Quando essa vizinhança ultrapassa os limites da imagem, os índices acessados são ajustados utilizando a função `np.clip()`, que força os acessos a permanecerem dentro do intervalo válido. Como consequência, os pixels da borda da imagem original são reutilizados várias vezes nesses casos, resultando em valores interpolados que misturam a intensidade da borda com o valor nulo (zero) das regiões externas.

A seguir, por uma questão de tempo e o fato de que realizar uma análise exaustiva consumiria muito espaço, serão apresentados os resultados da imagem rotacionada 20 graus com cada um dos métodos de interpolação. E uma análise será feita sobre as maiores diferenças notadas.

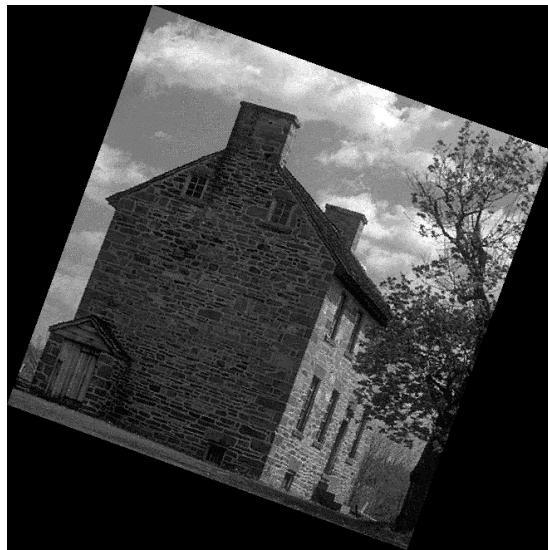


Figura 34: house.png rotacionada em 20 graus pelo método dos vizinhos

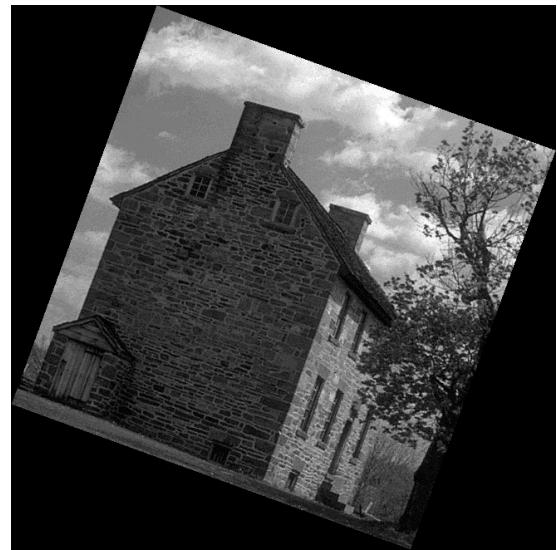


Figura 35: house.png rotacionada em 20 graus pelo método bilinear



Figura 36: house.png rotacionada em 20 graus pelo método bicubico



Figura 37: house.png rotacionada em 20 graus pelo método de Lagrange

Ao rotacionar a mesma imagem utilizando os quatro métodos de interpolação (vizinho mais próximo, bilinear, bicúbica e Lagrange), é possível observar diferenças sutis na qualidade e suavidade dos resultados. O método do vizinho mais proximo apresenta uma rotação rápida, mas com bordas visivelmente serrilhadas e pouco suavizadas, especialmente em ângulos não múltiplos de 90° , criando uma aparência pixelada. Já a interpolação bilinear suaviza essas transições, mas ainda mantém uma certa distorção nas bordas, embora em menor grau, e pode gerar uma leve suavização de objetos.

A interpolação bicúbica oferece uma transição mais suave e natural entre os pixels, com bordas bem definidas e menos "borradas". Esse método proporciona um resultado mais fiel à geometria da imagem original, mesmo em rotações pequenas. Por fim, o método de Lagrange também apresenta suavização das bordas, com resultados bastante similares

à bicúbica, mas com um custo computacional ligeiramente maior, o que torna a diferença visual mais difícil de perceber em rotações pequenas.