
**UNIVERSIDADE FEDERAL DE SERGIPE**
**DEPARTAMENTO DE COMPUTAÇÃO**

PROCESSAMENTO DE IMAGENS
Beatriz Trinchão Andrade

# PROJETO PRÉ OCR

LETÍCIA CENA DOS SANTOS
ANY CAROLINY SOUZA SILVA

**SÃO CRISTÓVÃO**
**2024**

**Sumário**

# 1. Introdução

Neste trabalho, desenvolvemos um sistema que aplica técnicas de operações morfológicas como erosão e dilatação além do filtro da mediana com o intuito de promover reconhecimento ótico de palavras, colunas, linhas (OCR) utilizando Python 3.6. Isto é feito após a remoção de ruídos na imagem fornecida de maneira a aprimorar a precisão e a legibilidade do processo de reconhecimento.

# 2. Técnicas Aplicadas

## 2.1. Filtro da Mediana:

A técnica de filtro de mediana foi aplicada para remover ruídos permitindo uma melhor visualização das letras e obter homogeneidade do fundo branco. Este filtro é particularmente eficaz na remoção de ruídos impulsivos, preservando os contornos das letras e minimizando a perda de detalhes importantes. O filtro da mediana é calculado a partir da aplicação de uma máscara (também conhecida como kernel), de dimensão m e n (largura e altura) em formato de matriz de forma que o resultado do somatório do produto da vizinhança junto ao elemento central da matriz dispostos em cálculo de mediana matemática obtenha-se a predominância dos itens que mais aparecem, no caso das imagens aqui estudadas, o fundo branco.

## 2.2. Operações Morfológicas (Dilatação e Erosão):

As operações morfológicas desempenharam um papel essencial no processamento das imagens binárias. A dilatação foi empregada para expandir as regiões de interesse, expandindo pixels pretos próximos e preenchendo lacunas, enquanto a erosão foi utilizada para reduzir o tamanho dos objetos, eliminando pequenos ruídos e refinando as formas dos caracteres. Essas técnicas contribuíram para a melhoria da segmentação e da clareza dos elementos textuais na imagem.

## 3. Código desenvolvido

No código criado algumas operações são realizadas conforme descrição abaixo

### 3.1. Leitura de Imagem em Formato PBM:

- Implementamos uma função leitor_PBM para ler arquivos de imagem no formato PBM (Portable Bitmap), que é uma representação de imagem binária.

```
recebe o caminho de um arquivo .pbm como entrada e lê todo o seu conteúdo,
retornando uma  string contendo as linhas do arquivo. """
def leitor_PBM(arquivo):
    # Abre o arquivo em modo de leitura
    with open(arquivo,'r') as arquivo:
        linhas = arquivo.read()
    # Retorna o conteúdo do arquivo em formato string
    return linhas
```

### 3.2. Análise e Conversão da Imagem:

- A função parse_string_array analisa a string do arquivo PBM, extrai largura, altura e os pixels da imagem, convertendo-os em um vetor bidimensional.

```
"" parse_string_vetor
recebe uma string contendo o conteúdo de um arquivo .pbm e o analisa para
extrair
largura, altura e os pixels da imagem """
def parse_string_vetor(pbm_string):
    # Divide a string em uma lista de linhas, separadas pelo caractere de quebra
de linha
    linhas = pbm_string.split('\n')

    # Remove linhas de comentário que começam com '#' da lista
    linhas = [linha for linha in linhas if not linha.startswith('#')]

    # Obtem a largura e altura da imagem a partir da segunda linha e converte
para inteiros
    largura, altura = map(int, linhas[1].split())
    # Obtem os pixels da imagem a partir da terceira linha
    pixels = linhas[2:]

    vetor_inteiros = []
    for linha in pixels:
        # Converte cada caractere da linha para inteiro e adiciona a lista de
pixels
        vetor_inteiros.extend([int(char) for char in linha])

    # Converte a lista de pixels em uma matriz bidimensional
```

```
    vetor_convertido = [vetor_inteiros[i:i+largura] for i in range(0,
len(vetor_inteiros), largura)]

    return largura, altura, vetor_convertido
```

### 3.3. Filtro de Mediana para Remoção de Ruído:

- Utilizamos o filtro de mediana para remover ruídos da imagem. A função filtro_mediana percorre a imagem aplicando a mediana na vizinhança de pixels, substituindo o pixel central pelo valor mediano.

```
""" mediana
    calcula a mediana de uma lista de inteiros """
def mediana(lista):
    return int(statistics.median(lista))

""" filtro_mediana
aplica o filtro da mediana a matriz de pixels """
def filtro_mediana(largura, altura, matriz):
    matriz_vizinhos = matriz_auxiliar(matriz)

    # Percorre as linhas da matriz (exceto as bordas)
    for i in range(1,altura-1):
        # Percorre as colunas da matriz (exceto as bordas)
        for j in range(1,largura-1):
            # Obtem os valores dos vizinhos do pixel atual
            vizinhanca = obter_vizinhanca(i,j, matriz_vizinhos)
            novo_pixel = mediana(vizinhanca)
             # Substitui o valor do pixel atual pelo valor da mediana
            matriz[i][j] = novo_pixel

    return matriz
```

### 3.4. Operações de Dilatação e Erosão:

- Implementamos funções para dilatação e erosão de imagens binárias, utilizando o ponto mínimo (erosão) e o ponto máximo (dilatação) da vizinhança para ser utilizado de forma estruturante na nova imagem a ser gerada. Erosão:

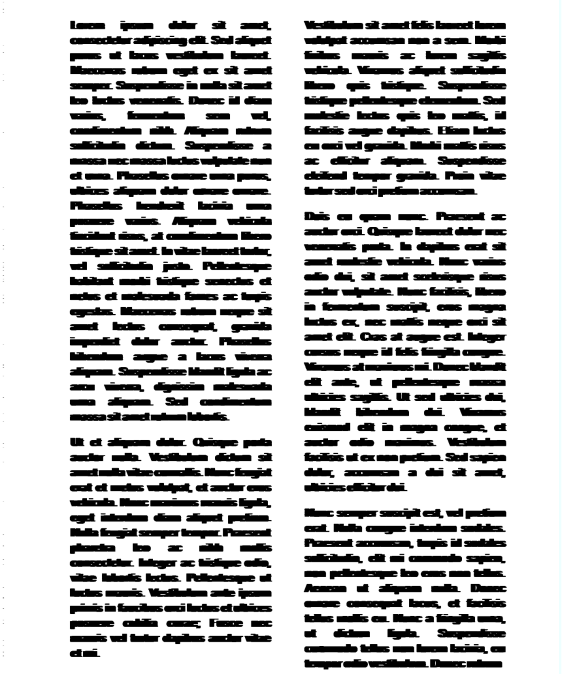
```
def erodir_imagem(imagem, elemento):
    # Obtendo as dimensões da imagem e do elemento estruturante
    altura, largura = imagem.shape
    altura_elem, largura_elem = elemento.shape
```

```
    # Inicializando uma matriz para a imagem erodida com zeros
    imagem_erodida = np.zeros((altura, largura), dtype=np.uint8)

    # Calculando o tamanho do padding necessário ao redor da imagem
    pad_altura = altura_elem // 2
    pad_largura = largura_elem // 2

    # Iterando sobre cada pixel da imagem
    for i in range(pad_altura, altura - pad_altura):
        for j in range(pad_largura, largura - pad_largura):
            # Calculando o mínimo dos valores resultantes da
multiplicação entre a região da imagem e o elemento estruturante
            if np.min(imagem[i-pad_altura:i+pad_altura+1,
j-pad_largura:j+pad_largura+1] * elemento) == 1:
                # Se o mínimo for 1, define o pixel na imagem erodida
como 1 (branco)
                imagem_erodida[i, j] = 1

    return imagem_erodida
```

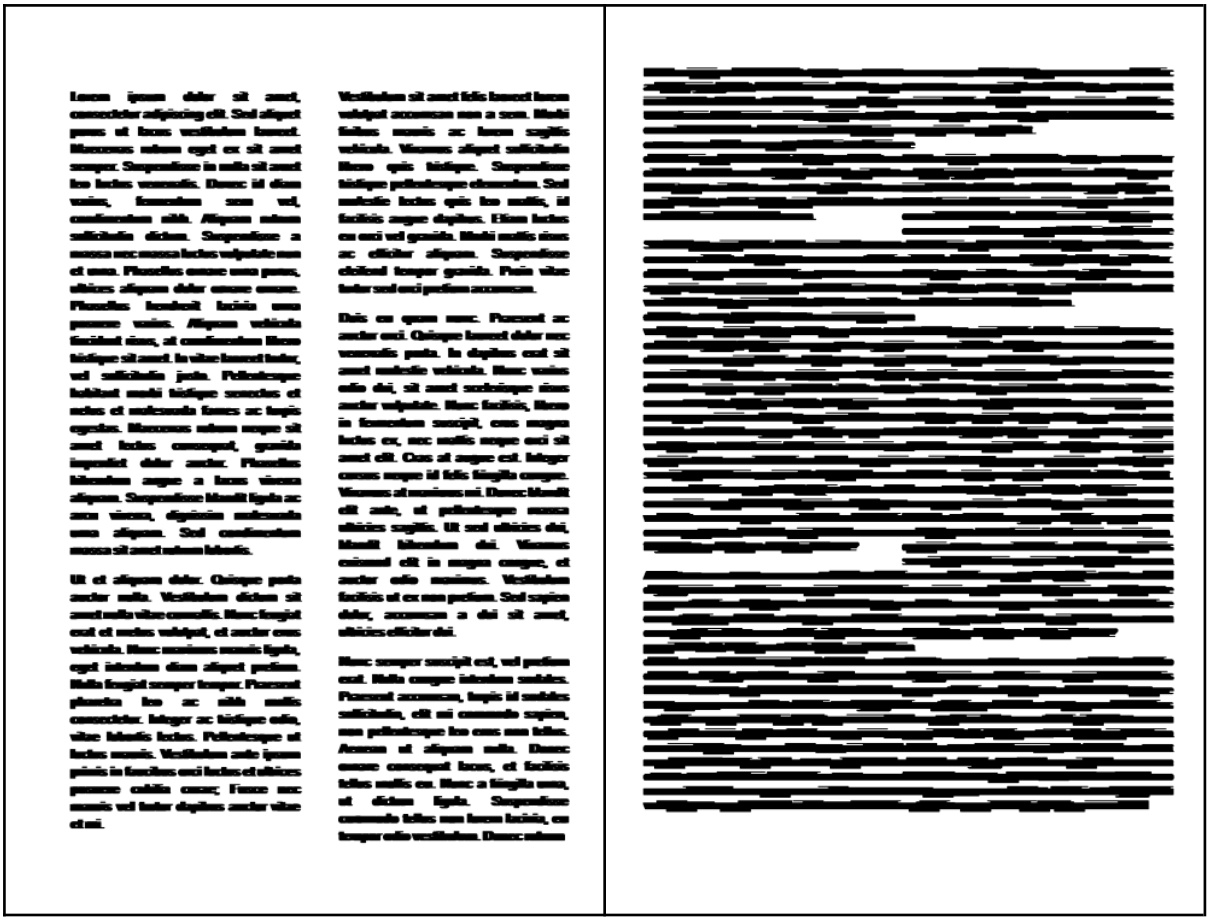
Dilatação:

```
def dilatar_imagem(image, elemento):
    # Obtem as dimensões da imagem
    altura, largura = len(image), len(image[0])
    # Obtem as dimensões do elemento estruturante
    elemento_altura, elemento_largura = len(elemento), len(elemento[0])
    # Cria uma matriz para a imagem dilatada
    imagem_dilatada = [[0 for _ in range(largura)] for _ in range(altura)]

    # Percorre cada pixel da imagem
    for y in range(altura):
        for x in range(largura):
            # Se o pixel atual for preto (1)
            if image[y][x] == 1:
                # Percorre o elemento estruturante
                for i in range(elemento_altura):
                    for j in range(elemento_largura):
                        # Verifica se a posição do elemento estruturante
está dentro dos limites da imagem
                        if 0 <= y + i < altura and 0 <= x + j < largura:
                            # Se tiver uma sobreposição entre o pixel
atual e o elemento estruturante
                            # define o pixel correspondente na imagem
dilatada como preto (1)
                            if elemento[i][j] == 1:
                                imagem_dilatada[y + i][x + j] = 1
    return imagem_dilatada
```

Com isto, ao aplicarmos erosão na imagem em questão temos um resultado de diminuição da intensidade de cores e no caso da dilatação temos um aumento de intensidade e área ocupada pelos pixels expandidos.

| Erosão | Dilatação |
| --- | --- |
|  |  |

Para aumentar a dilatação foi realizado um aumento de kernel, no âmbito da dimensão da máscara, em largura e altura. Com isto, conseguimos um resultado onde é possível visualizar as palavras delimitadas em formato retangular conforme imagem abaixo à esquerda. Na imagem a direita, com ampliando ainda mais a máscara aplicada foi possível deixar mais clara a relação de linhas sendo visualizadas.



### 3.5. Contador de linhas

A função abaixo, “contador_linhas” realiza o cálculo de linhas por colunas na imagem. Esta mesma função também atua no reconhecimento do número de colunas existentes na imagem.

```
def contador_linhas(vetor_imagem, qtd_colunas):
    # Verifica se o número de colunas é igual a 2
   if qtd_colunas == 2:
       # Cria uma lista para armazenar o número de linhas na primeira coluna
       array_2_col1 = []
       #  Percorre as colunas nos limites que definem a primeira coluna
       for c in range(400, 1100):
           qtd_linhas = 0
           # Percorre as linhas da primeira coluna
           # Desconsidera as 300 primeiras linhas
           # oois são espaços em branco
           for i in range(300, altura-1):
               if vetor_imagem[i][c] == 1 and vetor_imagem[i+1][c] == 0:
                   qtd_linhas += 1

           array_2_col1.append(qtd_linhas)

       array_2_col2 = []
```

```
    #  Percorre as colunas nos limites que definem a segunda coluna
    for c in range(1500, 2000):
        qtd_linhas = 0
        # Percorre as linhas da segunda coluna
        # Desconsidera as 300 primeiras linhas
        # oois são espaços em branco
        for i in range(300, altura-1):
            if vetor_imagem[i][c] == 1 and vetor_imagem[i+1][c] == 0:
                qtd_linhas += 1

        array_2_col2.append(qtd_linhas)
    #  Retorna o número dentre o numero de linhas obtidos nas duas colunas
    return (max(array_2_col1),max(array_2_col2))

# O número de colunas nao sendo 2, realiza o mesmo processo para três colunas
elif qtd_colunas == 3:
    array_3_col1 = []
    for c in range(300, 800):
        qtd_linhas = 0
        for i in range(300, altura-1):
            if vetor_imagem[i][c] == 1 and vetor_imagem[i+1][c] == 0:
                qtd_linhas += 1

        array_3_col1.append(qtd_linhas)

    array_3_col2 = []
    for c in range(990, 1500):
        qtd_linhas = 0
        for i in range(300, altura-1):
            if vetor_imagem[i][c] == 1 and vetor_imagem[i+1][c] == 0:
                qtd_linhas += 1

        array_3_col2.append(qtd_linhas)

    array_3_col3 = []
    for c in range(1660, 2160):
        qtd_linhas = 0
        for i in range(300, altura-1):
            if vetor_imagem[i][c] == 1 and vetor_imagem[i+1][c] == 0:
                qtd_linhas += 1

        array_3_col3.append(qtd_linhas)
    return (max(array_3_col1),max(array_3_col2),max(array_3_col3))
```

## 4. Tratamento de erros

- Tratamento de Borda na Aplicação do Filtro de Mediana
    - Para evitar a leitura das bordas da imagem ao aplicar o filtro de mediana, ajustamos as iterações para que a vizinhança considerasse somente pixels válidos dentro da imagem. Dessa forma, há uma verificação se a linha está dentro dos limites da matriz

```
"" obter_vizinhanca
recebe uma posição (linha, coluna) na matriz m e retorna uma
lista contendo os valores dos pixels vizinhos, incluindo o próprio pixel. """
def obter_vizinhanca(linha, coluna, m):
   # Verifica se a linha está dentro dos limites da matriz
   if (linha+1 < len(m) and linha-1 >= 0):
       # Obtem a 8 vizinhança
       vizinhos = [ m[linha-1][coluna][0], m[linha+1][coluna][0],
               m[linha][coluna-1][0], m[linha][coluna+1][0],
               m[linha-1][coluna+1][0], m[linha+1][coluna+1][0],
               m[linha-1][coluna-1][0], m[linha+1][coluna-1][0]]
       # Adiciona o valor do próprio pixel a lista de vizinhos
       vizinhos.append(m[linha][coluna][0])
       # Ordena a lista de vizinhos
       vizinhos.sort()
       return vizinhos
   return None
```

- Eficiência Computacional
    - Implementamos algoritmos otimizados para garantir uma execução eficiente, minimizando o tempo de processamento.

## Referências Bibliográficas

Rafael C. Gonzalez, Richard C. Woods - Fundamentals of Computer Graphics (Editora A. K. Peters - 3a edição - 2009)

H. Pedrini, W.R. Schwartz. Análise de Imagens Digitais: Princípios, Algoritmos e Aplicações (Editora Thomson Learning - 2008)