Trabalho 5

16 de Janeiro de 2021

Nome: Christian Hideki Maekawa - **RA:** 231867

1 Introdução

Foi desenvolvido um programa para realizar as transformações geométricas de escala e rotação em uma imagem. O fator de escala e o valor do angulo de rotação devem permitir valores contínuos (ou seja, valores em ponto flutuante).

2 O Programa

O programa foi implementado usando ubuntu 18.04.5 LTS e python 3.6.9. As bibliotecas utilizadas para este trabalho foram wget 3.2, click 7.1.2, numpy 1.18.5, matplotlib 3.2.2, opencv python 4.1.2.30. e pathlib 1.0.1.

A seguir coloquei os comandos para mostrar as configuração do ambiente.

[ ]:
```
!lsb_release -d
```

```
Description:    Ubuntu 18.04.5 LTS
```

[ ]:
```
!python --version
```

```
Python 3.6.9
```

2.1 Como executar

O programa foi desenvolvido para executar por linha de comando. O programa python chama `app.py` e contém 3 tipos de comandos:

- `download` : Cria uma ambiente de exemplo e baixa imagens.
- `prog` : Comando para aplicar as transformações geométricas
- `crop`: Comando para exibir as 4 interpolações um imagem.

2.2 Entrada

Imagens baixadas utilizando o wget. As imagens foram baixados do link.

2.3 Saída

A saída do comando `download` são as imagens de exemplo do site e um arquivo tx. A saída do comando `prog` é um arquivo com sufixo _coded que tem uma mensagem dentro da image. A saída do `crop` são 4 imagens com interpolações diferentes.

3 Parâmetros Utilizados

[ ]:
```
# Exemplos de comandos
python app.py download <str/path> # sem parâmetro criar uma pasta result
python app.py prog -i <str/image> -a <str/angle> -e <int/scale> -d <tuple/
 ↪dimension> -m <str/method> -i <str/input> -o <str/output> # sem parâmetro usa␣
 ↪o baboon.png
python app.py crop -i <str/image> -d <tuple/dimention> -ds <float/
 ↪dimenstionscale> -e <float/scale> # sem parâmetro usa o baboon.png
```

4 Solução

4.1 Download das imagens

As entradas das soluções são baixadas utilizando o wget. São baixadas 4 imagens de exemplo, baboon, monalisa, peppers e watch. Conforme a função abaixo. Essa função é utilizada pelo comando `download`.

[ ]:
```
def setup(path):
    """
    Essa função é responsável por baixar as imagens de exemplo.
    """
    [i.unlink() for i in path.rglob("*.png")] # Procura por imagens png e deleta
    wget.download("https://www.ic.unicamp.br/~helio/imagens_png/baboon.
 ↪png",out=str(path))
    wget.download("https://www.ic.unicamp.br/~helio/imagens_png/butterfly.
 ↪png",out=str(path))
    wget.download("https://www.ic.unicamp.br/~helio/imagens_png/city.png",␣
 ↪out=str(path))
    wget.download("https://www.ic.unicamp.br/~helio/imagens_png/house.png",␣
 ↪out=str(path))
    wget.download("https://www.ic.unicamp.br/~helio/imagens_png/seagull.png",␣
 ↪out=str(path))
```

4.2 Bibliotecas utilizadas

Foi utilizado, click para fazer as configurações de comandos, wget para baixar as imagens, pathlib para fazer manipulação de arquivos e pastas, o matplotlib para fazer os gráficos, numpy para fazer as operações matemáticas de vetorização e opencv para carregar a imagem e salvar.

[ ]:
```
import click
import wget
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
import cv2
```

4.2.1 Carregar imagem

Para carregar a imagen no programa é utilizada a função `loadImg`.É utilizada um dict para guardar a label da imagem e o conteúdo dela. Essa estrutura é utilizada para salvar imagem com nome da imagem.

[ ]:
```
def loadImg(path):
    """
    Essa função é responsável por carregar uma imagem. Path do pathlib é␣
 ↪utilizada para pegar uma imagem
    com final png da variável path o conteúdo vetorizado é armazenado no␣
 ↪dicionario de img.
    E o nome dessa imagem fica salvo no dict como referencia na hora salvar.
    """
    img = {}
    img[Path(path).stem] = cv2.cvtColor(cv2.imread(f"{path}"), cv2.
 ↪COLOR_BGR2RGB) # Carrega imagens BGR para RGB
    return img
```

4.2.2 Exibe recortes da imagem com todas as interpolação

Exibi 4 imagens cada uma contendo uma interpolação da mesma região. Para utilizar a função é preciso escolher a dimensão onde quer escolher o recurte e a imagem que quer recortar. Tem outros parametros opcionais como escolher o tamanho da escala para fazer o recorte e o tamanho do zoom que deseja fazer. Existe um padrão pré definido para processar em um tempo menor.

[ ]:
```
def cropImg(img,h,w,p):
    """Funcao para fazer o recorte na imagem"""
    new_h, new_w = np.min([int(h*p),int(img.shape[0])]), np.
 ↪min([int(w*p),int(img.shape[1])])
    return img[w:new_h,h:new_w,:]


# Comando para exibir as imagens recortada
@env.command()
@click.option('input', '--input','-i',type=click.Path(),default=str(_output /␣
 ↪'baboon.png'))
@click.option('dimension','--dimension','-d', nargs=2, type=int)
@click.option('dimensionscale','--ds','-ds', type=float, default=1.1)
@click.option('scale','--scale','-e', type=float,default=2)
def crop(input,dimension,dimensionscale,scale):
```

```
    """
    Funcao para mostrar imagem com interpolacao
    """
    img = loadImg(input) # Carrega dict image
    label = list(img.keys())[0] # carrega a label
    img = img[label] # Carrega imagem
    cropped = cropImg(img,dimension[0],dimension[1],dimensionscale) # recorta
    methods = ["vizinho","bilinear","bicubic","lagrange"] # Define os metodos
    imgs2 = [] # Guarda as imagens pos processada
    angle = None # Define metodo vazio para a funcao processar usando escala
    dimension = () # Define metodo vazio para a funcao processar usando escala
    imgs2.append(cv2.cvtColor(vizinhoproximo(cropped,scale,angle,dimension).
↪astype(np.uint8),cv2.COLOR_RGB2BGR))
    imgs2.append(cv2.cvtColor(bilinear(cropped,scale,angle,dimension).astype(np.
↪uint8),cv2.COLOR_RGB2BGR))
    imgs2.append(cv2.cvtColor(bicubic(cropped,scale,angle,dimension).astype(np.
↪uint8),cv2.COLOR_RGB2BGR))
    imgs2.append(cv2.cvtColor(lagrange(cropped,scale,angle,dimension).astype(np.
↪uint8),cv2.COLOR_RGB2BGR))

    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2) # Inicializa um espaço␣
↪para 4 imagens
    fig.suptitle(f"Imagens")
    fig.set_size_inches(15,15)
    for ax, img,b in zip(fig.get_axes(),imgs2,methods): # exibe imagens
        ax.imshow(img)
        ax.label_outer()
        ax.set_xticks([]), ax.set_yticks([])
        ax.set_title(f"{label} method={b}")
    plt.show()
```

4.2.3 Função que faz chamadas das interpolações

Define qual método e parâmetro o usuário definiu.

[ ]:
```
# Funcao que faz chamada da interpolacao
@app.command()
@click.option('angle','--angle','-a', type=click.IntRange(-360, 360, clamp=True))
@click.option('scale','--scale','-e', type=float)
@click.option('dimension','--dimension','-d', nargs=2, type=int)
@click.option('method', '--method','-m',required=True)
@click.option('input', '--input','-i',type=click.Path(),default=str(_output /␣
↪'baboon.png'))
@click.option('output', '--output','-o',type=click.Path(),default=str(_output /␣
↪'baboon_out.png'))
def prog(angle,scale,dimension,method,input,output):
```

```
   """Comando fazer interpolacao o metodo pode ser [vizinho, bilinear, bicubic,␣
↪lagrange]"""
   print(f"{angle},{scale},{dimension},{method},{input},{output}")
   if(((angle!=None) + (scale!=None) + (len(dimension)>0)) == 1):  # Verifica o␣
↪metodo escolhido
       img = loadImg(input) # Carrega imagem
       label = list(img.keys())[0] # Carrega nome da imagem
       img = img[label] # Carrega a imagem
       choose = None
       if method == "vizinho": # Escolhe o metodo do vizinho proximo
           custom_img = cv2.cvtColor(vizinhoproximo(img,scale,angle,dimension).
↪astype(np.uint8),cv2.COLOR_RGB2BGR)
       elif method == "bilinear": # Escolhe o metodo bilinear
           custom_img = cv2.cvtColor(bilinear(img,scale,angle,dimension).
↪astype(np.uint8),cv2.COLOR_RGB2BGR)
           choose = f"{method}_{}"
       elif method == "bicubic": # EScolher o bicubic
           custom_img = cv2.cvtColor(bicubic(img,scale,angle,dimension).
↪astype(np.uint8),cv2.COLOR_RGB2BGR)
       elif method == "lagrange": # Escolher lagrange
           custom_img = cv2.cvtColor(lagrange(img,scale,angle,dimension).
↪astype(np.uint8),cv2.COLOR_RGB2BGR)
       else: # Esse metodo nao existe
           raise Exception("Metodo nao disponivel")
       if scale != None: # Define o nome com o parametro
         choose = f"{method}_{scale}"
       elif angle != None:
         choose = f"{method}_{angle}"
       elif len(dimension) > 0:
         choose = f"{method}_{dimension}"
       filename = Path(input).parent / f"{Path(input).stem}_{choose}.png"
       print(filename)
       if not cv2.imwrite(str(filename),custom_img):
           raise Exception("Could not write image")
   elif (((angle!=None) + (scale!=None) + (len(dimension))) == 0):
       raise Exception("Faltando um argumento de transformacao")
   else:
       raise Exception("Esse algoritmo nao suporta mais que uma transformacao
```

4.3 Algoritmo

Essa parte do relatório estará focada em entender o código, mais a baixo estará a análise de cada método.

4.3.1 Funções auxiliares

[ ]:
```
def extractScale(img,new_height, new_width):
    """
    Essa função foi criado para calcular qual é o fator de escala do eixo x e y␣
 ↪ele pega o tamanho da dimensao
    e divide pela dimensao original
    """
    height, width, channel = img.shape
    return (new_height/height, new_width/width)

def extractDimension(img,scale):
    """
    Essa função criada para extrair a dimensão da nova imagem
    """
    height, width, channel = img.shape
    return (round(height*scale), round(width*scale),3)

def transformScale(p,s):
    """
    Faz produto e soma do ponto e do valor da escala
    """
    S = np.diag([s[0],s[1]])
    P = np.array(p).T
    return np.dot(P,S)[0:2]

def iTransformScale(p,s):
    """
    Transformacao inversa
    """
    S = np.diag([s[0],s[1],1])
    P = np.array([p[0],p[1],1]).T
    return np.linalg.solve(S, P)[0:2]

def transformGrade(p,g):
    """
    Faz produto e soma do ponto e do valor da rotacao
    """
    G =  np.array([np.cos(g), -np.sin(g),0 , np.sin(g), np.cos(g),0,0,0,1]).
 ↪reshape((3,3))
    P = np.array([p[0],p[1],1]).T
    return np.dot(G,P)[0:2]

def iTransformGrade(p,g):
    """
    Transformacao inversa da rotacao
    """
```

```
    G =  np.array([np.cos(g), -np.sin(g),0 , np.sin(g), np.cos(g),0,0,0,1]).
↪reshape((3,3))
    P = np.array([p[0],p[1],1]).T
    return np.linalg.solve(G, P)[0:2]
```

4.3.2 Vizinho mais próximo

Função criada para fazer a interpolação considerando os vizinhos mais próximos.O cria a dimensão da nova imagem e a partir dela calcula o ponto equivalente na imagem original utilizando criterio da vizinhanca mais próxima que equivale a usar o round do numpy.

[ ]:
```
def vizinhoproximo(img,scale,angle,dimension):
    """
    Função criada para fazer a interpolação considerando os vizinhos mais␣
↪próximos.
    O cria a dimensão da nova imagem e a partir dela calcula o ponto equivalente␣
↪na imagem original utilizando
    criterio da vizinhanca mais próxima que equivale a usar o round do numpy.
    """
    new_img = np.zeros(img.shape) # Define tamanho da imagem de saida
    ref_h, ref_w, ref_c = new_img.shape # Pega os tamanho
    if(scale!=None):
        ref_e = (scale, scale) # Define escala
        for i in range(ref_h): # percorre pelo novo formato
            for j in range(ref_w):
                copy_x, copy_y = tuple(iTransformScale((i,j), ref_e).round().
↪astype(int)) # Pega a coordenada a partir da transformacao inversa
                if copy_x < img.shape[0] and copy_y < img.shape[1]: # Para␣
↪considerar a tecnica do vizinho mais proximo e feito o arredondamento
                    new_img[i,j,0] = img[copy_x, copy_y,:] # Atribui o pixel da␣
↪transformada no pixel da nova imagem
    elif(len(dimension)>0):
        ref_e = extractScale(img,dimension[0],dimension[1]) # Extrai a escala␣
↪correspondente da dimensao
        for i in range(ref_h): # percorre pelo novo formato
            for j in range(ref_w):
                copy_x, copy_y = tuple(iTransformScale((i,j), ref_e).round().
↪astype(int)) # Para considerar a tecnica do vizinho mais proximo e feito o␣
↪arredondamento
                    new_img[i,j,0] = img[copy_x, copy_y,:]
    elif(angle!=None):
        ref_a = np.deg2rad(angle) # Converte para radiano
        mid_row = (ref_h+1)//2 # Pega o centro da imagem para girar
        mid_col = (ref_w+1)//2
        for i in range(ref_h):
            for j in range(ref_w):# Percorre a imagem
                xoff = i - mid_row # Aplica o offset para girar o local certo
```

```
                yoff = j - mid_col

                copy_x, copy_y = np.round(iTransformGrade((xoff,yoff),ref_a)).
↪astype(int) # Para considerar a tecnica do vizinho mais proximo e feito o␣
↪arredondamento
                copy_x += mid_row # Correcao do offset
                copy_y += mid_col # Correcao do offset
                if copy_x >= 0 and copy_y >= 0  and copy_x < img.shape[0] and␣
↪copy_y < img.shape[1]:
                    new_img[i,j,:] = img[copy_x,copy_y,:]
  return new_img
```

4.3.3 Bilinear

O método bilinear calcula a interpolacao criando uma nova imagem e achando seu valor equivalente similiar ao método do vizinho próximo, mas diferente do anterior ele utiliza um critério a mais para corrigir o valor do pixel utilizando média ponderada.

[ ]:
```
def bilinear(img,scale,angle,dimension):
  """
  O método bilinear calcula a interpolacao criando uma nova imagem e achando seu␣
↪valor equivalente
  similiar ao método do vizinho próximo, mas diferente do anterior ele utiliza␣
↪um critério a mais para corrigir
  o valor do pixel utilizando média ponderada.
  """
    if(scale!=None):
        new_img = np.zeros(extractDimension(img,scale)) # Extrai nova dimensao␣
↪da imagem
        ref_h, ref_w, ref_c = new_img.shape # Pega as dimensoes
        ref_e = (scale, scale) # Pega a escala
        for i in range(ref_h): # Percorre pela imagem
            for j in range(ref_w):
                copy_x, copy_y = tuple(iTransformScale((i,j), ref_e)) # Pega a␣
↪inversa
                copy_x_prev = copy_x.astype(int) # Pega o ponto x mais proximo
                copy_y_prev = copy_y.astype(int) # Pega o ponto y mais proximo
                copy_x_next = copy_x_prev + 1 # Pega o ponto x + 1 mais proximo
                copy_y_next = copy_y_prev + 1 # Pega o ponto x + 1 mais proximo
                dy_next = copy_y_next - copy_y # Acha diferencial da x + 1
                dx_next = copy_x_next - copy_x # acha diferencial da y + 1
                dy = 1 - dy_next # acha diferencial da prev
                dx = 1  - dx_next # acha diferencial da prev
                if copy_x_prev >= 0 and copy_y_prev >= 0 and copy_x_next < img.
↪shape[0] and copy_y_next < img.shape[1]:
                    for c in range(3): # Calcula a media ponderada
```

```
                        new_img[i][j][c] = dy *␣
↪(img[copy_x_prev][copy_y_next][c] * dx_next + img[copy_x_next][copy_y_next][c]␣
↪* dx) \
                        + dy_next * (img[copy_x_prev][copy_y_prev][c] * dx_next␣
↪+ img[copy_x_next][copy_y_prev][c] * dx)
    elif(len(dimension)>0):
        new_img = np.zeros((dimension[0], dimension[1], 3)) # Cria a nova imagem
        ref_h, ref_w, ref_c = new_img.shape # Pega as dimensoes
        ref_e = extractScale(img,dimension[0],dimension[1]) # Pega o valor da␣
↪escala
        for i in range(ref_h): # Percorre pela imagem
            for j in range(ref_w):
                copy_x, copy_y = tuple(iTransformScale((i,j), ref_e)) # Pega a␣
↪inversa
                copy_x_prev = copy_x.astype(int)
                copy_y_prev = copy_y.astype(int)
                copy_x_next = copy_x_prev + 1
                copy_y_next = copy_y_prev + 1
                dy_next = copy_y_next - copy_y
                dx_next = copy_x_next - copy_x
                dy = 1 - dy_next
                dx = 1  - dx_next
                if copy_x_prev >= 0 and copy_y_prev >= 0 and copy_x_next < img.
↪shape[0] and copy_y_next < img.shape[1]:
                    for c in range(3): # Calcula a media ponderada
                        new_img[i][j][c] = dy *␣
↪(img[copy_x_prev][copy_y_next][c] * dx_next + img[copy_x_next][copy_y_next][c]␣
↪* dx) \
                        + dy_next * (img[copy_x_prev][copy_y_prev][c] * dx_next␣
↪+ img[copy_x_next][copy_y_prev][c] * dx)
    elif(angle!=None):
        new_img = np.zeros(img.shape)
        ref_h, ref_w, ref_c = new_img.shape
        ref_a = np.deg2rad(angle)
        mid_row = (ref_h+1)//2
        mid_col = (ref_w+1)//2
        for i in range(ref_h):
            for j in range(ref_w):
                xoff = i - mid_row
                yoff = j - mid_col

                copy_x, copy_y = iTransformGrade((xoff,yoff),ref_a)
                copy_x += mid_row
                copy_y += mid_col
                copy_x_prev = copy_x.astype(int)
                copy_y_prev = copy_y.astype(int)
```

```
                copy_x_next = copy_x_prev + 1
                copy_y_next = copy_y_prev + 1
                dy_next = copy_y_next - copy_y
                dx_next = copy_x_next - copy_x
                dy = 1 - dy_next
                dx = 1  - dx_next
                if copy_x_prev >= 0 and copy_y_prev >= 0 and copy_x_next < img.
↪shape[0] and copy_y_next < img.shape[1]:
                    for c in range(3): # Calcula a media ponderada
                        new_img[i][j][c] = dy *␣
↪(img[copy_x_prev][copy_y_next][c] * dx_next + img[copy_x_next][copy_y_next][c]␣
↪* dx) \
                        + dy_next * (img[copy_x_prev][copy_y_prev][c] * dx_next␣
↪+ img[copy_x_next][copy_y_prev][c] * dx)
    return new_img
```

4.3.4 Bicubic

O método bicubic calcula a partir da nova imagem o pixel correspondente ao mesmo tempo é feito a correção do pixel agora considerando as 16 pontos dando um efeito de embaçamento muitas vezes não desejado.

[ ]:
```
def P(t):
  """
  Auxilia na equação R
  """
    return t if t > 0.0  else 0

def R(s):
  """
  Auxilia na equação bicubic
  """
    s = float(s)
    return (1/6) * ( (P(s+2)**3) -4*(P(s+1)**3) +6* (P(s)**3) -4*(P(s-1)**3) )

def bicubic(img,scale,angle,dimension):
          """
    O método bicubic calcula a partir da nova imagem o pixel correspondente ao␣
↪mesmo tempo
    é feito a correção do pixel agora considerando as 16 pontos dando um efeito␣
↪de embaçamento.
    """
    if(scale!=None):
        new_img = np.zeros(extractDimension(img,scale))
        ref_h, ref_w, ref_c = new_img.shape
        for i in range(ref_h):
            for j in range(ref_w):
```

```
                copy_x = i * (img.shape[0]/new_img.shape[0])
                copy_y = j * (img.shape[1]/new_img.shape[1])

                copy_x_prev = int(np.floor(copy_x))
                copy_y_prev = int(np.floor(copy_y))
                dx = copy_x - copy_x_prev
                dy = copy_y - copy_y_prev
                for n in range(-1,3):
                    for m in range(-1,3):
                        copy_x_next = copy_x_prev + m
                        copy_y_next = copy_y_prev + n
                        if copy_x_next >= 0 and copy_x_next >= 0 and copy_x_next␣
↪< img.shape[0] and copy_y_next < img.shape[1]:
                            new_img[i,j,:] += img[copy_x_next,copy_y_next,:] *␣
↪R(m-dx) * R(dy - n)
    elif(len(dimension)>0):
        new_img = np.zeros((dimension[0], dimension[1], 3))
        ref_h, ref_w, ref_c = new_img.shape
        for i in range(ref_h):
            for j in range(ref_w):
                copy_x = i * (img.shape[0]/new_img.shape[0])
                copy_y = j * (img.shape[1]/new_img.shape[1])

                copy_x_prev = int(np.floor(copy_x))
                copy_y_prev = int(np.floor(copy_y))
                dx = copy_x - copy_x_prev
                dy = copy_y - copy_y_prev
                for n in range(-1,3):
                    for m in range(-1,3):
                        copy_x_next = copy_x_prev + m
                        copy_y_next = copy_y_prev + n
                        if copy_x_next >= 0 and copy_x_next >= 0 and copy_x_next␣
↪< img.shape[0] and copy_y_next < img.shape[1]:
                            new_img[i,j,:] += img[copy_x_next,copy_y_next,:] *␣
↪R(m-dx) * R(dy - n)
    elif(angle!=None):
        new_img = np.zeros(img.shape)
        ref_h, ref_w, ref_c = new_img.shape
        ref_a = np.deg2rad(angle)
        mid_row = (ref_h+1)//2
        mid_col = (ref_w+1)//2
        for i in range(ref_h):
            for j in range(ref_w):
                xoff = i  - mid_row
                yoff = j  -  mid_col

                copy_x, copy_y = iTransformGrade((xoff,yoff),ref_a)
```

```
                copy_x += mid_row
                copy_y += mid_col
                copy_x_prev = int(np.floor(copy_x))
                copy_y_prev = int(np.floor(copy_y))
                dx = copy_x - copy_x_prev
                dy = copy_y - copy_y_prev
                for n in range(-1,3):
                    for m in range(-1,3):
                        copy_x_next = copy_x_prev + m
                        copy_y_next = copy_y_prev + n
                        if copy_x_next >= 0 and copy_y_next >= 0 and copy_x_next␣
↪< img.shape[0] and copy_y_next < img.shape[1]:
                            new_img[i,j,:] += img[copy_x_next,copy_y_next,:] *␣
↪R(m-dx) * R(dy - n)
    return new_img
```

4.3.5 Lagrange

O método de lagrange contorna o problema do método bicubic de deixar a imagem suavilizada. Os traços ficam mais intesos que o método anterior considerando 16 pontos em vez de 4.

[ ]:
```
def if_valid(x,y,x_max,y_max):
  """
  Auxilia na validação da dimensão na função L
  """
    return True if (x >= 0) & (y >= 0) & (x < x_max) & (y < y_max) else False

def L(img,n,x,y,dx):
  """
  Função auxiliar para f_
  """
    a1 = 0
    a2 = 0
    a3 = 0
    a4 = 0
    if if_valid(x-1,y+n-2,img.shape[0],img.shape[1]):
        a1 = (-dx*(dx-1)*(dx-2)*img[x-1,y+n-2,:])/6
    if if_valid(x,y+n-2,img.shape[0],img.shape[1]):
        a2 = ((dx+1)*(dx-1)*(dx-2)*img[x,y+n-2,:])/2
    if if_valid(x+1,y+n-2,img.shape[0],img.shape[1]):
        a3 = ((-dx)*(dx+1)*(dx-2)*img[x+1,y+n-2,:])/2
    if if_valid(x+2,y+n-2,img.shape[0],img.shape[1]):
        a4 = ((dx)*(dx+1)*(dx-1)*img[x+2,y+n-2,:])/6
    return a1+a2+a3+a4

def f_(img,x,y,dx,dy):
  """
```

```
  Função auxiliar para calcular lagrange
  """
    a1 = (-dy * (dy-1) * (dy-2) * L(img,1,x,y,dx))/6
    a2 = ((dy+1)*(dy-1)*(dy-2)*L(img,2,x,y,dx))/2
    a3 = (-dy*(dy+1)*(dy-2)*L(img,3,x,y,dx))/2
    a4 = (dy*(dy+1)*(dy-1)*L(img,4,x,y,dx))/6
    return a1+a2+a3+a4
def lagrange(img,scale,angle,dimension):
  """
  O método de lagrange contorna o problema do método bicubic de deixar a imagem
  suavilizada. Os traços ficam mais intesos que o método anterior considerando␣
 ↪16 pontos
  em vez de 4.
  """
    if(scale!=None):
        new_img = np.zeros(extractDimension(img,scale))
        ref_h, ref_w, ref_c = new_img.shape
        for i in range(ref_h):
            for j in range(ref_w):
                copy_x = i * (img.shape[0]/new_img.shape[0])
                copy_y = j * (img.shape[1]/new_img.shape[1])

                copy_x_prev = int(np.floor(copy_x))
                copy_y_prev = int(np.floor(copy_y))
                dx = copy_x - copy_x_prev
                dy = copy_y - copy_y_prev
                new_img[i,j,:] = f_ (img,copy_x_prev,copy_y_prev,dx,dy)
    elif(len(dimension)>0):
        new_img = np.zeros((dimension[0], dimension[1], 3))
        ref_h, ref_w, ref_c = new_img.shape
        for i in range(ref_h):
            for j in range(ref_w):
                copy_x = i * (img.shape[0]/new_img.shape[0])
                copy_y = j * (img.shape[1]/new_img.shape[1])

                copy_x_prev = int(np.floor(copy_x))
                copy_y_prev = int(np.floor(copy_y))
                dx = copy_x - copy_x_prev
                dy = copy_y - copy_y_prev
                new_img[i,j,:] = f_ (img,copy_x_prev,copy_y_prev,dx,dy)
    elif(angle!=None):
        new_img = np.zeros(img.shape)
        ref_h, ref_w, ref_c = new_img.shape
        ref_a = np.deg2rad(angle)
        mid_row = (ref_h+1)//2
        mid_col = (ref_w+1)//2
        for i in range(ref_h):
```

```
            for j in range(ref_w):
                xoff = i  - mid_row
                yoff = j  -  mid_col

                copy_x, copy_y = iTransformGrade((xoff,yoff),ref_a)
                copy_x += mid_row
                copy_y += mid_col
                copy_x_prev = int(np.floor(copy_x))
                copy_y_prev = int(np.floor(copy_y))
                dx = copy_x - copy_x_prev
                dy = copy_y - copy_y_prev
                new_img[i,j,:] = f_ (img,copy_x_prev,copy_y_prev,dx,dy)
    return new_img
```

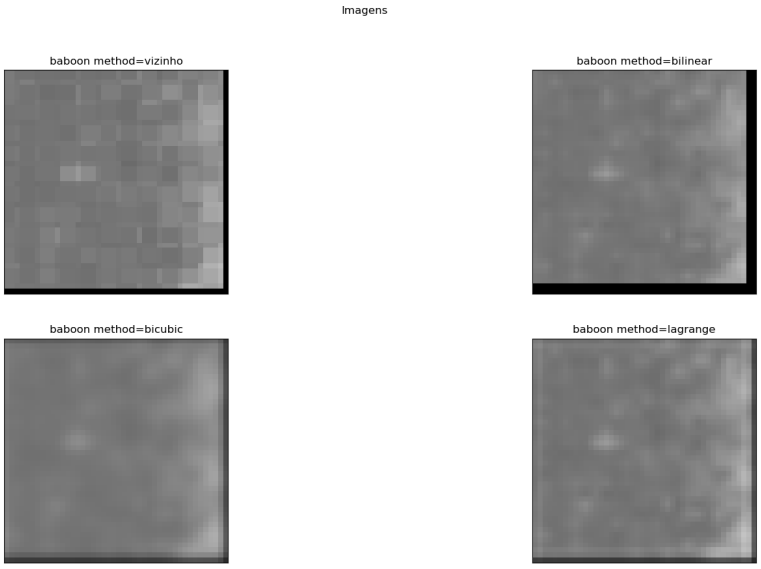
5 Análise de eficiência

Para esse experimento observou uma complexidade $N * M$ que depende principalmente do tamanho da imagem. No último caso que são considerados 16 pontos a mais a complexidade é multiplica por um fator de 4.

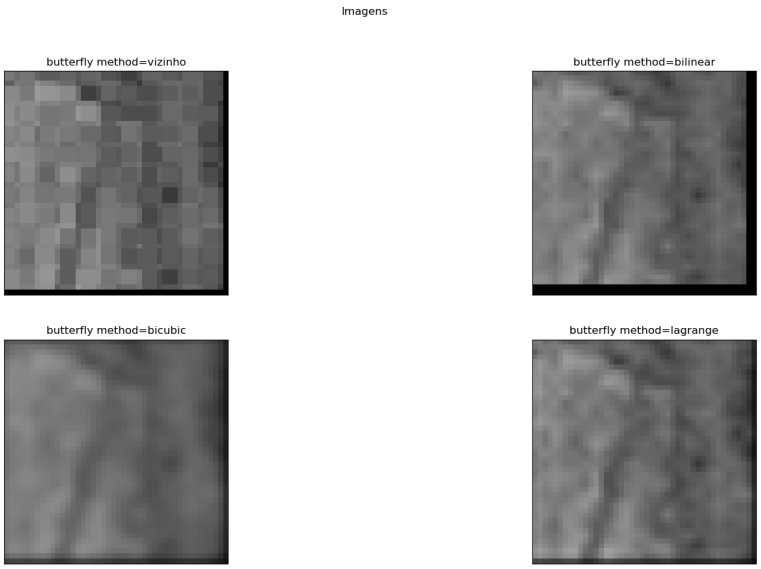
6 Resultados

A seguir, serão apresentados resultados obtidos com o programa de interpolação.

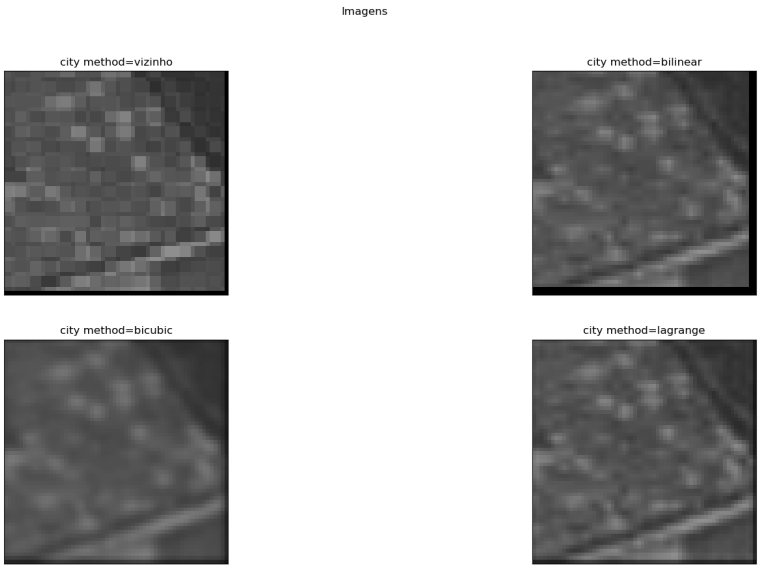
A primeira imagem é um recorte da face do babuíno. Onde percebeu a seguinte característica. A aproximação por vizinhos próximo deixou a imagem pixelada, o segundo método utilizando biliar teve um resultado bom perdendo um pouco dos detalhes. Já imagem bicubic deixou a imagem mais suave perdendo um pouco dos contornos e por fim a imagem Lagrange manteve os detalhes e também manteve os detalhes. Para uma imagem com mais qualidade optaria pela lagrange um rápida a bilinear e um método mais suave a bicubica.



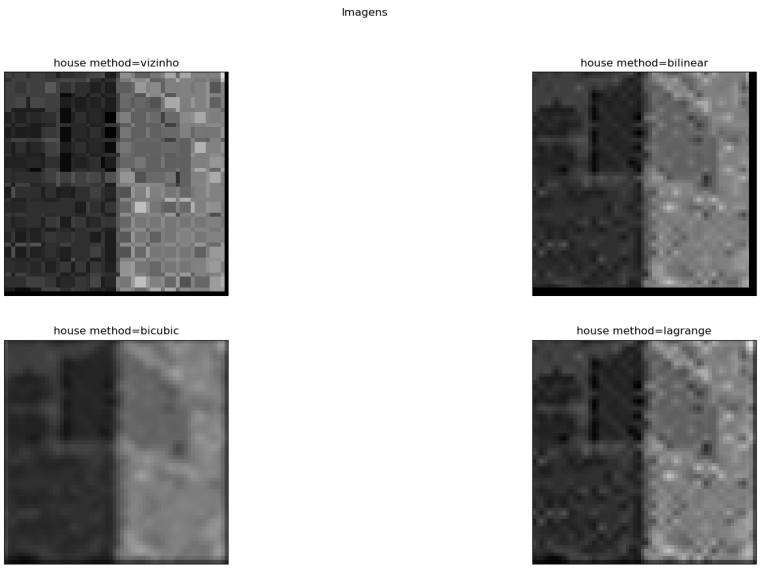
Essa segunda imagem é proveniente da imagem da borboleta como ela possui mais detalhes é possível observar que houve um pixa-lamento maior na primeira a segunda possui mais detalhes mas perdeu um pouco da borda terceiro perdeu os tons de cinzas mais escuros enquanto ultimo conseguiu preservar os detalhes. O melhor método seria o bilinear por te atingido resultado próximo do lagrange.



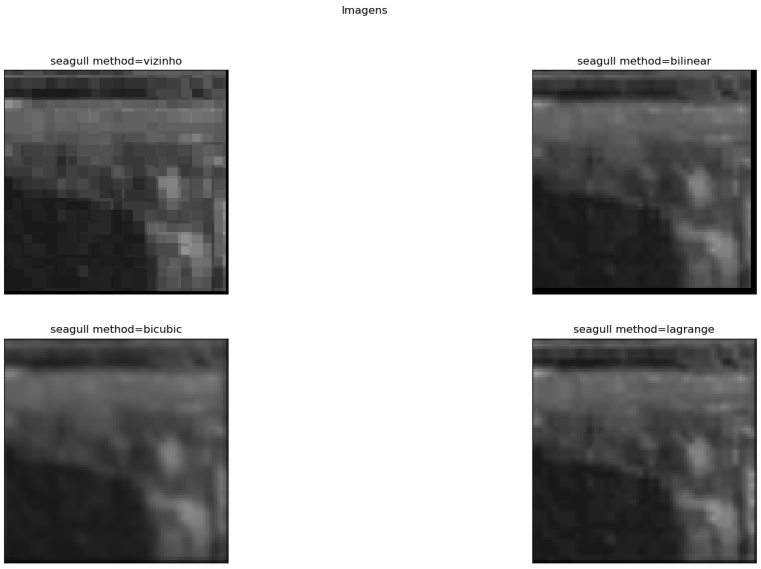
A terceira imagem contém um recorte do telhado. É possível perceber que nenhuma imagem perdeu a essência. O que percebeu maior mudança foi no detalhamento e na suavização. O melhor método seria vizinhos próximo já que não percebeu uma perda significativa.



A quarta imagem é um recorte na quina da casa, em todas as imagens foi possível identificar a quina. Percebeu que houve deterioração da imagem dos pixeis vizinhos não parece se corresponder muito bem o melhor método foi a bi cubica que não deixei os pixeis do tijolo sobre sair muito.

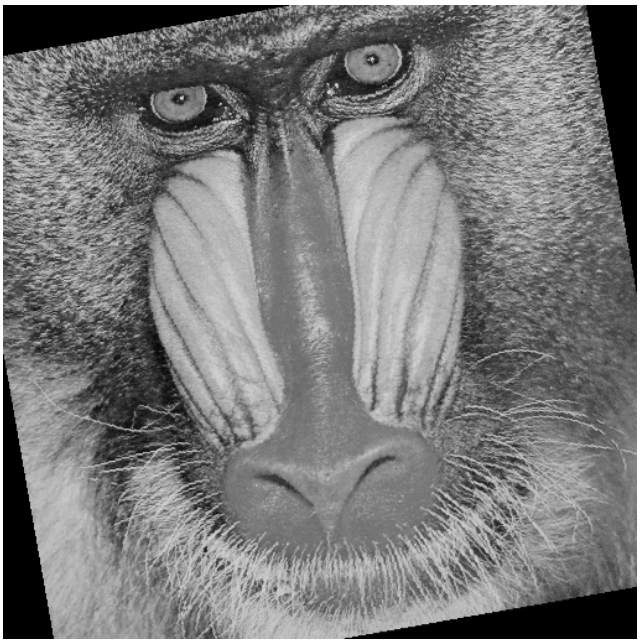


A quinta imagem é a imagem do seagull onde observeou que houve os mesmos resultados anteriores.

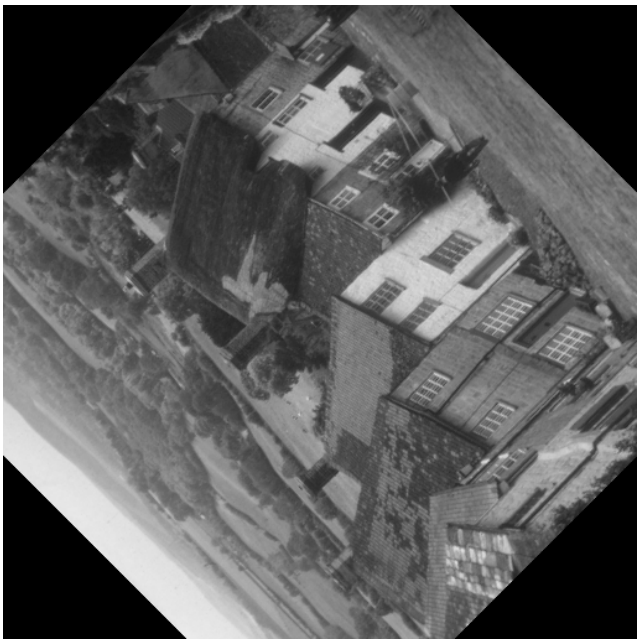


As próximas imagens apresentaram resultados de rotação onde é possível observar o efeito ta interpolação sobre uma imagem toda.

Utilizando vizinhos próximo observou a imagem sofre um pouco pixalamento.



Bilinear apresentou um resultado bastante satisfatório.



Bi cubica também apresentou bom resultado para a imagem da casa porque as outras técnicas re saltaram um pouco os tijolos da casa.

Lagrange também apresentou bom resultado. O único problema é sua complexidade um pouco maior.

7 Conclusão

As técnicas de interpolação no geral são boas e variam conforme a necessidade do projeto. A interpolação dos vizinhos próximos é uma técnica simples e rápida e exige pouco poder de processamento para um requisito que exige limite de hardware seria a melhor escolha. Já a segunda opção parece a melhor opção para obter um resultado que visualmente tem um resultado satisfatório para a visão humana.A terceira opção tem melhor resultado para imagens com grandes variação de cores e detalhes. A última opção, Lagrange, é o que apresentou maior nível de detalhamento e teve melhor resultado mas a percebeu que a medida que ia sendo desenvolvido os algoritmos mais complexo e lento os algoritmos iam se tornando. Também observou na rotação do lagrange para a imagem de seagull que houve uma acentuação de contorno na região dos olhos da imagem. Mas no geral todos os testes tiveram resultado bons.