数据可视化 HW4

姓名：刘佰川 学号：16307130214

一、问题：Python 编程实现在二维图片上画出等值线/等高线（不能调用某个算法库里面的等值线函数）；应用：输入原始图片（如 JPG、PNG 格式）和显示等值线数值或数量参数，输出带有等值线/等高线的图片。

算法思想：参照图片二值化算法，只需要灰度值为特定值的点描黑，即可以得到一条黑色的等值线。

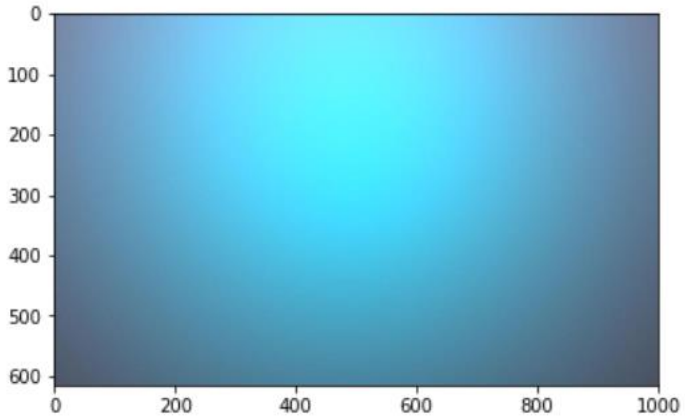
代码实现分为以下部分：

```
#导入需要的包
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import copy
```

```
#导入图片
img = Image.open("hw4_isocontour_c.jpg")
```

```
#显示原图
plt.imshow(img)
```

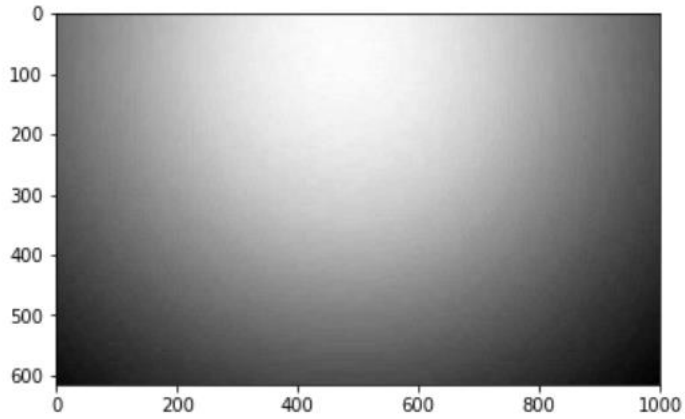
```
<matplotlib.image.AxesImage at 0x1ff8ab4ca20>
```




```
#转化成灰度格式的image
gray = np.array(img.convert('L'))
r,g,b = img.split()
```

```
#显示灰度值图片
plt.imshow(gray,cmap='gray')
```

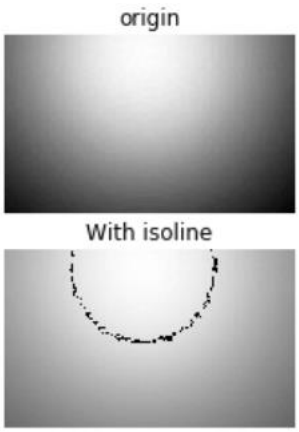
```
<matplotlib.image.AxesImage at 0x1ff8b0260b8>
```




```
xsize,ysize = gray.shape
```

```
#绘制等值线
threshold = 180
for x in range(xsize):
    for y in range(ysize):
        if gray[x,y] == threshold:
            change_gray[x,y] = 0
```

```
#显示图片
plt.figure('threshold using' + str(threshold))
plt.subplot(2,1,1)
plt.title("origin")
plt.imshow(gray,cmap='gray')
plt.axis('off')
plt.subplot(2,1,2)
plt.title("With isoline")
plt.imshow(change_gray,cmap='gray')
plt.axis('off')
plt.show()
```



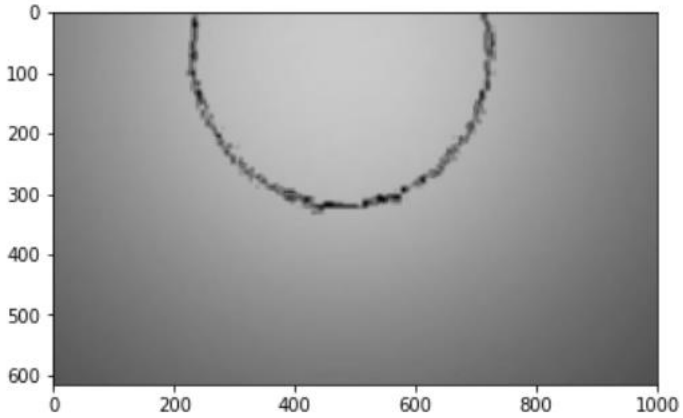
在做出此图后发现两个问题：第一是以灰度值 180 做出的等值线是一些离散的像素点，而非一条平滑的等值线，第二是将 C=180 的点全部的灰度变为 0 之后，图片上各个部分的灰度似乎都是变得更高了，所以整张图片变得更亮。对于第一个问题，我选择利用 cv2 包中的高斯滤波函数去平滑图片，但效果并不明显，对于第二个问题我还没完全明白，希望能够得到解惑。

利用高斯滤波平滑：

```
#保存新图片
new_gray = Image.fromarray(change_gray)
new_gray.save("new_gray.jpg")
```

```
import cv2
```

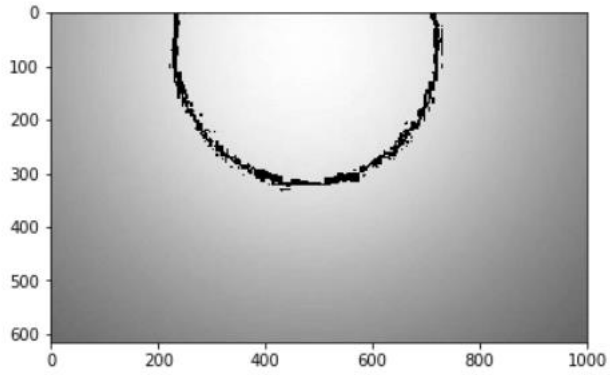
```
#高斯滤波 使离散的点能够变得稍微连续
img1 = cv2.imread('new_gray.jpg')
blured = cv2.blur(img1,(8,8))
plt.imshow(blured)
plt.show()
```



在采取老师所描述的方法，每四个点进行遍历，如果同时存在大于和小于阈值的点，将所有的点设置为 0 实现平滑。效果很好。

```
#导入图片
img = Image.open("hw4_isocontour_c.jpg")
#转化成灰度格式的image
gray = np.array(img.convert('L'))
xsize,ysize = gray.shape
#四个点进行遍历，若存在同时有比阈值大和小的值，将四个值都处理成0
change_gray = copy.copy(gray)
C = 180
#赋值
for x in range(xsize-1):
    for y in range(ysize-1):
        mark1 = 0 #大于
        mark2 = 0 #小于
        if gray[x,y] >= C or gray[x+1,y] >= C or gray[x,y+1] >= C or gray[x+1,y+1] >= C:
            mark1 = 1
        if gray[x,y] <= C or gray[x+1,y] <= C or gray[x,y+1] <= C or gray[x+1,y+1] <= C:
            mark2 = 1
        if mark1 == 1 and mark2 == 1:
            change_gray[x,y] = 0
            change_gray[x+1,y] = 0
            change_gray[x,y+1] = 0
            change_gray[x+1,y+1] = 0
plt.imshow(change_gray,cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x2096f5ea208>
```

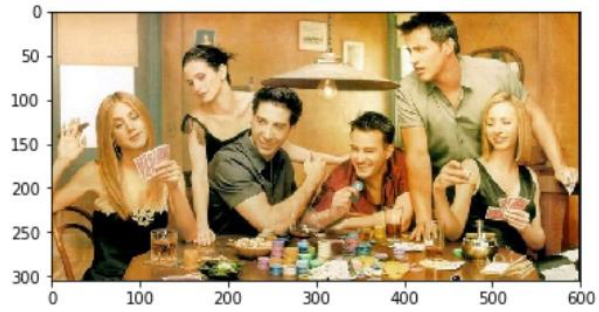


二、Python 编程实现灰度直方图均衡化算法（不能调用某个算法库里面的函数），并应用该算法在某个图片上，显示对比与原图的差别（提示：显示对比可以用两个图相减后的结果图，注意调节对比度范围，显示出差别）。

算法思想：利用离散形式的累积分布函数的值对归一化后的灰度值进行转化，因为采用累积分布函数的形式，在每一个灰度值上的概率密度函数都为 1。（在二值化图像的显示中，灰度值在[0,255]和[0,1]的形式都能正确显示）。

```
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x28a71f0a5c0>
```



```
#显示灰度图片
gray = np.array(img.convert('L'))
```

```
plt.imshow(gray,cmap='gray')
```

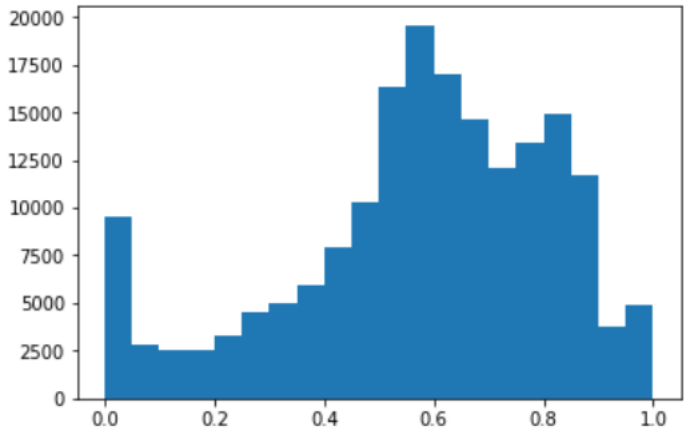
```
<matplotlib.image.AxesImage at 0x28a7066d198>
```


```
xsize,ysize = gray.shape
```

```
#记录灰度像素值
pixel = []
for x in range(xsize):
    for y in range(ysize):
        pixel.append(gray[x,y])
```

```
#归一化
maxiunm = max(pixel)
stand_pixel = [ x / float(maxiunm) for x in pixel]
```

```
#用直方图显示原图像的灰度分布
plt.hist(stand_pixel,bins = 20)
```



```
#产生新图像，准备处理
gray = gray / maxiunm
new_gray = copy.copy(gray)
```

```
#全局变量，方便函数处理
global stand_pixel
global new_pixel
new_pixel = []
```

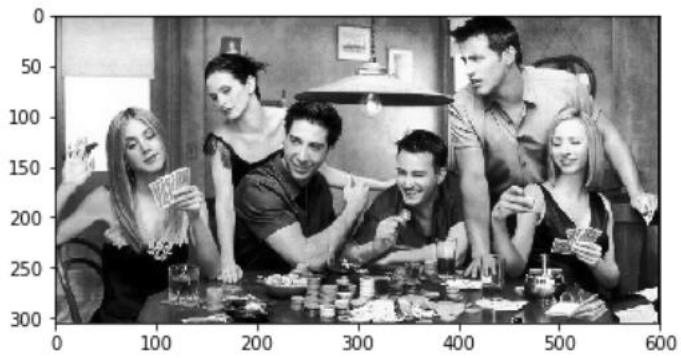
0.6235294117647059

```
#相当于实现累计函数的功能
def trans(x):
    n = len(stand_pixel)
    i = 0
    for ele in stand_pixel:
        if ele < x:
            i = i + 1
    new_pixel.append(float(i) / n)
    return  float(i) / n
```

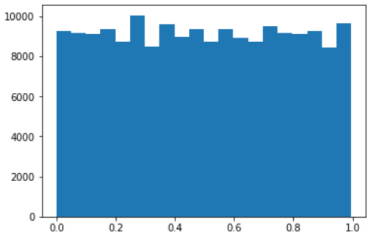
```
#赋值
for x in range(xsize):
    for y in range(ysize):
        new_gray[x,y] = trans(new_gray[x,y])
```

```
#显示均衡化的图片
plt.imshow(new_gray,cmap='gray')
```

<matplotlib.image.AxesImage at 0x28a730b9f98>



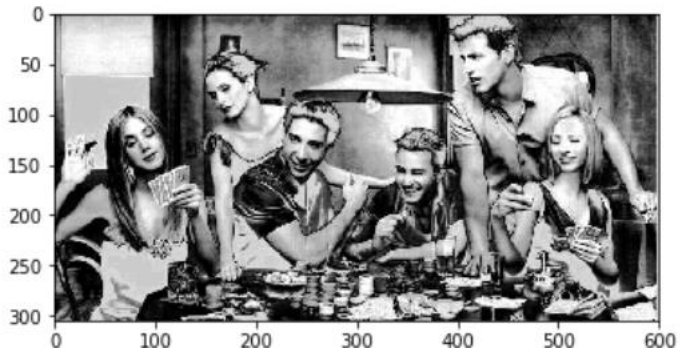

```
#显示均衡化图像的灰度直方图
plt.hist(new_pixel,bins = 20)
```



根据直方图，可以看到很好地实现了均衡化。

```
#显示图片之差
plt.imshow(-(gray - new_gray),cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x28a71e9ffd0>
```



显示图片之差进行对比（新图减原图），用头发部分进行解释，头发部分原本是灰度值较低的地方，均衡化的过程中提高了其的亮度，所以相减之后会比较亮。

但在实践过程中，明显发现算法的运行时间过长的问题，主要在于实现累积分布函数时没有优化，比如是对每个点处理，而非对可能的值先进行处理，再对应到点，导致循环的时间过长。偷懒自己没有进行优化，而是直接使用 numpy 中的函数实现累积分布函数，但此种方法并非对所有可能的值进行处理，所以采用了插值法处理。

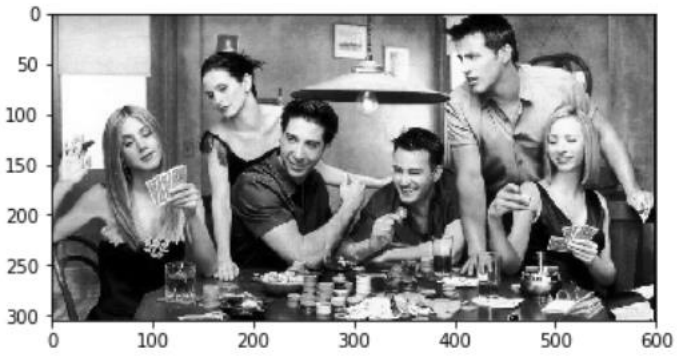
```
#对均衡化寻找更快的解决方案
def histeq(im,nbr_bins=256):
    """对一幅灰度图像进行直方图均衡化"""
    imhist,bins = np.histogram(im.flatten(),nbr_bins,normed=True)
    cdf = imhist.cumsum()#累积分布函数
    cdf = 255 * cdf / cdf[-1] #归一化
    #使用累积分布函数的线性插值
    im2 = np.interp(im.flatten(),bins[:-1],cdf)
    return im2.reshape(im.shape),cdf
```

```
new_gray1,cdf = histeq(gray)
```

```
C:\software\anoconda3\lib\site-packages\ipykernel_launcher.py:4: VisibleDepr
ecationWarning: Passing `normed=True` on non-uniform bins has always been br
oken, and computes neither the probability density function nor the probabil
ity mass function. The result is only correct if the bins are uniform, when
density=True will produce the same result anyway. The argument will be remov
ed in a future version of numpy.
  after removing the cwd from sys.path.
```

```
plt.imshow(new_gray1,cmap='gray')
```

此种方法均衡化的效果相同，但是运行时间大大减少。



三、Python 编程实现线性插值算法（不能调用某个算法库里面的插值函数）；读出一幅图像，利用线性插值把图片空间分辨率放大或缩小 N 倍，保存图片。

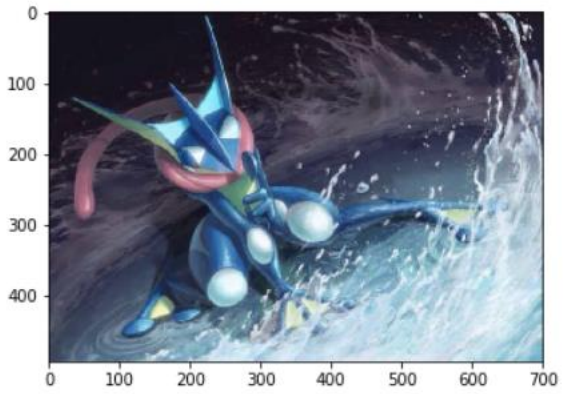
算法思想：直接是用二次插值，对于新图上的每一个点，先对应到原本图像的对应 4 个点上，用对应的点采取对应的公式进行插值。

```
import math
def bi_linear(src,target_size):
    img = Image.open(src)
    img = np.array(img)
    #原图像的尺寸
    #图片是size numpy是shape
    xsize,ysize,zsize = img.shape
    tar_x,tar_y,tar_z = target_size[0],target_size[1],target_size[2]
    #目标RGB图像
    target_pic = np.zeros((tar_x,tar_y,tar_z ),dtype = np.uint8)
    for k in range(tar_z):
        for i in range(tar_x):
            for j in range(tar_y):
                #在原图像找到对应坐标
                corr_x = (i+0.5)/tar_x*xsize-0.5
                corr_y = (j+0.5)/tar_y*ysize-0.5
                #找到对应的四个点 保证不超过边界
                p1 = (math.floor(corr_x),math.floor(corr_y))
                p2 = (min(p1[0],xsize-1),min(p1[1]+1,ysize-1))
                p3 = (min(p1[0]+1,xsize-1),min(p1[1],ysize-1))
                p4 = (min(p1[0]+1,xsize-1),min(p1[1]+1,ysize-1))
                #进行二分插值 先对y 再对x
                f1 = (p2[1]-corr_y)*img[p1[0],p1[1],k] + (corr_y-p1[1])*img[p2[0],p2[1],k
                f2 = (p2[1]-corr_y)*img[p3[0],p3[1],k] + (corr_y-p1[1])*img[p4[0],p4[1],k
                target_pic[i,j,k] = (p3[0]-corr_x)*f1 + (corr_x-p1[0])*f2
    plt.imshow(target_pic)
    #保存图片
    new = Image.fromarray(target_pic)
    new.save("new.jpg")
```

显示原图：

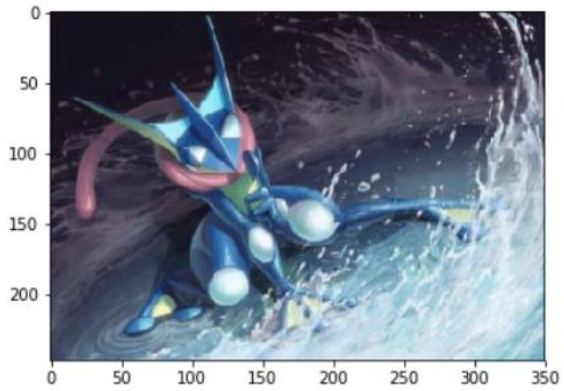
```
src = "q3.jpg"
img = Image.open(src)
img = np.array(img)
x,y,z = img.shape
print(img.shape)
target_size = [x,y,3]
bi_linear(src,target_size)
```

```
(494, 700, 3)
```



显示尺度缩小 2 倍的图像，可见图像的清晰度有降低。

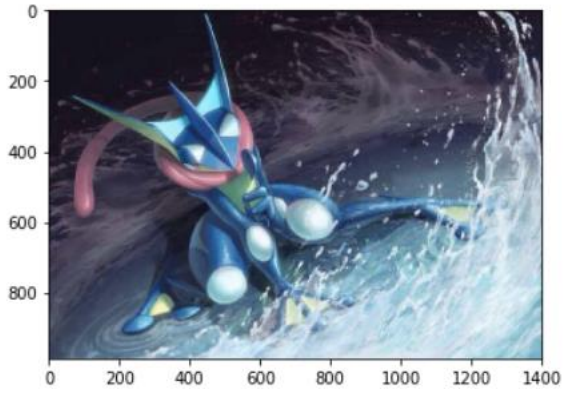
```
src = "q3.jpg"
img = Image.open(src)
img = np.array(img)
#高 宽 channels
x,y,z = img.shape
target_size = [x//2,y//2,3]
bi_linear(src,target_size)
```



显示尺度扩大 2 倍的图像，可见图像的清晰度有提升。

```
src = "q3.jpg"
img = Image.open(src)
img = np.array(img)
x,y,z = img.shape
print(img.shape)
target_size = [x*2,y*2,3]
bi_linear(src,target_size)
```

```
(494, 700, 3)
```



四、Python 编程实现强化边界滤波算法代码（不能调用某个算法库里面的函数）；并应用一张图片上，显示与原图的对比差别。

算法思想：利用二阶导数求出边界（拉普拉斯算子），再叠加到原图。此处应注意到拉普拉斯算子求出来的是边界图，一开始概念理解有误，认为其实处理之后的图片，走了很多弯路。此处算法采用了滤波的形式。

```
#利用拉普拉斯算子实现强化边界的滤波算法
laplacian = np.zeros([3,3],dtype=float)
#填充滤波
laplacian[0,1] = 1
laplacian[1,0] = 1
laplacian[1,1] = -4
laplacian [1,2] = 1
laplacian[2,1] = 1

img = Image.open("q4.jpg")
img = np.array(img.convert('L'))
height,weight = img.shape
new_img = copy.copy(img)
turn = 1
#进行多次滤波，使效果更明显
for k in range(turn):
    for x in range(weight - 1):
        for y in range(height - 1):
            pixel = 0
            for i in [-1,0,1]:
                for j in [-1,0,1]:
                    pixel = pixel + img[y+j,x+i]*laplacian[i+1,j+1]
            new_img[y,x] = pixel
```

```
plt.figure()
plt.subplot(2,1,1)
plt.title("origin")
plt.imshow(img,cmap='gray')
plt.axis('off')
plt.subplot(2,1,2)
plt.title("After filtering")
#plt.imshow(new_img,cmap='gray')
plt.imshow(new_img+img,cmap='gray')
plt.axis('off')
plt.show()
```

五、课后阅读自学内容

已完成