图像分割

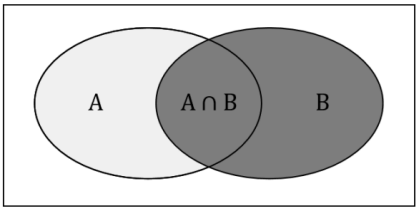
图像分割的方法综述

本题共提供4对图像（输入图像与理想分割结果）， 前3对图像为识别一个区域， 最后一对图像为识别2个物体区域。

尝试使用不少于三种方法来实现这四幅图的图像分割， 建议至少有一种方法为相对前沿的分割方法， 以提高作业的质量。分割的结果除视觉效果外， 可以尝试用 Jaccard 或 Dice 指数来评价分割效果的优劣。

- Jaccard 指数

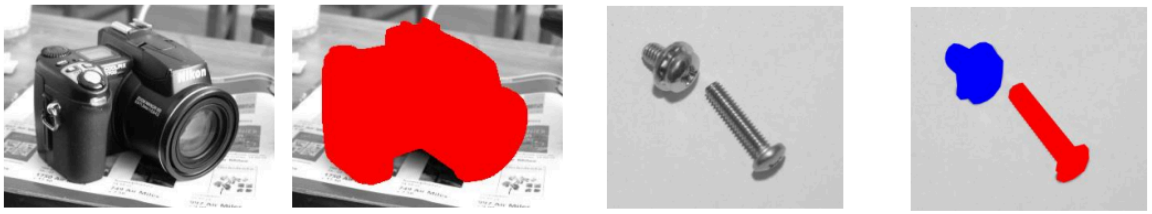
$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$



- Dice系数

$$D(A,B) = \frac{2|A \cap B|}{|A| + |B|}$$

指标的取值范围都是0到1，指标值越接近 1，说明分割结果与真实分割结果越相似。



0. 分割评价方式

以下图像分割均使用 Jaccard 或 Dice 指数来评价分割效果的优劣。

- Jaccard 指数

  Jaccard 指数是一种用于衡量两个集合相似度的指标，其原理是通过计算两个集合的交集与并集之间的比值来确定它们的相似程度。具体来说，Jaccard指数定义为两个集合交集大小除以它们的并集大小，即

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

该指数的取值范围在0到1之间，值越大表示两个集合越相似，值为0表示两个集合没有任何交集，值为1表示两个集合完全相同。在Python中可以通过自主编写函数来求 Jaccard 指数，该函数需要两个参数，为需要进行比较的两个二值图像，首先使用 np.logical_and() 函数计算出两个二值图像的交集，然后使用 np.logical_or() 函数计算出它们的并集。最后，将交集的像素数除以并集的像素数，得到 Jaccard 指数。

- Dice 指数

  Dice指数是另一种用于衡量两个集合相似度的指标，其原理是通过计算两个集合的交集的两倍与它们的元素个数之和之间的比值来确定它们的相似程度。具体来说，Dice指数写成表达式的形式为

$$D(A,B) = \frac{2|A \cap B|}{|A| + |B|}$$

  其中 A 和 B 是两个集合，|A|和|B|分别表示它们的元素个数。该指数的取值范围在0到1之间，值越大表示两个集合越相似，值为0表示两个集合没有任何交集，值为1表示两个集合完全相同。在Python中可以通过自主编写函数来求 Dice 指数，该函数需要两个参数，为需要进行比较的两个二值图像，首先将两个二值图像除以 255，将像素值归一化到0到1之间，然后使用 np.sum() 计算它们的交集。接着，将交集的像素数乘以2，除以两个二值图像的像素数之和（同样使用 np.sum() 计算），得到Dice指数。

Jaccard 指数和 Dice 指数的自编计算函数代码如下。

```
import cv2
import numpy as np

def jaccard_index(img1, img2):
    intersection = np.logical_and(img1, img2)
    union = np.logical_or(img1, img2)
    jaccard_index = np.sum(intersection) / np.sum(union)
    return jaccard_index

def dice_coefficient(img1, img2):
    intersection = np.sum(img1 / 255 * img2 / 255)
    dice_coefficient = (2. * intersection) / (np.sum(img1 / 255) + np.sum(img2 /
255))
    return dice_coefficient
```

1. 基于阈值的分割

针对题目的要求，首先尝试最简单的方法之一：基于阈值的分割。

阈值分割是一种简单的图像分割方法，它基于像素灰度值的阈值来将图像分成两个或多个部分。具体来说，该方法将图像中所有像素的灰度值与预先设定的阈值进行比较，将像素分为两个类别：高于阈值和低于阈值。这种方法适用于图像中具有明显灰度差异的区域，但对于灰度变化不明显的区域效果不佳。

实现步骤

基于阈值进行图像分割的具体步骤有以下几点：

- 灰度值统计：首先，需要对图像进行灰度值统计，通常是将图像中每个像素的灰度值根据其出现的频率绘制成灰度直方图。灰度直方图可以看作是灰度值分布的一种统计表示，能够帮助我们更好地了解图像的灰度特征和分布情况。

- 阈值选取：在得到灰度直方图后，需要根据图像的特点和需求选择合适的阈值。阈值通常是一个灰度值，用于将图像中灰度值高于或低于该值的像素分成不同的类别。阈值的选取通常是一个经验性的过程，可以通过试错和观察结果来确定。
- 分割处理：确定阈值后，可以将图像中的像素根据其灰度值与阈值之间的关系分为不同的类别。通常将灰度值高于阈值的像素标记为前景（或目标）像素，将灰度值低于阈值的像素标记为背景像素。分割处理可以通过二值化操作来实现，即将灰度值高于阈值的像素设为1，将灰度值低于阈值的像素设为0，从而得到一个二值化图像作为输出结果。
- 后处理：在得到分割结果后，通常需要进行后处理来进一步完善分割效果。常见的后处理方法包括去噪、边缘连接和形态学处理等，可以根据实际需求选择合适的后处理方法。

使用 Python 进行基于阈值的图像分割则可写成以下几个步骤：

- 读取图像：首先，从本地读取需要进行分割的图像和理想分割图像，并将得到的三通道数据将图像转换为灰度图像。
- 阈值选取：使用 cv2 库中的函数 cv2.THRESH_OTSU 针对图像使用Otsu自适应阈值算法自动选取阈值。
- 分割处理：确定阈值后，使用反二进制阈值分割方法。使用 cv2.THRESH_BINARY_INV 将灰度值高于阈值的像素设为黑色（0），低于阈值的像素设为白色（255）。
- 结果展示：将得到的分割保存到新的变量中，展示分割结果。计算相应的 Jaccard 指数和 Dice 指数用于评价分割效果。
- 去噪：尝试几种去噪方式，如形态学开运算方法、小波变换等。重新计算相应的 Jaccard 指数和 Dice 指数，和原始结果进行比较。

代码与结果

img1

```
img1 = cv2.imread('Image1.png')
seg1 = cv2.imread('Seg1.png')
gray_img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray_seg1 = cv2.cvtColor(seg1, cv2.COLOR_BGR2GRAY)
ret_img1, thresh_img1 = cv2.threshold(gray_img1, 0, 255,
cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
ret_seg1, thresh_seg1 = cv2.threshold(gray_seg1, 0, 255,
cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
cv2.imshow('image1', img1)
cv2.imshow('seg1', seg1)
cv2.imshow('thresh_img1', thresh_img1)
cv2.imshow('thresh_seg1', thresh_seg1)
cv2.waitKey(0)
cv2.destroyAllWindows()
print("Image1")
Jaccard_index_1 = jaccard_index(thresh_img1, thresh_seg1)
print("Jaccard指数为：", Jaccard_index_1)
Dice_index_1 = dice_coefficient(thresh_img1, thresh_seg1)
print("Dice指数为：", Dice_index_1)
```

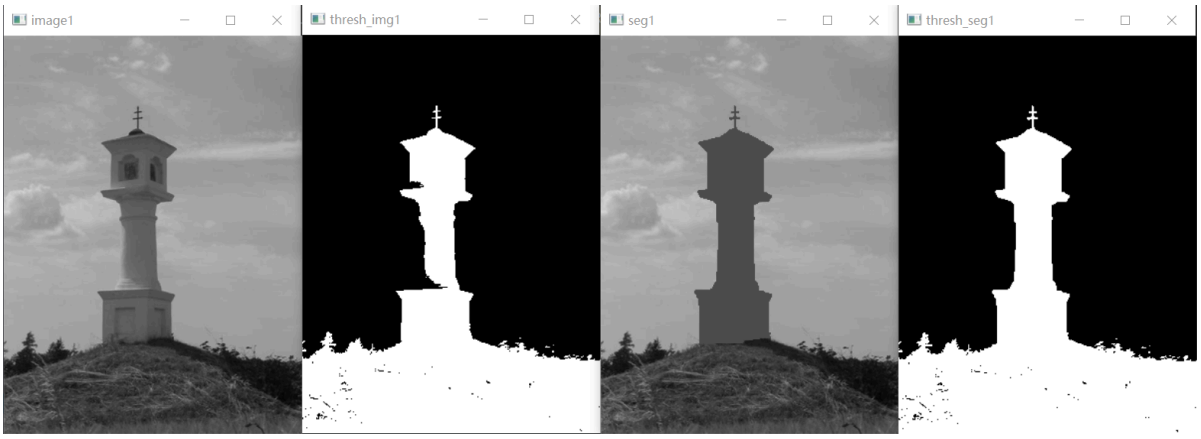


image1
thresh_img1
seg1
thresh_seg1

img2

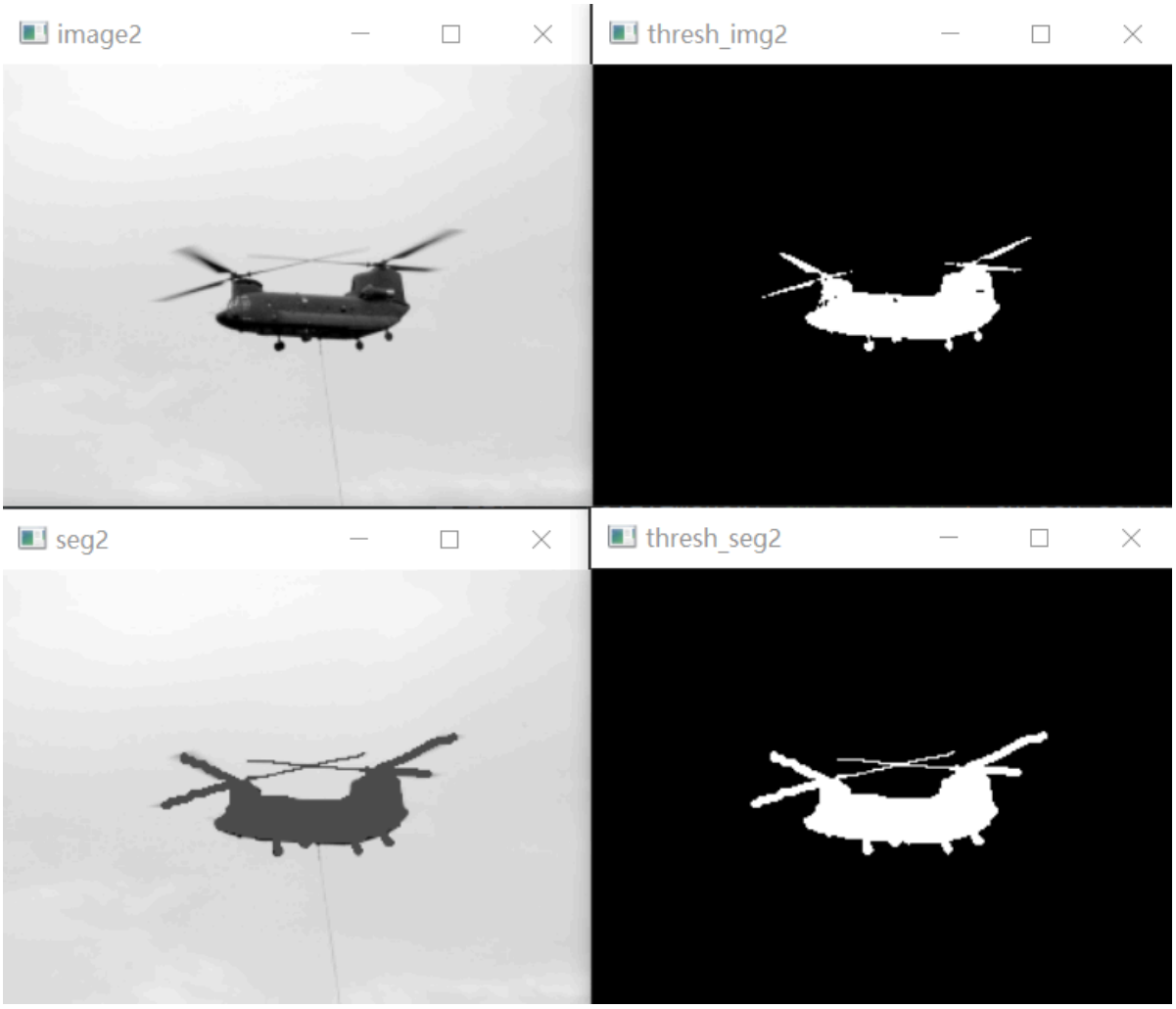
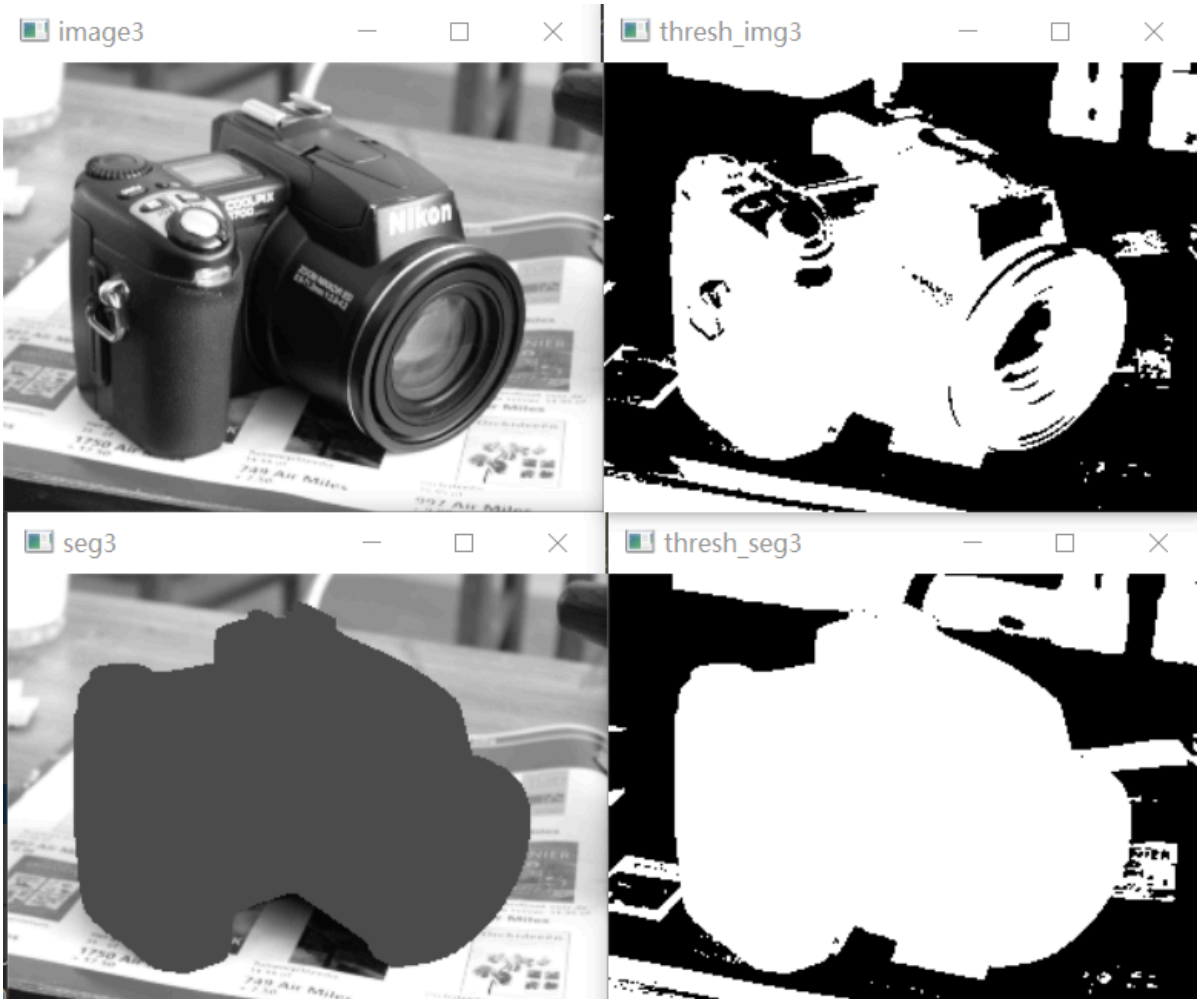
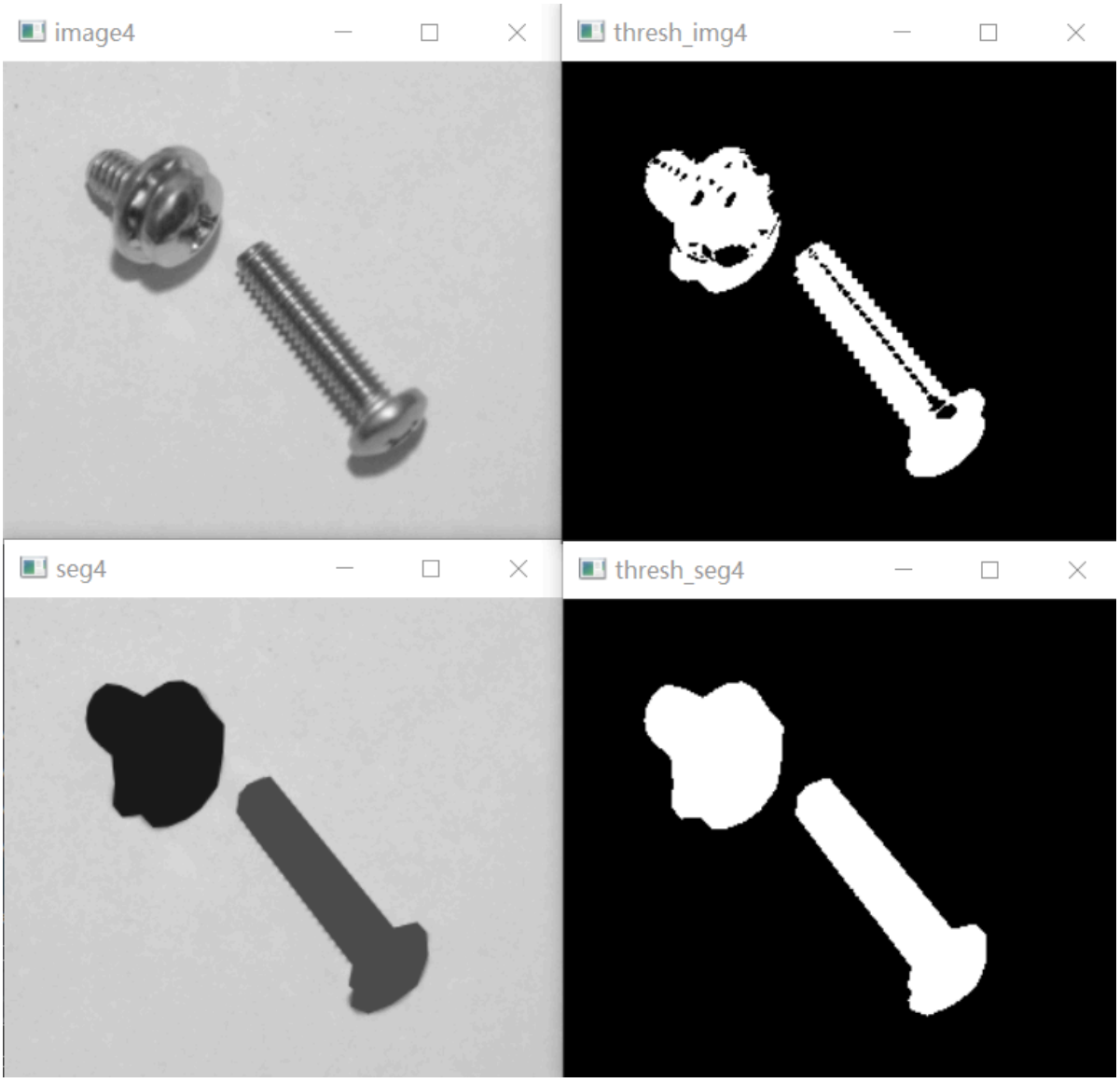


image2
thresh_img2
seg2
thresh_seg2

img3



img4



分割成功率

```
Image1
Jaccard指数为： 0.971164780419126
Dice指数为： 0.9853714819444254
平均分割效果为： 0.9782681311817757
Image2
Jaccard指数为： 0.8119209388511427
Dice指数为： 0.8961990795977501
平均分割效果为： 0.8540600092244464
Image3
Jaccard指数为： 0.7930461639252205
Dice指数为： 0.884579750238148
平均分割效果为： 0.8388129570816842
Image4
Jaccard指数为： 0.8429325378614043
Dice指数为： 0.9147730809935878
平均分割效果为： 0.8788528094274961
```

根据结果给出的四张图上基于阈值进行分割的分割效果，可以看出基于阈值进行图像分割的方法确实在对比度较高的图片（需要分割的前景部分和背景对比度高）上表现较好，在这种情况下可以展示较多的图片细节。然而当图片本身具有一定光影效果时，如 img3 的相机本体和反光部分就被强行分开成前景和后景，难以整个分割出来，这张图的分割效果也是几张图里最差的。

2.基于边缘的分割

基于Canny边缘检测的图像分割方法是一种传统的分割方法，其基本思路是利用边缘检测算法将图像中的物体轮廓提取出来，从而实现分割。

实现步骤

基于Canny边缘检测的图像分割方法的具体流程如下：

1. 对原始图像进行灰度变换，将RGB三通道的图像转换成灰度图像。
2. 对灰度图像进行高斯滤波，以去除噪声。
3. 计算滤波后的灰度图像的最优阈值，作为后面步骤中双阈值处理的参考值。
4. 使用Canny算法计算出图像的边缘，进行双阈值处理，将像素分为强边缘、弱边缘和非边缘三类；对弱边缘进行边缘跟踪，将弱边缘与强边缘相连的像素也作为边缘。
5. 得到分割后的图像并计算分割效果。

代码与结果

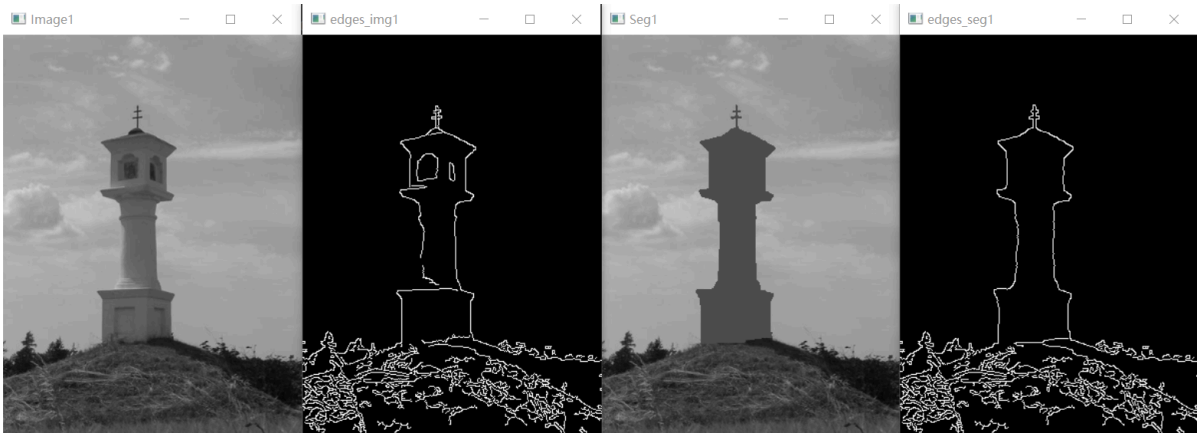
寻找图像边缘的最优阈值使用以下代码：

```
# 计算图像灰度直方图
hist = cv2.calcHist([blur_img1], [0], None, [256], [0, 256])
# 寻找最优阈值
total_pixels = blur_img1.shape[0] * blur_img1.shape[1]
sum_pix = 0
max_var = 0
best_thresh = 0
for i in range(256):
    sum_pix += i * hist[i]
    w0 = np.sum(hist[:i+1]) / total_pixels
    w1 = 1 - w0
    if w0 == 0 or w1 == 0:
        continue
    mean0 = sum_pix / (w0 * total_pixels)
    mean1 = (np.sum(hist[i+1:] * np.arange(i+1, 256)) / (w1 * total_pixels))
    var = w0 * w1 * (mean0 - mean1) ** 2
    if var > max_var:
        max_var = var
        best_thresh = i
```

img1

```
# 读取图像print("Image1")
img1 = cv2.imread('Image1.png')
seg1 = cv2.imread('Seg1.png')
gray_img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray_seg1 = cv2.cvtColor(seg1, cv2.COLOR_BGR2GRAY)
```

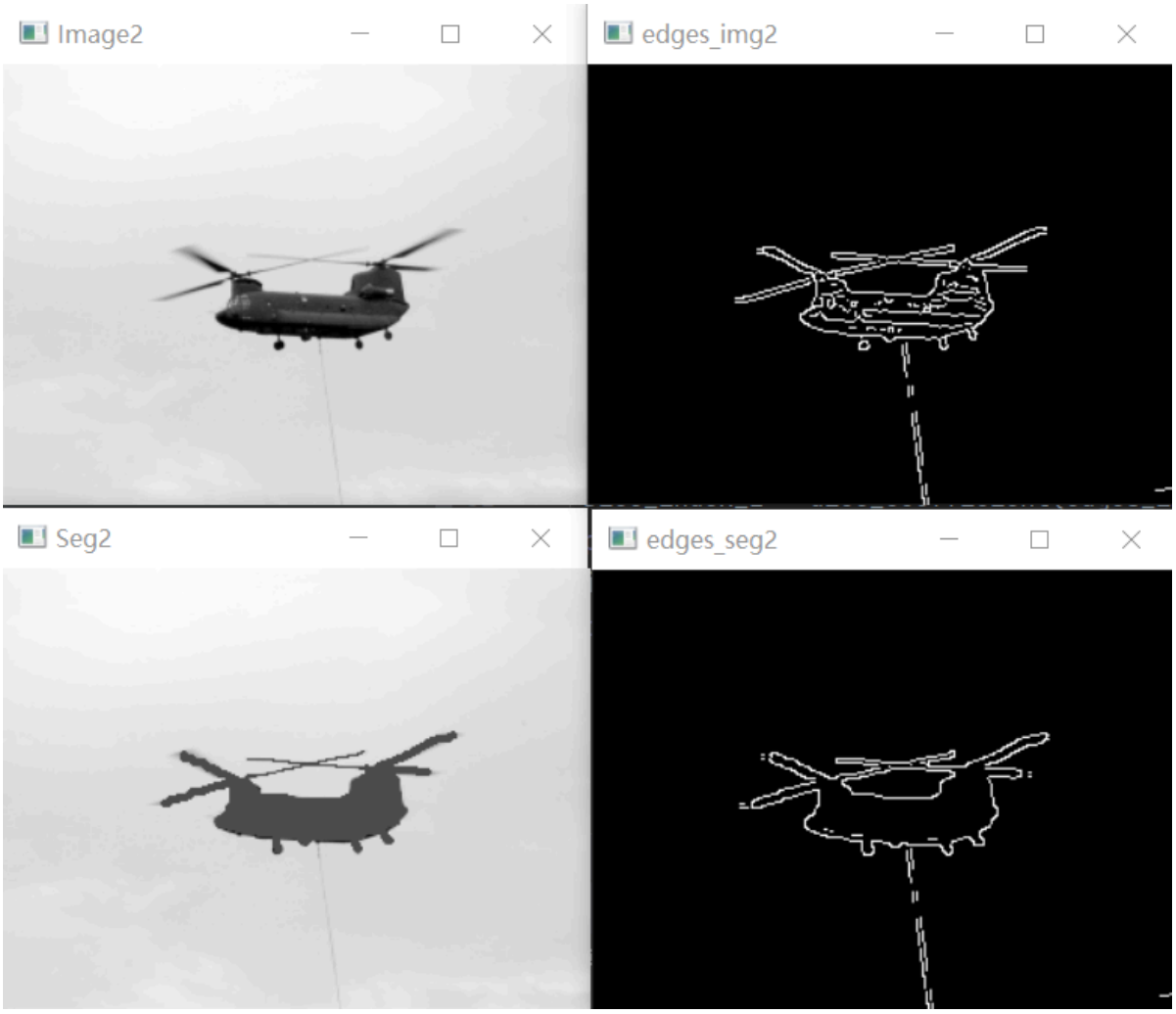
```
# 计算高斯滤波
blur_img1 = cv2.GaussianBlur(gray_img1, (3, 3), 0)
blur_seg1 = cv2.GaussianBlur(gray_seg1, (3, 3), 0)
# 应用Canny边缘检测
print(best_thresh)
edges_img1 = cv2.Canny(blur_img1, best_thresh * 0.4, best_thresh * 1.3)
edges_seg1 = cv2.Canny(blur_seg1, best_thresh * 0.4, best_thresh * 1.3)
# 显示结果
cv2.imshow('Image1', img1)
cv2.imshow('Seg1', seg1)
cv2.imshow('edges_img1', edges_img1)
cv2.imshow('edges_seg1', edges_seg1)
cv2.waitKey(0)
cv2.destroyAllWindows()
Jaccard_index_1 = jaccard_index(edges_img1, edges_seg1)
print("Jaccard指数为：", Jaccard_index_1)
Dice_index_1 = dice_coefficient(edges_img1, edges_seg1)
print("Dice指数为：", Dice_index_1)
index_1 = (Jaccard_index_1 + Dice_index_1)/2
print("平均分割效果为：", index_1)
```




```
Image1
99
Jaccard指数为： 0.8273489932885906
Dice指数为： 0.9055183178771463
平均分割效果为： 0.8664336555828684
```

img2

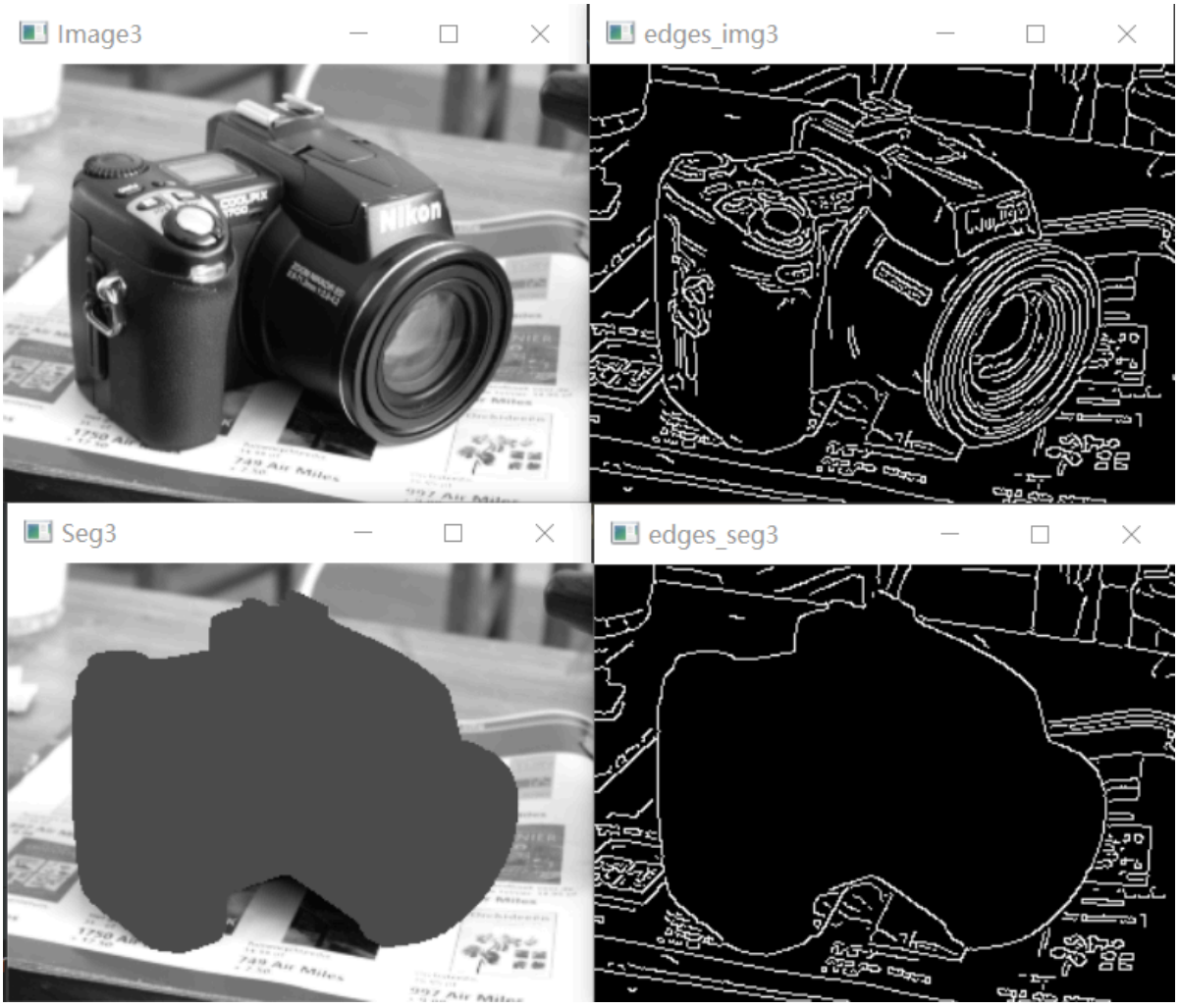
```
edges_img2 = cv2.Canny(gray_img2, best_thresh * 0.3, best_thresh * 0.4)
```




```
Image2
87
Jaccard指数为： 0.33217592592592593
Dice指数为： 0.49869678540399653
平均分割效果为： 0.41543635566496123
```

img3

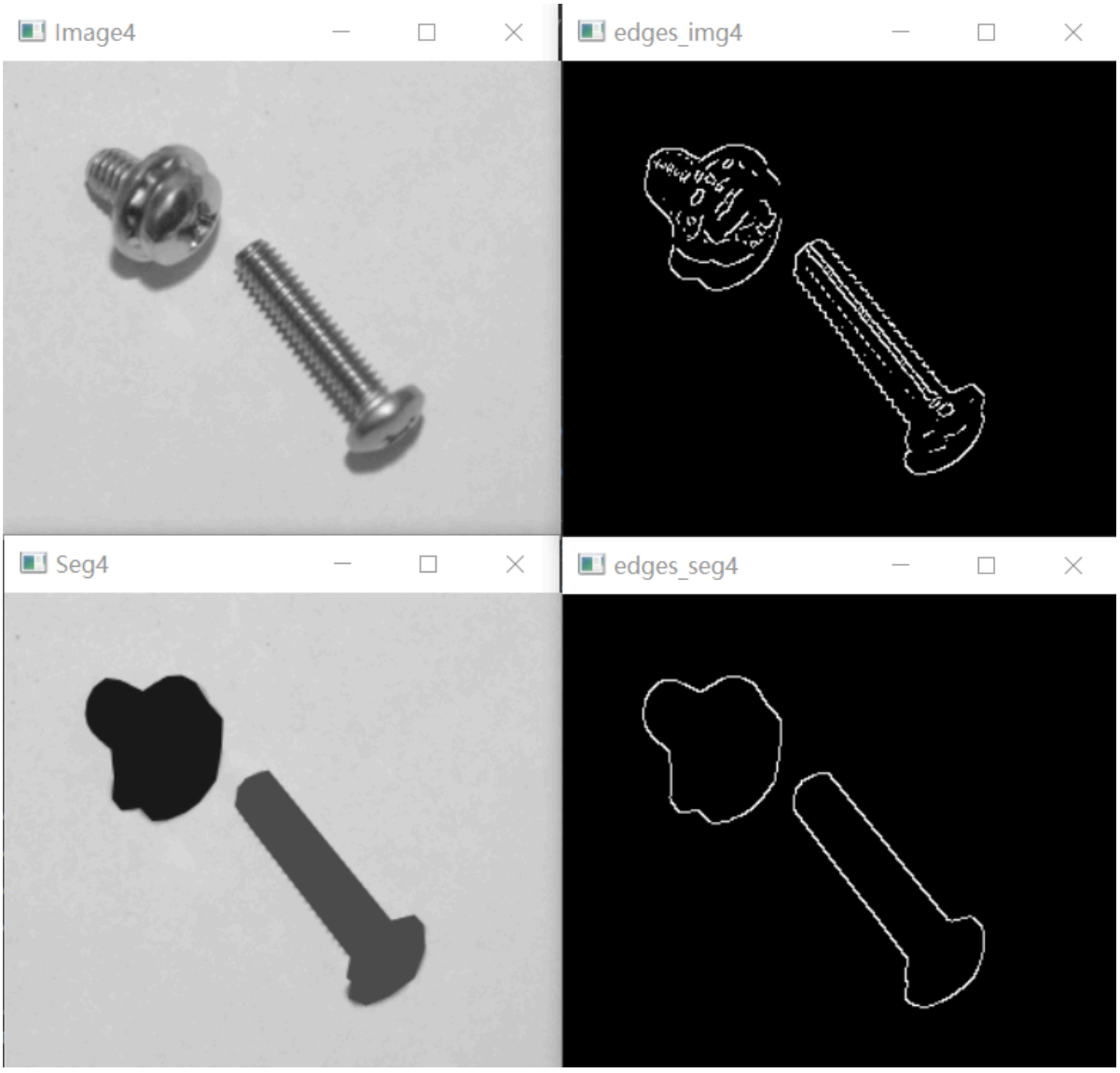
```
edges_img3 = cv2.Canny(gray_img3, best_thresh * 0.7, best_thresh * 0.9)
```




```
Image3
76
Jaccard指数为： 0.41388916525718833
Dice指数为： 0.585461966082612
平均分割效果为： 0.4996755656699002
```

img4

```
edges_img4 = cv2.Canny(gray_img4, best_thresh * 2.0, best_thresh * 2.0)
```




```
Image4
77
Jaccard指数为： 0.15228174603174602
Dice指数为： 0.26431338786052516
平均分割效果为： 0.2082975669461356
```

可以看出由于基于边界的分割对于边界非常敏感，可能会将目标分割对象的内部也识别出对应的边缘（如 img2 和 img3），同样在 img4 中，由于螺丝钉的锯齿在待分割图像中较明显而目标图像中覆盖了锯齿，因此造成二者尽管肉眼看相似度较高，但是使用 Jaccard 指数和 Dice 指数计算出的相似度较低。

该分割方法的准确度普遍较低也可能是因为以下两个原因：

- 和 img4 中体现出的一样，分割算法对边缘，尤其是锯齿状边缘较为敏感，直接导致锯齿边缘的像素点与目标图像边缘的像素点重合度不高。
- 算法无法识别边缘是在待分割物品的真实边缘还是物品内部，造成边缘的冗余。

总而言之，基于边缘的分割对于边缘较为敏感导致了它不够稳健、对图像本身的要求较高。

3.基于区域增长的分割

区域增长法是一种基于像素相似性的图像分割方法，其基本思想是从种子点开始，将与种子点相邻的像素逐个加入到同一区域中，直到所有像素都被分配到某个区域为止。这种方法适用于具有相似颜色或纹理的区域。

实现步骤

使用区域增长法进行图像分割的具体步骤如下：

- 选择种子点：首先需要选择一个或多个种子点，作为区域增长的起点。种子点可以手动选择，也可以通过自动算法选择。
- 确定相似性准则：在区域增长过程中，需要确定像素之间的相似性准则，以便将相似的像素加入到同一区域中。常用的相似性准则包括像素灰度值之差、像素颜色之差、像素纹理特征等。
- 确定生长准则：在区域增长过程中，需要确定像素加入区域的生长准则，以便控制区域的生长方向和速度。常用的生长准则包括像素与区域的相似性、像素与种子点的距离、像素与区域边界的距离等。
- 区域增长：从种子点开始，按照相似性准则和生长准则，逐个将与当前区域相邻的像素加入到同一区域中，直到所有像素都被分配到某个区域为止。
- 后处理：对分割结果进行后处理，包括去噪、填补空洞、分割边界平滑等。

代码与结果

首先是区域增长的自编函数代码，其中针对不同的图片可以调整的参数是 threshold 阈值。

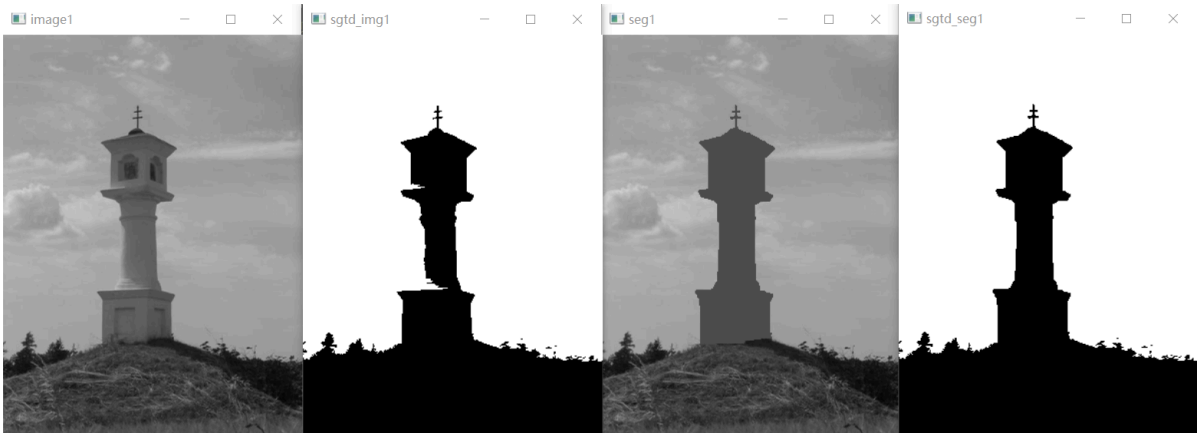
```
import cv2
import numpy as np

def region_growing(img, seed):
    threshold = 66
    seed_value = img[seed[0], seed[1]]
    rows, cols = img.shape[:2]
    segmented = np.zeros_like(img)
    segmented[seed[0], seed[1]] = 255
    queue = []
    queue.append(seed)
    while queue:
        current_point = queue.pop(0)
        for i in range(-1, 2):
            for j in range(-1, 2):
                if i == 0 and j == 0:
                    continue
                if current_point[0] + i < 0 or current_point[0] + i >= rows or
current_point[1] + j < 0 or \
                        current_point[1] + j >= cols:
                    continue
                diff = abs(int(img[current_point[0] + i, current_point[1] + j]) -
int(seed_value))
                if diff < threshold and segmented[current_point[0] + i,
current_point[1] + j] == 0:
                    segmented[current_point[0] + i, current_point[1] + j] = 255
                    queue.append((current_point[0] + i, current_point[1] + j))
    return segmented
```

下面可以根据自编的函数对示例图片进行分割，这段代码中可以调整的是种子点的位置选取，可以通过图片大小以及种子位置的灰度值和 threshold 阈值来进行判断和调整。

img1

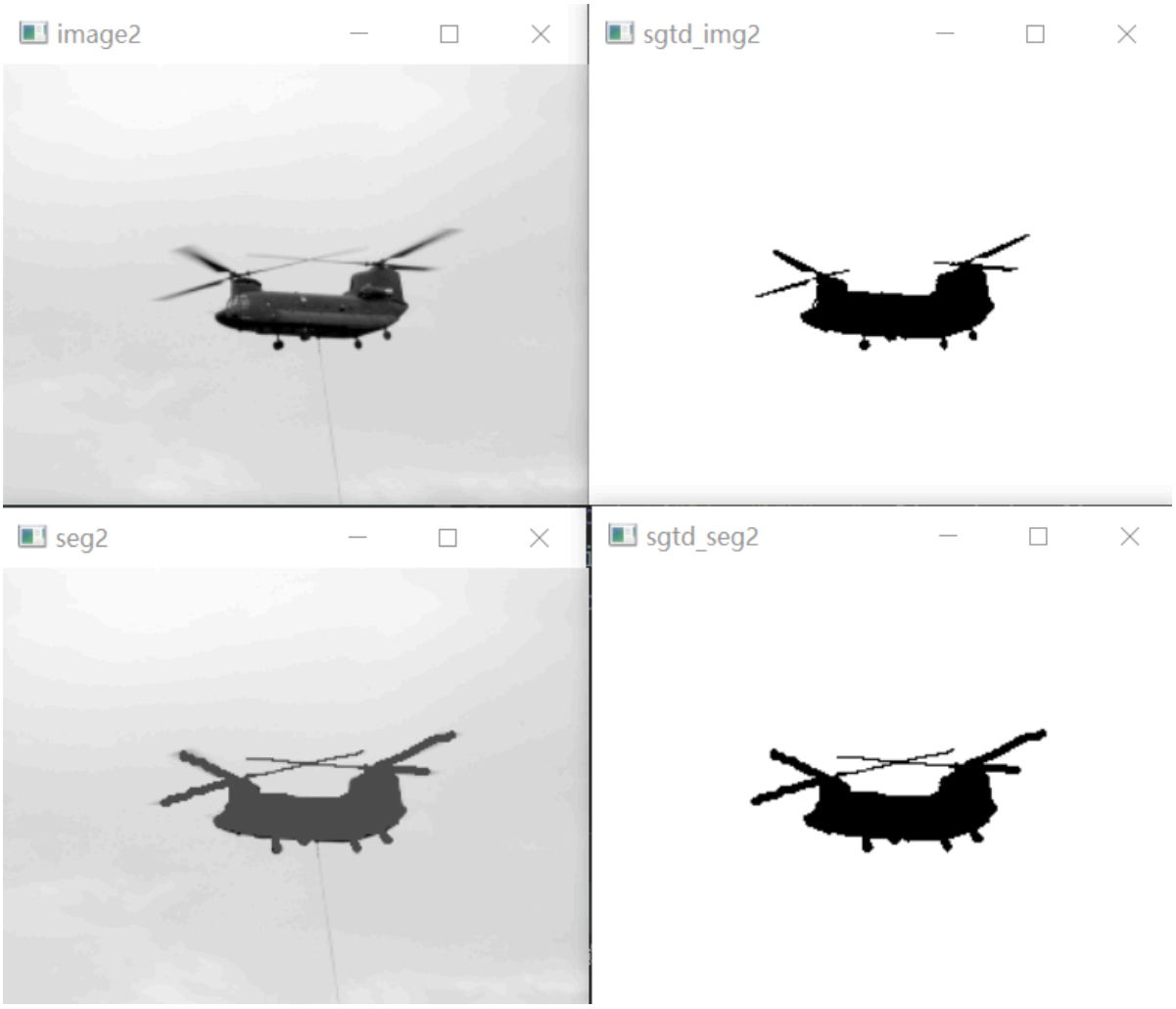
```
print("Image1")
img1 = cv2.imread('Image1.png')
seg1 = cv2.imread('Seg1.png')
height1, width1, channels1 = img1.shape
print('Image size: {} x {} pixels'.format(width1, height1)) # 查看图片大小
gray_img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray_seg1 = cv2.cvtColor(seg1, cv2.COLOR_BGR2GRAY)
# 选取种子点
x1, y1 = 150, 145
pixel_value = img1[y1, x1]
print('Pixel value at ({}, {}): {}'.format(x1, y1, pixel_value))
seed1 = (x1, y1)
# 进行区域增长
sgtd_img1 = region_growing(gray_img1, seed1)
sgtd_seg1 = region_growing(gray_seg1, seed1)
# 显示分割结果
cv2.imshow('image1', img1)
cv2.imshow('seg1', seg1)
cv2.imshow('sgtd_img1', sgtd_img1)
cv2.imshow('sgtd_seg1', sgtd_seg1)
cv2.waitKey(0)
cv2.destroyAllWindows()
# 计算分割效果
Jaccard_index_1 = jaccard_index(sgtd_img1, sgtd_seg1)
print("Jaccard指数为：", Jaccard_index_1)
Dice_index_1 = dice_coefficient(sgtd_img1, sgtd_seg1)
print("Dice指数为：", Dice_index_1)
index_1 = (Jaccard_index_1 + Dice_index_1)/2
print("平均分割效果为：", index_1)
```




```
Image1
Image size: 300 x 400 pixels
Pixel value at (150, 145): [52 52 52]
Jaccard指数为： 0.9402530707327403
Dice指数为： 0.9692066307387953
平均分割效果为： 0.9547298507357678
```

可以看出对于 img1，在种子点取（150，145），threshold 取 66 时计算出的分割效果较好，是因为输入图像和理想分割图像分割结果较为相似，但实际上我们的目的是需要分割出特定的物品，该分割进行区域增长后把灯塔下面的土地也包括进来，这是由于土地灰度值较高，比起和背景天空来说和物品本身更为接近。但是由于灯塔本身也是有亮面和暗面且区分度较大，因此如果降低阈值会导致分割不出完整的灯塔。

img2

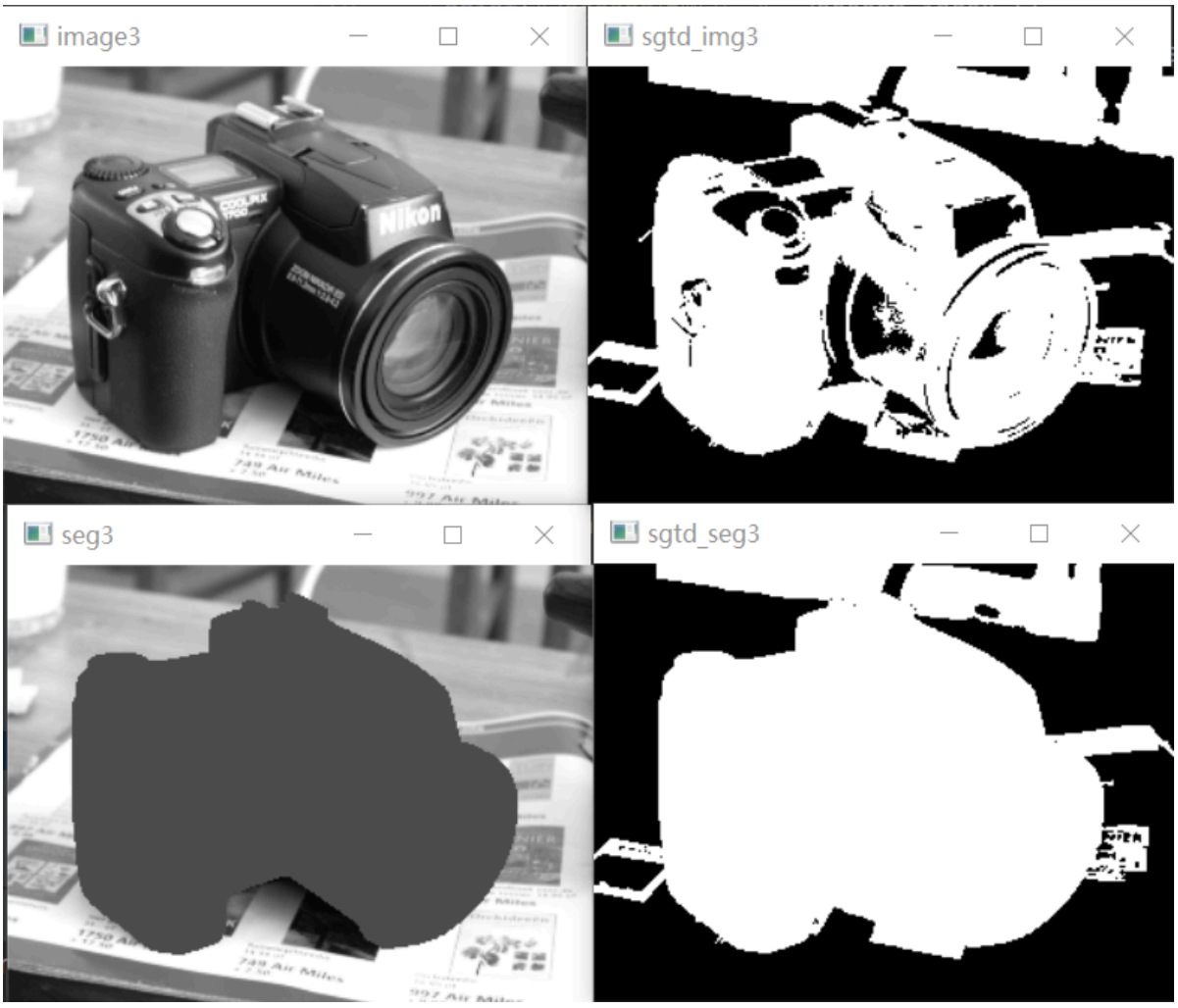



```
Image2
Image size: 300 x 225 pixels
Pixel value at (150, 145): [218 218 218]
Jaccard指数为： 0.9918807112867375
Dice指数为： 0.9959238077525147
平均分割效果为： 0.9939022595196261
```

img3

在针对 img3 进行调整的时候我们将区域增长函数的阈值提高到了75，种子点设置在x3, y3 = 99, 100

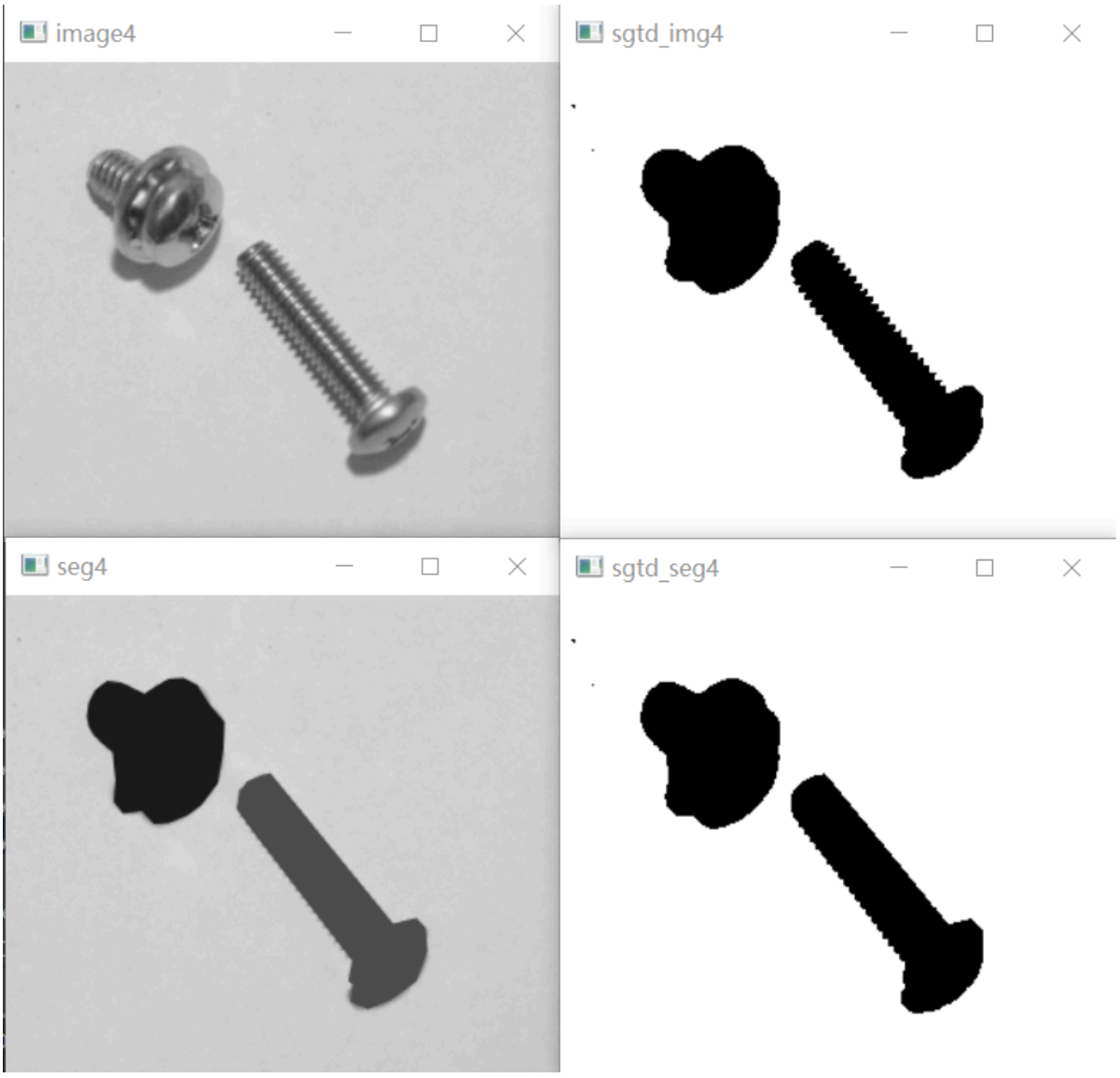
这是由于该图片需要分割的物品即使转化成灰度图像依旧可以看出形状较为不规则、在不同的面有反光的效果，因此需要较高的阈值覆盖反光区域。




```
Image3
Image size: 300 x 225 pixels
Pixel value at (99, 100): [63 63 63]
Jaccard指数为： 0.8157473331526693
Dice指数为： 0.8985251617980272
平均分割效果为： 0.8571362474753483
```

img4

在针对 img3 进行调整的时候我们将区域增长函数的阈值降低到20，种子点设置在x4, y4 = 150, 145




```
Image4
Image size: 300 x 258 pixels
Pixel value at (150, 145): [84 84 84]
Jaccard指数为： 0.9970338983050847
Dice指数为： 0.9985147464460005
平均分割效果为： 0.9977743223755426
```

区域增长的准确率主要取决于阈值的选择和种子点位置的选择，需要进行多次尝试才能找到最合适的参数。

4.基于深度学习的分割

U-Net原理

U-Net是一种用于图像分割任务的卷积神经网络，由Ronneberger等人在2015年提出。与传统的卷积神经网络不同，U-Net通过对称的编码器和解码器结构实现了跳跃式连接（skip connections），从而可以更好地保留图像中的细节信息，提高分割的准确性。

U-Net的网络结构包括以下几个部分：

- 编码器：U-Net的编码器部分包括多个卷积层和池化层，用于逐步降低图像的空间分辨率和通道数。每个卷积层都包括一个卷积操作、一个批归一化操作和一个激活函数，用于提取图像的特征表示。每个池化层用于将图像的空间分辨率减半，同时保留特征图的主要信息。

- 解码器：U-Net的解码器部分包括多个反卷积层和卷积层，用于逐步还原图像的空间分辨率和通道数。每个反卷积层都包括一个反卷积操作、一个批归一化操作和一个激活函数，用于将特征图的空间分辨率还原到原始大小。每个卷积层用于将特征图的通道数降低到最终的分类数目。
- 跳跃式连接：U-Net的编码器和解码器之间具有跳跃式连接，将编码器的某一层的特征图与解码器的对应层的特征图进行连接。这种跳跃式连接可以帮助网络恢复图像的细节信息，同时避免了解码器中的信息流失。
- 损失函数：U-Net的损失函数采用二元交叉熵损失函数，用于比较预测结果和真实标签之间的差异。优化器通常采用Adam优化器，用于最小化损失函数并更新网络参数。

实现步骤

U-Net 进行图像分割的思路如下：

1. 数据准备：首先需要准备一组带有标注的图像数据集，其中每张图像都有对应的标注图像，标注图像中的每个像素都标记了该像素属于哪个类别（如前景或背景）。
2. 网络结构：U-Net 是一种基于卷积神经网络的图像分割模型，其网络结构由编码器和解码器两部分组成。编码器部分用于提取图像特征，解码器部分用于将特征图还原为原始图像大小，并生成分割结果。U-Net 的特点是在解码器部分使用了跳跃连接，可以帮助网络更好地保留图像细节信息。
3. 训练模型：使用准备好的数据集对 U-Net 进行训练，目标是使模型能够准确地将输入图像分割成对应的类别。训练过程中需要选择合适的损失函数，常用的有交叉熵损失函数和 Dice 损失函数等。
4. 预测分割结果：训练完成后，可以使用 U-Net 对新的图像进行分割。具体方法是将待分割的图像输入到 U-Net 中，得到对应的分割结果。

分割结果可以是像素级别的标注，也可以是对每个区域进行分类的结果。需要注意的是，U-Net 是一种比较经典的图像分割模型，但并不是适用于所有场景的最佳选择。在实际应用中，需要根据具体情况选择合适的模型和算法。

代码与结果

模型搭建。

```
from keras.models import *
from keras.layers import *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras import backend as keras
from sklearn.metrics import jaccard_score
from sklearn.metrics import f1_score
from sklearn.preprocessing import Binarizer
import cv2
import numpy as np

def unet(pretrained_weights = None,input_size = (256,256,1)):
    # 输入层
    inputs = Input(input_size)

    # 编码器
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(inputs)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
```

```
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool1)

    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv4)
    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool4)
    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv5)
    drop5 = Dropout(0.5)(conv5)

    # 解码器
    up6 = Conv2D(512, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(drop5))
    merge6 = concatenate([drop4,up6], axis = 3)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge6)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv6)

    up7 = Conv2D(256, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv6))
    merge7 = concatenate([conv3,up7], axis = 3)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge7)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv7)

    up8 = Conv2D(128, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv7))
    merge8 = concatenate([conv2,up8], axis = 3)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge8)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv8)

    up9 = Conv2D(64, 2, activation = 'relu', padding = 'same', kernel_initializer
= 'he_normal')(UpSampling2D(size = (2,2))(conv8))
    merge9 = concatenate([conv1,up9], axis = 3)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge9)
```

```
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv9)
    conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv9)

    # 输出层
    conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

    # 创建模型
    model = Model(inputs = inputs, outputs = conv10)

    # 编译模型
    model.compile(optimizer = Adam(lr = 1e-4), loss = 'binary_crossentropy',
metrics = ['accuracy'])

    # 加载预训练权重
    if(pretrained_weights):
        model.load_weights(pretrained_weights)

    return model
```

img1

```
# 准备数据
img1 = cv2.imread('/content/sample_data/Image1.png')
img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img1 = cv2.resize(img1, (256, 256))
img1 = np.expand_dims(img1, axis=0)
seg1 = cv2.imread('/content/sample_data/Seg1.png')
seg1 = cv2.cvtColor(seg1, cv2.COLOR_BGR2GRAY)
seg1 = cv2.resize(seg1, (256, 256))
seg1 = np.expand_dims(seg1, axis=0)
seg1 = np.expand_dims(seg1, axis=3)
# 加载模型
model = unet(pretrained_weights = None,input_size = (256,256,1))
# 进行预测
pred_img1 = model.predict(img1)
pred_seg1 = model.predict(seg1)
# 评估结果
pred_img1[pred_img1 >= 0.5] = 1
pred_img1[pred_img1 < 0.5] = 0
pred_seg1[pred_seg1 >= 0.5] = 1
pred_seg1[pred_seg1 < 0.5] = 0

from sklearn.preprocessing import Binarizer
bi = Binarizer(threshold=0.5)
pred_img1_2d = np.reshape(pred_img1, (pred_img1.shape[0]*pred_img1.shape[1],
pred_img1.shape[2]))
pred_img1_2d = bi.transform(pred_img1_2d)
pred_seg1_2d = np.reshape(pred_seg1, (pred_seg1.shape[0]*pred_seg1.shape[1],
pred_seg1.shape[2]))
pred_seg1_2d = bi.transform(pred_seg1_2d)
jaccard_index = jaccard_index(pred_img1_2d, pred_seg1_2d)
print("Jaccard指数为：", jaccard_index)
dice_index = dice_coefficient(pred_img1_2d, pred_seg1_2d)
```

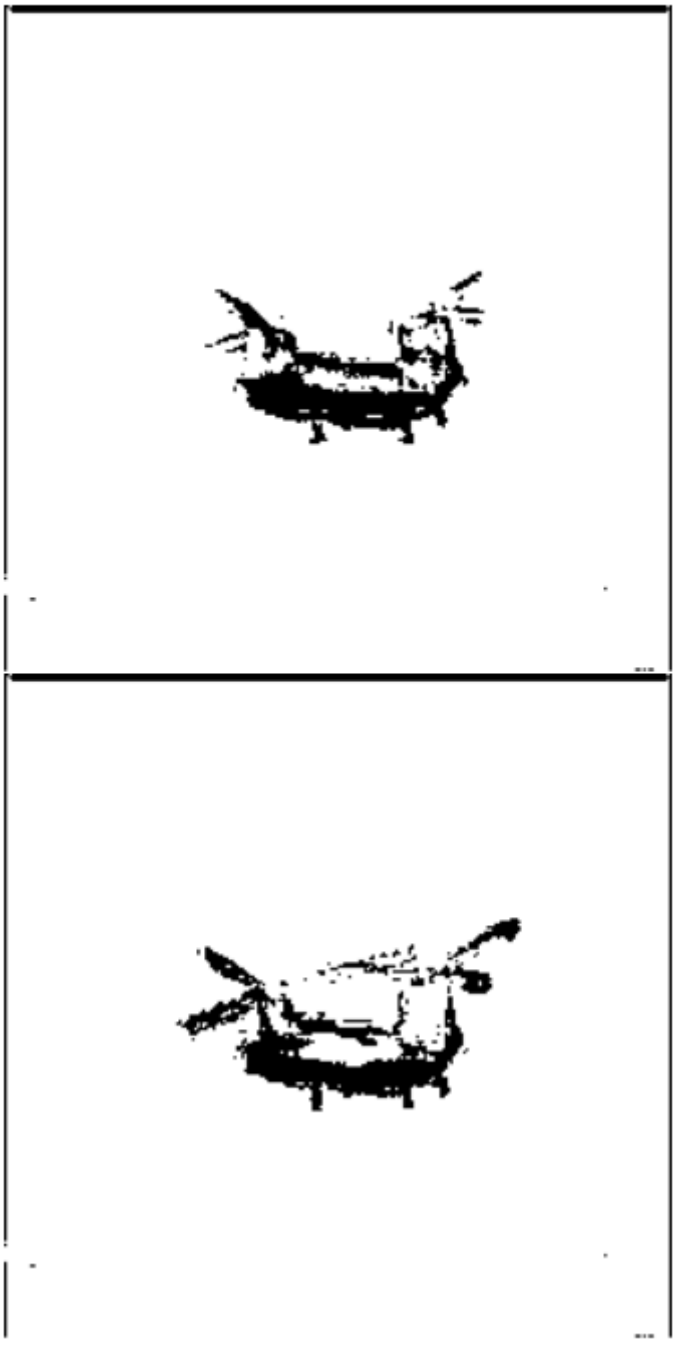
```
print("Dice指数为：", dice_index)
index = (jaccard_index + dice_index)/2
print("平均分割效果为：", index)

show_img1 = pred_img1_2d*255
show_seg1 = pred_seg1_2d*255
cv2_imshow(show_img1)
cv2_imshow(show_seg1)
```

```
Jaccard指数为：  0.9521396273400468
Dice指数为：  0.9754831201673996
平均分割效果为：  0.9638113737537232
```

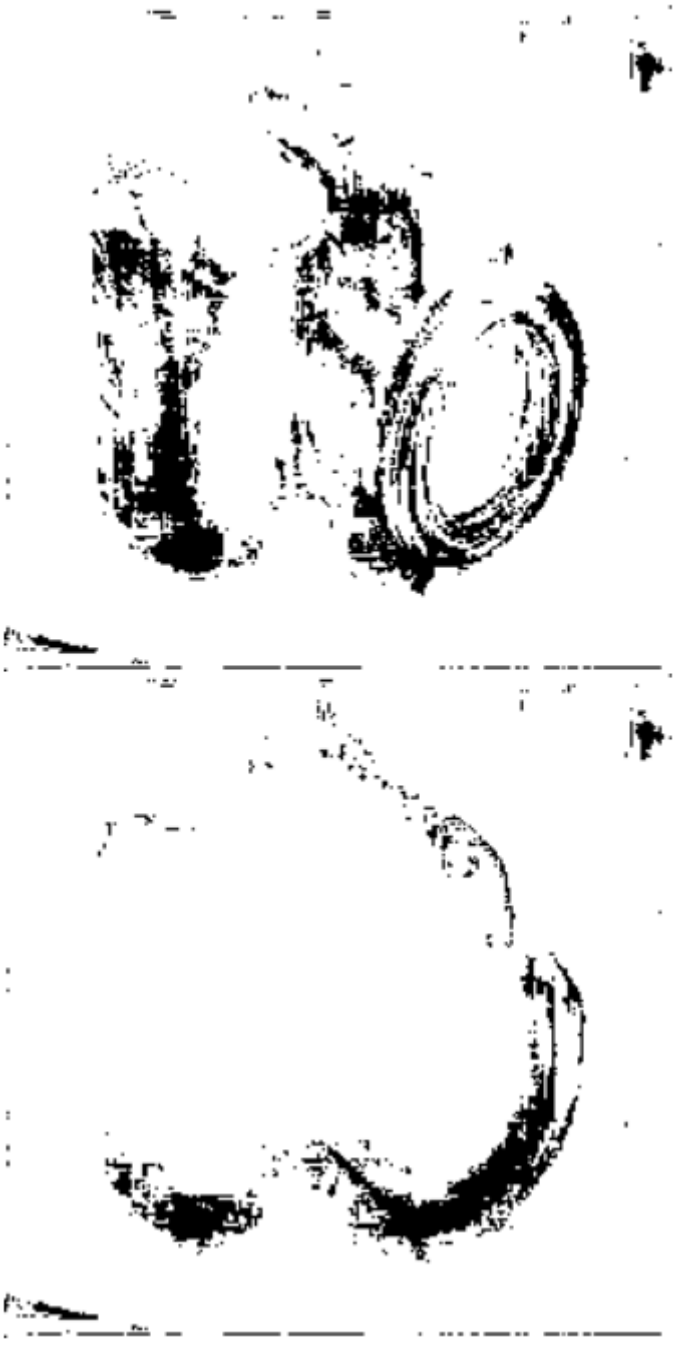
img2

Jaccard指数为： 0.9802014491499665
Dice指数为： 0.9898661885287551
平均分割效果为： 0.9850338188393608

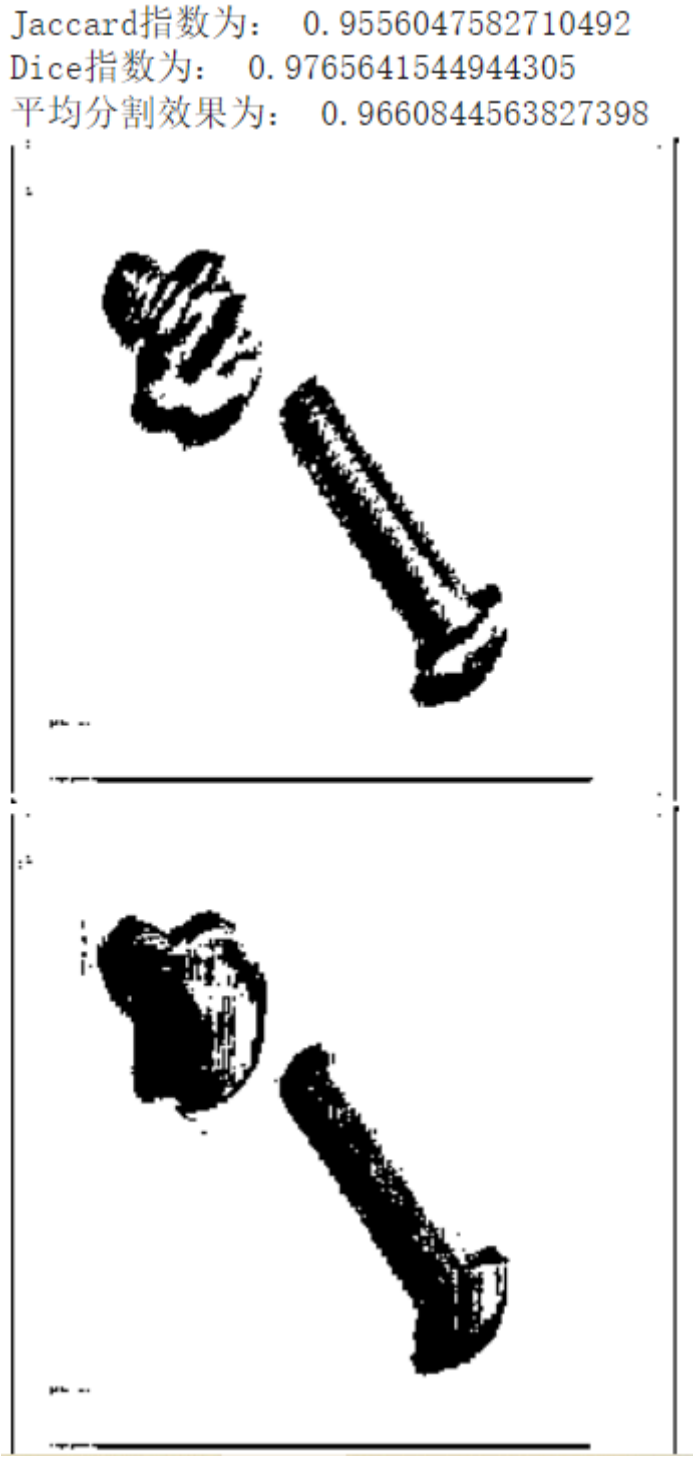


img3

Jaccard指数为： 0.9107304416438978
Dice指数为： 0.9510335385103077
平均分割效果为： 0.9308819900771028



img4



总结与反思

本次作业中使用四种方法，分别基于阈值、边缘、区域增长和深度学习的方法对图像进行分割，并且使用 Jaccard 和 Dice 两种评价方法对分割结果做出评价，在四组图片和四种方法互相之间进行了比较。然而根据对于图像分割这一命题而言，无论是使用 Jaccard 、 Dice 或者 PSNR、SSIM的评价准则都显得不够科学，因为这些评价方式都是用于判断两张图片的“相似”程度，可以在图像去噪这方面能够发挥较好的作用，但是并不是完全适配图像分割的评价。

在本文中使用 Jaccard 或者 Dice 的方法都是对输入图像与理想分割结果分别做出相同的操作之后比较两张图片的相似程度，并非真正意义上的“分割效果”。如在基于阈值的分割和基于区域增长的分割两种方式中 Image1 和 Seg1 的最终相似度很高，但是经过处理之后的前景区域都包含了“草地”的部分，而不是我们所希望的只有“灯塔”部分，这是由于灰度图像中“草地”的部分灰度值与“灯塔”的部分灰度值相似的

原因，但是依旧给“分割出特定部分”的实验目的造成困扰。因此在基于边缘的分割部分中考虑将边缘围成的区域进行上色之后再比较，但是并没有成功。基于边缘的分割中由于边缘的判定较为敏感，细微的边缘线位置不同就会导致 Jaccard 或者 Dice 指数明显低于其他方法，且基于边缘的分割方法同样无法识别我们希望要分割的目标物品，在目标物品内部的明暗交界处也会产生很多边缘界限，这是该方法的 Jaccard 或者 Dice 指数较低的另一个原因。

此外，还尝试搭建 U-Net 网络对图像进行分割，取得的效果没有明显好于前面的方法，可能是参数调整不到位的原因。