指纹图像提取项目文档

项目背景

从iPhone5s搭载TouchID之后，指纹身份验证已经普及进了家家户户，从前令人难以琢磨的指纹信息与人身份的匹配问题也被科学家击破。

本次实验中，我们先不考虑如何对不同的指纹进行唯一的识别，我们先来看一步更重要的部分——如何进行指纹图像的提取，提取后的指纹效果关系到最终识别时到准确度和指纹信息的可靠性等多种因素，因此找到更好、更快的指纹图像提取算法十分重要。

阈值处理

一幅图像包括目标物体、背景还有噪声，要想从多值的数字图像中直接提取出目标物体，常用的方法就是设定一个阈值T，用T将图像的数据分成两部分：大于T的像素群和小于T的像素群。这是研究灰度变换的最特殊的方法，称为图像的二值化（Binarization）。

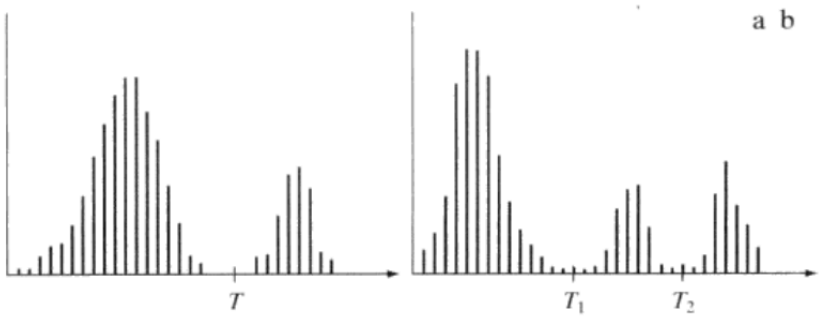
公式：

$$g(x,y)=\begin{cases}1, & f(x,y)>T\\ 0, & f(x,y)\le T\end{cases}$$

特点： 适用于目标与背景灰度有较强对比的情况，重要的是背景或物体的灰度比较单一，而且总可以得到封闭且连通区域的边界。

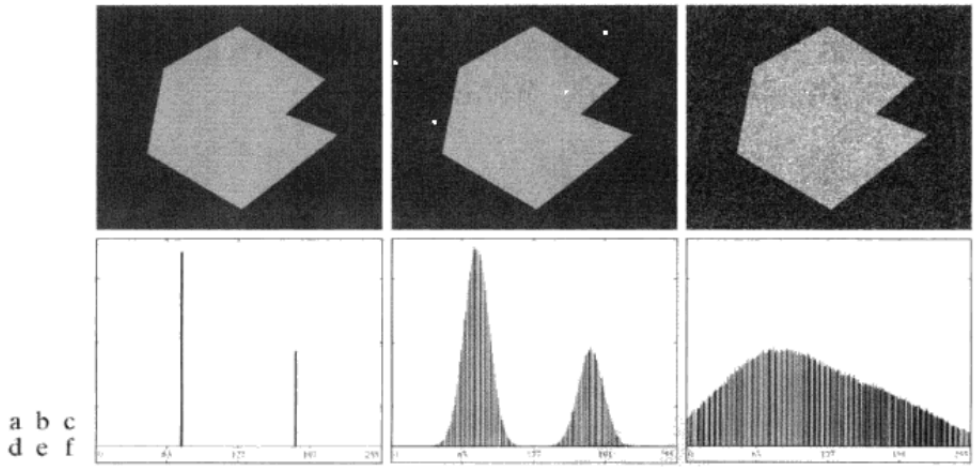
例子：

- 如图a中我们可以明显的看到像素在T附近具有明显界限，因此将T作为阈值，可以划分两部分的像素值，达到图像提取的目的
- 如图b中我们可以看到在T1和T2两处具有明显的界限，因此可以将0~T1, T1~T2, T2~255划分成三部分像素区间



图像阈值处理中噪声的作用

- **图像a：** 只有两种像素，没有任何招生，因此其直方图由两个波峰组成
  - 根据直方图d，对图a进行分割十分容易，选取两个模式之间的任何位置作为阈值都可以很好的进行图像分割
- **图像b：** 在原图中添加了均值为0、标准差为10个灰度级的高斯噪声
  - 根据直方图e，尽管相应的直方图模式较宽，但是仍可以明显看到分界，在两个波峰之间的中间位置的阈值可以很好的分割图像
- **图像c：** 在原图中施加均值为0、标准差为50个灰度级的高斯噪声
  - 根据直方图f，现在由于被噪声污染很严重，导致没法直观的寻找到合适的阈值进行图像分割

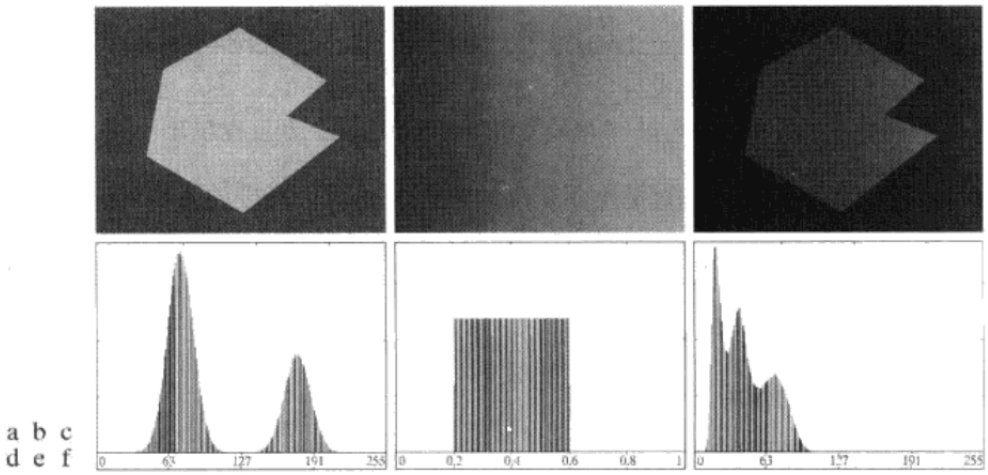


图像阈值处理中光照和反射的作用

- **图像a：** 上图中的图b
- **图像b：** 非均匀光照的影响

- **图像c**：图a与图b的乘积
  - 根据直方图f，波峰之间的较深的波谷在模式的分离点处被污染，如果没有附加的处理，分离事不可能的
  - 如果光照非常均匀，但图像的反射不均匀也会得到类似的结果

    > 例如. 物体的表面或背景自然翻身变化的情况就是如此



简单阈值实战

ret, dst = cv2.threshold(src, thresh, maxval, type)

- src： 输入图，只能输入单通道图像，通常来说为灰度图
- thresh： 阈值
- maxval： 当像素值超过了阈值（或者小于阈值，根据type来决定），所赋予的值
- type：二值化操作的类型，包含以下5种类型：
  - cv2.THRESH_BINARY ： 正向二值化,如果当前的像素值大于设置的阈值(thresh)，则将该点的像素值设置为maxval；否则，将该点的像素值设置为0；

    具体的公式如下：

    $$dst(x,y) = \begin{cases} MaxValue, & if\ src(x,y) > threshold \\ 0, & otherwise \end{cases}$$
  - cv2.THRESH_BINARY_INV ： 反向二值化，如果当前的像素值大于设置的阈值(thresh)，则将该点的像素值设置为0；否则，将该点的像素值设置为maxval； 具体的公式如下：

    $$dst(x,y) = \begin{cases} 0, & if\ src(x,y) > threshold \\ MaxValue, & otherwise \end{cases}$$
  - cv2.THRESH_TRUNC ： 如果当前的像素值大于设置的阈值(thresh)，则将该点的像素值设置为threshold；否则，将该点的像素值不变； 具体的公式如下：

    $$dst(x,y) = \begin{cases} threshold, & if\ src(x,y) > threshold \\ src(x,y), & otherwise \end{cases}$$

- cv2.THRESH_TOZERO ： 如果当前的像素值大于设置的阈值(thresh)，则将该点的像素值不变；否则，将该点的像素值设置为0； 具体的公式如下：

$$dst(x,y) = \begin{cases} src(x,y), & if\ src(x,y) > threshold \\ 0, & otherwise \end{cases}$$

- cv2.THRESH_TOZERO_INV：如果当前的像素值大于设置的阈值(thresh)，则将该点的像素值设置为0；否则，将该点的像素值不变； 具体的公式如下：

$$dst(x,y) = \begin{cases} 0, & if\ src(x,y) > threshold \\ src(x,y), & otherwise \end{cases}$$

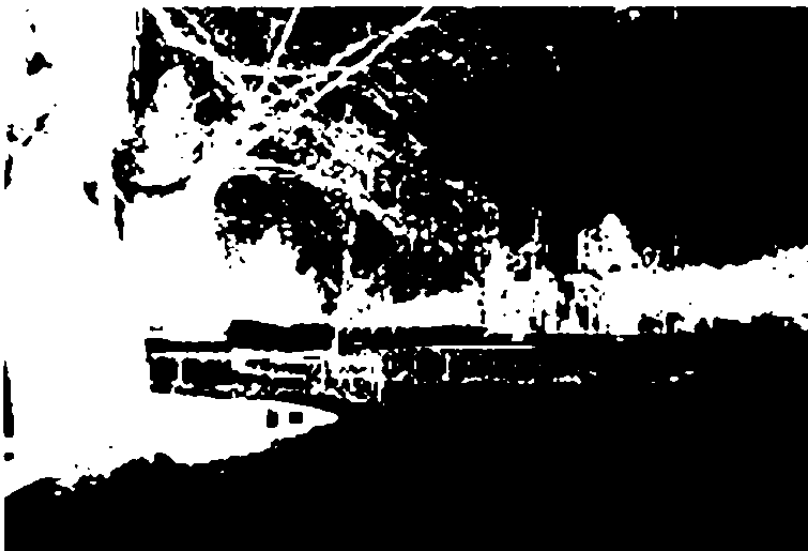
算法实现

```
def Basic_Thresholding(img):
    '''
    :description: 简单阈值处理

    :param img: 灰度图像

    :return : 阈值处理后的图像
    '''
    blurred = cv2.GaussianBlur(img, (5, 5), 0)       # 高斯滤波降噪
    (T, thresh_inv) = cv2.threshold(blurred, 127, 255, cv2.THRESH_BINARY_INV)
    return thresh_inv

image = cv2.imread('../Resources/snow.png')
gray = cv2.cvtColor(image, cv2.COLOR_BGRA2GRAY)     # rgb图像变换为灰度图像

thresh_inv = Basic_Thresholding(gray)   # 阈值处理后的图像

image_mask = cv2.bitwise_and(gray, gray, mask=thresh_inv)   # 在阈值处理后的图像上加掩膜
```

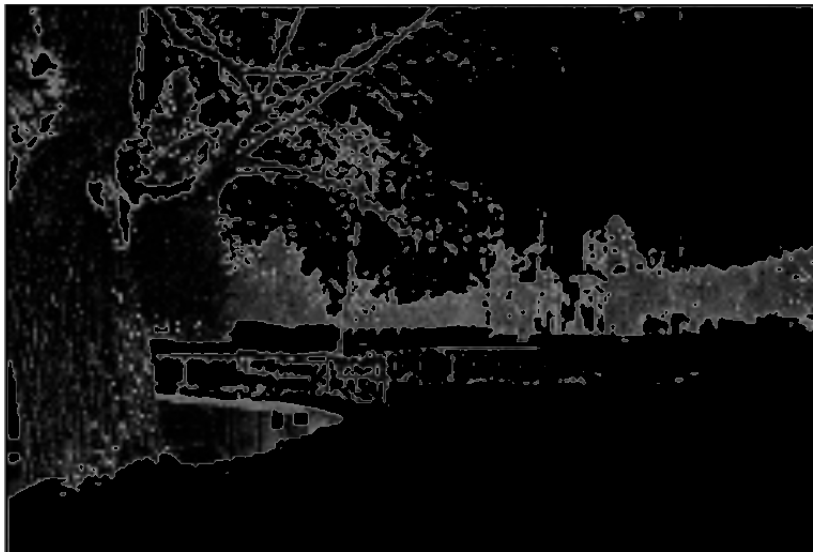
算法效果

原图

简单阈值处理后的图像



增加掩膜



自适应阈值实战

自适应阈值可以看成一种局部性的阈值，通过设定一个区域大小，比较这个点与区域大小里面像素点 的平均值（或者其他特征）的大小关系确定这个像素点的情况

cv2.adaptiveThreshold()

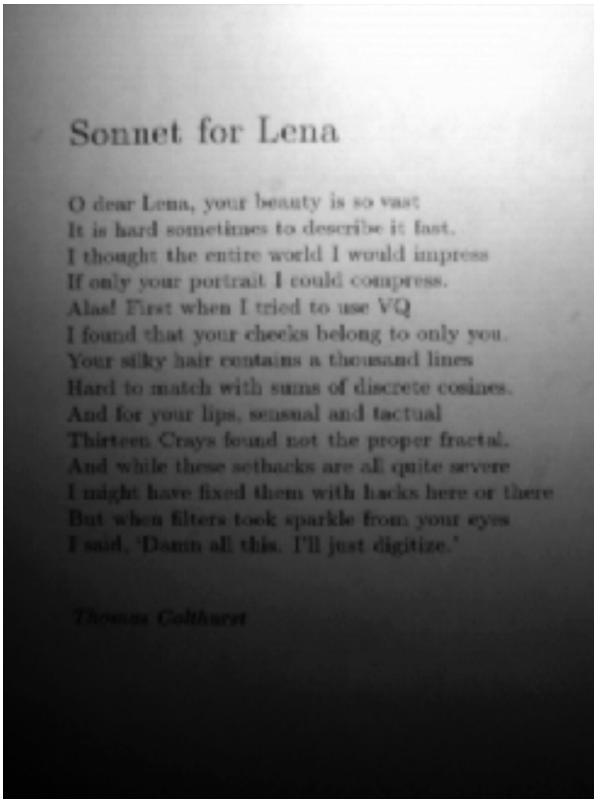
- **src**：原图，即输入图像，是一个8位单通道的图像；
- **maxValue**：分配给满足条件的像素的非零值；
- **adaptiveMethod**：自适应阈值的方法，通常有以下几种方法；
    1. **ADAPTIVE_THRESH_MEAN_C**，阈值T(x,y)是(x,y)减去C的Blocksize×Blocksize邻域的平均值
    2. **ADAPTIVE_THRESH_GAUSSIAN_C** ，阈值T(x，y)是(x，y)减去C的Blocksize×Blocksize邻域的加权和(与高斯相关)，默认sigma(标准差)用于指定的Blocksize；具体的情况可以参见getGaussianKernel函数；
- **thresholdType**：阈值的类型必须是以下两种类型，
    1. **THRESH_BINARY**，正向二值化
    2. **THRESH_BINARY_INV** ，反向二值化
- **blockSize**：像素邻域的大小，用来计算像素的阈值，blockSize必须为奇数，例如，3，5，7等等；
- **C**：从平均数或加权平均数减去常量。通常，它是正的，但也可能是零或负数。

算法实现

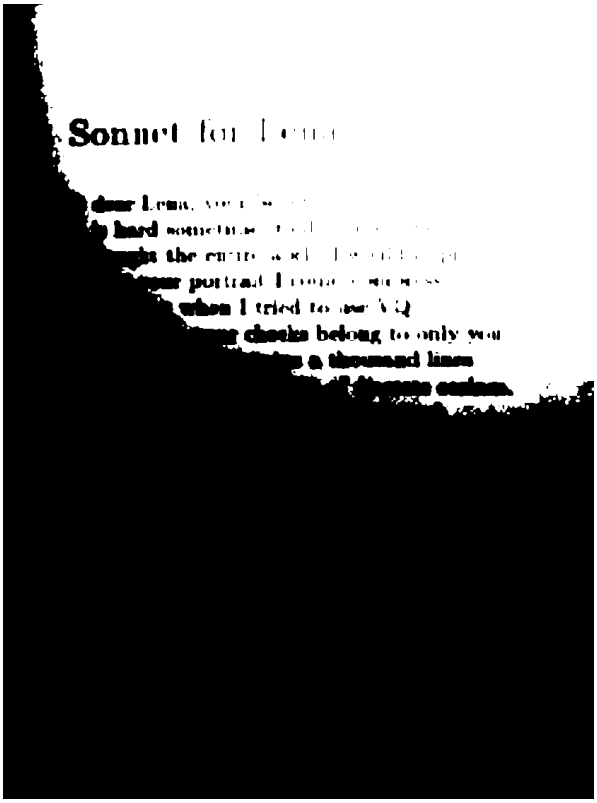
```
def Adaptive_Thresholding(image):
    '''
    :description: 自适应阈值处理

    :param image: 原图

    :return gray: 原图的灰度图像
    :return thresh: 简单阈值处理后的图像
    :return th1_1: adaptive Mean thresholding处理后的图像
    :return th2: adaptive Gaussian thresholding处理后的图像
    '''
    gray = cv2.cvtColor(image, cv2.COLOR_BGRA2GRAY)     # rgb图像变换为灰度图像

    blurred = cv2.medianBlur(gray, 1)       # 高斯滤波降噪

    (T, thresh) = cv2.threshold(blurred, 127, 255, cv2.THRESH_BINARY)   # 简单阈值处理
    th1 = cv2.adaptiveThreshold(gray, 255,
                                cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)   # 自适应阈值处理
    th1_1 = cv2.medianBlur(th1, 5)      # adaptive mean thresholding
    th2 = cv2.adaptiveThreshold(gray, 255,
                                cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)   # adaptive gaussian thresholding

    return [gray, thresh, th1_1, th2]
```

算法效果

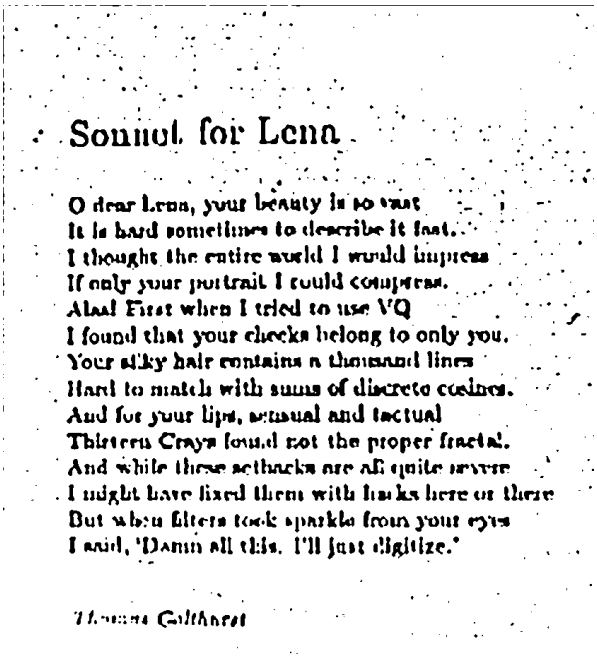
原图



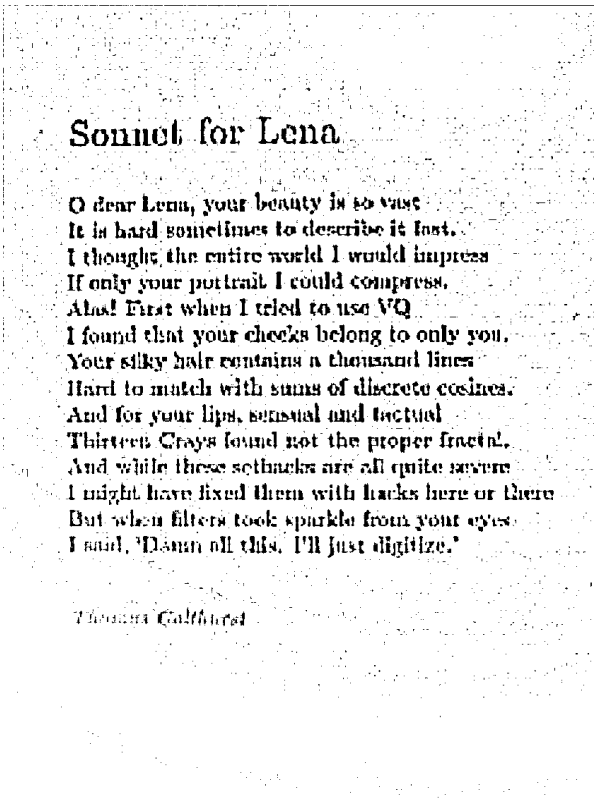
简单阈值处理



Adaptive Mean Thresholding 阈值处理



Adaptive Gaussian Thresholding 阈值处理



全局阈值处理实战

算法思想

1. 为全局阈值T选择一个初始估计值

2. 用公式进行阈值为T的分割，产生两组像素
    - G1由灰度值大于T的所有像素组成
    - G2由所有小雨等于T的像素组成
3. 对G1和G2对像素分别计算平均灰度值m1和m2
4. 计算一个新的阈值:

$$T = \frac{1}{2}(m_1 + m_2)$$

5. 重复步骤2～4，直到连续迭代中的T值间的差小于一个预定义的参数ΔT为止

算法实现

将rgb图像转化为灰度图像

```
def rgb2gray(img):
  '''
    :description: 将rgb图像转化为灰度图像

    :param img: 原始rgb图像

    :return : 转化后的灰度图像
    '''
  h, w = img.shape[0], img.shape[1]
  img1 = np.zeros((h,w),np.uint8)

  for i in range(h):
    for j in range(w):
      img1[i,j] = 0.144 * img[i,j,0] + 0.587 * img[i,j,1] + 0.299 * img[i,j,2]

      return img1
```

根据当前阈值计算新阈值

```
def threshold(img, T):
    '''
    :description: 根据当前阈值计算新阈值

    :param img: 灰度图像
    :param T: 当前的阈值

    :return : 计算得到的新阈值
    '''
    h, w = img.shape[0], img.shape[1]
    G1 = G2 = 0
    g1 = g2 = 0

    for i in range (h):
        for j in range (w):
```

```
            if img[i,j]>T:
                G1 += img[i,j]
                g1 += 1
            else:
                G2 += img[i,j]
                g2 += 1

    m1, m2 = int(G1/g1), int(G2/g2)   # m1，m2计算两组像素均值
    T0 = int((m1+m2)/2)   # 据公式计算新的阈值
    return T0
```

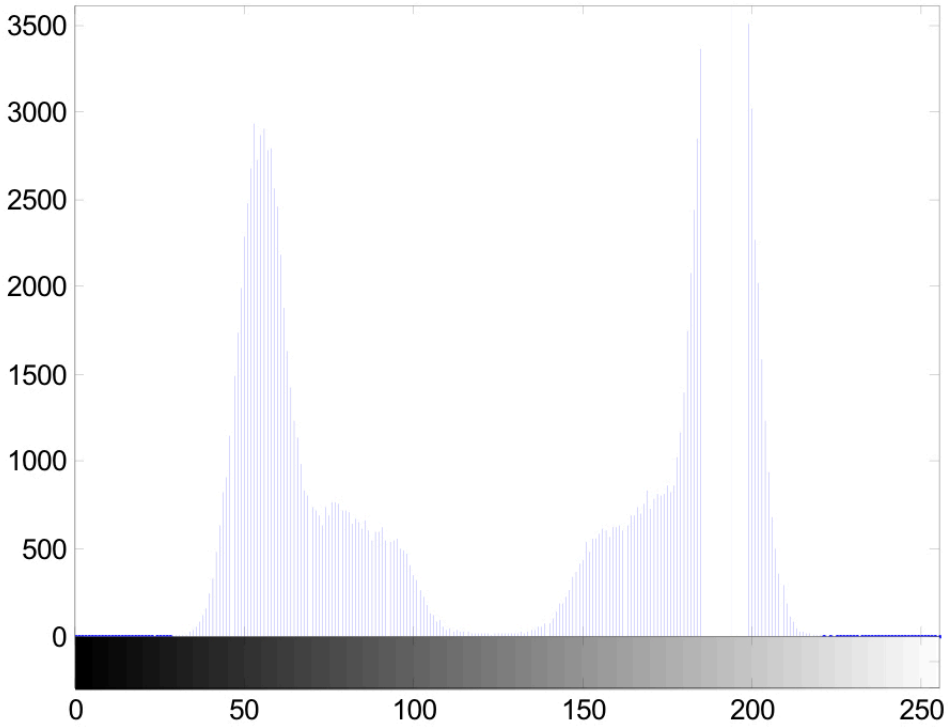
全局阈值处理

```
def Global_Thresholding(img, T):
    '''
    :description: 基本全局阈值处理

    :param img: 待处理图像
    :param T: 初始阈值（通常取为原图的平均灰度值，这里默认原图铺满所有像素）

    :return : 阈值处理后的图像
    '''
    h, w = img.shape[0], img.shape[1]
    img1 = np.zeros((h,w),np.uint8)
    T0 = T
    T1 = threshold(img,T0)

    for k in range (100):   # 迭代次数为经验值，可据实际情况选定
        if abs(T1-T0) == 0:   # 若新阈值减旧阈值差值为零，则为二值图最佳阈值
            for i in range (h):
                for j in range (w):
                    if img[i,j] > T1:
                        img1[i,j] = 255
                    else:
                        img1[i,j] = 0
            break
        else:
            T2 = threshold(img , T1)
            T0 = T1
            T1 = T2   # 变量转换，保证if条件为新阈值减旧阈值

    return img1
```

算法效果

原图

原图对应的直方图



全局阈值处理后图图像

算法评价

当与物体和背景相关的直方图模式间存在一个相当清晰的波谷时，这个简单的算法工作的非常好。

在速度是一个重要因素的情况下，参数ΔT用于控制迭代的次数。通常，ΔT越大，算法的迭代次数越少。

所选的初始阈值必须大于图像中最小灰度级而小于最大灰度级（图像的平均灰度对于T来说是较好的初始选择）

Otsu最佳全局阈值处理实战

算法思想

OTSU用来自动对基于聚类的图像进行二值化，或者说，将一个灰度图像退化为二值图像。

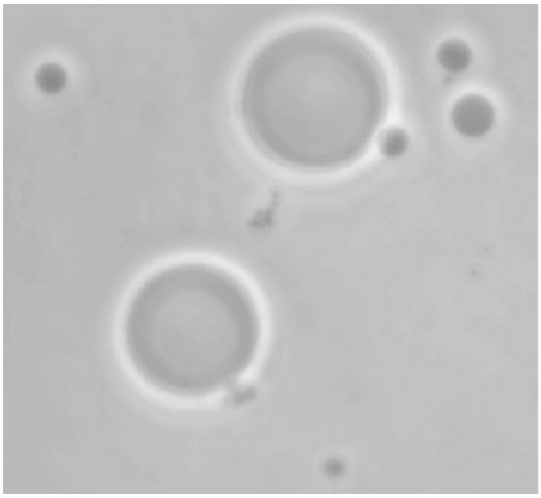
该算法假定该图像根据双模直方图（前景像素和背景像素）把包含两类像素，于是它要计算能将两类分开的最佳阈值，使得它们的类内方差最小；由于两两平方距离恒定，所以即它们的类间方差最大。

算法实现

```
def Otsu_Thresholding(image):
    '''
    :description: Otsu最佳全局阈值

    :param image: 原图

    :return gray: 原图的灰度图像
    :return thresh: 简单阈值处理后的图像
    :return th1: 对灰度图像进行Otsu最佳全局阈值处理后对图像
    :return blurred: 高斯模糊后的图像
    :return th2:对高斯滤波后对图像进行Otsu最佳全局阈值处理后的图像
    '''
    gray = cv2.cvtColor(image, cv2.COLOR_BGRA2GRAY)      # rgb图像变换为灰度图像

    (T, thresh) = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)  # 简单阈值处
理

    (T, th1) = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)  # 对灰度图像进行Otsu最佳全局阈值处理

    blurred = cv2.GaussianBlur(gray, (5, 5), 0)      # 高斯滤波降噪

    (T, th2) = cv2.threshold(blurred, 127, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)  # 对高斯滤波后对图像进行Otsu最佳全局阈值处理

    return [gray, thresh, th1, blurred, th2]
```

算法效果

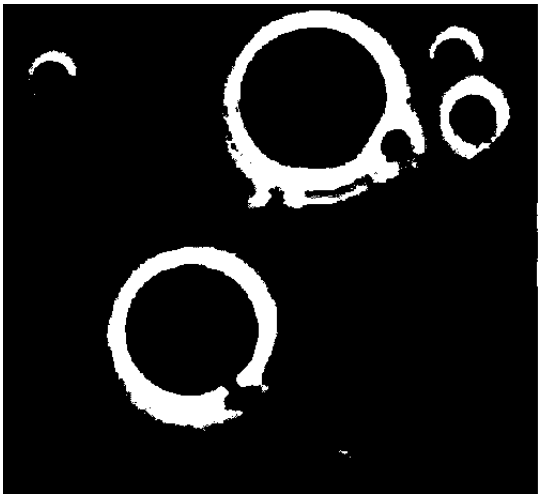
原图



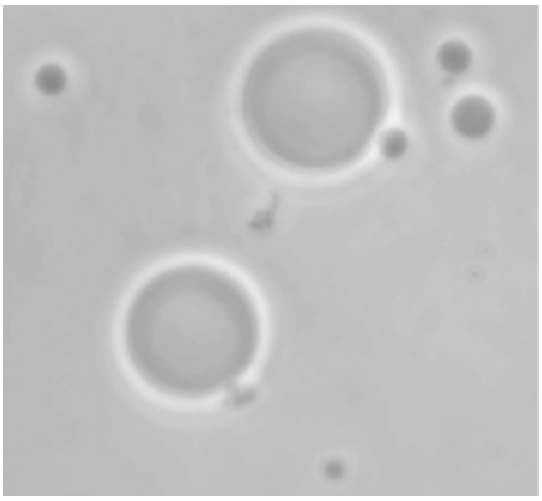
简单阈值处理后的图像

由于原始图像像素普遍较亮，所以阈值后的图像全为白像素

对灰度图进行Otsu最佳全局阈值处理



高斯模糊



对高斯滤波后对图像进行Otsu最佳全局阈值处理