轮廓检测、过滤和绘图

轮廓可以定义为定义图像中物体边界的一系列点。轮廓可表达图像物体边界的关键信息，编码了物体形状的主要信息。这些信息作为图像描述，例如SIFT、傅立叶描述符或形状上下文等的基础，可用于形状分析和对象检测和识别。以下主题中讨论与轮廓相关的关键点

以下主题中讨论与轮廓相关的关键点

- 压缩轮廓
- 图像矩
- 与轮廓相关的函数
- 轮廓滤波
- 轮廓识别
- 轮廓匹配

轮廓介绍

轮廓可以看作一条将所有点沿着边界连接起来的曲线。当它们能够表达形状边界时，对这些点的分析可以揭示形状分析，及对象检测与识别的关键信息。OpenCV提供了许多函数检测和处理轮廓。在深入研究这些函数之前，先看一个轮廓例子的结构。例如，下面的函数模拟检测假想图像中的轮廓

```
def get_one_contour():
    """Returns a 'fixed' contour"""

    cnts = [np.array(
        [[[600, 320]], [[563, 460]], [[460, 562]], [[320, 600]], [[180, 563]], [[78, 460]], [[40,
         [[179, 78]], [[319, 40]], [[459, 77]], [[562, 179]]], dtype=np.int32)]
    return cnts
```

轮廓是由np. int32类型，大小范围为[-2147483648, 2147483647]整数点组成的数组。调用这个函数取得轮廓数组。在本例中，该数组只检测到一个轮廓

```
# Get a sample contours:
contours = get_one_contour()
print("contour shape: '{}'".format(contours[0].shape))
print("'detected' contours: '{}' ".format(len(contours)))
```

可用OpenCV提供的函数来处理轮廓。定义get_one_contour()函数很有趣，它以一种简单的方法使用轮廓，可供调试和测试轮廓相关功能。一般情况下，真实图像中检测到的轮廓有数百个点，调试代码较困难。因此，请将此函数放在手边。

OpenCV提供了cv2.drawContours()，图像中绘制轮廓。可调用这个函数看看轮廓是什么样的。编写了draw_contour_points()函数，绘制图像中的轮廓点。此外，用np.squeeze()函数来消除一维数组，比如使用[1,2,3]代替[1,2,3]。例如，打印上一个函数中定义的轮廓，将得到以下结果

```
[600 320
563 460
460 562
320 600
180 563
78 460
40 320
77 180
179 78
319 40
459 77
562 179]
```

执行以下代码行之后

```
squeeze = np.squeeze(cnt)
```

打印squeeze，输出

```
600 320] [563 460] [460 562] [320 600] [180 563] [ 78 460] [ 40 320] [ 77 180] [179 78]
[319 40] [459 77] [562 179
```

此时，我们可以遍历这个数组的所有点。

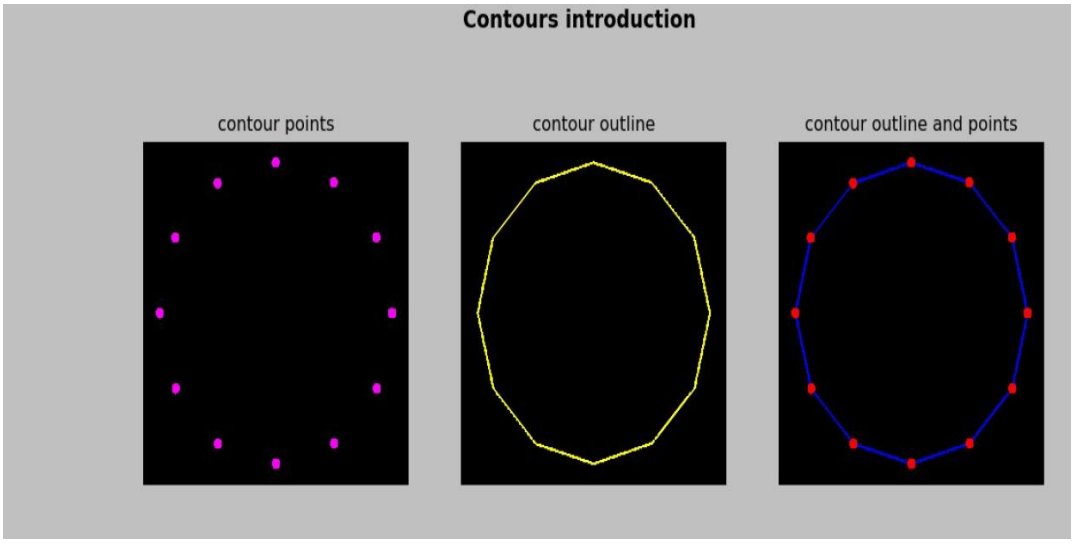
因此，draw_contour_points()函数的代码如下

```
def draw_contour_points(img, cnts, color):
"""Draw all points from a list of contours"""
   for cnt in cnts:
     squeeze = np.squeeze(cnt)
     for p in squeeze:
       p = array_to_tuple(p)
       cv2.circle(img, p, 10, color, -1)
   return img
```

在上一个函数中，用了array_to_tuple()函数，将数组转换为tuple

```
def  array_to_tuple(arr):
    """Converts array to tuple"""
    return tuple(arr.reshape(1, -1)[0])
```

轮廓的第一个点[600 320]被转换成(600,320)，可以在cv2.circle()中使用。轮廓完整代码见contours_introduction.py。屏幕截图中可以看到这个脚本的输出



为了完成对轮廓的介绍，编写脚本contours_introduction_2.py。编写了函数build_sample_image()和build_sample_image_2()。这些函数在图像中绘制基本形状，提供一些可预测(或预定义)的形状。

这两个函数的目的与前面脚本中定义的get_one_contour()函数相同，即帮助我们理解与轮廓相关的关键概念。build_sample_image()函数的代码如下

```
def build_sample_image():
    """Builds a sample image with basic shapes"""

    # Create a 500x500 gray image (70 intensity) with a rectangle and a circle inside:
    img = np.ones((500, 500, 3), dtype="uint8") * 70
    cv2.rectangle(img, (100, 100), (300, 300), (255, 0, 255), -1)
    cv2.circle(img, (400, 400), 100, (255, 255, 0), -1)

    return img
```

可以看到，函数绘制了两个填充的形状(一个矩形和一个圆形)。该函数创建具有两个外部轮廓的图像。

build_sample_image_2()函数的代码如下

```
def build_sample_image_2():
    """Builds a sample image with basic shapes"""

    # Create a 500x500 gray image (70 intensity) with a rectangle and a circle inside (with inte
    img = np.ones((500, 500, 3), dtype="uint8") * 70
    cv2.rectangle(img, (100, 100), (300, 300), (255, 0, 255), -1)
    cv2.rectangle(img, (150, 150), (250, 250), (70, 70, 70), -1)
    cv2.circle(img, (400, 400), 100, (255, 255, 0), -1)
    cv2.circle(img, (400, 400), 50, (70, 70, 70), -1)

    return img
```

此函数绘制两个填充矩形(一个在另一个矩形中)和两个填充圆(一个在另一个圆中)。该函数创建具有两个外部和两个内部轮廓的图像。

在contours_introduction_2.py，在加载图像后，我们将其转换为灰度值并进行阈值处理，得到二值图像。这个二进制图像稍后将使用cv2.findContours()函数来查找轮郭。创建的图像只有圆形和方形。因此，调用cv2.findContours()将找到所有这些创建的轮廓。cv2.findcontours()方法的签名如下

```
cv2.findContours(image, mode, method[, contours[, hierarchy[, offset]]]) -> image, contours, hier
```

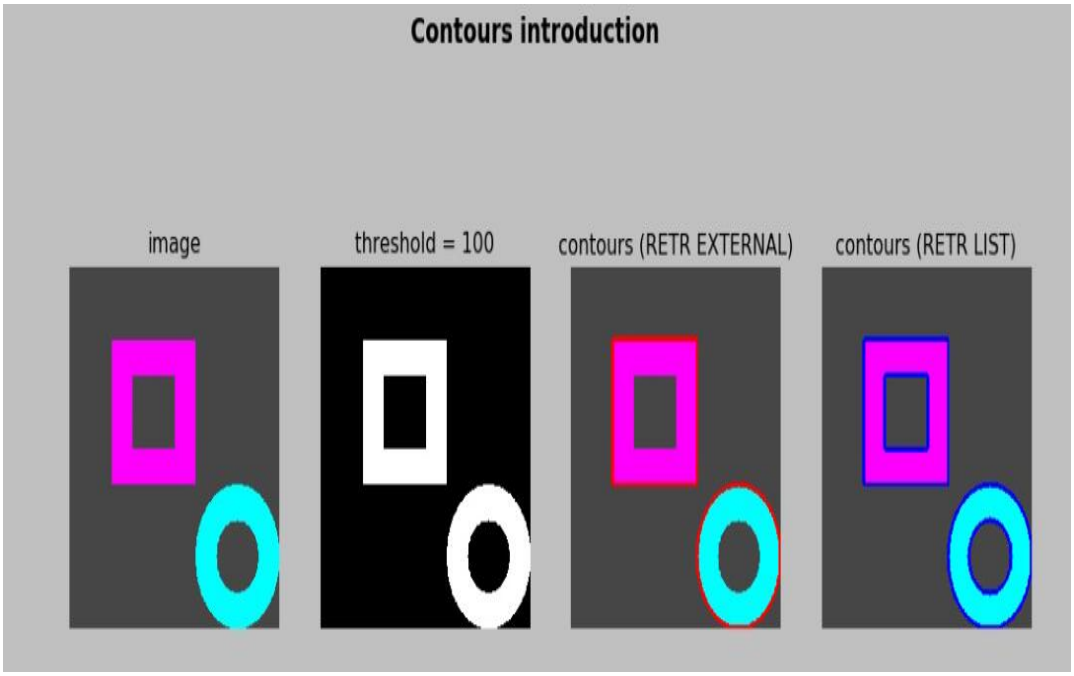
OpenCV提供了cv2.findcontour()，可用于检测二进制图像中的轮廓,例如阈值操作后的图像。该函数实现了基于边界跟踪的数字化二值图像拓扑结构分析算法。需要注意的是，在OpenCV 3.2版本之前，源图像被修改，在OpenCV 3.2之后，调用函数后，源图像不再被修改。源图像视为二值图像，非零像素值为1。此函数返回检测到的轮郭线，包含定义边界的所有检索到的点

检索到的轮廓可以以不同的模式输出，cv2.RETR_EXTERNAL，只输出外部轮廓。cv2.RETR_LIST，输出没有任何层次关系的轮廓，cv2.RETR_TREE，通过建立层次关系输出所有的轮廓，输出向量hierarchy包含关于层次关系的信息，为每个检测到的轮廓提供入口。对于第i个轮廓contours[i]，当j在[0,3]区间时，hierarchy[i][j]包含如下内容

- hierarchy[i][0]:同一层次结构下的下一个轮廓的索引
- hierarchy[i][1]:同一层次结构上的前一个轮廓的索引
- hierarchy[i][2]:第一个子轮廓层次结构的索引
- hierarchy[i][3]:父轮廓的索引

hierarchy[i][j]的负值时，如果j=0则没有下一个，j=1时没有上一个，j=2时没有子轮廓，j=3时没有父轮廓。method参数设置了检索每个被检测轮廓点的近似方法。

contours_introduction_2.py 截幕



截屏中，外部(cv2.RETR_EXTERNAL)和内部(cv2.RETR_LIST)都是通过调用cv2. findcontour()来计算的。

压缩轮廓

检测到的轮廓可以通过压缩来减少点的数量。OpenCV提供了几种减少点数的方法。可通过参数方法来设置，也可设置cv2.CHAIN_APPROX_NONE禁用压缩，存放所有边界点，不执行压缩。

cv2.CHAIN_APPROX_SIMPLE方法可以用于压缩检测到的轮廓，压缩了轮廓的水平、垂直和对角线部分，只保留了端点。例如，如果我们用cv2.CHAIN_APPROX_SIMPLE压缩矩形的轮廓，可由四个点组成。

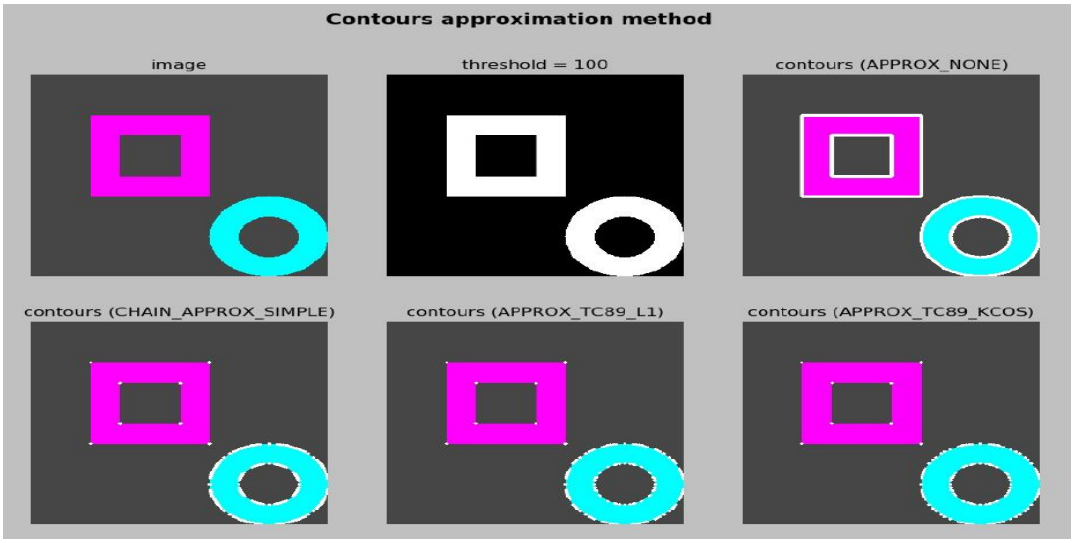
最后，OpenCV基于Teh-Chin算法(一种非参数化方法)为压缩轮廓提供了两个额外的标记。该算法的第一步是根据每个点的局部属性确定支持区域(ROS)。.

接着计算每个点的相对重要性，并通过非极大值压缩检测支配点。使用三种不同的显著性度量，对应于不同程度的离散曲率度量的准确性

K-cosine measure
K-curvature measure
One curvature measure (k = 1 of 2))

OpenCV提供与离散曲率度量相关的两个标志cv2.CHAIN_APPROX_TC89_L1,cv2.CHAIN_APPROX_TC89_KCOS。可查看“On the Detection of Dominant Points on Digital Curves”(1989)。

在contours_approximation_method.py。前面提到的四个标志
(cv2.CHAIN_APPROX_NONE,cv2.CHAIN_APPROX_SIMPLE,cv2,CHAIN_APPROX_TC89_L1和
cv2.CHAIN_APPROX_TC89_KCOS)对图像中检测到的两个轮廓进行编码。在屏幕截图中可以看到这个脚本的输出



可以看到，定义轮廓的点用白色表示，显示了四种方法(cv2CHAIN_APPROX_NONE,cv2.CHAIN_APPROX_SIMPLE cv2.CHAIN_APPROX_TC89_L1和cv2.CHAIN_APPROX_TC89_KCOS)压缩所提供的两个形状的检测轮廓。

图像矩

在数学中，力矩可以看作是函数形状的特定量度。图像矩可看作图像像素强度的加权平均值。这些矩函数，编码了一些有趣的特性。图像矩对于描述检测到的轮廓的属性是有用的，例如，物体的质心，或物体的面积，等等。

cv2.moments()可用于计算矢量形状或栅格化形状的前三阶的所有图像矩。

此方法的签名如下

```
retval = cv.moments(array[, binaryImage])
```

为了计算检测到的轮廓的矩，例如，第一个检测到的轮廓，执行以下步骤

```
M=cv2.moments(contours[0])
```

打印M，得到：

{'m00': 235283.0, 'm10': 75282991.16666666, 'm01': 75279680.83333333, 'm20': 28496148988.333332, 'm11'

可以看到，有三种不同类型的图像矩(m_{ji}, mu_{ji}, nu_{ji})。

空间矩m_ji计算如下

$$m_{ji} = sum_{x,y} = (array(x,y) \cdot x^j \cdot y^i)$$

中心矩mu_{ji}的计算方法如下

$$mu_{ji} = \sum_{x,y}(array(x,y \cdot (x-\bar{x})^j) \cdot (y-\bar{y})^i)$$

$$\bar{x} = \frac{m_{10}}{m_{00}}$$

前面的方程对应于质心。

- 根据定义，中心矩对于平移是不变的。因此，中心矩适合于描述物体的形式。空间矩和中心矩的缺点是都依赖于对象的大小，不是尺度不变的。

归一化中心矩nu_{ij}的计算方法如下

$$nu_{ji} = \frac{mu_{ji}}{m_{00}^{(i+j)/(2+1)}}$$

- 根据定义，归一化中心矩是平移和尺度的不变量.

下一图像矩的值计算如下：

$$mu_{00} = m_{00}, nu_{00} = 1, nu_{10} = mu_{10} = mu_{01} = mu_{10} = 0$$

因此，这些图像矩不会被储存起来

矩通常按其顺序进行分类，其计算是基于矩m_ji的指标j和i的和(j+i)

基于矩的物体特征

如前所述，矩是由轮廓计算出的特征，允许对物体进行几何重建。虽然没有直接易懂的几何意义，但基于矩可以计算出一些几何性质和参数。

contours_analysis.py 首先计算检测到的轮廓的矩，然后计算一些目标特征

```
M=cv2.moments(contours[0])
print("Contour  area:  '{}'".format(cv2.contourArea(contours[0])))
print("Contour  area:  '{}'".format(M['m00']))
```

矩m_{00}给出了轮廓的面积，相当于函数cv2.contourArea()。计算轮廓的质心，执行以下操作

```
print("center X : '{}'".format(round(M['m10'] / M['m00'])))
print("center Y : '{}'".format(round(M['m01'] / M['m00'])))
```

圆度 k 是轮廓接近圆轮廓程度的度量。以下公式可以计算出轮廓的圆度

$$k = \frac{P^2}{A \cdot 4 \cdot \pi}$$

P是轮廓的周长，A是相应的面积。如果是正圆，结果是1。得到的值越高，圆的程度越少。

这可以通过rounness()函数计算

```
def roundness(contour, moments):
    """Calculates the roundness of a contour"""
    length = cv2.arcLength(contour, True)
    k = (length * length) / (moments['m00'] * 4 * np.pi) return k
```

偏心率是测量一个轮廓能被拉长多少的一种方法。偏心率 ε 可由长半轴a和短半轴b计算，公式

$$\epsilon = \sqrt{\frac{a^2 - b^2}{b^2}}$$

计算轮廓偏心的一种方法是先计算出与轮廓相吻合的椭圆，然后由计算出的椭圆a和b，根据前面的公式计算 ε 。代码如下

```
def eccentricity_from_ellipse(contour):
    """Calculates the eccentricity fitting an ellipse from a contour"""
    (x, y), (MA, ma), angle = cv2.fitEllipse(contour)
    a=ma/2
    b=MA/2
    ecc = np.sqrt(a ** 2 - b ** 2) / a
    return ecc
```

另一种方法是通过以下公式利用轮廓矩计算偏心距

$$\epsilon = \sqrt{1 - \frac{\frac{mu_{20}+mu_{02}}{2} - \sqrt{\frac{(mu_{20}+mu_{02})^2}{2}}}{\frac{mu_{20}+mu_{02}}{2} + \sqrt{\frac{(mu_{20}+mu_{02})^2}{2}}}}$$

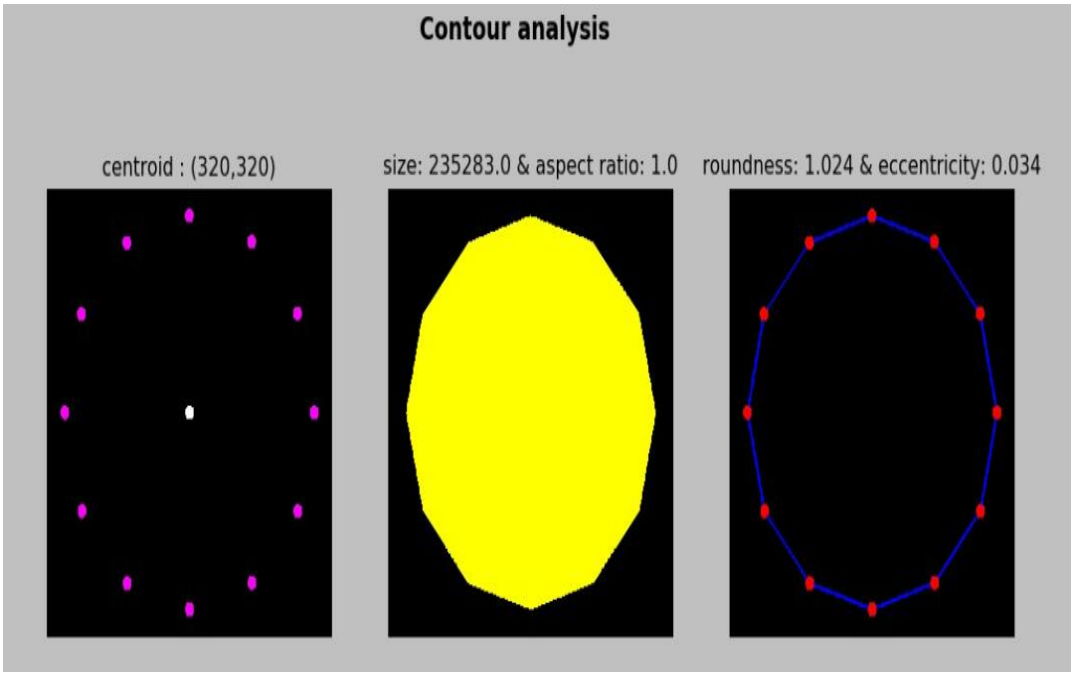
可用eccentricity_from_moments()来执行

```
def  eccentricity_from_moments(moments):
    """Calculates the eccentricity from the moments of the contour"""
    a1 = (moments['mu20'] + moments['mu02']) / 2
    a2 = np.sqrt(4 * moments['mu11'] ** 2 + (moments['mu20'] - moments['mu02']) ** 2) / 2 ecc = r
    return  ecc
```

为取得更多描述轮廓的特征，可以计算附加属性。例如，用cv2.boundingRect()计算的最小边界矩形的尺寸，很容易计算长宽比。长宽比是轮廓的边界矩形的宽高比

```
def  aspect_ratio(contour):
    """Returns the aspect ratio of the contour based on the dimensions of the bounding rect"""
    x, y, w, h = cv2.boundingRect(contour) res = float(w) / h
    return res
```

这些属性contours_analysis.py脚本中计算的。屏幕截图中可看脚本的输出



在前面的截图中，通过打印脚本中计算的所有属性来显示轮廓分析

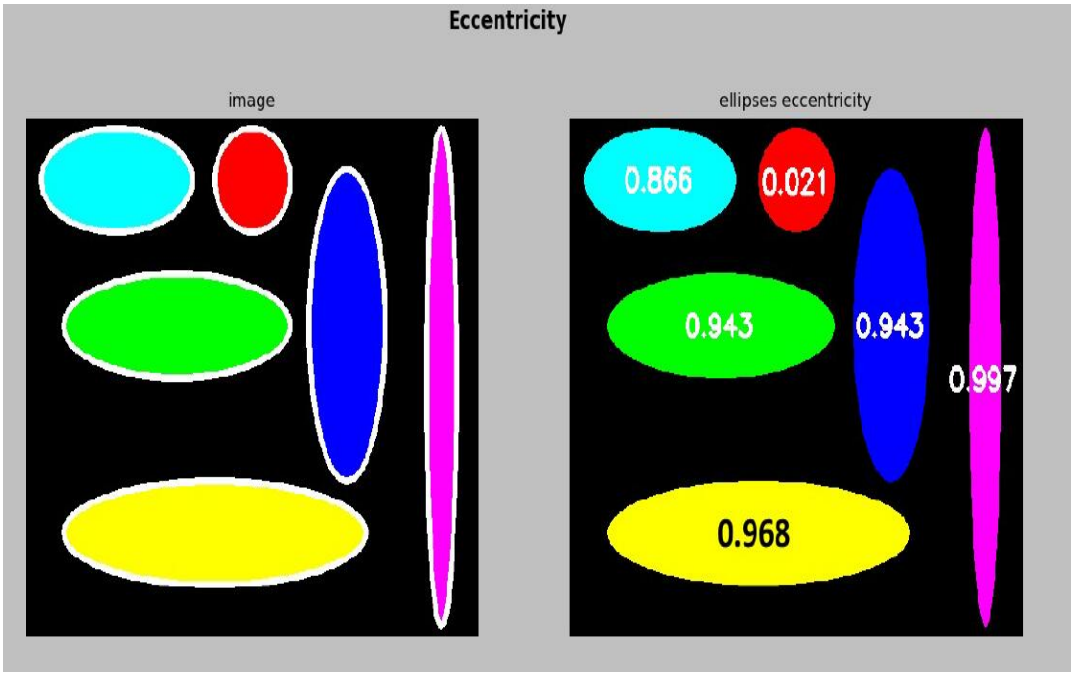
- 在前面的例子中，只使用二阶矩来计算简单的对象特征。为了更精确地描述复杂对象，应该使用高阶矩或更复杂的矩(例如Zernike、勒让德)。从这个意义上说，对象越复杂，为了最小化从矩重构对象的误差，矩的阶数应该越高。参见"Simple Image Analysis"。

本节内容见contours_ellipses.py。在这个脚本中，首先构建一个要使用的图像，绘制了不同的椭圆，由build_image_ellipses()执行的。用OpenCV函数cv2.ellipse()绘制了六个椭圆。然后在阈值图像中检测所绘制椭圆的轮廓，并计算其特征。计算圆度和偏心度。在得到的图像中，只显示出偏心。

在下一个截幕中，可以看到这个脚本的输出。偏心值画在每个轮廓的形心中间。这个功能是通过函数get_position_to_draw()来执行的

```
def get_position_to_draw(text, point, font_face, font_scale, thickness):
    """Gives the coordinates to draw centered"""
    text_size = cv2.getTextSize(text, font_face, font_scale, thickness)[0] text_x = point
    text_y = point[1] + text_size[1] / 2 return round(text_x), round(text_y)
```

这个函数返回的x, y坐标绘制文本居中位置点与文本的具体特征画是必要的计算文本字体大小设置的参数font_face,字体规模是font_scale设定的参数和厚度参数设定的厚度。截图中可以看到这个脚本的输出



可以看到，使用前面提到的函数eccentricity_from_moments计算出的偏心是这样显示的。使用所提供的两个公式计算了偏心距，得到了非常相似的结果。

Hu矩不变量

Hu不变矩主要是利用归一化中心矩构造了7个不变特征矩，由二阶矩和三阶矩可以导出7个不变矩。Hu矩不变量在平移、缩放和旋转方面是不变的，所有的矩(除了第七个)在反射方面是不变的。第七个反射改变符号，能够区分镜像。OpenCV提供了cv2.HuMoments()来计算7个Hu矩不变量。此方法的签名如下

cv2.HuMoments(m[, hu]) → hu

这里，m对应于用cv2.moments()计算的矩。输出hu对应于七个hu不变矩。

Hu矩不变量定义如下

$$
\begin{aligned}
hu[0] &= \eta_{20} + \eta_{02} \\
hu[1] &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
hu[2] &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
hu[3] &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
hu[4] &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - 3\eta_{03})(\eta_{21} + \eta_{03})[3(\eta \\
hu[5] &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
hu[6] &= (\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})^2 - (\eta_{21} + \eta_{13})^2] - (\eta_{30} - 3\eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta
\end{aligned}
$$

η_{ji} 代表 nu_{ji}

在contours_hu_moments.py脚本中，计算了七个Hu图像矩不变量。先用cv2.moments()计算图像矩。计算矩，参数可以是矢量形状，也可以是图像。如果binaryImage参数为true(仅用于图像)，则输入图像中的

所有非零像素都将被视为1。脚本中用矢量形状和图像计算矩。通过计算得到的矩，计算Hu矩不变量。

下面解释关键代码。我们首先加载图像，将其转换为灰度，然后应用cv2.threshold()得到二值图像

```
# Load the image and convert it to grayscale: image = cv2.imread("shape_features.png")
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply cv2.threshold() to get a binary image
ret, thresh = cv2.threshold(gray_image, 70, 255, cv2.THRESH_BINARY)
```

计算的图像矩使用阈值图像。然后计算质心，最后计算hu矩不变量

```
#  Compute  moments:
M  =  cv2.moments(thresh,  True) print("moments:        '{}'".format(M))
# Calculate the centroid of the contour based on moments:
x, y = centroid(M)
#  Compute  Hu  moments: HuM  =  cv2.HuMoments(M)
print("Hu moments: '{}'".format(HuM))
```

重复这个过程，在本例中，传递的是轮廓而不是二值图像。因此，先计算二值图像中轮廓的坐标

```
# Find contours
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

#  Compute  moments:
M2 = cv2.moments(contours[0])
print("moments:      '{}'".format(M2))

# Calculate the centroid of the contour based on moments:
x2,  y2  =  centroid(M2)

#  Compute  Hu  moments:
HuM2   =   cv2.HuMoments(M2)
print("Hu moments: '{}'".format(HuM2))
```

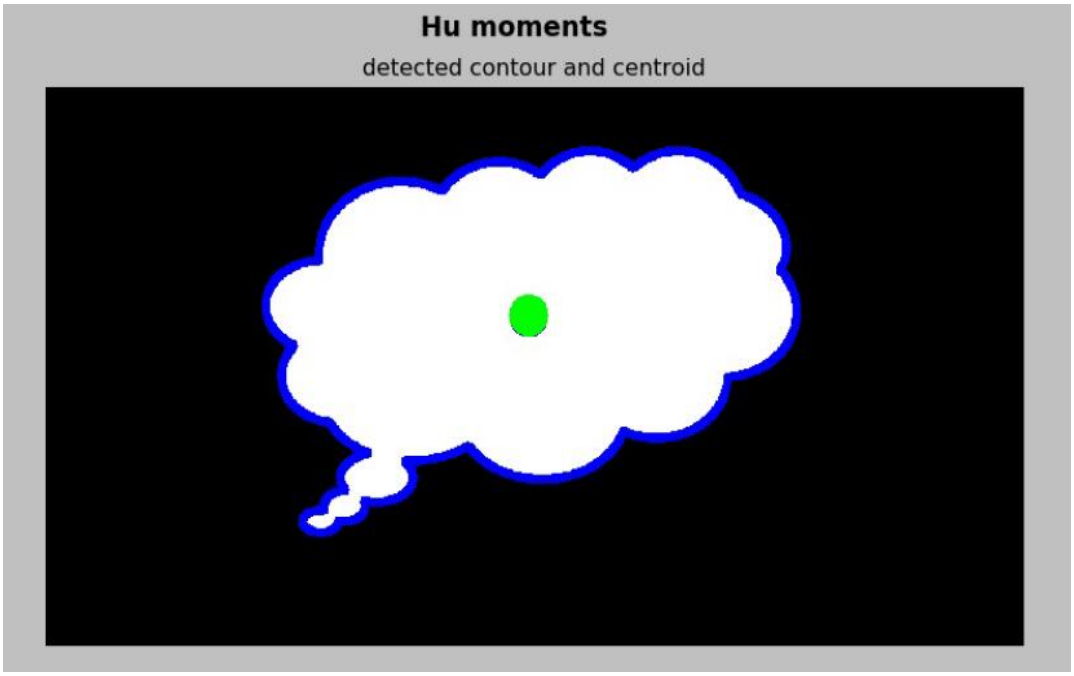
中心如下所示

```
print("('x','y'): ('{}','{}')".format(x, y))
print("('x2','y2'):  ('{}','{}')".format(x2,  y2))
```

可以看到，计算出的力矩，Hu力矩不变量，和质心非常相似，但不一样。例如，得到的质心如下

```
('x','y'): ('613','271')
('x2','y2'): ('613','270')
```

可以看到，y坐标相差一个像素。其原因是有限的光栅分辨率。轮廓的矩估计与同一栅格化轮廓的矩计算略有不同。截屏幕中可以看到此脚本的输出，其中显示了两个中心体，以突出显示y坐标中的这个小差异



在contours_hu_moments_properties.py 加载三个图像。第一个是原图像，第二个与原来的相对应，但旋转了180度。第三个点对应于原点的垂直反射。这可以在脚本的输出中看到。打印计算出的Hu矩不变量来自上述三幅图像。

此脚本的第一步是使用cv2.imread()加载图像，并利用cv2.cvtColor()将其转换为灰度。第二步是应用cv2.threshold()获得二值图像。用cv2.HuMoments()计算Hu矩

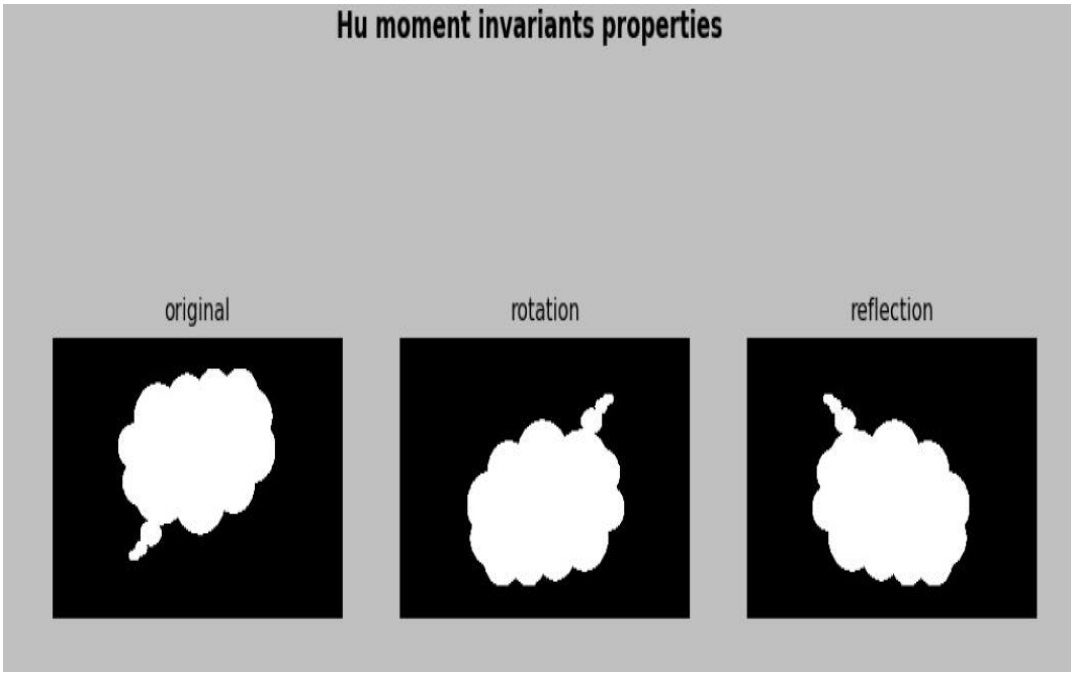
```
# Load the images (cv2.imread()) and convert them to grayscale (cv2.cvtColor()):
image_1 = cv2.imread("shape_features.png")
image_2 = cv2.imread("shape_features_rotation.png")
image_3 = cv2.imread("shape_features_reflection.png")
gray_image_1 = cv2.cvtColor(image_1, cv2.COLOR_BGR2GRAY)
gray_image_2 = cv2.cvtColor(image_2, cv2.COLOR_BGR2GRAY)
gray_image_3 = cv2.cvtColor(image_3, cv2.COLOR_BGR2GRAY)

# Apply cv2.threshold() to get a binary image:
ret_1, thresh_1 = cv2.threshold(gray_image_1, 70, 255, cv2.THRESH_BINARY)
ret_2, thresh_2 = cv2.threshold(gray_image_2, 70, 255, cv2.THRESH_BINARY)
ret_3, thresh_3 = cv2.threshold(gray_image_3, 70, 255, cv2.THRESH_BINARY)

# Compute Hu moments cv2.HuMoments():
HuM_1 = cv2.HuMoments(cv2.moments(thresh_1, True)).flatten()
HuM_2 = cv2.HuMoments(cv2.moments(thresh_2, True)).flatten()
HuM_3 = cv2.HuMoments(cv2.moments(thresh_3, True)).flatten()

# Show calculated Hu moments for the three images:
print("Hu moments (original): '{}'".format(HuM_1))
print("Hu moments (rotation): '{}'".format(HuM_2))
print("Hu moments (reflection): '{}'".format(HuM_3))

# Plot the images:
show_img_with_matplotlib(image_1, "original", 1)
show_img_with_matplotlib(image_2, "rotation", 2)
show_img_with_matplotlib(image_3, "reflection", 3)

# Show the Figure:
plt.show()
```

计算的HU矩不变量如下

```
Hu moments (original): '[ 1.92801772e-01 1.01173781e-02 5.70258405e-05 1.96536742e-06 2.46949980e
Hu moments (rotation): '[ 1.92801772e-01 1.01173781e-02 5.70258405e-05 1.96536742e-06 2.46949980e
Hu moments (reflection): '[ 1.92801772e-01 1.01173781e-02 5.70258405e-05 1.96536742e-06 2.4694998
```

可以看到，计算出的Hu矩不变量在这三种情况下是相同的，除了第七种情况。此差异在前面显示的输出中以粗体突出显示。符号改变了。

下面的截图显示了用于计算Hu矩不变量的三个图像



zernike矩

自Hu矩以来，图像广泛应用于图像处理和目标分类识别中。图像矩相关技术大为发展。

Zernike就是一个很好的例子。Teague提出了基于正交Zernike多项式基集的泽尼克矩。OpenCV不提供计算Zernike矩的函数。但是，其他Python包可以用于此目的。

mahotas包提供了zernike_moments()函数，该函数可用于计算Zernike矩。zernike_moments()的签名如下

mahotas.features.zernike_moments(im, radius, degree=8, cm={center_of_mass(im)})

此函数计算以cm为圆心的半径圆上的Zernike矩(如果没有使用cm，则计算图像的质心)。使用的最大度数是按度数设置的(默认为8)。

例如，如果使用默认值，可按如下方式计算Zernike矩

```
moments = mahotas.features.zernike_moments(image, 21)
```

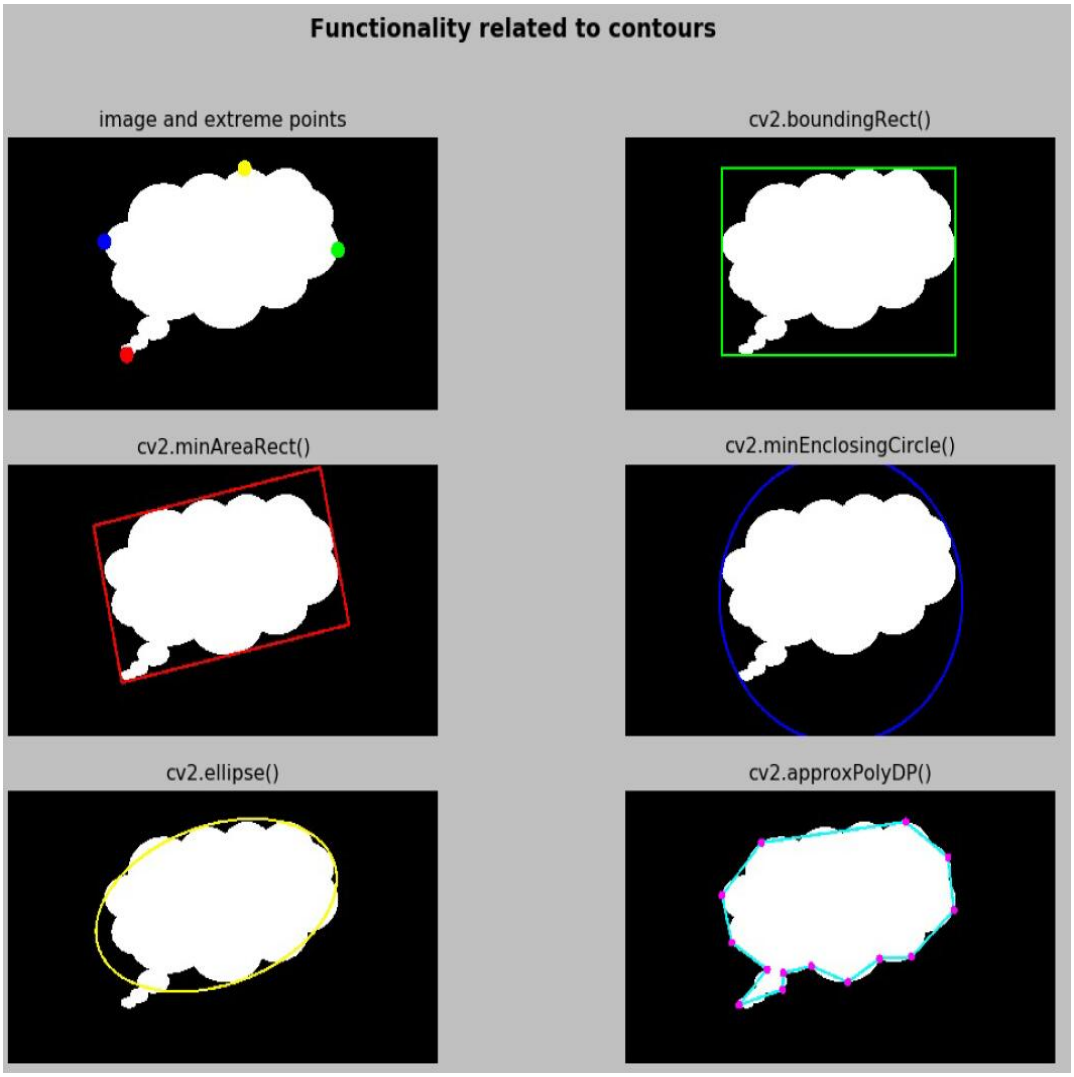
在本例中，使用半径21。Zernike矩特征向量是25维的。

与轮廓相关的功能

到目前为止，我们已经看到了一些由图像矩导出的轮廓属性,例如，质心、面积、圆度或偏心度等。OpenCV还提供了一些与轮廓相关的有趣功能，可以用来进一步描述轮廓。

主要使用与轮廓相关的五个OpenCV函数和一个计算给定轮廓的极值点的函数。

在描述这些函数计算的内容之前，先显示脚本的输出，因为生成的图像可以帮助我们理解前面提到的每个函数



cv2.boundingrect()返回包围所有轮廓点的最小边界矩形

```
x, y, w, h = cv2.boundingRect(contours[0])
```

cv2.minarearect()返回最小的旋转(如果必要的话)矩形，它包围了轮廓的所有点

```
rotated_rect    =    cv2.minAreaRect(contours[0])
```

为了提取旋转后的矩形的四个点，可以使用cv2.boxPoints()函数，它返回旋转后的矩形的四个顶点

```
box     =     cv2.boxPoints(rotated_rect)
```

cv2.minenclosingcircle()返回包围轮廓的所有点的最小圆(它返回中心和半径)

```
(x, y), radius = cv2.minEnclosingCircle(contours[0])
```

cv2.fitellipse()返回适合(具有最小最小平方误差)所有轮廓点的椭圆

```
ellipse    =    cv2.fitEllipse(contours[0])
```

cv2.approxpolydp()根据给定的精度，返回给定轮廓的轮廓近似值。这个函数使用Douglas-Peucker算法。

参数确定了精度，确定了原始曲线与其近似值之间的最大距离。因此，得到的轮廓是一个与给定轮廓相似、点更少的抽取轮廓

```
approx  =  cv2.approxPolyDP(contours[0],  epsilon,  True)
```

extreme_points()计算定义给定轮廓的四个极值点

```
def extreme_points(contour):
    """Returns extreme points of the contour"""
    index_min_x = contour[:, :, 0].argmin()
    index_min_y = contour[:, :, 1].argmin()
    index_max_x = contour[:, :, 0].argmax() index_max_y = contour[:, :, 1].argmax()

    extreme_left = tuple(contour[index_min_x][0])
    extreme_right= tuple(contour[index_max_x][0])
    extreme_top  = tuple(contour[index_min_y][0])
    extreme_bottom = tuple(contour[index_max_y][0])

    return extreme_left, extreme_right, extreme_top, extreme_bottom
```

np.argmin()返回沿轴方向的最小值的索引。如多次出现最小值，返回与第一次出现对应的索引。
np.argmax()返回最大值的索引。一旦计算了索引(例如index)，我们将得到数组的相应部分(例如，contour[index]-40 320)，并访问第一个(例如，contour[index][0]-[40 320])。最后将它转换为一个元组(例如，tuple(contour[index][0])(40,320))。

可以以更紧凑的方式执行这些计算

```
index_min_x = contour[:, :, 0].argmin()
extreme_left = tuple(contour[index_min_x][0])
```

This code can be rewritten as follows:
可以按如下方式重写此代码

```
extreme_left = tuple(contour[contour[:, :, 0].argmin()][0])
```

过滤轮廓

前面几节中，我们已经了解了如何计算检测到的轮廓的大小。轮廓大小可以根据图像矩或使用OpenCV函数cv2.contourArea()计算。在本例中将根据每个轮廓的计算大小对检测到的轮廓进行排序。

因此，sort_contours_size()函数是关键

```
def  sort_contours_size(cnts):
    """ Sort contours based on the size"""
    cnts_sizes = [cv2.contourArea(contour) for contour in cnts]
    (cnts_sizes,  cnts)  =  zip(*sorted(zip(cnts_sizes,  cnts)))
    return cnts_sizes,  cnts
```

在解释这个函数的代码之前，我们将介绍一些关键点。操作符*可以与zip()一起使用来解压缩列表

```
coordinate = ['x', 'y', 'z']
value = [5, 4, 3]
result  =  zip(coordinate,  value)
print(list(result))
c, v =  zip(*zip(coordinate, value))
print('c  =',  c)
print('v =', v)
```

输出如下

```
[('x', 5), ('y', 4), ('z', 3)] c = ('x', 'y', 'z')
v = (5, 4, 3)
```

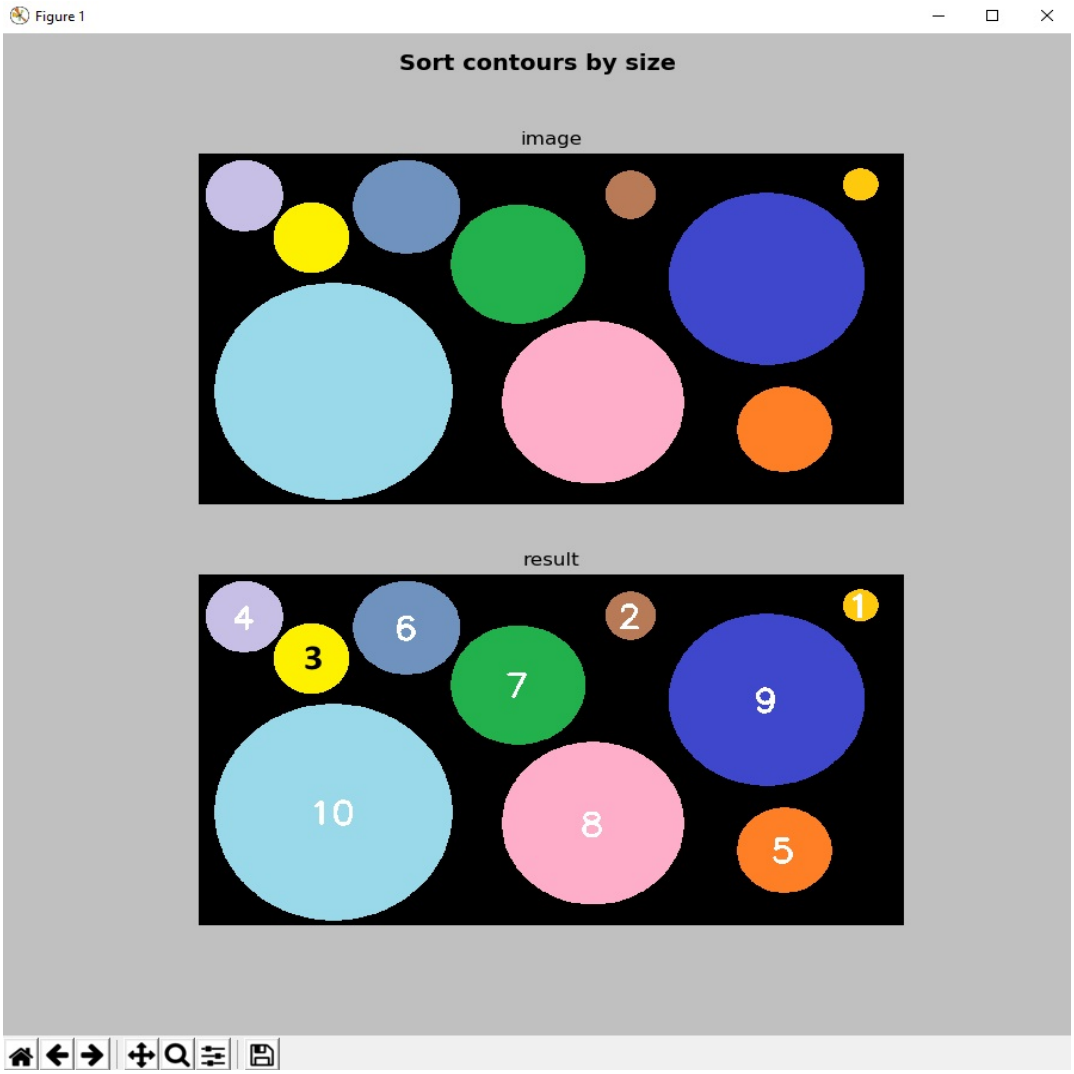
合并排序后的函数

```
coordinate = ['x', 'y', 'z'] value = [5, 4, 3]
print(sorted(zip(value, coordinate)))
c, v = zip(*sorted(zip(value, coordinate)))
print('c  =',  c)
print('v =', v)
```

输出如下

```
[(3, 'z'), (4, 'y'), (5, 'x')]
c = (3, 4, 5)
v = ('z', 'y', 'x')
```

sort_contours_size()函数根据大小对轮廓进行排序。脚本还输出轮廓中心的排序编号。在截屏中可以看到contours_sort_size.py的输出



在截图的上半部分显示的是原始的图像，在截图的下半部分，原始的图像修改了，每个轮廓中间都包含了排序号

识别轮廓

我们之前介绍过cv2.approxPolyDP()，它用Douglas-Peucker算法，以更少的点来近似一个轮廓。这个函数中的一个关键参数是设置逼近精度的。见contours_shape_recognition.py。我们基于检测到的顶点的数量(cv2.approxPolyDP()的输出)，用cv2.approxPolyDP()来识别轮廓(例如，三角形，正方形，长方形，五边形，六边形，等等)。为了抽取点的个数，给定一个轮廓，首先计算轮廓的周长。在周长的基础上，建立了epsilon参数。这样，抽取的轮廓是不变的比例。参数的计算方法如下

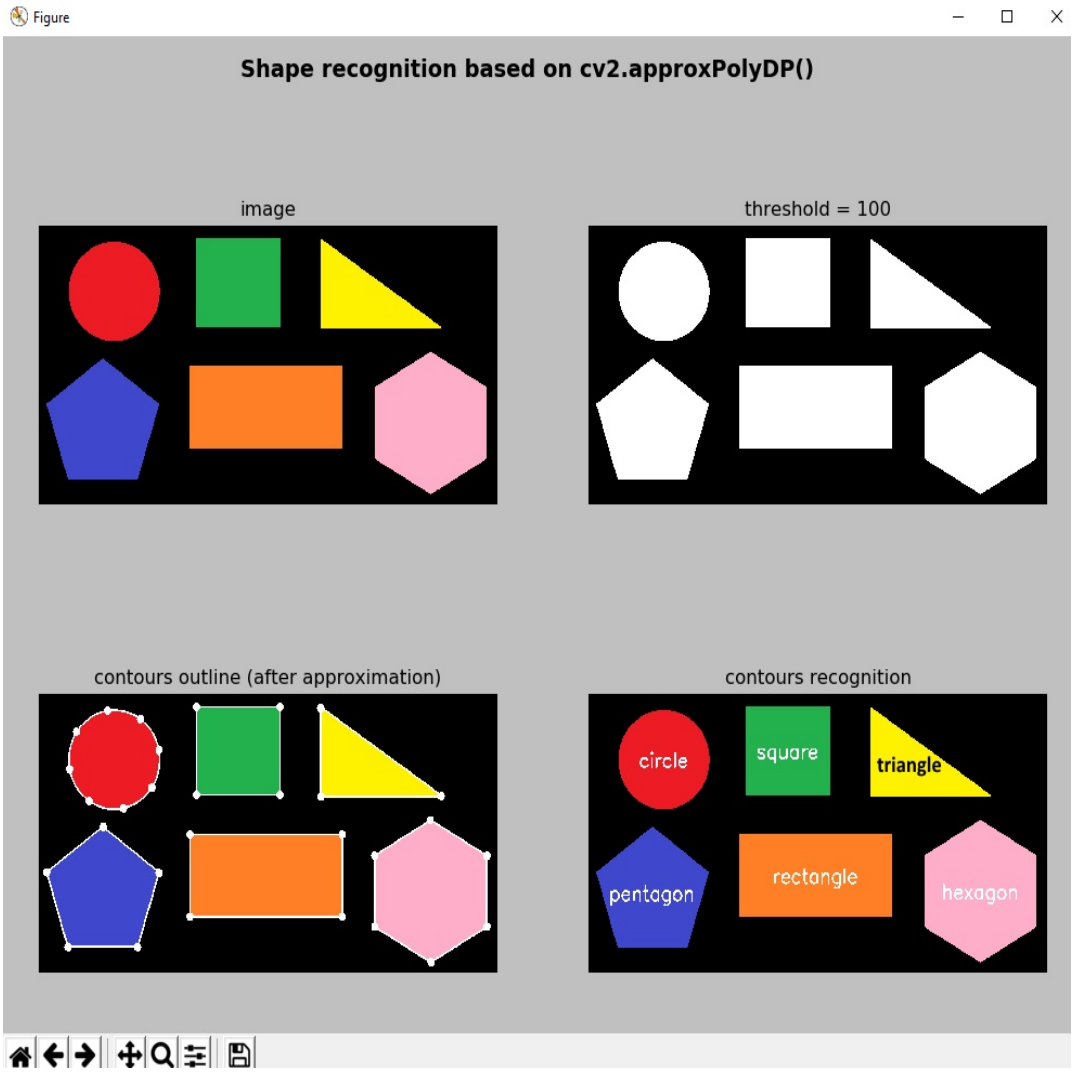
```
epsilon = 0.03 * perimeter
```

常数0.03是经过多次试验确定的。例如，如果这个常数更大(例如0.1)，那么epsilon参数也会更大，从而降低了近似精度

这样就得到了一个点较少的轮廓，并且得到了缺失的顶点。轮廓的识别是不正确的，因为它是基于检测到的顶点的数量。另一方面，如果这个常数更小(例如0.001)，那么epsilon参数也会更小，这样近似精

度就会提高，导致近似轮廓点更多。在这种情况下，由于得到了假的顶点，轮廓的识别也会被错误地执行。

截屏中可以看到contours_shape_recognition.py脚本的输出



在前面的截图中，显示了关键步骤(阈值、轮廓逼近和轮廓识别)。

匹配轮廓

Hu矩不变量既可用于目标匹配，又可用于目标识别。在这一节中，将看到如何匹配基于Hu矩不变量的轮廓。OpenCV提供了cv2.matchShapes()，可以使用三种比较方法来比较两个轮廓。所有这些方法都使用Hu矩不变量。这三个实现的方法是cv2.CONTOURS_MATCH_I1,cv2。CONTOURS_MATCH_I2,
cv2.CONTOURS_MATCH_I3。

如果A表示第一个对象，B表示第二个对象，则

$$m_i^A = sign(h_i^A) - log(h_i^A)$$
$$m_i^B = sign(h_i^B) - log(h_i^B)$$

h_i^A, h_i^B分别是A、B的Hu矩

cv2.CONTOURS_MATCH_I1:

$$I_1(A,B) = \sum_{i=1...7} |\frac{1}{m_i^A} - \frac{1}{m_i^B}|$$

cv2.CONTOURS_MATCH_I2:

$$I_2(A,B) = \sum_{i=1...7} |m_i^A - m_i^B|$$

cv2.CONTOURS_MATCH_I3:

$$I_3(A,B) = \sum_{i=1...7} \frac{|m_i^A - m_i^B|}{|m_i^A|}$$

在contours_matching.py,利用cv2.matchShapes()匹配几个轮廓的轮廓圆形度

首先，用OpenCV函数cv2.circle()在图像中绘制圆。这是参考图像。调用build_circle_image()生成图像。然后，加载图像match_shapes.png，绘制许多不同的形状。这两幅图像准备好了，下一步找到这两幅图像的轮廓

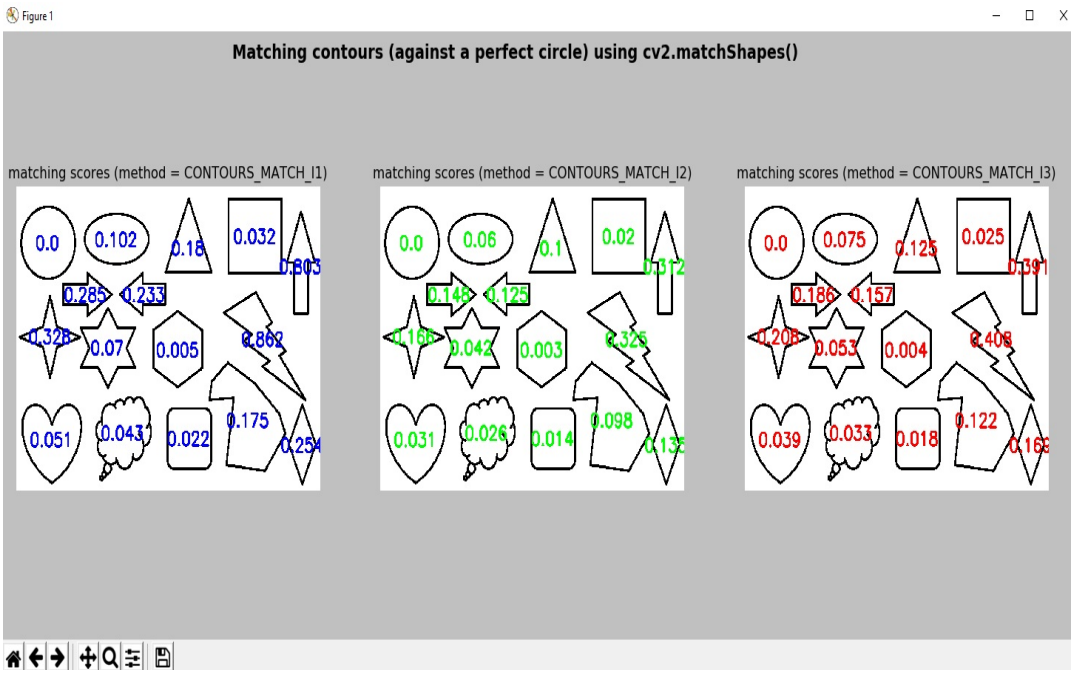
1.使用cv2.cvtColor()将它们转换为灰度
2.使用cv2.threshold()对它们进行二进制化
3.使用cv2.findContours()查找轮廓

At this point, we are ready to compare all the contours extracted from match_shapes.png against the contour extracted from the image built using the build_circle_image() function:

所有从 match_shapes.png 中提取的图像，与用build_circle_image()函数生成图像中提取的轮廓相比较。

```
for contour in contours:
# Compute the moment of contour:
M = cv2.moments(contour)
# The center or centroid can be calculated as follows:
cX = int(M['m10'] / M['m00'])
cY = int(M['m01'] / M['m00'])

# We match each contour against the circle contour using the three matching modes:
ret_1 = cv2.matchShapes(contours_circle[0], contour, cv2.CONTOURS_MATCH_I1, 0.0) ret_2 = (

# Get the positions to draw:
(x_1, y_1) = get_position_to_draw(str(round(ret_1, 3)), (cX, cY), cv2.FONT_HERSHEY_SIMPLEX, 1.2,
(x_2, y_2) = get_position_to_draw(str(round(ret_2, 3)), (cX, cY), cv2.FONT_HERSHEY_SIMPLEX, 1.2,
(x_3, y_3) = get_position_to_draw(str(round(ret_3, 3)), (cX, cY), cv2.FONT_HERSHEY_SIMPLEX, 1.2,

# Write the obtainted scores in the result images:
cv2.putText(result_1, str(round(ret_1, 3)), (x_1, y_1), cv2.FONT_HERSHEY_SIMPLEX, 1.2, (255, 0, (
cv2.putText(result_2, str(round(ret_2, 3)), (x_2, y_2), cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 255, (
cv2.putText(result_3, str(round(ret_3, 3)), (x_3, y_3), cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 0, 25
```

contours_matching.py 截屏



可以看出，result_1图像使用匹配模式cv2.CONTOURS_MATCH_I1，image result_2使用匹配模式cv2.CONTOURS_MATCH_I2，最后，result_3使用匹配模式cv2.CONTOURS_MATCH_I3显示匹配分数。

总结

本章回顾了OpenCV提供的与轮廓相关的主要函数，编写函数比较和描述轮廓。还提供了一些有趣的函数帮助调试代码。提供了创建简化轮廓和创建具有简单形状的图像的函数。第4章介绍与图像处理相关技术，第5章，图像处理技术，回顾了图像处理中的关键点，第6章，构造和构建直方图，介绍了直方图，

第7章，阈值技术，这一章为如何处理轮廓。下一章将介绍增强现实，可以定义为现实的改进，现实世界的视图增强与叠加计算机生成的元素。

问题

1. 如果要检测二值图像中的轮廓，应该使用什么函数?
2. OpenCV提供了哪四个标识来压缩轮廓?
3. OpenCV提供了什么函数来计算图像矩?
4. 什么矩提供了轮廓的大小?
5. OpenCV提供了什么函数来计算7个Hu矩不变量?
6. 如果想要得到给定轮廓的轮廓近似值，应该使用什么函数?
7. 在contour_function.py脚本中定义的extreme_points()函数用更紧凑的方式重写。
8. 如果想用Hu矩不变量作为特征来匹配轮廓，应用什么函数